Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Vývoj progresivní webové aplikace pro recenze filmů

Mykola Holubiev

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Mykola Holubiev

Informatika

Název práce

Vývoj progresivní webové aplikace pro recenze filmů

Název anglicky

Development of a progressive web application for movie reviews

---

Cíle práce

Cílem je navrhnout, implementovat a otestovat webovou aplikaci pro recenze filmů. Práce bude hodně zaměřena na praktické použití moderních webových technologií. Aplikace bude používána několika odborníky z filmového umění, kteří budou psát své recenze a mít roli správce. Také budou přítomni redaktoři a moderátoři. Přihlášení uživatelé budou moct ohodnotit recenzi na pětibodové škále a psát své názory a komentáře. Aplikace musí být vytvořena ve stylu progresívní webové aplikace, aby uživatelé mohli číst recenze také offline.

Metodika

Práce se bude skládat z teoretické a praktické části. Nejprve bude provedena analýza požadavků a dostupných webových technologií pro tvorbu aplikace. Následujícím krokem bude implementace zvolených technologií v daném projektu. Serverová část aplikace bude běžet na Node.js + Express a PostgreSQL. Klientská část bude implementována pomocí knihoven React + Next.js a Redux. Dokumentace bude podle standardu UML a hotová aplikace bude otestována.

Doporučený rozsah práce

40-80 stran

Klíčová slova

JavaScript; Node.js; REST API; React; Next.js; Redux; PWA

Doporučené zdroje informací

MOHAN, Mehul. Advanced Web Development with React: SSR and PWA with Next.js using React with advanced concepts. ENG ed. New Delhi: BPB Publications, 2020. ISBN 978-9389423594.

R. YOUNG, Alex, Bradley MECK, Mike CANTELON, Tim OXLEY, Marc HARTER, TJ HOLOWAYCHUK a Nathan RAJLICH. Node.js in Action. 2nd ed. Shelter Island: Manning Publications, 2017. ISBN 978-1617292576.

TIELENS THOMAS, Mark. React in Action. Shelter Island: Manning Publications, 2018. ISBN 978-1617293856.

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 1. 11. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 23. 11. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 03. 2022

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Vývoj progresivní webové aplikace pro recenze filmů" jsem vypracoval(a) samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor(ka) uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2022 _____

Poděkování

Rád(a) bych touto cestou poděkoval(a) doc. Ing. Vojtěchu Merunkovi, Ph.D. za vedení a pomoc při napsaní této bakalářské práce.

Vývoj progresivní webové aplikace pro recenze filmů

Abstrakt

Tato bakalářská práce je věnována problematice vývoje webových aplikací. V teoretické části jsou popsány důležitá východiska pro pochopení, jak fungují pokročilejší webové aplikace. Toto zahrnuje popis jazyků JavaScript a TypeScript, jejich odlišnosti a možnosti. Dále jsou probrané důležité látky a základy, které se týkají serverové a klientské části aplikací. Nakonec je zmíněno, co je progresivní webová aplikace, a k čemu slouží.

Hlavním cílem praktické části je vytvoření progresivní webové aplikace pro recenze filmů. Předpokladem pro ni slouží poptávka několika odborníků z filmového umění. Proces vytvoření je rozdělen na tři části: analýza, návrh a implementace. Každá následující část je navázaná na předchozí. Nejdřív se analyzují požadavky a funkcionality. Pak následuje návrh zpracovaný hlavně podle UML specifikace. Na závěr je popsán proces implementace s ukázkami zdrojové kódu a vzhledu výsledné aplikace.

Klíčová slova: JavaScript, TypeScript, backend, frontend, Node.js, Express.js, React, Next.js, PWA.

Development of a progressive web application for movie reviews

Abstract

This bachelor thesis is devoted to the issue of web applications development. The theoretical part describes important basics for understanding of how more advanced web applications work. This includes a description of the JavaScript and TypeScript languages, their differences and capabilities. Furthermore, important topics and basics related to the server and client part of applications are explained. Finally, what a progressive web application is and what it is for is mentioned.

The main goal of the practical part is to create a progressive web application for movie reviews. The prerequisite for it is the demand of several film experts. The creation process is divided into three parts: analysis, design and implementation. Each subsequent part is linked to the previous one. Requirements and functionalities are analyzed first. Then design part follows, which is processed mainly according to the UML specification. Finally, the implementation process is described with samples of source code and the appearance of the finished application.

Keywords: JavaScript, TypeScript, backend, frontend, Node.js, Express.js, React, Next.js, PWA.

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Seznam použitých zkratek

1 Úvod

V dnešní době se nedá představit moderní svět bez internetu a webových stránek, jež obsahuje. S časem se webové stránky stály většími a složitějšími tak, že se dnes považují za aplikaci. Toto je jedním z důvodů, proč hodně podnikatelů se rozhodují zahájít svůj nový byznys spojený právě s webovými technologiemi. Tak se nyní mezi populárními produkty pro byznys v oblasti informačních technologií považují progresivní webové aplikace.

Tato bakalářská práce se zabývá problematikou vývoje webových aplikací. Teoretické části popisuje důležité základy pro pochopení, jak fungují pokročilejší webové aplikace. Toto obsahuje popis jazyků JavaScript a TypeScript, jejich specifika, odlišnosti a možnosti. Dále jsou probrané důležité látky a základy, které se týkají serverové a klientské části aplikací. Na konci je uvedeno, co je progresivní webová aplikace, k čemu slouží a jaké musí splňovat charakteristiky.

Hlavním cílem praktické části je vytvoření progresivní webové aplikace pro recenze filmů. Předpokladem pro ni slouží poptávka několika odborníků z filmového umění. Proces vytvoření je rozdělen na tři části: analýza, návrh a implementace. Každá následující část je navázaná na předchozí. Nejdřív se analyzují požadavky a funkcionality. Jsou také uvedeny požadavky na role uživatelů. Pak je proveden návrh, který obsahuje diagramy zpracovaný hlavně podle UML specifikace. Nakonec se popisuje proces implementace s ukázkami zdrojové kódu a vzhledu výsledné aplikace.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této práce je vytvoření minimálního životaschopného produktu, který je představen progresivní webovou aplikací. Předpokladem pro vytvoření této aplikace slouží poptávka několika odborníků z filmového umění, kteří budou ji řídit. Po dosažení určitého počtu uživatelů by se měla aplikace směřovat ke komerčnímu využití, což je možné následně realizovat přidáním placených funkce a možnosti prostřednictvím předplatného. Proto hlavním účelem této práce je vytvořit minimální životaschopný produkt, který bude schopen přilákat uživatelé a dát důležitou zpětnou vazbu pro následující vývoj aplikace.

2.2 Metodika

Teoretická část bakalářské práce bude zpracována podle odborné literatury a zdrojů. Bude se dělit na tři základní podkapitoly. První podkapitola bude věnována popisu jazyků JavaScript a TypeScript. Druhá část se bude zabývat látkami a základy serverové části webových aplikací. V poslední části bude popsána klientská strana aplikací a její technologie.

Praktická část se nejdříve bude zabývat analýzou požadavků a funkcionalit. Poté proběhne návrh systému především pomocí UML specifikace. Následně podle návrhu bude webová aplikace implementována.

3 Teoretická východiska

3.1 JavaScript a TypeScript

JavaScript a TypeScript jsou v dnešní době velice populární jazyky. Bez nich se nedá představit současný vývoj informačních systému a aplikací, zejména webových. Neustále se vyvíjí, zlepšují a zjednodušují pro pohodlnou práci vývojářů. Oba jazyky mají různé vhodné případy využití, ačkoliv jsou obdobné.

3.1.1 JavaScript

Podle autora knihy (Haverbeke, 2018) JavaScript je skriptovací jazyk, který byl vytvořen Brendanem Eichem a představen v roce 1995 hlavně pro napsání programů a prezentační vrstvy ve webovém prohlížeči. V současné době JavaScript slouží skriptovacím a dotazovacím jazykem u některých databází, jako jsou MongoDB a CouchDB. Také se dost často používá i pro tvorbu serverové části webových aplikací, což je umožněno díky softwarovému systémy Node.js.

3.1.2 ECMAScript

Autor (Mohan, 2018) píše, že ECMAScript je standardizace jazyka JavaScriptu, která je normovaná organizaci Ecma International. Tato organizace se nyní snaží vydávat novou verzi ES každý rok, která výrazně zlepšují a zjednodušuje JavaScript. S verzi ES6, která byla vydána v roce 2015, se objevila důležitá zlepšení, jako jsou třídy, moduly, šipkové funkce, asynchronní programování, generátory atd. S verzi ES8 v roce 2018 se objevil operátor odpočinku neboli šíření pro objekty, který umožňuje snadně kopírovat jejich vlastností. Ne všechny prohlížeči podporují nové verze ECMAScriptu, počínající od ES6,

proto je nutné konvertovat kód do ES5 verze pomocí transpilátoru.

```
// ES6 modul
import express from "express"

// ES6 šipková funkce
const arrowFunction = () => {
    return 'arrowFunction'
}

// ES6 generátor
function* generator() {
    yield 'Hello'
    yield 'World!'
}

// ES8 operátor odpočinku neboli šíření
const object = {
    prop: 'ES8'
}

const newObject = { ...object }
```

Ukázka kódu 1: ES6 modul, šipková funkce a ES8 operátor odpočinku neboli šíření. Zdroj: vlastní zpracování

3.1.3 TypeScript

Podle knihy (Choi, 2020) JavaScript není moc vhodný pro vytvoření velkých škálovatelných projektu a je dost těžké psát na něm čistý kód. Toto je způsobeno nedostatkem možnosti uvádění datových typu. Proto v roce 2012 Microsoftem byl vytvořen TypeScript, který je nadstavbou pro JavaScript. TypeScript přinesl možnosti známé z objektově orientovaného programování, jako jsou statické typování, třídy, moduly, rozhraní atd.

Díky tomu programátoři mají všechny výhody jako při práci s ostatními objektově orientovanými jazyky, do nichž patří rychlé napovídání a automatická kontrola kódu, které výrazně usnadňují a zrychlují vývoj, jak to píše (Choi, 2020).

3.1.4 Transpilátory

Podle autora (Gupta, 2021) transpilátory jsou nástroje, které umožňují ekvivalentně přeložit zdrojový kód jednoho programovacího jazyka na jiný se stejnou úrovní abstrakce. Tradiční kompilátory na rozdíl od transpilátoru kompilují jazyk na vysoké úrovni abstrakce do jazyka na nízké úrovni. Dobrými příklady transpilátorů jsou Babel, který konvertuje ES6 kód do ES5, a TypeScript pro konverzi kódu do JavaScriptu.

3.1.5 Klíčové odlišnosti jazyků

Tabulka 1: Klíčové odlišnosti jazyků. Zdroj: (Pang, 2021)

Kritérium	JavaScript	TypeScript
Základ	Prostý skriptovací jazyk pro vývoj dynamického obsahu webových stránek, který je podporován všemi prohlížeči	Nadstavba JavaScriptu pro škálovatelnost a čistotu kódu ve velkých projektech
Typování	Dynamické typování	Statické typování
Velikost projektu	Nejlépe pro malé projekty	Nejlépe pro velké projekty
Přístupnost ve prohlížeči	Může se používat přímo ve prohlížeči	Musí být zkonvertován na JavaScript prostřednictvím Transpilátoru
Knihovny	Všechny JS knihovny fungují automaticky	JS knihovny a JS kód fungují automaticky, protože TypeScript je nadstavba

Podle článku (Pang, 2021) ačkoliv JavaScript nemá výhody, které má TypeScript, neznamená toto, že by se měl být jim úplně nahrazen. JavaScript je vhodnější při vývoji malých projektu, kde flexibilita a rychlost jsou prioritní. Je to způsobeno, tím že běží přímo v prohlížeči, je lehce spustitelný a laděný.

Na druhou stranu, velké škálovatelné projekty, kde je potenciál velkého počtu chyb, měly by se vyvíjet na TypeScriptu. Protože dovoluje odhalit chyby během doby kompilace a standardizovat kód, jak je to napsáno autorem (Pang, 2021).

3.2 Backend

Podle článku (Front End vs Back End) *backend* je serverová část webových aplikací. Skládá se ze serveru, který má spojení s databází. Má za úkol odpovídat na dotazy klientské části aplikace. Je zodpovědný za bezpečnost a zálohování dat. Vývoj *backendu* je hodně zaměřen na rychlost a reakci webu. Programovací a skriptovací jazyky používané pro vývoj *backendu* jsou Java, .Net, JavaScript, Python, PHP, Ruby, Perl, atd. Jazyky se používají k vytváření dynamických webů, které neustále zpracovávají a aktualizují data.

3.2.1 Hypertext Transfer Protocol

HTTP je jednoduchý protokol sloužící pro přenos obsahu na webu. K provedení funkce přenosu využívá spolehlivý protokol transportní vrstvy TCP (Transmission Control Protocol). Hlavním principem je odesílání odpovědi na požadavky klientu. Každý požadavek musí obsahovat metodu, adresu s parametry, hlavičku a případně tělo, jak se to píše ve článku (Úvod do HTTP a HTTPS, 2022).

Podle (Úvod do HTTP a HTTPS, 2022) HTTPS (Hypertext Transfer Protocol Secure) je zabezpečená verze HTTP, jehož implementuje pomocí kryptografického protokolu TLS (Transport Layer Security). Kromě další konfigurace potřebné k nastavení TLS je použití HTTPS protokolu stejné jako u HTTP.

```
metoda a adresa
POST http://api.example.com/movies

hlavička
User-Agent: cojeapi/1.0 (+https://cojeapi.cz)
Authorization: c630c64829efbd162eeb5f9e9f878e9ef3fcf757
Content-Type: application/json

tělo
{
  "title": "Ariel",
  "director": "Aki Kaurismäki",
  "year": 1988,
  "duration": 73
}
```

Obrázek 1: Příklad HTTP požadavku. Zdroj: (Javorek, 2020)

3.2.1.1 HTTP metody

Podle článku (HTTP request methods, 2021) HTTP definuje sadu metod požadavku, které označují požadovanou akci pro daný zdroj. Existují bezpečné a idempotentní metody.

- bezpečné – metody, které nesmí měnit stav serveru, tzn. můžou provádět jenom operace pro čtení (ang. read-only). Všechny bezpečné metody jsou také idempotentní.
- idempotentní – metody, jejichž opakované volání působí stejně a nechávají server ve stejném stavu.

Tabulka 2: HTTP metody. Zdroj: (HTTP request methods, 2021)

Metoda	Popis	Bezpečnost	Idempotence
GET	Požaduje reprezentaci zdroje	Ano	Ano
HEAD	Identická metodě GET, ale server navrací tělo zprávy	Ano	Ano
POST	Odesílá data na server. Vytvoří zdroj	Ne	Ne
PUT	Nahradí všechny aktuální reprezentace cílového zdroje novým datovým obsahem	Ne	Ano
DELETE	Smaže zadaný zdroj	Ne	Ano
CONNECT	Vytvoří klientu TCP tunel	Ne	Ne
OPTIONS	Popisuje možnosti komunikace pro cílový zdroj	Ano	Ano
TRACE	Sleduje dotaz zasílaný na server	Ano	Ano
PATCH	Částečně upraví zdroj	Ne	Ne

3.2.1.2 Stavové kódy HTTP

Ve článku (HTTP response status codes, 2022) se píše, že stavové kódy HTTP označují, jak byl konečně HTTP požadavek serverem zpracován.

- 100–199: informační odpověď
- 200–299: úspěšná odpověď
- 300–399: odpověď přesměrování
- 400–499: odpověď chyby klientu
- 500–599: odpověď chyby serveru

3.2.2 JavaScript Object Notation

JSON formát vznikl kolem roku 2020 a jeho autorem je Douglas Crockford. Rychle se stál nejoblíbenějším formátem mezi vývojáři díky své stručnosti, kterou neměl předchůdce XML. JSON je navržený tak, aby připomínal javascriptové objekty. Velmi často se používá pro předání dat v tělech HTTP požadavku a odpovědi, jak to píše například autor (Javorek, 2020).

3.2.3 Application Programming Interface

Podle článku (Application Programming Interface, 2020) API se stará o zpřístupnění dat a funkcí ze serveru či programu pro externí vývojáře. Umožňuje komunikaci mezi různými

služby a produkty a také jejích vzájemné využití dat prostřednictvím zdokumentovaného rozhraní. Vývojáři můžou jednoduše používat rozhraní a nepotřebují vědět jeho implementaci, což je jedním z hlavních cílů API. Používání API za poslední desetiletí prudce vzrostlo, proto by nebyla dnes bez API většina nejpoužívanějších aplikací možná.

Princip fungování API:

1. Klient zahájí volání API pro načtení dat prostřednictvím požadavku. Tento požadavek bude zpracován z klientu na server přes Uniform Resource Identifier (URI) a zahrne metodu požadavku, hlavičku a případně jeho tělo.
2. Po obdržení platného požadavku API zavolá externí program nebo server.
3. Server odešle API zpracována požadovaná data.
4. API odešle tato data klientu.

3.2.4 REST API

REST API je rozhraní navržené podle architektury REST (Representational state transfer). REST byl poprvé definován v roce 2000 počítačovým vědcem Royem Fieldingem v rámci jeho doktorské práce. Tato architektura poskytuje relativně vysokou úroveň flexibility a svobody při vývoje API. Snaží se také co nejvíce spolehnout na vlastnosti a schopnosti protokolu HTTP. Proto dnes vývoj API ve stylu REST je nejběžnějším, jak je to napsáno ve článku (REST APIs, 2021).

3.2.4.1 Omezení REST API

Podle autora (Dogilo, 2018) REST API může být vyvinuto v jakémkoliv programovacím jazyce. Musí však splňovat šest architektonických omezení:

1. Klient – server

   V knize (Dogilo, 2018) se píše, že omezení klient – server říká, že klient musí být zcela nezávislý na serveru. Hlavním principem tohoto omezení je proces oddělení zodpovědností (ang. separation of concerns). Toto umožňuje oddělení *frontend* kódu zodpovědného za reprezentaci webu od kódu na straně serveru, který by se měl starat o ukládání a zpracování dat.

2. Bezestavovost

   REST API má být bezstavové, tzn. že každý požadavek musí obsahovat všechny informace nezbytné pro jeho zpracování. Server nesmí ukládat žádná data související s klientským požadavkem, jak se to píše ve článku (REST APIs, 2021).

3. Cachování

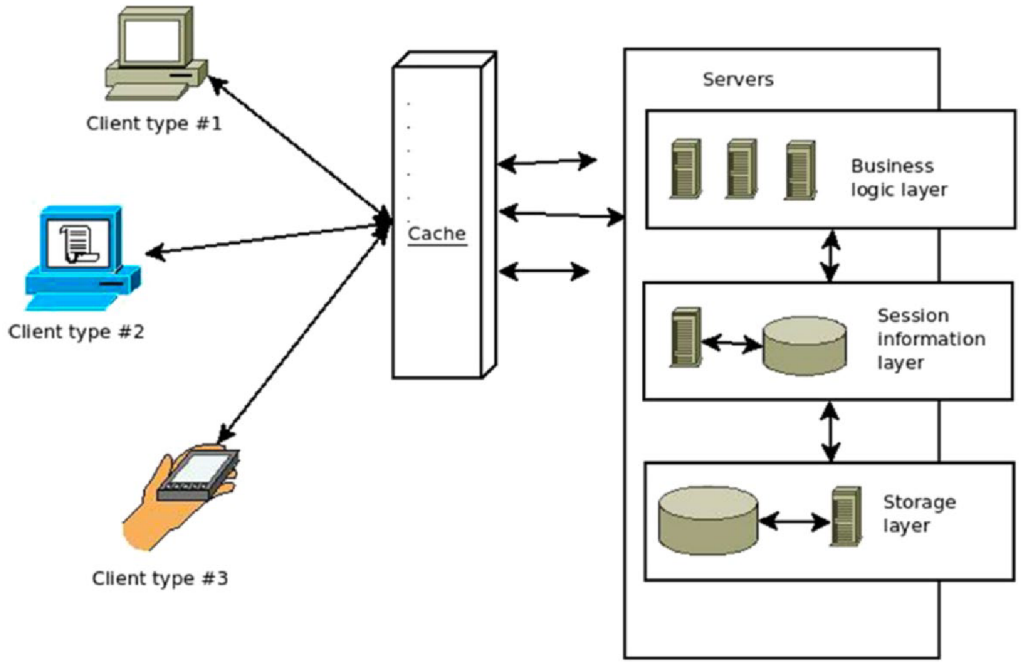
   Podle autora (Dogilo, 2018) každá odpověď na požadavek, pokud je to možné, musí být explicitně nebo implicitně nastavena jako cacheovatelná. Cachováním odpovědí lze například dosáhnout menšího počtu interakci se serverem a databází a následně zjevně zlepšit výkonnost klientské aplikace. Je však nutné sledovat, aby data v mezipaměti nebyla zastaralá. Proto toto omezení záleží na typu implementovaného systému .

4. Jednotné rozhraní

   Toto omezení říká, že všechny požadavky na stejný zdroj by měly vypadat stejně, bez ohledu na to, odkud požadavek pochází. REST API by mělo zajišťovat, že stejná data, budou patřit pouze k jednomu jednotnému identifikátoru zdroje (URI). Zdroje by neměly být příliš obsáhlé, ale měly by mít v sobě všechny požadované klientem informace, jak je to napsáno ve článku (REST APIs, 2021).

5. Vrstevnatost

   Vrstevnatost umožňuje rozdělit částí systému do vrstev, z nichž každé lze používat jenom nižší a sdělovat svůj výstup pouze vyšší vrstvě. Tím se dá zjednodušit celkovou složitost systému a udržovat jeho částí pod kontrolou, jak to píše (Dogilo, 2018).



Obrázek 1: Příklad aplikace s různými vrstvami. Zdroj: (Dogilo, 2018)

6. Kód na vyžádání

   Podle článku (REST APIs, 2021) REST API ve většině případech odesílá statické zdroje. Ale někde odpovědi mohou obsahovat také spustitelný kód. V tomto případě by měl kód běžet pouze na vyžádání klienta.

3.3 Frontend

Frontend je klientská část webových aplikací, která zobrazují výstupy z backednu, s nimž komunikuje prostřednictvím požadavku uživatele, vyjadřuje autor (Strelec). Obvykle běží v prohlížeči a slouží pro zobrazení uživatelského rozhraní a další zpracování, ke kterému může dojít na klientu, jako je například čtení či zápis *cookie*, píše autor článku (Chamikara, 2019).

Některé webové aplikace mají vyvíjen primárně *frontend* a dost triviální *backend*. Toto se často stává u jednodušších webových stránek a progresivních webových aplikací, jak píše například (Strelec).

Webovou aplikaci se dá spustit pomocí URL adresy v prohlížeči, který odešle HTTP požadavek na server. Po zpracování požadavku server vrátí zdrojový kód pro renderování aplikace na klientu, píše autor (Strelec).

Podle autora (Chamikara, 2019) interakce mezi uživatel a webovou aplikací je možná díku třem komponentům:

- HTML (Hypertext Markup Language) je hypertextový značkovací jazyk, který definuje strukturu webových stránek.
- CSS (Cascading Style Sheets) je jazyk, který určuje styl zobrazeného obsahu na stránkách.
- JavaScript umožňuje interaktivitu webových stránek

3.3.1 Objektový model dokumentu

Podle knihy (Haverbeke, 2018) objektový model dokumentu (ang. Document Object Model) je reprezentace HTML dokumentu, kde každý prvek představuje objekt, s nimž lze interagovat pomocí JavaScriptu .

DOM používá stromovou strukturu, kde každý uzel může odkazovat na jiné uzly, které zase mohou mít své vlastní potomky, jak to píše autor (Haverbeke, 2018).

3.3.2 Frontendové frameworky

Podle článku (Introduction to client-side frameworks, 2018) s rostoucí složitostí a možností interakce ve webových aplikacích se začaly vyvíjet různé javascriptové knihovny. Každá z nich obsahuje nástroje, které řeší určitou sadu problémů. Snahou vývojářů standardizovat kód, jenž by určoval strukturu celého projektu, objevily se frameworky – knihovny, které determinují způsob vývoje aplikace.

Frameworky umožňují homogenitu a předvídatelnost ve webových aplikacích. Tyto vlastnosti jsou zásadní pro životnost softwaru a pomáhají vyvíjet škálovatelné a stálé udržovatelné projekty, píše se ve článku (Introduction to client-side frameworks, 2022).

Frontendové frameworky mají za cíl abstrahovat interakci s prohlížečem a DOMem. Toto umožňuje deklarativně definovat a interagovat s DOM prvky na vyšší úrovni, jak to píše autor (Copes, 2020).

Používají se nejčastěji pro vytvoření jednostránkových webových aplikací (ang. Single-page application). Tento druh aplikací nenačítá a nevykresluje pro každou stránku nový HTML soubor. Místo toho načte jediný HTML a neustále v něm aktualizují DOM, píše se ve článku (Introduction to client-side frameworks, 2022).

Podle autora článku (Copes, 2020) mezi nejpoužívanějšími *frontendovými frameworky* patří Ember, Angular, Vue a React.

3.3.3 React

React je javascriptový *framework* od společnosti Facebook pro tvorbu *frontendu*. Umožňuje vyvíjet uživatelská rozhraní deklarativním a komponentně řízeným způsobem. Styl programování v Reactu je čerpán hlavně z funkčních a objektově orientovaných programovacích konceptů. Hlavními prvky pro vývoj rozhraní jsou komponenty, které vykreslovací systém Reactu spravuje a udržuje tím zobrazení aplikace v synchronizaci. Komponenty by měly snadno integrovat mezi sebou, řídit se předvídatelným životním cyklem a mít si možnost udržovat svůj vlastní vnitřní stav, píše autor (Tielens, 2018).

3.3.4 Next.js

Next.js je vývojový *framework* založený na Reactu. Je vyvinut a podporován společnosti Vercel (dříve Zeit). Hlavní vlastnosti Next.js je podpora různých metod vykreslení, které běžný React nemá. Pro napsaní stylu aplikace používá Styled-JSX, který umožňuje psát CSS jako JavaScript kód (Durgesh, 2021). Některé vlastnosti jsou v Next.js standardizované, jako jsou například routování a metody pro komunikaci se serverem. Prostřednictvím modulárního balíčkovače Webpacku a ostatních knihoven *framework* umožňuje minimální nastavení potřebné konfigurace a rychlý pohodlný vývoj, jak to píše autor článku (Ossera, 2021).

3.3.4.1 Metody vykreslení

React používá vykreslování na straně klienta, tzn. obsah se vykreslí, až klient zpracuje přijatý JavaScript kód. Proto uživatel musí mít povolený JavaScript ve svém prohlížeči. Metody vykreslení Next.js umožňují místo toho předběžné vykreslování či generaci HTML stránek. Toto způsobí lepší výkonnost webové aplikace a lepší indexovatelnost a procházení webových stránek, což je nezbytné pro optimalizaci pro vyhledávače (SEO). Díky automatickému dělení kódu (ang. *automatic code splitting*) je možné načítat pouze ty soubory JavaScript a CSS, které jsou nezbytné pro danou stránku. Toto způsobí mnohem rychlejší načtení webové stránky, protože prohlížeč má ke stažení méně. Po načtení stránky spustí se JavaScript pro plnou interakci s aplikací. Tento proces se nazývá hydratace, jak vyjadřuje autor (Correia, 2021).

Statické generování

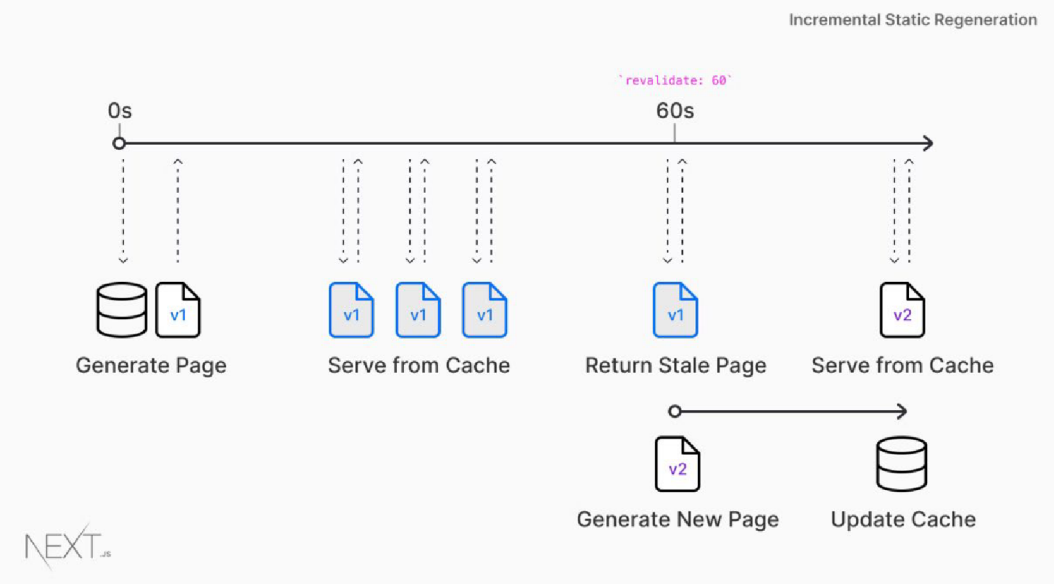
Metoda statického generování (ang. *Static Site Generation* (SSG)) generuje HTML stránky během doby sestavení aplikace. Jestli stránka nepotřebuje žádná externí data, tak bude vygenerována automaticky. V případě, že stránka musí mít nějaká data z externího API, Next.js si stáhne a načte tato data předem a pak vygeneruje HTML. Celý vygenerovaný obsah se uchová do CDN a na každý požadavek určité stránky se odešle její verze uložená v mezipaměti. Toto výrazně zlepšuje výkonnost aplikace, jak píše autor (Sangtiani, 2022).

```
export async function getStaticProps() {
  // Stažení dat z externího API
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Předání dat stránce skrz vlastností
  return {
    props: {
      posts,
    },
  }
}
```

Ukázka kódu 2: Příklad metody statického generování. Zdroj: (GetStaticProps)

Inkrementální statická regenerace

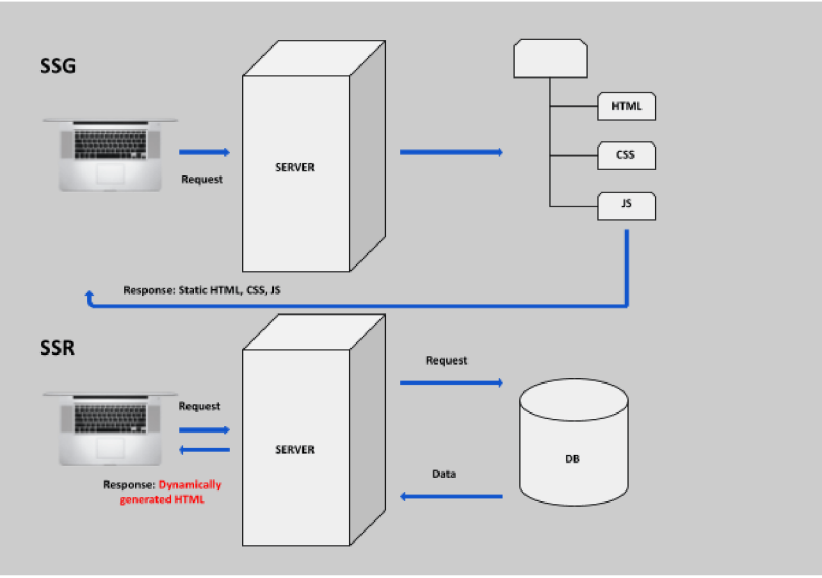
Podle autora článku (Sangtiani, 2022) statické generování není vhodné pro stránky s dynamickým obsahem, protože HTML soubory se vygenerují jednou během doby sestavení. Inkrementální statická regenerace (ang. *Incremental Site Regeneration* (ISR)) řeší tento problém tím, že generuje HTML v pravidelných intervalech v době běhu aplikace. Při požadavku se stránka vygeneruje staticky s inicializací určitého intervalu. Na všechny požadavky před koncem intervalu se budou odesílat stránky z mezipaměti. Po ukončení intervalu s prvním novým požadavkem se spustí regenerace stránky a na všechny ostatní požadavky se už budou odesílat nově vygenerované HTML.



Obrázek 2: diagram toku požadavku v inkrementální statické regeneraci, zdroj: (Incremental Static Regeneration)

Vykreslení na straně serveru

Vykreslení na straně serveru (ang. *Server-Side Rendering* (SSR)) je nejvhodnější metoda pro stránky s dynamickým obsahem, který se často mění. Vykonává se vždycky na straně Next.js serveru, který má za cíl generovat HTML stránku pro každý požadavek. Je nejpomalejší proti ostatním metodám, protože neukládá vygenerované stránky do mezipaměti, jak píše například (Sangtiani, 2022).



Obrázek 3: porovnání statického generování a vykreslení na straně serveru. Zdroj: (Ossera, 2021)

3.3.5 Progresivní webová aplikace

Podle článku (Koďousková, 2021) progresivní webové aplikace (PWA) jsou takové aplikace, které implementují v sobě funkcionality tradičních webových a nativních aplikací. Nativní aplikace, jež vyvíjí pouze pro jednu konkrétní platformu (například Android či IOS), vnášejí do PWA důležité funkcionality, jako jsou například možnost práce *offline*, push notifikace a přístup ke hardwaru zařízení. Progresivní webovou aplikaci se dá také umístit na ploše zařízení s její vlastní ikonou. Častou formou PWA je jednostránková webová aplikace.

3.3.5.1 Charakteristiky PWA

Níže jsou uvedené charakteristiky, se kterými je nutné počítat během vývoje progresivní webové aplikace, zpracováno podle článku (Nyakundi, 2021).

- Responzivita
  Moderní svět technologií produkuje hodně zařízení, které mají různou velikost obrazovky. A vývojáři mají na starosti zajistit, že uživatel bude pohodlně využívat produkt. Proto je nutné vyvíjet aplikaci tak, aby obsah byl dostupný z jakékoliv obrazovky.
- Instalovatelnost
  Uživatelé mají tendenci používat více nainstalovanou aplikaci než navštěvovat webovou stránku v prohlížeči. Proto s PWA je možné poskytnout uživatelům vzhled a dojem, že používají běžnou aplikaci.
- Práce v *offline* režimu
  Možnost využívat aplikaci, i když není internet připojení, je mnohem lepší než zobrazovat výchozí *offline* stránku. Dobrým příkladem je Twitter, který dovoluje uživatelům zase prohlédnou si záznamy, jež mohli prominout.
- Objevitelnost
  PWA jako webová aplikace musí mít objevitelné stránky pro vyhledavače. Toto umožňuje zvětšit návštěvnost aplikace. Je to také velkou výhodou oproti nativním aplikacím, které tuto možnost nemají.
- Vzhled
  Vzhled PWA by měl působit dojmem běžné nativní aplikace.

- Multiplatformnost
  Progresivní webová aplikace musí fungovat ve všech prohlížečích a operačních systémech.

3.3.5.2 Web App Manifest

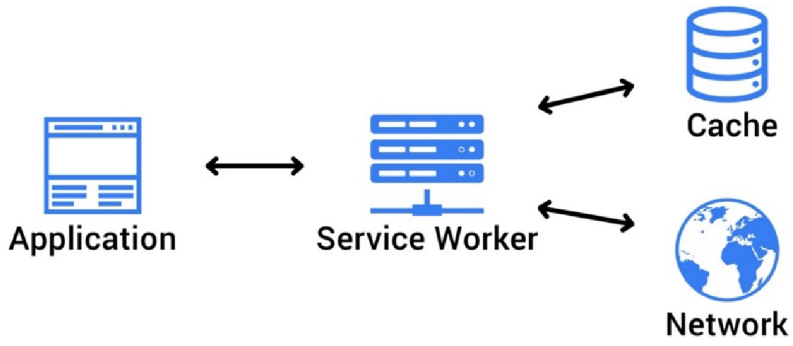
Podle článku (Web app manifests, 2022) manifest webové aplikace poskytuje nezbytnou informaci o PWA. Je to soubor ve formátu JSON. Umožňuje stahovat aplikaci a zobrazovat ji v podobě nativní aplikace. Manifest obsahuje název aplikace, autora, ikony, verzi, popis a seznam všech potřebných zdrojů. Připojuji se ve HTML souboru pomocí prvku *<link>* ve hlavičce.

```
<link rel=”manifest” href=”manifest.json”>
```

Ukázka kódu 3: Připojení manifestu. Zdroj: (Web app manifests, 2022)

3.3.5.3 Service Worker

Service Worker je skript, který slouží pro propojení PWA s prohlížečem. Vykonává hodně důležitých úloh na pozadí, do nichž patří aktualizace a cachování dat, nastavení push notifikací atd. Umožňuje práci aplikace v režimu *offline* díky různým strategiím cachování dat (Koďousková, 2021).



Obrázek 4: Service Worker. Zdroj: (Onyango, 2018)

4 Vlastní práce

Tato kapitola popisuje proces vytvoření dané aplikace. První podkapitola je věnována analýze. Především byla provedena analýze požadavku a funkcionalit. Pak následovala analýza požadavku na role a výběr vhodných technologií. Druhá podkapitola se zabývá návrhem systémy a případu jeho užití. V poslední kapitole je popsán proces implementace této aplikace.

4.1 Analýza

Nezbytnou částí vytváření jakékoliv aplikace je analýza požadavků a funkcionalit. Analýza umožňuje určit správný postup při návrhu a vývoji systému a předejít spoustě chyb. Pouze po provedení analýzy lze přejít k návrhu a implementaci aplikace.

4.1.1 Požadavky aplikace

Z cíle a předpokladu pro vytvoření dané aplikace plynou další požadavky:

4.1.1.1 Autentizace

- Autentizace musí být bezpečná
- Uživatel bude konečně registrován až po ověření emailem
- Uživatel by měl zůstat přihlášen v aplikaci až se nerozhodne odhlásit sám
- Uživatel má možnost přihlašování z různých zařízení

4.1.1.2 Autorizace

- Implementace rolí administrátorů, redaktoru, moderátoru, autoru a uživatelů
- Každá role má svůj význam a povolené možnosti
- Jeden uživatel aplikace může mít i více roli

4.1.1.3 Vytvoření recenze

- Autor má k dispozici seznam existujících filmů a seriálu pro vytvoření recenze
- Autor má možnost nahrát zobrazení ke své recenzi
- Autor může následně upravovat svou recenzi

4.1.1.4 Zobrazování recenzí

- Na hlavní stránce se zobrazují nejnovější recenze
- Je možné filtrování recenzí podle autoru a názvu filmu
- Pro již otevřené stránky by mělo fungovat cachování, aby následně bylo možné číst recenzi i bez přístupu do síti

4.1.1.5 Vzhled aplikace

- Aplikace má moderní a přehledný vzhled
- Aplikace má být přístupná z mobilu
- Použití knihovny Material UI

4.1.1.6 Aplikace obecně

- Funguje indexace stránek s recenzemi, aby vyhledávače mohly načíst obsah recenze (SEO optimalizace)
- Aplikaci je možné stáhnout z prohlížeče
- Stáhnutá aplikace má vlastní ikonu

4.1.2 Požadavky na role

Každá role musí mít svůj význam a omezení počet možností. Přidáním nové role uživateli získá další přístupy v aplikaci.

4.1.2.1 Administrátor

- Možnost přidávat či mazat role uživatelům

4.1.2.2 Autor

- Možnost vytvářet nové recenze
- Možnost opravovat a mazat své již stávající recenze
- Možnost měnit stav publikování svých recenzí

4.1.2.3 Redaktor

- Možnost opravovat text všech recenzí

- Možnost vidět všechny nepublikované recenze
- Možnost měnit stav publikování všech recenzí

4.1.2.4 Moderátor

- Možnost mazat komentáře všech uživatelů
- Možnost zakázat či zase povolit uživateli psát komentáře

4.1.2.5 Uživatel

- Možnost získat tuto rolí po dokončení registrace
- Možnost psát komentáře k recenzím
- Možnost měnit své heslo
- Možnost obnovit své heslo přes email

4.1.3 Výběr vhodných technologií

Před tím, než začít psát aplikaci je nutné určit vhodné technologie. Pro realizaci uvedených požadavků bude vyhovovat standardní rozdělení aplikace na prezentační vrstvu – *frontend*, a serverovou část pracující s databází – *backend*.

Aby výběr technologií pro tyto jednotlivé části byl správný, především je nutné řešit jakým způsobem budou mezi sebou komunikovat. Zvolena byla architektura rozhraní REST.

4.1.3.1 Backend

Pro napsaní *backendu* byl zvolen Node.js s *frameworkem* Express.js. Toto řešení se stává poslední dobou velice populární, protože umožňuje realizaci spolehlivého, rychlého a škálovatelného backendu a má dobrou dokumentaci. Také velkým plusem tohoto výběru je to, že celá aplikace může být vyvinuta v jednom jazyce, JavaScriptu nebo TypeScriptu.

Programovacím jazykem byl zvolen TypeScript, protože je vhodnější pro velké a škálovatelné projekty. Toto bude platit i pro *frontend*.

Jako databáze se bude používat PostgreSQL a pohodlnou práci s ní zabezpečí ORM Sequelize.

Pro validaci uživatelských vstupu bude použitá knihovna *express-validator*.

4.1.3.2 Frontend

Podle analýze požadavek aplikace by měla být následně SEO optimalizována. Pro toto je především nutné, aby stránka při kontrole vyhledávacími roboty měla všechny HTML prvky, což není možné při použitý obvyklých *frontendových frameworků*, jako jsou React, Angular nebo Vue.js. Proto bude vhodný *framework*, který umožňuje buď statické generování HTML stránek nebo vykreslení na straně serveru. Pro danou aplikaci byl zvolen *framework* Next.js, který má v sobě tyto obě možnosti.

Pro uchování stavu aplikace se bude používat Redux s knihovnou Redux Toolkit, která dovolí pracovat s Reduxem pohodlněji a zmenšení počet šablonového kódu. S ním se také bude využít Redux Saga *middleware* pro zpracování asynchronních úkolů pro vedlejší účinky.

Pro volání *backendového* API bude využitá knihovna Axios.

Jako *framework* pro tvorbu uživatelského rozhraní byl zvolen Material UI. Plyne tento výběr i z požadavku na vzhled této aplikace, který je charakteristický pro Material UI. Tato knihovna má také sadu předefinovaných komponent, což usnadní vývoje aplikace.

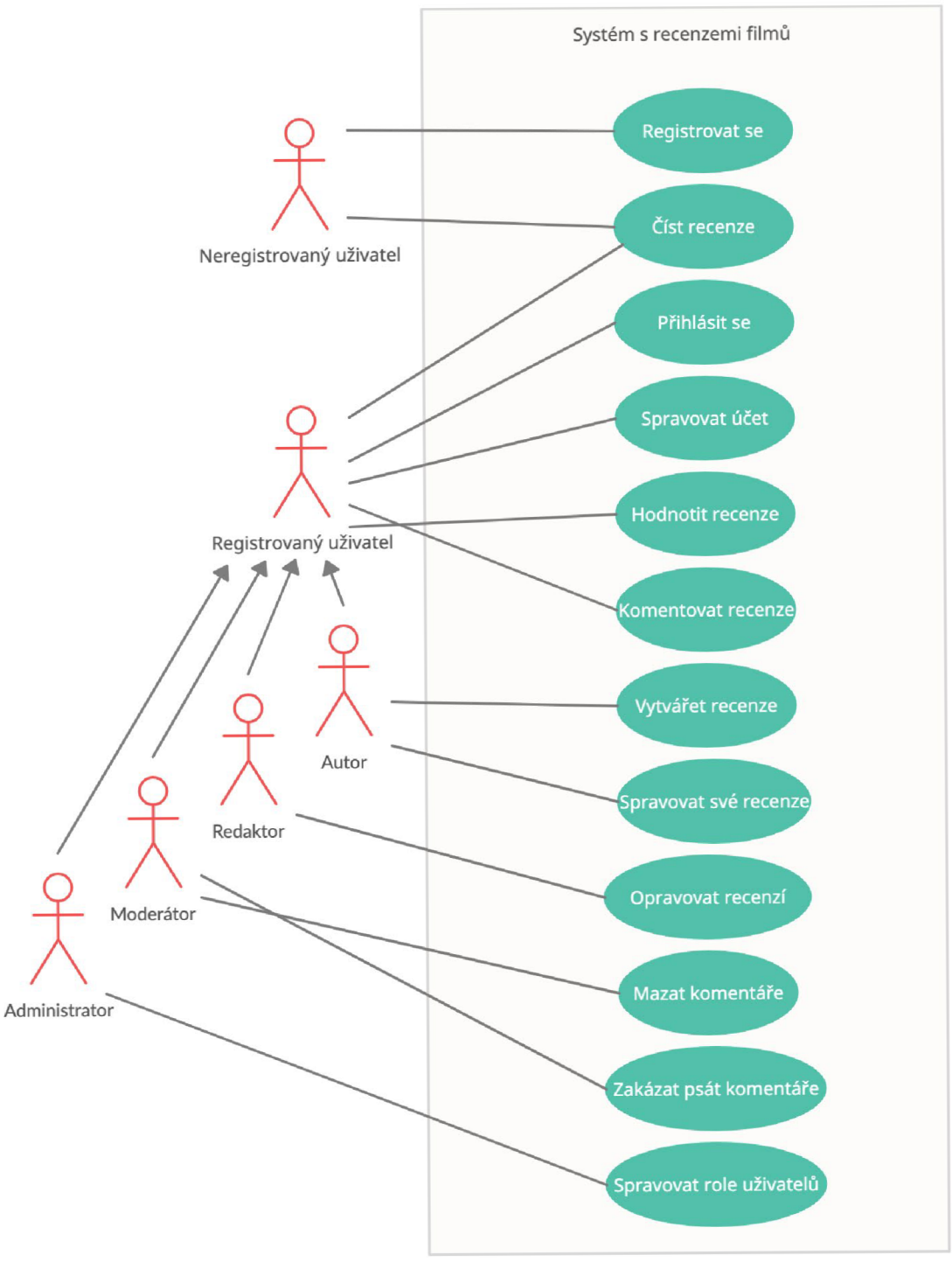
Pro implementaci funkcionalit PWA v Next.js aplikaci bude využitá knihovna *next-pwa*.

4.2 Návrh systému

Podle hotové analýzy požadavku a funkcionalit lze začít s návrhem systému a případy jeho užití.

4.2.1 Diagram případu užití

Na základě požadavku byl vytvořen diagram případů užití, který definuje činnost aktérů. Do nichž patří neregistrovaný uživatel, registrovaný uživatel, autor, redaktor, moderátor a administrátor.



Obrázek 5: diagram případů užití

4.2.2 Diagram tříd

Podle analýzy byl vytvořen diagram tříd. Hlavním cílem tohoto modelu je znázornit souvislosti mezi objekty, jejich strukturu a operace. Každá třída je navržena tak, aby její objekty co nejlíp odpovídaly skutečnosti.

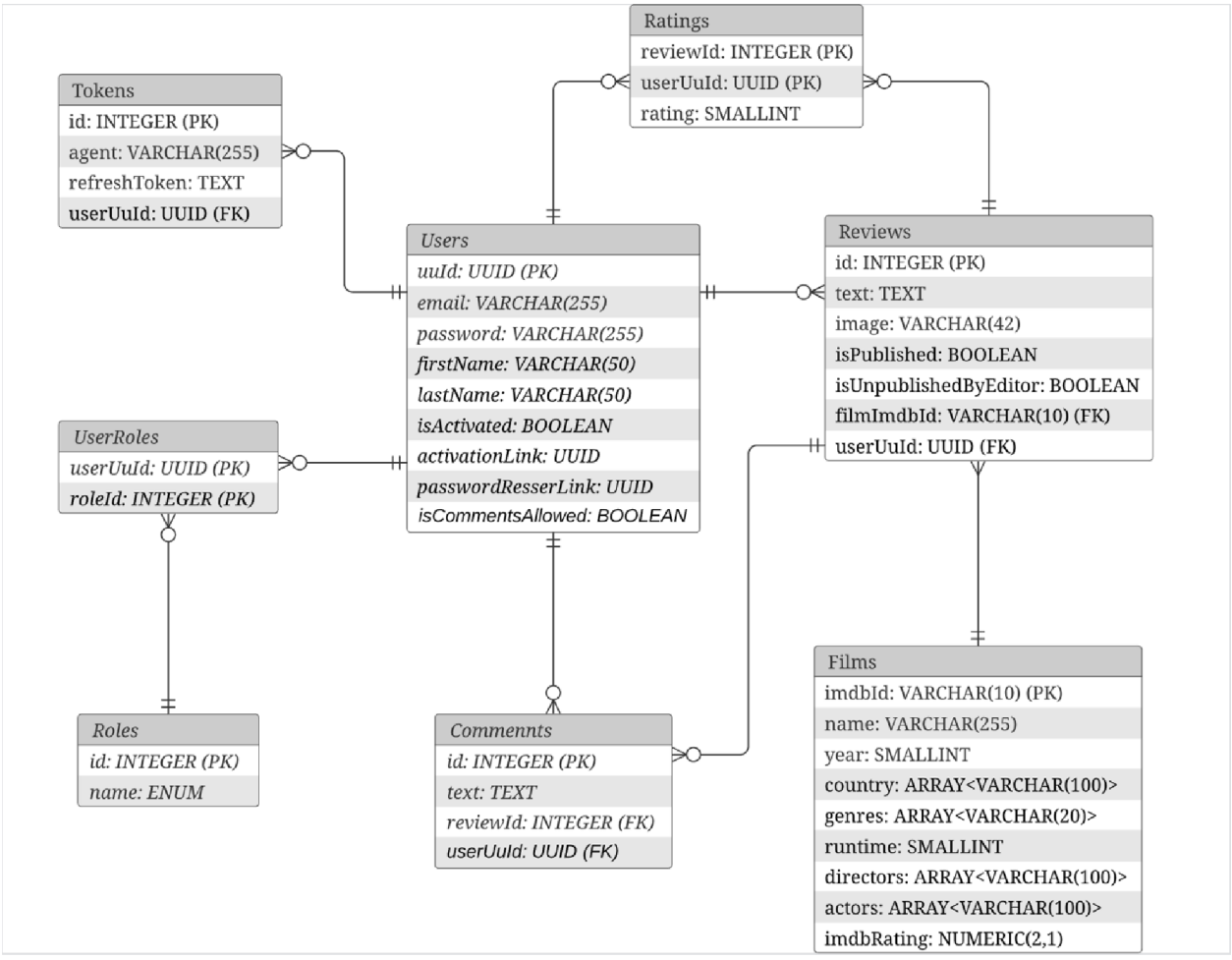
Obrázek 6: diagram tříd

4.2.3 Databázový model

Databázové tabulky a relace mezi nimi byly vytvořeny v souladu s analýzou požadavků. Návrh je reprezentován pomocí Entity Relationship diagramu.

Tabulky:

- Users – entity této tabulky reprezentují registrované uživatelé daného systému
- Roles – role, podle nichž je určen přístup k jednotlivým částem a funkcím aplikace
- UserRoles – spojovací tabulka pro přiřazení uživatelům rolí
- Tokens – obnovovací tokeny, pomocí nimž uživatel obnoví přístupový token
- Reviews – recenze napsané autorem
- Ratings – hodnocení uživatelů k recenzím
- Comments – komentáře uživatelů k recenzím
- Films – filmy, k nimž už existuje aspoň jedna recenze



Obrázek 7: databázový model

4.3 Implementace

Implementace této aplikace byla provedena podle návrhu. Celý vývoj byl uskutečněn na notebooku s operačním systémem macOS. Jako vývoje prostředí bylo použit WebStorm.

4.3.1 Struktura složek projekt

Celý projekt se dělí podle jeho hlavních části na dvě základní složky: server a client.

4.3.1.1 Server

Tato složka obsahuje Node.js Express server. Složka *„dist“* zahrnuje v sobě výchozí JavaScript soubory po zpracování typescriptového kódu transpilátorem. Největší složkou je *„node_modules“*, ve které se nacházejí všechny knihovny stažené pomocí balíčkovače Yarn. Ve složce *„static“* se ukládají statické soubory, jako jsou například obrázky. Soubor *„.env“* má v sobě sadu konstant nezbytných pro fungování aplikace. Velmi důležitou složkou obsahující základní kód pro fungování aplikace je *„src“*. Dál bude uvedená tabulka se strukturou této složky.

Tabulka 3: struktura složky src u backendu

Název	**Popis**
controllers	Tady jsou uložené kontrolery (řadiče), které slouží pro přijímání vstupních dat a předávání je příslušné aplikační logice.
db	Zde jsou všechny soubory spojené s konfigurací a vytvořením databáze.
dtos	Obsahuje třídy, které vytvářejí objekty pro přenos dat po síti.
errors	Zahrnuje v sobě pomocnou třídu s metodami pro vytváření objektů popisujících chyby, k nimž došlo během zpracování požadavků z klientu.
middlewares	Zde jsou umístěny *middleware* funkce, které jsou vyvolány směrovací vrstvou Express.js před konečným zpracováním požadavku.
routes	Obsahuje směrovače seřazené podle jejich účelu.
services	Tady jsou servisy s metodami, které obsahují logiku, již budou využívat příslušné kontrolery.
types	Zahrnuje v sobě pomocné TypeScript typy a rozhraní.
utils	Zde jsou umístěny různé pomocné funkce pro napsaní kódu.
validators	Obsahuje funkce pro validaci určitých atributu při volání *express-validator middleware*.

4.3.1.2 Client

Tato složka zahrnuje v sobě celou Next.js aplikaci. Ve složce *„.next“* se umisťují sestavené soubory po spouštění příkazu sestavení. Právě odsud bude běžet aplikace ve webovém serveru. Je tady také jako u backendu složka *„node_modules“* se všemi knihovnami. Ve složce *„public“* jsou umístěny statické soubory. Soubory *„.env“* a složka *„src“* mají stejný účel jako u serveru.

Tabulka 4: struktura složky src u frontendu

Název složky	**Popis**
components	Zde jsou umístěny React komponenty, které se budou vykreslovat na obrazovku. Většina komponent je roztříděná po dalším složkám, které se nazývají podle jejich účelu v aplikaci. Některé z nich se používají v kódu na více místech. Složka *„UI“* obsahuje pouze znovupoužitelné komponenty.
constants	Tady jsou uložené pomocné konstanty pro používaní v kódu.
hoc	Obsahuje komponenty vyššího řádu, které umožní používat logiku v ostatních komponentách znovu.
hooks	Zahrnuje v sobě vytvořené React *hooky* – funkce, které umožňují pracovat se stavem a životním cyklem funkčních komponent.
http	Zde jsou umístěny servisy s metodami pro volání API.
models	Obsahuje pomocné TypeScript typy a rozhraní.
pages	Tady se nacházejí stránky aplikace, podle jichž názvu se uskuteční směrování.
redux	Zahrnuje v sobě soubory s logikou pro fungování Redux.
styles	Tady jsou obecné CSS a Styled-JSX styly.
theme	Obsahuje soubor s definováním stylového tématu pro MaterialUI.
utils	Zde jsou umístěny různé pomocné funkce pro napsaní kódu.

4.3.2 Databáze

ORM Sequelize vytváří databázové tabulky pomocí abstrakcí, které se nazývají modely. Každý model je reprezentován v kódu pomocí třídy, která se dědí od třídy *„Model“*. Modely umožňují stanovit název tabulky, její sloupce a vazby s ostatními tabulkami.

Při použití Sequelize s TypeScriptem třída modelu přijímá dva generický typy. První typ, jejž musí třída implementovat, definuje všechny atributy tabulky. Druhý typ je určen k definici jen těch atributu, které jsou nezbytné pro vytváření entity.

```
export interface IUser {
    uuId: string
    email: string
    password: string
    firstName: string
    lastName: string
    isActivated: boolean
    activationLink: string
    passwordResetLink: string | null
    isCommentsAllowed: boolean
    createdAt: Date
    updatedAt: Date
}
```

Ukázka kódu 4: rozhraní *IUser*

Rozhraní *„IUser“* pro zvláštní případy obsahuje vlastnosti *„createdAt“* a *„updatedAt“*, které se implementují modelem automaticky. Proto pomocí utility *„Omit“* je nutné tyto vlastnosti vynechat.

```
type UserAttributes = Omit<IUser, 'createdAt' | 'updatedAt'>

interface UserCreationAttributes
    extends Optional<
        UserAttributes,
        | 'uuId'
        | 'isActivated'
        | 'activationLink'
        | 'passwordResetLink'
        | 'isCommentsAllowed'
    > {}

export class User
    extends Model<UserAttributes, UserCreationAttributes>
    implements UserAttributes
{
    uuId!: string
    email!: string
    password!: string
    firstName!: string
    lastName!: string
    isActivated!: boolean
    activationLink!: string
    passwordResetLink!: string | null
    isCommentsAllowed!: boolean
    roles!: IRole[]

    static associate() {
        User.belongsToMany(Role, {
            through: UserRole,
            as: 'roles',
            foreignKey: 'userUuId',
        })
        User.hasMany(Token, {
            as: 'tokens',
            foreignKey: { name: 'userUuId' },
        })
        User.hasMany(Review, {
            as: 'reviews',
            foreignKey: { name: 'userUuId' },
        })
        User.hasMany(Comment, {
            as: 'comments',
            foreignKey: { name: 'userUuId' },
        })
        User.hasMany(Rating, {
            as: 'ratings',
            foreignKey: { name: 'userUuId' },
        })
    }
}
```

Ukázka kódu 5: třída modelu *User*

Jakmile aplikace naváže spojení s databází, třída modelu spustí statickou metodu „init“, která vytvoří tabulku v databázi. Tato metoda přijímá jako první argument objekt s kompletním popisem sloupců včetně integritních omezení.

```
export = (sequelize: Sequelize, dataTypes: typeof DataTypes) => {
    User.init(
        {
            uuId: {
                type: dataTypes.UUID,
                defaultValue: dataTypes.UUIDV4,
                primaryKey: true,
                unique: true,
                allowNull: false,
            },
            email: {
                type: dataTypes.STRING,
                unique: true,
                allowNull: false,
            },
            password: {
                type: dataTypes.STRING,
                allowNull: false,
            },
            firstName: {
                type: dataTypes.STRING(50),
                allowNull: false,
            },
            lastName: {
                type: dataTypes.STRING(50),
                allowNull: false,
            },
            isActivated: {
                type: dataTypes.BOOLEAN,
                defaultValue: false,
                allowNull: false,
            },
            activationLink: {
                type: dataTypes.UUID,
                unique: true,
                allowNull: false,
            },
            passwordResetLink: {
                type: dataTypes.UUID,
                unique: true,
            },
            isCommentsAllowed: {
                type: dataTypes.BOOLEAN,
                defaultValue: true,
                allowNull: false,
            },
        },
        {
            sequelize,
            modelName: 'User',
        }
    )

    return User
}
```

Ukázka kódu 6: metoda *init* u třídy modelu *User*

Celá databáze byla vytvořena podle navrženého Entity Relationship diagramu.

4.3.3 Autentizace

Autentizace je proces, který ověřuje identitu uživatelé. Má na starosti zabezpečovat přístup k systému a chránit data. Dokáže zjistit, jestli daný uživatel je opravdu ten, za kterého se vydává.

Pro danou aplikaci byla zvolena JSON web token (JWT) autentizace. Je to moderní a bezpeční způsob pro výměnu informací pomocí tokenu. Tokeny se skládají z hlavičky, obsahu dat a podpisu, přičemž podpis je uskutečněn pomocí tajného hesla šifrovaného HMAC algoritmem.

V této aplikaci JWT autentizace je implementována na základě dvou tokenu: přístupového a obnovovacího. Pomocí přístupového tokenu je možné získat přístup k systému. Obnovovací token slouží pro aktualizaci přístupového tokenu, když ten je vypršen, a musí se ukládat do databáze.

4.3.3.1 Backend

Na *backendu* veškerá logika zodpovědná za autentizaci je umístěná ve servise *„UserService“*. Tuto logiku používá stejnojmenný kontroler, který poskytuje své metody příslušným adresám u směrovače *„userRouter“*.

```
userRouter.post(
    '/register',
    body('email')
        .custom(emailValidator())
        .isEmail()
        .withMessage('Invalid email'),
    body('password')
        .isLength({ min: 8, max: 50 })
        .withMessage(
            'Password should contain at least 8 and at most 32 characters'
        ),
    body('firstName')
        .isLength({ min: 1, max: 50 })
        .withMessage(
            'First name should contain at least 1 and at most 32 characters'
        ),
    body('lastName')
        .isLength({ min: 1, max: 50 })
        .withMessage(
            'Last name should contain at least 1 and at most 32 characters'
        ),
    UserController.register
)
```

Ukázka kódu 7: směrovač *userRouter*

U adresy „/register“ se před voláním metody kontroleru spustí *express-validator middleware*, který validuje určité atributy u těla požadavku. Umožňuje také nastavit chybovou hlášku při neúspěšné validaci pomocí metody *„withMessage“*. Atribut *„email“* se validuje pomocí vlastní funkce *„emailValidator“*, která hledá v databázi uživatele se stejným emailem. V případě nalezení takového uživatele, který má účet již ověřený přes email (tzn. atribut *„isActivated“* má hodnotu *true*), tak skončí validace chybovou hláškou.

```
export const emailValidator =
    (isExisting: boolean = true) =>
    async (email: string, { req }: Meta) => {
        const candidate = await User.findOne({
            where: { email },
            attributes: ['uuId', 'isActivated'],
        })

        req.user = candidate

        if (isExisting) {
            if (candidate?.isActivated) {
                return Promise.reject('This email is already taken')
            }
        } else {
            if (!candidate?.isActivated) {
                return Promise.reject(
                    'No account with this email has been found'
                )
            }
        }
    }
```

Ukázka kódu 8: funkce *emailValidator*

Po ukončení procesu validace middleware předá požadavek metodě kontroleru.

```
async register(req: Request, res: Response, next: NextFunction) {
    try {
        const errors = validationResult(req)

        if (!errors.isEmpty()) {
            return next(
                ApiError.badRequest('Validation error', errors.array())
            )
        }

        await UserService.register(req.body, req.user)
        return res.json({ message: 'Successfully registered' })
    } catch (e) {
        next(e)
    }
},
```

Ukázka kódu 9: metoda *register* kontroleru *UserController*

Tato metoda spustí funkci „validationResult“, která vrátí objekt. Tento objekt obsahuje pole s dalšími objekty popisující, v jakých atributech nebyla validace úspěšná a proč. V případě výskytu aspoň jedné neúspěšné validace, bude pomocí metody „next“ spuštěn následující v řetězci middleware, který je zodpovědný za zpracování a odesílání klientu chyb.

```
export const errorMiddleware: ErrorRequestHandler = (err, req, res, next) => {
    console.error(err)

    if (err instanceof ApiError) {
        return res
            .status(err.status)
            .json({ message: err.message, errors: err.errors })
    }

    return res.status(500).json({ message: 'Internal server error' })
}
```

Ukázka kódu 10: middleware *errorMiddleware*

O samotnou logiku registrace se stará metoda servisu, která získá všechny nezbytné vstupy pro vytvoření uživatele.

```
async register(inputs: IUserInput, candidate: User | undefined) {
    const { email, password, firstName, lastName } = inputs

    const activationLink = v4()

    const user = {
        email,
        password: await hash(password, 12),
        firstName,
        lastName,
        activationLink,
    }

    if (candidate) {
        await candidate.update(user)
    } else {
        await User.create(user)
    }

    await MailService.sendActivationMail(
        email,
        `${process.env.API_URL}/api/users/activate/${activationLink}`
    )
},
```

Ukázka kódu 11: metoda *register* servisu *UserService*

Heslo každého uživatele se uloží do databáze pomocí Bcrypt ve šifrované formě, která zahrnuje v sobě kryptografickou sůl. Čím vyšší je počet kol soli, tím bezpečnější je heslo a doba zpracování je delší.

Po uložení uživatelé do databáze se mu odešle email s odkazem pro aktivaci účtu ve formě UUID. Je toto uskutečněno díky servisu *„MailService“* a jeho metodě *„sendActiovationMail“*.

Přihlášení do systému uskuteční na základě emailu uživatele a hesla. Zaprvé servis zkontroluje, jestli takový aktivní uživatel existuje v databázi. Potom se provede kontrola hesla. V případě úspěšné kontroly se dále vytvoří objekt pro přenos dat po síti (DTO), který obsahuje údaje uživatelé včetně jeho rolí. Tento objekt se na konci pošle v odpovědi klientu ve formátu JSON. Pak servis *„TokenService“* buď vygeneruje nové, nebo obnoví tokeny uživatele na základě vlastnosti hlavičky požadavku *„user-agent“*, který představuje uživatelské zařízení. Takový přístup umožňuje uživatelům přihlašování s více různých zařízení. Nově vygenerovaný přístupový token má nastavenou dobu vypršení po 15 minutách, zatímco obnovovací token má po třech měsících. Těla tokenů obsahují stejný objekt pro přenos dat s údaji uživatele.

```
generateTokens(payload: UserDto) {
    const accessToken = sign(payload, process.env.JWT_ACCESS_SECRET!, {
        expiresIn: '15min',
    })

    const refreshToken = sign(payload, process.env.JWT_REFRESH_SECRET!, {
        expiresIn: '90d',
    })

    return {
        accessToken,
        refreshToken,
    }
},
```

Ukázka kódu 12: metoda *generateTokens* servisu *TokenService*

```
async login(
    inputs: Omit<IUserInput, 'firstName' | 'lastName'>, agent: string
) {
    const { email, password } = inputs
    const user = await User.findOne({
        where: { email },
        attributes: [
            'uuId',
            'email',
            'password',
            'firstName',
            'lastName',
            'isActivated',
            'isCommentsAllowed',
        ],
        include: [
            {
                model: Role,
                as: 'roles',
                attributes: ['name'],
                through: {
                    attributes: [],
                },
            },
        ],
    })
    if (!user || !user.isActivated) {
        throw ApiError.badRequest('Incorrect email or password')
    }

    const isPassEquals = await compare(password, user.password)
    if (!isPassEquals) {
        throw ApiError.badRequest('Incorrect email or password')
    }

    const userDto = new UserDto(user)
    const tokens = TokenService.generateTokens({ ...userDto })
    const tokenFromDb = await Token.findOne({ where: { agent } })

    if (tokenFromDb) {
        await Token.update(
            { refreshToken: tokens.refreshToken },
            { where: { userUuId: userDto.uuId, agent } }
        )
    } else {
        await Token.create({
            agent,
            refreshToken: tokens.refreshToken,
            userUuId: userDto.uuId,
        })
    }
    return {
        tokens,
        user: userDto,
    }
},
```

Ukázka kódu 13: metoda *login* servisu *UserService*

Poté *„TokenService“* pomocí metody *„setCookie“* přidá tokeny do odpovědi jako HttpOnly cookie. Tento druh cookie není přístupný v prohlížeči pro JavaScript, což dělá autentizaci bezpečnější proti XSS útokům. Doba vypršení cookie je stejná jako u tokenu.

```
setCookie(tokens: ITokens, res: Response) {
    const { accessToken, refreshToken } = tokens

    res.cookie('accessToken', accessToken, {
        httpOnly: true,
        maxAge: 1000 * 60 * 15,
        domain: process.env.DOMAIN,
        sameSite: 'none',
        secure: true,
    })

    res.cookie('refreshToken', refreshToken, {
        httpOnly: true,
        maxAge: 1000 * 60 * 60 * 24 * 90,
        domain: process.env.DOMAIN,
        sameSite: 'none',
        secure: true,
    })
},
```

Ukázka kódu 14: metoda *setCookie* servisu *TokenService*

Pro udržení uživatele v systému po uzavření aplikaci v prohlížeči a pak nové její návštěvě se bude uživatel automaticky přihlášen na základě údajů z obnovovacího tokenu. Tuto logiku obsahuje metoda *„refresh“* u *„UserService“*.

```
async refresh(refreshToken: string) {
    if (!refreshToken) {
        throw ApiError.badRequest('No refresh token')
    }

    const userData = TokenService.validateRefreshToken(
        refreshToken
    ) as UserDto

    const tokenFromDb = await Token.findOne({ where: { refreshToken } })

    if (!userData || !tokenFromDb) {
        throw ApiError.badRequest('Invalid refresh token')
    }

    const user = await User.findOne({
        where: { uuId: userData.uuId },
        attributes: [
            'uuId',
            'email',
            'firstName',
            'lastName',
            'isActivated',
            'isCommentsAllowed',
        ],
        include: [
            {
                model: Role,
                as: 'roles',
                attributes: ['name'],
                through: {
                    attributes: [],
                },
            },
        ],
    })

    const userDto = new UserDto(user!)

    const tokens = TokenService.generateTokens({ ...userDto })

    await tokenFromDb.update({ refreshToken: tokens.refreshToken })

    return {
        tokens,
        user: userDto,
    }
},
```

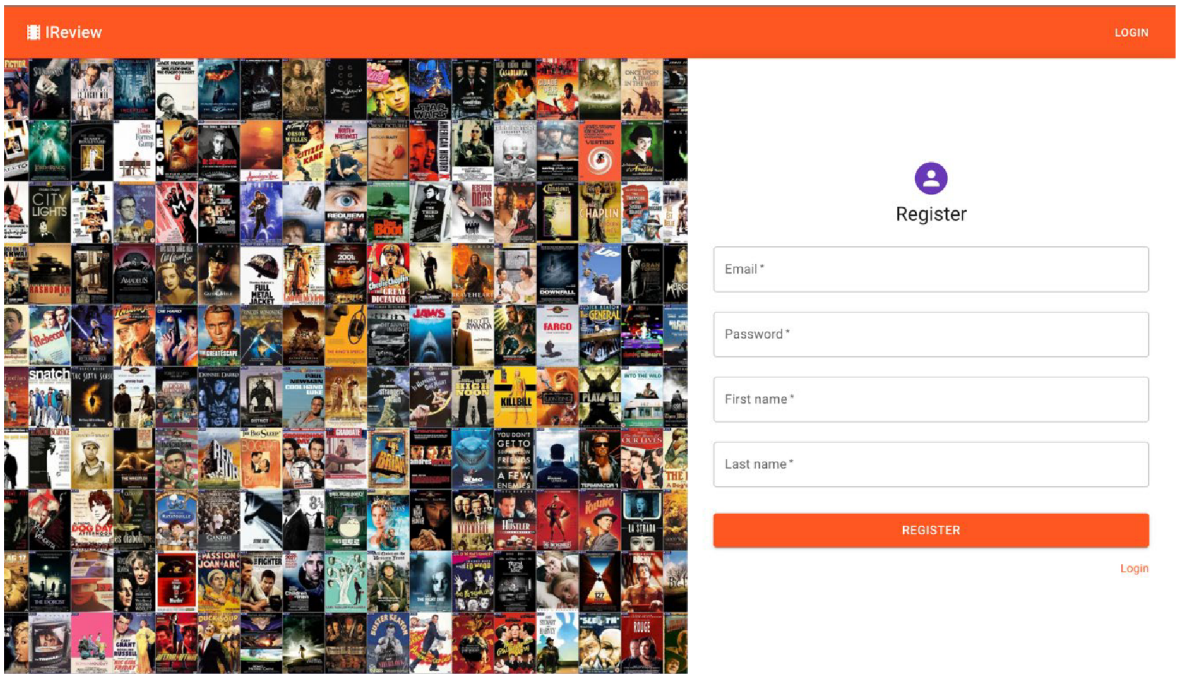
Ukázka kódu 15: metoda *refresh* servisu *UserService*

4.3.3.2 Frontend

Uživatelské rozhraní pro autentizaci je implementováno pomoci sady MaterialUI komponent. Pro dosažený responzivity je využít systém mřížek MaterialUI. Lokální stav poliček formuláře se řídí pomocí knihovny Formik.

```
<Grid container sx={{ height: '100%' }}>
    <Grid item xs={false} sm={4} md={7} sx={{ position: 'relative' }}>
        <Image
            priority
            src="/films-collage.jpg"
            alt="IReview"
            layout="fill"
            objectFit="cover"
        />
    </Grid>
    <Grid
        item square
        component={Paper}
        elevation={10}
        display="flex"
        alignItems="center"
        justifyContent="center"
        xs={12} sm={8} md={5}
    >
        <Box sx={styles.formBox as SxProps}>
            <FormHeader
                isRegisterUrl={isRegisterUrl}
                isLoginUrl={isLoginUrl}
                isForgetPasswordUrl={isForgetPasswordUrl}
                isResetPasswordUrl={isResetPasswordUrl}
                isPasswordReset={isPasswordReset}
            />
            {renderFormContent()}
        </Box>
    </Grid>
</Grid>
```

Ukázka kódu 16: komponenta formulář *<Form />*



Obrázek 8: vzhled stránky pro registraci

Stisknutím tlačítka *„LOGIN“* se obsah formuláře změní pro přihlášení uživatelé. Pokud uživatel stiskne tlačítko pro registraci, tak Redux zahájí akci „REGISTER“ prostřednictvím funkce *„dispatch“*. Tato akce je představena objektem, jejž vrátí funkce *„register“*, která přijímá vstupy zadané uživatelem. Následně tuto akci zpracuje Redux Saga *middleware* představený jako funkce generator.

```
function* handleRegisterLogin({
    payload,
}: PayloadAction<RegisterInputs | LoginInputs>): Generator<
    StrictEffect,
    void,
    IUser
> {
    try {
        yield put(setIsAuthLoading(true))

        if ('firstName' in payload) {
            yield call(UserApi.register, payload)

            yield put(setIsRegistered(true))
        } else {
            const userData = yield call(UserApi.login, payload)

            yield put(setUser(userData))
            yield put(setIsLoggedOut(false))

            yield fork(logoutHandler)
        }
    } catch (e) {
        if (axios.isAxiosError(e)) {
            if (e.response!.data.errors) {
                yield put(
                    setValidationErrors(mapToError(e.response!.data.errors))
                )
            } else {
                yield put(setLoginError(e.response!.data.message))
            }
        }
    } finally {
        yield put(setIsAuthLoading(false))
    }
}
```

Ukázka kódu 17: funkce generator *handleRegisterLogin*

Zaprvé pomocí funkce *„put“*, která přijímá Redux akci *„setIsAuthLoading“*, změní se stav pro znázornění průběhu načítání. Poté se přes funkci *„call“* spustí volání API, a následně získána data se uloží do Redux stavu. V případě přihlášení se dál vyvolá funkce *„fork“*, která spustí asynchronně úkol poslouchající, kdy uživatel spustí akci pro odhlášení z aplikace. Jestli během volání API adresy došlo u nějakých políček k validační chybě, tak vrátí API pole s objekty popisující tyto chyby. Pomocí funkce utility *„mapToError“* se získané pole převede do formy čitelné pro MaterialUI komponenty a uloží se do stavu. Toto způsobí vykreslení chybových hlášek u políček, jež neprošli validaci.

Volání API je realizováno pomocí Axios.

```
async register(inputs: RegisterInputs) {
    await $api.post('users/register', inputs)
},
async login(inputs: LoginInputs) {
    const { data } = await $api.post<IUser>('users/login', inputs)

    return data
},
async logout() {
    await $api.post('users/logout')
},
async refresh() {
    const { data } = await $api.get<IUser>('users/refresh')

    return data
},
```

Ukázka kódu 18: axios metody servisu *UserApi* pro volání API

4.3.4 Autorizace

Autorizace je proces zjištění přístupových oprávnění uživatele. Používá se také tento pojem pro vyjádření povolení k nějakému úkonu nebo operaci.
V dané aplikaci uživatel získává přístup k nějakým části aplikaci či její možnostem na zásadě přiřazených rolí. Všechny role jsou definovány v soulady s analýzou požadavků na role.

4.3.4.1 Backend

Na *backendu* přístup k určitým API adresám je kontrolován díku funkce *middleware „authMiddleware“*. Tato funkce přijímá jako argumenty pole s rolemi. V případě, že uživatel má přirazenou aspoň jednu roli z tohoto pole, tak bude mu povolen přístup. Role určitého uživatele se získávají z jeho přístupového tokenu. Pro srovnání rolí uživatelé s rolemi z pole argumentu se používá metoda *„compareUserRoles“* u servisu *„RoleService“*.

```
export const authMiddleware =
    (roles: RolesEnum[], isCommentsAllowed?: boolean) =>
    async (req: Request, res: Response, next: NextFunction) => {
        try {
            const { accessToken } = req.cookies as { accessToken: string }

            if (!accessToken) {
                return next(ApiError.unauthorized())
            }

            const userData = TokenService.validateAccessToken(
                accessToken
            ) as UserDto

            if (!userData) {
                return next(ApiError.unauthorized())
            }

            if (roles) {
                if (isCommentsAllowed) {
                    const user = await User.findOne({
                        where: { uuId: userData.uuId },
                    })

                    if (!user?.isCommentsAllowed) {
                        return next(ApiError.forbidden())
                    }
                } else {
                    if (
                        !(await RoleService.compareUserRoles(
                            userData.uuId,
                            roles
                        ))
                    ) {
                        return next(ApiError.forbidden())
                    }
                }
            }

            req.userUuId = userData.uuId

            next()
        } catch (e) {
            return next(ApiError.unauthorized())
        }
    }
```

Ukázka kódu 19: middleware *authMiddleware*

```
reviewRouter.post(
    '',
    authMiddleware([RolesEnum.WRITER]),
    upload.single('image'),
    ReviewController.create
)
```

Ukázka kódu 20: použití authMiddleware u směrovače *reviewRouter*

4.3.4.2 Frontend

Na *frontendu* autorizace je docílena pomocí komponenty vyššího řádu *„withRoles“*. Používá se u stránek, které mají omezený přístup. K takovým stránkám patří správa recenzí, uživatelů a účtu. Tato komponenta zahrnuje v sobě logiku obdobnou *middleware „authMiddleware“* u *backendu*. Role uživatele jsou brány ze stavu Redux.

```
const withRoles =
    (Component: ComponentType, roles: RolesEnum[] = [RolesEnum.USER]) =>
        () => {
            const user = useAppSelector(authSelectors.user)
            const isRefreshLoading = useAppSelector(authSelectors.isRefreshLoading)

            if (isRefreshLoading) {
                return <></>
            }

            if (!roles.some((role) => user?.roles.includes(role))) {
                return <Custom404 />
            }
            return <Component />
        }

export default withRoles
```

Ukázka kódu 21: komponenta vyššího řádu *withRoles*

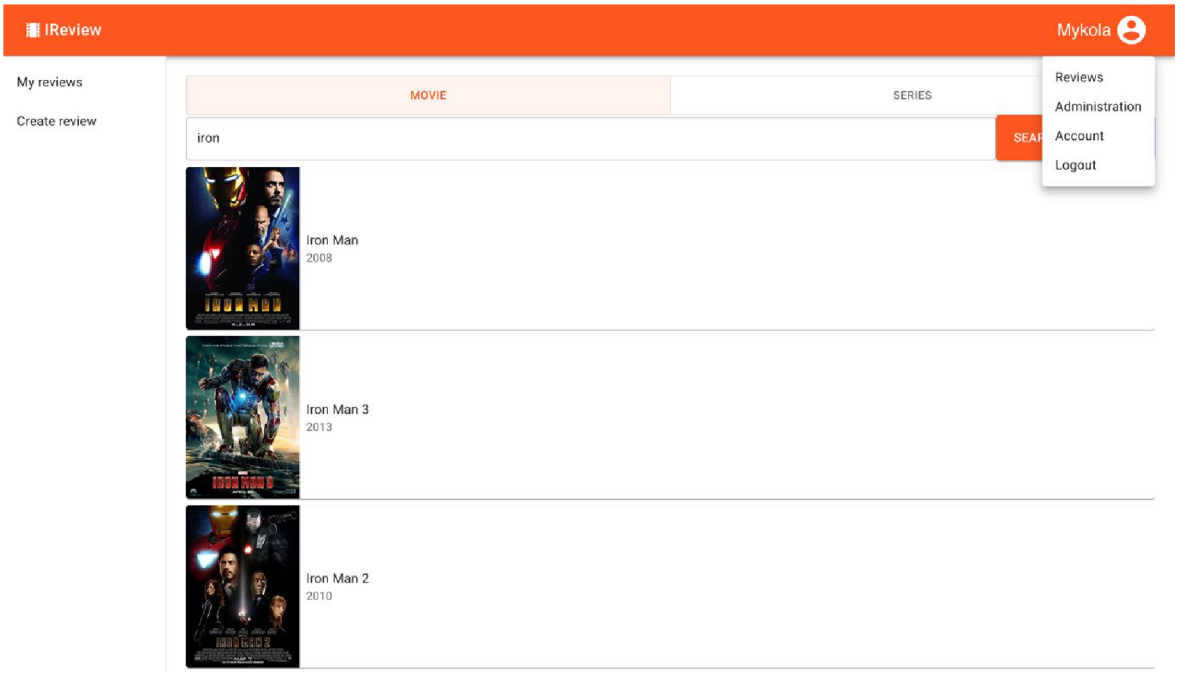
V případě, že uživatel nemá přístup ke stránce tato komponenta vykreslí chybovou stránky *„<Custom404 />“*. Jinak bude vykreslena původní cílová stránka.

```
const Reviews = () => {
    return (
        <>
            <Head>
                <title>Reviews</title>
                <meta name="robots" content="noindex, nofollow" />
            </Head>
            <Drawer />
        </>
    )
}

export default withRoles(Reviews, [RolesEnum.WRITER, RolesEnum.EDITOR])
```

Ukázka kódu 22: použití *withRoles* u komponenty *<Reviews />*

4.3.5 Správa recenzí

Vytvoření recenzí je možné pro uživatelé, které mají roli autora. Po přihlášení do aplikace se jim vykreslí ikona uživatelé s jeho jménem v pravé horní části stránky. Stisknutím této ikony si budou moci otevřít stránku *„Reviews“*.



Obrázek 9: vzhled stránky pro výběr filmů

Na této stránce je k dispozici autorům sekce *„Create reviews“*, kde lze vyhledat podle názvu film či seriál pro vytvoření recenze. Hledání filmů je implementováno pomocí OMDb API, které poskytuje zdarma velkou databázi s filmy a seriály.

```
async fetchFilmsByTitle(inputs: IOmdbInputs) {
    const { data } = await axios.get<{
        Search: IOmdbFilm[]
        totalResults: string
    }>(process.env.NEXT_PUBLIC_OMDB_URL!, {
        params: {
            apikey: process.env.NEXT_PUBLIC_OMDB_API_KEY,
            s: inputs.title,
            page: inputs.page,
            type: inputs.type,
        },
    })
```

Ukázka kódu 23: metoda *fetchFilmsByTitle* servisu *OmdbApi*

Kliknutím určitého filmu se autoru zobrazí dialogové okno s podrobnějším popisem filmu. Po stisknutí tlačítka *„CREATE REVIEW“* se otevře stránka pro vytvoření recenze, kde autor si bude moct nahrát obrázek recenze a napsat text. Recenzi lze vytvořit a buď publikovat hned, nebo uložit ji nepublikovanou jako koncept. Všechny recenze autora jsou dostupné ve sekci *„My reviews“*, kde je možné tyto recenze dál upravovat.

Na *backendu* se o správu recenzí stará servis *„ReviewService“*. Při vytvoření recenze se provádí kontrola, jestli se k danému filmu už vytvářely recenze. V případě, že film předtím žádné recenze neměl, tak uloží se do databáze. Jinak recenze bude asociovaná s filmem už existujícím v databázi.

```
async create(
    review: IReviewCreateInputs,
    film: IFilm,
    imageFileName: string,
    userUuId: string
) {
    const filmCandidate = await Film.findOne({
        attributes: ['imdbId'],
        where: { imdbId: film.imdbId },
    })

    if (!filmCandidate) {
        await Film.create(film)
    }

    await Review.create({
        ...review,
        image: imageFileName,
        filmImdbId: film.imdbId,
        userUuId,
    })
},
```

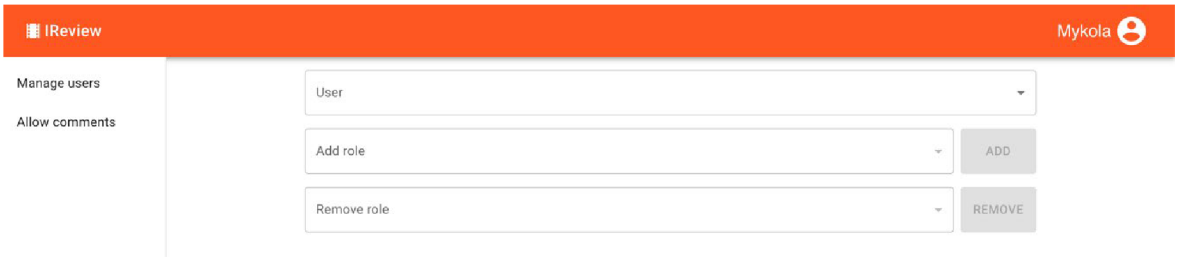
Ukázka kódu 24: metoda *create* servisu *ReviewService*

Pro uživatelé mající roli redaktora bude dostupná na stránce *„Reviews“* sekce *„Unpablished reviews“*. V této sekci se zobrazují recenze, které byly odstraněny z publikace redaktory. Takové recenze mají v databázi atribut *„isUnpublishedByEditor“* s hodnotou *true*. Po výběru recenze redaktoru se otevře stránka pro úpravu zvolené recenze.

4.3.6 Správa uživatelů

Velmi důležitou části dané aplikace je správa uživatelů. Uživatelé s rolí administrátora mají možnost přiřadit či odebrat role ostatním uživatelům, zatímco moderátoři můžou zakázat či zase povolit uživatelům psát komentáře.

Na *frontendu* je to realizováno pomocí stránky *„Administration“*, kde pro tyto role jsou příslušné sekce *„Manage users“* a *„Allow comments“*.



Obrázek 10: vzhled stránky pro správu uživatelů

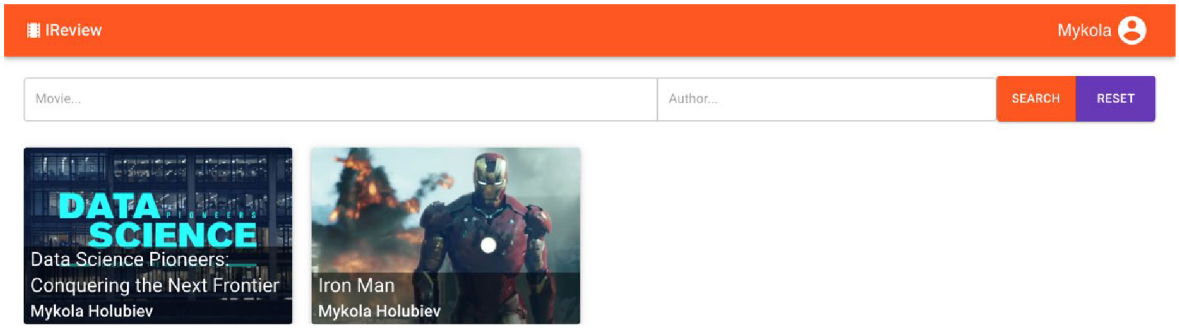
Na *backendu* se o tuto logiku starají servisy *„UserServis“* a *„RoleServis“*.

```
async addUserRole(data: Omit<IUserRole, 'createdAt' | 'updatedAt'>) {
    await UserRole.create({ ...data })
},
async findUserRoles(userUuId: string) {
    const user = await User.findOne({
        include: [
            {
                model: Role,
                as: 'roles',
                attributes: ['name'],
                through: {
                    attributes: [],
                },
            },
        ],
        attributes: ['uuId'],
        where: { uuId: userUuId },
    })

    return user!.roles.map(({ name }) => name)
},
```

Ukázka kódu 25: metody servisu *RoleService* pro přidání a získání rolí uživatele

4.3.7 Úvodní stránka s recenzemi

Na úvodní stránce této aplikace jsou zobrazeny všechny publikované recenze. Je také políčko pro jejich filtraci podle názvu filmu a jména autora.



Obrázek 11: vzhled úvodní stránky

Jako metoda vykreslení pro recenze na úvodní stránce byla zvolena inkrementální statická regenerace. Je to hlavně z důvodu, že nové recenze se můžou objevovat dost často, a uživatel bude chtít je přečíst co nejdříve.

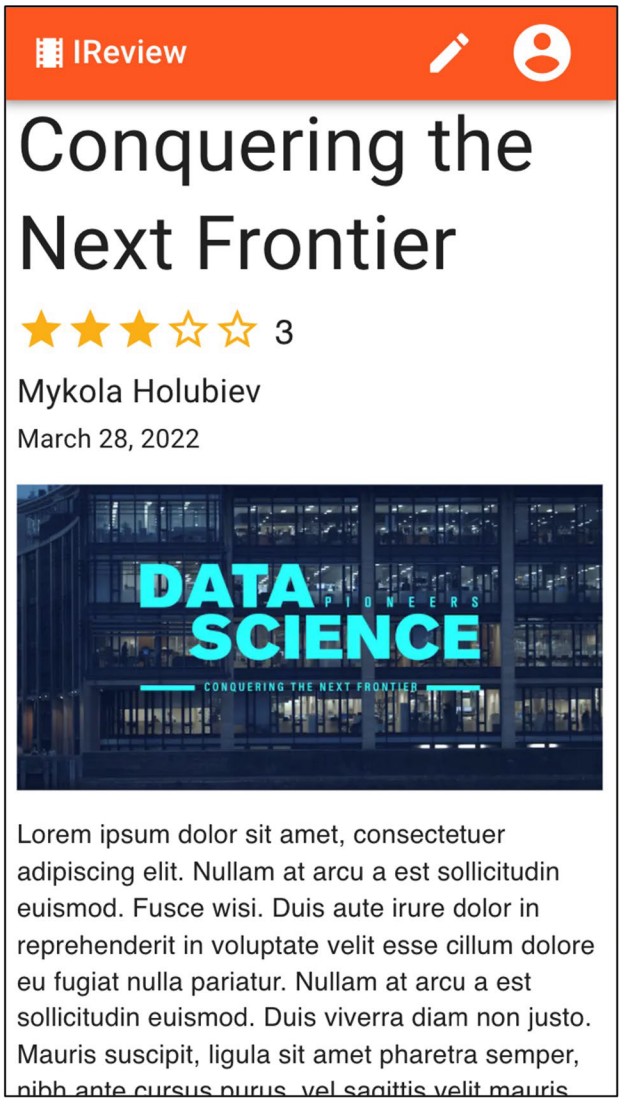
```
export const getStaticProps = wrapper.getStaticProps(
    ({ dispatch }) =>
        async () => {
            const data = await ReviewApi.fetch()

            dispatch(setReviews(data))

            return { props: {}, revalidate: 15 }
        }
)
```

Ukázka kódu 26: použití funkce *getStaticProps*

Před vykreslením funkce *„getStaticProps“* zavolá API a uloží stažená data do Redux stavu. Poté následuje generace stránek s těmito daty. Jakmile uživatel navštíví stránku pošle mu server vygenerovaný HTML soubor. Regenerace stránky pro rychlý přehled se provádí po 15 sekundách. Je to umožněno díky vlastnosti *„revalidate“*.

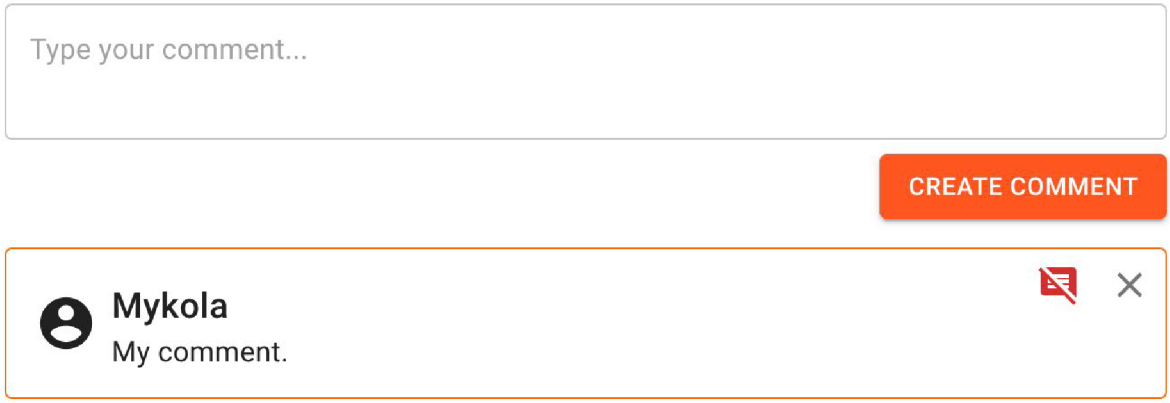
4.3.8 Stránka s recenzí

Z této stránky uživatel si bude moct číst text recenze, hodnotit ji a psát k ní své komentáře. Pokud tuto stránku otevřel autor dané recenze, tak si má možnost stisknutím tlačítka „Edit“ vedle ikony uživatelé otevřít stránku pro editaci recenze.



Obrázek 12: mobilní verze vzhledu stránky s recenzí

Stisknutím hvězdičky uživatel má možnost ohodnotit recenzi. Pod textem recenze se nachází poličko pro pasní komentářů.



Obrázek 13: vzhled komentářů

Stisknutím křížku se dá smazat komentář. Pro moderátory je tady také umožněno hned zakázat či povolit psát uživateli komentáře po stisknutí ikony s komentářem vedle křížku. Pro zákaz ikona je červená a přeškrtnutá, zatímco pro povolení je šedá.

Pro vykreslení této stránky byla zvolena metoda vykreslení na straně serveru. Je to z důvodu, že tato stránka má dynamický obsah jako hodnocení a komentáře. Aktualizace takového obsahu by se měla provádět na stránce ihned.

```
export const getServerSideProps = wrapper.getServerSideProps(
    ({ dispatch }) =>
        async ({ params, req }) => {
            try {
                const refreshToken = req.cookies.refreshToken

                const data = await ReviewApi.fetchOne(
                    +params!.id!,
                    refreshToken
                )

                dispatch(setCurrentReview(data))

                return { props: { isUser: !!refreshToken } }
            } catch (e) {
                return { notFound: true }
            }
        }
)
```

Ukázka kódu 27: použití metody *getServerSideProps*

Funkce *„getServerSideProps“* zavolá metodu *„fetchOne“* u servisu *„ReviewApi“* pro získání dat. Tato metoda přijímá dva argumenty: identifikátor primární klíč recenze a cookie s obnovovacím tokenem. Identifikátor recenze je brán z URL adresy. Obnovovací token slouží tady k určení uživatele na backendu, aby mu odpovídal vracený dynamický

obsah. Poté v případě úspěšného získaní dat, uloží je Redux do svého stavu. Jinak bude vykreslená chybová stránka oznamující, že taková recenze neexistují. Nakonec funkce pošle stránce vlastnost *„isUser“* a provede generaci stránky. Podle této vlastnosti se komponenta *„<Review />“* dozví, jestli existuje dynamický obsah pro daného uživatelé.

4.3.9 Progresivní webová aplikace

Aby tato webová aplikace fungovala jako progresivní, byla použita knihovna *next-pwa*. Tato knihovna umožňuje splnit všechny požadavky dané aplikace pro PWA s minimálním nastavením a konfigurací. Pro cachování dat používá automaticky strategii *runtimeCaching*, která umožňuje uložení obsahu do mezipaměti během používaní aplikace. V případě fungování aplikace v *offline* režimu bude se vždycky zobrazovat poslední aktuální obsah z mezipaměti. Aby se obsah stránky byl opravdu uložen, je nutné navštívit tuto stránku alespoň jednou.

Pro definování dané aplikace jako PWA byl vytvořen soubor manifest.

```
{
    "theme_color": "#ff5722",
    "background_color": "#ff5722",
    "display": "standalone",
    "scope": "/",
    "start_url": "/",
    "name": "IReview",
    "short_name": "IReview",
    "description": "Film reviews Next.js App",
    "icons": [
        {
            "src": "/icon-192x192.png",
            "sizes": "192x192",
            "type": "image/png"
        },
        {
            "src": "/icon-256x256.png",
            "sizes": "256x256",
            "type": "image/png"
        },
        {
            "src": "/icon-384x384.png",
            "sizes": "384x384",
            "type": "image/png"
        },
        {
            "src": "/icon-512x512.png",
            "sizes": "512x512",
            "type": "image/png"
        }
    ]
}
```

Ukázka kódu 28: použití manifestu

V manifestu jsou definovány název této aplikace, její popis, barva tématu a ikony pro různé velikosti obrazovky. Samotný manifest soubor a ikony jsou uložené ve složce *„public“*.

5 Výsledky a diskuse

Po dokončení implementace daná aplikace byla sestavena a následně spouštěna v produkčním režimu. Všechny případy užití aplikace byly otestovány. Pro testování chování aplikace na různých zařízení byly využity XCode a Android Studio.

Velmi důležitou částí při vytvoření této aplikace byla analýza. Během ní bylo zjištěno, jaké technologie budou nejvhodnějšími pro implementaci. Pro frontend byl zvolen Next.js, protože má všechny nezbytné metody vykreslení pro realizaci požadavků dané aplikace. Backendová část byla implementována pomocí frameworku Express.js a softwarového systému Node.js. Jako jediný programovací jazyk pro celou aplikaci byl použit TypeScript, což výrazně usnadnilo vývoj.

Analýza umožnila také vytvoření detailnějšího návrhu, který byl k dispozici v průběhu vytvoření aplikace. Tento návrh dovolil vyvíjet celou aplikaci po krocích a neztratit se mezi její jednotlivými části.

Vytvořena aplikace vyhovuje úplně všem požadavkům na ni. Dalším krokem by bylo její nasazení a poskytnutí uživatelům pro sbíraní zpětné vazby. Poté by se dálo aplikaci upravit a následně vyvíjet pro komerční účel.

6 Závěr

Teoretická část této bakalářské byla věnována popisu důležitých základů a technologií pro fungování webových aplikací. Čtenář byl seznámen se specifikami jazyků JavaScript a TypeScript, jejich rozdíly a případy použití. Dál následovaly podkapitoly věnované backendové a frontendové části aplikace. Bylo uvedeno k čemu každá z částí slouží, a jak mezi sebou komunikuji. Nakonec je zmíněno, co je progresivní webová aplikace, a jaké by měla splňovat charakteristiky.

Praktická část dané práce se zabývala postupem vytvoření webové aplikace. Nejdřív byla provedena analýza požadavku a funkcionalit. Pak následoval návrh, který byl zpracován hlavně pomocí UML specifikace. Největší a poslední části vlastní práce je podkapitola s implementací aplikace. Zde jsou popsány nejdůležitější postupy vývoje dané aplikace s její zdrojovými kódy.

7 Seznam použitých zdrojů

1. Application Programming Interface (API). IBM Cloud Education [online]. 2020 [cit. 2022-03-06]. Dostupné z: https://www.ibm.com/cloud/learn/api
2. CHAMIKARA, Nuwan. Let's Learn About Client-Side Web Development. Medium [online]. 2019 [cit. 2022-03-07]. Dostupné z: https://medium.com/@gmnchamikara/lets-learn-about-client-side-web-development-38ebdda73d42
3. CHOI, David. *Full-Stack React, TypeScript, and Node*. Birmingham: Packt, 2020. ISBN 978-1-83921-993-1.
4. COPES, Flavio. What is a JavaScript Frontend Framework?: A little introduction to frontend frameworks. Flavio Copes [online]. 2020 [cit. 2022-03-15]. Dostupné z: https://flaviocopes.com/what-is-a-frontend-framework/
5. CORREIA, Rita. Understanding Rendering in Next.js. DEV Community [online]. 2021 [cit. 2022-03-18]. Dostupné z: https://dev.to/ritaxcorreia/understanding-rendering-in-next-js-1m1b
6. DOGILO, Fernando. REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development. 2nd ed. New York City: Apress, 2018. ISBN 978-1-4842-3714-4.
7. DURGESH, Kumar. Next.js Introduction. GeeksforGeeks [online]. 2021 [cit. 2022-03-18]. Dostupné z: https://www.geeksforgeeks.org/next-js-introduction/
8. Front End vs Back End. *Education Wiki* [online]. [cit. 2022-03-03]. Dostupné z: https://cs.education-wiki.com/5407557-front-end-vs-back-end
9. GetStaticProps. Next.js [online]. [cit. 2022-03-20]. Dostupné z: https://nextjs.org/docs/basic-features/data-fetching/get-static-props
10. GUPTA, Lokesh. Transpiler vs Compiler. *HowToDoInJava* [online]. 2021 [cit. 2022-03-02]. Dostupné z: https://howtodoinjava.com/typescript/transpiler-vs-compiler/
11. HAVERBEKE, Marijn. *Eloquent JavaScript*. 3rd ed. San Francisco: No Starch Press, 2018. ISBN 1593279507.
12. HTTP request methods. *MDN Web Docs* [online]. 2021 [cit. 2022-03-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

13. HTTP response status codes. *MDN Web Docs* [online]. 2022 [cit. 2022-03-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
14. Incremental Static Regeneration. Next.js [online]. [cit. 2022-03-20]. Dostupné z: https://vercel.com/docs/concepts/next.js/incremental-static-regeneration#
15. Introduction to client-side frameworks. MDN Web Docs [online]. 2022 [cit. 2022-03-15]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction#a_brief_history
16. JAVOREK, Honza. Základní pojmy. *Co je API?* [online]. 2020 [cit. 2022-03-03]. Dostupné z: https://cojeapi.cz/03-zakladni-pojmy.html
17. KOĎOUSKOVÁ, Barbora. CO JSOU PROGRESIVNÍ WEBOVÉ APLIKACE (PWA) A JAKÉ MAJÍ VÝHODY. Rascasone [online]. 2021 [cit. 2022-03-20]. Dostupné z: https://www.rascasone.com/cs/blog/progresivni-webova-aplikace-vyhody
18. MOHAN, Mehul a Narayan PRUSTY. *Learn ECMAScript*. 2nd ed. Birmingham: Packt, 2018. ISBN 978-1-78862-006-2.
19. NYAKUNDI, Hillary. What is a PWA? Progressive Web Apps for Beginners. FreeCodeCamp [online]. 2021 [cit. 2022-03-20]. Dostupné z: https://www.freecodecamp.org/news/what-are-progressive-web-apps/
20. ONYANGO, Gerald a Dennis PADIERNOS. International Service Worker Caching Awareness Day. Netlify [online]. 2018 [cit. 2022-03-21]. Dostupné z: https://www.netlify.com/blog/2018/09/21/international-service-worker-caching-awareness-day/
21. OSSERA, Witold. Next.js for CTOs. Why do we love Next.js & what is it used for? The Software House [online]. 2021 [cit. 2022-03-18]. Dostupné z: https://tsh.io/blog/what-is-next-js-used-for/
22. PANG, Avelon. TypeScript vs. JavaScript. *Medium* [online]. 2021 [cit. 2022-03-01]. Dostupné z: https://medium.com/geekculture/typescript-vs-javascript-e5af7ab5a331
23. REST APIs. IBM Cloud Education [online]. 2021 [cit. 2022-03-06]. Dostupné z: https://www.ibm.com/cloud/learn/rest-apis

24. SANGTIANI, Kunal. Different forms of Pre-rendering in NextJS. GeeksforGeeks [online]. 2022 [cit. 2022-03-20]. Dostupné z: https://www.geeksforgeeks.org/different-forms-of-pre-rendering-in-nextjs/
25. STRELEC, Michal. Jak funguje webová aplikace? Michal Strelec [online]. [cit. 2022-03-07]. Dostupné z: https://www.strelec.pro/napsal/jak-funguje-webova-aplikace
26. TIELENS THOMAS, Mark. React in Action. Shelter Island: Manning Publications, 2018. ISBN 978-1617293856.
27. Úvod do HTTP a HTTPS. *Microsoft Docs* [online]. 2022 [cit. 2022-03-03]. Dostupné z: https://docs.microsoft.com/cs-cz/azure/rtos/netx-duo/netx-duo-web-http/chapter1
28. Web app manifests. MDN Web Docs [online]. 2022 [cit. 2022-03-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Manifest

8 Přílohy

usecase-diagram.jpg:

Diagram případu užití ve formátu JPEG

class-diagram.png:

Diagram tříd ve formátu PNG

erd-diagram.png:

Entity Relationship diagram ve formátu PNG

film-reviews-source.zip:

Archiv se zdrojovým kódem celé aplikace. Také dostupný z:

https://github.com/glbvnik/film_reviews