VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

POČÍTAČOVÉ VIDĚNÍ JAKO WEBOVÁ SLUŽBA

COMPUTER VISION AS A SERVICE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ADAM JEŽ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ROMAN JURÁNEK, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20151/2016/xjezad00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Jež Adam, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Počítačové vidění jako webová služba**
Computer Vision as a Service
Kategorie: Web

Pokyny:

1. Prostudujte možnosti vytváření webových služeb
2. Vyberte vhodné algoritmy počítačového vidění
3. Navrhněte webovou službu, která podporuje autentizaci uživatelů, předávání dat a několik algoritmů počítačového vidění
4. Implementujte navrženou službu
5. Vytvořte prezentační materiály pro vaši aplikaci (web, poster, video)

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

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

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

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

Vedoucí: **Juránek Roman, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

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

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Cílem této diplomové práce je vytvořit webovou službu pro sdílení a snadný přístup k algoritmům počítačového vidění. V současné době existuje velké množství algoritmů a pro jejich autory je přínosné tyto algoritmy jednoduše sdílet s dalšími lidmi i z jiných vědních oborů. Hlavní částí práce je vytvoření architektury webové služby a navržení způsobu zpracování požadavků na spuštění algoritmů. Součástí implementace webové služby je vytvoření webového rozhraní, které umožní jednoduché spuštění algoritmů s vlastními daty, a klientská knihovna pro usnadnění integrace aplikačního rozhraní služby.

Abstract

The goal of this thesis is to create a web service for sharing and easy access to computer vision algorithms. Currently, there is a large number of algorithms and it's beneficial for their authors to simply share them with other people, even from other disciplines. The main part of the thesis consists of creating web service architecture and suggesting a method for request processing to run algorithms. Part of the implemented service is a web interface that allows use of algorithms with its own data, and client library that makes integration into other apps easier.

Klíčová slova

webová služba, webová aplikace, počítačové vidění, REST API, webové technologie, ASP.NET Core

Keywords

web service, web application, computer vision, REST API, web technologies, ASP.NET Core

Citace

JEŽ, Adam. *Počítačové vidění jako webová služba.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Juránek Roman.

Počítačové vidění jako webová služba

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Romana Juránka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Adam Jež
23. května 2017

Poděkování

Tímto bych rád poděkoval panu Ing. Romanu Juránkovi, Ph.D. za vedení a poskytnuté rady při vypracování diplomové práce. Dále bych rád poděkoval za poskytnuté kredity k službě Microsoft Azure, které sloužily pro testovací účely.

Obsah

Kapitola 1

Úvod

V současné době existuje velké množství algoritmů počítačového vidění, které jsou používány v mnoha oborech, a další algoritmy jsou vyvíjeny. Práce je zaměřena na algoritmy počítačového vidění, ale použít lze i jiné typy zejména multimediálních algoritmů. Pro autory těchto algoritmů je velmi přínosné sdílení algoritmů s dalšími lidmi, kteří je mohou otestovat, a ověřit jejich funkčnost na svých datech. Příkladem může být použití algoritmu pro detekci obličejů vývojářem ve své aplikaci, ve které zašle webový dotaz s vlastní fotografií a získá přesné pozice obličejů.

Cílem této práce je vytvoření webové služby, která zpřístupní vyvíjené algoritmy počítačového vidění širší skupině lidí, jak přes webové stránky pro jednoduché vyzkoušení a testovaní, tak přes webové aplikační rozhraní (zkráceně API) pro integraci do aplikací. V rámci zpřístupnění API bude vytvořena klientská knihovna k usnadnění použití. Vývojáři prostřednictvím ní budou mít možnost použít algoritmy počítačového vidění ve svých programech a využít tak komplexní algoritmy jednoduše ve svých aplikacích přes webové aplikační rozhraní.

V následujícím textu bude nejdříve v kapitole 2, Analýza požadavků pro webovou službu, přiblíženo, pro koho přesně webová služba bude a jaké bude její využití. Dále bude představené již existující řešení a následně budou zmíněny rozdíly se službou vyvíjenou v rámci této práce.

V kapitole 3, Návrh webové služby, bude dopodrobna diskutována architektura webové služby, budou zvoleny ukázkové algoritmy počítačového vidění a popsána jejich integrace do webové služby. V této kapitole proběhne také výběr webových technologií, zvolení architektury aplikačního rozhraní a návrh schématu databáze.

Po vytvoření návrhu bude v kapitole 4, Implementace, vysvětlena samotná implementace webové služby. Na začátku kapitoly bude představeno použití webového frameworku a jeho hlavní principy použité napříč frameworkem. Dále budou vysvětleny technologie a způsoby komunikace mezi servery webové služby. Navazuje sekce objasňující dedikovaný server, kde budou spouštěny algoritmy počítačového vidění. Dále budou představeny technologie usnadňující nasazení webové služby a výsledné použití. V kapitole následuje stručný popis použitých algoritmů počítačového vidění a výsledné webové služby. Následující část je věnovaná implementaci klientské knihovny pro přístup k aplikačnímu rozhraní služby. Ke konci kapitoly bude charakterizován způsob testování webové služby a diskutován směr dalšího vývoje služby.

V závěrečné kapitole 5 budou zhodnoceny dosažené výsledky této práce. V příloze se nachází podrobný popis zprovoznění webové služby B společně s popisem obsahu přiloženého média A.

Kapitola 2

Analýza požadavků pro webovou službu

Základním cílem této práce je navrhnout a implementovat webovou službu, která bude umožňovat snadné využití algoritmů počítačového vidění. Mezi uživatele služby patří ti, které chtějí otestovat funkčnost algoritmů, a také vývojáři, kteří chtějí algoritmy využít ve svých programech. Příkladem algoritmů počítačového vidění je detekce obličeje, rozpoznání znaků nebo identifikace vozidel. Vstupem těchto algoritmů jsou obrazová data nejčastěji ve formě obrázku nebo videa. Tato kapitola stručně definuje, jaké problémy služba řeší, uživatele služby a shrne požadavky na funkčnost webové služby.

2.1 Co má webová služba řešit?

Při vývoji algoritmů počítačového vidění je problémovou částí následné sdílení výsledného algoritmu mezi ostatní zájemce. Pakliže chce zájemce pouze otestovat algoritmus na svých datech, je nutné projít dlouhým procesem instalace/nastavení prostředí operačního systému pro spuštění algoritmu. Dalším scénářem využití je při vývoji různorodých služeb a aplikací, kde bychom chtěli využít jednoho konkrétního algoritmu počítačového vidění pro vylepšení služby. Příkladem může být automatické ořezání profilové fotografie uživatele tak, aby se uprostřed nacházel obličej.

První zmíněné využití webové služby řeší následující problémy, které nastávají při sdílení algoritmů:

- nejdříve je nutné provést sdílení zdrojových nebo binárních souborů,
- instalace bývá komplikovaná a často vyžaduje kompilaci programů ze zdrojových kódů,
- spuštění algoritmů většinou vyžaduje základní znalost práce s příkazovou řádkou.

Druhé zmíněné využití řeší problémy, kdy vlastní implementace algoritmu počítačového vidění není triviální a vyžaduje určité znalosti nebo implementace algoritmu je volně dostupná, ale její použití z jiných frameworků je složité, nebo někdy nemožné. Naopak volání API přes HTTP zvládají všechny známé programovací jazyky a frameworky.

2.2 Scénáře využití webové služby

Webová služba by měla problémy zmíněné v předcházející části řešit jednoduchým rozhraním pro spouštění algoritmů pomocí tří různých scénářů. Prvním je spuštění algoritmů z webového prohlížeče, které slouží pro vyzkoušení, případně demonstraci algoritmů. Druhým scénářem je spuštění algoritmů zasláním jednoduchého dotazu na aplikační server, což využijí vývojáři aplikací pro doplnění funkčnosti ve svých programech a pro testování algoritmu bez nutnosti webové prezentace, například pomocí programu curl [1]. Poslední scénář počítá s využitím knihovny, která zaobaluje vytváření dotazů na aplikační server. Tento scénář pouze vylepšuje a usnadňuje použití druhého scénáře.

2.3 Aktéři v systému

Analýzou uvedených požadavků byli rozeznáni následující aktéři:

1. uživatel přistupující z webového prohlížeče,
2. vývojář využívající API ve svém programu,
3. administrátor spravující algoritmy a uživatele.

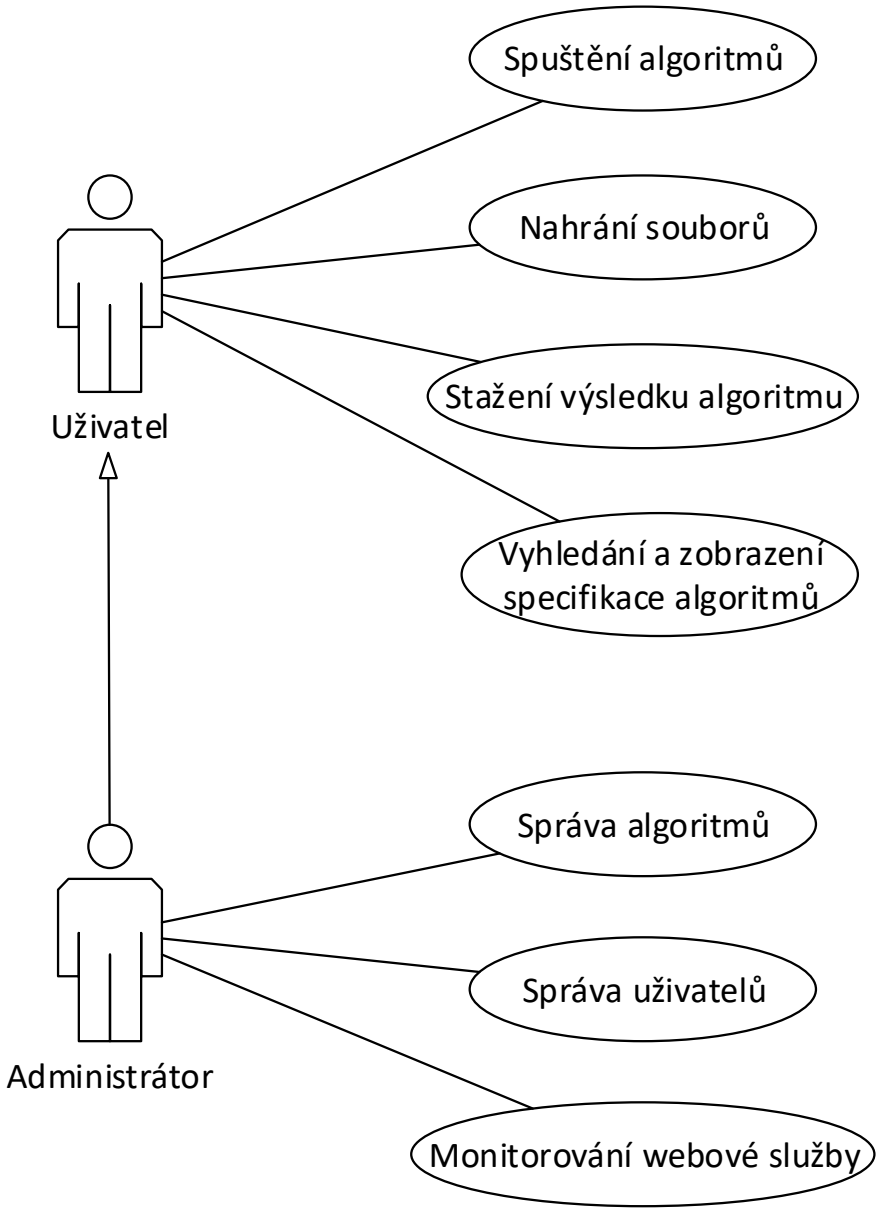
V mnoha případech jsou první a druhý zmíněný aktér stejným uživatelem, jenž využívá webové rozhraní pro vyzkoušení algoritmu, který následně využije ve svém programu. Dále v této práci nejsou tito aktéři nijak rozlišeni a jsou nazýváni jednoduše jako *uživatel webové služby*.

2.4 Případy užití

Z nadefinovaných požadavků a zjištěných aktérů v systému byl vytvořený diagram případů užití 2.1. Z důvodu přehlednosti byly do diagramu zahrnuty jen důležité akce. Některé akce byly sdruženy pod jeden obecný případ užití. Akce typu *Správa* v sobě zahrnuje operace vytvoření, čtení, aktualizace a mazání.

Hlavní případ užití je *spuštění algoritmů*, ale jelikož vstupem algoritmů bude většinou nějaký multimediální soubor, je důležité nejdříve tento soubor nahrát na servery webové služby, což zahrnuje případ *nahrání souborů*.

[1] https://curl.haxx.se/



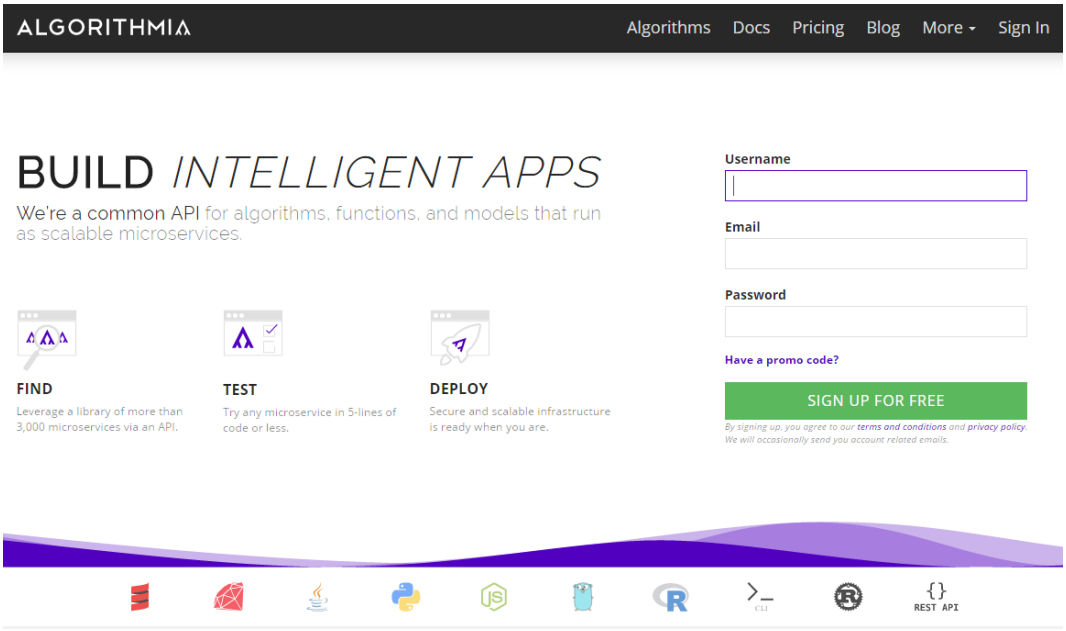
Obrázek 2.1: Diagram případů užití

2.5 Podobné služby

Nejvíce se popsané webové službě podobá služba s názvem **Algorithmia** [2]. Nabízí použití algoritmů počítačového vidění, analýzy textu, strojového učení a dalších. Algoritmy zde může vývojář přidávat a použití je zde do určitého počtu spuštění zadarmo. Vytvoření algoritmu je možné pouze v podporovaných programovacích jazycích. V současné době služba podporuje následující programovací jazyky: Java, Scala, Python, Ruby, Rust a JavaScript. Omezení je způsobeno nutností použití knihovny, která zajišťuje integraci se službou a která existuje pouze ve zmíněných programovacích jazycích.

[2] https://algorithmia.com

Používání služby již není omezené programovacím jazykem. Přistoupit ke službě lze jednoduše přes HTTP protokol nebo použitím dostupných knihoven, které jednotlivá HTTP volání zaobalují, a zjednodušují tím integraci do aplikace.



Obrázek 2.2: Úvodní stránka služby Algorithmia, jejíž hlavní výhodou je možnost přidání vlastních algoritmů.

Hlavním rozdílem a výhodou vytvářené webové služby by měla být nezávislost na programovacím jazyce, ve kterém je algoritmus napsaný. Tím bude omezena úroveň integrace algoritmu, například nebude možné reportovat průběh spuštěného algoritmu. Dále služba Algorithmia nenabízí možnost hostování webové služby na vlastním serveru. Tuto možnost by měla služba, vytvořená v rámci této práce, nabízet. Služba Algorithmia může sloužit jako inspirace například pro návrh aplikačního rozhraní a příklady využitých algoritmů.

Kapitola 3

Návrh webové služby

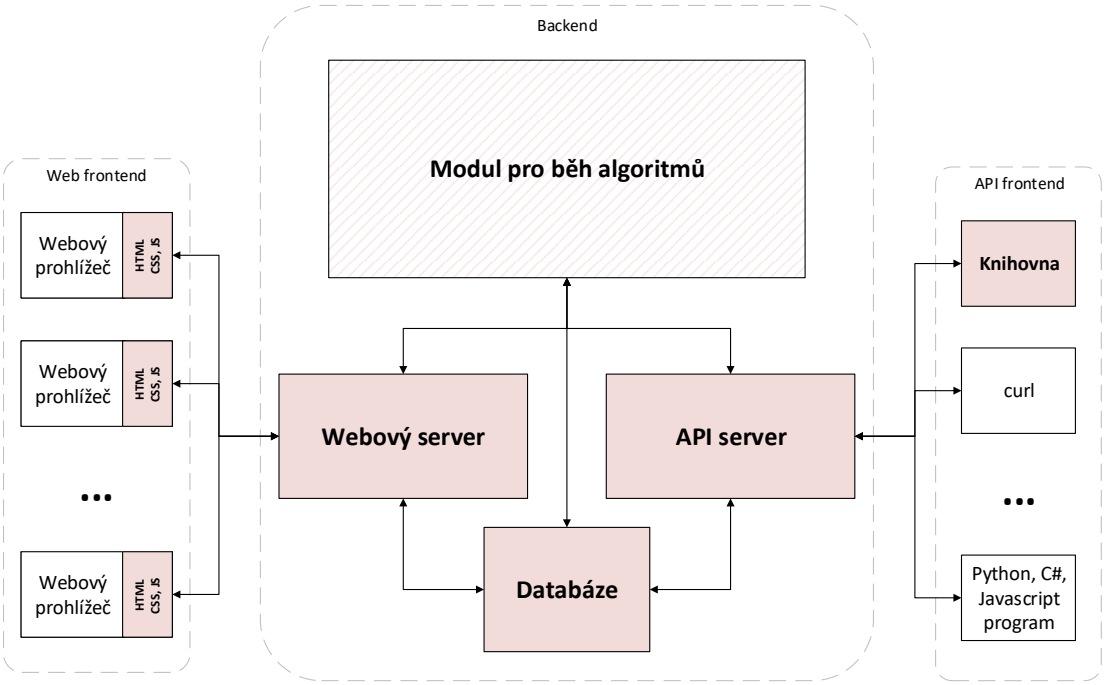
V této kapitole je popsána architektura webové služby a základní koncepty služby, které výrazně ovlivní následující implementaci. Nejdůležitějšími sekcemi jsou v úvodu kapitoly, konkrétně rozdělení webové služby viz 3.1, modul pro algoritmy počítačového vidění viz 3.2 a integrace algoritmů do webové služby viz 3.3. Dále se v této kapitole nachází výběr použitých technologií a algoritmů pro demonstraci použití a funkčnosti služby. Výběr webového frameworku je zdůvodněn v sekci 3.6. V závěru kapitoly je navrženo databázové schéma viz 3.8.

3.1 Logické rozdělení webové služby

Ve webové službě byli identifikovány logické části. Výsledná realizace se však může od té logické lišit. Nejobecnější rozdělení služby je na *backend*, *web frontend* a *API frontend* (viz obrázek 3.1). Backend je část služby, která běží na serverech a jehož implementace je hlavní součástí této práce. Web a API frontend slouží k prezentování a přístupu ke službě a liší se zejména ve formě odpovědi na dotaz.

Na obrázku 3.1 jsou barevným pozadím zvýrazněné části, které je nutné implementovat. Část backend obsahuje:

- **Modul pro běh algoritmů** – tato část obstarává běh algoritmů počítačového vidění a bude podrobněji popsána v následující části 3.2.
- **Webový server** – obstarává HTTP dotazy z webových prohlížečů a prezentuje výsledky přes webové stránky, lze se přes něj přihlásit k účtu a tento účet spravovat a spouštět algoritmy, které mají vytvořené prezentační stránky.
- **API server** – obstarává HTTP dotazy z aplikací, ale výsledky neprezentuje ve formě webové stránky, nýbrž ve formě jednoho z vybraného způsobu reprezentace dat (např. JSON, XML, YAML). Skrz API server lze spouštět veškeré dostupné algoritmy.
- **Databáze** – obsahuje data pro správu uživatelských účtů, správu algoritmů, správu uživateli nahraných souborů a záznamy s jednotlivými spuštěními algoritmů s výsledkem daného spuštění.
- **Knihovna** – zaobaluje zasílání jednotlivých HTTP dotazů do volání metod, analyzuje odpovědi z API serveru a zjednodušuje přístup k API v daném programovacím jazyce. Tato část je popsána až v další kapitole, přesněji 4.6.



Obrázek 3.1: Logické rozdělení webové služby na nezávislé systémy. Bloky s barevným pozadím představují systémy potřebné implementovat.

Poslední součástí implementace, která nebyla popsána, je prezentační vrstva na klientské straně, konkrétně ve webovém prohlížeči. Standardními prostředky (HTML, CSS, JS) bude nutné vytvořit prezentaci algoritmů a správu uživatelského účtu. Tato vrstva má za úkol zpracovávat uživatelskou interakci na webové stránce přehledně a intuitivně.

3.2 Modul pro běh algoritmů

Důležitou částí této práce je výběr architektury modulu zpracovávající běh algoritmů počítačového vidění. Zvolená architektura ovlivní náročnost implementace, náročnost nasazení na produkční servery a celkovou robustnost řešení. Tento modul webové služby má na starost:

- spouštění a monitorování algoritmů,
- dostupnost uživatelem nahraných souborů,
- zpracování výsledků algoritmů.

Pro realizaci se nabízejí dvě základní možnosti:

1. Algoritmy budou spouštěny v rámci webového a API serveru.
2. Bude vytvořen oddělený server, který bude zpracovávat algoritmy a komunikovat s webovým a API serverem.

První případ je oproti druhému výrazně jednodušší na implementaci a při malém počtu uživatelů nemá větší nevýhody. Jedinou nevýhodou je sdílení zdrojů serveru (operační paměť, procesorový čas) mezi webovým serverem a spuštěnými algoritmy. Při spuštění náročných algoritmů může být ovlivněn výkon webového serveru.

Druhý případ počítá s oddělenými servery pro webový server a pro server, kde poběží algoritmy – *server pro běh algoritmů*. Toto řešení by nabízelo možnost škálovat zátěž pouze přidáním dalších serverů pro běh algoritmů. Nevýhodou je nutnost specifikování a zabezpečení komunikačního protokolu mezi oběma servery s čímž souvisí složitější a náročnější implementace.

V rámci této práce byl zvolen nejprve první způsob z důvodu jednoduchosti a rychlosti implementace. Po úspěšném zprovoznění a ověření funkčnosti byl vybrán druhý, složitější a robustnější způsob architektury služby.

3.2.1 Architektura odděleného serveru pro běh algoritmů

Kromě komunikačního protokolu zmíněného v předešlé části je nutné také navrhnout způsob sdílení souborů mezi servery. V případě, kdy bylo spouštění algoritmů v rámci webového serveru, stačilo ukládat nahrané soubory lokálně na pevný disk. Při spuštění algoritmu byly uživatelem nahrané soubory dostupné, jelikož se nacházely na stejném serveru. Nabízí se dvě možnosti, kde nahrané soubory ukládat:

1. uložit soubory na webový server nebo
2. uložit soubory na samostatný server.

Oba uvedené způsoby přinášejí nutnost v případě žádosti o spuštění algoritmu distribuovat soubory na server, kde daný algoritmus bude spuštěn. Výhodou prvního způsobu je jednodušší architektura, a tím i jednodušší implementace. Naopak nevýhodou tohoto řešení je svázání uložiště s webovým serverem. To neumožní použití žádného existujícího řešení.

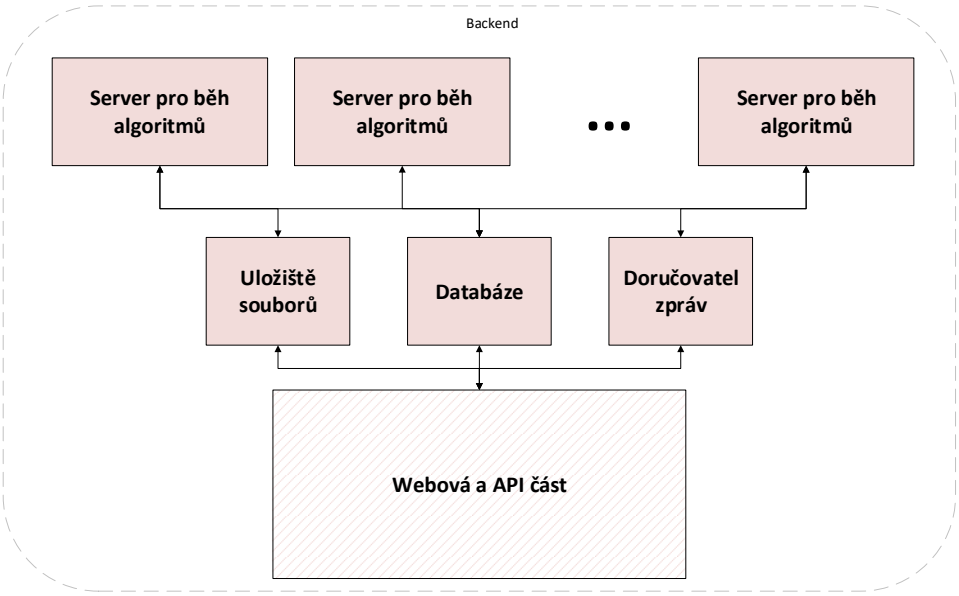
Druhé řešení přináší několik výhod. Umožňuje škálovat webový server přidáním další instance webového serveru. V prvním případě by to znamenalo mít několik oddělených uložišť souborů a muset řešit složité dohledávání a přesouvání souborů v případě zrušení instancí webového serveru. Další výhodou druhého řešení je menší zatížení webového serveru. Z těchto důvodů bylo zvoleno pravě toto řešení. Správa uživatelských souborů a její integrace ve webové službě bude rozebrána v kapitole 3.4.

Výsledná architektura odděleného serveru je zobrazena na obrázku 3.2. Kromě zmíněných částí je součástí architektury databáze uvedená v předcházející sekci 3.1 a systém pro doručování zpráv. Tento systém zajišťuje přerozdělení zpráv mezi servery pro běh algoritmů. Tyto zprávy obsahují instrukce pro spuštění algoritmů. Přes stejný systém je následně zaslána zpráva webovému serveru o běhu algoritmu. Podrobněji bude tato část popsaná v následující sekci 3.2.2.

3.2.2 Komunikace mezi servery

Základním požadavkem komunikace mezi webovými servery a servery pro běh algoritmů je doručení příkazu ke spuštění algoritmů a následné zaslání výsledku průběhu zpět na webový server. Jednotlivé servery pro běh algoritmů musí být možné dynamicky přidávat a odebírat z komunikačního kanálu. To umožní přidávat instance serverů na základě aktuální zátěže.

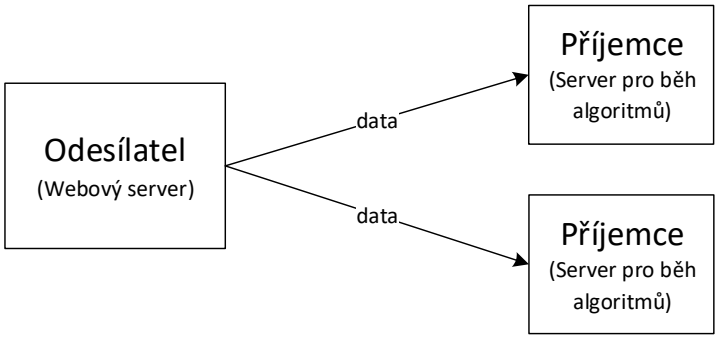
Komunikace může probíhat přímým spojením mezi vzdálenými subjekty viz 3.3. Jedná se o jednoduché a rychlé řešení k výměně informací. Toho může být dosaženo například



Obrázek 3.2: Architektura části systému mající na starost spouštění algoritmů počítačového vidění a komunikaci s webovým a API serverem (podrobnější popis této části je na obr. 3.1).

přes protokol TCP/IP s použitím BSD socketů. K uskutečnění spojované komunikace musí být splněno několik předpokladů [11]:

1. všechny subjekty musí být dostupné v jeden moment,
2. každý subjekt musí znát adresu (např. IP adresu) jiných subjektů,
3. subjekty se musí shodnout na formátu zpráv.



Obrázek 3.3: Přímá, spojovaná komunikace

Závislost mezi subjekty může být měřena jako počet předpokladů nutných ke komunikaci těchto subjektů [11]. Zasílání zpráv je příklad volně propojené (anglicky loosely coupled) komunikace, kde zpráva je informační blok, který se snaží uvedené předpoklady minimalizovat. Místo zasílání informace přímo na specifickou adresu, může být zpráva zaslaná na adresovatelný kanál a o doručení zprávy se postará samostatný systém, který zprávu do-

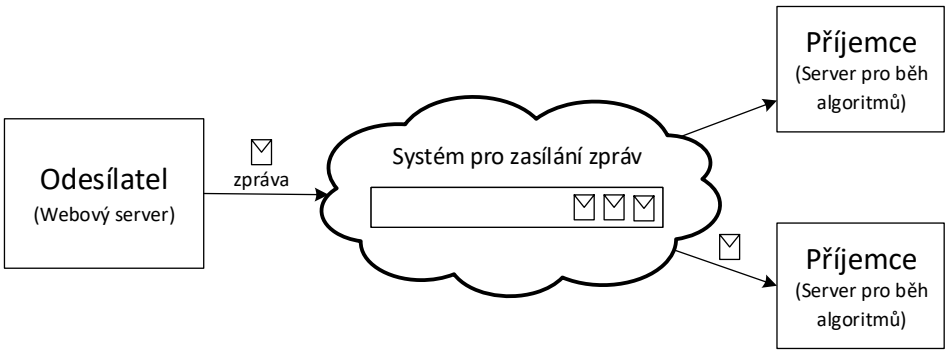
ručí. Pro odstranění dočasné závislosti mezi subjekty může takovýto kanál zprávu uchovávat a doručit až při dostupnosti příjemce.

Na obrázku 3.2 lze vidět systém pro doručení zpráv zobrazený jako samostatný modul. Oproti případu, kdy by doručovaní zpráv zajišťoval samotný webový server, má oddělený modul několik výhod:

1. obecně lepší rozdělení odpovědnosti jednotlivých modulů (webový server obsluhuje HTTP dotazy, modul pro doručování zpráv doručuje zprávy),
2. v případě přidání další instance webového serveru (při velké zátěži) existuje pouze jeden centrální modul pro doručení zpráv, čímž je zjednodušena komunikace mezi servery,
3. jediný modul řeší doručování zpráv, proto může zajistit spravedlivé rozdělení zpráv, a tím rovnoměrně zatížit všechny servery pro běh algoritmů,
4. servery nemusí znát svoje umístění (TCP/IP adresu), kromě adresy modulu pro doručování zpráv.

Systém pro zasílání zpráv

Systém pro zasílání zpráv, jak je znázorněno na obrázku 3.4, působí jako nezávislá vrstva mezi objekty, které chtějí komunikovat. Nejčastější implementace tohoto systému je tzv. *zprostředkovatel zpráv* [11] (anglicky message broker) a je zodpovědný za zprostředkování zpráv od jednoho objektu k jinému tak, aby se objekty mohly soustředit na to, co zasílají, a ne jak to doručit.



Obrázek 3.4: Komunikace pomocí zasílání zpráv s využitím systému pro zasílání zpráv

Základním stavebním blokem těchto systémů je zpráva [11]. Skládá se z těla, které obsahuje samotná strukturovaná data (např. JSON, XML, protobuf), a hlavičky, která obsahuje hodnoty ve formě klíč-hodnota používaných pro směrování zprávy.

Standardně se využívají různé modely pro komunikaci, které definují, jak je zpráva doručena od odesílatele k příjemci. Nejznámější modely pro komunikaci jsou *fronty* a *témata*. Fronty jsou používány pro implementaci komunikace typu point-to-point, kde v případě, že neexistuje příjemce na daném kanále, je zpráva uchována pro pozdější doručení. V případě, že na daném kanále existuje více příjemců, je zpráva doručena pouze jednomu z nich. Model

založený na tématech slouží pro klasický publish/subscribe scénář, kde v případě neexistujícího příjemce je zpráva zahozena, jinak je zpráva doručena všem příjemcům. Podpora komplexnějších scénářů závisí na zvolené technologii a protokolu.

Systémy pro zasílání zpráv jsou pokročilé samostatné služby, které nabízejí možnost zasílání zpráv distribuovaným aplikacím. Jsou výrazně využívány pro implementování komunikace a integrace v distribuovaných systémech [8], s výjimkou datově náročných a vysoce výkonnostních případů, kde existence samostatné oddělené služby není vhodná [11].

	Multiplatformnost	Licence	Administrace	Pokročilé funkce
RabbitMQ	ano	Mozzila Public License	HTTP API pro správu a monitoring	Podpora velkého počtu protokolů, dobře zdokumentované, mnoho klientských knihoven
Kafka	ano	Apache 2.0	Pokročilé rozhraní od třetí strany	Dosahuje vysokého výkonu i s persistencí zpráv
ActiveMQ	ano	Apache 2.0	Základní webová konzole	Dobře zdokumentované
ZeroMQ	ano	LGPL	Zatím neexistuje	Mnoho klientských knihoven

Tabulka 3.1: Porovnání nejznámějších systémů pro doručování zpráv. Přímé výkonnostní srovnání těchto systémů aktuálně neexistuje [1], avšak ZeroMQ a Kafka se považují za jedny z nejvýkonnějších, co se týče propustnosti zpráv.

Na základě uvedeného srovnání v tabulce 3.1 a po vyzkoušení dostupných klientských knihoven k jednotlivým systémům byl zvolen zprostředkovatel zpráv *RabbitMQ*. Klientské knihovny pro tento systém jsou aktivně vyvíjeny a nabízejí vysokou míru abstrakce pro jednoduchou implementaci. K výběru napomohla dobrá dokumentace a podpora zabezpečení pomocí uživatelských účtů i certifikátů podepsaných samy sebou (anglicky self-signed).

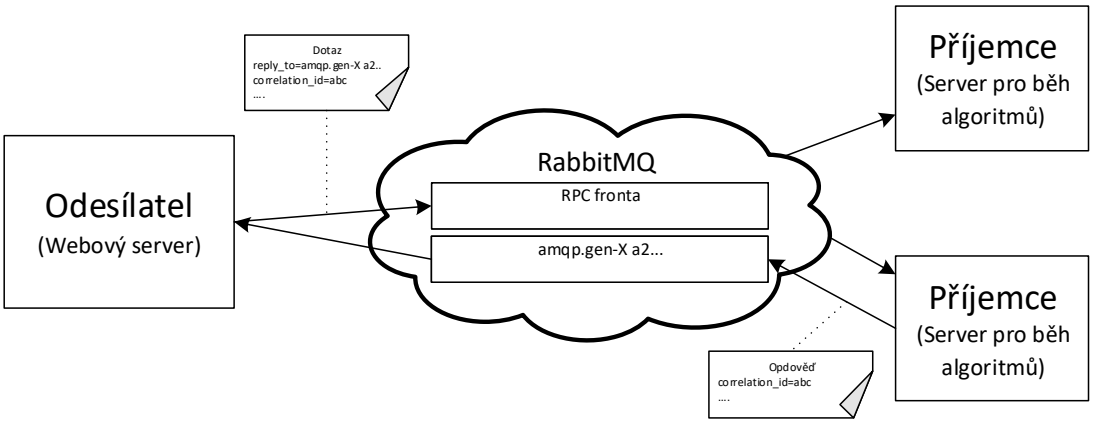
RabbitMQ

RabbitMQ je open-source software pro zprostředkování zpráv, který podporuje protokoly AMQP a STOMP pro doručování zpráv. RabbitMQ server je implementovaný v jazyce Erlang, který je zejména používaný v telekomunikačním odvětví. Klientské knihovny jsou dostupné pro všechny významné programovací jazyky. Architektura RabbitMQ je vysoce modulární, proto mohou být další protokoly pro doručování zpráv přidány skrze přídavné moduly (např. MQTT, HTTP).

RabbitMQ podporuje hlavní funkce zasílání zpráv, jako je perzistence, shlukování a vysokou dostupnost. Kromě klasických modelů pro zasílání zpráv pomocí tématu a fronty lze využít i pokročilejší model typu dotaz/odpověď (známý taky jako vzdálené volání procedur, dále jen RPC). Oproti zmíněným modelům počítá tento model s obousměrnou komunikací, typicky používanou pro rozdělení úkonu mezi servery a následném získání výsledku daných úkonů. Implementace v RabbitMQ je znázorněna na obrázku 3.5. Dotaz je zaslán stejným typem fronty, který by se použil v případě modelu fronty (viz 3.4), ale s tím rozdílem, že v hlavičce je uveden název fronty pro odpověď a unikátní identifikátor pro každou zprávu.

Ve výchozím nastavení je rozdělování zpráv mezi příjemce implementované metodou *round-robin*. To znamená příjemci zpráv se pravidelně střídají. V případě, kdy jeden z příjemců neustále dostává výpočetně náročné úkony (spuštění algoritmů náročných na zdroje) a druhý dostává jednoduché úlohy, bude jeden server neustále zatížený, zatímco druhý bude nevyužitý. K zamezení tohoto jevu lze využít nastavení `prefetch_count`, které umožňuje

[1] Podrobné výkonnostní srovnání dvou zmíněných systémů je zde http://hiramchirino.com/stomp-benchmark/



Obrázek 3.5: Komunikační model typu dotaz/odpověď, tak jak je implementovaný v zprostředkovateli zpráv RabbitMQ.

nastavit, kolik zpráv může být zasláno serveru před tím, než budou předchozí zprávy úspěšně zpracovány. Při tomto nastavení je nutné dbát na možné zaplnění front v případě, kdy jsou všichni příjemci zaneprázdněni.

3.3 Integrace algoritmů

Při návrhu služby bylo nejdůležitější určit, jak výrazně bude potřeba upravit existující implementace algoritmů počítačového vidění k jejich integraci do vytvářené služby. Tímto rozhodnutím je ovlivněno, jak náročné bude přidávání nových algoritmů do služby a také kolik dodatečných informací bude mít služba k dispozici. Mezi dodatečné informace patří například informování o aktuálním průběhu algoritmu (kolik času zbývá k jeho dokončení), kdy v případě dlouhotrvajících algoritmů je vhodné uživatele informovat o jejich průběhu. Tím by ale bylo potřeba definovat rozhraní pro komunikaci programů se serverem, který tento program spustil a čeká na jeho dokončení. Zjednodušeně byly na výběr následující možnosti:

1. Ke všem programům bude potřeba implementovat způsob komunikace se serverem, například přes standardní výstup nebo pomocí dodatečné knihovny, která s touto komunikací pomůže a kterou bude potřeba implementovat.

2. Program nebude potřeba nijak upravovat. Server nebude mít žádné informace o průběhu algoritmu.

Zvoleno bylo druhé zmíněné řešení s minimálním množstvím zásahů do již existujících programů, kdy při vytváření algoritmu není třeba brát větší zřetel k webové službě. Dále jsou využity proměnné prostředí, což je název a hodnota v textové formě, k předání dodatečných informací. Proměnné prostředí jsou předány každému procesu, který je má následně uložené v paměti [16]. Konkrétně je všem programům předána proměnná prostředí s názvem `DESTINATION_DIRECTORY`, která obsahuje cestu ke složce, kde může být zapsán případný výstup z programu. Tento způsob byl vyhodnocen jako nejvhodnější řešení problému, kdy webová služba neví, v jakém formátu bude výstup daného algoritmu, a tedy server vytvoří

složku pro každé spuštění algoritmu, do které případné výstupní soubory algoritmus uloží. Zvažovány byly i další možnosti, mezi které patří změna tzv. *Working Directory* (adresář, ve kterém je algoritmus spuštěn) pro vytvářený proces. Tím by se ale programy, které mají reference na další potřebné knihovny (skripty) dány v relativní formě, staly nefunkční. Dalším zvažovaným řešením bylo předání cesty argumentem. V tomto případě by se musely programy, které takový argument neočekávají, dodatečně upravit.

3.3.1 Spuštění algoritmů

Podobný přístup jako u integrace programů byl zvolen i pro spouštění algoritmů. Při spouštění algoritmů jsou předány argumenty příkazové řádky. Server pro běh algoritmů nijak nevaliduje argumenty, se kterými je program spouštěn, pouze překládá argumenty do tradičního tvaru pro konzolové aplikace (tvar se řídí pokyny POSIX pro argumenty příkazové řádky utilit [17]). Překlad argumentů probíhá z formátu JSON a zaveden je pro dosažení přehlednějšího zápisu argumentů a snadnějšího zpracování speciálních argumentů, viz dále.

Uživatel musí předem vědět, s jakými argumenty je potřeba algoritmus spustit. To se dozví přes prezentační stránky algoritmu, případně přes dokumentaci k danému algoritmu. Další možností je spustit algoritmus s argumentem `--help`. Na spuštění s tímto argumentem by většina algoritmů měla zobrazit možné způsoby spuštění. V tomto případě ale musí uživatel vědět, jak zapsat argumenty tak, aby byly přeloženy do požadovaného formátu.

Překlad argumentů

Uživatel zadá vstupní argumenty algoritmů ve formátu JSON a ty jsou následně přeloženy do formátu daného standardem POSIX [17]. Kořenový element JSON může být jakýkoliv, ale nejčastěji je nejvhodnější pole. Ve formátu JSON je pole heterogenní struktura, může tedy obsahovat hodnoty různých datových typů. Pole je přeloženo tak, že jednotlivé hodnoty jsou přeloženy zvlášť a výstup jejich překladů je oddělený mezerou.

```
[12, "option1"] ⇒ 12 "option1"
```

Jednoduché hodnotové typy, mezi které patří řetězec, číslo, boolean hodnoty a null, se nepřekládají a jsou na výstup vypsány ve stejném tvaru. Použití pole a jednoduchých hodnotových typů by stačilo pro vytvoření jakékoliv reprezentace argumentů příkazové řádky. Pro zpřehlednění zadávání argumentů ve formátu JSON byla přidána možnost zadat argumenty pomocí JSON objektu. Ten se může nacházet na kterémkoliv místě v poli, ale nemůže být obsažen v jiném objektu. Objekt obsahující páry klíč a hodnota se přeloží následovně:

```
{ "threshold": 5, "m": 0.6 } ⇒ --threshold=5 -m 0.6
```

Speciální přístup musel být zvolen v případě zadání souboru jako argumentu. Nejdříve je potřeba nahrát soubor na webovou službu a následně uživatel použije získaný identifikátor nahraného souboru. Při zadání argumentů je daný identifikátor specifikován ve formě `"local://ID"` a následně je přeložen serverem, kde bude spuštěn algoritmus počítačového vidění, na cestu k lokálnímu souboru. Překlad může vypadat následovně:

```
"local://ba85d2da6462418c90994db9e5213433" ⇒ C:\UserFiles\2\image.jpg
```

Identifikátor souboru je ve formátu `UUID` [10], který je podrobněji vysvětlen v následující kapitole 3.4. Webová služba nalezne záznam souboru s daným identifikátorem a nejdříve zkontroluje, jestli je uživatel vlastníkem nahraného souboru. V kladném případě získá název souboru a vytvoří cestu k souboru, která je navíc daná identifikátorem uživatele, v tomto případě číslo 2. Výsledný překlad všech argumentů z formátu JSON potom vypadá například takto:

```
[{"threshold": 5, "f": 0.6}, "local://ba85d2da6462418c90994db9e5213433"]
                                    ⇓
        --threshold=5 -f 0.6 -t C:\UserFiles\2\random_file2
```

Z příkladu lze vidět, že poziční argumenty nijak nejsou ošetřeny, a uživatel si musí pořadí ohlídat sám. V ideálním případě budou algoritmy používat pouze pojmenované argumenty, ve kterých je zřetelný sémantický význam argumentu. Pro zadání pojmenovaných argumentů je nutné použít JSON objekty, které v sobě obsahují páry klíč a hodnota, kde klíč musí být řetězec a hodnota může být jakýkoliv JSON datový typ. Pro zachování jednoduchosti přijímá překladač pouze jednoduché datové typy. Hodnota nemůže být JSON objekt nebo pole, jelikož by se vyjadřovací schopnost nijak nezvýšila, pouze by přibyly možnosti, jak zapsat výsledné argumenty příkazové řádky.

Existují dva speciální případy, ve kterých se může zadání argumentů zjednodušit:

1. Algoritmus přijímá pouze pojmenované argumenty – argumenty nemusí být zadány v seznamu, jelikož nezáleží na jejich pořadí. Příklad: `{ "threshold": 5, "min": 0.6 }`
2. Algoritmus přijímá jenom jeden argument – argument může být zadán samostatně. Příklad: `"example_argument"`

3.4 Správa uživatelských souborů

Webová služba spouští algoritmy počítačového vidění. Vstupem těchto algoritmů jsou výhradně multimediální soubory: obrázky a videa. Je tedy nutné dovolit uživatelům použít své multimediální soubory jako vstupní argumenty pro spuštění algoritmů. Zacházení se soubory se liší od klasických argumentů, například textu. Soubory obsahují velké množství dat v binární podobě, nečitelné formě pro uživatele.

V předcházející kapitole 3.3.1 byl uveden formát JSON pro zadávání argumentů. JSON formát neobsahuje způsob pro vkládání souborů, a nelze tedy poslat zároveň s argumenty i soubor, který bude při spuštění algoritmu využit. Řešením je nahrání souboru na servery webové služby ještě před samotným spuštěním algoritmu. Tento způsob také dovoluje opakovaně využít stejný soubor, například spuštěním s mírně odlišnými argumenty. Kromě uložení souborů na vlastní server se do databáze přidává záznam s následujícími informacemi:

- identifikátor záznamu,
- identifikátor souboru v uložišti,
- identifikátor uživatele, který soubor nahrál,
- md5 hash,
- typ souboru (MIME) a

- velikost souboru.

Pro identifikátor záznamu byl zvolen datový typ UUID, který byl vytvořen zejména pro decentralizovanou správu identifikátoru [10]. Jelikož uživatelé služby přímo s identifikátory souborů pracují, byl zvolen datový typ, který uživateli nedá informace, kolik je již nahraných souborů. Sdílení souborů mezi uživateli není dovoleno, jelikož by přinášelo množství bezpečnostních rizik. Poslední zmíněné informace jsou uchovány zejména z důvodů monitorování a vytváření statistik. Velikost souborů lze později využít při zavedení omezení na celkovou velikost všech uživatelem nahraných souborů, což by byla jedna z podmínek při produkčním nasazení.

3.4.1 Uložiště souborů

Pro ukládání souborů, jak již bylo zmíněno v předcházející části 3.2.1, bylo zvoleno řešení obsahující samostatný server. Požadavky na tento server jsou: uložení souboru, získání souboru, smazání souboru. Implementovat server s těmito požadavky by bylo relativně jednoduché, kdyby nebylo potřeba řešit autentizaci ostatních serverů k přístupu k souborům. V takovém případě by se implementace zkomplikovala, a proto bylo zvoleno již existující řešení.

Mezi nejvhodnější kandidáty pro uložiště souborů patří dokumentová databáze MongoDB. V případě využití cloudových služeb je možné použít tam jimi nabízené služby, např. Azure Storage, Amazon Elastic File System a Google Cloud Storage.

MongoDB

MongoDB je zdarma dostupná, open-source, multiplatformní dokumentová databáze [2]. Řadí se mezi NoSQL databáze. Ukládá dokumenty ve formátu BSON, který vychází z binární reprezentace formátu JSON.

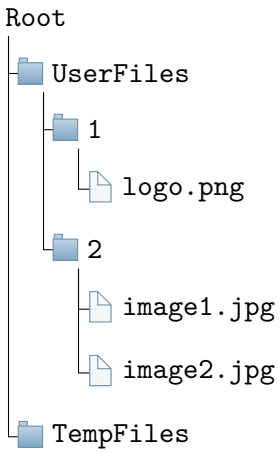
Mezi hlavní rysy MongoDB patří vyvažování zátěže a replikace dat přes více počítačů, použití jako uložiště souborů a přítomnost agregačního frameworku. Dokumenty ve formátu BSON mají maximální velikost 16 MB, pro větší soubory se používá specifikace GridFS (File System) popisující ukládání a získávání souborů [3]. Namísto uložení souboru v jednom dokumentu, GridFS rozdělí soubor na části a ty následně uloží do samostatného dokumentu. Spolu s daty jsou uložena také metadata o souboru, mezi která patří md5 hash, typ souboru atd. Specifikace GridFS nepřináší žádná omezení na maximální velikost souboru.

3.4.2 Uživatelské soubory na serveru pro běh algoritmů

Uživatelské soubory je nutné dopravit na server, kde poběží spouštěný algoritmus. Při specifikování argumentů obsahujícího soubor je daný soubor zkopírován z globální uložiště do lokální adresářové struktury viz 3.6. Server ukládá uživatelské soubory do oddělených složek podle identifikátoru uživatele pro zamezení úmyslného nebo neúmyslného ovlivnění.

Kromě složky pro uživatelské soubory je na serveru složka pro ukládání dočasných souborů. Ve složce pro dočasné soubory se vytvářejí adresáře pro uložení výsledků algoritmů. V případě, že po ukončení algoritmu, obsahuje adresář minimálně jeden soubor, jsou soubory uvnitř zabaleny a zkomprimovány do jednoho výsledného souboru. Tento soubor je následně nahrán do globálního uložiště a identifikátor souboru zaslán uživateli v rámci odpovědi.

[2] https://www.mongodb.com/
[3] https://docs.mongodb.com/manual/core/gridfs/



Obrázek 3.6: Ukázka adresářové struktury na serveru pro běh algoritmů obsahující soubory dvou uživatelů.

3.5 Výběr algoritmů

Pro demonstraci služby byly zvoleny algoritmy s ohledem na rozdílné typy vstupních argumentů a také rozdílné prezentování výsledků těchto algoritmů. Mezi rozdílné typy argumentů patří: text, číslo, obrázek a všeobecně jakýkoli typ souboru. Rozdíl v algoritmech může také být v použití libovolného počtu obrázků na vstupu nebo v pojmenovaných argumentech. Výstup programů provádějící dané algoritmy může být jak textový, tak v podobě souborů. Jako základní algoritmy pro demonstraci funkčnosti služby byly vybrány následující:

- detekce obličejů
  - vstup: obrázek
  - výstup: pozice obličejů na obrázku nebo obrázek s vyznačenými obličeji
- zabarvení černobílé fotografie
  - vstup: černobílá fotografie
  - výstup: barevná fotografie
- porovnání podoby obrázků
  - vstup: dva obrázky
  - výstup: textové skóre
- vytvoření panoramatického snímku
  - vstup: více obrázků
  - výstup: jeden obrázek
- obecně Haarovy kaskády
  - vstup: konfigurační soubor Haarovy kaskády a obrázek
  - výstup: obrázek s označenými detekovanými oblastmi

- rozpoznání poznávacích značek
  - vstup: obrázek
  - výstup: poznávací značka v textové podobě
- optické rozpoznávání znaků
  - vstup: obrázek
  - výstup: text z obrázku ve strojově čitelné podobě
- stabilizace obrazu ve videu
  - vstup: video
  - výstup: stabilizované video

Pro některé z uvedených algoritmů bude vytvořena prezentační stránka, která dovolí prostřednictvím webového prohlížeče spouštět algoritmy a pomocí webového formuláře měnit vstupní argumenty algoritmů.

3.6 Výběr backend technologií

Výběr technologií je zvolen na základě požadavků pro danou technologii. Nejdůležitějším rozhodnutím je zvolení webového frameworku, od kterého se bude odvíjet zvolení dalších technologií. V tabulce 3.2 je uvedeno základní porovnání webových frameworků, které patří mezi nejpoužívanější.

	ASP.NET	ASP.NET Core	Django	Ruby on Rails	Symfony2
Programovací jazyk	C#	C#	Python	Ruby	PHP
Multiplatformnost	ne	ano	ano	ano	ano
Licence	Apache 2.0	Apache 2.0	BSD	MIT	MIT
Open-source	ne	ano	ano	ano	ano
Výkon – plain text odpověď[4]	neuvedeno	1.	2.	3.	neuvedeno
Výkon – JSON serializace[5]	neuvedeno	1.	2.	3.	4.

Tabulka 3.2: Porovnání webových frameworků (Pozn.: u výkonnostního porovnání je uvedeno relativní pořadí od nejvýkonnějšího frameworku)

.

Nejlépe ze zmíněných frameworků dopadl relativně nový webový framework *ASP.NET Core*, který vynikal hlavně v oblasti výkonnosti. ASP.NET Core vznikl evolucí z frameworku ASP.NET, který ale byl pevně svázaný s operačním systémem Windows.

[4] Zdroj: https://www.techempower.com/benchmarks/#section=data-r13&hw=ph&test=plaintext
[5] Zdroj: https://www.techempower.com/benchmarks/#section=data-r13&hw=ph&test=json

3.6.1 ASP.NET Core

ASP.NET Core[6] je open-source, multiplatformní framework pro vytváření aplikací s připojením k počítačové síti, jako jsou webové aplikace, IoT aplikace a mobilní backendy. Základním konceptem frameworku je modulárnost, kdy podle potřeb vývojáře lze přidávat další funkčnost frameworku pomocí balíčků. Webový framework může být spuštěný na operačních systémech Windows, Mac a linuxových distribucích. První verze 1.0 byla představena v červnu 2016[7].

3.7 Architektura aplikačního rozhraní

Pro přístup k aplikačnímu rozhraní (zkráceně API) webové služby existují dva nejčastější způsoby: SOAP (Simple Object Access Protocol) a REST (Representational State Transfer). Před výběrem protokolu budou zběžně představeny společně s jejich hlavními vlastnostmi.

3.7.1 SOAP

SOAP je protokolem pro výměnu zpráv založených na XML přes síť, hlavně pomocí HTTP protokolu [14]. Vytvořený byl společností Microsoft v roce 1998. Formát SOAP tvoří základní vrstvu komunikace mezi webovými službami a poskytuje prostředí pro tvorbu složitější komunikace. Existuje několik různých druhů šablon pro komunikaci na protokolu SOAP [14]. Nejznámější z nich je RPC šablona, kde jeden z účastníků komunikace je klient a na druhé straně je server. Server ihned odpovídá na požadavky klienta. Mezi nevýhody protokolu patří velká „výřečnost“ komunikace a složitost. Jako další nevýhodu lze považovat složitost parsování a validací zpráv.

3.7.2 REST

REST je architektura rozhraní pro jednoduché vytváření, čtení a editování informací ze serveru pomocí HTTP volání. REST navrhl a popsal v roce 2000 Roy Fielding (jeden ze spoluautorů protokolu HTTP) v rámci disertační práce. V této práci popsal několik požadavků, které REST služba musí splňovat [3]:

- REST je určené pro architekturu klient–server,
- komunikace musí být bezstavová tak, aby každý požadavek klienta obsahoval veškeré potřebné informace k pochopení požadavku,
- manipulace se zdroji by měla být jednotná pomocí URI a je na klientovi, v jaké formě data získá,
- použití dalších webových mezivrstev v podobě proxy serverů, brán (gateway) atp. nesmí nijak narušit funkčnost služby,
- server by měl v každé odpovědi deklarovat chování dat s ohledem na mezipaměť (anglicky cache), jestli lze data v mezipaměti uchovat, nebo je nutné se na data pokaždé dotazovat.

[6] https://www.asp.net/core
[7] Zdroj: https://blogs.msdn.microsoft.com/dotnet/2016/06/27/announcing-net-core-1-0/

Většina reálných REST služeb nesplňuje veškeré zmíněné požadavky. Rozhraní REST se používá pro jednotný přístup ke zdrojům. Zdrojem mohou být data, stejně jako stavy aplikace (pokud je lze popsat konkrétními daty). REST je tedy na rozdíl od SOAP orientován datově, nikoli procedurálně. REST nijak nepředepisuje formát jednotlivých zpráv, lze tedy zvolit například XML, HTML, YAML, nejčastěji se však používá JSON.

Kvůli jednoduchosti a velkému rozšíření bylo vybráno rozhraní REST pro architekturu API serveru. Dalším výrazným plusem je nezávislost architektury na formátu přenášených dat, což přispívá k snadnějšímu napojení na různé technologie a různé programovací jazyky.

3.7.3 Identifikace zdrojů

Adekvátní zvolení URI (anglicky celým názvem uniform resource identifier, česky jednotný identifikátor zdroje) je bezesporu důležitou částí při vytváření srozumitelného a snadno použitelného API. Základní princip architektury REST zahrnuje rozdělení API do logických zdrojů, např. uživatel, algoritmus, soubor. S těmito zdroji je manipulováno skrze HTTP dotazy, kde metody `GET`, `POST`, `PUT`, `HEAD`, `DELETE` mají rozdílný význam. Při vytváření REST API existuje mnoho pravidel a doporučení. Mezi nejdůležitější patří následující [12]:

- zdroji by měla být výhradně podstatná jména,
- pro názvy kolekcí zdrojů v URL by mělo být použito množné číslo,
- pro dokument (např. jeden záznam v databázi) by mělo být použito jednotné číslo,
- GET musí být použito pro získání reprezentace zdroje,
- POST musí být použito pro vytvoření nového zdroje v kolekci,
- DELETE musí být použito pro odstranění zdroje,
- PUT se používá pro vytvoření a aktualizaci uložených zdrojů.

Ve webové službě byly identifikovány tři hlavní zdroje: algoritmy, běhy algoritmů a soubory. V tabulce 3.3 jsou výsledné specifikované URI nejdůležitějších HTTP dotazů.

Pro spuštění algoritmu byl namísto klasických číselných identifikátorů zvolen tzv. *kód* algoritmu, např. hello-world, to-grayscale, find-faces. Tím je dosaženo větší přehlednosti, kdy hned z URL adresy lze vidět, který algoritmus je spouštěn. Administrátor bude muset při vytváření algoritmu pouze zadat unikátní řetězec reprezentující daný algoritmus.

3.8 Návrh schématu databáze

Na základě požadavků si nejdříve nadefinujeme všechny hlavní entity, které v systému vystupují. Patří mezi ně entity `uživatel`, `soubor`, `algoritmus` a `běh algoritmu`, které lze vidět v diagramu 3.7 i s atributy a vztahy mezi entitami. Jelikož entita `uživatel` slouží jak pro administrátora, tak pro klasického uživatele (uživatel vystupující v diagramu užití 2.1), je potřeba tyto dva typy rolí nějak rozlišit. K tomu slouží entita, která nebyla uvedena v diagramu a která obsahuje název role, do níž uživatel patří. Navázána je na entitu `uživatel` vztahem 1:N, což dovoluje přiřadit více rolí.

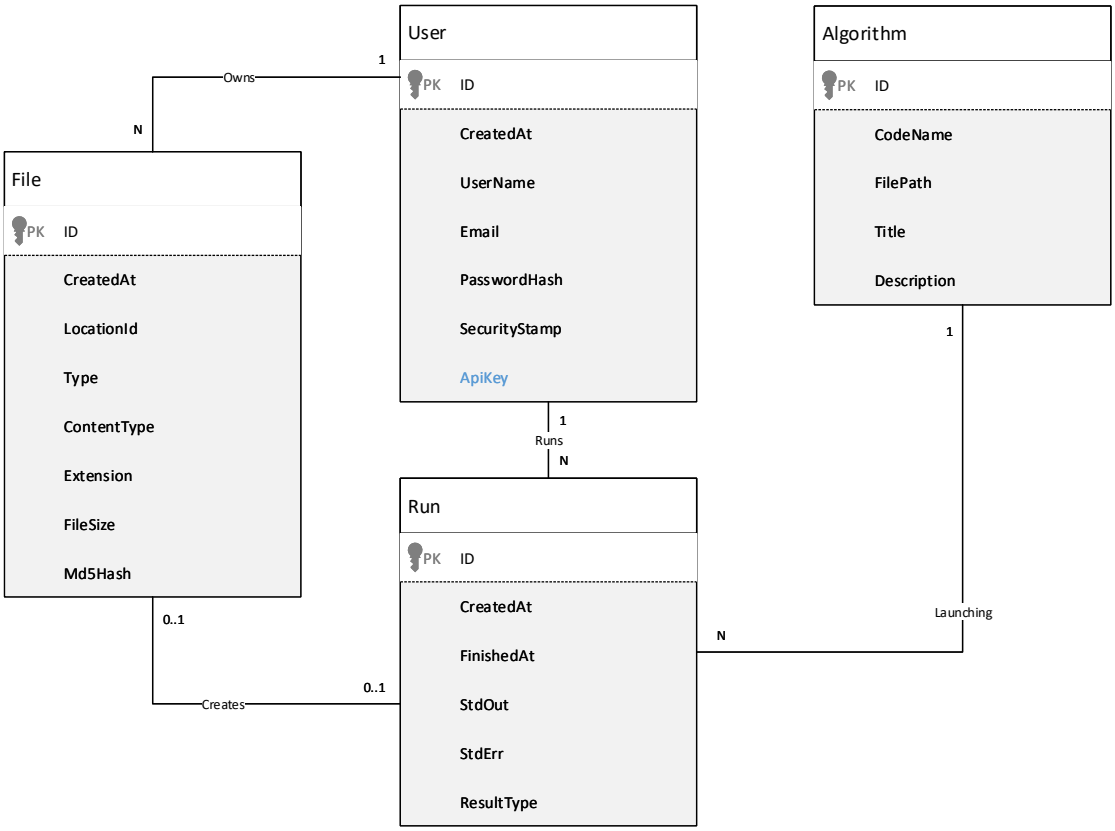
Metoda	URI	Popis dotazu
`POST`	`/algorithms/hello-world`	spustí algoritmus *Hello World*
`GET`	`/algorithms/hello-world`	získá základní informace o algoritmu
`GET`	`/algorithms/`	získá seznam dostupných algoritmů
`GET`	`/runs/{run-id}`	získá informace o průběhu algoritmu s identifikátorem `{run-id}`
`POST`	`/files`	nahraje soubor(y) na webovou službu
`GET`	`/files`	získá informace o všech souborech uživatele
`GET`	`/files/{file-id}`	stáhne soubor s identifikátorem `{file-id}`
`DELETE`	`/files/{file-id}`	smaže soubor s identifikátorem `{file-id}`
`HEAD`	`/files/{file-id}`	získá metainformace o souboru s identifikátorem `{file-id}`

Tabulka 3.3: Výsledné specifikované URI pro nejdůležitější HTTP dotazy API serveru

Za povšimnutí v entitě `uživatele` stojí atribut `API klíč` (anglicky API Key). Bude sloužit k zajištění autentizace pro API. Entita `algoritmus` obsahuje relativní cestu k spustitelnému souboru daného programu implementující algoritmus a základní popis tohoto algoritmu.

Pro ukládání jednotlivých spuštění daného algoritmu slouží entita `běh algoritmu`. Ta obsahuje zaznamenaný standardní výstup a standardní chybový výstup při běhu algoritmu. Může taky obsahovat referenci na zkomprimovaný soubor, který obsahuje soubory vygenerované spuštěným algoritmem. Entita zahrnuje dva atributy zaznamenávající přesný čas vytvoření a ukončení běhu algoritmu, ze nichž lze získat celkové trvání spuštěného algoritmu. V posledním atributu (viz ResultType) je aktuálně uložený stav běhu: úspěšně ukončený, neúspěšně ukončený, pořád běží a ukončený vypršením časového limitu.

V entitě `soubor` jsou uchovány následující informace: identifikátor pro uložiště souborů, typ souboru (nahraný uživatelem, vygenerovaný algoritmem), typ souboru z hlediska formátu, přípona souboru, velikost souboru a md5 hash. Název souboru se neuchovává a při použití souboru na serveru pro běh algoritmů je vygenerován náhodný název. Většina programů při kontrole vstupních souborů kontroluje správný formát soubory pomocí přípony, proto je nutné příponu souboru zachovat. Vygenerovaný název se tedy sestává z náhodného názvu a původní přípony.



Obrázek 3.7: ER diagram

Kapitola 4

Implementace

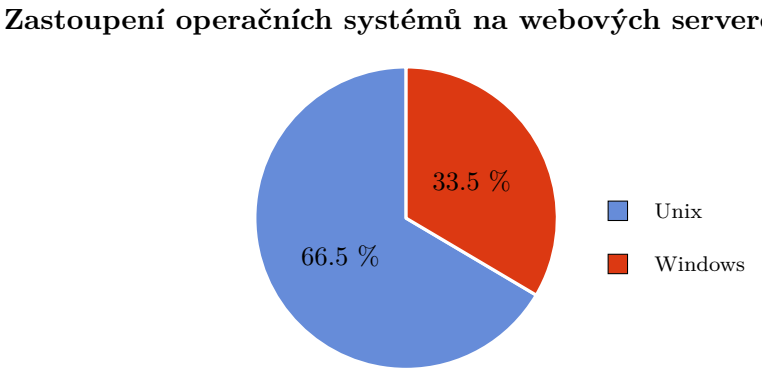
Po návrhu lze přistoupit k implementaci. Z ní jsou v této kapitole popsány pouze ty nejdůležitější části. Patří mezi ně samotný webový framework ASP.NET Core. Dále jsou popsány základní principy, které jsou napříč celým frameworkem používány. Jedná se hlavně o techniku předávání závislosti (anglicky *Dependency Injection*), jež je podrobně popsaná v sekci 4.1.3. Přístup k databázi a použití knihovny *Entity Framework Core* jsou popsány v části 4.1.5. Následuje představení platformy *Docker* a důvody k jejímu použití. Použité technologie pro dokumentaci API jsou rozebrány v sekci 4.1.7. Poté následuje sekce věnující se architektuře s odděleným serverem. Nejdůležitější je popis samotného serveru pro běh algoritmů viz 4.2. Popis části frontend, resp. výsledná webová služba je v sekci 4.4, resp. 4.5. Vytvořená klientská API knihovna je rozebrána v sekci 4.6. Na závěr je vysvětleno provedené testování služby viz 4.7 a stručně rozebrán možný směr dalšího vývoje viz sekce 4.8.

4.1 ASP.NET Core

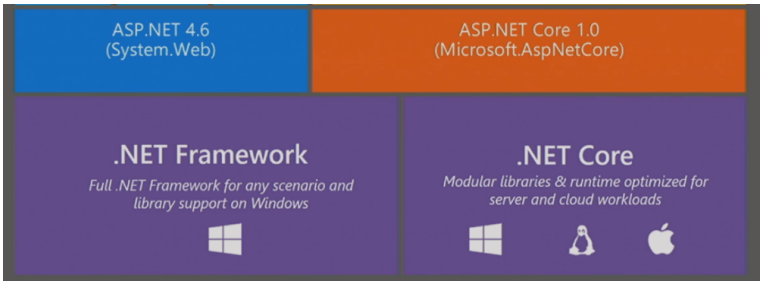
V předchozí kapitole 3.6.1 byl tento webový framework představen. První verze byla uvedena v červnu 2016, jedná se tedy o nový a moderní framework. Jeho vývojáři uplatnili získané zkušenosti z vývoje frameworku ASP.NET, který byl pouze pro operační systém Windows, přičemž všechny jeho závislosti na operačním systému byly abstrahovány. Lze jej tedy použít ve většině používaných linuxových systémech (např. Ubuntu, Debian, RHEL), které mají majoritní zastoupení na webových serverech. K roku 2017 mají linuxové systémy zastoupení 66,5 % na webových serverech, viz 4.1.

Dalším vylepšením oproti předchozímu webovému frameworku od firmy Microsoft je modulárnost v podobě balíčků. Její hlavní výhodou je, že uživatel používá pouze tu funkcionalitu, kterou nutně potřebuje, což má za následek větší výkon a menší velikost aplikace.

ASP.NET Core může běžet na `.NET 4.5.1+` a `.NET Core` frameworcích [2]. Jedná se o prostředí potřebné pro běh aplikací, které nabízí množství knihoven a obsahuje jazykovou interoperabilitu pro několik programovacích jazyků. Obě zmíněné verze jsou vyvíjené zároveň. .NET Core je multiplatformní open-source framework s pouze částečnou funkcionalitou oproti .NET frameworku, který je určený pouze pro operační systém s Windows a který je vyvíjený od roku 2002. První verze frameworku .NET Core byla vydána v roce 2016.



Obrázek 4.1: Graf znázorňuje procentuální zastoupení operačních systémů na webových serverech pro 10 milionů nejnavštěvovanějších webových stránek k 20. dubnu 2017.
Zdroj dat: https://w3techs.com/technologies/overview/operating_system/all



Obrázek 4.2: Přehled architektury webového frameworku ASP.NET Core, který může běžet na .NET Framework i .NET Core [9].

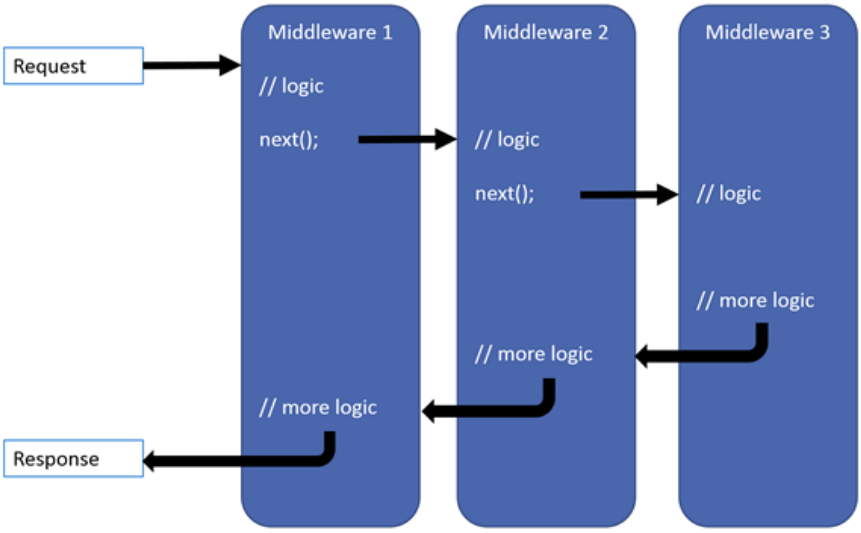
Pro vývoj webové služby byl zvolen multiplatformní .NET Core. Kromě možnosti spuštění pod různými operačními systémy nabízí větší flexibilitu při nasazování a větší výkon, hlavně díky nižší paměťové náročnosti a rychlejším časům spuštění [2].

Aktuálně jsou podporované dvě verze frameworku ASP.NET Core: 1.0.x s označením dlouhodobé podpory a 1.1.x, která je použita v práci díky tomu, že obsahuje nejaktuálnější technologie. Vydání nové verze frameworku .NET Core 2.0 a ASP.NET Core 2.0 je plánováno na 3. kvartál roku 2017 [1]. Mezi hlavní novinky nové verze webového frameworku ASP.NET Core patří *Razor Pages* (zjednodušený způsob vytváření stránek) a *SignalR* (knihovna pro real-time komunikaci mezi serverem a webovým prohlížečem) spolu s dalším vylepšením výkonu [2].

4.1.1 Koncept middleware komponent

Middleware je softwarová komponenta, která je umístěna v řetězci, jenž zpracovává síťové požadavky a odpovědi [1]. Každá komponenta si vybírá, zda předá požadavek další komponentě v řetězci, a může provádět určité akce před a po vyvolání další komponenty v řetězci, viz 4.3. Tento řetězec v ASP.NET Core zpracovává každý HTTP dotaz. Řetězec se nastavuje v metodě `Configure` v `Startup` třídě.

[1] Zdroj: https://github.com/aspnet/Home/wiki/Roadmap
[2] Zdroj: https://blogs.msdn.microsoft.com/webdev/2017/05/10/aspnet-2-preview-1/



Obrázek 4.3: Diagram znázorňuje možný průběh programu během zpracování HTTP dotazu. Jednotlivé komponenty rozhodují, zda požadavek předají dál, nebo samy vrátí odpověď. Zdroj: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware

Každá middleware komponenta může provádět operace před další komponentou a po ní. Komponenta může také rozhodnout, že nepředá žádost další komponentě, což je často žádoucí, protože to umožňuje vyhnout se zbytečné práci. Například pro middleware, který obsluhuje statické soubory, je vhodné neposílat požadavek další komponentě. Nejpoužívanější middleware komponenty jsou dostupné v rámci frameworku s tím, že vývojář může implementovat vlastní pro specifické účely.

Pořadí, ve kterém jsou middleware komponenty zaregistrovány do řetězce v metodě `Configure`, určuje pořadí, ve kterém obsluhují jednotlivé HTTP požadavky. Toto pořadí je důležité hlavně pro middleware zpracovávající zabezpečení, který musí být zaregistrován mezi prvními.

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseStaticFiles();
app.UseApiAuthentication();
app.UseIdentity();
app.UseMvc();
app.UseSwagger();
```

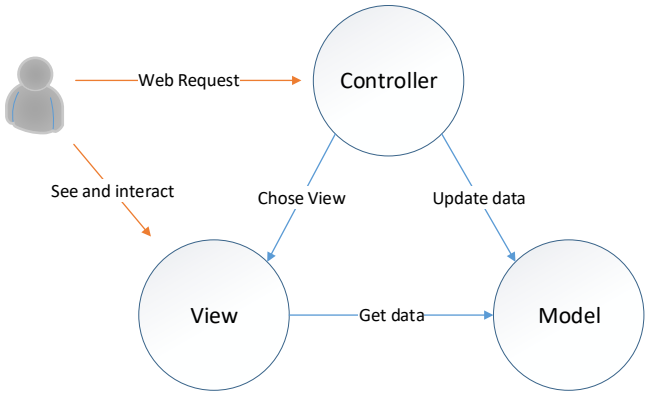
Ukázka 1: Ukázka konfigurace middleware řetězce, který definuje, jak bude každý požadavek zpracován a v jakém pořadí budou jednotlivé moduly spouštěny. Ukázka je z metody `Configure` vytvořené ve webové aplikace.

V ukázce kódu 1 lze vidět registrace middleware komponent do řetězce. Když se webová aplikace nachází v testovacím prostředí, je zaregistrován middleware pro zpracování výjimek, který zobrazí vývojáři přehlednou stránku s popisem chyby. Další v pořadí je middleware pro zpracování statických souborů, mezi které patří skripty, styly, obrázky apod. Následuje registrace dvou middleware komponent, jež řeší autentizace uživatelů, jak pomocí API klíče (první zvýrazněný řádek), tak pomocí *Cookies* (druhý zvýrazněný řádek). Nejdůležitější middleware *MVC* je zaregistrován mezi posledními. Obsluhuje všechny API dotazy a dotazy na HTML stránky. Poslední je zaregistrována komponenta pro vytvoření Swagger dokumentace, viz 4.1.7.

4.1.2 ASP.NET Core MVC

ASP.NET Core MVC je prezentační framework pro vytváření webových aplikací a API aplikací pomocí návrhového vzoru model-pohled-kontrolér (anglicky model-view-controlller, zkráceně MVC). Jedná se o jediný oficiální prezentační framework nad ASP.NET Core. Naproti tomu ASP.NET obsahuje tři frameworky: Web Forms, Web API a MVC. Jelikož tento návrhový vzor ovlivňuje architekturu aplikace mnohem více než klasické návrhové vzory, je často chápán jako architektonický vzor.

MVC rozděluje aplikaci do tří hlavních skupin komponent: modely, pohledy a kontroléry, viz 4.4. Pomocí tohoto vzoru jsou požadavky uživatele směrovány do kontroléru, který je zodpovědný za práci s modelem k provádění uživatelských akcí anebo získání výsledků dotazů. Kontrolér zvolí pohled, jenž je odeslán uživateli a poskytne mu všechna data modelu, které pohled potřebuje.



Obrázek 4.4: Schéma architektonického vzoru MVC

Nejvýznamnější výhodou vzoru MVC je rozdělení zodpovědnosti jednotlivých částí. Každá část má jeden konkrétní úkol. Tímto vznikne kód, který je snadnější měnit, testovat a ladit. Následuje podrobnější popis jednotlivých částí MVC architektonického vzoru.

1. `Model` – reprezentuje stav aplikace a jakoukoliv obchodní logiku nebo operace, které by měly být provedeny.

2. `Pohled` – je zodpovědný za prezentaci obsahu prostřednictvím uživatelského rozhraní. V ASP.NET Core MVC je použit tzv. *Razor Engine* pro vložení .NET kódu do značkovacího jazyka HTML. Veškerá logika uvnitř by se měla vztahovat k prezentaci obsahu.

3. `Kontrolér` – zpracovává interakci od uživatele, pracuje s modelem a nakonec vybírá pohled, který se má vykreslit.

V případě použití API volání se využije také MVC vzoru s tím rozdílem, že pohledy jsou vygenerovány ve formátu, který zadá uživatel, nejčastěji JSON. V takovémto případě se nepoužije zmíněný *Razor Engine*, ale pouze se výsledný objekt serializuje do potřebného formátu. Ve výsledné aplikaci jsou vytvořené kontroléry zvlášť pro jednotlivé API zdroje, např. `FilesController`, `RunsController` a `AlgorithmsController`. Pro prezentační část aplikace jsou kontroléry rozděleny podle logických částí, např. `AccountController`, `AdminController` a `ManageController`.

ASP.NET Core MVC není pouze implementace návrhového vzoru MVC, ale obsahuje velké množství knihoven ulehčující vývoj webové služby. Mezi hlavní části MVC frameworku patří směrování, navázání modelů, validace modelů, filtry a dependency injection.

Směrování

Před zpracováním dotazu musí být vybrán správný kontrolér a metoda v něm, jež se o zpracování postará. K tomu slouží směrování. Výběr probíhá na základě zadané URL adresy. V ASP.NET Core MVC existují dva způsoby směrování, které jdou kombinovat.

První způsob, ideální pro prezentační část webové služby, je směrování založené na *konvencích*. Směrování založené na konvencích umožňuje globálně definovat formáty URL adres, která aplikace přijímá, a jak každý z těch formátů mapuje na konkrétní metodu v kontroléru. Ve webové službě je použitý jediný formát: `"{controller}/{action}/{id?}"` Výsledné URL adresy se skládají z názvu kontroléru, názvu metody a případného identifikátoru, jenž je předán jako parametr metody.

Druhý způsob je směrování založené na atributech. Tento způsob umožňuje označit kontroléry a metody atributy, které definují URL adresu konkrétní metody. Tento způsob se používá pro API část webového serveru, protože dovoluje určit URL adresu tak, aby splňoval REST specifikaci. Příklad označení kontroléru `FilesController`: `[Route("Files")]`. Metoda, která zpracovává nahrávání souborů, je označena atributem: `[HttpPost("")]`, jenž kromě adresy určuje, pro kterou HTTP metodu se má metoda použít. Výsledná URL adresa pro zavolání zmíněné metody by potom vypadala takto: `~/files/`.

Filtry

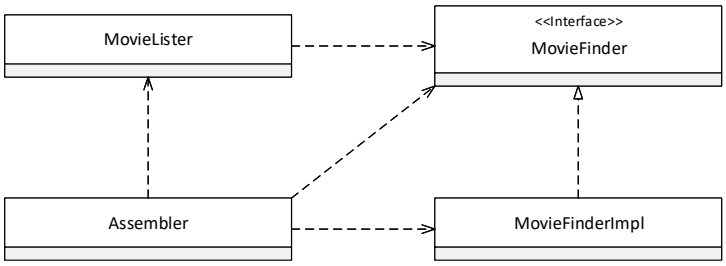
Filtry umožňují definovat logiku, která se provede před metodou a po metodě, nebo při vyhození výjimky. Filtry se dělí na globální a neglobální. Globální filtry jsou použity při každém HTTP volání. Neglobální jsou volány jen v těch místech, které jsou explicitně označeny atributem obsahující daný filtr.

Jeden z filtrů použitých v aplikaci je zavolán globálně při každé vyhozené výjimce. V aplikaci byl definovaný vlastní typ výjimek, které mají v sobě definovaný stavový kód. Při zachycení této výjimky je místo zahlášení interní chyby serveru (HTTP kód 500) vrácena odpověď obsahující HTTP stavový kód ve výjimce s případnou doplňující zprávou. To umožňuje například definovat výjimku `NotFoundException`, která obsahuje 404 stavový kód. Při nenalezení objektu požadovaného uživatelem je pouze vyhozena tato výjimka a poté vygenerována správná odpověď (obsahující 404 HTTP stavový kód).

4.1.3 Dependency Injection

Dependency Injection (dále také jen *DI*) je návrhový vzor sloužící ke snížení závislostí mezi jednotlivými částmi systému. Jeho provedení vychází z obecnějšího návrhového vzoru *Inversion of Controls* (IoC), jenž upřednostňuje kontrolu mimo objekt nad situací, kdy si objekt sám „říká“ o komponenty v rámci svého kódu. Pojem Dependency Injection poprvé představil Martin Fowler ve svém článku *Inversion of Control Containers and the Dependency Injection pattern* [5].

Základní myšlenkou DI je mít samostatný objekt, který se nazývá *assembler*, jenž poskytne třídě všechny potřebné závislosti [5]. Tyto závislosti jsou definovány rozhraním a assembler vybere vhodné objekty, které tato rozhraní implementují. Příklad tohoto principu lze vidět na obrázku 4.5.



Obrázek 4.5: Příklad závislostí při použití DI návrhového vzoru [5]. `MovieLister` ke svému chodu potřebuje objekt implementující rozhraní `IMovieFinder`. `Assembler` ví, kterou implementaci má použít, a při vytváření `MovieLister` poskytne objekt `MovieFinderImpl`.

Existují tři hlavní způsoby, jak se DI provádí [5]:

1. První způsob se nazývá *Constructor Injection*, a jak už z názvu vyplývá, třída definuje svoje závislosti skrze parametry v konstruktoru. Tato metoda dodržuje princip explicitních závislostí (anglicky *Explicit Dependencies Principle*), který vyžaduje, aby metody a třídy explicitně požadovaly veškeré závislosti potřebné ke správné funkčnosti.

2. Další metodou je *Setter Injection*, ta vyžaduje předání závislostí pomocí metod.

3. Poslední metoda ve zmíněném článku je *Interface Injection*, jež je podobná s předcházející metodou, ale názvy metod jsou zde definovány pomocí speciálního rozhraní.

IoC kontejner

Pro knihovny, které zajišťují funkčnost assembleru, se ustálilo slovní spojení *IoC kontejner*. ASP.NET Core obsahuje jednoduchý vestavěný kontejner, jenž ve výchozím stavu podporuje pouze Constructor Injection. Jelikož však tento jednoduchý kontejner nepodporuje pokročilejší techniky, mezi které patří registrace tříd pro návrhový vzor *dekorátor*, byl zvolen vyspělejší IoC kontejner s názvem *DryIoC*.

DryIoC se vyznačuje vysokou optimalizací, malou paměťovou náročností a v porovnání výkonu patří mezi nejvýkonnější kontejnery [3]. Podporuje pokročilejší scénáře konstrukce objektů a rozhraní knihovny je dobře zdokumentované.

```
public void ApiKeyManagerRegistration(IRegistrator registrator)
{
    registrator.Register<IApiKeyManager, ApiKeyManager>();
    registrator.Register<IApiKeyManager, CachedApiKeyManager>(
        setup: Setup.Decorator);
}
```

Ukázka 2: Ukázka kódu obsahuje registraci závislostí pro rozhraní `IApiKeyManager`. Toto rozhraní nabízí metody na získání API klíče uživatele a vygenerování nového API klíče. Prvně je zaregistrován `ApiKeyManager`, který obsahuje výchozí implementaci pro zmíněné rozhraní. Druhý je zaregistrovaný `CachedApiKeyManager` označený jako dekorátor, jenž rozšiřuje funkcionalitu optimalizací, která přidává ukládání a získávání API klíče z mezipaměti. Označení dekorátor zajistí, že při získání závislosti `IApiKeyManager` je nejprve vytvořen objekt `ApiKeyManager`, který je vložen do parametru konstruktoru třídy `CachedApiKeyManager`, ten je následně předán.

Před používáním IoC kontejnerů je nutné zaregistrovat všechny potřebné závislosti, viz ukázka 2. Nejjednodušší registrace může vypadat následovně:

```
registrator.Register<IService, SomeService>();
```

Zmíněná registrace říká IoC kontejneru ať kdykoliv, když bude potřebovat získat službu `IService`, vytvoří instanci `SomeService`. Po inicializaci pak stačí zavolat metodu:

```
container.Resolve<IService>();
```

V metodě `Resolve` kontejner vytvoří objekt `SomeService` se všemi jeho závislostmi a vrátí ho přes návratovou hodnotu.

Při registraci lze specifikovat podrobnější nastavení, mezi které patří hlavně opětovné použití už vytvořeného objektu. Ve výchozím stavu není vytvořený objekt znovu použit. U některých objektů může být výhodné vytvořit instanci objektu pouze jednou a tu znovu využívat (implementace návrhového vzoru *Singleton*). Poslední možností je opětovné využití vytvořených objektů v rámci jednoho webového dotazu. Speciální nastavení existuje pro použití návrhového vzoru dekorátor, viz ukázka 2. Podle návrhového vzoru dekorátor i dekorovaný objekt musí implementovat stejné rozhraní. Jelikož je v konstruktoru dekorátoru předán dekorovaný objekt, IoC kontejner musí vědět, že při vytváření dekorátoru nemá vytvořit znovu dekorátor, ale naopak dekorovaný objekt.

4.1.4 Autentizace a autorizace

Autentizace, autorizaci a správu uživatelů zajišťuje knihovna *ASP.NET Core Identity*, která je součástí frameworku. Nejdůležitější funkcionalitou je vytváření uživatelských účtů a přihlášení pomocí uživatelského jména a hesla nebo přes externí přihlašovací poskytovatele,

[3] Porovnání výkonu IoC kontejnerů pro platformy ASP.NET a ASP.NET Core:
http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison
a https://github.com/danielpalme/IocPerformance

jako jsou Facebook, Google, Microsoft, Twitter a další. Při použití lokálního účtu je heslo ve výchozím nastavení zašifrováno pomocí metody *HMAC-SHA256* [1]. Pro uložení uživatelských dat lze použít SQL databázi nebo implementovat vlastní perzistentní uložiště.

Knihovna řeší autorizaci různými způsoby [1]. Základním způsobem je použití rolí. Každý uživatel může mít přiřazen libovolný počet rolí. Role je reprezentována textovým řetězcem, např. finanční ředitel, správce sítě. K jednotlivým typům volání je specifikována množina rolí, které jsou oprávněny k použití. Dalším způsobem je autorizace založená na tvrzeních (anglicky claim-based). Tvrzení je dvojice název–hodnota, jež reprezentuje vlastnost uživatele, například datum narození. Tvrzení se většinou používají spolu s poslední technikou zásad (anglicky custom policy-based). Tato metoda kombinuje role a tvrzení a dovoluje vytvořit zásady, které se skládají z libovolného počtu povinných tvrzení (případně rolí) ve specifikovaném tvaru. Následně, stejně jako u rolí, stačí k volání specifikovat zásadu, která musí být splněna. Například zásada *18 a více let*, která se skládá z tvrzení obsahující datum narození a kontroly, že je uživateli více než 18 let.

Ve webové službě jsou používány pouze lokální účty. Uživatelská data jsou ukládána do SQL databáze. Pro autentizaci na stránkách je použita *cookie* autentizace. Pro API volání musel být použit odlišný mechanismus, který dovoluje jak manuální zadání vývojářem, tak použití statického klíče v aplikaci komunikující s webovou službou. Byla zvolena autentizace pomocí tzv. *API klíče*, která ověřuje validní hodnotu klíče v hlavičce HTTP dotazu. Cookie autentizace je považována za bezpečnější formu přihlášení, a proto lze pomocí API klíče používat pouze API volání, nelze měnit uživatelské heslo, email apod. API klíč musí být uložený v hlavičce každého dotazu a předpokládá se tedy, že bude uložený například ve skriptu, jenž provádí volání. Pro případného útočníka je jednodušší získat API klíč než kombinaci přihlašovacích údajů. V případě úniku API klíče je nutné, aby se uživatel přihlásil přes email a heslo a požádal o změnu API klíče.

```
Authorization: Simple W0OggbqByc3+Kf0gSjf6AuV/XvUhHd7TT4tzFya9JQE=
```

Ukázka 3: Ukázka hlavičky pro autentizaci pomocí API klíče. Pro API klíč bylo zvoleno autentizační schéma s názvem `Simple`, jelikož se jedná o jednoduchý způsob autentizace API volání.

Vygenerování nového API klíče má na starost třída `RndApiKeyGenerator`. V ní se nachází metoda `Generate`, která využívá `RNGCryptoServiceProvider` pro vygenerování 32 bajtů velkého klíče. Třídu `RNGCryptoServiceProvider` poskytuje framework .NET Core a nachází se ve jmenném prostoru `System.Security.Cryptography`. Slouží pro vygenerování sekvence kryptograficky silně náhodných hodnot. Z ukázky 3 si lze všimnout, že zobrazený API klíč obsahuje více než 32 znaků. HTTP protokol dovoluje do hlaviček vložit pouze ASCII znaky [4]. Z tohoto důvodu je uživateli zobrazen API klíč v podobě ASCII řetězce. K překonvertování se používá kódování *Base64* [4], které převádí binární data do posloupnosti ASCII znaků. Každé 3 bajty dat převede na 4 znaky, čímž také naroste délka řetězce oproti velikosti dat.

[4] RFC 4648 https://tools.ietf.org/html/rfc4648

4.1.5 Přístup k databázi

Pro přístup k databázi se ve webových aplikacích ustálilo používání objektově-relačních mapperů (dále jen *ORM*), které poskytují potřebnou abstrakci při používání databázových systémů. Vývojář nemusí psát specifické SQL dotazy pro danou odnož SQL jazyka, která je podporována v používaném systému řízení báze dat. ORM dovoluje vývojáři kdykoliv vyměnit například MySQL server za Microsoft SQL server (je nutné, aby daný ORM podporoval vybraný databázový systém).

Jelikož je framework ASP.NET Core relativně nový, není výběr ORM velký. Nejznámějším a nejpoužívanějším je Entity Framework Core (dále také *EF Core*) vyvíjený v rámci ASP.NET Core, ale jeho použití je volitelné. Další možností je použití knihovny Dapper [5]. Dapper se považuje za mikro ORM, což značí menší abstrakční vrstvu, a proto dosahuje většího výkonu. SQL dotazy nejsou vygenerovány knihovnou, ta pouze mapuje výsledky dotazů na objekty. Zvolena byla knihovna EF Core, hlavně díky rychlejšímu prototypování a možnosti kdykoliv změnit databázový server za jiný.

EF Core je odlehčená, rozšiřitelná a multiplatformní verze známější knihovny Entity Framework [13]. V tuto chvíli jsou podporovány následující databázové systémy [13]:

- Microsoft SQL Server,
- PostgreSQL,
- MySQL,
- Oracle a další.

Knihovny poskytující přístup k vybrané databázi (anglicky database providers) jsou vytvářeny jak přímo společnostmi spravujícími daný databázový systém, tak i neoficiálně jinými vývojáři. V případě MySQL lze vybrat ze tří dostupných knihoven, z nichž jedna je oficiální od společnosti Oracle.

V projektu byli použiti a otestováni dva databázoví poskytovatelé. Při testování webové služby byl použit Microsoft SQL Server, který má nejlepší podporu všech funkcí EF Core a nově je dostupný také na linuxových operačních systémech. Druhou testovanou databází byla PostgreSQL, jež patří mezi nejpoužívanější databáze dostupné zcela zdarma [6].

Při komunikaci s databází se používá *LINQ* (anglicky Language Integrated Query). LINQ je název pro sadu technologií určených na integraci dotazování přímo do jazyka C#. V ukázce 4 lze vidět použití technologie LINQ. Použití připomíná deklarativní zápis SQL jazyka. LINQ lze použít pro dotazování se na data v paměti, v XML, ale taky v SQL databázi, jak lze vidět ve zmíněné ukázce.

4.1.6 Docker

Docker je nástroj pro snadné vytváření, nasazení a spouštění libovolných aplikací s použitím kontejnerů [15]. Kontejner dovoluje vývojáři zabalit aplikaci se všemi potřebnými knihovnami a dalšími závislostmi a publikovat celou aplikaci jako jeden balík. Tímto způsobem si může být vývojář jistý, že na jakémkoliv jiném linuxovém stroji s rozdílným nastavením bude aplikace fungovat. Při použití pod operačním systémem Windows nebo Mac je nainstalován virtuální linuxový stroj a veškeré operace se dějí v něm [15].

[5] https://stackexchange.github.io/Dapper/
[6] Zdroj: https://db-engines.com/en/ranking

```
public File GetByHash(Byte[] hash, int userId)
{
    return Context.Files
        .Where(file => file.Hash == hash && file.UserId == userId)
        .FirstOrDefault();
}
```

```
SELECT TOP(1) [f].[Id], [f].[ContentType], [f].[CreatedAt],
↪  [f].[Extension], [f].[FileSize], [f].[Hash], [f].[LocationId],
↪  [f].[Type], [f].[UserId]
FROM [Files] AS [f]
WHERE ([f].[Hash] = @__hash_0) AND ([f].[UserId] = @__userId_1)
```

Ukázka 4: Horní ukázka obsahuje kód využívající EF Core k vytvoření dotazu na databázi. Dotaz slouží k získání souboru podle hashe a identifikátoru uživatele. Metoda `FirstOrDefault` zajistí získání pouze jednoho nebo žádného souboru. Spodní ukázka obsahuje SQL kód vygenerovaný knihovnou EF Core po provedení metody `GetByHash`. Vygenerovaný SQL kód využívá SQL parametry pro předání argumentů a tím zamezuje útoku *SQL Injection*. Metoda `GetByHash` je použita při kontrole, jestli se uživatel nesnaží nahrát soubor, který už nahrál dříve.

Docker slouží k podobným účelům jako virtuální stroj. Ale oproti virtuálnímu stroji Docker nevytváří celý operační systém pro danou aplikaci, ale běží na stejném hostujícím linuxovém jádře. To vede ke zvýšení výkonu a zmenšuje velikost nasazené verze aplikace.

Před spuštěním kontejneru s danou aplikací je potřeba vytvořit obraz (anglicky image), jenž slouží jako předpis pro vytvoření kontejneru. Zjednodušeně obraz obsahuje všechny potřebné soubory a parametry pro spuštění kontejneru. Stav obrazu se později nedá změnit, pouze vytvořit nový obraz. Pro vytvoření obrazu se používá *Dockerfile*. Dockerfile je textový soubor obsahující jednotlivé příkazy vedoucí k vytvoření obrazu. První instrukcí v souboru je `FROM` následovaná názvem obrazu, který je použit jako bázový obraz. Tímto způsobem lze vytvářet dědičnost obrazů. Pro ASP.NET Core projekty je už vytvořený základní obraz [7].

```
1 FROM microsoft/aspnetcore:1.1.1
2 ENTRYPOINT ["dotnet", "CVaS.Web.dll"]
3 ARG source=.
4 WORKDIR /app
5 EXPOSE 80
6 COPY $source .
```

Ukázka 5: Ukázka obsahu souboru Dockerfile pro webovou aplikaci. První řádek obsahuje název bázového obrazu. Poté následují další příkazy včetně příkazu, jenž určuje hlavní příkaz, který se provede po spuštění, anebo příkazu pro zkopírování potřebných souborů do obrazu.

[7] Zdrojový kód k obrazu: https://github.com/aspnet/aspnet-docker

Pro vytvoření obrazu z Dockerfile slouží příkaz `build`. Je nutné specifikovat cestu k složce, kde se nachází Dockerfile. V následujícím příkladu použití se Dockerfile nachází v současné složce. V příkladu je použit volitelný parametr `--tag`, pomocí kterého lze pojmenovat obraz. Pojmenování ve službě Docker se řídí formátem: `repositář/název-obrazu:verze`.

```
docker build --tag=adamjez1/cvas:latest .
```

Vytvoření a následné spuštění kontejnerů se provádí příkazem `run`. S příkazem musí být zadán název obrazu, z něhož se kontejner odvodí. V uvedeném příkladu je použitý volitelný parametr `-p`, kterým lze určit mapování síťových portů mezi kontejnerem a hostitelským operačním systémem.

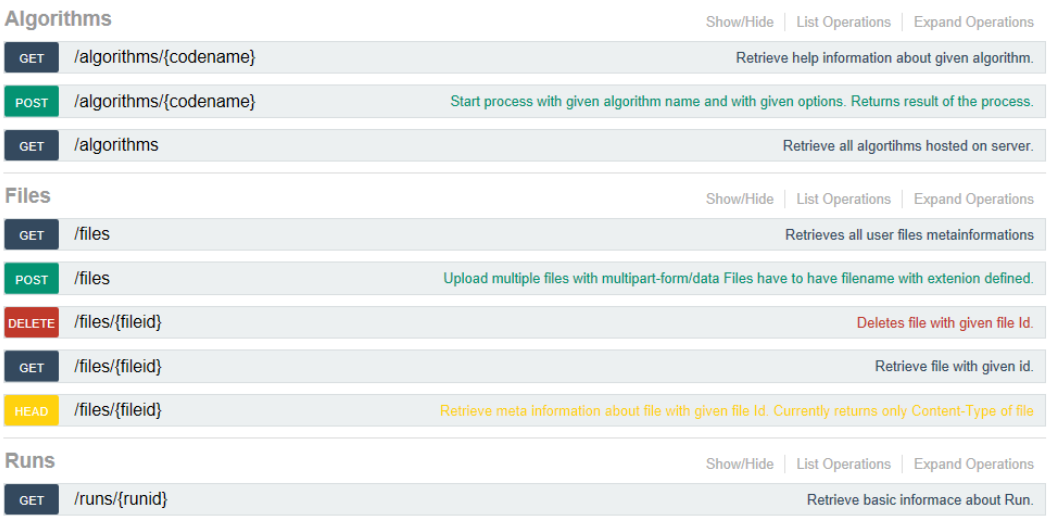
```
docker run -p 80:80 adamjez1/cvas:latest
```

4.1.7 Dokumentace API

Dokumentace jednotlivých API volání je důležitou součástí veřejného API. Pro snadný návrh a dokumentaci API existuje mnoho technologií. Mezi nejznámější patří framework *Swagger* a platforma *Apiary*. Zvolena byla technologie Swagger hlavně, díky velké podpoře různých nástrojů a knihoven.

Swagger je formální specifikace pro popis REST API služeb. Umožňuje jednoduše objevit a pochopit API webové služby bez nutnosti pročítaní dokumentace nebo čtení zdrojových kódů. Na Swagger je navázán ekosystém nástrojů, které umožňují prezentaci API, vygenerování serverové a klientské části aplikace a vytváření specifikací REST API.

Použití specifikace v ASP.NET Core projektech je snadné díky dostupnosti knihovny *Swashbuckle*[8], která na základě kódu vygeneruje danou specifikaci a vytvoří pro ni prezentaci. Způsob autentifikace nebo návratové typy jednotlivých API volání je nutné manuálně popsat v nastavení Swashbuckle.



Obrázek 4.6: Výsledná Swagger specifikace vygenerovaná pomocí knihovny Swashbuckle zobrazená jako HTML stránka

[8] https://github.com/domaindrivendev/Swashbuckle

Na obrázku 4.6 lze vidět výslednou vygenerovanou specifikaci pomocí knihovny Swashbuckle a její vizualizaci za použití Swagger UI. V detailu API volání jsou zobrazeny podrobnosti a lze si volání přímo z webového prohlížeče vyzkoušet. Pro vygenerování klientské části aplikace lze použít nástroj *Swagger CodeGen*[9]. Stačí vložit specifikace Swagger a zvolit výsledný jazyk a framework, který bude vygenerován. Vybrat si lze z mnoha možností, například: TypeScript+Angular2, Android, Ruby, Scala.

4.1.8 Knihovny pro komunikaci mezi servery

V minulé kapitole 3.2.2 byla popsána komunikace mezi servery z hlediska protokolů a software pro zprostředkování zpráv, v této části je diskutován výběr a použití knihoven. Kritérii pro výběr knihoven byly:

- dostupnost na platformě .NET Core,
- podpora protokolu AMQP 0-9-1 (protokol používaný ve vybraném zprostředkovateli RabbitMQ),
- jednoduché rozhraní a aktuální dokumentace.

Těmto požadavkům vyhovují pouze knihovny EasyNetQ a RawRabbit. Obě knihovny využívají oficiální knihovnu RabbitMQ.Client, která samotná nabízí pouze malou úroveň abstrakce a její použití není jednoduché. EasyNetQ i RawRabbit používají velice podobné rozhraní, ale EasyNetQ má několikanásobně větší uživatelskou základnu, a proto byl zvolen. Další známé knihovny MassTransit a Rebus nejsou v současné době dostupné na novější platformu .NET Core.

Použití knihovny EasyNetQ je velice jednoduché. Nejdříve je nutné vytvořit instanci objektu `IBus`, který si udržuje spojení s RabbitMQ serverem. Vytvoření probíhá pomocí metody `CreateBus`, viz 6. V argumentu metody je nutné zadat řetězec, který obsahuje informace potřebné k připojení RabbitMQ serveru. Kromě povinné informace o umístění serveru je možné zadat například parametry `prefetchcount`, viz 3.2.2 a `timeout` pro určení maximální doby čekání na odpověď. Vytvořená instance objektu `IBus` existuje po celý životní cyklus aplikace.

V případě, kdy chce webový server zaslat zprávu se spuštěním algoritmu, je nutné zavolat metodu `Send` na objektu `IBus`. Metoda na základě typu odesílané a přijímané zprávy vytvoří RabbitMQ frontu, na níž zašle serializovanou zprávu ve formátu JSON, a následně čeká na odpověď.

Server pro běh algoritmů musí zaregistrovat funkci, která zpracovává příchozí zprávu a vrací odpověď. K tomu slouží metoda `Respond` na objektu `IBus`, kde je opět nutné specifikovat typy zpráv. Metody `Send` a `Respond` využívají komunikační model typu dotaz/odpověď vysvětlený v minulé kapitole.

4.1.9 Práce s uložištěm souborů

V minulé kapitole 3.4.1 byly zmíněny důvody k použití samostatného severu pro uložiště souborů. Také byly uvedeny důvody pro výběr MongoDB jako uložiště souborů. Aby však nebyly webový server a server pro běh algoritmů napevno navázány na jedno konkrétní uložiště, byla vytvořena abstrakční vrstva pro práci s uložištěm, viz 7.

[9] Dostupné na https://github.com/swagger-api/swagger-codegen

```
bus = RabbitHutch.CreateBus("host=localhost;user=guest;password=guest");

public RunResultMessage Send(CreateAlgorithmMessage message)
{
  return bus.Request<CreateAlgorithmMessage, RunResultMessage>(message)
}

bus.Respond<CreateAlgorithmMessage, RunResultMessage>(request =>
{
  return messageProcessor.Process(request);
});
```

Ukázka 6: Ukázka obsahuje inicializaci objektu implementující rozhraní `IBus`, zaslání a přijímání zpráv na základě typů odesílané a přijímané zprávy.

```
public interface IFileStorage
{
    string Save(string filePath, string contentType);
    string Save(Stream stream, string fileName, string contentType);
    FileResult Get(string id);
    void Delete(string id);
}
```

Ukázka 7: Rozhraní pro zpracování manipulace se soubory na vzdáleném uložišti

Rozhraní `IFileStorage` bylo implementováno třídami `MongoDbFileStorage`, `FileSystemStorage` a `AzureBlobStorage`. Třída `FileSystemStorage` je určena pro testování, kdy jako uložiště je použit lokální souborový systém. Třída `AzureBlobStorage` byla implementována při nasazování služby na Azure, což je popsáno až v další sekci 4.7.2.

Integrace souborového uložiště MongoDb byla implementována skrze třídu `MongoDbFileStorage`. Ta využívá oficiální knihovnu *MongoDB Driver* [10]. V aktuální verzi knihovny 2.3 přibyla podpora frameworku .NET Core a všechna volání jsou nyní asynchronní. Každému nahranému souboru je přiřazeno tzv. `ObjectId`, to je následně uloženo spolu se záznamem do SQL databáze.

4.2 Server pro běh algoritmů

Server pro běh algoritmů byl implementovaný v jazyce C# na multiplatformním frameworku .NET Core. Na stejném frameworku běží i webový server s tím rozdílem, že webový server využívá navíc webový framework ASP.NET Core. Jelikož oba servery běží na stejném frameworku, mohou tak sdílet významnou část kódu.

Server pro běh algoritmů (dále jen server) využívá již zmíněné technologie: IoC kontejner DryIoC, objektově relační mapper EF Core, pro komunikaci mezi servery knihovnu EasyNetQ a stejné napojení na uložiště souborů, které bylo zmíněno v předcházející sekci.

[10] https://docs.mongodb.com/ecosystem/drivers/csharp/

Po inicializaci IoC kontejneru a spojení s RabbitMQ server čeká na přijetí zprávy z webového serveru. Před tím, než webový server zašle zprávu, vytvoří záznam běhu v databázi. Vytvořený záznam má označení: nedokončeno. Následující postup je prováděný pro každou přijatou zprávu:

1. Je přijata zpráva obsahující identifikátor záznamu běhu, argumenty a informace o algoritmu, který má být spuštěn.
2. Nejdříve je nutné získat z databáze entitu algoritmu.
3. Následně probíhá kontrola, zdali lokálně existuje spouštěný algoritmus.
4. Z předaných argumentů jsou vybrány ty, které se odkazují na uživatelský soubor. Probíhá kontrola jestli soubor není v dočasném lokálním uložišti. V případě, že není, je stažen z uložiště souborů.
5. Algoritmus s přeloženými argumenty je spuštěn a server čeká na jeho skončení.
6. Po jeho ukončení je aktualizován záznam s během algoritmu a případně nahrán soubor obsahující výsledek algoritmu.

Lokální uložiště uchovává soubory i po skončení algoritmu. Soubory jsou smazány až v případě, kdy není daný soubor delší dobu využit. K pravidelnému mazání souboru se využívá knihovna *FluentSheduler* [11]. Ta pomáhá s naplánováním pravidelných úkolů. V konfiguraci serveru lze nastavit, jaká doba musí uplynout od použití, aby byl soubor smazán. V konfiguraci je taky možné nastavit maximální velikost adresáře s dočasnými soubory. V případě, kdy je překročena maximální velikost, jsou smazány všechny soubory.

4.2.1 Spouštění programu

Ke spuštění programů (vytvoření procesu) se v .NET frameworku používá třída `Process`. Ta obsahuje metody `Start` ke spuštění procesu a `Kill` pro ukončení procesu. Nastavení parametrů spuštění se provádí pomocí třídy `ProcessStartInfo`. V ní je nutné zadat cestu k algoritmu a řetězec obsahující argumenty. Před samotným spuštěním procesu je přesměrován standardní výstup a standardní chybový výstup do nadřazeného procesu. Aby bylo možné přesměrovat standardní výstupy, je nutné nastavit vlastnost `UseShellExecute` na false. Vlastnost `UseShellExecute` ovlivňuje, v jakém módu bude nový proces spuštěný. Při ukončení procesu je předán návratový kód.

Při testování byl zjištěn rozdíl mezi linuxovými operačními systémy a Windows. Na linuxových systémech lze spouštět skripty (např. python, bash) jako spustitelné soubory pomocí *Shebang*. Shebang je syntaxe pro specifikaci interpretu na prvním řádku spouštěného skriptu např. `#!/usr/bin/python`. Operační systém Windows nic podobného nemá, a proto byl implementován rozdílný způsob pro specifikaci interpretu skriptu. Při zjištění, že server běží pod systémem Windows, je odekorována třída pro spouštění algoritmů třídou `WindowsProcessServiceDecorator`. Tato třída, na základě přípony spouštěného programu, spustí potřebný interpret a skript je předán jako argument. To se děje jenom pro programy s příponou, které byly uvedeny v konfiguraci serveru.

Při specifikování interpretu podle přípony v konfiguraci nastává problém při využívání více verzí interpretu pro stejnou příponu. Finálním řešením je použít pomocný skript, například Powershell, viz 8, který potřebný skript spustí se správným interpretem. Na pomocné

[11] https://github.com/fluentscheduler/FluentScheduler

```
& "C:/Python27/python" skript.py $args
exit $LastExitCode
```

Ukázka 8: Příklad Powershell skriptu pro spouštění skriptů s konkrétním interpretem.

skripty se vztahuje podmínka, že musí všechny používat stejnou verzi interpretu, kterou stačí zadat do konfigurace.

4.3 Algoritmy počítačového vidění

Většina zvolených algoritmů v sekci 3.5 byla implementována v prostředí Python 3.5.2 s použitím knihovny OpenCV. Jejich implementace ve většině případů spočívá v základním ošetření argumentů a zavoláním OpenCV funkcí. Dále jsou popsány složitější algoritmy.

4.3.1 Stabilizace obrazu ve videu

Pro stabilizaci obrazu videa byl použit open-source multimediální framework *FFmpeg*, který je distribuován pod licencí LGPL. Ta obsahuje modul *libavfilter*, jehož hlavní součástí jsou tzv. *filtry*. Filtry jsou efekty, které mohou transformovat nebo jinak zpracovávat vstupní data. Filtry lze řadit za sebe a vytvářet tím graf filtrů.

V první fázi je videosoubor analyzován filtrem (`videostabdetect`), který vygeneruje soubor obsahující relativní posun a rotaci mezi jednotlivými snímky. Druhá fáze využívá filtr (`vidstabtransform`), který pomocí informací v konfiguračním souboru kompenzuje pohyb aplikací transformací na vstupní video. Skript spouštějící program FFmpeg a ošetřující vstupní argument byl implementován v Powershell, což je skriptovací jazyk pro systém Windows. Jeho přepsání do linuxově kompatibilní verze (např. Bash) je triviální.

4.3.2 Rozeznání poznávacích značek

Rozeznávání SPZ bylo implementováno open-source knihovnou *OpenALPR* (Automatic License Plate Recognition) s licencí AGPLv3. Knihovna je napsaná v programovacím jazyce C++ s možným použitím v jazycích C#, Python, Go a Node.js. Knihovna pro zpracování využívá OpenCV a knihovnu Tesseract, která zajišťuje rozpoznání znaků. Výstupem analýzy obrázku nebo videa jsou nejpravděpodobnější textové reprezentace poznávací značky.

4.3.3 Zabarvení černobílé fotografie

K zabarvení černobílé fotografie byla použita konvoluční síť natrénovaná v práci *Colorful Image Colorization* [18]. Konvoluční síť byla natrénovaná na jednom miliónu fotografií z Imagenet datové sady. Ke zprovoznění algoritmu je nutné mít nainstalovanou knihovnu pro hluboké učení Caffe a další základní knihovny (numpy, pyplot, skimage, scipy) pro Python.

4.3.4 Optické rozpoznání znaků

Optické rozpoznání znaků (anglicky Optical Character Recognition, zkráceně OCR) je implementované knihovnou *Tesseract*. Knihovna je open-source, napsaná v C++ a podporuje

všechny nejpoužívanější operační systémy. Optické rozpoznání je zde založeno na neuronové síti. Pro spuštění byl vytvořen skript podobný již zmíněnému příkladu 8.

4.4 Frontend

V ASP.NET Core MVC se pro vytváření HTML stránek používá *Razor Engine*. Ten dovoluje vkládat C# kód do HTML stránek a umožňuje vytvářet hierarchickou strukturu mezi stránkami tak, aby základní rozložení stránky mohlo být definované v jediném souboru a vzhled jednotlivých stránek v samotných souborech. Takto vytvořené soubory mají příponu `.cshtml`.

Pro vytváření samostatných celků slouží v Razor Engine tzv. pohledové komponenty (anglicky *view components*) [7]. Jejich provedení je zcela oddělené od zbytku stránky. Pohledové komponenty byly použity pro zobrazení navigačního menu a statusu webové služby. Takto vytvořené pohledové komponenty lze použít HTML syntaxí: `<vc:web-service-status />`. Cílem této syntaxe je, aby co nejvíce připomínala použití HTML entit.

4.4.1 Použité knihovny

Při vývoji byly použity JS/CSS frameworky *Boostrap* a *jQuery*, které pomáhají při vytváření responzivního designu a ulehčují práci s objektovým HTML modelem (známý jako DOM). Pro framework Bootstrap nebyl použit výchozí styl, nýbrž styl s názvem *Flatly* pro plochý a moderní vzhled. K usnadnění kopírování API klíče ze stránky byla využita knihovna *clipboard*, která dovoluje vkládání řetězce do schránky.

Knihovny na stránkách s algoritmy

Na stránkách, jež zobrazují a dovolují spustit algoritmy, byly použity následující knihovny:

- Pro nahrání souborů byla využita javascriptová knihovna *fine-uploader*, která nabízí validaci vstupních souborů, tzv. metodu Drag&Drop pro nahrání souboru a další vylepšení.
- Po provedení algoritmu některé algoritmy vytvoří zabalený soubor (*.zip), ve kterém se nachází výsledek algoritmu (např. zabarvená černobílá fotografie). Nejdříve je nutné rozbalit výsledný soubor, což se děje pomocí knihovny *zipjs* [12]. V případě, kdy zabalený soubor obsahuje obrázek, je zobrazen uživateli.
- Pro případné zobrazení obrázku získaného spuštěním algoritmu je použita knihovna *featherlight*. Tato knihovna umožňuje vytvoření modálního okna s obrázkem.

Správa balíčků a minifikace

Pro správu frontend frameworků, knihoven, stylů, zkráceně balíčků byl použit správce balíčku *Bower*. Bower na základě konfiguračního souboru `bower.json`, ve kterém jsou uloženy názvy a verze balíčků, stáhne a nakopíruje požadované soubory na zvolené místo v souborovém systému.

K zabalení a minifikaci vlastních javascript skriptů a css stylů je použita knihovna s názvem *Bundler and Minifier* [13]. Konfigurace se provádí pomocí souboru `bundleconfig.json`.

[12] https://gildas-lormeau.github.io/zip.js/
[13] https://github.com/madskristensen/BundlerMinifier

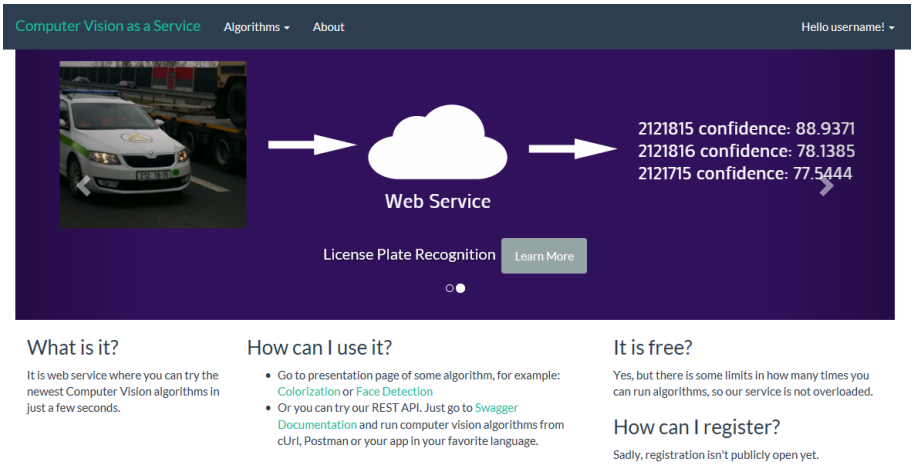
Následně zadáním příkazu `dotnet bundle` nebo při spuštění buildu webového projektu se provede minifikaci a zabalení skriptů a stylů.

4.5 Výsledná webová služba

Služba byla vytvořena podle návrhu z předešlé kapitoly 3. Ze sekce 3.7.3, která obsahovala popis jednotlivých HTTP volání, byly všechny implementovány. Na vygenerované stránce knihovnou SwashBuckle, jež je na obrázku 4.6, lze vidět dokumentaci k daným voláním.

V oblasti webového serveru byly vytvořeny základní stránky určené k prezentaci a demonstraci funkčnosti služby. Patří mezi ně:

- úvodní stránka s představením služby 4.7,
- přihlašovací a registrační stránka,
- stránka zajišťující správu uživatelského účtu, kde si uživatel může změnit heslo a resetovat API klíč,
- prezentační stránky pro 9 algoritmů, které obsahují stručný popis algoritmu a možnost jeho spuštění s různými parametry, jsou podrobněji popsány dále v 4.5.1,
- administrační sekce obsahující statistiky používání a stav webové služby.



Obrázek 4.7: Úvodní stránka webové služby

4.5.1 Stránky pro prezentaci algoritmů

Všechny stránky určené pro prezentaci algoritmů sdílejí stejné rozložení, které je uložené v souboru `_AlgorithmLayout.cshtml`. Přidání prezentační stránky pro algoritmus se skládá pouze z vytvoření vlastní stránky, např. `Colorization.cshtml`. Název stránky musí být shodný s tzv. `code-name` algoritmu, jenž je uložen v databázi.

Pro snadné vytváření prezentačních stránek byl vytvořen skript `algorithm-client.js`, který obstarává veškerou asynchronní komunikaci s webovým serverem. Pro komunikaci se používá technologie *AJAX* a využívají se stejné API volání, jaké mají k dispozici uživatelé.

Jediný rozdíl je ten, že pro autentizaci se používají cookies, tzn. s každým dotazem je zaslaná cookie získána při přihlášení.

Některé algoritmy mají možnost upravení parametrů spuštění. K tomu slouží formulář nad částí pro nahrání souboru, viz 4.8b. Upravené parametry budou použity při každém dalším spuštění algoritmu. Pro spuštění algoritmu musí uživatel zvolit nahrání souboru, buď pomocí dialogového okna, nebo metodou drag&drop. V tu chvíli je soubor nahrán pomocí technologie AJAX na webový server. Po úspěšném nahrání souboru je spuštěn algoritmus. Parametry spuštění algoritmu jsou v modálním okně zobrazeny uživateli, aby věděl, jak provést dané API volání ručně. Po skončení algoritmu je zobrazen uživateli výsledek. V případě, že výsledek obsahuje také soubor a jedná se o obrázek, je obrázek zobrazen uživateli s možností zvětšení přes celé okno.

4.5.2 Příklady API volání

Před spuštěním algoritmu přes API volání je nutné nahrát soubor, případně použít už nahraný soubor. Následující příklady využívají utilitu příkazové řádky *curl*. Všechny API volání musí obsahovat API klíč pro autentizaci uživatele viz `API_KEY`.

Následuje příklad nahrání souboru `/home/user/test.jpg`.

```
curl -X POST -F "file=@/home/user/test.jpg" \
     -H 'Authorization: Simple API_KEY' \
     http://localhost:5000/files
```

Ve výchozím nastavení bude vygenerovaná odpověď ve formátu JSON. Obsahuje identifikátory pro všechny nahrané soubory, v tomto případě jeden identifikátor.

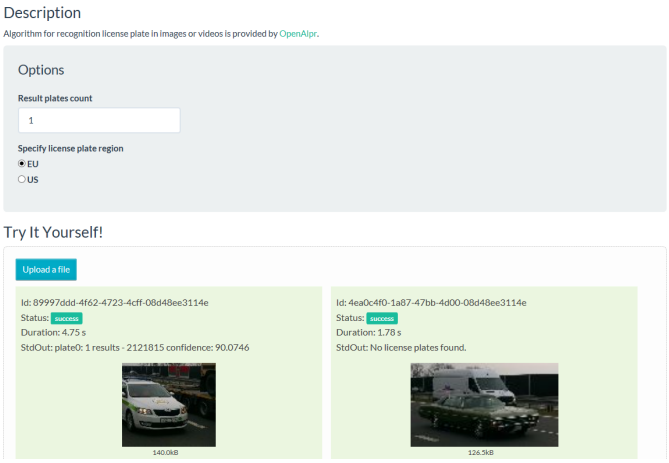
```
[
  { "id": "51e38431-0034-4488-62a3-08d48efa9952", "fileName": "test.jpg" }
]
```

Po úspěšném nahrání souboru je spuštěn algoritmus pro obarvení černobílé fotografie.

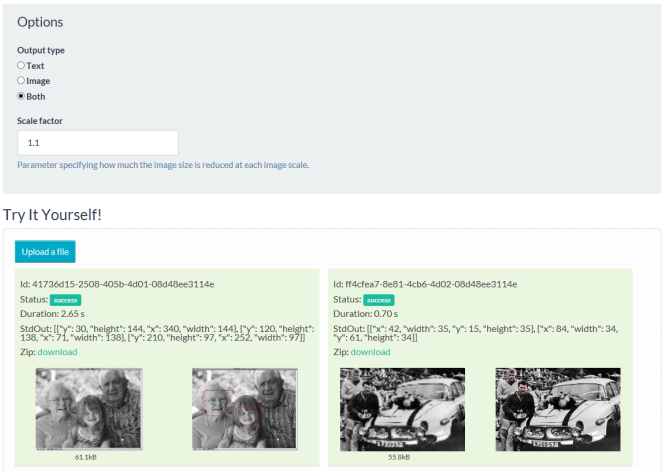
```
curl -X POST -d '["local://b8e0a392-315d-4453-aa2e-08d48eef95e9"]' \
     -H 'Content-Type: application/json' \
     -H 'Authorization: Simple API_KEY' \
     http://localhost:5000/algorithms/Colorization
```

Odpověď obsahuje identifikátor běhu, status, délku trvání atd. Jelikož algoritmus vygeneroval alespoň jeden soubor, odpověď obsahuje URL adresu archivu obsahující výsledné soubory. V tomto případě archiv obsahuje obarvenou fotografii.

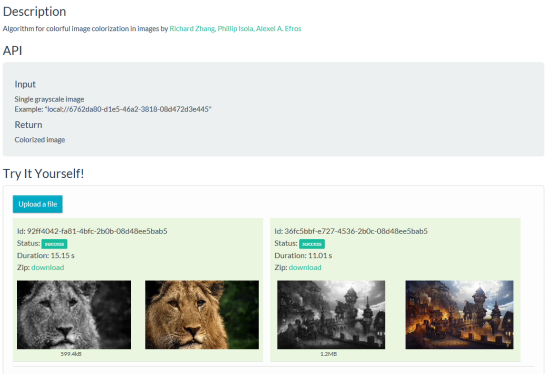
```
{
  "id": "e94645c3-c59f-49f1-1cfc-08d48eef962e",
  "stdOut": "",
  "stdErr": "",
  "status": "success",
  "createdAt": "2017-04-29T11:41:54.9741Z",
  "finishedAt": "2017-04-29T11:42:08.4226789Z",
  "duration": 13448.5789,
  "zip": "http://localhost:5000/files/ef669290-e713-4dcc-aa2f-08d48eef95e9"
}
```



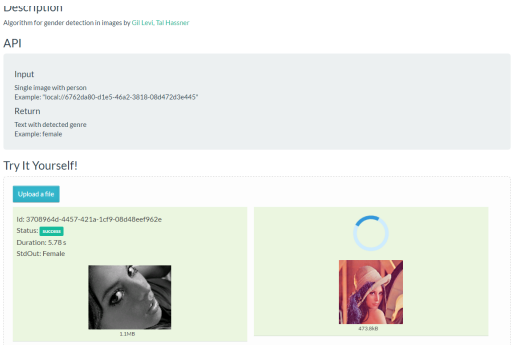
(a) Rozpoznání SPZ



(b) Detekce obličejů



(c) Zabarvení černobílé fotografie



(d) Detekce pohlaví osoby

Obrázek 4.8: Prezentační stránky algoritmů s možností úpravy parametrů viz 4.8a a 4.8b. Dále stránky obsahují stručný popis algoritmu společně se zdroji. Na obrázku 4.8d lze vidět proces čekání na doběhnutí algoritmu.

4.5.3 Časové limity

Podle návrhu v kapitole 3.3 nejsou vstupní argumenty algoritmů nijak validovány a kontrola nad algoritmy je minimální. Pro případy, kdy by byly algoritmy spouštěny nad velkým množstvím dat a jejich běh by přetěžoval server pro běh algoritmů, byl představen tzv. *tvrdý* časový limit.

Naopak druhý časový limit, tzv. *měkký*, byl zaveden z důvodu odlehčení webovému serveru, aby server nemusel udržovat hodně aktivních HTTP spojení.

Oba zmíněné časové limity jsou měřeny od spuštění programu provádějícího algoritmus.

- Po vypršení měkkého časového limitu je dotaz přerušen a je zaslána HTTP odpověď. Odpověď informuje uživatele o průběhu algoritmu a vrací identifikátor spuštění. Pomocí identifikátoru se může uživatel později dotázat, jestli program už doběhl, případně jaký je jeho výsledek. Tento interval byl přidán proto, aby měl uživatel co nejdříve odpověď, že algoritmus probíhá, a nedomníval se, že je server zahlcen.

- Druhý, tvrdý časový limit, řeší případy, kdy se program zacyklí nebo je spuštěn nad příliš velkým objemem dat a jeho průběh příliš zatěžuje server. Po uplynutí tohoto intervalu je program ukončen a zapsán jako neukončený do databáze.

Časové limity lze měnit v konfiguraci služby, případně měkký časový limit může uživatel API měnit zasláním nepovinného parametru při spuštění algoritmu. Uživatel může zadat hodnotu měkkého limitu v rozmezí 0 až hodnota tvrdého limitu. V případě hodnoty mimo interval je hodnota saturována do limitu. Použití lze najít v případech, kdy uživatele ihned po spuštění algoritmu nezajímá jeho výsledek a kontrola výsledku je provedena někdy v budoucnu. V takovém případě uživatel zadá hodnotu 0, která znamená odpověď webového serveru ihned po spuštění algoritmu.

4.6 Klientská API knihovna

Pro co nejjednodušší použití byla vytvořena klientská knihovna. Klientská knihovna ulehčuje používání webové služby. Uživatel této knihovny se nemusí starat o vytváření HTTP dotazů a parsování odpovědí.

Klientská knihovna byla implementována v programovacím jazyce Python verze 3.4. Python byl vybrán, protože v počítačovém vidění zaujímá významnou roli, což je dáno dostupností mnoha knihoven (OpenCV, Caffe, SciPy, scikit-learn) a jednoduchým prototypováním.

Klientská knihovna má název `cvas` (Computer Vision as a Service). Pro vytváření HTTP dotazů využívá knihovnu `requests` a pro zpracování formátu JSON stejnojmennou knihovnu `json`. K vytvoření a distribuci Python balíčku je použita knihovna `Setuptools`[14]. Funkcionalita knihovny je demonstrována na ukázce 9.

4.7 Testování

Nedílnou součástí jakéhokoliv vývoje je průběžné testování. Pro složitější části kódu byly vytvořeny unit testy, viz sekce 4.7.1. Režijní nároky webové služby byly měřeny výkonnostními testy 4.7.2. Výsledné nasazení do produkčního prostředí je popsáno v sekci 4.7.2.

[14] https://github.com/pypa/setuptools

```
import cvas

client = cvas.client("zVes/1008G1YhtKNiNa0WY5ZGubLK+DYYu0g+e1hcmQ=",
↪  "http://cvas.azurewebsites.net")

file = client.upload_file("~/images/car1.jpg")
algorithm = client.algorithm("license-plate-recognition")
result = algorithm.run([{ "c" : "eu", "n": "2"}, file])

print(result.status)
```

Ukázka 9: Ukázka použití klientské knihovny pro programovací jazyk Python verze 3. Ukázka obsahuje vytvoření klienta, který uchovává API klíč a URL adresu vzdáleného serveru. Poté je na server nahrán lokální soubor. Následně je spuštěn algoritmus s označením *license-plate-recognition* s parametry ve formě objektu. Na posledním řádku je vytištění statusu běhu algoritmu na standardní výstup.

Uživatelské testování nebylo provedeno, jelikož hlavní částí práce nebylo uživatelské rozhraní webové prezentace, nýbrž architektura webové služby.

Vývoj s testováním probíhal na operačních systémech Windows 10 a Ubuntu 16.04. Na zmíněných systémech byl testován, jak webový server tak server pro běh algoritmů. K vývoji bylo použito IDE Visual Studio, které je pouze pro Windows, a editor Visual Studio Code, který je multiplatformní.

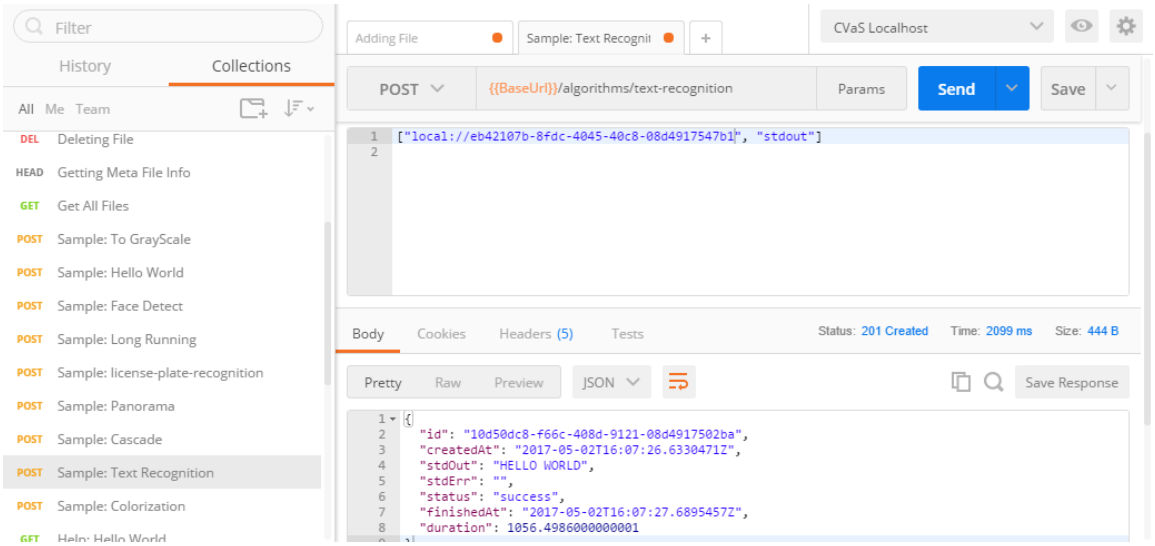
Ke zkoušení a ověřování funkčnosti API volání byl použit software *Postman*. Ten umí ukládat jednotlivá API volání do kolekcí a případně je sdílet mezi vývojáři. API volání nemusí být ručně vytvořena, lze je importovat použitím konfiguračního souboru Swagger vygenerovaného knihovnou Swashbuckle 4.1.7. Pro jednoduché použití napříč různými prostředími (vývojové, produkční) nabízí Postman správu proměnných. Hodnoty proměnných, například API klíč nebo název domény, jsou uloženy v konfiguraci prostředí. Při zasílání HTTP dotazu jsou použity hodnoty z aktivní konfigurace prostředí.

Prezentační část webového serveru byla testována v nejpoužívanějších prohlížečích: Google Chrome, Firefox, Microsoft Edge. Rychlost odezvy jednotlivých stránek byla měřena pomocí knihovny *Miniprofiler* [15]. Tato knihovna měří trvání části kódu, která obsluhuje každý HTTP dotaz, přímo na webovém serveru. Speciální úseky kódu, mezi které patří SQL dotazy, renderování HTML a uživatelem označené úseky kódu, jsou měřeny zvlášť. Díky těmto informacím lze zjistit příčina dlouho trvajících HTTP dotazů.

4.7.1 Unit testy

Unit test je automatizovaný kus kódu zaměřený na ověření funkčnosti a korektnosti malé části systému. Malá část systému může představovat metodu, třídu, několik tříd [6]. Unit testy jsou napsány programátory a jejich spuštění by mělo být výrazně rychlejší než u jiných druhů testů, což umožňuje testy spouštět často. Technika unit testování je jednou z klíčových součástí vývojových metod extrémního programování a programování řízeného testy.

[15] http://miniprofiler.com/



Obrázek 4.9: Snímek z aplikace Postman pro vytváření, zasílání a správu HTTP dotazů. Na snímku lze vidět zaslaný API dotaz pro spuštění algoritmu na optické rozpoznání znaků.

Pro unit testy byl vytvořen projekt s názvem `UnitTests`, který obsahuje 40 testů. Testy pokrývají část zpracovávající překlad argumentů a ukládání objektů do mezipaměti. Projekt využívá open-source nástroj *xUnit.net* [16].

```
[Fact]
public void JsonParser_TooComplexObjectParse_ExceptionExpected()
{
    dynamic jObject = new JObject();
    jObject.Array = new JArray(new[] { 1, 2, 3 });

    Assert.Throws<ArgumentException>(() => _parser.Parse(jObject));
}
```

Ukázka 10: Ukázka unit testu, který kontroluje vyhození výjimky při zadání špatné struktury vstupních argumentů.

4.7.2 Výkonnostní testy

Výkonnostní testy byly vytvořeny pro zjištění režijních nároků webové služby při spouštění algoritmů počítačového vidění. Dalším důvodem pro provádění testů bylo otestování vytvořené architektury s odděleným serverem pro běh algoritmů. V rámci testů bylo vytvořeno testovací prostředí pomocí služby *Microsoft Azure*.

Testovací prostředí

Testovací prostředí bylo vytvořeno v cloudové službě Microsoft Azure zejména pomocí virtuálních strojů. Cloudová služba byla použita, protože pro otestování webové služby je

[16] https://xunit.github.io/

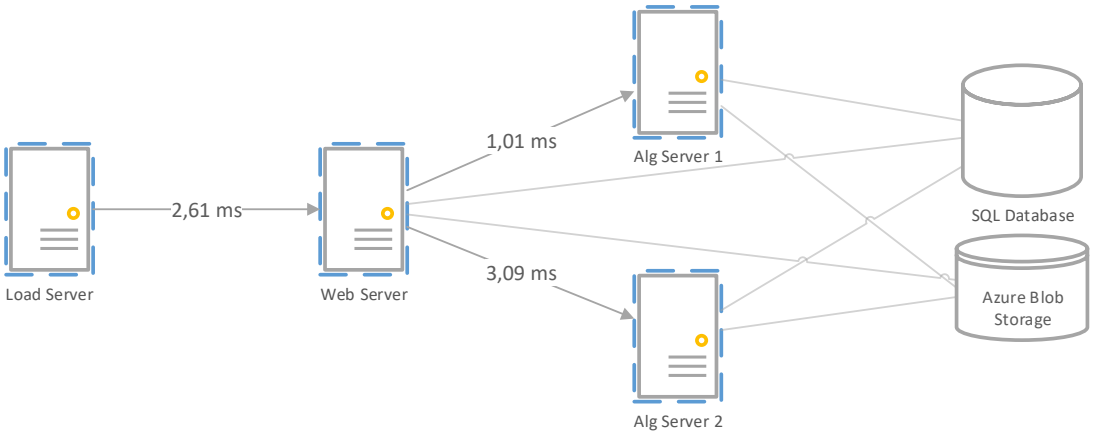
potřeba větší množství samostatných serverů. Aby byly výsledky testů relevantní, výkonnost serverů musí být porovnatelná s výkonností při reálném nasazení. Celkem byly použity čtyři virtuální stroje se stejnou hardwarovou a softwarovou konfigurací, viz tabulka 4.1.

Operační systém	Ubuntu Server 16.04 LTS
Procesor	Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz
Počet jader / vláken na jádro	2 / 1
Velikost operační paměti	7 GB
Pevný disk	SSD 14 GB

Tabulka 4.1: Hardwarová a softwarová specifikace použitých virtuálních strojů na službě Microsoft Azure.

Kromě virtuálních strojů byla využita Azure SQL databáze a *Azure Blob Storage*. Azure Blob Storage je použit namísto MongoDB jako uložiště uživatelských souborů. Oproti MongoDB nabízí ve službě Azure lepší poměr cena versus velikost uložiště a zprovoznění je výrazně jednodušší. Pro SQL databázi byla zvolená cenová úroveň *standard* (existuje ještě jedna základnější cenová úroveň *basic*).

Na diagramu 4.10 je znázorněné propojení virtuálních strojů společně s průměrnou odezvou mezi servery. Odezva byla měřena s použitím utility *Ping*, která využívá servisní ICMP protokol. `Load Server` slouží pro zasílání dotazů a vytváření zátěže směrem k webovému serveru. `Web Server` obsahuje kromě samotné webové aplikace (ASP.NET Core MVC aplikace) také RabbitMQ server pro rozesílání zpráv mezi servery pro běh algoritmů. `Alg Server N` reprezentují servery pro běh algoritmů, které jsou připojené k RabbitMQ serveru.



Obrázek 4.10: Testovací prostředí vytvořené ve službě Microsoft Azure. Jednotlivé servery byly na službě vytvořené pomocí virtuálních strojů, které všechny měly stejnou konfiguraci.

Rychlost odezvy

Důležitým parametrem výkonnosti služby je rychlost odezvy (angl. latence). Měřena byla pouze rychlost odezvy dotazů pro spuštění algoritmů, které jsou jádrem služby.

Parametry testování byly zvoleny s ohledem na zjištění režijních nároků služby. Měření probíhalo pouze s jedním serverem pro běh algoritmů a bez žádné zátěže. V rámci měření

nebyl spouštěn reálný algoritmus, ale Bash skript, který byl prázdný a nic neprováděl. Při všech měřeních byly záznamy z databáze pro API klíč a pro spouštěný algoritmus v mezipaměti. Všechny dotazy obsahovaly následující argumenty:

```
["Hello World", {"file": "local://a9a5c3fb-ed6a-4a27-513f-08d4946e624c"}]
```

Byly vytvořeny dvě testovací sady po 22 měřeních. Jedna sada měřila rychlost odezvy v případě, kdy je soubor, který se nachází v argumentech spuštění, uložen v mezipaměti a druhá sada měřila případ, kdy je potřeba soubor stáhnout lokálně z Azure Blob Storage. Velikost uvedeného souboru je 1,24 MB.

Během měření bylo zjištěno, že i v případě, kdy je spouštěn prázdný Bash skript, trvá provedení skriptu přibližně 100 ms. Avšak při spuštění Bash skriptu z konzole a změřením unixovou utilitou *time* bylo trvání přibližně 1 ms. Při investigaci tohoto chování byla nalezena implementace třídy `Process` v .NET Core. Implementace pro unixové systémy kontrolovala ukončení procesu každých 100 ms, což způsobovalo velkou odezvu i pro velmi krátké procesy. Toto chování bylo s pomocí reportováno a následně opraveno [17]. Ale jelikož oprava bude až v rámci příští verze .NET Core (verze 2.0), byl interval běhu skriptu odečten od naměřených hodnot.

Dotazy byly vytvářeny a měřeny z tzv. *Load* serveru pomocí unixové utility *curl*. Hodnoty byly zaznamenávány dvěma způsoby: samotnou utilitou a knihovnou Miniprofiler, jež měřila délku zpracování na samotném webovém serveru.

	Způsob měření	Průměr	Směrodatná odchylka
Bez souboru v mezipaměti	cUrl	**235,12 ms**	37,90 ms
	Miniprofiler	224,01 ms	36,87 ms
Se souborem v mezipaměti	cUrl	**45,88 ms**	6,07 ms
	Miniprofiler	38,51 ms	6,16 ms

Tabulka 4.2: Průměrná rychlost odezvy pro dotazy, v rámci kterých jsou spuštěny algoritmy počítačového vidění.

V tabulce 4.2 je uvedený průměr a směrodatná odchylka časů pro dotazy na dvě testovací sady. Z výsledných hodnot lze říci, že největším podílem na režijních nárocích služby je dopravení souboru na server, kde poběží algoritmus. Když je soubor ponechán v mezipaměti na serveru, jsou režijní nároky, vzhledem k délce trvání algoritmů počítačového vidění, marginální.

Počet dotazů za sekundu

Jako další parametr výkonnosti serveru byl zvolen počet zpracovaných dotazů za sekundu. Účelem testů bylo názorně ukázat výhody architektury s odděleným serverem pro běh algoritmů. Měření probíhalo pomocí nástroje *wrk* [18], který generuje HTTP dotazy v takové míře, aby server stíhal odpovídat. Příklad spuštění nástroje wrk při měření lze vidět v ukázce 11.

Pro co nejpřesnější simulaci zátěže byl místo algoritmu počítačového vidění spouštěn skript pro výpočet n-tého prvočísla. Argumentem spuštění bylo pořadí prvočísla, čím větší pořadí, tím větší zátěž. Skript byl implementován v jazyce Python, verze 2.7. Před samotným testováním bylo zjištěno, jakou dobu v průměru zabere výpočet zvoleného pořadí na

[17] Nareportovaný problém https://github.com/dotnet/corefx/issues/19470
[18] https://github.com/wg/wrk

```
wrk -d 30 --timeout 30 -t 2 -c 32 -s stress-test.lua
↪  http://{WEB_SERVER_IP}:{WEB_SERVER_PORT}/algorithms/prime-number
```

Ukázka 11: Ukázka spuštění nástroje *wrk* pro měření výkonnosti HTTP serverů. S těmito argumenty byla spuštěna veškerá měření. Spuštění se lišila pouze použitým skriptem, který obsahuje HTTP argumenty dotazu. Argument `-d` specifikuje délku trvání testu, parametr `-t` počet použitých vláken pro generování dotazů, parametr `-c` maximální počet otevřených TCP spojení. Hodnoty parametrů byly odpozorovány jako nejvhodnější.

použitých virtuálních strojích. Měření bylo opakované 5krát s čtyřmi různými argumenty (konkrétně 370, 550, 840 a 1150). Testované byly dva scénáře: pouze s jedním serverem pro běh algoritmů a se dvěma servery. Jednotlivé servery pro běh algoritmů byly připojeny k RabbitMQ serveru s parametrem `prefetch_count` s hodnotou 4 (parametr, viz sekce 3.2.2), která se jevila jako nejvhodnější. Hodnota parametru dovoluje pouze čtyři souběžně spuštěné algoritmy na jednom serveru (v případě jiné hardwarové konfigurace by se hodnota lišila).

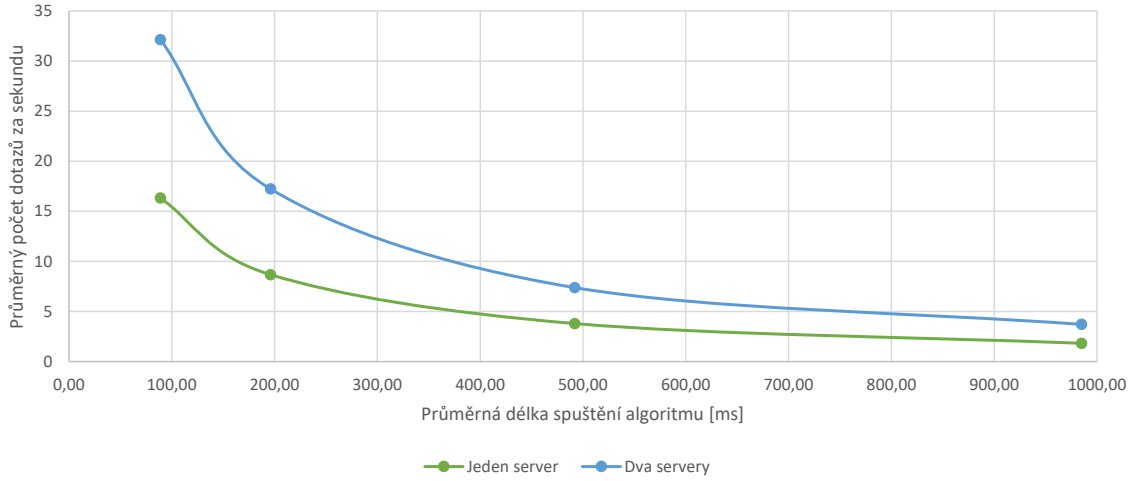
Z výsledků měření zobrazených v grafech viz 4.11 lze konstatovat výhody zvolené architektury jako prokázané. V případě použití dvou serverů pro běh algoritmů místo jednoho je počet dotazů zpracovaných za sekundu přibližně dvojnásobný. Stejně tak rychlost odezvy je téměř dvakrát menší. Tímto způsobem lze přidáváním dalších serverů zvětšovat výkonnost webové služby. Pro srovnání bylo měření pomocí nástroje wrk také provedeno na úvodní webovou stránku s požadavkem HTTP GET s výsledky: **645** dotazů za sekundu s průměrnou rychlostí odezvy **50,89 ms**.

4.8 Směr dalšího vývoje

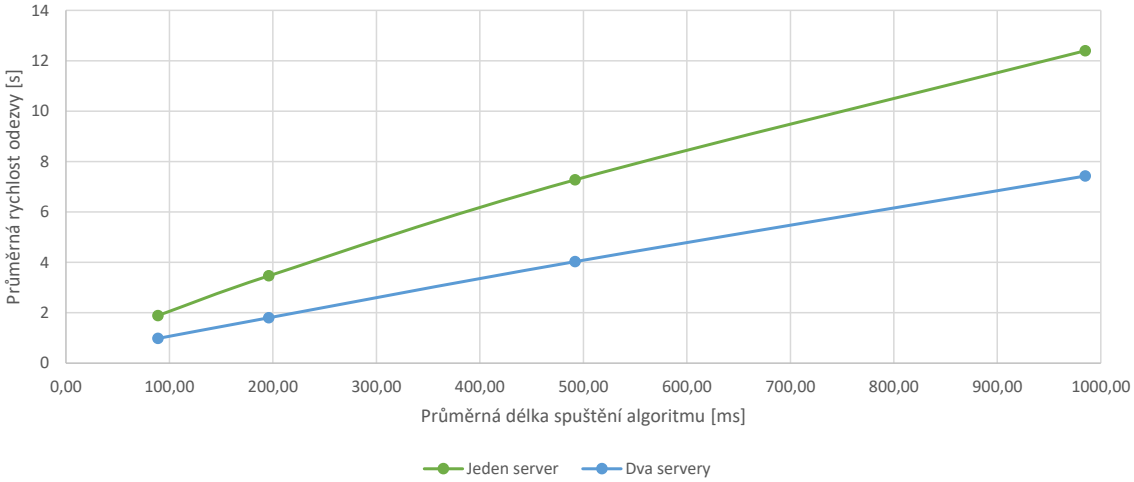
Současný stav webové služby je plně použitelný. Umožňuje nasazení na vlastní servery s možností použití různých SQL databází, uložišť pro soubory i server pro běh algoritmů v rámci webového serveru (implementováno v rámci semestrální práce, možnost byla ponechána).

Pro nasazení jako komerčního řešení by bylo potřeba doimplementovat několik funkcí. Mezi nejdůležitější funkce patří přidání limitu pro používaní služby. Například limit omezující počet spuštěných algoritmů za časovou jednotku. Bez limitu může uživatel se zlými úmysly spustit velké množství algoritmů zároveň, a tím učinit službu nepoužitelnou pro ostatní (rychlost odezvy služby by byla degradována). Další limit by byl pro maximální velikost nahraných souborů na uložiště, případně implementovat automatické mazání nejdávněji použitých souborů. Aktuálně služba nabízí pouze omezené množství algoritmů počítačového vidění, které by bylo potřeba výrazně zvětšit. Jedním ze způsobů zvětšení počtu algoritmů by mohlo být povolit uživatelům nahrání vlastních algoritmů. V takovém případě by bylo potřeba vytvořit tzv. *sandbox* prostředí pro tyto algoritmy tak, aby nemohly ovlivnit nebo nabourat chod celého serveru pro běh algoritmů.

Pro efektivní využití serverů pro běh algoritmů by bylo možné použít orchestrační nástroj, který by například podle aktuálního vytížení spouštěl nebo vypínal servery. K snazšímu využití API z různých programovacích jazyků by bylo potřeba nabídnout další klientské knihovny.



(a) Průměrný počet zpracovaných dotazů za sekundu



(b) Průměrná rychlost odpovědi pro jeden dotaz při daném počtu dotazů za sekundu viz 4.11a

Obrázek 4.11: Grafy zobrazují závislost náročnosti spouštěných algoritmů (náročnost vzhledem k potřebným zdrojům) na počet dotazů za sekundu, viz 4.11a, resp. rychlosti odezvy, viz 4.11b.

Kapitola 5

Závěr

Tato práce představuje návrh a realizaci webové služby zaměřené na zjednodušení přístupu k algoritmům počítačového vidění včetně podpory autentizace uživatelů, předávání dat a několika algoritmů počítačového vidění.

Zjištěny byly hlavní požadavky, mezi které patří zpřístupnění algoritmů počítačového vidění pomocí webové služby použitím jak webových stránek, tak aplikačního rozhraní a snadné napojení již existujících implementací algoritmů.

Navržená služba využívá architekturu s dedikovaným serverem pro běh algoritmů, jež umožňuje škálování služby podle aktuální zátěže, a obsahuje samostatné uložiště souborů. Pro aplikační rozhraní služby je použita architektura REST. Hlavním principem spouštění programů na serveru pro běh algoritmů je překlad vstupních parametrů na argumenty příkazové řádky. Nejdůležitějšími zvolenými technologiemi jsou webový framework ASP.NET Core a zprostředkovatel zpráv RabbitMQ.

Realizace služby se skládala zejména z implementace webového serveru a serveru pro běh algoritmů. Byly zprovozněny vybrané algoritmy počítačového vidění, např. detekce obličejů, detekce státních poznávacích značek a obarvení černobílých fotografií. K většině algoritmů byla vytvořena prezentační webová stránka pro jejich demonstraci. Vytvořena byla také ukázková klientská knihovna k usnadnění integrace aplikačního rozhraní v programovacím jazyce Python.

Na závěr byly provedeny výkonnostní testy, které potvrdily výhody zvolené architektury a které ukázaly minimální režijní nároky webové služby s implementovanými optimalizacemi. Většina optimalizací se skládá ze zavedení různých mezipamětí při přístupu k databázi a uložišti souborů.

Současný stav webové služby plně splňuje zadání diplomové práce. Před komerčním nebo veřejným nasazením by bylo potřeba provést několik úprav, které jsou diskutovány v sekci 4.8. Mezi hlavní patří implementace opatření před zneužitím nebo znefunkčněním služby pro ostatní uživatele. Opatření by mohlo mít podobu limitů na počet spuštěných algoritmů za časovou jednotku nebo na velikost nahraných dat pro jednotlivé uživatele. Navržen byl možný směr dalšího vývoje, který obsahuje použití orchestračních nástrojů na správu serverů pro běh algoritmů a možnost přidání vlastních algoritmů uživateli, což by zahrnovalo implementaci tzv. sandbox prostředí pro běžící algoritmy.

Literatura

[1] Chambers, J.; Paquette, D.; Timms, S.: *ASP.NET Core Application Development: Building an application in four sprints*. Developer Reference, Pearson Education, 2016, ISBN 9781509304097.

[2] Chowdhuri, S.: *ASP.NET Core Essentials*. Community experience distilled, Packt Publishing, 2016, ISBN 9781785889547.

[3] Fielding, R.: *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[4] Fielding, R.; Gettys, J.; Mogul, J.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, RFC Editor, January 1997.
URL http://www.rfc-editor.org/rfc/rfc2068.txt

[5] Fowler, M.: *Inversion of Control Containers and the Dependency Injection pattern*. [Online; navštíveno 30. 12. 2016].
URL http://www.martinfowler.com/articles/injection.html

[6] Fowler, M.: *UnitTest*. [Online; navštíveno 02. 05. 2017].
URL https://martinfowler.com/bliki/UnitTest.html

[7] Freeman, A.: *Pro ASP.NET Core MVC*. Apress, 2016, ISBN 9781484203972.

[8] Hohpe, G.; Woolf, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Signature Series (Fowler), Pearson Education, 2012, ISBN 9780133065107.

[9] Hunter, S.; Hanselman, S.: Introducing ASP.NET Core 1.0, 2016, build 2016.
URL https://channel9.msdn.com/Events/Build/2016/B810

[10] Leach, P.; Mealling, M.; Salz, R.: A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, RFC Editor, July 2005.
URL http://www.rfc-editor.org/rfc/rfc4122.txt

[11] Magnoni, L.: Modern messaging for distributed sytems. In *Journal of Physics: Conference Series*, ročník 608, IOP Publishing, 2015, str. 012038.

[12] Masse, M.: *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, 2011, ISBN 9781449319908.

[13] Millerr, R.: *Entity Framework Core*. [Online; navštíveno 07. 01. 2017].
URL https://docs.microsoft.com/en-us/ef/core/index

[14] Mitra, N.; Lafon, Y.: *SOAP Version 1.2 Part 0: Primer (Second Edition)*. Recommendation, W3C, Duben 2007, https://www.w3.org/TR/soap12-part0.

[15] Mouat, A.: *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, 2015, ISBN 9781491915929.

[16] Silberschatz, A.; Galvin, P.; Gagne, G.: *Operating System Concepts, 9th Edition*. Wiley Global Education, 2012, ISBN 9781118559635.

[17] *Utility Conventions*. [Online; navštíveno 19. 04. 2017].
URL
http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

[18] Zhang, R.; Isola, P.; Efros, A. A.: Colorful Image Colorization. *ECCV*, 2016.

Přílohy

Příloha A

Obsah přiloženého paměťového média

Na přiloženém CD jsou následující adresáře a soubory:

- adresář `app` – obsahuje veškeré soubory webové služby, včetně souboru `README`, který obsahuje stručný návod ke kompilaci a zprovoznění aplikace,
- soubor `app.tar` – zabalený webový server ve formě obrazu platformy Docker,
- adresář `webserver` – obsahuje všechny potřebné zkompilované soubory pro webový server,
- adresář `algserver` – obsahuje všechny potřebné zkompilované soubory pro server pro běh algoritmů,
- adresář `client-library` – obsahuje zdrojové soubory pro klientskou knihovnu,
- adresář `samples` – obsahuje podadresáře s jednotlivými implementacemi algoritmů počítačového vidění zmíněné v sekci 3.5,
- adresář `docs` – obsahuje zdrojový tvar technické zprávy včetně všech příloh potřebných ke kompilaci,
- soubor `DP.pdf` – technická zpráva ve formátu PDF,
- soubor `poster.pdf` – plakát ve formátu PDF,
- soubor `video.mp4` – prezentační video ve formátu MP4.

Příloha B

Zprovoznění webové služby

Zprovoznění webové služby se skládá z několika kroků. Je to instalace webového serveru, instalace serveru pro běh algoritmů, zprovoznění RabbitMQ serveru, MongoDB uložiště a MS SQL nebo PostgreSQL databáze. Instalace zmíněných externích služeb není předmětem této kapitoly.

Na začátku vývoje byl server pro běh algoritmů implementován v rámci webového serveru a tato možnost zprovoznění služby byla zachována pro jednoduché případy bez potřeby škálování služby.

Následuje popis instalace webového serveru viz B a serveru pro běh algoritmů viz B.1.5. Oba servery běží na frameworku .NET Core verze 1.1.2. Pro spuštění je nutno mít nainstalované běhové prostředí (anglicky runtime) [1]. To se netýká případu využití služby Docker pro nasazení webového serveru.

B.1 Instalace webového serveru

Instalace webového serveru se liší podle zvoleného způsobu nasazení: s použitím služby Docker viz B.1.4 nebo bez použití viz B.1.5. Oba způsoby se neliší v použití reverzního proxy serveru, správy databáze a konfiguraci webového serveru.

B.1.1 Reverzní proxy server

ASP.NET Core využívá multiplatformní webový server s názvem *Kestrel*. Jelikož je Kestrel poměrně nový webový server a ještě nemá implementována všechna potřebná zabezpečení proti vnějším útokům, je oficiálně doporučováno pro produkční nasazení použit reverzní proxy server [2].

Při testování na operačním systému Ubuntu byl použit reverzní proxy server *Nginx*. Nginx je plnohodnotný webový server, který přeposílá požadavky na vytvořenou aplikaci obsahující Kestrel. K zprovoznění proxy serveru je nutné editovat soubor `/etc/nginx/sites-available/default` a přidat nový záznam instruující proxy server. Příklad záznamu, viz 12, nastavuje přeposílání všech dotazů z portu 80 na lokální port 5000.

[1] Možnost stažení pro různé OS: https://www.microsoft.com/net/download/core#/runtime

[2] Zdroj: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel#when-to-use-kestrel-with-a-reverse-proxy

```
server {
    listen 80;
    location / {
        proxy_pass http://localhost:5000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection keep-alive;
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Ukázka 12: Příklad záznamu, který nastavuje reverzní proxy server Nginx.

B.1.2 Správa databáze

Databáze je vytvářena na základě modelu pomocí objektově relačního mapperu Entity Framework Core. Při změně v modelu je přidána migrace, která transformuje současný stav databáze na nový stav databáze. Pro všechny tyto úkony (vytvoření databáze, přidání migrace a vymazání databáze) byly vytvořeny skripty pro příkazový řádek Bash. Skripty se nacházejí ve složce `scripts`.

K vytvoření nebo aktualizaci databáze stačí spustit skript `./updatedatabase.sh`, jenž provede akci na databázi nastavenou v konfiguračních souborech. Po úpravě modelu je nutné spustit skript společně s názvem migrace, např.: `./addmigration.sh Init`.

Databázové migrace mají tvar podle použitého databázového poskytovatele. To znamená, že při použití Microsoft SQL bude vypadat migrace jinak než při použití PostgreSQL. Při prvotním nasazení je nutné přidat první migraci, která celou databázi vytvoří. Po přidání migrace stačí spustit webový server (webový server při spuštění kontroluje automaticky nové migrace a případně je aplikuje), nebo spustit skript `updatedatabase.sh`.

B.1.3 Konfigurace

Konfigurace webového serveru probíhá dvěma způsoby: na základě konfiguračních souborů ve formátu JSON a na základě proměnných prostředí. Konfigurace je importována podle pevně daného pořadí. Později importované nastavení přepisuje dříve definované nastavení.

Základní konfigurační soubor je `appsettings.json`. Tento soubor většinou obsahuje obecné nastavení, mezi které patří například nastavení stupně logování. V případě webového serveru obsahuje také nastavení měkkého a tvrdého časového limitu a nastavení přihlašovacích údajů pro automaticky vytvořeného uživatele, viz 13. Soubor je načten ze složky, v níž se nachází webový server (konkrétně soubor `CVaS.Web.dll`, který je vstupním bodem webového serveru).

Název dalšího konfiguračního souboru je `appsettings.{název_prostředí}.json` a záleží na prostředí, v němž je webový server spouštěn. Prostředí se nastaví pomocí proměnné prostředí, například pro linuxový operační systém:

```
export ASPNETCORE_ENVIRONMENT=Development
```

Po provedení zmíněného příkazu by byl načten soubor `appsettings.Development.json`. V něm se nachází nastavení specifické pro prostředí, ve kterém je webový server nasazen.

```
{
  "Database": {
    "DefaultUsername": "username",
    "DefaultEmail": "adamjez@outlok.cz",
    "DefaultPassword": "Password1!"
  },
  "Algorithm": {
    "HardTimeoutInSeconds": 60,
    "LightTimeoutInSeconds": 30
  }
}
```

Ukázka 13: Obsah konfiguračního souboru `appsettings.json` s obecným nastavením, konkrétně specifikace časových limitů a přihlašovací údaje pro automaticky vytvořeného uživatele s rolí administrátora.

V případě webového serveru to jsou hlavně tzv. připojovací řetězce (anglicky connection strings). Jsou to řetězce, které obsahují informace o vzdáleném serveru a způsobu připojení k němu.

```
"ConnectionStrings": {
    "MsSql": "Server={server_hostname},1433;Initial
    ↪  Catalog=CVaS;Persist Security Info=False;User
    ↪  ID={username};Password={password};Encrypt=True;",
    "MongoDb":
    ↪  "mongodb://{username}:{password}@{server_hostname}:27017",
    "RabbitMq":
    ↪  "host={server_hostname};username={username};password={password}"
  },
  "Mode": {
    "IsLocal": false
  },
```

Ukázka 14: Obsah konfiguračního souboru `appsettings.Development.json` s nastavením pro konkrétní prostředí. Nejčastěji obsahuje připojovací řetězce ke všem serverům. V ukázce se také nachází vypnutí lokálního serveru pro běh algoritmů, což je výchozí nastavení.

Zvolení typu služby (Microsoft SQL Server nebo PostgreSQL, resp. MongoDB nebo Azure Blob Storage) probíhá podle klíče k danému připojovacímu řetězci. V ukázce 14 jsou připojovací řetězce s klíči `MsSql` a `MongoDB`, a proto budou použity tyto služby. Pro použití služby PostgreSQL, resp. Azure Blob Storage je nutné zadat připojovací řetězec s klíčem `PostgreSql`, resp. `AzureStorage`.

V konfiguraci lze zapnout lokální mód nastavením objektu `Mode` s vlastností `IsLocal` na `true`. V případě zapnutého lokálního módu je server pro běh algoritmů obsažen ve webovém serveru, a proto je nutné, aby konfigurace také obsahovala veškerá nastavení zmíněná dále v sekci B.1.5. Při zapnutém lokálním módu je možné použít uložiště ve formě lokálního

souborového systému, a není tedy nutné zadávat připojovací řetězec pro uložiště ani pro RabbitMQ server.

Naposledy je importována konfigurace ve formě proměnné prostředí. Slouží většinou pro nastavení, které není vhodné ukládat do souborů, nebo pro nasazení ve službě Microsoft Azure, kde je možné proměnné prostředí specifikovat ve webovém portálu. Například následující nastavení přepíše nastavení uvedené v souboru `appsettings.Development.json`.

```
export ConnectionStrings:RabbitMq=
↪  "host={server_hostname};username={username};password={password}"
```

B.1.4 S využitím služby Docker

Využití služby Docker bylo popsáno v sekci 4.1.6. Pro zprovoznění webového serveru stačí stáhnout z oficiálního repositáře vytvořený obraz s názvem `adamjez1/cvas`, viz ukázka 15. Obraz je veřejně dostupný. Po stažení je možné vytvořit z obrazu kontejner příkazem `docker run`. V parametrech tohoto příkazu je nutné předat všechna potřebná nastavení pomocí proměnných prostředí, což se dělá použitím parametru `-e`.

```
docker pull adamjez1/cvas:latest

docker run \
    -p 5000:80 \
    -e "ASPNETCORE_ENVIRONMENT=Production" \
    -e "ConnectionStrings:MsSql=Server={server_hostname},1433;Initial
    ↪  Catalog=CVaS;Persist Security Info=False;User
    ↪  ID={username};Password={password};Encrypt=True;" \
    -e "ConnectionStrings:RabbitMq=
    ↪  host={server_hostname};username={username};password={password}"
    ↪  \
    -e "ConnectionStrings:AzureStorage= DefaultEndpointsProto-
    ↪  col=https;AccountName={account_name};AccountKey={account_key}"
    ↪  \
    adamjez1/cvas:latest
```

Ukázka 15: Ukázka obsahuje nejprve stažení obrazu z oficiálního repositáře a následné spuštění webového serveru s parametry.

Jak již bylo zmíněno v sekci 4.1.6, je nutné také přidat mapování portů tak, aby port, na kterém poslouchá webový server, byl dostupný z hostujícího operačního systému. Specifikovaný port hostujícího systému se musí shodovat s portem zadaným v konfiguraci proxy serveru v předcházející sekci B.1.1. V ukázce 15 je namapován port 80 kontejneru na port 5000 hostujícího systému. Ve výchozím nastavení webový server puštěný v kontejneru poslouchá na portu 80.

Spuštění kontejneru společně s potřebným nastavením, viz ukázka 15, není jednoduché, a proto existuje nástroj *Docker Compose*, který tuto činnost ulehčuje. V souboru, nejčastěji nazvaném `docker-compose.yml`, je specifikováno veškeré potřebné nastavení ke spuštění kontejneru. Další výhoda nástroje Docker Compose je v možnosti definování a spuštění aplikace skládající se z více kontejnerů, např. RabbitMQ, MongoDb a Web server. Nejjednodušší použití nástroje Docker Compose vypadá následovně: `docker-compose up`.

Příkaz `docker-compose up` spustí kontejnery podle konfigurace. Příkaz musí být spuštěn ve složce, kde se nachází konfigurační soubor `docker-compose.yml`. V případě rozdílného názvu souboru je nutné specifikovat název pomocí parametru `-f`. Pro webový server byl konfigurační soubor vytvořen a kromě samotného webového serveru se spustí také RabbitMQ a MongoDB server, ke kterým se webový server připojí.

B.1.5 Bez služby Docker

Bez využití služby je nejprve nutné nainstalovat .NET Core Runtime, což je popsáno na začátku kapitoly. V přiloženém paměťovém médiu se nachází adresář `webserver` (viz A), který obsahuje zkompilované soubory potřebné pro spuštění webového serveru.

Ke spuštění webového serveru je nutné zadat následující příkazy:

```
export ASPNETCORE_ENVIRONMENT=Development
export ASPNETCORE_URLS=http://localhost:5000
dotnet CVaS.Web.dll
```

Proměnná prostředí s názvem `ASPNETCORE_URLS` ovlivňuje port, na kterém bude webový server naslouchat.

Manuální spuštění webového serveru není doporučováno, jelikož při pádu aplikace, nebo při restartování operačního systému nebude server dostupný. Pro takovéto účely slouží služba *supervisor*. Kromě opětovného spuštění webového serveru supervisor taky monitoruje webový server. K zprovoznění služby supervisor je nutné přidat soubor do adresáře `/etc/supervisor/conf.d/`. Nový soubor bude obsahovat konfiguraci služby, která může vypadat jako v ukázce 16.

```
[program:cvas]
command=/usr/bin/dotnet /var/cvas/CVaS.Web.dll
directory=/var/cvas/
autostart=true
autorestart=true
stderr_logfile=/var/log/cvas.err.log
stdout_logfile=/var/log/cvas.out.log
environment=ASPNETCORE_URLS=http://localhost:5000,
↪  ASPNETCORE_ENVIRONMENT=Production
user=www-data
stopsignal=INT
stopasgroup=true
killasgroup=true
```

Ukázka 16: Ukázka konfiguračního souboru pro službu supervisor. Nastaveno je například automatické spuštění po chybě i po spuštění operačního systému.

Přidání konfiguračního souboru je nutné udělat jako tzv. *superuser*. Následně už stačí jenom restartovat službu supervisor:

```
sudo service supervisor stop
sudo service supervisor start
```

B.2 Instalace serveru pro běh algoritmů

Instalace serveru pro běh algoritmů je v mnoha ohledech stejná jako instalace webového serveru. Proto budou zmíněny jenom hlavní rozdíly. Jedním z hlavních rozdílů je nevyužití služby Docker pro nasazení, protože by Docker obraz musel už při vytváření obsahovat instalaci všech potřebných programů pro algoritmy a přidání dalšího by bylo možné jen po vytvoření nového obrazu.

Oproti webové konfiguraci se pro nastavení prostředí používá proměnná prostředí s názvem `NETCORE_ENVIRONMENT`. Pro tento server je nutné nastavit cesty k důležitým složkám a nastavit způsob mazání dočasných souborů, viz 17.

```
"DirectoryPaths": {
  "Algorithm": "/var/cvas/algorithms",
  "Temporary": "/var/cvas/temporary",
},
"FilesCleaning": {
  "PeriodInMinutes": 20,
  "DrivePressureSpaceInMB": 1000,
  "FileCacheRetentionTimeInMinutes": 30,
  "DirectoryMaxSpaceInMB": 2000
}
```

Ukázka 17: Ukázka obsahu konfiguračního souboru pro server pro běh algoritmů.

Důležité je nastavit cestu ke složce, jež obsahuje algoritmy počítačového vidění a složku, do níž budou vkládány dočasné soubory. Mezi dočasné soubory patří uživatelské soubory a výsledky algoritmů.

Nastavení způsobu mazání dočasných souborů obsahuje:

- časový interval skenování dočasné složky,
- velikost složky, při které nastává agresivnější mazání,
- počet minut nepřistoupení k souboru, po kterých bude soubor smazán,
- maximální velikost složky, při které jsou mazány všechny soubory.

V případě operačního systému Windows je nutné pro algoritmy ve formě skriptů přidat cesty k interpretům těchto skriptů (viz 4.2.1):

```
"Interpreters": {
  ".py": "C:\\Programs\\Python\\Python35-32\\python.exe",
  ".ps1": "C:\\Windows\\System32\\WindowsPowerShell\\powershell.exe"
}
```

Stejně jako v případě webového serveru je vhodné spouštět tento server pomocí služby supervisor. Zkompilované soubory pro server se nachází na paměťovém médiu ve složce `algserver`.

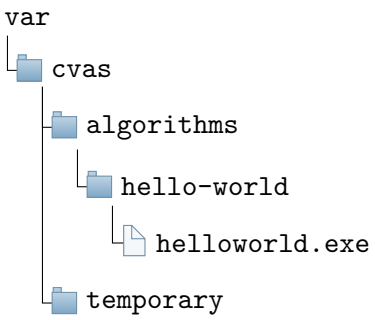
B.3 Přidání algoritmu

K přidání nového algoritmu je zapotřebí přidat záznam do databáze a přidat spustitelné soubory do očekávané adresářové struktury. Přidávání není nijak automatizované, jelikož to nebylo požadováno, a tuto činnost provádí jenom administrátor služby.

Přidání záznamu do databáze může vypadat následovně:

```
INSERT INTO Algorithms (CodeName, Description, FilePath, Title)
↪ VALUES ('hello-world', 'Sample description for hello world
↪ algorithm', 'helloworld.exe', 'Hello World!');
```

V takovémto případě musí adresářová struktura odpovídat obrázku B.1. To znamená, že adresář s názvem odpovídající kódovému jménu algoritmu musí byt umístěn ve složce s algoritmy, v uvedeném příkladu `hello-world`. Adresář dále musí obsahovat spustitelný soubor s názvem odpovídajícímu záznamu, v uvedeném příklady `helloworld.exe`.



Obrázek B.1: Ukázka adresářové struktury na serveru pro běh algoritmů obsahující jeden *helloworld* algoritmus. Název základních adresářů byl použit podle konfigurace v ukázce 17.

Následné přidání prezentační stránky pro algoritmus je jednoduché. Stačí v projektu s webovým serverem do složky `Views/Presentation/` vložit soubor, jehož název odpovídá kódovému jménu po menší transformaci. V případě kódového jména `hello-world` bude název souboru `HelloWorld.cshtml` (děje se to z důvodu dodržení konvencí pojmenování souborů). Obsah souboru už nebude dále popsán, je možné jej odvodit od obsahu ostatních souborů/stránek.