České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jakub Pýcha**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Serverová část aplikace SerialFreak**

Pokyny pro vypracování:

Navrhněte a implementujte serverovou část webové aplikace SerialFreak agregující zdroje informací o seriálech. Aplikace bude integrována se sociální sítí Facebook pro realizaci systému doporučování v rámci sociálních vazeb na této síti. Aplikace nabídne možnost evidence zhlédnutých epizod a jejich hodnocení. Analyzujte technologie pro realizaci zvolené architektury a pro nasazení do cloudového prostředí s ohledem na cenu provozu. Určete způsoby získávání dat z externích zdrojů a vhodné z nich implementujte. Aplikaci průběžně testujte a integrujte.

Seznam odborné literatury:

Haupt, Florian, et al. "A model-driven approach for REST compliant services." Web Services (ICWS), 2014 IEEE International Conference on. IEEE, 2014.

Vedoucí: Ing. Pavel Strnad, Ph.D.

Platnost zadání: do konce letního semestru 2016/2017

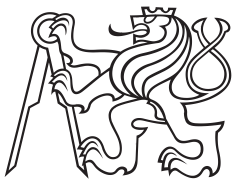
L.S.

prof. Dr. Michal Pěchouček, MSC.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 2. 2016

Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Serverová část aplikace SerialFreak

Jakub Pýcha
Otevřená informatika

Leden 2017
Vedoucí práce: Ing. Pavel Strnad, Ph.D.

Poděkování / Prohlášení

V úvodu bych chtěl poděkovat především vedoucímu mé práce Ing. Pavlu Strnadovi, Ph.D. za záštitu projektu, shovívavost a trpělivost.

Dále bych rád poděkoval mému kolegovi Bc. Filipu Dyrčíkovi za morální oporu v těžkých chvílích projektu a příkladnou spolupráci.

V poslední řadě chci poděkovat mé rodině, ženě Markétě, za podporu a naději do mě vloženou a rodičům, bez kterých bych nebyl tam, kde jsem.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 9. 1. 2017

.....

Abstrakt / Abstract

Cílem této diplomové práce je navrhnout a implementovat serverovou část webové aplikace agregující zdroje informací o seriálech. Práce se zabývá výběrem možných technologií pro implementaci, včetně výběru vhodného cloudového prostředí s důrazem na horizontální škálovatelnost aplikace. Vybrané technologie a postupy poté aplikuje do praxe.

Klíčová slova: Cloud computing, REST, API, MongoDB, Node.js, Microsoft Azure, škálovatelnost, Redis

The main goal of this master thesis is to design and implement server side of the web application. The server side of the web application aggregating information sources about series. This paper deals with the selection of possible technologies for implementation including selection of suitable cloud environment with emphasis on a horizontal scalability of the application. Selected technologies and procedures are applied in practice.

Keywords: Cloud computing, REST, API, MongoDB, Node.js, Microsoft Azure, scalability, Redis

Title translation: Server Side of the application SerialFreak

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

V poslední době s rozšířením dostupnosti vysokorychlostního internetu a digitální televize se zvedá obliba sledování seriálů, ať již se jedná o sledování systematické, nebo náhodné. Popularitu získává obzvlášť sledování seriálů na internetu po sériích, případně na doporučení přátel. Sami patříme mezi nadšené diváky různých seriálů, a proto bychom chtěli sobě i ostatním zjednodušit sledování a nabídnout jim nějakou přidanou hodnotu. Jelikož v počátcích nedokážeme určit, jak bude projekt přijat veřejností, nelze ani zhodnotit, jak velkou infrastrukturu budeme potřebovat, a proto přichází řeč na nasazení do cloudu.

Cloud computing provází v poslední době relativně velká popularita, která je spojená s jednoduchým nasazením v řádu minut a nulovou počáteční investicí. Velikou výhodu tato technologie nabízí obzvlášť pro rozvíjející se týmy a projekty, které nedokáží určit očekávané zatížení, nebo pro projekty, které vyžadují dočasný velký výpočetní výkon, jakožto třeba e-shopy v předvánoční době. Každá technologie či architektura i nasazení do cloudu má svá pozitiva a negativa, ty bych rád analyzoval a navrhl vhodné architektonické řešení i poskytovatele.

Velkou motivací proč řešit tento projekt, který je náš vlastní, je také to, že se jedná o projekt námi vytvořený, spravovaný a vyvíjený, což jsou věci, které by softwarový inženýr měl zvládnout od počátku do konce. Bude se jednat o komplexní projekt, ve kterém budume muset řešit architektonické problémy, problémy vývojového modelu, prostředí, nasazení, problémy integrace, ale možná i problémy finanční. Komplexnost této věci spojená s kompletní realizací, je to, co si chceme vyzkoušet.

1.1 Cíle

Chtěli bychom se zaobírat řešením problému, jak seriály a epizody doporučovat, případně řešit určitý management zhlédnutých epizod seriálů. Celý tento proces by se měl pravděpodobně odehrávat na sociální úrovni, kde přátelé sledující seriály jsou hlavním zdrojem informací.

Práce bude vycházet z kvalitativního uživatelského výzkumu, který bude realizován v rámci práce Bc. Filipa Dyrčíka [1].

Abychom dokázali nějakým způsobem naše úsilí monetizovat, musíme získat velkou uživatelskou základnu, díky které budeme moci získat finanční prostředky z reklamy. S velkou uživatelskou základnou stoupají požadavky na výkon aplikace. Jakožto začínající projekt si nemůžeme dovolit investici do fyzického hardwaru a nasadit řešení on-premise. Vybízí se tedy možnost nasadit celý projekt do cloudového prostředí, kde existuje několik modelů nasazení. V našem případě lze uvažovat o IaaS a PaaS, vhodnost jednotlivých variant bude zhodnocena v analýze.

Celý projekt je rozdělen do dvou částí, z nichž ta zpracovávaná v této práci spočívá v implementaci serveru, zajištění vývojového prostředí a nasazení aplikace.

Cílem práce bude analýza prostředí pro běh, vývojových platforem i finanční stránky provozu. Následovat bude fáze návrhu systému a prostředí pro vývoj spolu se se samot-

nou implementací, nasazením a testováním. Projekt bude v kritických sekcích testován a dokumentován.

Kapitola 2
Úvodní studie

Tato kapitola se zabývá rešerší současných řešení v oblasti seriálových portálů. Na základě výzkumu jsou stanoveny požadavky na aplikaci a určena rizika projektu.

2.1 Rešerše současných řešení

V současné době existuje několik portálů zabývajících se seriály. V naší oblasti zájmu je převážně česká komunita lidí sledující seriály, avšak i zahraniční portály by nám mohly být cennou inspirací.

2.1.1 TVMaze

TVMaze[1]) je webový portál obsahující databázi seriálů, kterou dokáže určitým způsobem personalizovat. Poskytuje uživatelům hodnocení na úrovni epizod i seriálů. Dále si zde lze vést evidenci zhlédnutých epizod a tím zjistit, na jakou epizodu se má divák dívat dále. Uživatelům portálu také umožňuje upravovat informace o seriálech i epizodách. Web dále nabídne uživateli personalizovaný kalendář s jeho seriály, vysílacími daty a časy, který lze exportovat do standardních kalendářů typu Google calendar nebo MS Outlook.

Vyhledávání seriálu, případně epizody by dle mých představ mělo být rychlejší. Nyní lze hledat fulltextově, bohužel bez napovídání, nebo filtrovat seznam všech seriálů na základě několika parametrů, jako je žánr seriálu, vysílací stanice a typ seriálu.

Zajímavou záležitostí týkající se portálu je snaha kolem sebe vytvořit velkou komunitu, a tím dostat uživatele ke tvorbě webu samotného. Uživatel je odměňován různými odznaky a statistikami. Velmi kladně hodnotím interakci s vývojáři formou hlasování, , kde uživatelé hlasují, která nová vlastnost by měla být implementována.

Celý portál je zpracován hezky a přehledně, s velmi decentní reklamou. Po technické stránce hodnotím kladně responzivitu webu a tím jeho přístupnost široké uživatelské základně. Portál poskytuje veřejné API, které lze využívat pod licencí CC BY-SA, tedy ho lze volně používat pod podmínkou uvedení zdroje.

2.1.2 IMDb

IMDb[2]) je velmi známý portál, hlavně v oblasti filmů. Jedná se o webovou databázi filmů, tvůrců, herců a také seriálů. Tato databáze se řadí mezi největší, avšak je více zaměřena na samotné filmy nežli seriály.

Uživatelům nabídne hlavně informace v oblasti hodnocení, ať již se jedná o recenze uživatelské, či odborné, které mohou být na úrovni seriálu i epizod. Poskytuje také spoustu dalších informací, jako jsou trailery, vysílací časy, popis děje, případně zajímavosti a technické specifikace.

[1]) `http://www.tvmaze.com/`
[2]) `http://www.imdb.com/`

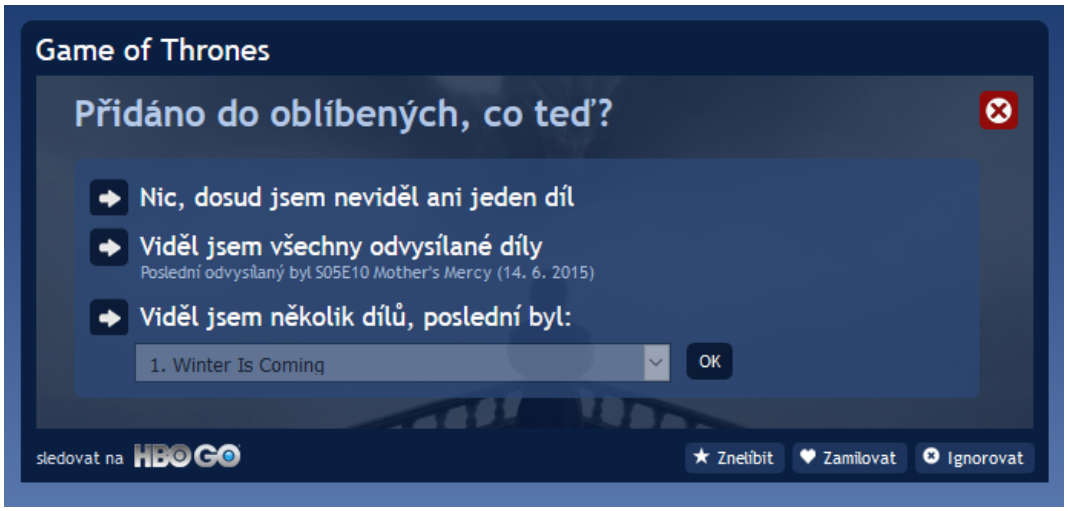
Vyhledávání je velmi rychlé a přesné, lze hledat obecně ve všech dostupných atributech, případně hledat pouze na základě atributu jednoho. Vyhledávání nabízí napovídání s náhledem, což je příjemné, a je možné již na úrovni hledání zjistit, zda se jedná o hledanou položku. Dohledat lze seriály, ale i konkrétní epizody.

Portál se po technické stránce snaží jít jinou cestou, než je tomu u portálu TVMaze, má jeden web, který není přizpůsobený pro mobilní zařízení, avšak nabízí nativní aplikace pro platformu Android a iOS. Po designové stránce se zdá být v oblasti seriálů lehce zmatečný. Oficiální API neposkytuje, některé informace lze však získat skrze OMDb API [1]).

2.1.3 SerialZone

SerialZone[2]) Se řadí mezi největší seriálové weby českého internetu. Nabízí lokalizované informace nejen o seriálech, ale i české titulky a vlastní autorský obsah, jako jsou články o seriálech, hercích a tvůrcích.

Uživatelé tohoto portálu si mohou vést evidenci zhlédnutých epizod, u níž je atraktivně řešeno označení zhlédnutých epizod, viz. 2.1.



Obrázek 2.1. Přidání seriálu do oblíbených na serialzone.cz

Po kliknutí na tlačítko ‘oblíbit‘ se portál táže, zda chce divák pokračovat dál, jak je vidět na obrázku. Toto usnadní uživateli označování zhlédnutých epizod a pokryje velkou část možných rozhodnutí. Odkaz ke sledování na oficiálním zdroji je také výhodou, poskytuje možnost legálního sledování během chvilky, problémem však je, že seriálů, které lze sledovat oficiálně, na internetu není mnoho.

Vyhledání seriálu je v pořádku, nenabídne našeptávání avšak po odeslání hledaného výrazu dojde rovnou k přesměrování na nejlepší výsledek s možností vrátit se na skutečné hledání.

Web není responzivní, ani nemá mobilní aplikaci, jeho prohlížení na mobilních zařízeních není pohodlné. Nevystavuje žádné veřejné API.

2.1.4 Edna

Edna[3]) je postavena převážně na autorském obsahu. Nabízí články, informace o seriálech a české titulky pro některé seriály.

[1]) `https://www.omdbapi.com/`
[2]) `http://www.serialzone.cz/`
[3]) `http://www.edna.cz/`

Pozitivní hodnocení zasluhuje funkcionalita doporučení seriálu, kde na základě seriálu, který sledujeme nebo se nám líbí, je doporučen nějaký nový.

Web není přizpůsoben pro mobilní zařízení, ani nenabízí mobilní aplikace. Veřejné API také nevystavuje.

2.1.5 Shrnutí

Po analýze současných řešení v oblasti seriálů hodnotím nejlépe portál TVMaze. Obdobná lokalizovaná aplikace by mohla uživatele zaujmout. Potenciál je také v pokrytí mobilních zařízeních, jelikož žádný z českých portálů nenabízí přijatelné řešení pro obsluhování portálu z mobilního telefonu.

2.2 Seznam požadavků

Požadavek na softwarový systém lze definovat jako specifikaci toho, co by mělo být implementováno, viz Arlow [2]. Jak je zmíněno v této knize, existují 2 základní typy požadavků. První skupina jsou požadavky funkční, které symbolizují, co bude systém nabízet z pohledu funkcionality. Jsou to požadavky, které budou viditelné pro koncového uživatele. Naopak druhá skupina požadavků, nazývající se nefunkční požadavky, specifikuje vlastnosti systému a jeho omezení. Nefunkční požadavky mají dopad na výslednou architekturu aplikace, dokáží ovlivnit navrženou architekturu a ze své podstaty může být změna architektury velmi nákladná.

Při formulaci požadavků na systém jsem vycházel z jednoduchého formátu, jenž je doporučen v Arlow[2]. Tento formát vypadá takto [id] [systém] bude [funkce] , kde id je jednoznačný identifikátor požadavku, systém reprezentuje jeho název a funkce značí vykonávanou funkcionalitu. Každý funkcionální požadavek bude mít stanovenou prioritu na škále 1-3, kde 3 je nejvyšší priorita a 1 nejnižší.

Funkcionální požadavky na systém vychází z práce mého kolegy[1], který se zabýval kvalitativním uživatelským výzkumem. Z tohoto výzkumu jsme se pokusili sestavit funkční požadavky na naší aplikaci. Nefunkční požadavky reflektují požadovanou funkcionalitu a naše předpoklady.

2.2.1 Funkční požadavky

1. **SerialFreak bude umožňovat přihlášení přes Facebook**
   Priorita 3
   Potřebujeme uživatelům nabídnout co možná nejlepší informace, a zároveň obsáhnout velkou množinu seriálů. Není tedy v našich silách tyto informace udržovat ručně, proto budou data získávána ze serverů, jež poskytují veřejné API s informacemi o seriálech.
2. **SerialFreak bude agregovat informace o seriálu z více zdrojů**
   Priorita 2
   SerialFreak bude agregovat informace z různých zdrojů tak, aby dodal uživateli nejlepší možná data.
3. **SerialFreak bude doporučovat seriál na základě aktivity přátel**
   Priorita 2
   V SerialFreaku se nebudou vytvářet nové vazby mezi přáteli, ale budou využity stávající ze sociální sítě Facebook. Na základě aktivity přátel a jejich oblíbených seriálů se bude uživateli doporučovat pro něj zajímavý seriál vycházející z vazby na přítele a jeho seriály.

4. **SerialFreak bude umožňovat osobní doporučení od přítele**
   Priorita 3
   Uživateli bude umožněno dát doporučení na seriál svému příteli. Ten bude moci toto doporučení akceptovat a tím si přidat seriál do sledovaných, nebo ho odmítnout.
5. **SerialFreak bude umožňovat vedení evidence zhlédnutých epizod**
   Priorita 3
   Uživatel bude moci označit epizody seriálu jako zhlédnuté. Tím mu nebudou nabízeny k dalšímu sledování. Variant, jak označit zhlédnuté epizody, by mělo být několik, protože v některých případech je třeba označit epizody hromadně. Typickým příkladem budiž přidání staršího seriálu ke sledování.
6. **SerialFreak bude umožňovat hodnocení zhlédnutých epizod**
   Priorita 3
   Na úrovni epizod bude umožněno hodnotit epizodu na jednoduché škále „líbilo-nelíbilo“.
7. **SerialFreak bude doporučovat epizody ke zhlédnutí**
   Priorita 2
   Na základě preferencí u odebíraného seriálu budou doporučovány na hlavní obrazovce epizody ke zhlédnutí. Přímo na této obrazovce bude umožněno hodnotit danou epizodu, případně ji označit za zhlédnutou.
8. **SerialFreak bude umožňovat Výběr epizody ke zhlédnutí**
   Priorita 3
   Na epizodu seriálu se půjde dostat skrze vyhledávání, případně detail seriálu.
9. **SerialFreak bude nabízet kalendář s daty vysílání sledovaných seriálů**
   Priorita 1
   Každý uživatel bude mít vlastní personalizovaný kalendář, ve kterém se mu zobrazí vysílací časy a data následujících epizod jím odebíraných seriálů. Tento kalendář bude mít týdenní nebo měsíční rozložení.
10. **SerialFreak bude nabízet přehled aktivit přátel v aplikaci**
    Priorita 2
    Uživatel by měl mít přehled o aktivitě přátel. Přehled těchto aktivit bude ve formě takzvaného feedu. Zobrazovat se budou informace o hodnocení epizody, jejím zhlédnutí a informace o odebírání seriálu.
11. **SerialFreak bude umožňovat reagování na aktivity přátel**
    Priorita 1
    Ve feedu aktivit přátel bude umožněno přidat komentář, případně si tuto aktivitu oblíbit.
12. **SerialFreak bude nabízet sdílení vlastních aktivit na Facebook**
    Priorita 1
    Informace o přidání seriálu ke sledování a hodnocení epizody a seriálu bude umožněno volitelně sdílet na zeď svého profilu na Facebooku.
13. **SerialFreak bude notifikovat na zvolené události na Facebooku**
    Priorita 3
    Do notifikačního centra na sociální síti Facebook přijde informace o tom, že byla vydána epizoda odebíraného seriálu, případně o tom, že byl uživateli doporučen seriál, ať již se bude jednat o doporučení systémové nebo uživatelské. Notifikaci na Facebook bude možné v nastavení vypnout, a to i na úrovni jednotlivých druhů notifikací.
14. **SerialFreak bude umožňovat uživatelskou kategorizaci seriálu**
    Priorita 3

Každý seriál bude ve fázi přidání do sledovaných možné kategorizovat v rámci 3 oblastí, které vzešly z uživatelského průzkumu. První je délka seriálu „krátké/dlouhé“, kde hranice mezi těmito možnostmi byla stanovena uživateli na 30 minut. Tuto oblast však nebude muset uživatel zadávat ručně při odebírání seriálu, tato vlastnost bude zjištěna ze seriálu a kategorizace bude automatická. Další oblastí je „dějové/nedějové“. Toto třídění je důležité z pohledu doporučování epizod, protože uživatelé na nedějové koukají náhodně, spíše v závislosti na tom, jak se jim který díl líbil. Na dějové koukají sekvenčně podle pořadí vydání. Poslední oblastí je „odpočinkové/k zamyšlení“. Tato kategorie bude umožňovat filtrovat epizody ke zhlédnutí na základě toho, co uživatel v aktuální chvíli vyžaduje.

15. **SerialFreak bude umožňovat přidání vybraných seriálů do sledovaných**
    Priorita 3
    Každý seriál, který bude dohledatelný v aplikaci, půjde přidat do sledovaných. Toto způsobí, že se zanese do kalendáře a také bude nabízen na hlavní obrazovce ke zhlédnutí.
16. **SerialFreak bude mít vlastní notifikační centrum**
    Priorita 2
    Ve vlastním notifikačním centru se bude zobrazovat více informací, nežli se bude odesílat do Facebooku, kde by některé informace mohly uživatele obtěžovat.

2.2.2 Nefunkční požadavky

- **Klient-server aplikace**
  Systém bude navržen a implementován jako klient-server aplikace, a to z toho důvodu, abychom určitou část zátěže předali na klienta.
- **Nasaditelné do cloudového prostředí**
  Aplikace musí být nasaditelná do cloudového prostředí, které nám umožní rychlé nasazení aplikace s dostatečnou elasticitou.
- **Horizontálně škálovatelná**
  Aplikace musí být horizontálně škálovatelná ve vybraném cloud prostředí. Systém bude flexibilně reagovat přidáváním instancí v případě stoupající zátěže.
- **90% žádostí zpracovaných do 500ms, zbylé do 2s při současném používání 20 uživateli**
  Aplikace musí být dostatečně rychlá, aby práce aplikace nezdržovala uživatele od jejich akce.
- **Dostupnost 99,8% (doba výpadku: 17,52H/rok)**
  Jak je uvedeno v SLA [3] k MS Azure App services, Microsoft zaručuje dostupnost 99,95% . Zbylá doba je rezerva pro případné změny z naší strany.
- **Zabezpečená komunikace po HTTPS**
  Komunikace mezi klientem a serverem bude probíhat striktně po HTTPS, případně po zabezpečené verzi WebSocketu.
- **Nasazená aplikace musí mít možnost monitorování požadavků**
  Nasazená aplikace bude mít možnost monitorování požadavků na aplikaci.
- **Autentifikace pomocí Facebooku**
  Uživatelé se budou autentizovat pomocí Facebooku a pro použití aplikace musí být striktně přihlášený, bez přihlášení nebude možno aplikaci používat.
- **Klientská část aplikace bude responzivní**
  Aby byla aplikace snadno ovladatelná i na zařízeních s menší úhlopříčkou, bude klientská část responzivní. Jak je psáno ve W3C [4], responzivní design je o tom

použít HTML a CSS pro schování, změnění velikosti nebo přesunutí obsahu tak, aby vypadal dobře na jakémkoliv displeji.

2.3 Analýza rizik projektu

Projektové riziko je dle [5] „nejistá událost, nebo podmínka, která jestliže nastane, tak má dopad na nejméně jeden cíl projektu“. Tyto nejisté problémy, které mohou nastat, se snaží řídit a identifikovat takzvaný krizový management. Analýza rizik v rámci krizového managementu vede k jejich identifikaci a řízení těchto rizik tak, aby dopad na projekt v případě jejich naplnění byl co nejmenší, ideálně žádný.

Při identifikaci rizik je třeba určit několik atributů každého rizika. Jedná se o pravděpodobnost, že se tak stane, protiopatření, jeho dopad a pochopitelně popis samotného rizika. V [6] jsou definovány 3 hlavní kategorie rizik a to vývojová rizika, technická rizika a business rizika.

2.3.1 Vývojová rizika

Jsou to rizika, která ohrožují samotný projektový plán. Patří sem problémy s časovým rozvržením projektu či problémy personální (odchod člena týmu, problémy s požadavky na projekt, případně rizika spojená se zákazníkem, např. nespolupráce z jeho strany). Příkladem tohoto typu rizika může být například nasazení nové technologie na projekt, v kterém bude překročena doba učení a projekt se tím pádem dostane do skluzu.

2.3.2 Technická rizika

Jedná se o kategorii, která ohrožuje hlavně kvalitu dodávaného produktu a jeho odevzdání. Patří sem například problémy spojené s implementací, návrhem produktu či údržbou. V případě výskytu tohoto typu rizika může být obtížné požadovanou funkcionalitu naimplementovat.

2.3.3 Business rizika

Tato kategorie se týká ekonomického úspěchu produktu, který tato rizika mohou narušit. V případě, že nastane toto riziko, může dojít k úplnému zastavení projektu, případně jeho omezení. Typickým příkladem této kategorie budiž ztráta podpory ze strany managementu nebo ztráta investora.

2.3.4 Management rizik

Pro management rizik existují dva hlavní přístupy. Reaktivní přístup reaguje až v případě, že riziko nastane a poté ho až řeší, naopak proaktivní přístup se snaží identifikovat rizika dříve, má připravený plán jejich řešení a dokáže na ně zareagovat již v jejich počátku.

2.3.5 Identifikace rizik

Jak je zmíněno výše, u každého rizika budou definovány určité atributy a jeho kategorie. Pravděpodobnost rizika budu určována na stupnici 0-10, kde 10 symbolizuje jistotu, že riziko nastane, a 0, že s jistotou nenastane. Dopad na projekt bude hodnocen na stupnici 1-3, kde 1 je nejmenší a 3 největší.

1. **Odchod člena týmu**

Jelikož celý projekt je prací týmu o dvou lidech, je dopad tohoto rizika velmi vysoký. Při ztrátě jednoho člena by došlo k zastavení vývoje poloviny aplikace a pravděpodobně by došlo k zastavení celého projektu.

Pravděpodobnost: 3

Dopad: 3

Kategorie: Vývojová rizika

Opatření: Vytvořit seznam potencionálních zájemců o projekt, kteří by mohli chybějícího člena nahradit.

2. **Náklady na provoz budou větší, než se očekává**

   Provoz projektu bude vyžadovat měsíčně větší finanční prostředky, než jsou dostupné na provoz. Tím by mohlo dojít k zastavení běhu, případně jeho omezení, nebo nesplnění některých nefunkčních požadavků.

   Pravděpodobnost: 5

   Dopad: 2

   Kategorie: Business rizika

   Opatření: Mít připravený plán pro migraci aplikace na vlastní virtuální server.

3. **Vývoj v nové technologii nebude tak rychlý, jak se očekává**

   Jelikož serverová i klientská část bude vyvíjena na technologiích, s kterými se žádný z členů nesetkal, je možné, že odhadovaná doba na naučení bude překročena a tím bude oddálen konec projektu, případně nedodána celá funkcionalita.

   Pravděpodobnost: 6

   Dopad: 2

   Kategorie: Vývojová rizika

   Opatření: Vlastnit kontakt na lidi, kteří s těmito technologiemi pracují a dokáží v případě nouze pomoci.

4. **Nezájem ze strany uživatelů**

   V případě nezájmu uživatelů o aplikaci bychom nezískali velkou uživatelskou základnu, která je nutná pro úspěšné provozování portálu a pokrytí jeho provozních nákladů.

   Pravděpodobnost: 5

   Dopad: 2

   Kategorie: Business rizika

   Opatření: Mít připravený plán na propagaci našeho portálu. Zajistit propagaci na portálech s online sledováním seriálů, případně na Facebooku.

5. **Malé příjmy z reklamy**

   Uživatelská základna bude dostatečná, avšak reklama nezajímavá, špatně umístěná, nebo bude mnoho uživatelů používat doplňky pro blokování reklamy.

   Pravděpodobnost: 7

   Dopad: 2

   Kategorie: Business rizika

   Opatření: Navrhnout jiné řešení monetizace portálu, například nějaký druh prémiové funkcionality.

6. **Horizontální škálování aplikace**

   Portál nedokáže reagovat na zvyšující se uživatelskou základnu rozšiřováním počtu instancí. Z toho důvodu by došlo k narůstání odezvy a uživatelé by ztráceli zájem naše řešení používat. Toto riziko může nastat také v případě nějakého úzkého hrdla celé aplikace, kterým by mohla být např. databáze. Dopad rizika není markantní, protože se ve většině případů dá škálovat i vertikálně, tedy zakoupit silnější instanci.

   Pravděpodobnost: 2

Dopad: 1

Kategorie: Technická rizika

Opatření: Před finálním nasazením provést zátěžové testy a sledovat latence požadavků na server spolu se zátěží serveru.

7. **Špatně odhadnutý rozsah projektu**

   Špatný odhad velikosti projektu způsobí zpoždění vývoje a tím jeho pozdní odevzdání, které by v tomto případě mohlo vyústit k neudělení zápočtu za diplomovou práci.

   Pravděpodobnost: 6

   Dopad: 3

   Kategorie: Vývojová rizika

   Opatření: Průběžně sledovat vývoj projektu vzhledem k časovým možnostem členů týmu, případně omezit některou méně prioritní funkcionalitu.

8. **Rychlost vývoje serveru vůči klientovi bude rozdílná**

   Jestliže bude průběh vývoje serveru vůči klientovi razantně odlišný, může dojít k nekonzistenci mezi těmito komponentami. To by mohlo vyústit ve vzájemnou nekompatibilitu těchto komponent.

   Pravděpodobnost: 7

   Dopad: 1

   Kategorie: Vývojová rizika

   Opatření: Průběžně integrovat klientskou a serverovou část, tak aby nedošlo k vzájemné nekompatibilitě mezi sebou. Nutností je také pevně definované API, které budou klient i server dodržovat.

9. **Ztráta zdroje dat, případně nesplnění licenčních podmínek**

   Jestliže dojde ztrátě zdroje dat, ze kterého čerpáme, celý portál se octne bez přísunu nových dat, a tím přestane být pro uživatele zajímavý. Stejný problém nastane, jestliže u námi zvoleného zdroje dojde ke změně licenčních podmínek, následkem čehož nepůjde zdroj dále používat.

   Pravděpodobnost: 5

   Dopad: 3

   Kategorie: Technická rizika

   Opatření: Diverzifikovat zdroje dat tak, abychom v případě ztráty jednoho dokázali plnohodnotně nahradit tento hlavní zdroj.

Kapitola 3
Návrh

Tato kapitola se pokusí vysvětlit, jak celý projekt vznikl a byl navržen. Od procesu vývoje po výběr prostředí a návrh architektury.

3.1 Proces vývoje

Každý softwarový projekt musí být nějakým způsobem řízený a spravovaný. Některé projekty využívají dnes již zastaralý vodopádový model, jiné se snaží držet některé z metodik agilního vývoje. V našem případě byl projekt vyvíjen agilně, kdy s postupem vývoje docházelo k drobným změnám, na které jsme mohli velice rychle reagovat a celý projekt se v průběhu času přizpůsoboval okolnostem.

Jedná se o projekt dvou lidí, jehož části jsou striktně odděleny na úrovni aplikačního rozhraní. Ideálním postupem je sestavit na základě funkčních požadavků definici tohoto rozhraní. Z definice následně vznikne mock pro klienta a serverová část bude implementovat odděleně jednotlivé části API.

Téměř každý softwarový projekt využívá verzovací systém a tento projekt nebude výjimkou. Projekt bude mít dva Git repozitáře, jeden pro serverovou, tedy mou část a druhý pro klientskou část, tedy práci mého kolegy.

3.2 API

Api je rozhraní sloužící ke komunikaci mezi komponentami, které jsou na úrovni této definice svázány. Jedná se o soubor funkcí, které komponenta veřejně vystavuje a umožňuje tak jeho využití jinými komponentami.

Protože je v našem případě nutné klást důraz na rozdělení práce každého člena týmu, byl návrh rozhraní mezi serverovou a klientskou částí klíčový.

Pro vzájemné propojení bylo zvoleno REST API. Toto rozhodnutí vyplynulo z požadavků na aplikaci, v kterých je kladen důraz na škálování a distribuovatelnost. Server bude poskytovat data skrze HTTP protokol. Jedná se o protokol, který je široce implementován na většině platforem a není problém pro něj nalézt mnoho podpůrných knihoven a nástrojů.

3.3 REST

Rest je pojem, který definuje architekturu možného rozhraní. Tuto architekturu popsal Roy Fielding ve své disertační práci [7] v roce 2000. Architektura je datově orientována, což můžeme chápat tak, že nám poskytuje zdroje, nad kterými lze provádět operace. Jinak je tomu například v architektuře SOAP, která se orientuje na služby, které poskytuje.

V definici architektury jsou popsány základní operace nad daty, které v případě použití HTTP odpovídají standardním metodám protokolu. CRUD je tak realizován

skze HTTP metody POST, GET, PUT a DELETE. Implementace webových služeb pomocí tohoto architektonického stylu na HTTP se označuje RESTful rozhraní.

Každá RESTful webová služba je definována následující sadou vlastností:

- Klient - server
- Bezestavovost
- Použití HTTP protokolu

Jak bylo zmíněno, tato architektura je založena na principu zdrojů, kde každý zdroj reprezentuje stav nebo chování aplikace a musí být jedinečně identifikován URI identifikátorem.

Obecná konvence praví, že zdroj má být reprezentován podstatným jménem a ne slovesem. Tedy validní název zdroje je například *user* a ne *createUser*, ten by tak neměl symbolizovat jím prováděnou operaci, ale zdroj, nad kterým bude provedena operace v závislosti na HTTP metodě požadavku. Běžným příkladem z praxe je otevření webové stránky v prohlížeči, kdy po zadání identifikátoru dojde k vytvoření požadavku a prohlížeči je vrácena reprezentace požadovaného zdroje, tedy webová stránka.

3.3.1 Bezestavovost

Tato vlastnost architektury je důležitá z pohledu horizontálního škálování aplikace. Každý požadavek si s sebou nese veškeré informace nutné ke zpracování jeho samého a není tedy vyžadován stav zapamatovaný na serveru. Poté není podstatné, zda se požadavek dostane na jeden či druhý stroj, a z pohledu funkce by měla být odpověď totožná v obou případech.

3.3.2 HTTP protokol

V úvodu již bylo řečeno, že běžné CRUD operace se provádí standardní sadou HTTP metod. V následujícím seznamu bude vyloženo, jaká je běžná praxe chování dané metody vzhledem ke zdroji. Technicky vzato zde však žádné omezení není a je tedy možné chování dané metody naimplementovat dle libosti programátora.

- GET - Tato metoda se používá pro získávání reprezentace zdroje. Nejběžnějším příkladem této metody je otevření webové stránky v prohlížeči.
- POST - Slouží k vytvoření nové reprezentace zdroje, jehož následné identifikátory by měl server nějakým způsobem vrátit. Nejčastěji se tak děje v hlavičce HTTP odpovědi na tento požadavek.
- PUT - Touto metodou se provádí změny již konkrétní vytvořené reprezentace zdroje. Korektní chování je tedy nejdříve zdroj vytvořit a až poté případně konkrétní zdroj editovat.
- DELETE - Metoda, která smaže reprezentaci.

S HTTP protokolem souvisí také pojem idempotence [8] metod. Tímto pojmem rozumíme vlastnost metody, která nám říká, že opakovaným voláním nedojde ke změně vnitřního stavu aplikace. Idempotentní jsou metody GET, PUT a DELETE. Jestliže zavoláme metodu GET několikrát po sobě, stále dostaneme stejnou reprezentaci, tak jako se tomu stalo při prvním volání. Při volání metody PUT se bude měnit stále stejný zdroj a změna bude stejná, tedy i vnitřní stav aplikace bude nadále nezávislý na počtu opakovaných volání. Nejinak tomu bude i v případě DELETE metody, jelikož pokud smažeme reprezentaci několikrát po sobě, stav bude stále smazáno, ať již byla metoda zavolána jednou, nebo stokrát. Idempotentní naopak není metoda POST, která s každým voláním vytvoří novou reprezentaci zdroje a vnitřní stav aplikace je tedy po každém volání jiný.

3.3.3 HATEOAS

Jedná se o princip z REST architektury [9], kdy by každá vrácená reprezentace měla poskytnout další informace o možných operacích souvisejících se zdrojem, nad kterým je prováděna operace. Můžeme to chápat tak, že na URI *http://neco.cz/user* je nám vrácena reprezentace, která obsahuje odkazy na další stavy spojené s tímto účtem. Příkladem může být *http://neco.cz/user/address* odkazující na adresu získaného uživatele.

3.3.4 Návrh API

Pro popis rozhraní můžeme využít různorodé nástroje a jazyky. V tomto projektu byl pro popis použit Blueprint [1]), což je jazyk umožňující zapsat definici rozhraní spolu s jejími požadavky a odpověďmi. Tento popis je také možné použít pro definici mock serveru, který následně může používat klient místo dočasně neimplementovaného serveru.

Blueprint je jazyk pocházející od české firmy Apiary[2]), tento jazyk nám umožňuje popsat strukturu a chování jednotlivých zdrojů. Jazyk byl zvolen na základě mých předešlých zkušeností i doporučení vedoucího práce.

Apiary poskytuje nástroj, ve kterém je možné zapsat definici pomocí zmíněného Blueprintu. Tento nástroj však nenabízí pouze to, umožní uživateli psát dokumentaci k rozhraní, stejně tak mu poskytne mock server a možnost sledovat požadavky jdoucí na tento server. Definici lze také využít pro generování integračního testu na server, tímto se však budu zabývat až v jiné fázi projektu.

Pokusím se zde v seznamu rozepsat zdroje vzniklé v závislosti na stanovených funkčních požadavcích. Bližší informace a podrobnější rozpis zdrojů lze nalézt na `https://app.apiary.io/serialfreak1` nebo v příloze C.

- /login Tento zdroj bude sloužit pro přihlášení do serverové aplikace.
- /search Zdroj bude sloužit pro vyhledávání a s ním spojenou funkcionalitu.
- /recommendations Reprezentuje doporučení seriálů pro právě přihlášeného uživatele.
- /notifications Jedná se o zdroj upozornění pro právě přihlášeného uživatele.
- /series Zdroj poskytující informace o seriálech, jejich epizodách atd.
- /users Reprezentuje uživatele a s ním spojené požadavky ohledně seriálů.

Jako formát pro výměnu dat se bude používat JSON. Hlavní výhody použití, namísto XML, bych vyjma lepší práce v Javascriptu, vytkl také menší datovou náročnost a subjektivně snažší čitelnost.

3.4 Výběr implementačního prostředí

Projekt vzniká takzvaně na ‘zelené louce‘, a proto, není implementační prostředí určeno a je nutné nějaké zvolit.

Každý jazyk, případně framework a knihovna mají svá specifika a vlastnosti, které mohou být v určitém typu projektu výhodou a v jiném zase nevýhodou. Pro implementaci aplikace byla zvažována primárně dvě prostředí. Prvním z nich bylo Java EE, případně Spring a druhé Node.js.

3.4.1 Statická, dynamická typovost

V prostředí programovacích jazyků se vyskytují dvě základní paradigmata ohledně určování datového typu proměnných.

[1]) `https://help.apiary.io/api_101/api_blueprint_tutorial/`
[2]) `https://apiary.io/`

Můžeme asi říci, že základním modelem je statické typování, kdy dochází k definici typu na úrovni zápisu každé proměnné. Příkladem tohoto přístupu je například Java, C++ atd. Některé jazyky umožňují takzvaný 'type inference'[10], kdy je možné deklarovat proměnné bez typu s tím, že typ je určen na základě inicializované hodnoty. Statické typování má výhodu v tom, že kompilátor za nás dokáže ohlídat chyby spojené s typem proměnné a vyvarujeme se tím drobným chybám v aplikaci. Na stranu druhou je neustále nutné psát typové deklarace, pokud tedy není použit jazyk, který ovládá type inference.

Dynamické typování nevyžaduje deklaraci datového typu, avšak v některých jazycích je možné ji přesto deklarovat. Typ proměnné je určen až v běhu. To znamená, že v závislosti na datech je možné, že jedna a ta samá proměnná může být v jednom okamžiku řetězec a v druhém číslo. Výhodu dynamického typování lze spatřit v rychlejší tvorbě kódu, která je bohužel vykoupena nemožností kontroly ze strany kompilátoru a IDE. To také souvisí s tím, že u tohoto typového systému není možné ze strany IDE napovídat relevantní výsledky při našeptávání.

3.4.2 Java

Jazyk Java vznikl v roce 1995 [11] a dnes je vlastněn společností Oracle. Jedná se o interpretovaný objektový jazyk, staticky typovaný. Implementace v prostředí Java je vhodná obzvlášť z pohledu dlouhé historie tohoto prostředí a s ní spojené odladěnosti. Pro implementaci podobné funkcionality nabízí Java framework Java EE, který dokáže nabídnout moduly pro webové služby(Jax-RS), orm vrstvu (JPA) a další podobné nástroje. U Springu je situace obdobná a nabízí pro implementaci ekvivalentní knihovny frameworku Java EE.

Pro Javu existuje balíčkovací systém *Maven*[1]). Pomocí tohoto systému je možné řešit závislosti na různých knihovnách bez toho, aniž bychom je museli někde ručně stahovat. Tento repozitář obsahuje na základě webu Module Counts [2]) v době psaní diplomové práce(listopad 2016) kolem 165 000 modulů.

3.4.3 Node.js

V případě Node.js je situace opačná. Pro tuto platformu se píše v jazyce Javascript. Tento jazyk je na rozdíl od Javy dynamicky typovaný a platforma je oproti Javě o 14 let novější. Ta se těší v poslední době také veliké oblibě, obzvláště mezi webovými a mobilními vývojáři. Ve využití JavaScriptu spatřuji několik výhod.

Jednou z výhod je, že node dokáže být při stejném nasazení v porovnání s běžnými webovými technologiemi propustnější. Můžeme se podívat na benchmark v [12], kde dochází k porovnávání Node.js, PHP a Pythonu. Test je prováděn metodou benchmarku i uživatelského scénáře. Pro benchmark je použito vrácení obligátního 'Hello world', počítání Fibonačiho čísla a také dotaz do databáze. Prakticky ve všech testech node exceluje, zvládá odpovídat na 1500 požadavků za sekundu pro výpočet Fibonačiho čísla 20. Php je v tomto případě schopné odpovídat 170 požadavkům za sekundu.

Za další výhodu může být považován jeden jazyk na serveru, potenciálně i klientovi. Jelikož se v podstatě stále jedná o ten samý známý Javascript z prohlížeče, je jednoduché se pro webové vývojáře adaptovat na serverový Javascript.

Node také tím, že je oproti Javě více nízko úrovňový a v jádru jedno vláknový, nevyžaduje takové hardwarové prostředky, jako je tomu například u Java EE.

[1]) `https://maven.apache.org/`
[2]) `http://www.modulecounts.com/`

Pro Node.js je dostupný balíčkovací manažer NPM. Je to jistá obdoba Mavenu v Javě, kde pro definici závislostí slouží soubor *package.json* namísto *pom.xml*. V tomto repozitáři bylo dostupných na základě webu Module Counts [1]) v době psaní diplomové práce (listopad 2016) kolem 360 000 modulů, což je více jak 2x tolik než v případě Mavenu.

Tato platforma bude více přiblížena v sekci Node.js 4.1

3.4.4 Rozhodnutí

Na základě zmíněných vlastností, bylo rozhodnuto pro použití prostředí Node.js. Nemalou měrou k tomu přispělo také to, že Java EE prostředí ovládám a rád bych se pokusil proniknout do jiného paradigmatu programování a jazyka. Pro potřeby pochopení jiného paradigmatu a v počátcích vývoje mi byla nápomocna kniha Node.js in Action [13].

3.5 Datová vrstva

Každá aplikace si v případě potřeby ukládá data. Ta mohou být uložena mnoha způsoby, nejčastějším je použití databáze, stejně jako v případě naší aplikace.

Nelze si nevšimnout, že se v poslední době vynořil na povrch fenomén známý pod názvem NoSQL databáze, který je postaven na jiném principu než tradiční relační databáze. Pojmenování NoSQL by mohlo ve čtenáři evokovat, že se jedná o databázi, která nepoužívá k dotazování SQL jazyk, avšak tomu tak není a i jazyk SQL může být v některých případech pro dotazování použit.

3.5.1 Relační databáze

Relační databáze je tu s námi již mnoho let a enginy na kterých, tento typ databází pracuje, jsou za tuto dobu značně výkonné a efektivní.

Každý druh databáze má určité vlastnosti, které bych nerad nazval výhodami, či nevýhodami. Je to vlastnost daného druhu databáze, se kterou nelze nic dělat a je nutné ji přijmout.

- Transakce

  Transakce je nedílnou součástí relačních databází a je velice důležitá, pokud se implementuje systém, kde je nutné udržet stav dat po určitou dobu, bez jakékoliv možnosti tyto data poškodit, nebo změnit. Příkladem budiž připsání peněz na bankovní účet. V tomto případě jistě nechceme, aby byla byť sebemenší šance neúspěšné operace a byl připuštěn nevalidní stav. Transakci můžeme také jinak chápat jako výčet operací, kterými dostaneme databázi do stavu jiného. V případě problému není provedena ani jedna operace a stav databáze je obnoven na původní.
- Referenční integrita

  Referenční integrita je omezení, které vylučuje existenci jednoho záznamu bez druhého. Praktickým příkladem je vazba člověka na místo narození, kdy každý člověk se někde narodí a existence člověka bez místa narození není přípustná. Tato vlastnost nám tak dokáže zaručit konzistenci dat mezi sebou a nemusíme se tedy obávat, že by výstupem mohl být člověk bez místa narození.
- Schéma

  Relační databáze jsou také úzce spjaty s tím, že každá databázová entita má schéma neboli definici toho, jak jsou data uložena a jaký je datový typ každého

[1]) `http://www.modulecounts.com/`

atributu. Tím je zaručeno, že není možné, aby například věk člověka byl v jednom případě reprezentován řetězcem a v dalším číslem. Klient požadující tato data si tedy poté může být jist, že každý vrácený věk dané entity bude definoveného datového typu.

- Spojování tabulek

  Neposlední věcí typickou pro relační databáze je možnost spojovat tabulky mezi sebou neboli provádět takzvaný JOIN. Je tak možné vytvořit pohled na data, která jsou reprezentována mnoha spojenými tabulkami. Tato vlastnost je také spojena s pojmem zvaným normalizace databáze. Účelem normalizace[14] je využití výhod relačního modelu a pracovat s daty takovým způsobem, aby data nebyla ukládána redundantně a byla dalším způsobem nerozložitelná.

3.5.2 NoSQL

Přestože může působit, že NoSQL databáze byly vynalezeny až v posledních několika letech, není tomu tak. Dle [15] sahají počátky těchto databází již do let šedesátých minulého století.

Obrovský vzestup tento druh zaznamenal však až s pomyslným příchodem Webu 2.0 [16] a velkých internetových společností (Facebook, Google). Web přestal sloužit primárně k prezentaci informací, ale také k tomu, že uživatelé tvoří obsah samotný.

To přineslo velké množství informací, které je nutné ukládat a analyzovat. V tomto odvětví však neposkytují standardní relační databázové systémy dobrou podporu a výkon, protože udržovat referenční integritu, transakce a možnost spojení tabulek skrze více databázových uzlů není snadné. Proto je problémové a náročné škálovat relační databáze horizontálně a je preferováno škálování vertikální, které pochopitelně naráží na limity aktuálně dostupného hardwaru.

NoSQL koncept má také jistá specifika, jako je tomu u relačních databází. Specifika však nemusí platit globálně přes všechny implementace této kategorie databází.

- Nepodpora transakcí

  Protože udržení transakce je náročné, obzvlášť pokud je databáze na více strojích, neimplementují NoSQL databáze transakce. V některých implementacích je možné transakce použít, avšak například pouze na úrovni jednotlivých kolekcí.
- Neexistující referenční integrita

  Udržení integrity napříč kolekcemi a různými instancemi je velmi náročná úloha, ve většině implementací se vypouští i tato vlastnost. Na úrovni databáze tedy nelze zaručit konzistenci stavu a je třeba.
- Nepodpora spojování kolekcí

  Spojení mezi kolekcemi je taktéž věc v tomto konceptu výjimečná. Argumentem, proč toto neexistuje je opět výpočetně náročné spojování různých kolekcí napříč instancemi.
- Bez schématu

  NoSQL databáze se v některých materiálech nazývají bezschémové [17], nebo databáze s dynamickým schématem. Ke snadnému pochopení si opět dovolím použít praktický příklad.

  V relační databázi by existovala entita člověk, která by měla atributy adresa, jméno a věk. Do této entity je možné vkládat pouze záznamy, které mají tyto atributy stejného datového typu. V bezschémové databázi můžeme vložit jeden záznam s věkem reprezentovaným číslem a druhý záznam s věkem reprezentovaným řetězcem.

  Tím, že databáze nevyžaduje schéma, je možné jednotlivé kolekce snadno rozšiřovat a ukládat data, o jejichž struktuře nic nevíme. Toto ‘schéma‘ je možné měnit za běhu.

Při zamyšlení jistě čtenáře napadne, že to může být zdroj problémů v aplikaci, protože programátor se nemůže spolehnout na strukturu dat, která dostane. Následně je již na programátorovi, který přístup mu vyhovuje více, zda tvorba schématu a vždy stejná struktura dat v aplikaci, nebo rychlé ukládání dat s nutnou kontrolou při zpracovávání.

- Horizontální škálovatelnost

  Protože tento koncept nevyžaduje vlastnosti běžné pro relační databáze, je snazší provádět horizontální škálování a sharding 3.5.3.
- Výkon

  NoSQL neřeší integritní omezení, výkon těchto databází by měl být vyšší, než je tomu v případě databází relačních.

Dle [18] lze dále koncept rozdělit na několik typů, které se odvíjí od způsobu, jakým jsou uložená data.

- Dokumentové

  Databáze, která páruje klíč dokumentu s dokumentem. Tento dokument následně může obsahovat komplexní datovou strukturu skládající se z polí, vlastností a podobjektů. V těchto typech je většinou umožněno se dotazovat a pracovat i nad úrovní dat obsažených v dokumentu.

  Příkladem je například jedna z nejznámějších dokumentových databází MongoDb případně CouchDB
- Key-value

  Je v základu pouze jednoduchá mapa, v níž je pro každý klíč definována hodnota. Redis a obdobné paměťové keše jsou typickým příkladem této kategorie.
- Grafové

  Slouží k ukládání informací o grafech, uzlech a vazbách mezi nimi. Vhodným použitím může být řešení sociálních vazeb mezi uživateli.

  Zástupcem kategorie je například Neo4J .
- Sloupcové

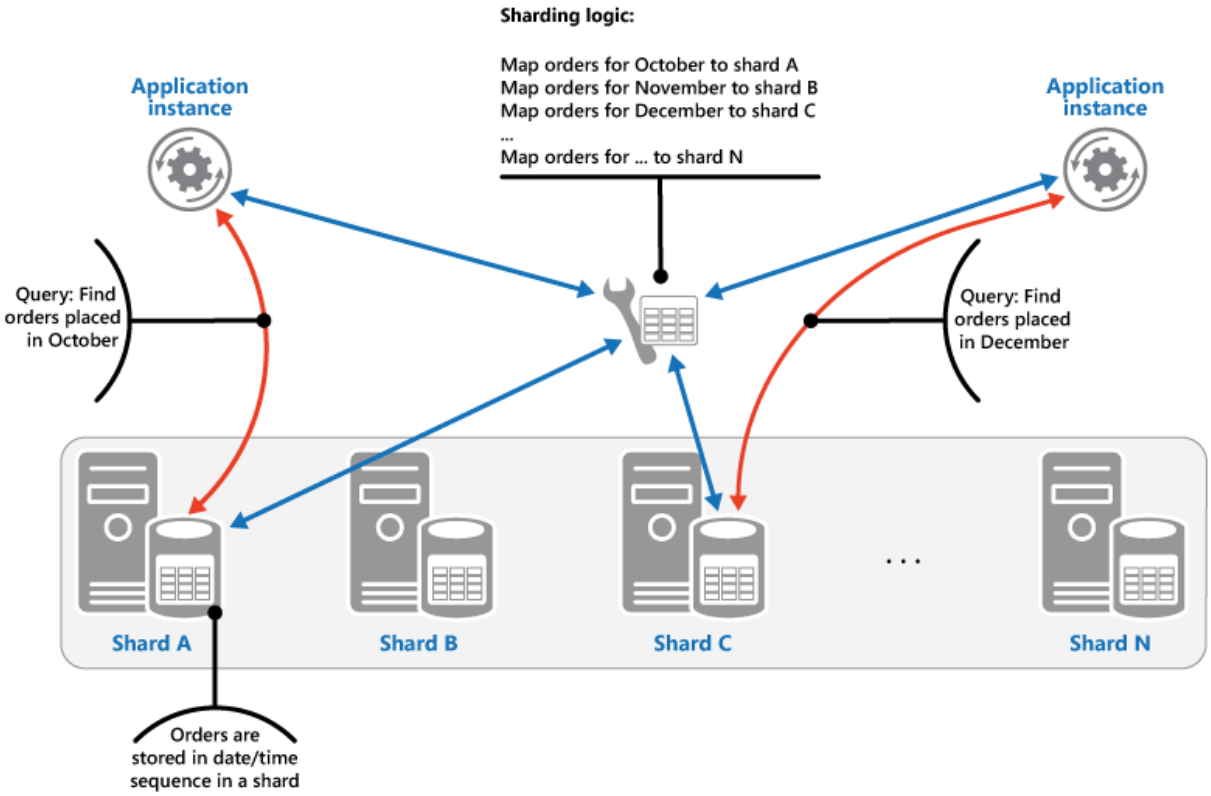
  Základem tohoto typu je řádek tabulky, který je identifikován řádkovým klíčem. Tento řádek může obsahovat nedefinovaný počet sloupců, z nichž každý obsahuje název atributu a jeho hodnotu. Tento model by se mohl tvářit jako standardní relační databáze, avšak řádek zde může obsahovat rozdílné sloupce v rámci jedné tabulky.

  Cassandra a HBase je příkladem sloupcové databáze.

3.5.3 Sharding

Výše jsem zmínil, že relační databázovou vrstvu není možné neomezeně vertikálně škálovat, což je způsobeno tím, že výkon jednoho stroje je limitován současnou situací na trhu procesorů, pamětí a pevných disků. Kdežto možnost škálovat aplikace horizontálně je defacto neomezená. Kvůli těmto omezením existuje přístup nazývající se sharding [19].

Pojmem se rozumí horizontální rozdělení na úrovni dat, kde se každý oddíl nazývá shard. Každý z těchto shardů je poté možné nasadit na jinou instanci databázového stroje. Lépe si pojem popíšeme na obrázku 3.1.



Obrázek 3.1. Ukázka horizontálního rozdělení databáze, převzato z [19].

Mějme tabulku, která obsahuje například 100 milionů záznamů. Jelikož se jedná o rozsáhlou tabulku, rozhodneme se tabulku horizontálně rozdělit na 5 uzlů neboli shardů. Analýzou tabulky dojdeme k závěru, že ideálním řešením bude rozdělit tabulku po 20 milionech záznamů. Dílčí části se rozdistribuují na jednotlivé databázové stroje. V aplikaci si ale nechceme pamatovat, kde je jaká část dat uložena, proto se před tyto stroje předřadí stroj jiný, který v sobě implementuje logiku směrování požadavků na jednotlivé instance. Přijde-li požadavek na objednávky pro listopad, router ví, na který shard tento požadavek přeposlat, a také tak udělá.

Hloubavější čtenáře jistě napadne, že router musí směrování určit na základě nějakého atributu dotazu. Tento atribut se nazývá shard key a je složen z atributů nejčetnějších dotazů.

3.5.4 Výběr

SerialFreak má být sociální sítí, která pracuje s daty poskytnutými třetí stranou. Její vývoj probíhá agilně s rychle se měnícími požadavky. Data, která budou zpracovávána, není nutné mít na úrovni databáze konzistentní a transakce nejsou vyžadovány. Nefunkční požadavky kladoucí důraz na cloudové prostředí a snadnou škálovatelnost také nahrávají použití NoSQL databáze.

Pro naše řešení je vhodnější použití dokumentové NoSQL databáze, která klade důraz na rychlé změny schématu a možnost škálování.

3.6 Zdroje dat

Aplikace musí pracovat nad databází seriálů, jejich sezón a epizod. Tato databáze může být rozsáhlá a nebylo by v našich silách ji udržovat, proto potřebujeme získat data z volně přístupné externí databáze.

Klíčovou podmínkou ve volbě databáze byla možnost volného použití dat, případně použití s uvedením autora. Hledání databáze bylo započato u programu Kodi[1]). To je open source aplikace pro přehrávání a správu multimediálního obsahu. Tato aplikace získává data o seriálech a filmech ve výchozím stavu ze dvou zdrojů. A to konkrétně z themoviedb.org3.6.1 pro filmy a z thetvdb.com3.6.2 pro seriály.

3.6.1 The Movie Database

Tato databáze, dále jen TMDb byla založena v roce 2008 [20] a poskytovala obrázky k filmům. V roce 2009 došlo k rozšíření na databázi poskytující informace o filmech a také veřejné API.

Databáze se v prvotní fázi zaměřovala hlavně na filmy a neposkytovala databázi seriálů. V roce 2013 [21] došlo k přidání televizních seriálů. Jako zdroj přidaných seriálových dat je uváděn právě konkurenční TVDB.

V dnešní době obsahuje dle [21] kolem 312 tisíc filmů, 65 tisíc seriálů a 1 200 000 epizod. Databáze je volně uživatelsky editovatelná, tedy ji každý uživatel může editovat dle libosti, přidávat nové epizody, seriály a informace o nich.

Databáze poskytuje veřejné HTTP API, které je rozsáhlé a poskytuje data od základních seriálových informací po doporučování seriálů.

Dle [22] je možné používat API volně pod podmínkou přímého nekomerčního použití a uvedení zdroje dat. Umístění loga TMDb ve vybrané sekci našeho klienta lze považovat za uvedení zdroje.

Pro tuto databázi existuje několik knihoven v npm. Nejpoužívanější je knihovna moviedb [23].

3.6.2 TheTVDB.com

TVDB je komunitní databáze televizních seriálů. Neposkytuje tedy žádná data o filmech a je zaměřená čistě na televizní seriály.

Taktéž se jedná o rozsáhlou databázi, kde statistiky [24] mluví za vše. Obsahuje kolem 65 tisíc seriálů a 2 200 000 epizod. Databáze je uživatelsky editovatelná a otevřená, každý uživatel tak může editovat data o seriálu, epizodě, nebo nahrávat obrázky.

Databáze poskytuje veřejné HTTP API, které vrací odpovědi v formátu XML. Toto rozhraní je spíše jednoduššího rázu a poskytuje převážně informace o seriálech, jejich sezónách a epizodách.

Data jsou poskytována pod CC-BY-NC 4.0 licencí. Tato licence nám dává svobodu v použití dat pokud je uveden autor, v tomto případě TVDB.

V npm se vyskytuje i pro tuto databázi několik knihoven, které nám mohou usnadnit práci s API. Za nejpoužívanější můžeme označit knihovnu node-tvdb [25].

3.6.3 Výběr

Pro rozhodování, kterou databázi použít byla sestavena tabulka.

Při analýze se nepodařilo dohledat žádné informace o limitu požadavků na API databáze TheTVDB, což by mohlo způsobit jistý problém při implementaci.

Dle statistik má obsáhlejší databázi seriálů a epizod TVDB, bohužel má horší API a neuvádí limit požadavků. Do rozhodování bylo tedy ještě zapojeno dohledání konkrétních seriálů a nejnovějších epizod.

První pokus směřoval k dohledání trochu exotičtějšího seriálu, jímž je Autobazar Monte Karlo vysílaný internetovou televizí Stream.cz. Seriál byl překvapivě založen v obou databázích s tím, že obě obsahovaly základní popis seriálu.

[1]) `https://kodi.tv/`

	The Movie database	TheTVDB.com
Počet seriálů	65 tis.	65 tis.
Počet epizod	1 200 tis.	2 200 tis.
Lokalizovaná	ano	ano
API	ano	ano
Knihovna pro API	ano	ano
Otevřená databáze	ano	ano
Limit požadavků	40 za 10 sekund	neuvádí

Tabulka 3.1. Porovnání seriálových databází.

Druhý pokus se týkal značně populárnějšího zahraničního seriálu, kterým je Teorie velkého třesku. Nebylo překvapením, že seriál byl založen v obou databázích včetně několika jazykových mutací a obrázků.

Znatelné rozdíly panovaly v počtu zveřejněných epizod, respektive že některá z databází neobsahovala nejnovější epizody. V době psaní se například TMDB nacházela ve stavu, kdy byla oproti TVDB pozadu o 3 poslední epizody v případě Autobazaru Monte Karlo a 2 v případě Teorie velkého třesku.

Nakonec jsem se přiklonil k výběru TVDB, jelikož je databáze v oblasti seriálů udržovanější a častěji aktualizovaná ze strany komunity. Časté aktualizace jsou důležité kvůli možnosti upozorňování na nové epizody. V pozdějších fázích by bylo ideální v případě datových nesrovnalostí odkazovat uživatele přímo na TVDB, kde by mohli data doplnit. Tato data by se následně přenesla do našeho systému.

Ideálním stavem je kombinace obou zdrojů dat. Dosáhli bychom tím jistě lepšího pokrytí požadovaných seriálů a potenciálně kvalitnějších dat.

3.7 Cloud computing

Cloud computing je dle NIST [26] model dovolující všudypřítomný, pohodlný síťový přístup ke sdíleným počítačovým zdrojům, kterými jsou například servery, úložiště či služby. Toto celé se děje na požádání a je podmínkou, aby to celé bylo spojeno s minimálním úsilím na správu ze strany poskytovatele i uživatele. Celý model cloud computingu se stává z několika oblastí.

Počty vyhledávání ve vyhledávači Google odráží míru popularity dané technologie, na obr. 3.2 můžeme vidět popularitu vyhledávání výrazu cloud computing. Cloud computing je trend, který stále stoupá, a proto i my jsme se rozhodli zvážit tuto variantu pro nasazení naší aplikace.

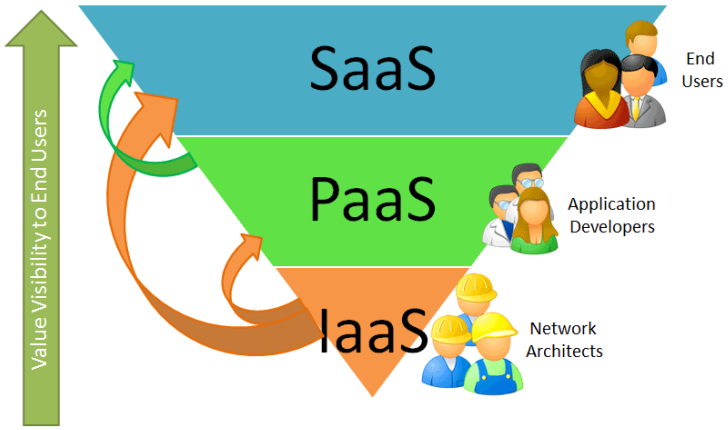
Obrázek 3.2. Trend vyhledávání výrazu cloud computing ve vyhledávači Google

3.7.1 Základní charakteristiky

- Samoobsluha na vyžádání
  Zákazníkovi jsou poskytnuty prostředky dle potřeby automaticky bez vyžádání interakce s každou službou poskytovatele.
- Síťový přístup
  Všechny možnosti nabízené cloudem jsou přístupné skrze počítačovou síť prostřednictvím standardních mechanizmů jako je webový prohlížeč, případně formou tenkých, či tlustých klientů.
- Rychlá elasticita
  Možnosti poskytované cloudem by měly jít snadno škálovat přidáním dalších instancí v případě potřeby, odebráním instancí v případě přebytku výkonu, toto by se v ideálním případě mělo dít automaticky.
- Sdružování prostředků
  Nabízené prostředky jsou v cloudovém prostředí poskytovány v multi tenant režimu, což znamená, že jedna instance prostředku je sdílena mezi více zákazníků. Tímto je dosaženo menších nákladů na údržbu. Zákazník si v cloudovém prostředí typicky nemůže vybrat konkrétní hardware, pouze určit na vyšší úrovni, jaký druh prostředku vyžaduje, jeho výkon, případně jeho geografickou lokaci.

3.7.2 Model služeb

V cloudovém prostředí jsou nabízeny typicky základní 3 modely služeb, které se liší hlavně mírou abstrakce nad hardwarem, a platí, že vrstva výše nedokáže zasáhnout do vrstvy níže. Lepší představu poskytne ilustrace 3.3.



Obrázek 3.3. Dělení zákazníků podle míry abstrakce nad hardwarem, převzato z [27].

- Software as a service (SaaS)
  Je nejvyšší úroveň abstrakce nad hardwarem, tato oblast je zajímavá pro samotné koncové uživatele, kteří využívají software jako službu. Tento software běží na cloudové infrastruktuře a je dostupný odkudkoliv. Zákazník nedokáže ovlivnit nic, co se týká fyzického zařízení, na kterém tento software běží, dokáže měnit pouze nastavení samotného softwaru. Typickým příkladem této oblasti může být úložiště Dropbox, případně Office 365.

- Platform as a service (PaaS)
  Jedná se o prostředí zajímavé pro vývojáře. Na této úrovni dostane zákazník běhové prostředí o výkonu, který požaduje. Toto prostředí je možné integrovat s dalšími službami nabízenými cloudovým poskytovatelem. Zákazník nemá žádnou kontrolu nad vrstvou, která je pod ním, tedy např. nad sítí, operačním systémem, diskovým prostorem. Má kontrolu pouze nad běhovým prostředím a aplikací samotnou. Typickým příkladem je Azure App services případně Google App engine.
- Infrastructure as a service (IaaS)
  Zde se dostáváme z pohledu nasazení nejníže. Na této úrovni si může zákazník propůjčit prostředky týkající se infrastruktury, jako je síť, úložiště, výpočetní stroje atd. Správu těchto zařízení si však již řeší sám, má tedy kontrolu nad operačními systémy, úložišti, firewally, nasazením vlastních aplikací, softwarem atp. Avšak ani na této úrovni nedokáže ovlivnit fyzickou infrastrukturu. Příkladem mohou být Azure virtual machines, Amazon EC2.

3.7.3 Model nasazení

Tato charakteristika rozděluje jednotlivé implementace cloudu na základě toho, kde se z pohledu zákazníka celá infrastruktura vyskytuje.

- Public cloud
  Jedná se o veřejnou službu, kterou může využít jakákoliv osoba. Veškeré operace se vykonávají na infrastruktuře poskytovatele, která je sdílená mezi více zákazníků. Výhodou jsou nulové náklady na spuštění, zákazníka zajímají pouze poplatky za provoz, které se účtují za reálně použitý. Nevýhoda spočívá v tom, že musíme odevzdat poskytovateli i svá data, což není v případě citlivých dat zcela žádoucí.
- Private cloud
  Je opak veřejného cloud, veškerá infrastruktura i provozování leží na bedrech zákazníka a provozovatel a uživatel je typicky jedna a ta samá osoba. Tento druh cloudu si mohou dovolit většinou pouze velké organizace a společnosti, protože vyžaduje velkou fyzickou infrastrukturu, která je spojená s vysokými náklady na pořízení a provoz samotný. Výhoda řešení spočívá v kompletní správě celé infrastruktury, a tím i možných úprav na míru.
- Hybrid cloud
  Je spojení a kombinace dvou předešlých. V hybridní implementaci jsou některé služby provozovány na straně zákazníka a některé na straně veřejného cloudu. Toto řešení vyžaduje určitou míru údržby ze strany zákazníka. Typickým příkladem mohou být data uložená na straně zákazníka v privátním cloudu a výpočetní prostředky u poskytovatele v cloudu veřejném.

3.7.4 Důvody použití

Pro nás,jakožto pro projekt bez velkých finančních možností, je použití cloud computingu ideální variantou, která má však i určité nevýhody.

Výhody

- Nulová počáteční investice
  Pro provoz aplikace v cloudu není potřeba žádná počáteční investice na nákup infrastruktury. Platit budeme pouze za provoz aplikace a služeb s tím spojených.
- Rychlá konfigurace prostředí
  V případě využití PaaS je běhové prostředí velmi rychle nakonfigurováno a během chvíle je možné aplikaci nasadit a spustit.

- Škálování a balancování zátěže

  V případě rostoucí uživatelské základny nebudeme muset řešit zvyšující se nároky, ale cloud bude automaticky reagovat na zvyšující se zátěž. Load balancing je v případě cloudového prostředí také vyřešen.
- Dostupnost služeb

  Většina cloudových poskytovatelů na své služby poskytuje smluvní garanci o dostupnosti služeb, tzv. SLA. Dostupnost služeb nabízenou poskytovateli by se nám na vlastní infrastruktuře nepodařilo současnými prostředky dosáhnout.
- Bezpečnost

  Data, nastavení, výpočetní uzly, vše je v cloudu zálohované a redundantní, tedy v případě pádu jednoho není problém nahradit jeden uzel druhým. Tohoto parametru by se nám podařilo dosáhnout až s vynaložením velkého úsilí.

Nevýhody

- Vendor lock-in

  Poskytovatelé služeb vystavují různorodá API a specificky implementované prostředky. Použitím těchto specifických vlastností dojde k uzavření aplikace do služeb jednoho poskytovatele. Migrace aplikace do prostředí poskytovatele jiného nemusí být snadná, typicky je hodně nákladná, protože může vyžadovat přepsání některých klíčových částí aplikace.
- Cena provozu z dlouhodobého hlediska

  Z dlouhodobého hlediska se při konstantní zátěži mohou dostat náklady vysoko a nákup hardwaru, případně jeho pronájem a vlastní administrace by mohla být značně levnější.

3.8 Microsoft Azure

V této sekci bude specifikován tento konkrétní cloudový poskytovatel a důvody, proč jsem zvolil právě jeho.

3.8.1 Komponenty

MS Azure nabízí různé komponenty pro sestavení infrastruktury naší aplikace. Každá komponenta má svou kategorii, jak je vidět na obr. 3.4.

Obrázek 3.4. Přehled Azure komponent, převzato z [28].

3.8.2 Výpočetní uzly

Pro nasazení serverové části aplikace máme v MS Azure několik variant. Protože nemáme prostředky na údržbu vlastní infrastruktury, je pro nás zajímavá pouze varianta PaaS, což splňují Azure App services a Azure Cloud services.

Azure App services poskytují běhové prostředí nad virtuálními stroji, které nevyžaduje administraci, protože do těchto strojů není možno zasahovat. Při vytváření tohoto typu výpočetního uzlu je možné zvolit velikost běhového prostředí, zda je uživatel ochoten prostředí s někým sdílet, případně zda vyžaduje dedikovaný virtuální stroj s více procesory. Pochopitelně se od tohoto parametru odvíjí cena, která se pohybuje od 0 Euro po 752.9 Euro za měsíc provozu jedné instance. U levnějších variant ve verzi basic není možné automaticky škálovat instance, lze pouze ručně navýšit počet instancí.

Azure cloud services jsou infrastrukturně hodně podobné Azure app services, akorát je možné více zasahovat do přiděleného virtuálního stroje a doinstalovat si případně vlastní software. V této variantě jsou poskytovány 2 druhy virtuálních strojů. První

druh nazýván Web role a jedná se o Windows Server s IIS, který slouží primárně k zachycení požadavků. Dalším druhem je worker role, kde se jedná o Windows Server bez IIS a slouží ke zpracování dlouho trvajících požadavků dodaných na tuto instanci například pomocí Azure Queues. Ceny se pohybují od 13 Euro po 1949 Euro za měsíc.

3.8.3 Datové uzly

Pro uložení dat nabízí MS Azure několik možností, které jsou rozdílné způsobem uložení, případně přidanou funkcionalitou. Dle architektury 3.9 a nefunkčních požadavků 2.2.2 nás zajímá hlavně Azure DocumentDB a Redis Cache.

Azure DocumentDB je dokumentová NoSQL databáze. Nabízí několik přednastavených úrovní výkonu a také možnost uživatelského nastavení výkonu. V této službě poskytovatel nabízí [29] nízkou latenci, prakticky neomezené škálování a možnost globální replikace dat. Cena je vypočítána na základě spotřebovaného místa a počtu vyhrazených RU jednotek. RU jednotka je jednotka žádosti, kde 1 RU jednotka odpovídá propustnosti operace čtení s dokumentem o velikosti 1kB.

Azure Redis cache [30] je založená na open source projektu Redis, který nabízí datové key-value úložiště v operační paměti. Toto úložiště může být použito jako mezi paměť pro rychlý přístup ke kritickým datům. Cena se odvíjí od dedikované velikosti paměti, počtu připojených klientů a začíná na 14 Euro za měsíc a končí na 2786 Euro.

3.8.4 Důvody použití

Kterého cloudového poskytovatele použít, bylo posuzováno na základě několika parametrů, mezi nimiž figurovala dokumentace, kalkulačka nákladů a v neposlední řadě nativní podpora Node.js. Do finálního rozhodování byly zařazeny tři z největší veřejných cloudů [31], a to Amazon AWS, Microsoft Azure a Google cloud. Parametry budu hodnoceny na škále 1-3, kde 1 je nejlepší a 3 nejhorší.

Cloudový poskytovatel Amazon je jeden z největších poskytovatelů veřejného cloudu[32]. Zhodnocení parametrů bylo následující: dokumentace 2, nativní podpora node.js 1, kalkulačka nákladů byla shledána nepřehlednou, proto byla ohodnocena známkou 3.

Microsoft Azure je ze zvažovaných poskytovatelů nejnovější a byl založen v roce 2010 [33] pod názvem Windows Azure. Dokumentace byla ohodnocena známkou 1, je přehledná s obsáhlými tutoriály a není problém se v ní orientovat. Nativní podpora je v případě App services i Cloud services hodnocena 1 . Kalkulačka je přehledná a nebyl shledán žádný problém při počítání nákladů, tedy také 1.

Google cloud, dříve známý pod názvem Google App engine, byl hodnocen následovně: dokumentace 1, je přehledná, názorná, s kalkulačkou nákladů byla má spokojenost nižší, ale v rámci kontextu ostatních byla hodnocena známkou 2. S během node.js je to horší a Google nativně nabízí pouze Javu, Python, Go a PHP, avšak podporuje takzvané Managed VMs [34], díky kterým lze doinstalovat podporu například i pro node.js, proto zde hodnotím 2.

Parametr	Amazon	MS Azure	Google
Dokumentace	2	1	1
Podpora node.js	1	1	2
Kalkulačka	3	1	2

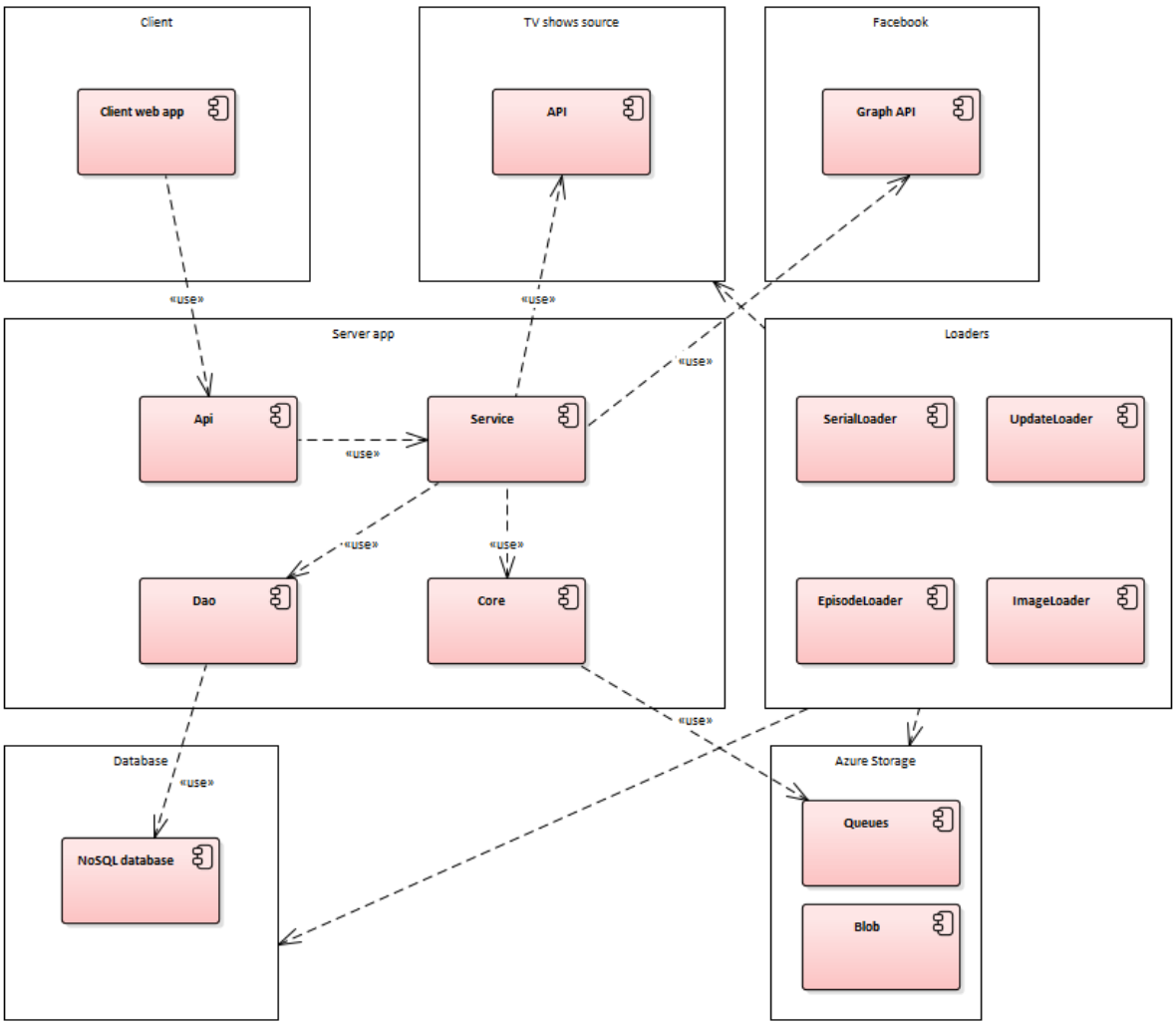
Tabulka 3.2. Výsledné hodnocení cloudových poskytovatelů.

Celkově na základě našich parametrů vyšel nejlépe MS Azure, k jehož volbě nemenší měrou přispělo to, že jsme obdrželi přístup se 100€ měsíčním kreditem na prozatím neurčitou dobu.

3.9 Návrh architektury

Architektura aplikace vždy reflektuje specifikované nefunkční požadavky. Naše aplikace bude navržena jako webová single page aplikace, která bude nasazena do cloudového prostředí ve formě PaaS. Cloudové prostředí bylo zvoleno MS Azure, a proto je diagram nasazení již ovlivněný tímto prostředím.

Aplikace bude obsahovat logické funkční bloky neboli komponenty, které můžeme vidět na diagramu 3.5. Na diagramu vidíme i vazby mezi nimi, které symbolizují jednosměrnou závislost jedné komponenty na druhé.



Obrázek 3.5. Navrhovaný diagram komponent aplikace.

- Client
  - Client web app

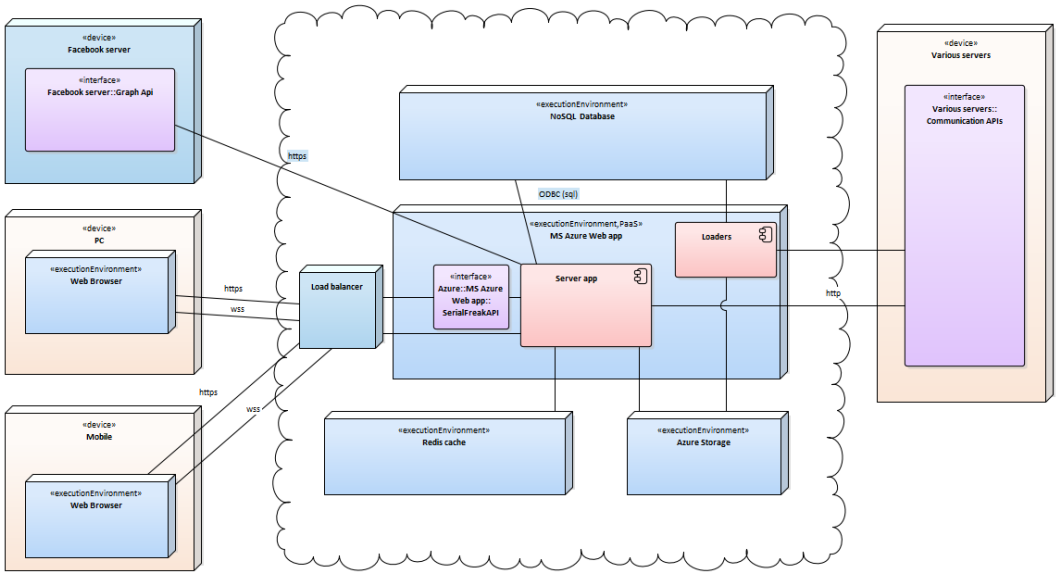
    Komponenta klientské aplikace, která bude komunikovat se serverem skrze REST API. Tato komponenta není součástí této práce a pracuje na ní kolega Dyrčík [1].

- Server
  - API
    Komponenta API prezentuje systém navenek ke klientovi. Jedná se o HTTP REST API zmíněné v 3.4.
  - Service
    Komponenta reprezentující samotnou business logiku aplikace. Zpracovává data ze své databáze a integruje je s externími zdroji.
  - Core
    Komponenta symbolizující pomocné funkce a třídy, které se starají o vkládání zpráv do front, zpracování lokalizace atd.
  - Dao
    Datová vrstva, která slouží pro přístup k databáze. Má programátora odstínit od práce se samotnou databází a snadnou změnu databáze.
- TV shows source
  - API
    Komponenta reprezentující externí API, která bude sloužit pro získávání dat o seriálech a epizodách.
- Database
  - NoSQL database
    Jedná se o vybranou dokumentovou NoSQL databázi, která bude ukládat seriálová data a uživatelskou interakci.
- Facebook
  - Graph API
    Rozhraní na sociální síti Facebook, skrze které je možné získávat přátele, zasílat notifikace atd.
- Storage
  - Queues
    Komponenta, která poskytuje aplikaci fronty, sloužící k asynchronnímu procesování informací.
  - Blob
    Neboli binární úložiště dat, bude sloužit serveru pro ukládání stažených obrázků.
- Loaders
  - SerialLoader
    Komponenta, která bude zpracovávat požadavky z odpovídající fronty. Konkrétně se bude jednat o požadavky na nahrání seriálu z externího zdroje, zpracování a jeho uložení do naší databáze.
  - EpisodeLoader
    Bude sloužit obdobně jako SerialLoader, avšak místo seriálů bude nahrávat konkrétní epizody.
  - UpdateLoader
    Komponenta bude periodicky kontrolovat dostupnost nových dat u externího poskytovatele, případně zařídí nahrání těchto nových dat skrze uložení úlohy do fronty.
  - ImageLoader

Komponenta, která bude zpracovávat požadavky na stažení obrázků do binárního úložiště.

Navrhovaný diagram nasazení je vidět na obr. 3.6. Klientská aplikace bude získána formou statického obsahu z Azure Web app. Před výpočetním uzlem se nachází load balancer pro rozdělování zátěže na jednotlivé uzly. V počátku bude zahájen provoz jednoho uzlu s možností v případě potřeby provoz rozšířit. Klientské aplikace jsou spuštěny ve webovém prohlížeči a se serverovým API komunikují skrze HTTPS, případně zabezpečenými Websockety. Serverová aplikace nasazená na Azure Web app komunikuje s Graph api poskytovaným Facebookem. Pro ukládání dat bude použita NoSQL databáze, případně pro potřebu splnění požadavku na odezvu mohou být některé informace uložené v paměťové keši Redis. Server bude komunikovat s různými API poskytujícími seriálová data. Komunikace není blíže specifikována.

Požadavky na dlouhotrvající asynchronní úlohy budou uloženy do Azure Storage, který poskytuje frontové úložiště. Z těchto front se budou požadavky distribuovat na jednotlivé workery, v naší aplikaci nazývané s příponou Loader. Workeři požadavky zpracují, uloží a při úspěšném pokusu smažou požadavky z fronty. Jestliže není požadavek úspěšný, po definované době se opět objeví ve frontě a pokus je opakován.



Obrázek 3.6. Diagram nasazení aplikace.

Kapitola 4
Implementace

Kapitola návrh nám ukázala, jak by se měl projekt SerialFreaku vyvíjet. V této kapitole čtenáři naopak přiblížím, jak se projekt vyvíjel, jaké vyvstaly problémy, a do detailu popíši některé procesy, aby bylo zjevné, jak je projekt vypracován.

4.1 Node.js

Node.js je Javascriptové běhové prostředí, které vzniklo zhruba před 7 lety, a to konkrétně roku 2009 z rukou Ryana Dahla. Ryan představil tuto platformu na konferenci JSConf [35].

Node.js běží na Chrome V8 Javascriptovém enginu [36] a používá událostmi řízený, neblokující IO model, tím dokáže dosahovat vysoké efektivity hlavně v IO operacích. Hlavní filozofie tohoto prostředí je taková, že veškerý interpretovaný kód běží v jednom jediném vlákně, tak jako JavaScript v běžném webovém prohlížeči. Node jde proti filozofii současných Java serverů, případně Apache [37], kteří s webovými požadavky pracují odlišně. Zmíněné servery pracují na principu vláken, případně procesů, kdy si každý server drží určitý thread pool s připravenou zásobou vláken, která mohou zpracovávat přijaté požadavky. Node jde opačným směrem a díky tomu, že využívá neblokující IO a pracuje na nízké úrovni, vystačí si s jedním vláknem. Využití jednoho vlákna namísto více vláknového zpracování má značnou výhodu z pohledu využitých hardwarových prostředků, kde každé vlákno vyžaduje určitou režii a paměť navíc. V případě použití jednoho vlákna nám tyto problémy odpadají a můžeme tím ušetřit i určitou část prostředků.

4.1.1 Architektura

Node pracuje s jedním vláknem a to tak, že využívá takzvanou event loop [38]. Jedná se o smyčku, která si vezme z fronty určitý požadavek a ten vykoná, poté následuje další požadavek atd. Když jsou všechny požadavky z fronty vyčerpány, proces běhu skončí. A jelikož většina dlouhotrvajících požadavků směřuje na IO operace, jsou primárně prováděny neblokujícím způsobem, ovšem i volání těch blokující je možné. Jeho zavoláním by však došlo k zablokování hlavní smyčky a tedy i běhu programu, proto tyto operace jdou jinou cestou, kterou si vysvětlíme později.

Abych čtenáři lépe vysvětlil rozdíl, mezi blokujícím a neblokujícím požadavkem, podíváme se na následující ukázky kódu.

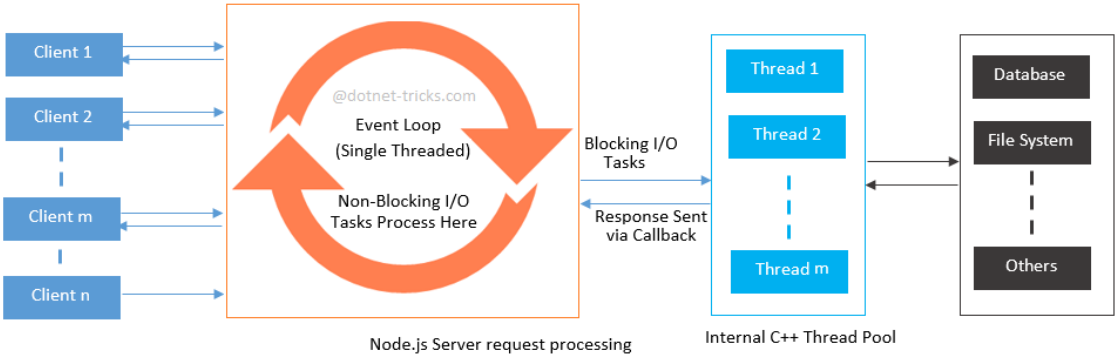
```
var result = db.find(‘SELECT * FROM TEMP‘);
console.log(result);
```

První ukázka pracuje s požadavkem na databázi blokujícím způsobem a to tak, že vykonávané vlákno zastaví svůj běh a čeká, než se mu vrátí požadavek z databáze, což může trvat dlouho.

```
db.find(‘SELECT * FROM TEMP‘, function(err, result){
console.log(result);
});
```

Druhá ukázka jde na dotaz jiným způsobem. Vytvoří požadavek na zpracování vyhledávání do databáze a předá mu funkci, takzvaný callback, která je zavolána, když dojde k dokončení dotazu z DB. Nedochází tedy k blokování hlavního vlákna a program si může běžet neustále dál.

Jelikož ne zcela všechny IO operace běží asynnchronně a proces zpracování požadavků nemusí být zjevný, pokusím se proces nastínit názorněji na modelu zpracování požadavků 4.1.



Obrázek 4.1. Model zpracování požadavků, převzato z [39].

Již bylo řečeno, že každý požadavek na zpracování je zařazen do fronty požadavků. Následně si jedno vláknová smyčka vezme požadavky z fronty a rozhoduje, se o jaký typ požadavku se jedná. Jestliže se jedná o neblokující požadavek, je zpracován ihned v hlavním vlákně, protože takovýto požadavek není náročný a nezastaví zpracování ostatních požadavků. Naopak je tomu u blokujících požadavků, které by mohly způsobit nepříjemné čekání na zpracování dalšího požadavku. Zde nastupuje interní thread pool, který si vezme tento požadavek a zpracovává ho paralelně s hlavní smyčkou. Po zpracování požadavku vrátí vlákno odpověď do smyčky, kde je předána zpět původnímu požadavku. Tvrzení, že je node.js jedno vláknový, není tedy zcela správné, neb vnitřně zpracovává některé operace více vláknově.

4.2 Nastartování projektu

Jelikož se jednalo o nový projekt, bylo nezbytné si připravit prostředí pro vývoj. To spočívalo v instalaci vybraného vývojové prostředí Node.js. Prostředí se v současné době vydává v LTS edici a Current edici. LTS verze softwaru se vyznačují tím, že se jedná o dlouhodobě podporovanou majoritní verzi. Pro ni jsou dodávány aktualizace a záplaty po určenou dobu, přestože se aktuální vývojový kanál nachází již několik majoritních verzí před LTS verzí. V tomto projektu bylo rozhodnuto upřednostnit odladěnou LTS verzi.

S nainstalováním Node.js se nainstaluje i balíčkovací manažer NPM, pomocí něhož řeším závislosti v mém serveru. Kouzelným příkazem *npm install [nazevBalicku]* lze nainstalovat jakýkoliv balíček z repozitáře [1]). Seznam projektových závislostí se ukládá do souboru *package.json*.

Pro vývoj projektu byl vybrán interpretovaný jazyk, proto není třeba nic kompilovat a program se rovnou spustí v interpretru. Složka projektu však obsahuje některé složky, případně soubory, které by se neměly dostat do distribuční verze. Mohli bychom se o postarat ručně a nakopírovat pouze potřebné soubory, ovšem bylo by to zbytečné a

[1]) `http://npmjs.com`

neefektivní. Pro tyto případy existují buildovací nástroje. V tomto případě se většinou nerozumí slovu build jakožto kompilaci, ale jako sestavení aplikace do distribuční podoby. Mezi hlavní zástupce buildovacích nástrojů patří v Javascript světě GULP[1]) a GRUNT[2]). Tyto dva nástroje si zkusíme v následujícím odstavci přiblížit.

Oba spojuje to, že jsou napsány v Javascriptu a jsou uvolněny pod open source licencí. Začneme tedy starším z nich, který vznikl v roce 2012 [40]. Grunt charakterizuje spojení 'Psaní konfigurace na místo kódu'. To je hlavní rozdíl v těchto dvou nástrojích. Gulp zapisuje definici buildu kódem, jež se podobá standardnímu Node.js kódu a standardem je použití streamů jinak hojně využívaných v Node.js. GRUNT je více deklarativní. Jelikož mi byl zápis gulpu bližší, zvolil jsem tento nástroj a připravil si strukturu projektu.

Před vývojem každého produktu je dobré stanovit si určitá pravidla a kontroly. U dynamicky typovaných jazyků toto platí dvojnásob, protože u staticky typovaných jazyků by k odhalení některých problémů pomohl alespoň kompilátor. Pro Javascript, jako pro Javu, existují nástroje statické analýzy kódu. Ty nám sice nenahradí typovou kontrolu, ale umí upozorňovat na věci, které by mohly v běhu přinášet problémy. Těmito problémy rozumějme například chybějící default větve ve switch, chybějící return po callbacku a další problémy obdobného ražení.

Pro statickou analýzu kódu existuje několik nástrojů. Mezi nejznámější a nejrozšířenější jsou řazeny ESLint, JSLint a JSHint. Já jsem se na základě [41] rozhodl pro použití ESLintu.

4.3 Facebook Graph API

Jelikož má být naše aplikace úzce spjatá s Facebookem, bylo nutné se s touto sociální sítí integrovat. Facebook pro tyto případy poskytuje otevřené rozhraní, které nazývá Graph API.

V jádru se jedná o standardní HTTP API [42], které pracuje na principu sociálního grafu. Ten obsahuje následující prvky.

- Uzly
  Facebook tento prvek popisuje doslova jako věc, kterou je například uživatel, komentář, notifikace.
- Hrany
  Prvky, které spojují uzly mezi sebou. Příkladem můžou být uživatelovy fotky.
- Vlastnosti
  Konkrétní vlastnosti daného uzlu, např. uživatelovo jméno.

Facebook používá pro autentifikaci uživatele princip access tokenů, kdy po úspěšném přihlášení dostane uživatel na určitou dobu přístupový token, který ho opravňuje k připojení na API. Servery díky tomu nemusí držet žádnou session na serveru a veškerou informaci dostanou v požadavku, který se může dostat na jakoukoliv instanci serveru. Ukládání session na serveru je náročné na prostředky, s kterými při velkém počtu uživatelů není radno plýtvat.

Jak bylo zmíněno, Graph API je standardní HTTP API, které je možné využívat v jakémkoliv prostředí, které umí vyvolat HTTP požadavek na server. Volání jednotlivých požadavků si však můžeme zpříjemnit již implementovanou knihovnou. Rozhodl jsem se

[1]) `http://gulpjs.com/`
[2]) `http://gruntjs.com/`

použít node.js knihovnu *fb*, uvolněnou pod Apache v2 licencí a umístěnou v repozitářích npm.

Obecně není žádoucí každému autentifikovanému uživateli poskytnout všechny informace. Proto Facebook používá v API systém oprávnění. Při přihlášení skrze Facebook na straně klienta je třeba požádat o všechna oprávnění nutná v naší aplikaci. Budeme žádat pouze o nejmenší nutnou množinu oprávnění, neb požadavek na oprávnění k něčemu, co viditelně nepotřebuje naše aplikace by mohlo v uživatelích vyvolat značnou nedůvěru.

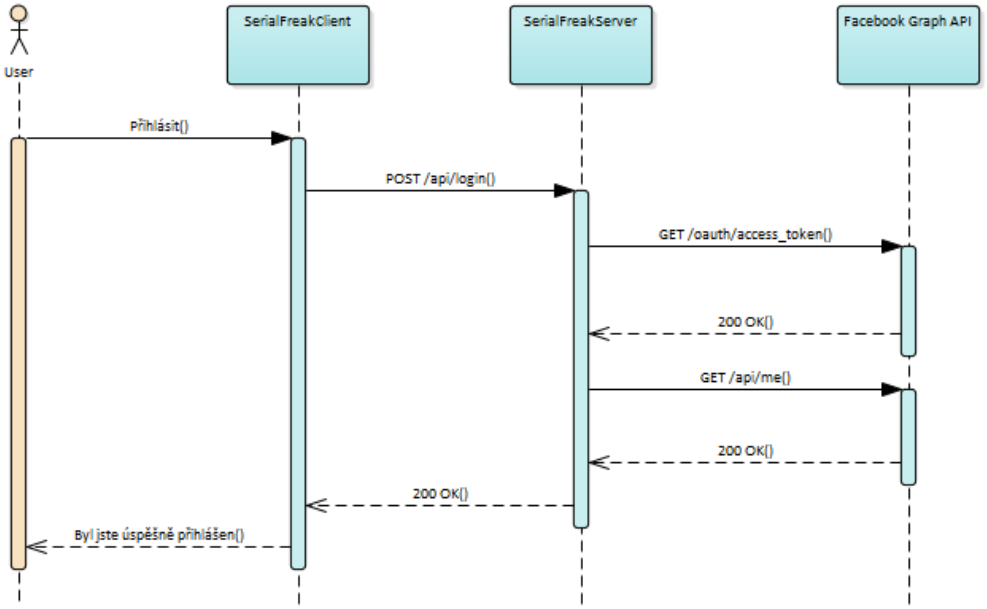
Z Facebooku budou získávány následující informace:

- Přihlášení skrze Facebook
  Nemusím se starat o správu účtů a pouze převezmu autentifikovaného uživatele pro naši aplikaci.
- Seznam přátel
  Naše aplikace se nemá pokoušet tvořit nová přátelství, která mají potencionální uživatelé již vytvořená na Facebooku. Proto jsme se rozhodl čerpat navázaná spojení z Graph API.
- Jméno a obrázek uživatele
  U uživatele chceme evidovat jeho obrázek a jméno. Tyto informace se budou získat z Graph APi.
- Přidání notifikace uživateli
  Uživatel bude dostávat zprávu o nové epizodě do notifikačního centra Facebooku, kde jistě bude trávit více času než na našem portále. Tímto docílíme zvýšení prokliků na naše stránky.

4.3.1 Přihlášení

Již bylo zmíněno, že pro volání API se ve Facebooku používá princip access tokenů. Tento token je nutné získat v prohlížeči a není možné si ho z principu vyžádat na serveru.

Server tedy bude implementovat zdroj pro přihlášení, který dostane token, získaný u uživatele v prohlížeči. Tento token si poté může ověřit a nadále s ním pracovat. Jak bude probíhat přihlášení, lépe uvidíme na následujícím sekvenčním diagramu 4.2



Obrázek 4.2. Sekvenční diagram přihlášení do aplikace SerialFreaku

Uživatel si otevře stránky SerialFreaku, kde dostane našeho webového klienta. Jestliže není přihlášen, bude mít možnost se přihlásit pomocí Facebooku.

Při přihlášení je požádán o výčet oprávnění pro naši aplikaci. Tento výčet obsahuje výchozí public_profile, email a user_friends.

Klient po úspěšném přihlášení přepošle access token na server, kde se ho následně server pokusí převést na dlouhotrvající access token, skrze který může server komunikovat s API po dobu 60 dní. Jestliže je převedení úspěšné, token se uloží do databáze uživatelů a je s ním zavoláno zjištění informací o uživateli. Při úspěchu obou operací se klientovi vrátí access token pro naší aplikaci a uživatel je úspěšně autentifikován. Implementace autentifikačního tokenu bude rozebrána níže 4.5.2.

4.3.2 Seznam přátel

SerialFreak využívá aktuální sociální vazby z Facebooku, proto na naší straně neudržujeme žádný seznam přátel daného uživatele. Tím máme zaručenou aktuálnost dat, bohužel zde narážíme na možné časté volání API. Proto je implementována časová mezipaměť, která slouží pro ukládání požadavku na přátele. Je to určitý kompromis mezi aktuálností dat a častým voláním API.

Volání seznamu přátel odstiňuje vrstva, která se stará o získání přátel z FB a jejich uložení do mezipaměti. Jako mezipaměť je zde použita NoSQL Key-value databáze Redis.

4.3.3 Notifikace uživateli

Server by se měl postarat o notifikování uživatele na novou epizodu seriálu, který sleduje. V ideálním případě jsme chtěli dosáhnout po kliknutí na notifikaci otevření stránek SerialFreaku.

Zde jsem nepředpokládal žádný problém a domníval jsem se, že stačí použít API a přidat konkrétnímu uživateli notifikaci. Byl jsem však vyveden z omylu, protože se jedná o jiný typ notifikací, takzvaných App-to-user.

Graph API je velmi dobře dokumentované a dle [43] se podařilo notifikace rychle zprovoznit. Problémem se však stalo otevření relevantního obsahu na našich stránkách.

Do notifikace je umožněno předat odkaz, tento odkaz je relativní v rámci nastavené url FB aplikace. Facebook se zachová tak, že po kliknutí na notifikaci dojde k otevření stránek Facebook aplikace, kde je embedována daná url složená z adresy aplikace a relativní adresy předané v notifikaci.

Celý problém se jeví relativně snadno a původní záměr byl odkázat na stránku seriálu. Bohužel je naše klientská aplikace routována dynamicky až v klientském prohlížeči a server tedy nebyl schopný vrátit validní požadavek. Nehledě na to, že požadavek na aplikaci se nevykonává standardní HTTP GET metodou, jak jsem si myslel, ale POST metodou. Přistoupili jsme tedy k řešení, kde je na serveru na tento POST požadavek vygenerována statická HTML stránka obsahující data o epizodě.

Naším primárním cílem notifikace bylo dostat uživatele na náš web, což jsme museli vyřešit tak, že v detailu epizody, který se zobrazuje na stránkách Facebooku, je odkaz otevření epizody v naší aplikaci. Zároveň jsme chtěli uživatelům nabídnout i funkcionalitu přímo na stránkách FB aplikace, proto je zde možnost epizodu rovnou vyhledat ve vyhledávači Google.

4.4 Zdroj dat

Zdrojem dat bylo v návrhové části vybráno TVDB. Pro přístup do API bylo třeba si zažádat o klíč, který nám byl následně udělen.

V prvotní fázi jsem se rozhodl použít knihovnu node-tvdb [1]). Která se zdála být v počátku bezproblémová, později jsem však narazil na problém s datovými typy, které knihovna vracela. Veškeré vrácené hodnoty byly reprezentovány řetězcem, což bylo způsobeno starou verzí rozhraní, která nedefinovala žádné schéma pro návratové XML.

Shodou okolností v době, kdy byl řešen problém, vydala databáze nové rozhraní V2. Toto je dobře dokumentované skrze Swagger 4.4.1, je možné si ho vyzkoušet skrze nástroj na webu a respektuje datové typy. Proto jsem neváhal a pustil se do implementace nové verze API postavené na Swaggeru.

4.4.1 Swagger

Dle oficiálních stránek [44] je Swagger open source framework s velkým ekosystémem, který slouží k návrhu, sestavení a dokumentování RESTful rozhraní.

Specifikace rozhraní se píše ve formátu JSON, případně ve formátu YAML. Pro psaní specifikace poskytuje webový i desktopový editor, ten umí definici validovat a upozorňovat na možné problémy. Zajímavou možností je vygenerování serverové i klientské části z definice. Generování je možné do značné množiny jazyků a frameworků od Javy, C# po Node.js, Javascript, Ruby atd..

Pro názornost se podíváme na klíčovou část YAML definice jednoduchého endpointu, který vrací obligátní ‘Hello world‘.

```
/greeting:
  get:
    summary: Hello world GET
    description: |
      Vrací pozdrav Hello World
    responses:
```

[1]) https://www.npmjs.com/package/node-tvdb

```
        200:
          description: Hello world pozdrav
          schema:
            type: object
            properties:
              greeting:
                type: string
          examples:
            application/json:
              greeting: Hello world
        default:
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
```

Na počátku je definován endpoint greeting, pod kterým je definována operace GET tohoto endpointu. Samotná definice má shrnutí a popis funkce, což slouží pro dokumentaci. Následně jsou definovány možné odpovědi, které jsou v tomto případě dvě. První je standardní korektní odpověď, která se má vrátit se stavovým kódem 200 a s popisujícím schématem. Následuje sekce examples, kde pro lepší dokumentaci můžeme uvést reálný příklad odpovědi. Výchozí odpověď definována vlastností default odkazuje na externí schéma definované v rámci definice rozhraní. Je tedy možné napsat souhrnně schéma všech odpovědí a požadavků a na ně potom pouze odkázat.

4.4.2 Klient

Javascript je dynamicky typovaný jazyk, a proto se přímo vybízí použít knihovnu, která vygeneruje za běhu Swagger klienta. Pro toto použití existuje knihovna s názvem swagger-client [45]. Podívejme se na příklad inicializace a použití knihovny proti definici TVDB.

```
new Swagger({
            url: 'https://api.thetvdb.com/swagger.json',
            usePromise: true
        }).then(function (client) {
         client.Series.get_series_id({ id: tvdbId, 'en'})
        })
```

V konstruktoru knihovny je předána definice API a parametr, který knihovně říká, že má použít promise místo callbacků. Po inicializaci můžeme již klienta použít. Ten obsahuje endpoint a nad ním operace. V tomto příkladu je endpoint Series a operace get_series_id. Název operace vychází z elementu operationId v definici, nebo se složí dle následujícího schématu. První blok je HTTP metoda a druhý je složen z cesty ke zdroji, kde / v cestě se překládá jako _ v názvu operace.

Pro použití API je třeba se identifikovat skrze zdroj */login*. Jestliže je autentifikace úspěšná, je vrácen JWT token, který se používá v následné komunikaci.

4.5 API

Návrh rozhraní serveru byl definován v 3.4. Následující text ukáže jak probíhala jeho implementace.

Pro implementaci RESTful API lze použít již existující knihovnu, která je schopná vyřešit za nás takzvané routování požadavků neboli předávání požadavků z dané url na

konkrétní funkci. Pro řešení těchto potřeb byl použit framework Express.js [46]. Tato knihovna se charakterizuje jako rychlý minimalistický webový framework pro Node.js.

Po inicializace zmíněné knihovny lze vytvořit velmi jednoduše zdroj zavoláním následujícího kousku kódu.

```
app.get('/', function (req, res) {
  res.send('Hello world')
})
```

Ten vytvoří zdroj na kořenu serveru, který reaguje na HTTP metodu GET, která vrací obligátní 'Hello world'. Toto řešení je však neelegantní, pokud definujeme více zdrojů, pro tyto případy existují ve frameworku takzvané Routery. Ty slouží k modularizaci celého API a fungují stejně jako definice standardní routy, včetně možnosti používat middleware funkce. Po definování tohoto routeru lze celý router vystavit následujícím kouskem kódu.

```
var temp = require('./temp');
app.use('/temp', temp);
```

Definice routeru se nachází v souboru temp.js, jehož instance je zde vyžádána a následně přiřazena hlavní express aplikaci.

Server obsahuje hlavní skript *App.js*, který spojuje dohromady všechny routery a vystaví je na rozhraní, jakmile dojde k inicializaci spojení do databáze. Všechny routery, které se používají, se nachází ve složce *api*. Dále zde existují přímo definované zdroje, jako je canvas pro Facebook, případně zdroj pro Let's encrypt [1]).

Při implementaci API bylo nutné vyřešit několik problémů. Mezi nimiž zmíním autentifikaci a validaci požadavků. Nejdříve si však ukážeme, čemu se říká middleware v prostředí express.js.

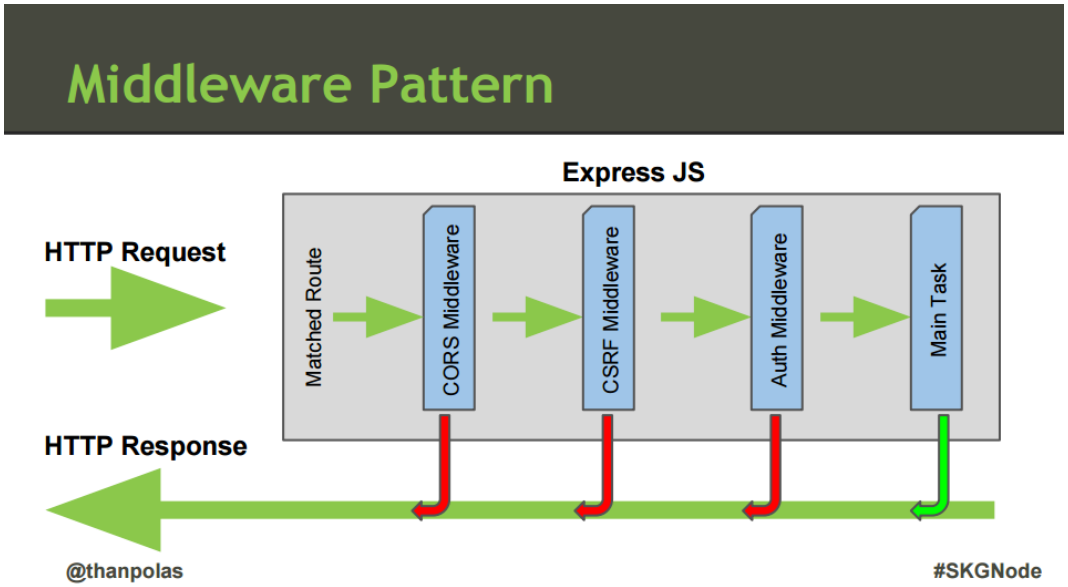
4.5.1 Middleware

Middleware je dle [47] funkce, která má přístup k objektu požadavku, objektu odpovědi a také má referenci na další middleware funkci. Tyto funkce mohou vykonávat následující úlohy:

- Spustit jakýkoliv kód
- Dělat změny v požadavku i odpovědi
- Ukončit cyklus požadavek - odpověď
- Zavolat následující middleware funkci.

Na následujícím obrázku si vysvětlíme jak tento vzor pracuje.

[1]) `https://letsencrypt.org/`



Obrázek 4.3. Middleware vzor používaný v express.js, převzato z [48].

Express.js zaregistruje požadavek, který se pokusí spojit se směrováním, jestliže takové nalezne, aplikuje se první middleware funkce v pořadí. V tomto případě se jedná o CORS Middleware, tato funkce provede operaci s požadavkem a zavolá další middleware, až po všech deklarovaných funkcích dojde požadavek k hlavní úloze tohoto směrování. Každá middleware funkce může požadavek ukončit a vrátit ihned odpověď, tedy se na další funkci v pořadí již nemusí dostat. Představme si příklad autentifikační middleware funkce, která validuje, zda je uživatel přihlášený. V případě, že validace proběhne korektně, zavolá další funkci v pořadí, jestliže ne, ukončí procesování požadavku a vrátí HTTP status 401.

Při používání middleware je třeba být obezřetný a zavolat následující funkci, jinak by se mohl požadavek 'ztratit'.

4.5.2 Autentifikace

Ani námi vystavované API není možné využívat bez autentifikace uživatele. Postup autentifikace je vidět na sekvenčním diagramu 4.2, zde se podíváme na implementaci access tokenu a ověřování přihlášeného uživatele.

Jak již bylo zmíněno, uživatel dostane po úspěšném přihlášení access token, s kterým poté přistupuje na server. Jedná se o JWT neboli JSON Web Token.

JWT [49] je otevřený standard (RFC 7519 [1])), který definuje kompaktní cestu, jak přenést data mezi účastníky v JSON formátu. Tyto informace jsou důvěryhodné a verifikovatelné, protože jsou digitálně podepsány některým z algoritmů. Jelikož je JWT kompaktní, chápejme úsporné, je možné ho snadno předávat v každém requestu, ať již v hlavičce, těle, nebo dotazu.

Web token má 3 hlavní části, které jsou odděleny tečkou. Tyto části jsou:

- Hlavička

  Ta se skládá ze dvou částí, kterými jsou typ a hashovací algoritmus. Celá tato část je zakódována pomocí base64.
- Obsah

[1]) `https://tools.ietf.org/html/rfc7519`

Obsah se skládá z takzvaných tvrzení, což může být entita uživatele a další potřebná metadata. Výsledný obsah je zakódován pomocí base64.

- Podpis

  K vytvoření podpisu potřebujeme obě zakódované předešlé části, tajný klíč a algoritmus specifikovaný v hlavičce. Tímto algoritmem se vytvoří podpis, který se připojí do výsledného tokenu. Ten slouží pro ověření integrity a důvěryhodnosti tokenu.

Pro implementaci tohoto ověřování jsem použil knihovnu *jsonwebtoken* [1]), s využitím výchozího algoritmu HMAC SHA 256.

Po úspěšné autentifikaci uživatele je nutné pro všechny zabezpečené zdroje odesílat vrácený access token na server. Dít se tak může v těle, hlavičce, nebo dotazu. Pro ověřování autentifikovaného uživatele je napsán jednoduchý middleware, který ověří platnost tokenu a přepošle požadavek dál, nebo vrátí odpověď 401 a k dalšímu zpracování požadavku nedojde.

```
// middleware pro kontrolu prihlaseneho uzivatele
  router.use(function (req, res, next) {
    // zkontroluji zda je token v tele, query, nebo hlavicce
    var token = req.body.token || req.query.token
     || req.headers['x-access-token'];
    // jestlize existuje
    if (token) {
      // verifikuji jeho platnost, vcetne expirace
      jwt.verify(token, app.get('secretKey'), function (err, decoded) {
        if (err) {
          return res.status(401).json({ success: false,
           message: 'Failed to authenticate token.' });
        } else {
          // vsechno je v poradku, ulozim dekodovaneho
          // uzivatele do requestu pro dalsi zpracovavani
          req.decoded = decoded;
          //preposli na dalsi middleware
          next();
        }
      });
    } else {
      // jestlize neni poskytnut zadny token, vrat 403
      return res.status(403).send({
        success: false,
        message: 'No token provided.'
      });
    }
  });
```

4.5.3 Validace

Pro klíčové, převážně neidempotentní požadavky je vhodné nasadit validaci požadavků, aby se do systému nezanášely neplatná data, která by mohla přinášet neočekávané chování programu. Na úrovní API jsem se rozhodl některé požadavky validovat skrze validátor *joi* [50].

Skrze joi je možné definovat schéma objektů předávaných v požadavcích a tato schémata následně použít pro již zmíněnou validaci.

[1]) `https://www.npmjs.com/package/jsonwebtoken`

Pro validaci je vytvořený jednoduchý middleware, který se postará o kontrolu a případně pošle požadavek dále. Na vzorovém příkladu přidání doporučení na seriál se podíváme na konkrétní zjednodušenou implementaci.

```
var Joi=require('joi');

module.exports=Joi.object().keys({
    serialId:Joi.number().positive().min(1).required()
});
```

Zmíněná definice nám říká, že se jedná o objekt, který má pole vlastností, které se v tomto případě skládá pouze z jediného serialId. Tato vlastnost musí být číslo, které je pozitivní, má minimální hodnotu 1 a je také striktně vyžadována pro přijmutí.

```
var Joi=require('joi');
module.exports=function validating(schema){
    return function(req,resp,next){
        Joi.validate(req.body,schema,{allowUnknown:true},function(err){
            if(err){
                err.status=400;
                next(err);}
            else next();
        });
    }
};
```

Exportovaná funkce má jako parametr schéma validovaného objektu a vrací middleware funkci, která již validuje příslušný objekt s předaným schématem. Jestliže není validace úspěšná předá, řízení chybovému middlewaru, jinak přejde funkce na další běžný middleware.

Na samotné použití těchto dvou definic se podívejme v následujícím kódu.

```
router.post('/users/:id/recommendation',
                    validator(schemas.recommendation),
                    function(req,resp, next){
//kod pro zpracovani doporuceni
}
```

Na úrovni zdroje je použit middleware, tedy daná validační funkce je aplikována pouze na cestu /users/:id/recommendation a HTTP metodu POST. Validátoru je předáno schéma, které jsme si nadefinovali výše a na základě kterého bude každý požadavek validován.

4.6 Datová vrstva

Pro ukládání dat byla zvolena NoSQL databáze, logickým krokem tedy bylo porozhlédnout se po variantě v rámci Azure cloudu. Azure nabízel z počátku vyhovující Azure DocumentDB [29].

Základním problémem bylo začít přemýšlet jinak, než je tomu ve světě relačních databází. K pochopení základních principů, jak navrhovat NoSQL databáze, mi posloužil článek NoSQL Data Modeling Techniques [51].

Modelování relační databáze vychází ze struktury dat a její návrh lze charakterizovat otázkou 'Jaké odpovědi máme ? '. NoSQL databáze modelujeme naopak dle přístupu k datům a modelování můžeme charakterizovat otázkou 'Jaké dotazy máme ? '.

Ve světě relačních databází se snažíme databázi co nejvíce normalizovat a vyvarovat se duplicitě dat. Ve světě NoSQL tomu je naopak, zde se snažíme data v podstatě denormalizovat a vytvořit co nejvíce ploché kolekce dat. Ploché rozumějme, že objekty v ní uložené neobsahují příliš mnoho do sebe zanořených objektů a polí.

Denormalizaci provádíme z důvodu nemožnosti spojovat data mezi kolekcemi, proto je nutné mít téměř všechna potřebná data uložená v rámci jednoho dokumentu. To samozřejmě přináší problémy s duplicitou dat, které musíme následně řešit aplikačně. Přínos tohoto řešení je při čtení, kdy není nutné dotazovat ještě jinou kolekci a na jeden jednoduchý dotaz jsme schopni dostat potenciálně veškerá potřebná data.

Na následujícím příkladu se podíváme, jak vypadají konkrétní data uložená v relační databázi a jak v NoSQL dokumentové databázi. Jako příklad si vezmeme uživatele e-shopu a jeho dodací adresy.

Při modelování relační databáze bychom se snažili nalézt jednotlivé nedělitelné entity. V tomto případě za tyto entity považujme entitu User a Address. Entity by vypadaly v relačních tabulkách následovně.

id	email	password	name
1	josef@novak.com	maruska	Josef Novák

Tabulka 4.1. Tabulka User

id	userId	address	city	phone	zip
1	1	Václavské náměstí	Praha 1	123456789	100 00
1	1	Staroměstské náměstí	Praha 1	987654321	100 00

Tabulka 4.2. Tabulka Address

V případě použití dokumentové databáze bychom zachovali uživatele v jednom dokumentu formátu JSON. Adresy by byly obsaženy přímo v něm formou pole adres.

```
{
  "id": 1,
  "email": "josef@novak.com",
  "password": "maruska",
  "name": "Josef Novák",
  "address": [
    {
      "id": 1,
      "userId": 1,
      "address": "Václavské náměstí",
      "city": "Praha 1",
      "phone": 123456789,
      "zip": 10000
    },
    {
      "id": 2,
      "userId": 1,
      "address": "Staroměstské náměstí",
      "city": "Praha 1",
      "phone": 987654321,
      "zip": 10000
```

```
      }
    ]
  }
```

4.6.1 Azure DocumentDB

DocumentDB, jak již název napovídá, je implementace dokumentové NoSQL databáze v rámci cloudového prostředí Azure a Microsoft ji poskytuje formou služby. Tato databáze používá pro dotazování podmnožinu SQL a nabízí SDK pro Javu, .Net, Python i Node.js [52], případně je možné využít REST API. Knihovna pro Node.js je dobře dokumentována a použití této databáze se zdálo z počátku snadné.

Datový model se s postupem času vyvíjel na základě implementovaných funkčních požadavků. Nedošlo tedy k žádnému prvotnímu návrhu kolekcí, ale bylo využito přednosti, který nám tento typ databází přináší, a to dynamického schématu.

Pro přístup do databáze byla napsána DAO vrstva, přes kterou budou procházet všechny požadavky na databázi a umožní nám případnou relativně snadnou změnu databáze.

Po implementaci jádra serveru, byl však zjištěn problém v použití zmiňované DocumentDB. Tato databáze v době mého použití neumožňovala řadit data na databázi, proto se implementace stránkování zdála v některých částech značně komplikovaná a pochyby o použití této databáze se stupňovaly. Posledním zjištěním, které přispělo ke změně databáze, byla má chyba, kdy jsem při analýze špatně stanovil cenu za tuto databázi.

Při vytváření kolekcí jsem použil nejnižší možnou cenovou úroveň, což byla uživatelsky definovaná úroveň s 400 RU, kde se cena pohybovala kolem 24 Euro za měsíc. V domnění, že tato cena se vztahuje na celou databázi, a ne jednotlivé kolekce, jsem narazil na finanční limity našeho účtu a bylo nutné udělat změnu.

4.6.2 MongoDB

Vedoucí práce mi doporučil open source dokumentovou databázi MongoDB [53]. Tato databáze byla vytvořena v roce 2007 a pod open source vývojový model přešla až v roce 2009. Databáze nabízí oficiální ovladače pro značnou množinu jazyků, mezi nimiž je i Node.js.

Data si vnitřně ukládá ve formátu BSON, což je binárně serializovatelný JSON používaný pro ukládání dokumentů. Tento formát podporuje více datových typů nežli JSON a je tak jeho nadmnožinou. Jedním z používaných datových typů je pro Mongo důležitý datový typ ObjectId. Objekty tohoto typu by měly být unikátní, seřaditelné a snadné na vygenerování. V Mongu musí každý ukládaný dokument obsahovat atribut _id, které je právě tohoto typu a slouží jako jednoznačný identifikátor dokumentu. Pokud toto pole není při ukládání předáno, ovladač automaticky toto pole vygeneruje a přiřadí.

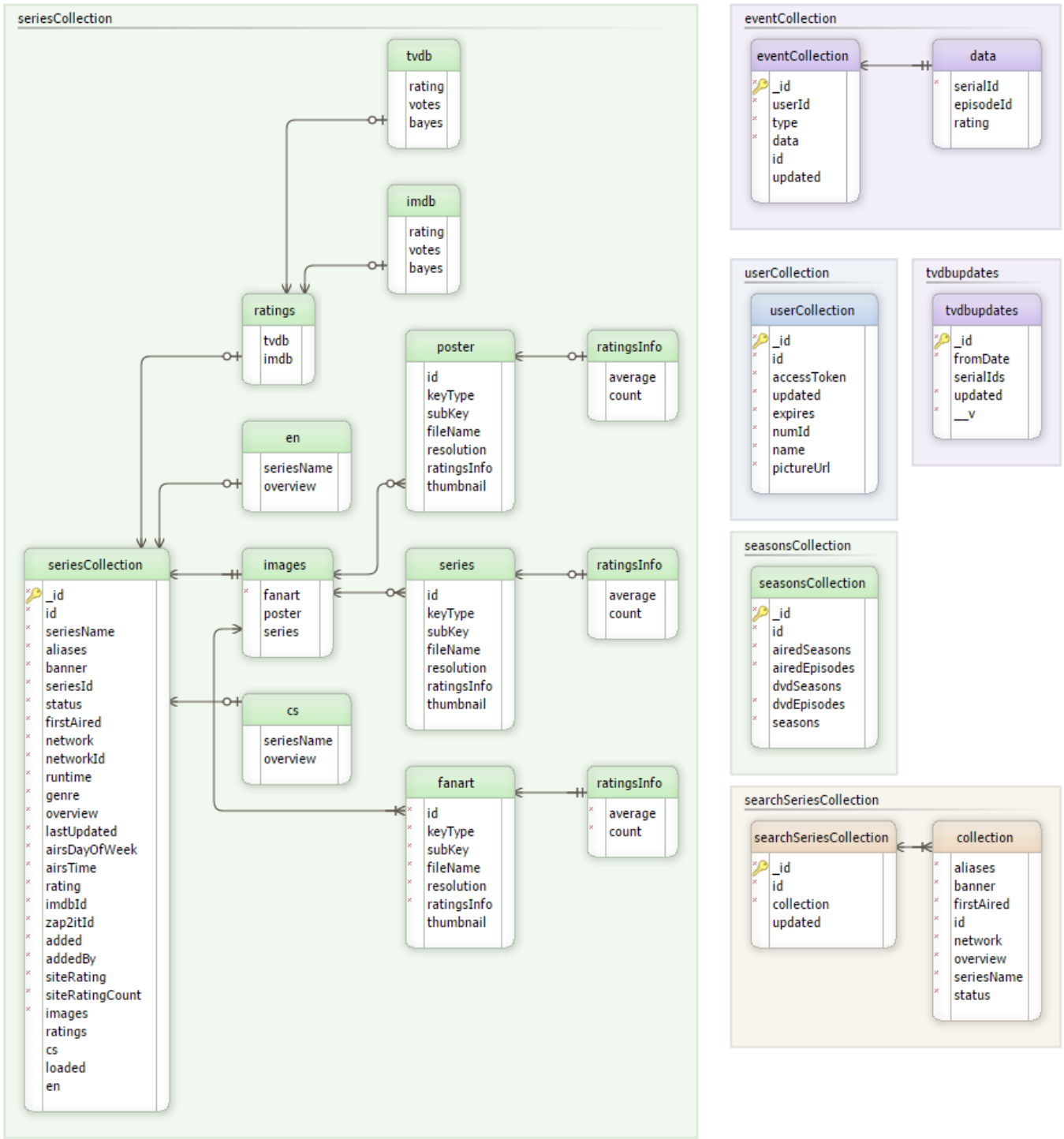
Klíčové vlastnosti této databáze jsou:

- Vysoký výkon
  Díky možnosti vkládat komplexnější struktury včetně polí a pod dokumentů je dosaženo menšího počtu volání databáze.
- Rich Query Language
  Mongo podporuje dotazovací jazyk pro standardní CRUD operace, ale i pro textové vyhledávání a geografické dotazy.
- Vysoká dostupnost

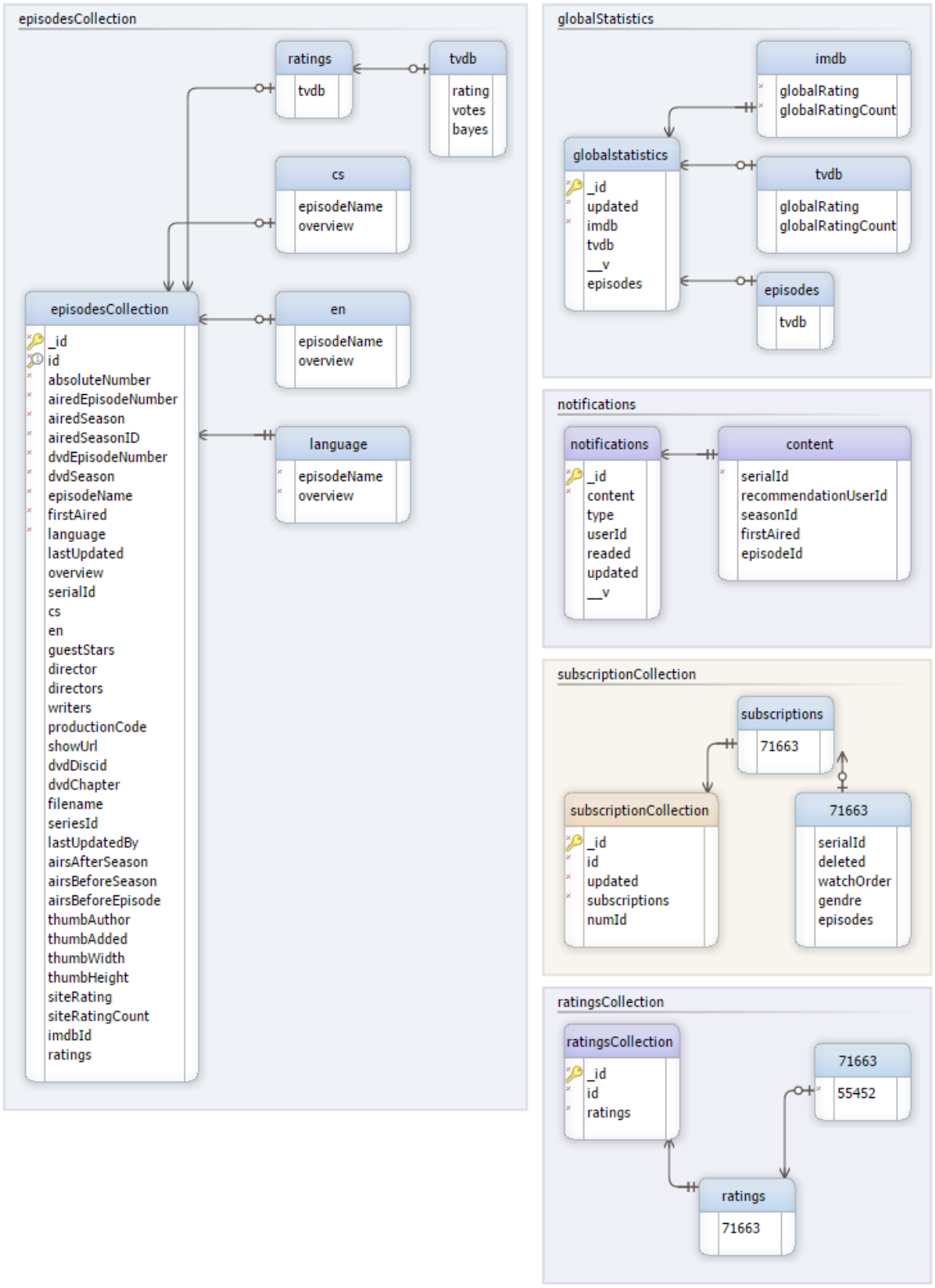
Databázi lze spustit tak, že se replikuje na několik uzlů, čímž je dosaženo redundance dat a ochrany proti selhání.

- Horizontální škálovatelnost
  Mongo podporuje sharding a data lze distribuovat na různé výpočetní uzly.

Implementace datové vrstvy nad MongoDB nepřinesla žádné větší problémy a subjektivně se mi s ní pracovalo lépe než s DocumentDB, možná i proto, že poskytuje lepší dokumentaci a je více využívaná ze strany vývojářů. Nyní se podívejme na výslednou strukturu celé databáze:



Obrázek 4.4. Schéma databáze 1



Obrázek 4.5. Schéma databáze 2

S vizualizací datového modelu byl obecně problém a většina nástrojů není na NoSQL databáze připravena, pokusil jsem se kolekce vizualizovat ER modelem, přestože není pro toto použití zcela vhodný. Popíši zde dvě kolekce, jejichž vizualizace v modelu není názorná. Bude se jednat o kolekci *subscriptionCollection* a *ratingsCollection*.

SubscriptionCollection je kolekce, která ukládá odebírané seriály každého uživatele. Dokument v ní uložený má identifikátor *id*, který je shodný s identifikátorem uživatele. Tento dokument obsahuje poddokument, který má vlastnost s názvem reprezentujícím identifikátor seriálu. Ten obsahuje další poddokument reprezentující stav uživatele k odebíranému seriálu a vlastnost *episodes*, což je kolekce epizod daného seriálu a uživatelova stavu k nim.

RatingCollection reprezentuje hodnocení epizod seriálů pro daného uživatele, kde id této kolekce je opět identifikátor uživatele. Tato reprezentace obsahuje podobjekt ratings symbolizující seriál a v něm další podobjekt jež obsahuje hodnocení epizod pro konkrétního uživatele.

4.6.3 Mongoose

Na závěr této sekce bych si ještě dovolil malou odbočku k mongoose[54], což je modul, který nám usnadní přístup k dokumentům uloženým v databázi a nabídne nám podobnou funkcionalitu, kterou známe ze známějších ORM frameworků (Hibernate). Ve světě dokumentových databází jsou tyto frameworky označovány zkratkou ODM, neboli objektově dokumentové mapování.

Tento modul se nevyužívá v celé datové vrstvě, ale pouze v minoritní části, kde není obsáhlé schéma dokumentů. Skrze tento modul si můžeme nadefinovat schéma požadovaného objektu, jak můžeme vidět na následujícím příkladu notifikací:

```
var Notification = new mongoose.Schema({
    userId: {type:String, required:true}
  , updated: { type: Date, default: Date.now }
  , type: {type:String, enum:['NEW_EPISODE', 'RECOMMENDATION']}
  , content:mongoose.Schema.Types.Mixed
  , readed: {type:Boolean, default:false}
  , _self: {type:String, required:false}
});
```

Pomocí tohoto ODM si definujeme datové typy, vyžadování vlastnosti, výchozí hodnotu atd. Jednoduchým použitím tohoto schématu si poté dokážeme vytáhnout z databáze všechny notifikace daného uživatele.

```
Notification.find({userId: userId}).sort({updated:-1});
```

Čtenář se může ptát, proč nebylo použito ODM pro celou datovou část aplikace. Důvodem nepoužití v majoritní části datové vrstvy je zbytečná komplikace s definicí modelu pro větší dokumenty, stejně tak úzká závislost na MongoDB, které jsem se chtěl vyvarovat. Přidaná hodnota ve snadnějším získávání dokumentů není znatelná, protože i získání skrze nativní ovladač je snadné, jak můžeme vidět v následujícím kódu pro získání notifikací.

```
mongo.db.collection('notifications').find({userId:userId})
   .sort({updated:-1});
```

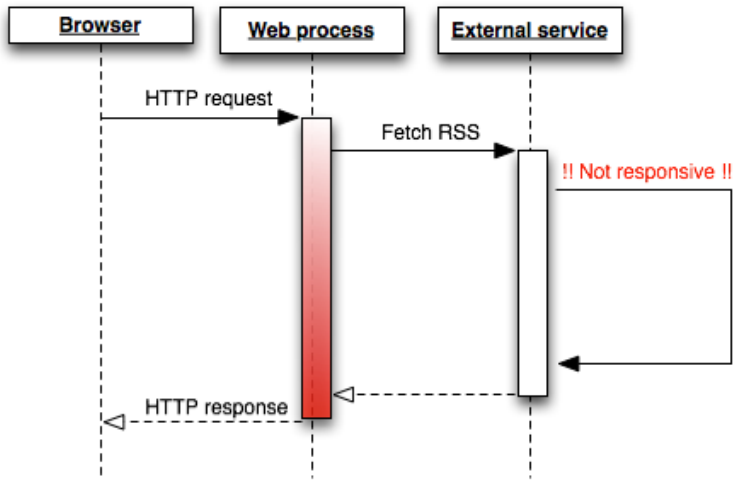
4.7 Loadery

Protože naše aplikace nedokáže pracovat pouze s daty online, ale provádí nad nimi i další operace, je nutné je uchovávat v naší databázi.

Načítání dat z externího zdroje se dělo v první fázi synchronně s požadavkem na nahrání, kdy se na základě události tento proces spustil. Toto řešení bylo nevhodné, protože se vytěžoval hlavní proces serveru, poskytovatel dat byl doslova bombardován požadavky na data a nedokázali jsme snadno regulovat maximální počet požadavků na poskytovatele. Proto jsem nakonec přistoupil k řešení, že se data budou načítat asynchronně, a byly pro ně vytvořeny takzvané Loadery.

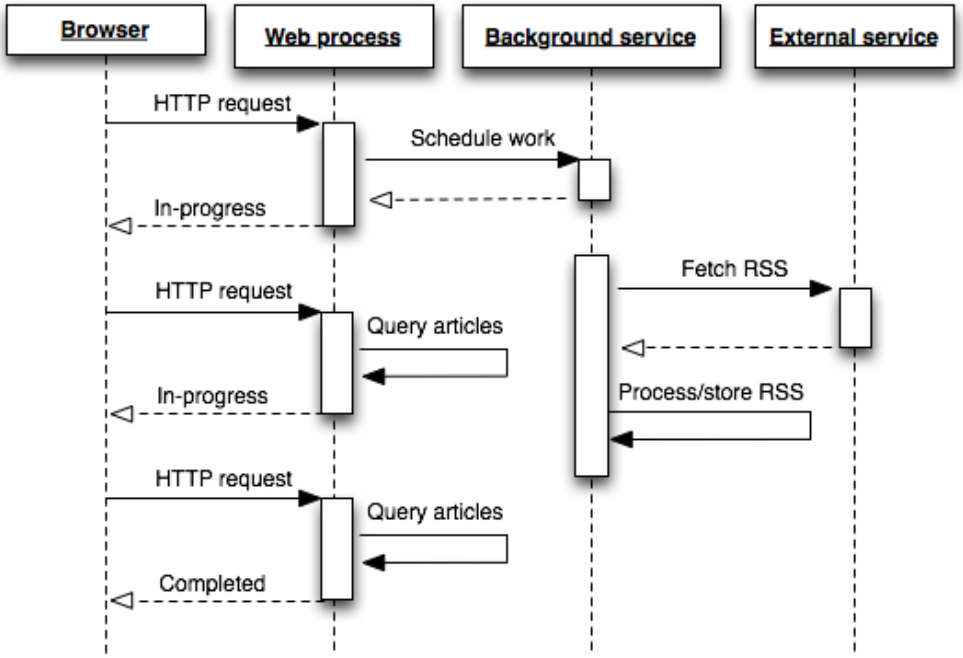
4.7.1 Architektura

Abychom dokázali data zpracovávat asynchronně a toto zpracování bylo transparentní a snadno škálovatelné, je vhodné použít princip takzvaných Worker queue [55]. Základním problémem synchronního zpracování požadavků je nemožnost ovlivnit čas odpovědi ze serveru třetí strany, tato odpověď může být okamžitá, nebo trvat několik minut a takovou dobu nechceme uživatele nechat čekat. Na následujícím sekvenčním diagramu se podívejme na špatné řešení tohoto problému.



Obrázek 4.6. Sekvenční diagram špatného přístupu, převzato z [55]

Na diagramu 4.6 vidíme, že došlo k zavolání hlavního webového procesu a ten by měl co nejrychleji vrátit odpověď, bohužel však tento proces závisí na externí službě, která je aktuálně mimo provoz. Jelikož je zpracování synchronní s odpovědí, budeme čekat do doby, než vyprší časový limit volání a až poté dostane uživatel odpověď. Tato implementace je špatná, protože klient nedostane po určitou dobu žádnou odezvu. Nyní se podívejme na přístup druhý, který využívá asynchronního zpracování.



Obrázek 4.7. Sekvenční diagram správného přístupu, převzato z [55]

Tento postup 4.7 můžeme v tomto případě považovat za správný. Klient zavolá požadavek, server naplánuje jeho zpracování na pozadí a dá ihned informaci klientovi o tom, že data budou dostupná později. Na pozadí se data v nejbližší možnou chvíli načtou, nezávisle na hlavním procesu serveru. Klient se neustále dotazuje na stav dat a uživatel tak dostává zpětnou vazbu o tom, že jsou data nahrávána. Po úspěšném nahrání jsou klientovi vrácena nahraná data.

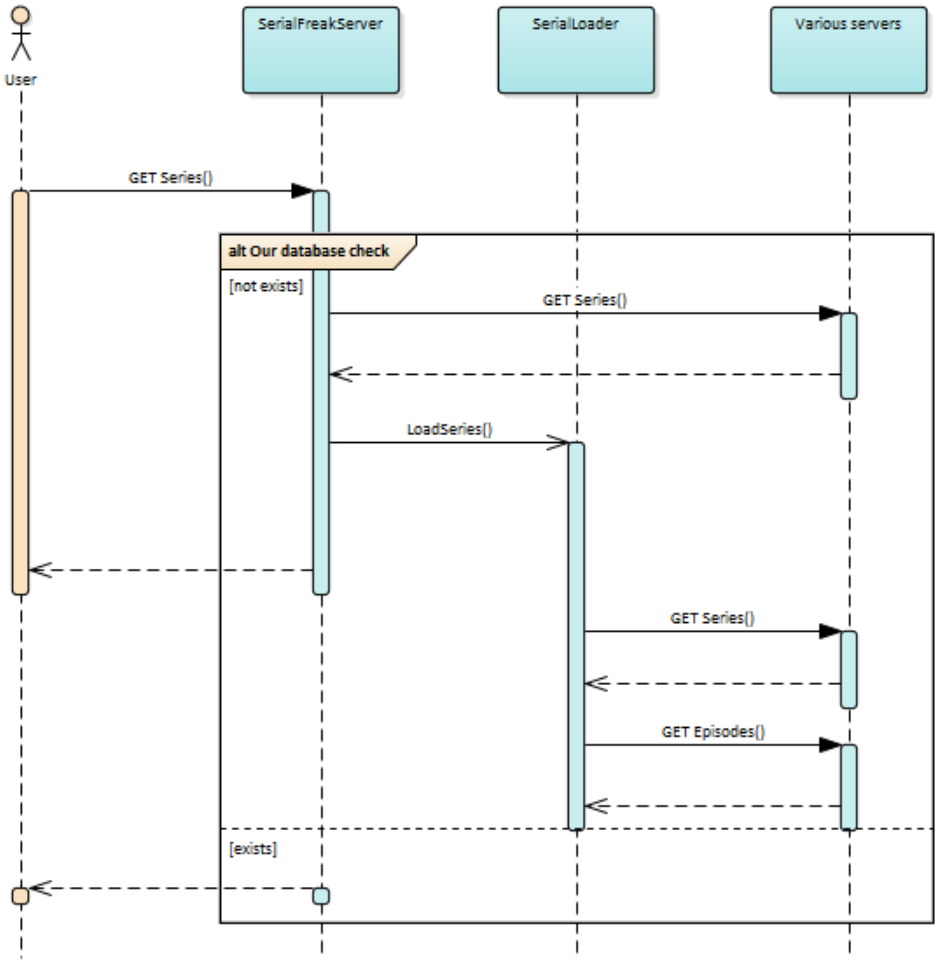
Pro implementaci tohoto postupu se používá princip front. Do fronty jsou klientem přidávány zprávy neboli úlohy pro zpracování. Na druhém konci fronty je libovolný počet procesů, který odebírá jednotlivé úlohy z fronty a zpracovává je na pozadí.

4.7.2 Nahrávání seriálů

Pro ilustraci použití front v naší aplikaci jsem vybral nahrávání seriálů do naší databáze. V tomto procesu je třeba nahrát následující data.

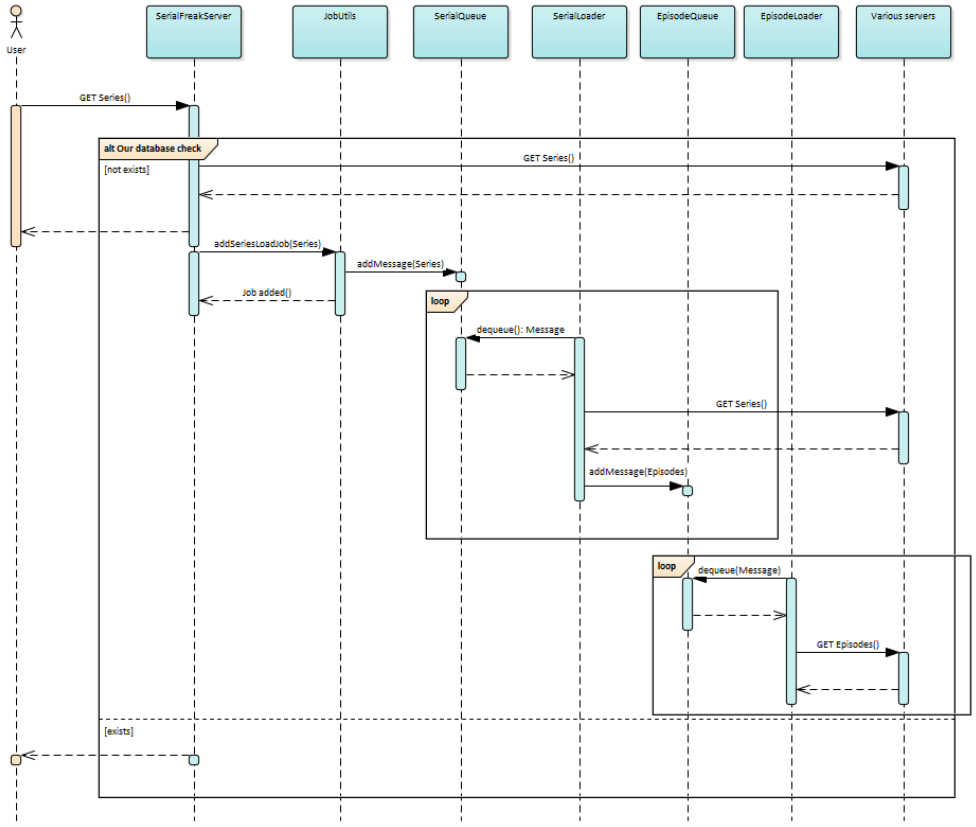
- Seriál
- Obrázky seriálů
- Epizody
- Obrázky epizod
- Hodnocení seriálů z IMDB

Na původní podobu zpracování se podíváme na následujícím sekvenčním diagramu. Pro přehlednost je zde zobrazeno pouze načítání seriálu a epizod.



Obrázek 4.8. Sekvenční diagram původního načítání seriálů.

Pro zpracování požadavku na seriál existují na serveru dvě disjunktní větve, které se dělí v závislosti na tom, zda seriál již máme, či nikoli. Pokud seriál máme, je uživateli rovnou vrácena informace o seriálu, jestliže však tuto informaci nemáme, zeptáme se na ni rovnou externího zdroje dat a vrátíme ji také rovnou uživateli. Na pozadí, avšak v hlavním procesu serveru se ihned spustí požadavek na nahrání dat do naší databáze, který provede potřebné dotazy a data uloží do databáze. Toto řešení naráželo na problémy s regulací i správou úloh pro nahrávání, bylo neprůhledné a nespolehlivé. Proto došlo k přepracování na asynchronní zpracování skrze fronty a workery. Výslednou podobu, opět pouze s nahráváním seriálů a epizod, vidíme na diagramu 4.9.



Obrázek 4.9. Sekvenční diagram aktuálního načítání seriálů.

V aktuálním způsobu zpracování je úloha dekomponována na dílčí části tak, že nahrávání epizod má na starosti jiný proces a fronta než nahrávání seriálu. Díky tomu je možné jednotlivé úlohy lépe přizpůsobovat. Na diagramu 4.9 již vidíme frontu *SerialQueue* pro nahrání seriálu a frontu *EpisodeQueue* pro epizody. Každý z workerů (SerialLoader, EpisodeLoader) si intervalově kontroluje, jestli se ve frontě nenachází úloha ke zpracování, jestliže ano, vezme si jí a zpracuje. Tímto intervalem kontroly také dokážeme regulovat počet požadavků volaných na externího databázi. Všímavý čtenář jistě zaregistroval, že při nedostupnosti seriálu v naší databázi je stejně nejdříve dotázána externí služba. Prozatím tomu tak je, protože jsme bohužel již nestihli aplikaci připravit na dotazování o existenci dat.

Pro ukládání zpráv jsem se rozhodl použít Azure Queues, které jsou obsaženy v rámci služby Azure Storage. Ty mají jisté nevýhody vůči komplexnější Azure Service Bus Queues [56], avšak nabízí příznivější cenu. Mezi nejvýraznější patří nepodpora Push API a detekce duplicit. Nevýhody se mi podařilo nakonec potlačit a nebylo nutné použít Service Bus Queues.

4.7.3 Azure Functions

Azure Functions je dle [57] služba, která umožňuje spouštět funkce neboli malé kousky kódu v cloudu. Jedná se čistě o funkční kód, bez jakéhokoliv nastavování, který reaguje na vstupní událost, která může být vyvolána z mnoha zdrojů, jako jsou fronty, tabulky, ale například i Dropboxu. Výborně se hodí pro zpracování dat, což je i náš případ užití. Funkci je možné napsat, otestovat a nasadit přímo ve webovém prohlížeči na portálu Azure. Výhodou těchto funkcí je, že jejich prostředí běží pouze po dobu běhu funkce a spustí se v závislosti na nastavené aktivační události, nemusíme tedy platit za

prostředí, které se nevyužívá. Jazykem implementace může být Javascript, proto jsem službu vyzkoušel.

Vhodným kandidátem se zdál být loader, který interaguje pouze s nativními komponentami v MS Azure, takovým je ImageLoader, který stahuje obrázky a ukládá je do Azure Blob Storage. Blob Storage [58] je úložiště velkých objemů nestrukturovaných (binárních) dat. Tato data mohou být následně dostupná veřejně skrze standardní HTTP protokol, z toho důvodu jsem ho vybral pro ukládání obrázků v naší aplikaci.

Ve funkcích je možné používat standardní Node.js knihovny, proto implementace stahování obrázků byla snadná. Pro představu zde ukáži kostru této funkce.

```
var http = require('http');
module.exports = function (context, myQueueItem) {

   //přečti si požadavek z myQueueItem
   //požadovanou URL načti do bytového pole

   var buffer = Buffer.concat(data);
   context.bindings.outputBlob = buffer;
   context.log('Image ' + name + ' is stored.');
   context.done();
}
```

Pro funkce je důležitý předaný objekt *context*, který v sobě má metodu pro logování, namapované výstupy(bindings) a také metodu done(), která celý běh funkce skončí. Při použití těchto funkcí se nemusíme starat o to, jak získat data z fronty, nebo jak je uložit do blobu. Reference na vstupy i výstupy dostaneme předané z konfigurace a naší starostí je pouze výkonný kód. Případná změna aktivační události nebo výstupu je tedy pouhá formalita a otázka konfigurace.

4.8 Aplikační logika

Aplikace z převážné většiny integruje data z několika zdrojů dohromady. Je zde však několik málo procesů, které stojí za zmínku a týkají se převážně řazení seriálů, jejich doporučování a doporučování epizod. Protože hlavní myšlenkou SerialFreaku je dávat uživateli zajímavé výsledky ke sledování, jsou tyto procesy pro práci aplikace důležité. Nebylo jim však věnováno tolik času, kolik by bylo žádoucí a do budoucna by se měly podrobit hlubší analýze.

4.8.1 Náhodná epizoda ke zhlédnutí

Pokud uživatel sleduje seriál náhodně, chceme uživateli doporučit tu nejlepší nezhlédnutou epizodu na základě hodnocení jeho přátel a hodnocení epizody z TVDB.

Shrňme si tedy, jaká data máme a jaká data vstoupí do tohoto výpočtu.

- Hodnocení epizody přáteli na SerialFreaku
- Zhlédnutí epizody přáteli na SerialFreaku
- Hodnocení epizody na TVDB
- Počet hodnotících epizody na TVDB

Hodnocení v naší aplikaci má v podstatě tři stavy: líbí, nelíbí a nehodnotil, databázově reprezentované 1,0 a undefined. Hodnocení na TVDB je reprezentováno na škále 0-10, přičemž je známa i informace o počtu hodnotících. Tyto dvě zdánlivě nesourodé

škály musíme spojit dohromady, abychom dokázali následně přiřadit každé epizodě číselné hodnocení, na základě kterého můžeme epizody seřadit.Výpočet si rozdělme na dvě disjunktní části, které bude potřeba na konci nějakým způsobem spojit. První část se týká hodnocení uživatelů na našem portálu a druhá hodnocení z TVDB.

Výsledné řešení by mělo být takové, aby zohledňovalo celkový počet hodnocení. Z pohledu uživatele je totiž prioritnější epizoda, která má 10x hodnocení líbí a 1x nelíbí, nežli epizoda, která má 1x líbí.

Mé úsilí jsem vynaložil na hledání řešení pro obdobný problém. Za nedlouho jsem narazil na článek[59] popisující hodnotící systém komentářů na portálu Reddit [1]) a článek [60], který popisuje téměř to samé. Oba dva se zabývají společným problémem, tedy jak seřadit data, u kterých se hlasuje binárním způsobem. Oba dva články na tento typ hodnocení doporučují stejný postup, a to použít dolní mez Wilsonova hodnocení. Toto hodnocení je definováno následujícím vzorcem 4.10.

$$\left(\hat{p} + \frac{z^2_{\alpha/2}}{2n} \pm z_{\alpha/2}\sqrt{[\hat{p}(1-\hat{p}) + z^2_{\alpha/2}/4n]/n}\right)/(1 + z^2_{\alpha/2}/n).$$

Obrázek 4.10. Definice Wilson hodnocení, převzato z [60].

Kde p je sledovaným podílem pozitivních hodnocení, n počtem všech hodnocení. Proměnná z je $(1 - \alpha/2)$ a udává kvantil[61] standardizovaného normálního rozdělení.

V NPM existuje knihovna[62], která nám celý tento výpočet usnadní. Jejími parametry jsou počet pozitivních hodnocení, počet celkových hodnocení a míra důvěryhodnosti. Ta symbolizuje, s jakou pravděpodobností je spodní odhad validní. Výsledek dolní meze se nachází převážně v intervalu 0-1, avšak může klesat až k zápornému nekonečnu.

Druhou částí hodnocení je hodnocení z TVDB. To je symbolizováno číslem na škále 0-10 a počtem hodnotících. Zde hledání směřovalo k již existujícím filmovým databázím a žebříčku nejlepších filmů. IMDb na svých stránkách uvádí [63] výpočet žebříčku 250 nejlepších filmů a vzhledem k tomu, že využívá stejný typ hodnocení jako TVDB rozhodl jsem se toto hodnocení vyzkoušet.

$$W = (Rv + Cm)/(v + m)$$

Vzorec pro výpočet je založen [64] na Bayesových odhadech. Nyní si popišme jednotlivé proměnné vzorce pro výpočet hodnocení epizod.

- W - Výsledné vážené hodnocení
- R - Průměrné hodnocení epizody
- v - Počet hodnotících
- m - Minimální počet hlasujících
- C - Průměrné hodnocení napříč všemi epizodami daného seriálu.

Tímto výpočtem se dostaneme k váženému hodnocení každé epizody, které se nachází na škále 0-10. Nyní nastává problém, jak tato dvě hodnocení spojit. Prozatím jsem se přiklonil k triviálnímu řešení, které hodnocení z TVDB převede také na škálu 0-1 a následně vypočte aritmetický průměr z těchto dvou hodnocení. Výsledky podávané pomocí tohoto spojení jsou pro současné použití dostatečné.

Hodnocení jednotlivých epizod pro každého uživatele je předpočítáváno asynchronně na pozadí. To se děje při každé změně hodnocení a zhlédnutí epizod v první úrovni sociálních vazeb uživatele, jinak řečeno, jestliže přítel uživatele zhlédne epizodu, která se mu líbila, dojde k přepočtení hodnocení epizod uživatele samotného.

[1]) `http://www.reddit.com/`

Kapitola 5
Testování

Důležitou součástí každého vývoje softwaru by mělo být testování. Testováním můžeme odhalit mnoho problémů a chyb, které při vývoji vznikají. Nelze odhalit chyby všechny, ale lze jejich počet snížit na minimum.

Testování rozdělím na dvě části a to na funkční a nefunkční [65]. Z pohledu mé práce, která má být snadno horizontálně škálovatelná, jsou důležité obzvlášť nefunkční testy.

5.1 Nefunkční testování

Nefunkční testování je kategorie testování software, která se snaží ověřit stanovené nefunkční požadavky a vyzkoušet, jak se aplikace chová pod zátěží. Tento typ testování nesouvisí přímo s funkcí samotné aplikace, ale s jejími vlastnostmi. Příkladem jsou hlavně různé kategorie výkonového testování, jako je zátěžové testování (Load testing) a stress testování.

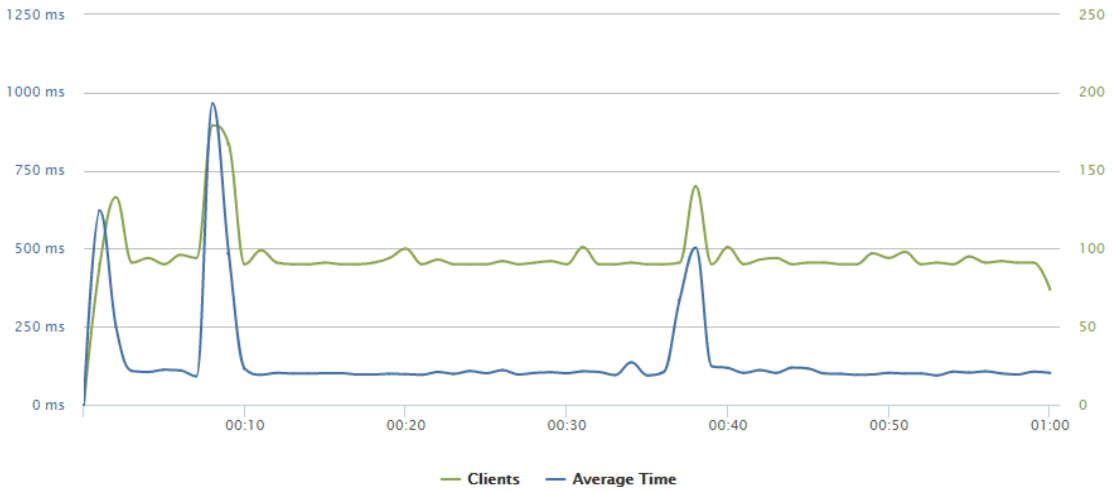
5.1.1 Zátěžové testování

Zátěžové testování [66] je typ softwarového testování, který slouží k získání představy o chování systému při specifické zátěži.

Podívejme se tedy na to, jak se chová naše aplikace při různém počtu požadavků na server. Pro toto testování si musíme definovat, co je to běžná zátěž. Uvažujme, že běžný uživatel naší aplikace vykoná 250 požadavků na server během minuty. Běžný provoz definujme jako 20 současně pracujících uživatelů. Z toho vyplývá, že by naše aplikace měla zvládnout vyřídit alespoň 5000 požadavků během minuty s rozumnou latencí. Za rozumnou latenci považujme dobu 500ms. Aplikace bude testována na instanci B1 6.1, na kterou byla nasazena.

Pro výkonové testování jsem se rozhodl využít službu Loader.io [1]) , která ve verzi zdarma umožňuje provést 10000 požadavků v rámci jednoto testovacího případu. Nadefinoval jsem si tedy svůj testovací případ, který bude testovat požadavky na detail seriálu, což je nejběžnější typ dotazu v klientské aplikaci.

[1]) `https://loader.io`



Obrázek 5.1. Výsledek zátěžového testu pro 20 uživatelů.

Průměrná doba odezvy během minutového testu byla 153ms, což splnilo stanovený požadavek. Problém však nastává u komplexnějších úloh typu domovské obrazovky. Zde již propustnost serveru a odezva dosahují horších hodnot.

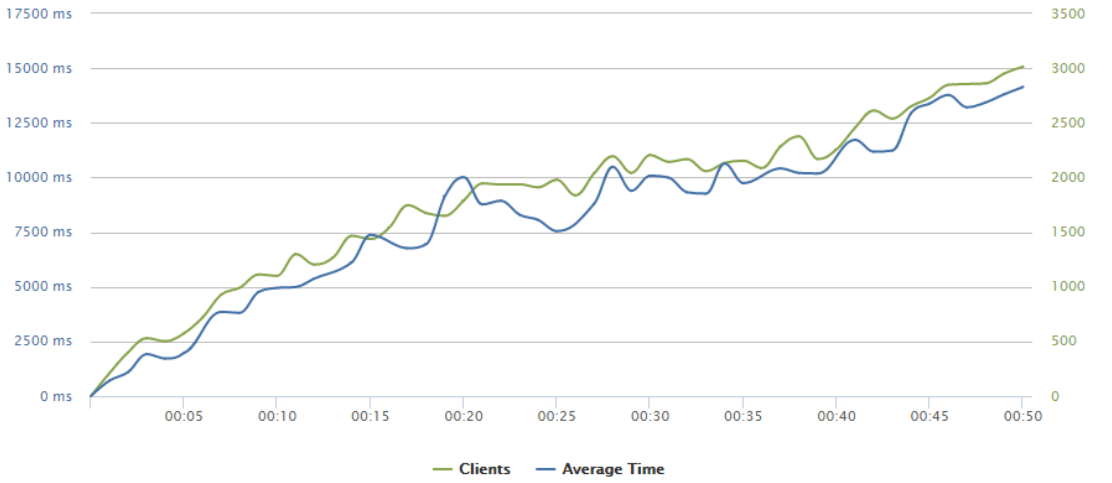
Procesor během testu vykazoval využití maximálně 50%, operační paměť 70% a využití prostředků na MongoDb bylo tak nízké, až bylo téměř neměřitelné. Server v současném nasazení vyhovuje požadovanému použití.

5.1.2 Stress testování

Druhým typem nefunkčního testování, které zde použijeme je stress testování. Pomocí tohoto druhu testování bychom měli najít limity našeho softwaru a podrobit ho větší než očekávané zátěži abychom zjistili chování v těchto nenadálých situacích.

Pro testování použijeme také zdroj pro detail seriálu. Postupným zvyšováním počtu požadavků narazíme na limit, kdy již aplikace nezvládá odpovídat a požadavky se začnou hromadit v HTTP frontě. Aplikace je poté zahlcena a odpovídá velmi pomalu, až téměř vůbec, a normální použití aplikace není možné.

Postupným zvyšováním počtu požadavků za sekundu, kde dolní hranicí byla předpokládaná zátěž, jsem se dostal na 200 požadavků ze sekundu, kdy už server požadavky hromadí a neodpovídá korektně. Na obrázku 5.2, vidíme narůstající počet neodbavených požadavků s kterými stoupá latence odpovědí.

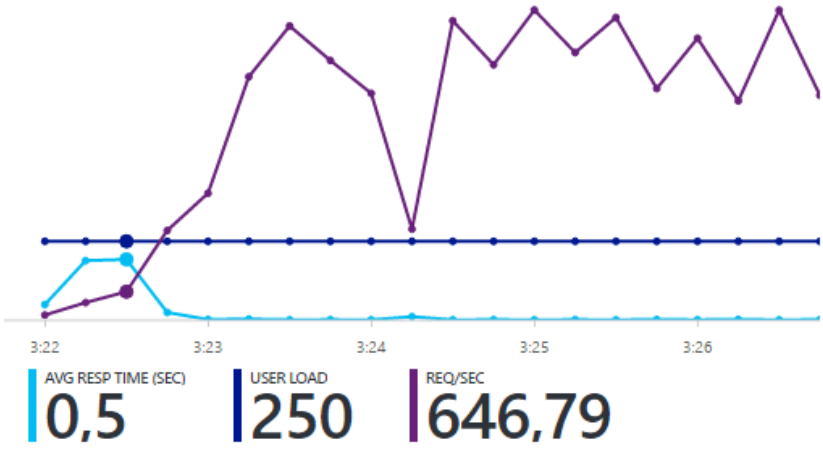


Obrázek 5.2. Výsledek stress testu při zátěži 200 požadavků za sekundu.

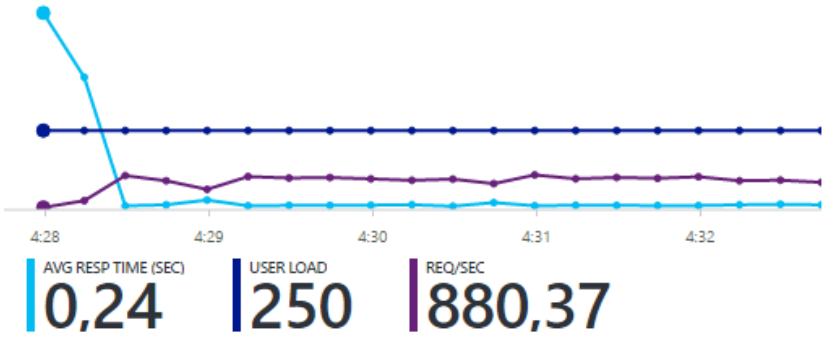
5.1.3 Horizontální škálování

Při tvorbě aplikace byl kladen důraz na možnost horizontálního škálování, ověříme si zde jak se aplikace chová při nasazení na dvě shodné instance B1.

Definujme si testovací případ, na kterém si ověříme chování při nasazení jedné a dvou instancí. Protože neumíme prozatím jednoduše škálovat databázi, rozhodl jsem se ji z tohoto testu vyřadit a otestovat pouze samotný výpočetní stroj, který poskytuje API. Pro tyto účely jsem použil zdroj stavu serveru, který se nedotazuje do databáze ani na externí služby.



Obrázek 5.3. Škálování: 1 instance.



Obrázek 5.4. Škálování: 2 instance.

Na obrázcích 5.3 a 5.4 vidíme průběh požadavků při zátěži 250 uživatelů. V případě nasazení na 2 instance se latence snížila o 52% a počet zpracovaných požadavků vzrostl na 132% . Při větším počtu konkurentních uživatelů by pravděpodobně tato hodnota ještě stoupla, bohužel se mi ale nepodařilo více než 250 uživatelů nasimulovat kvůli omezením tohoto testu na MS Azure.

5.2 Funkční testování

Funkční testování je kategorie testování software, kdy se snažíme ověřit funkčnost daného programu. Tato kategorie se dále dělí na na podkategorie, kterými jsou například

jednotkové testy, integrační testy, systémové testy a akceptační testy. Funkční testování odhalí funkční problémy v aplikaci, případně zkontroluje, zda se po jakékoliv změně chová aplikace stále dle požadavků. Při nalezení chyby v programu, která nebyla odhalena pomocí testů, je dobré test na tuto chybu doplnit, aby byl kontrolován její další možný výskyt v budoucnu.

Jelikož aplikace vystavuje veškerou svou funkcionalitu navenek skrze API, vhodným způsobem otestování se zdají být integrační testy a několik jednotkových testů, které pokryjí komplexnější procesy v aplikaci.

5.2.1 Jednotkové testování

Jednotkové testování, neboli Unit testování, se zabývá testováním nejmenších možných částí [67] testovaného softwaru, izolovaného od ostatních částí kódu. Pomocí těchto testů ověřujeme, zda se tato nejmenší část kódu chová dle očekávání. Tyto části se následně spojí a provádí se v ideálním případě integrační testování.

Jelikož do aplikace vstupuje několik zdrojů dat, bylo nutné testované části kódu od těchto zdrjů odstínit, protože jejich funkčnost není předmětem jednotkového testu. Odstínění od volání API třetích stran je vyřešeno skrze techniku, která se nazývá mockování. Pomocí této techniky dokážeme aplikaci doslova podstrčit požadované chování komponenty. Mockování těchto komponent jsem řešil skrze mockovací knihovnu *Sinon.js* [1]). Dalším vstupem, který se zapojuje do procesů, je naše primární databáze. Abych zaručil, že vždy dostanu data dle očekávání, vytvořil jsem testovací verzi databáze, která je vždy obnovena před každým jednotlivým testem. Pro tento typ testování jsem použil JavaScriptový testovací framework *mocha* [2]).

49.47% Statements 422/853 **40.06%** Branches 129/322 **37.75%** Functions 57/151 **49.41%** Lines 417/844

File		Statements		Branches		Functions		Lines	
eventService.js		92.31%	36/39	81.48%	22/27	100%	3/3	92.31%	36/39
fbService.js		91.18%	31/34	62.5%	5/8	100%	6/6	91.18%	31/34
homeService.js		82.28%	130/158	82.19%	60/73	100%	20/20	81.82%	126/154
recommendationService.js		98.61%	71/72	88.89%	16/18	100%	7/7	98.61%	71/72
seriesService.js		26.67%	48/180	15.22%	7/46	19.61%	10/51	26.82%	48/179
subscriptionService.js		31.15%	57/183	19.39%	19/98	19.05%	4/21	30.77%	56/182
tvdbService.js		28.24%	37/131	0%	0/30	24%	6/25	28.91%	37/128
userService.js		21.43%	12/56	0%	0/22	5.56%	1/18	21.43%	12/56

Obrázek 5.5. Pokrytí servisních tříd jednotkovými testy.

Jednotkovými testy jsem pokryl klíčové servisní třídy naší aplikace, které vidíme na obrázku 5.5.

5.2.2 Integrační testování

Integrační testování[68] je testování komponent a jejich rozhraní, které jsou přístupné a viditelné. Tyto komponenty by se měly skládat z jednotlivých otestovaných jednotek.

Pro integrační testování jsem na doporučení vedoucího práce použil framework *dredd* [3]). Tento testovací framework je vyvíjen pod splečností Apiary, u které máme vytvoře-

[1]) `http://sinonjs.org/`
[2]) `https://mochajs.org/`
[3]) `https://github.com/apiaryio/dredd`

nou definici API pomocí Blueprintu a mockovací server. Dredd je tedy také integrován do portálu a je možné sledovat výsledky jednotlivých testů včetně odpovědí serveru.

Framework po spuštění inicializuje lokální server a začne ho dotazovat jednotlivými požadavky z definice, jejichž odpovědi také validuje. Pro testování jsou vytvořeny dva takzvané 'hooky', které provádí obnovení testovací databáze před každým testem a provádí i autentifikaci uživatele. S autentifikací uživatele je spojený problém se získáním Facebook access tokenu pro aplikaci. Jelikož server využívá token k přihlášení na náš server i pro volání Facebook Graph API, je nutné, aby byl tento token platný i vůči Facebooku. Proto se musí po vypršení token manuálně v 'hooku' aktualizovat. Tento token se získá ze stránek aplikace na Facebook developers [1]).

5.2.3 Závěr testování

Klíčové části aplikace jsou pokryty jednotkovými testy a celé API testy integračními. Pokrytí jednotkovými testy by se mělo v budoucnu doplnit a pokrýt jimi v ideálním případě celou aplikaci. Nefunkční testování nám ukázalo, kolik aplikace zvládá požadavků a jaké hodnoty jsou hraniční. Na základě zjištěných hodnot byla ihned provedena optimalizace volání některých požadavků.

[1]) `https://developers.facebook.com/`

Kapitola 6
Nasazení

Aplikace byla vyvíjena standardně lokálně na mém počítači, kde jsem měl nainstalované všechny potřebné závislosti a prostředí. Po dokončení větší části práce jsem se pustil do nasazení celého projektu do prostředí MS Azure, kde je nyní aplikace umístěna . Nyní následuje popis nasazení každého jednotlivého samostatného funkčního celku.

6.1 Serverová aplikace

Tato část aplikace se nasazuje do služby Azure App Service. Způsobů nasazení je zde několik, od nahrávání na FTP po GIT. Já jsem zvolil variantu s GITem z důvodu verzování nasazení.

Tento vytvořený repozitář by měl obsahovat projekt, který chceme nasadit, v našem případě serverovou část aplikace. V kořenu této složky se musí nacházet soubor *package.json*, který definuje všechny závislosti aplikace a definuje také skript, jak má být aplikace spuštěna. Aplikace je tedy nasazována bez složky *node_modules*, ve které jsou staženy závislosti. Tyto závislosti se nainstalují na serveru automaticky. U některých nativních modulů dojde pochopitelně i ke kompilaci na cílovou platformu, proto bychom neměli tuto složku nasazovat. Pro snadnější vytvoření distribuční verze jsem definoval jednoduchý GULP skript, který připraví složku, která se následně commituje do repozitáře.

Azure app service má před nasazenou node.js aplikací nasazený ještě vlastní webový server IIS [69], který spravuje i samotný node proces skrze modul *iisnode*. Jeho výchozí konfiguraci dokážeme změnit souborem *iisnode.yml*, který se nachází v kořenu repozitáře. Můžeme si tedy například nastavit, při změně jakých souborů se má proces restartovat.

Poté co se lokální repozitář nahraje do App service, započne nasazení aplikace a vytvoření zálohy nasazení. Je tedy možné se kdykoliv vrátit na minulé nasazení přímo z portálu MS Azure. Bohužel je u základních plánů po dobu nasazování server nedostupný. U vyšších plánů je možné využít nasazovací sloty a aplikace může běžet bez výpadku.

Jistým problémem při nasazení bylo také zprovoznění SSL/TLS pro zabezpečenou komunikaci mezi klientem a serverem. Pro to, abychom mohli provozovat tuto zabezpečenou komunikaci, musí mít naše URL vystavený serverový certifikát, který poskytne klientovi, a ten si ověří jeho důvěryhodnost. Aby se však tvářil z pohledu prohlížeče důvěryhodně, musí jít o certifikát vydaný certifikační autoritou, která má svůj kořenový certifikát integrován v prohlížečích. Vydání takového certifikátu je většinou zpoplatněno. Existuje však otevřená certifikační autorita s názvem Let's Encrypt [1]).

Cílem této certifikační autority je zajisti co nejsnadnější cestu, jak získat serverový certifikát bez interakce člověka [70]. Řešení skrze tuto certifikační autoritu mi přišlo vhodné hlavně z důvodu bezúdržbovosti a ceny. Bohužel však App Service nemá tuto službu nativně zaintegrovanou, proto je třeba tuto službu doinstalovat. Rozepisovat

[1]) `https://letsencrypt.org/`

postup je zde zbytečné, proto pouze odkáži na návod, který jsem použil [71]. Po provedení tohoto postupu má naše URL validní serverový certifikát pro všechny majoritní prohlížeče a je možné provozovat komunikaci skrze HTTPS.

V současnosti je tato část aplikace nasazena na jedné instanci B1, která má konfiguraci 1 jádro CPU, 1,75GB Ram a 10 GB úložiště.

6.2 Loadery

Loadery jsou samostatné procesy, které mají běžet na pozadí a zpracovávat asynchronní úlohy. Jejich nasazení se provádí spolu s hlavní serverovou aplikací do App Service [72]. Pro nasazení těchto úloh, v App Service nazývaných webové úlohy, slouží složka *App_Data*. Tato složka obsahuje dvě podsložky (*triggered*, *continuous*), kde první složka slouží pro úlohy, které jsou spuštěny plánovaně, a druhá pro úlohy kontinuální.

Loadery se řadí mezi úlohy kontinuální, protože běží neustále na pozadí a kontrolují, zda není ve frontách nový požadavek na zpracování. Mezi plánované úlohy patří například kontrola aktualizací seriálů z TVDB, která se spouští každé 2 hodiny. Definice plánování těchto úloh se provádí pomocí standardního CRON zápisu, který můžeme zpropagovat skrze soubor *settings.job* [73].

Nutno podotknout, že v této podobě běží webové úlohy na stejném stroji jako serverová aplikace, tedy sdílí i stejné prostředky, což má výhodu v tom, že nemusíme platit poplatky za další stroj. Nelze však škálovat tyto procesy samostatně.

6.3 MongoDB

Databázi pro aplikaci jsem měl během vývoje nainstalovanou lokálně. Při nasazování této databáze však vyvstal problém, jak databázi nasadit do prostředí MS Azure. Microsoft nabízí formou služby pouze DocumentDB. Naskytly se tedy dvě možnosti, jednou z možností bylo umístit databázi na jiný cloud, který poskytuje databázi jako službu, nebo oželet snadnou škálovatelnost a bezúdržbovost a vytvořit si vlastní databázi na pronajatém virtuálním stroji v rámci MS Azure.

Osobně jsem se přikláněl více k variantě ponechat všechno v rámci jednoho cloudového prostředí, což se podařilo díky tomu, že lze nainstalovat plně nakonfigurovaný virtuální stroj z katalogu Bitnami [1]). Instalace tedy zabrala doslova pár kliknutí. Toto řešení však není ideální, protože se o virtuální stroj musíme starat od základu, tedy o všechny aktualizace, zálohy a podobné záležitosti. Škálovat databázi horizontálně by vyžadovalo také jistě větší úsilí oproti možnostem škálování v případě využití databáze jako služby. Do budoucna je třeba přenést databázi do prostředí, kde nám bude poskytnuta jako služba. Finančně je současné řešení přijatelné. Náklady na současnou instanci(1 CPU, 1.75GB Ram) jsou 17 Euro na měsíc.

Samotné databázové schéma není nutné vytvářet tak, jak je tomu u relačních databází. Díky dynamickému schématu se vytvoří při vložení prvního dokumentu do dané kolekce.

V době psaní práce (leden 2017) nabízel Microsoft nové řešení [74], kdy lze používat DocumentDB skrze MongoDB driver. Toto řešení by zřejmě vyřešilo náš problém s databází a mohli bychom ji používat formou služby, avšak cenová politika je z našeho úhlu pohledu stejně nepříjemná, jako tomu je u standardní DocumentDB. Proto i přes tuto možnost zůstává databáze nasazena na virtuálním stroji plně pod naší správou.

[1]) `https://bitnami.com/stack/mongodb`

6.4 Azure Storage

Azure Storage je souhrnný název pro různé typy úložišť, konkrétně jde o Blob, Fronty, Soubory a Tabulky. Jedná se o proprietární řešení Microsoftu, avšak pro lokální vývoj je nabízen Storage emulator [75]. Aplikaci jsem po celou dobu vyvíjel lokálně proti tomuto emulátoru a přesun do cloudu jsem řešil až ve fázi nasazení.

Pro přístup do těchto úložišť se používají přístupové klíče, které se převážně ukládají do proměnných prostředí, z kterých si je potom aplikace převezme a použije. Nejinak tomu bylo i v případě naší aplikace. Pro nasazení aplikace bylo třeba vytvořit nový účet úložiště na portálu Azure. Z tohoto účtu je třeba následně získat potřebné přístupové údaje a ty nastavit do proměnných prostředí serverové aplikace běžící v App Service.

Definice front ani nic jiného není třeba vytvářet, při použití aplikace kontroluje jejich existenci a případně je vytvoří.

Kapitola 7
Závěr

Cílem této práce bylo naimplementovat a nasadit serverovou část aplikace SerialFreak. Prvotní návrh aplikace vznikl z myšlenek v hlavě mé a mého kolegy Filipa Dyrčíka. Tento návrh byl dále rozšířen poznatky získaných z kvalitativního uživatelského výzkumu kolegy Dyrčíka [1]. Na základě výsledků výzkumu byly stanoveny funkční a nefunkční požadavky na aplikaci.

Z definovaných požadavků vzešel návrh aplikačního rozhraní mezi klientskou a serverovou částí aplikace, které poté striktně určovalo integrační vrstvu mezi dvěmi nezávislými pracemi. Následně byly analyzovány technologie pro implementaci serverové části včetně fenoménu poslední doby, cloudu.

Po analýze technologií jsem navrhl architekturu aplikace a vybral cloudové prostředí pro její běh. Architektura reflektuje nefunkční požadavky na aplikaci s důrazem na snadnou horizontální škálovatelnost a rozšíření.

Pro implementaci serveru jsem zvolil JavaScriptové prostředí Node.js. Implementace v jazyku JavaScript probíhala jinak, než jsem zvyklý z prostředí Java. Neexistují zde pro programátora vlákna a celý vývoj je tak silně asynchronní. Vývoj probíhal rychle díky velkému množství dostupných knihoven, které mi usnadnili práci.

Na základě studie byla aplikace realizována pro cloudové prostředí Microsoft Azure. Efektivita práce s Microsoft Azure jako PaaS byla místy rozporuplná. Nastaly problémy, jejichž řešení by bylo snažší při nasazení na vlastní infrastruktuře. Cenová politika služby je mířena především na firemní zákazníky. Pro soukromé a komunitní projekty by mohla být příliš nákladná. Na současný provoz aplikace SerialFreak jsou měsíční náklady cca 90 Euro.

Novou zkušeností pro mě byla také implementovaná NoSQL databáze MongoDB. Dynamické schéma databáze ve spojení s dynamickým typovým systémem Javascriptu vytvořilo efektivní způsob, jak ukládat a získávat data z databáze.

Celá aplikace je nasazena do zvoleného cloudového prostředí. Zátěžové testy ukázali, jak se aplikace chová za různých podmínek vytížení. Opakování zátěžových testů s použitím více instancí serveru prokázalo možnost požadovaného horizontálního škálování aplikace.

Tato práce analyzovala a spojila dohromady značnou množinu pro mě do té doby neznámých technologií, které se mi podařilo uvést do praxe. Aplikace SerialFreak je veřejně přístupná, byť zatím jen pro testovací uživatele. Uvolnění aplikace pro veřejnost podléhá dokončení funkčního testování a optimalizaci datového toku mezi serverem a klientem.

7.1 Budoucí vývoj

Přesun databáze

Databáze je v současném stavu nasazena na virtuálním počítači plně pod naši správou. Do budoucna je plánováno využití databáze jako služby z důvodu odstranění potřeby správy virtuálního stroje.

Výkon

Aplikace je architektonicky navržena správně, avšak je zde prostor k optimalizaci databázových dotazů a aplikačních procesů.

Testování

Klíčové části programu jsou otestovány, přesto je třeba v budoucnu pokrýt i zbylé části aplikace.

Vkládání odkazovaných objektů v API

Aplikace ctí zásady návrhu RESTful rozhraní a HATEOAS, kvůli čemuž potřebuje klient pro získání některých informací velký počet požadavků na server. Toto chování lze snadno upravit vkládáním odkazovaných objektů na vyžádání a tím také docílit menšího zatížení serveru.

Literatura

[1] Filip Dyrčík. *Klientská část aplikace serialfreak*. 2017.

[2] I. Neustadt a J. Arlow. *UML2 a unifikovaný proces vývoje aplikací*. 2016. ISBN 9788025142059.
https://books.google.cz/books?id=zBHqCwAAQBAJ.

[3] *Smlouva SLA pro Aplikační služba*.
https://azure.microsoft.com/cs-cz/support/legal/sla/app-service/v1_2/. (Online; navštíveno 15.02.2016).

[4] *HTML Responsive Web*.
http://www.w3schools.com/html/html_responsive.asp. (Online; navštíveno 10.01.2016).

[5] *11.1 Defining Risk Project Management for Instructional Designers*.
http://pm4id.org/chapter/11-1-defining-risk/. (Online; navštíveno 17.02.2016).

[6] Jennifer Campbell. *Lecture 5, Part 1:Risk*.
http://www.cdf.toronto.edu/~csc340h/winter/lectures/w5/L5-part1-6up.pdf. (Online; navštíveno 15.2.2016).

[7] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*.
http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. 2000. (Online; navštíveno 01/04/2017).

[8] *What is Idempotency?*
http://www.restapitutorial.com/lessons/idempotency.html. (Online; navštíveno 04.01. 2017).

[9] *Understanding HATEOAS*.
https://spring.io/understanding/HATEOAS.

[10] *Type Inference TypeScript*.
https://www.typescriptlang.org/docs/handbook/type-inference.html. (Online;navštíveno 5.1. 2017).

[11] *Java Timeline*.
http://oracle.com.edgesuite.net/timeline/java. (Online;navštíveno 2.1. 2017).

[12] K. Lei, Y. Ma a Z. Tan. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js. 2014, 661-668. DOI 10.1109/CSE.2014.142.

[13] M. Cantelon, T.J. Holowaychuk, M. Harter a N. Rajlich. *Node.js in Action*. Manning, 2013. ISBN 9781617290572.
https://books.google.cz/books?id=JV6rpwAACAAJ.

[14] *Teorie relačních databází: Normalizace*.
http://www.manualy.net/article.php?articleID=13. (Online;navštíveno 3.1. 2017).

[15] Tim O'Reilly. *A Brief History of NoSQL All About the Code.* http://blog.knuthaugen.no/2010/03/a-brief-history-of-nosql.html. 2007. (Online; navštíveno 17.12. 2016).

[16] Tim O'Reilly. *What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software.* https://mpra.ub.uni-muenchen.de/4578/1/mpra_paper_4578.pdf. 2007. (Online; navštíveno 17.12. 2016).

[17] *Vy ještě neznáte MongoDB?* https://www.tomas-dvorak.cz/posts/vy-jeste-neznate-mongodb/. (Online; navštíveno 5.1. 2017).

[18] MongoDB. *NoSQL Databases Explained — MongoDB.* https://www.mongodb.com/nosql-explained. (Online;navštíveno 17.12. 2016).

[19] *Sharding Pattern.* https://msdn.microsoft.com/en-us/library/dn589797.aspx. (Online; navštíveno 4.1. 2017).

[20] *Our History — The Movie Database (TMDb).* https://www.themoviedb.org/about/our-history. (Online;navštíveno 18.12. 2016).

[21] *General FAQ — The Movie Database (TMDb).* https://www.themoviedb.org/faq/general. (Online;navštíveno 18.12. 2016).

[22] *API FAQ — The Movie Database (TMDb).* https://www.themoviedb.org/faq/api. (Online;navštíveno 18.12. 2016).

[23] *moviedb.* https://www.npmjs.com/package/moviedb. (Online; navštíveno 18.12. 2016).

[24] *Online TV Database - An open directory of television shows for HTPC software.* http://thetvdb.com/?tab=reports. (Online;navštíveno 18.12. 2016).

[25] *node-tvdb.* https://www.npmjs.com/package/node-tvdb. (Online;navštíveno 18.12. 2016).

[26] *The NIST Definition of Cloud Computing.* http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf. (Online; navštíveno 15.02.2016).

[27] *Zadgaonikar, Hrushikesh .* https://hrushikeshzadgaonkar.files.wordpress.com/2011/05/cloud_stack.gif.

[28] *Přehled Azure komponent.* https://docs.microsoft.com/en-us/azure/fundamentals-introduction-to-azure.

[29] *Dokumentace ke službě DocumentDB — Microsoft Docs.* https://docs.microsoft.com/cs-cz/azure/documentdb/. (Online;navštíveno 2.1. 2017).

[30] *Azure Redis Cache – cloudová služba založená na mezipaměti Redis Microsoft Azure.* https://azure.microsoft.com/cs-cz/services/cache/.

[31] Larry Dignan. *Amazon, Microsoft, IBM and the cloud gang: Comparing the revenue.* http://www.zdnet.com/article/amazon-microsoft-ibm-and-the-cloud-gang-comparing-the-revenue/.

[32] David Ramel. *Microsoft No. 2 Behind Amazon in Cloud Market Share – Redmond Channel Partner.*

https://rcpmag.com/articles/2016/08/02/microsoft-behind-aws-in-cloud.aspx. (Online;navštíveno 2. 1. 2017).

[33] *Microsoft Azure*. 2016. https://en.wikipedia.org/w/index.php?title=Microsoft_Azure&oldid=704876698. Page Version ID: 704876698.

[34] Tomáš Zvěřina. *Node.js na Google Cloud Platform*. 2015. https://www.zdrojak.cz/clanky/node-js-na-google-cloud-platform/.

[35] *Ryan Dahl JSConf*. http://www.jsconf.eu/2009/speaker/speakers_selected.html.

[36] *Node.js*. https://nodejs.org/en/. (Online; navštíveno 04.01. 2017).

[37] *Apache: multi-threaded vs multi-process (pre-forked) — Articles — Zerigo*. https://www.zerigo.com/article/apache_multi-threaded_vs_multi-process_pre-forked. (Online;navštíveno 5. 1. 2017).

[38] Rambabu Posa. *Node JS Architecture - Single Threaded Event Loop - JournalDev*. http://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop. (Online;navštíveno 4.1. 2017).

[39] Shailendra Chauhan. *Node.js vs. other Server Side frameworks*. http://www.dotnettricks.com/learn/nodejs/nodejs-vs-other-server-side-frameworks. (Online; navštíveno 4.1. 2017).

[40] Roman Ožana. *Gulp vs. Grunt: souboj bez vítěze a poraženého - Zdroják*. https://www.zdrojak.cz/clanky/gulp-vs-grunt-souboj-bez-viteze-aporazeneho/. (Online;navštíveno 20.12. 2016).

[41] Jani Hartikainen. *A Comparison of JavaScript Linting Tools*. https://www.sitepoint.com/comparison-javascript-linting-tools/. (Online;navštíveno 21.12. 2016).

[42] *Overview - Graph API*. https://developers.facebook.com/docs/graph-api/overview/. (Online;navštíveno 22.12. 2016).

[43] *App-to-User Notifications - Games*. https://developers.facebook.com/docs/games/services/appnotifications. (Online;navštíveno 22.12.2016).

[44] *Swagger – The World's Most Popular Framework for APIs*. http://swagger.io/. (Online;navštíveno 22.12. 2016).

[45] *swagger-client*. https://www.npmjs.com/package/swagger-client. (Online;navštíveno 22.12. 2016).

[46] *Express - Node.js web application framework*. http://expressjs.com/. (Online;navštíveno 29.12. 2016).

[47] *Using Express middleware*. http://expressjs.com/en/guide/using-middleware.html. (Online;navštíveno 29.12. 2016).

[48] Thanasis Polychronakis. *Intro to node.js*. http://www.slideshare.net/thanpolas/intro-to-nodejs-39066435. (Online; navštíveno 29.12. 2016).

[49] *JSON Web Token Introduction - jwt.io*. https://jwt.io/introduction/. (Online;navštíveno 29.12. 2016).

[50] *hapijs/joi: Object schema validation.*
https://github.com/hapijs/joi. (Online; navštíveno 29.12. 2016).

[51] *NoSQL Data Modeling Techniques – Highly Scalable Blog.*
https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/. (Online;navštíveno 2.1. 2017).

[52] *Introduction to DocumentDB, a JSON database — Microsoft Docs.*
https://docs.microsoft.com/en-us/azure/documentdb/documentdb-introduction. (Online;navštíveno 2.1. 2017).

[53] *Introduction to MongoDB — MongoDB Manual 3.4.*
https://docs.mongodb.com/manual/introduction/. (Online;navštíveno 2.1. 2017).

[54] *Mongoose ODM v4.7.5.*
http://mongoosejs.com/. (Online;navštíveno 2.1. 2017).

[55] *Worker Dynos, Background Jobs and Queueing — Heroku Dev Center.*
https://devcenter.heroku.com/articles/background-jobs-queueing. (Online;navštíveno 3.1. 2017).

[56] *Azure Queues and Service Bus queues - compared and contrasted — Microsoft Docs.*
https://docs.microsoft.com/cs-cz/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted. (Online;navštíveno 3.1. 2017).

[57] *Přehled Azure Functions — Dokumentace Microsoftu.*
https://docs.microsoft.com/cs-cz/azure/azure-functions/functions-overview. (Online;navštíveno 3.1. 2017).

[58] rmcmurray. *How to use Blob storage from Node.js.*
https://docs.microsoft.com/cs-cz/azure/storage/storage-nodejs-how-to-use-blob-storage. (Online;navštíveno 3.1. 2017).

[59] Amir Salihefendic. *How Reddit ranking algorithms work – Hacking and Gonzo.* 2015.
https://medium.com/hacking-and-gonzo/how-reddit-ranking-algorithms-work-ef111e33d0d9.

[60] Evan Miller. *How Not To Sort By Average Rating.*
http://www.evanmiller.org/how-not-to-sort-by-average-rating.html. (Online;navštíveno 4.1. 2017).

[61] *Distribuční funkce.*
http://cit.vfu.cz/statpotr/potr/teorie/predn1/distrib.htm. (Online; navštíveno 3.1. 2017).

[62] *wilson-interval.*
https://www.npmjs.com/package/wilson-interval. (Online;navštíveno 4.1. 2017).

[63] *IMDb Votes/Ratings Top Frequently Asked Questions.*
http://www.imdb.com/help/show_leaf?votestopfaq. (Online;navštíveno 4.1. 2017).

[64] *Bayes estimator Wikipedia.*
https://en.wikipedia.org/wiki/Bayes_estimator. (Online;navštíveno 2.1. 2017).

[65] Tomáš Hlava. *Funkční Testy — Testování softwaru.*
http://testovanisoftwaru.cz/tag/funkcni-testy/. (Online;navštíveno 2.1. 2017).

[66] *What is Load testing in software testing?*
http://istqbexamcertification.com/what-is-load-testing-in-software/. (Online;navštíveno 5.1. 2017).

[67] *Unit Testing*.
https://msdn.microsoft.com/en-us/library/aa292197%28v=vs.71%29.aspx?f=255&MSPPError=-2147217396. (Online;navštíveno 2.1. 2017).

[68] *Integration Testing*.
https://msdn.microsoft.com/en-us/library/aa292128%28v=vs.71%29.aspx?f=255&MSPPError=-2147217396. (Online;navštíveno 6.1. 2017).

[69] *tjanczuk/iisnode: Hosting node.js applications in IIS on Windows*.
https://github.com/tjanczuk/iisnode. (Online;navštíveno 4.1. 2017).

[70] *How It Works - Let's Encrypt - Free SSL/TLS Certificates*.
https://letsencrypt.org/how-it-works/. (Online;navštíveno 4.1. 2017).

[71] Nik Molnar. *"Let's Encrypt" Azure Web Apps the Free and Easy Way*.
https://gooroo.io/GoorooTHINK/Article/16420/Lets-Encrypt-Azure-Web-Apps-the-Free-and-Easy-Way/20047. (Online;navštíveno 4.1. 2017).

[72] *Blog.Amit Apple*.
http://blog.amitapple.com/post/74215124623/deploy-azure-webjobs/. (Online;navštíveno 2.1. 2017).

[73] *Run Background tasks with WebJobs — Microsoft Docs*.
https://docs.microsoft.com/en-us/azure/app-service-web/web-sites-create-web-jobs. (Online;navštíveno 3.1. 2017).

[74] *What is DocumentDB protocol support for MongoDB? — Microsoft Docs*.
https://docs.microsoft.com/en-us/azure/documentdb/documentdb-protocol-mongodb. (Online;navštíveno 1.1. 2017).

[75] *Use the Azure Storage Emulator for Development and Testing — Microsoft Docs*.
https://docs.microsoft.com/en-us/azure/storage/storage-use-emulator. (Online;navštíveno 1.1. 2017).

Příloha A
Zkratky

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DAO	Data access object
DTO	Data transfer object
CSS	Cascading Style Sheets
FB	Facebook
FTP	File Transfer Protocol
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a service
IDE	Integrated Development Environment
IO	Input Output
J2EE	Java Platform Enterprise Edition
JPA	Java Persistence API
JSON	JavaScript Object Notation
JWT	Json web token
LTS	Long Term Support
MS	Microsoft
NIST	National Institute of Standards and Technology
NoSQL	Not only SQL
ODM	Object document mapping
ORM	Object-relational mapping
PaaS	Platform as a Service
REST	Representational state transfer
RU	Request unit
SaaS	Software as a service
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TMDB	The Movie Database
TVDB	TheTVDb.com
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	Ain't Markup Language

Příloha B
Obsah přiloženého CD

```
/
├── DokumentaceEA
│   └── serialfreak.eap
├── src
│   ├── DP-TEX ..... Text diplomové práce ve formátu Plain TEX
│   └── serialfreak-backend ..... Zdrojové soubory aplikace
├── apiary.apib ..... Definice API ve formátu Blueprint
└── pychaja1-dp-2017.pdf ..... Text diplomové práce ve formátu PDF
```

Příloha C
Definice API

Definice aplikačního rozhraní je obsáhlá a její rozsah není vhodný pro tisk, proto je uložena na CD jako soubor *apiary.apib*.