VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYUŽITÍ SLACK API PRO HEADLESSCMS

USE OF SLACK API FOR HEADLESSCMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JOZEF HRUŠKA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VLADIMÍR BARTÍK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Hruška Jozef**
Program: Informační technologie
Název: **Využití Slack API pro HeadlessCMS**
Use of Slack API for HeadlessCMS
Kategorie: Web

Zadání:

1. Prostudujte a seznamte se s principem fungování existujících nástrojů HeadlessCMS a dokumentací k Slack API.
2. Na základě získaných znalostí navrhněte vlastní řešení, které by využívalo Slack k propojení, transformaci a zobrazení obsahu v HeadlessCMS místo použití konvenční DB, která je využívána současnými řešeními. Návrh konzultujte s vedoucím.
3. Navržené řešení implementujte ve zvoleném prostředí.
4. Proveďte testování vytvořené aplikace a jejího propojení s aplikací Slack.
5. Práci zhodnoťte, srovnejte se stávajícími HeadlessCMS nástroji a navrhněte další možná budoucí vylepšení.

Literatura:

- SIMPSON, K.: *You don't know JS: ES6 and beyond*. Beijing: O'Reilly, 2016. ISBN 978-1-491-90424-4.
- STURGEON, P.: *Build APIs you won't hate*. Philip J. Sturgeon, 2015. ISBN 978-0-692-23269-9.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1-2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Bartík Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 16. října 2019

Abstrakt

Práca si dáva za cieľ vytvoriť redakčný systém s otvoreným aplikačným rozhraním (Headless CMS) a možnosťou správy obsahu v prostredí aplikácie Slack. Inštalácia a následné používanie systému nevyžaduje žiadnu konfiguráciu zo strany uživateľa. Otvorené (verejné) ako aj uzatvorené (skryté) aplikačné rozhranie je vybudované podľa špecifikácie GraphQL. Otvorené rozhranie slúži výhradne k čítaniu dát, tzn. že nie je možné dáta akokoľvek vkladať či modifikovať použitím tohto rozhrania. Výstupom práce je plne funkčný prototyp systému, ktorého súčasti boli implementované pomocou nástrojov React a Node.js s dôrazom na prívetivosť uživateľského rozhrania.

Abstract

The aim of this thesis is to create a content management system with open application interface (Headless CMS) and capability of content management inside Slack application. Installation and use of system does not require any configuration by user. Open (public) and closed (hidden) application interface is built following GraphQL specification. The opne interface is read-only which means it is not possible to insert or modify data through this interface. The output of this work is a fully functional prototype implemented with tools as React and Node.js with focus on user-friendly interface.

Kľúčové slová

RS, Slack, React, Prisma, Apollo, Node.js, Next.js, Redux, Bolt, GraphQL, TypeScript, PostgreSQL

Keywords

CMS, Slack, React, Prisma, Apollo, Node.js, Next.js, Redux, Bolt, GraphQL, TypeScript, PostgreSQL

Citácia

HRUŠKA, Jozef. *Využití Slack API pro HeadlessCMS*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vladimír Bartík, Ph.D.

Využití Slack API pro HeadlessCMS

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Vladimíra Bartíka, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Jozef Hruška
22. mája 2020

Poďakovanie

Rád by som poďakoval Ing. Vladimírovi Bartíkovi, Ph.D. za odborné vedenia a čas venovaný mojej bakalárskej práci. Ďaľej by som rád poďakoval Jánovi Vorčákovi, M.Sc., za inšpiráciu vedúcu k téme tejto práce, za jeho cenné rady a konzultácie.

Obsah

Kapitola 1

Úvod

Tradičné redakčné systémy pre správu obsahu sú obvykle zostavené z dvoch hlavných súčastí – *administračného* a *verejného webového rozhrania*. Administračné rozhranie slúži pre tvorbu a úpravu obsahu, webové na jeho nasledné zobrazenie. Webové rozhranie je typicky jednotné pre všetky typy zariadení na ktorých je zobrazované, tzn. že jeho používanie často nie je optimálne (napr. na mobilných telefónoch s dotykovou obrazovkou).

Pre tento dôvod sa začali využívať systémy bez webového rozhrania, disponujúce klasickým administračným a *verejným aplikačným rozhraním* (API[1]), ktoré umožňuje obsah získať a následne ho zobraziť optimálne na ľubovoľnej platforme. Takéto redakčné systémy sa nazývajú *Headless CMS*. Súčasné Headless CMS sú často robustné systémy, ktoré však vyžadujú komplexnú konfiguráciu predtým, ako ich je možné začať využívať. Niektoré takéto systémy potrebujú aj vlastnú infraštruktúru pre ich nasadenie. Tieto skutočnosti otvárajú priestor pre taký Headless CMS, ktorý by pomohol vyriešiť tieto prekážky. Takýto redakčný systém je cieľom tejto práce.

Navrhovaný systém poskytuje možnosť správy obsahu bez nutnosti úvodnej konfigurácie a infraštruktúry. Obsah je možné spravovať pomocou užívateľského rozhrania v službe Slack alebo doplnkového webového rozhrania.

Obsah kapitol

Prvá časť práce je venovaná teoretickým znalostiam nutným pre riešenie tejto práce. Kapitola 2 popisuje vývoj klientských aplikácií, kapitola 3 vývoj serverových aplikácií a kapitola 4 koncepcií headless redakčného systému. V druhej časti práce kapitola 5 popisuje návrh riešenia, kapitola 6 implementáciu jednotlivých častí návrhu a kapitola 7 priebežné a finálne testovanie výsledného riešenia. Posledná kapitola 8 obsahuje záverečné zhrnutie a hodnotenie autora práce.

[1] API – Application programming interface

Kapitola 2

Vývoj klientských aplikácii

Zobrazenie obsahu užívateľom. Kapitola popisuje teoretické znalosti nutné pre návrh a implementáciu klientských aplikácií (v prípade tejto práce webovej stránky a Slack aplikácie). Webové technológie sa vyvíjajú vysokou rýchlosťou a s nimi aj nároky užívateľov na rýchlosť, použiteľnosť, ale aj vzhľad takýchto aplikácií.

Sekcia 2.1 sa venuje základným pravidlám user experience[1], sekcie 2.2 až 2.5 približujú programovacie a značkovacie jazyky, ktoré sú použité v tejto práci. Posledné sekcie 2.6 a 2.7 popisujú dve hlavné knižnice využívané pre vývoj moderných webových aplikácií React a Next.js.

2.1 User experience (UX)

Prí návrhu uživateľského rozhrania je jedným z najdôležiteľších parametrov *uživateľská skúsenosť*. Dizajnéri sa snažia zaistiť aby ich návrh bol intuitívny, jednoduchý, ale zároveň aj plne použiteľný a originálny. Pri UX analýze sa dizajnér snaží vnímať svoj návrh zo strany koncového užívateľa. Vo svete však neexistuje jednotná definícia úkonov, ktoré vedú k dokonalému uživateľskému zážitku.

2.1.1 Porozumenie potrebám užívateľa

Dizajnér sa pri návrhu pozerá na produkt ako jeho koncový užívateľ. Analyzuje potreby, pre ktoré sa uživateľ rozhodol produkt využívať, ale aj problémy, ktoré užívateľovi bránia v jednoduchom a intuitívnom používaní daného produktu. Po porozumení potrieb užívateľa príchádza z riešeniami, ktoré by však nemali vytvoriť ďalšie problémy a prekážky v používaní produktu.

2.1.2 Prístupnosť (Accessibility)

Veľmi jednoducho dosiahnuteľná, ale často zanedbávaná vlastnosť je dobrá prístupnosť (použiteľnosť) pre ľudí so zdravotnými znevýhodneniami. Dizajnér musí zaistiť, aby farebné pozadia jednotlivých prvkov mali dostatočný kontrast od ich obsahu,alebo zvýrazniť prvok v prípade, že je užívateľom (túto vlastnosť majú v určitej podobe vstavané aj niektoré webové prehliadače, avšak nie vždy optimálne).

[1] user experience – uživateľská skúsenosť

2.2 HTML

HTML (Hypertext Markup Language) je *značkovací jazyk*, pomocou ktorého je možné popísať štruktúru webových stránok. Skladá sa zo stromu elementov [2.2.1], ktoré majú svoj obsah, parametre a sú ohraničené pomocou značiek (tags).

> HTML je nejzákladnejším stavebným kameňom webových stránok. „Hypertext“, v názve odkazuje k možnosti vytvorenia odkazov, ktorými je možné prepojiť webové stránky. (MDN [8])

Pre zobrazovanie HTML slúžia webové prehliadače. Každý webový prehliadač postupuje pri vykresľovaní niektorých častí HTML inak ako ostatné, preto je nutné skontrolovať či je webová stránka správne zobrazená na viacerých webových prehliadačoch.

2.2.1 Elementy

HTML element je oddelený od zbytku textu v dokumente pomocou značiek (tags), ktoré pozostávajú z názvu elementu ohraničeného znakmi „<“ a „>“. Názov elementu vo vnútri značky je *case insensitive*, tzn. že nezáleží či je písaný veľkými alebo malými písmenami. Napríklad značka `<title>` môže byť napísaná aj ako `<Title>` alebo `<TITLE>`. Všetky tieto zápisy sú správne. [8]

Značky sa klasifikujú na dve skupiny – *párové* a *nepárové*. Párové značky sú také, ktoré obsah elementu ohraničujú otvárajúcou (`<title>`) a uzatvárajúcou (`</title>`) značkou. Nepárové značky sú také, ktoré nemajú svoju uzatvárajúcu značku, napríklad obrázok (`<img />`).

Zoznam niektorých najpoužívanejších elementov:

- `head` – Obsahuje strojom čítateľné informácie (metadáta) o dokumente ako napríklad titulok, skripty alebo štýly. [8]
- `body` – Reprezentuje obsah HTML dokumentu, pričom sa v jednom dokumente môže nachádzať maximálne raz. [8]
- `title` – Definuje titulok dokumentu, ktorý je zobrazený vo webovom prehliadači. [8]
- `button` – Reprezentuje klikateľné tlačidlo, použiteľné napríklad pre potvrdzovanie formulárov alebo kdekoľvek inde v HTML dokumente ako štandardné tlačidlo. Tlačidlá sú v štýle jednotnom s platformou na ktorej sú zobrazované, ak nie sú priložené štýly, ktoré by ich upravovali. [8]

2.3 CSS

CSS (Cascading Style Sheets) je deklaratívny jazyk, ktorý dokáže kontrolovať ako sa webový stránky zobrazujú vo webových prehliadačoch. Prehliadače aplikujú CSS štýly priamo na elementy nimi upravené a potom ich zobrazia. Deklarácie štýlov obsahujú *vlastnosti* a ich *hodnoty*, ktoré určujú ako má webová stránka vyzerať. [8]

CSS je možné pridať do HTML dokumentu tromi spôsobmi:

- Importovaním externého CSS súboru v hlavičke dokumentu.

- Vložením medzi element `<style>` do hlavičky dokumentu.
- Vložením jednotlivých vlastností a ich hodnôt do značiek jednotlivých HTML elementov cez parameter `style`.

Jednotlivé vlastnosti sa elementom priraďujú použitím *CSS selektorov*. Existujú aj selektory alebo kombinátory, ktoré umožňujú zvoliť rodičovské alebo vedľajšie elementy. [8]

2.4 JavaScript

JavaScript je populárny *interpretovaný* programovací jazyk. Napriek tomu, že je známy predovšetkým ako skriptovací jazyk pre webové aplikácie, dnes je využívaný mnohými prostrediami mimo webových prehliadačov, ako napríklad Node.js [3.2] pre tvorbu sieťových aplikácií. [8]

Štandardom pre JavaScript je ECMAScript[2]. Od roku 2012 všetky moderné webové prehliadače podporujú ECMAScript verzie 5.1. Staršie prehliadače podporujú aspoň ECMAScript 3. V roku 2015 bola vydaná verzia ECMAScript 2015 (známa aj ako ECMAScript 6 alebo ES6). Odvtedy je štandard ECMAScript na cykle ročných vydaní. [8]

2.5 TypeScript

TypeScript je rozšírenie programovacieho jazyka JavaScript. Jedná sa o silne typovaný, objektovo orientovaný a kompilovaný programovací jazyk. [7]

TypeScript je obvykle vhodné skompilovať do natívneho JavaScriptu pre zachovanie kompatibility a lepšiu optimalizáciu. Využívanie TypeScriptu nie je nutné, avšak vďaka vlastnosti silného typovania umožní vývojárovi predísť chybám ešte pred kompiláciou.

> Vzťah medzi TypeScriptom a JavaScriptom je unikátny medzi modernými programovacími jazykmi. TypeScript existuje ako vrstva nad JavaScriptom; ponúka vlastnosti JavaScriptu a pridáva svoju vlastnú vrstvu navrch. Táto vrstva je nazývaná *typovací systém TypeScript*. (Dokumentácia TypeScript [7])

Využitie jednoduchého typu v TypeScripte by mohlo vyzerať následovne:

```
1  /* Type definition */
2  type Person = {
3    fname: string;
4    lname: string;
5  }
6
7  /* Type assignment */
8  const Osoba: Person = {
9    fname: "Jozef";
10   lname: "Hruska";
11 }
```

Výpis 2.1: Príklad zápisu v programovacom jazyku TypeScript.

[2] https://www.ecma-international.org/

Využitie programovacieho jazyka TypeScript nie je limitované pre vývoj klientských aplikácii. Rovnako ako pri JavaScripte sa jedná o univerzálny programovací jazyk, ktorý je vďaka nástrojom ako Node.js [??] možné využiť napríklad aj na tvorbu serverových aplikácií.

2.6 React

Populárna JavaScriptová knižnica pre *budovanie užívateľských rozhraní*. React je deklaratívny, efektívny a flexibilný. Dovoľuje vytvárať uživateľské rozhrania zložené z malých izolovaných častí kódu, nazývaných *komponenty* [2.6.2]. [2]

2.6.1 JSX

Syntaktické rozšírenie JavaScriptu inšpirované značkovacími jazykmi, avšak s možnosťou využívať plné schpnosti JavaScriptu. JSX vzniklo z dôvodu, že v moderných webových aplikáciach bolo čoraz častejšie nutné spájať vykreslovaciu logiku a logiku uživateľských rozhraní. [2]

```
const name = "Jozef";
const element = <h1>Hello, {name}!</h1>;

ReactDOM.render(
  element,
  document.getElementById("root")
);
```

Výpis 2.2: Príklad využitia JSX v React aplikácií. [2]

Atribúty JSX tagov môžu prijímať textové reťazce (`<div className='block'>`) alebo JavaScriptové výrazy (`<img src={item.image} />`), ktoré sa neskôr vyhodnotia.

Elementy JSX sú kompilované do volaní `React.createElement()`, ktoré vrátia obyčajné JavaScriptové objekty nazvané „React elementy“. [2]

```
const element = {
  type: 'h1',
  props: {
    className: 'greetings',
    children: 'Hello world!'
  }
};
```

Výpis 2.3: Príklad jednoduchého React elementu po kompilácií. [2]

Tieto objekty slúžia Reactu ako „popis“ pre zobrazenie. Využíva ich pre zostavenie a urdržovanie aktuálneho DOM[3]. [2]

2.6.2 Komponenty

Komponenty umožňujú vývojárovi rozdeliť uživateľské rozhranie do samostatných, izolovaných a *znovu použiteľných* častí. [2]

[3]DOM – document object model

Komponenty sa delia do dvoch skupín – *funkcionálne* a *triedne* komponenty. Najjednoduchší spôsob ako definovať komponent je obyčajná JavaScriptová funkcia:

```
const Title: React.FC = (props) => {
  return <h1>Welcome to {props.text}</h1>;
}
```

Výpis 2.4: Príklad definície funkcionálneho komponentu.

Avšak pre definíciu komponentu možeme použiť aj ES6 triedu:

```
class Title extends React.Component {
  render {
    return <h1>Welcome to {this.props.text}</h1>;
  };
}
```

Výpis 2.5: Príklad definície triedneho komponentu.

> Konceptuálne sú komponenty iba obyčajné JavaScriptové funkcie. Prijímajú vstupy nazývané *props* a vracajú React elementy popisujúce zobrazenie na obrazovke. (Dokumentácia React [2])

Vďaka veľkej komunite je React jedným z najpoužívanejších nástrojov pre tvorbu užívateľských rozhraní.

2.7 Next.js

Najväčším problémom moderných React a celkovo JavaScriptových aplikácií je, že *vykresľovanie obsahu prebieha na strane klienta*, a teda v jeho webovom prehliadači (client side render). Logika aplikácií, získavanie dát, smerovanie, všetko prebieha priamo vo webovom prehliadači. To spôsobuje nielen vysoké nároky na výpočetný výkon klientského zariadenia, ale aj problémy so SEO optimalizáciou (roboty, ktoré používajú webové vyhľadávacie nástroje ako napríklad Google majú problémy s indexovaním stránok, ktoré sú vykresľované na strane klienta).

Next.js ponúka riešenie na tieto problémy. Umožňuje webové aplikácie vykresliť vopred dvoma spôsobmi:

- `Statické vykresľenie pri kompilácií` – Stránky, sú zobrazované rovnako pri každej požiadavke a nevyžadujú teda akýkoľvek kontext môžu byť vykreslené už pri *kompilácií aplikácie*. HTML štruktúra je vygenerovaná vopred a zasielaná klientom, ktorí si danú stránku vyžiadali.
- `Vykresľovanie pri každej požiadavke` – Stránky, ktoré pre zobrazenie potrebujú konkrétny kontext pri požiadavke (napríklad profilová stránka užívateľa vyžaduje jeho ID pre získanie dát), sú vykreslené vopred len čiastočne a zbytok je prenechaný pre klienta, ktorý už potrebný kontext pozná.

V oboch prípadoch je ku HTML priložený minimálny JavaScript kód, ktorý sa vo webovom prehliadači inicializuje a zaistí tak interaktivitu webovej stránky. Tento proces sa nazýva *hydratácia*. [15]

Ďalšími benefitmi Next.js sú:

- `Nulová konfigurácia` – Žiadna vstupná konfigurácia nie je potrebná. Všetky funkcionality Next.js sú dostupné ihneď po inštalácií. [15]
- `Optimalizácia` – Automatické optimalizovanie balíčkov, ktoré sú vyžadované, čo umoňuje rýchlejšiu kompiláciu. [15]

Kapitola 3

Vývoj serverových aplikácii

`Server` – centrálny počítač z ktorého ostatné počítače získavajú informácie. [1]

Serverová aplikácia je proces spustený na centrálne dostupnom zariadení. Obvykle slúži ako zdroj informácií pre ostatné zariadenia, typicky v počítačovej sieti. Tento proces očakáva požiadavky a odpovedá na ne vopred určenou reakciou.

Serverové aplikácie je možné implementovať v rôznych programovacích jazykoch, no medzi najpoužívanejšie patrí PHP, Java, Python alebo JavaScript (popr. rozšírenie TypeScript). Aplikácia má obvykle určený komunikačný protokol, pomocou ktorého prijíma požiadavky a odosiela odpovede.

Táto kapitola pojednáva o základných princípoch vývoja serverových aplikácií. Sekcia 3.1 je venovaná komunikačnému protokolu HTTP, sekcia 3.2 JavaScriptovému prostrediu pre tvorbu sieťových aplikcií Node.js, sekcia 3.3 relačným databázam, sekcia 3.4 špecifikácií GraphQL. Posledná sekcia 3.5 je venovaná verejnej API služby Slack, ktorej znalosť je nutná pre vytvorenie tejto práce.

3.1 HTTP

Protokol HTTP je využívaný ako generický protokol pre prenos dát (správ) medzi klientom a serverom, napríklad HTML dokumentov. Komunikácia je zahájená klientom, typicky webovým prehliadačom.

> Hypertext Transfer Protocol (HTTP) je protokol na aplikačnej úrovni pre distribuované a spolupracujúce informačné systémy. Protokol HTTP je využívaný iniciatívou World Wide Web od roku 1990. (RFC 2616 [3])

3.1.1 Terminológia

- `Požiadavka (Request)` – HTTP správa odoslaná klientom, adresovaná pre server.
- `Odpoveď (Response)` – HTTP správa odoslaná serverom, adresovaná pre klienta. Odosiela sa po prijatí a spracovaní HTTP požiadavky od klienta.
- `Metóda (Method)` – Pole v hlavičke HTTP požiadavky. Definuje operáciu, ktorú má server vykonať po prijatí danej HTTP požiadavky.

3.1.2 Metódy

- `OPTIONS` – Reprezentuje požiadavku na popis komunikačných schopností príjemcu. Umožňuje klientovi vopred zistiť komunikačné možnosti servera bez nutnosti odosielania konkrétnej HTTP požiadavky na daný zdroj dát. [8]
- `GET` – Vyžiadanie si konkrétnych dát. Požiadavky s metódou GET by dáta mali výhradne získavať, nie ich odosielať. [8]
- `POST` – Odoslanie dát na server. Dáta sú priložené v tele správy. Typ odosielaných dát určuje hlavička *Content-Type*. [8]
- `PUT` – Vytvorenie nových alebo úprava existujúcich dát pomocou dát priložených v tele správy. [8]
- `DELETE` – Odstránenie dát. [8]

Typy metód, ktoré nesúvisia s touto prácou boli zámerne neuvedené.

```
fetch("http://localhost:5000/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.STRINGIFY({
    firstName: "Jozef",
    secondName: "Hruska",
  })
});
```

Výpis 3.1: Príklad odoslania HTTP POST metódy v prostredí TypeScript.

3.2 Node.js

Asynchrónny, udalosťami riadený Javascriptový runtime[1] vytvorený pre budovanie škálovateľných sieťových aplikácii. [10]

Node.js umožňuje spustiť JavaScriptový kód mimo prostredia webových prehliadačov a poskytuje rozšírené funkcionality ako prístup k súborovému systému alebo kryptografické metódy.

3.2.1 Node Package Manager - NPM

Najväčší softvérový register na svete. NPM umožňuje vývojárom open-source softvéru zdieľať JavaScriptové balíčky verejne alebo si vytvoriť súkromný repozitár a balíčky zdieľať iba v organizácií. [9]

NPM je rozdelené do troch hlavných častí:

- `Webová stránka` – Vyhľadávanie balíčkov, správu profilov a organizácií. [9]

[1] runtime – behové prostredie programu

- `Command Line Interface (CLI)` – Konzolové rozhranie pre interakciu s NPM. [9]
- `Register` – Verejná databáza JavaScriptového softvéru a metadát. [9]

Hlavnou časťou je pre vývojára práve `CLI`, ktoré mu umožňuje spravovať balíčky vo svojej aplikácií. Nainštalované balíčky ale ich závislosti sú inštalované do zložky v koreňovom adresári projektu `node_modules`. Nastavenia, informácie o projekte a zoznam nainštalovaných balíčkov sa uchováva v súbore `package.json`, taktiež umiestenom v koreňovom adresári projektu.

3.3 Relačné databázy

V dnešnej dobe štandardný typ databázy pre ukladanie a prístup k dátam vo vzájomných väzbách. Dáta sú rozdelené do tabuliek, kde každý riadok reprezentuje jednu entitu. Entita je vždy identifikovateľná svojim primárnym kľúčom, ktorý musí byť v danej tabuľke unikátny. Každý stĺpec tabuľky má vopred definovaný dátový typ a nesie hodnoty pre každý element v tabuľke.

SQL (*Structured Query Language*) je štandardizovaný dotazovací jazyk pre prístup a manipuláciu s dátami v relačných databázach.

3.3.1 PostgreSQL

Plnohodnotný open-source objektovo-orientovaný databázový systém, ktorý je v aktívnom vývoji už viac ako 30 rokov. PostgreSQL je kompatibilný so všetkými hlavnými platformami ako Linux, MacOS, Solaris a Windows. Tento systém si obľúbili mnohé veľké spoločnosti po celom svete. [14]

3.3.2 Prisma

Prisma je open-source *sada nástrojov pre správu databázy*. Nahradzuje tradičné ORM[2] a uľahčuje prístup k databáze pomocou automaticky generovaného klienta pre zostavovanie dotazov. Prisma je implementovaná v jazyku TypeScript, čo znamená, že *podporuje striktnú typovú kontrolu* a zmenšuje tým pravdepodobnosť chyby vo vývoji. [11]

```
await prisma.users.create({
    data: {
    firstName: "Alice",
    email: "alice@prisma.io",
    active: true,
  }
})
```

Výpis 3.2: Príklad tvorby užívateľa pomocou nástroja Prisma. [11]

Vďaka sade nástrojov `prisma` nie je nutné pristupovať k databáze priamo pomocou SQL dotazov, ale je možné využiť API automaticky generovaného klienta.

[2] ORM – Object-relational mapping

3.4 GraphQL

Dotazovací (query) jazyk pre API a runtime na strane servera pre vykonávanie dotazov pomocou *vopred definovaného* typového systému. GraphQL nie je viazané žiadnou špecifickou technológou ukladania dát, naopak pracuje ako backend pre už existujúci kód a dáta. [13] GraphQL nie je implementácia, ale iba špecifikácia. Existujú mnohé implementácie GraphQL implementované v mnohých programovacích jazykoch, na rôznych platformách.

GraphQL služba je vytvorená definovaním typov a ich jednotlivých polí. Každému polu každého typu je následne priradená funkcia. [13] Tieto funkcie sú nazývané „*resolvers*“. Tieto funkcie vyhodnocujú dáta jednotlivých polí pri prichádzajúcej požiadavke.

```
type Query {
  me: User
}

type User {
  id: ID
  name: String
}
```

Výpis 3.3: Príklad jednoduchej GraphQL schémy. [13]

Po spustení GraphQL služby (typicky URL adresa dostupná cez HTTPS) očakáva GraphQL dotazy, ktoré spracováva a vyhodnocuje. Obdržané dotazy najskôr skontroluje, či požadujú iba typy a polia, ktoré sú definované. V prípade, že sú dotazy v poriadku, Graphql spustí požadované resolvery a vracia výsledok. [13]

GraphQL rozlišuje tri typy operácií s dátami. Prvý typ operácie je *query* a slúži výhradne na čítanie dát. Druhým typ operácie sa nazýva *mutation* a slúži na úpravu, vkladanie alebo odstránenie dát. Tretím a posledným typom operácie je *subscription*. Ak GraphQL služba spracuje operáciu typu *subscription*, pošle vyžiadané dáta klientovi pri každej ich zmene.

Pre ukážku, *query* dotaz na GraphQL službu so schémou ukázanou vo výpise 3.3:

```
query {
  me {
    name
  }
}
```

Výpis 3.4: Príklad *query* dotazu na GraphQL službu. [13]

. . . by po spracovaní GraphQL službou mohol vrátiť JSON[3] objekt:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

Výpis 3.5: Príklad dát obdržaných z GraphQL služby. [13]

[3] JSON - javascript object notation

3.4.1 Apollo Client

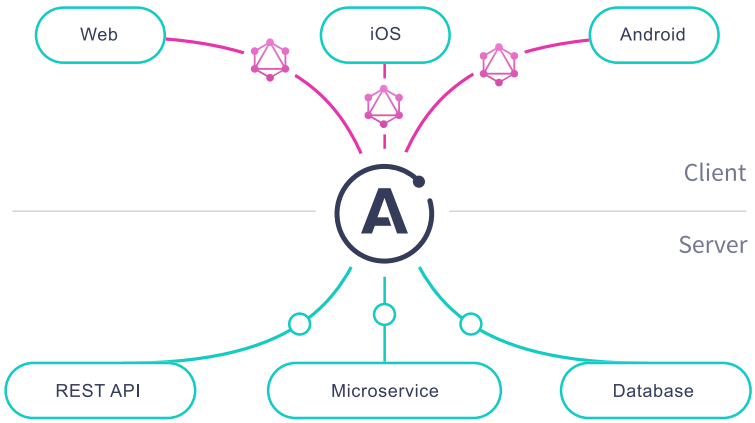
Implementácia GraphQL špecifikácie pre klientské aplikácie. Apollo Client je priamo prepojený s knižnicou React a poskytuje nástroje umožňujúce rýchly vývoj webových aplikácií, ktoré získavajú dáta z GraphQL služby.

> Apollo Client je knižnica umožňujúca plnohodnotnú správu stavu dát v JavaScriptových aplikáciach. Umožňuje vývojárovi len napísať požadovaný GraphQL dotaz a Apollo Client sa už postará o získanie, uloženie dát do cache a obnovenie uživateľského rozhrania. Využívanie knižnice Apollo Client vedie vývojára ku správnej štruktúre kódu. (Dokumentácia Apollo GraphQL [6])

Apollo GraphQL umožňuje využiť svoju cache aj pre ukladanie lokálneho stavu aplikácie. Vďaka tomu sa pre danú aplikáciu stáva jediným zdrojom dát.

3.4.2 Apollo Server

Serverová časť GraphQL implementácie, kompatibilná so všetkými GraphQL kientami, vrátane Apollo Client. Apollo Server dokáže pracovať ako samostatný GraphQL server alebo ako súčasť už existujúcej aplikácie ako napríklad Express[4] server. [6]



Obr. 3.1: Diagram architektúry Apollo Server. [6]

Apollo Server dokáže pôsobiť ako *jednotná brána* pre všetky klientské aplikácie. Brána má predom definovanú schému, tzn. že klient presne vie aké operácie môže vykonávať a aké dáta môže očakávať späť.

3.5 Slack API

Služba pre tímovú a pracovnú komunikáciu Slack umožňuje vývojárom vytvoriť vlastné aplikácie (Slack apps). Tieto aplikácie následne môžu vykonávať rôzne akcie v pracovných prostrediach, v ktorých sú nainštalované. Vo svojej podstate je Slack aplikácia iba HTTP server komunikujúci s verejnou API služby Slack. Vždy, keď užívateľ v aplikácií vykoná

[4] https://expressjs.com/

nejakú akciu, Slack API upozorní HTTP server danej aplikácie, ktorý akciu spracuje a vyhodnotí (napr. otvorí dialógové okno).

Aplikácie môžu vykonávať niektoré akcie bežných užívateľov:

- *Odosielať správy* do rôznych koverzácií. [12]
- *Čítať správy a konverzácie.* [12]
- *Vytvoriť, archivovať a spravovať konverzácie.* [12]
- *Reagovať na označenia* od užívateľov. [12]

. . . no verejná API aplikáciam umožňuje aj akcie mimo uživateľských práv:

- *Otvoriť dialógové okná* pre získanie alebo zobrazenie extra informácií. [12]
- *Zostaviť interaktívne komponenty* na ktoré môzu reagovať [12]
- *Vytvoriť a aktualizovať „Home tab“ (domovskú obrazovku)* kde môže užívateľ interagovať s aplikáciou. [12]
- *Definovať skratky* vďaka ktorým užívateľ môže rýchlo vykonať akcie v aplikácií. [12]

3.5.1 Block Kit

Framework pre tvorbu uživateľského rozhrania v Slack aplikáciách, ktorý ponúka kompromis kontroly a flexibility pri budovaní rozhraní v správach. [12]

Užívateľské rozhranie v Slack aplikácií sa skladá z blokov rôznych typov vo forme JSON objektov usporiadaných do lineárneho zoznamu.

```
{
  "type": "section",
  "text": {
    "type": "mrkdwn",
    "text": "Hello, *World*!"
  }
},
```

Výpis 3.6: Príklad jednoduchého bloku v Slack aplikácií.

Spolu s blokmi sú v tejto správe aj metadáta, pomocou ktorých API vie ako má so správou naložiť. Hlavným parametrom metadát je *type*, ktorý môže nadobudnúť jednu z dvoch hodnôt:

- `home` – Špecifikuje, že rozhranie má byť použité pre domovskú obrazovku (app_home).
- `modal` – Špecifikuje, že rozhranie má byť použité pre dialógové okno.

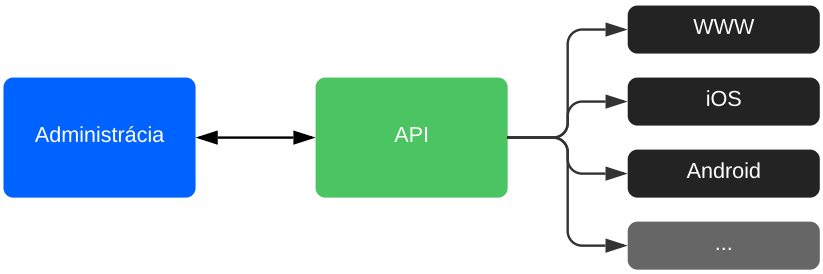
Lineárny zoznam s blokmi užívateľského rozhrania sa vloží do parametru „view“ v HTTP správe odoslanej na verejnú API služby Slack. Niektoré bloky podporujú formátovanie svojho obsahu pomocou modifikovanej verzie jazyka *markdown*.

Kapitola 4

Headless CMS

Headless CMS sú alternatívou k štandardným redakčným systémom ako napríklad populárny WordPress[1]. Rozdiel medzi headless a štandardnými redakčnými systémami je, že headless systémy nemajú vlastný frontend pre zobrazovanie obsahu a teda *neriešia samotné zobrazenie obsahu* konzumentom.

Takéto systémy typicky disponujú rozhraním REST[2] alebo GraphQL, ktoré implementuje aj táto práca. Jediným spôsobom ako získať obsah z redakčného systému je využiť niektoré z dostupných rozhraní poskytované konkrétnym riešením. Výhodou oproti tradičným redakčným systémom je možnosť získané dáta optimálne zobraziť na rôznych zariadeniach.



Obr. 4.1: Ilustračná schéma generického headless redakčného systému.

Členenie obsahu v takýchto redakčných systémoch je typicky v dvoch vrstvách – *kategórie* a *obsahové typy* (komponenty).

Kategórie sú zoznamy združujúce jednotlivé komponenty, môžu byť homogénne (všetky prvky zoznamu sú jedného typu) alebo heterogénne (prvky zoznamu sú typicky iných typov).

Komponenty sú atomickými prvkami headless redakčných systémov. Môžu nadobúdať rôznych typov, ktoré určujú ich vnútornú dátovú štruktúru. Typický príklad často používaných typov komponentov je napríklad `prostý text` alebo `odkaz`. Niektoré headless redakčné systémy umožňujú vytvárať aj vlastné typy komponentov a tak si prispôsobiť dáta vlastným špecifickým potrebám.

[1] https://wordpress.com/
[2] REST – Representational State Transfer

4.1 Existujúce riešenia

Headless redakčné systémy získavajú na popularite a v súčasnej dobe existuje množstvo riešení, ktoré sa typicky rozdeľujú do dvoch skupín podľa ukladania a správy dát – *API-based* a *Git-based*. V tejto sekcií sú predstavené dva najpopulárnejšie systémy z každej kategórie.

4.1.1 Strapi

Najpopulárnejší *API-based* headless redakčný systém. Disponuje administračným panelom zostaveným na mieru, REST aj GraphQL rozhraním, systémom uživateľských práv a mnohými inými vlastnosťami. Tento redakčný systém má aj vlastný obchod s aplikáciami, ktoré si môžu používatelia pridať a tým rozšíriť funkcionalitu.

> Strapi je flexibilný, open-source[3] headless CMS[4], ktorý dáva vývojárom slobodu voľby ich obľúbených nástrojov a zároveň dovoľuje editorom jednoducho spravovať a distribuovať ich obsah. (Dokumentácia Strapi.io [5])

V prípade, že by užívateľ Strapi redakčného systému mal nakonfigurovanú kolekciu s názvom „restaurants“, získanie celkového počtu týchto kolekcií v systéme by mohlo vyzerať takto:

```
1 GET "http://localhost:1337/restaurants/count" // Response: 1
```

Výpis 4.1: Príklad HTTP požiadavku na REST rozhranie Strapi.

Výhody: Vďaka možnosti definovať vlastné dátové typy je systém flexibilný.

Nevýhody: Pre použitie Strapi je nutné systém spustiť na vlastnej infraštruktúre, pripojiť k predom vytvorenej relačnej databáze a celý systém nakonfigurovať.

4.1.2 Netlify CMS

Najpopulárnejší *Git-based* headless redakčný systém. Netlify na rozdiel od *API-based* headless redakčných systémov nevyužíva pre ukladanie svojich dát relačnú databázu. Pre uloženie celého obsahu webovej aplikácie využíva repozitáre vytvorené v prostredí `git`.

> Jadro Netlify CMS tvorí React [2.6] aplikácia ktorá využíva rozhranie pre prácu s GitHub[5], GitLab[6] alebo Bitbucket[7] API. (Dokumentácia Netlify CMS [4])

Netlify sa využíva väčšinou pre menšie stránky ako sú napríklad dokumentácie alebo produktové stránky, pretože umožňuje udržovať obsah relevantný k danej verzií produktu.

Výhody: Veľmi jednoduchý na používanie, nevyžaduje vlastnú databázu.

Nevýhody: Vhodný len pre statické stránky, obtiažnejšie organizovanie obsahu.

[3] open-source – otvorený, verejný kód, zväčša vyvíjaný komunitou
[4] CMS – ang. content management system (redakčný systém)
[5] https://developer.github.com/v3/
[6] https://docs.gitlab.com/ee/api/
[7] https://confluence.atlassian.com/bitbucket/

Kapitola 5

Návrh riešenia

Kapitola popisuje návrh systému, ktorý je predmetom tejto práce (ďalej ako „**Slackify**“). Výsledný návrh vznikol po nadobudnutí požadovaných znalostí a analýz požiadaviek kladených na výsledný produkt.

Sekcia 5.1 je venovaná konkrétnym funkčným požiadavkám na systém, sekcia 5.2 prípadom použitia Slackify v rôznych scenároch, sekcia 5.3 architektúre systému aj s jej neúspešnými iteráciami. Posledná sekcia 5.4 popisuje špeciálne dátové typy využívané pre uchovávanie a distribúciu dát v redakčnom systéme Slackify.

5.1 Funkčné požiadavky

Zadanie, ktorého celé znenie je priložené na strane 2, popisuje výsledné riešenie práce ako *headless* redakčný systém. Súhrn hlavných požiadaviek na systém je nasledovný:

- Užívateľ systému by mal byť schopný spravovať obsah v *plnej miere* a bez obmedzení v prostredí služby Slack.
- Užívateľ by mal byť schopný začať využívať službu *okamžite* po nainštalovaní do svojho pracovného prostredia v službe Slack. To znamená, že žiadna konfigurácia alebo nasadenie na vlastnú infraštruktúru by nemalo byť požadované.
- Všetky vytvorené dáta by mali byť dostupné cez verejné API.
- Systém by mal disponovať webovým rozhraním, odkiaľ by malo byť možné spravovať obsah a nastavenia.

Hlavnou výsadou práce nie je čo najväčšia flexibilita konfigurácie, ale jednoduchosť v použití a možnosť rýchleho nasadenia.

5.1.1 Odchýlky návrhu od pôvodného zadania

Slack API umožňuje vytvoriť uživateľské rozhranie zložené z maximálneho počtu 100 blokov. Pre tento dôvod je maximálny počet zobrazených prvkov v zoznamoch Slack aplikácie limitovaný. Prvky zoznamov nad určený limit sú skryté a užívateľ ich môže spravovať iba vo webovom prostredí. Nie je teda možné spravovať obsah redakčného systému v plnej miere za použitia Slack aplikácie ako určuje prvá funkčná požiadavka.

5.1.2 Porovnanie s existujúcimi riešeniami

Slackify patrí do kategórie *API-driven* headless redakčných systémov. Na rozdiel od iných systémov z tejto kategórie však Slackify nevyžaduje žiadnu vstupnú konfiguráciu, pretože už disponuje vopred definovanými dátovými typmi, ktoré je možné distribuovať. Redakčný systém je teda možné využívať okamžite, no za cenu menšej flexibility.

5.2 Diagramy prípadov použitia

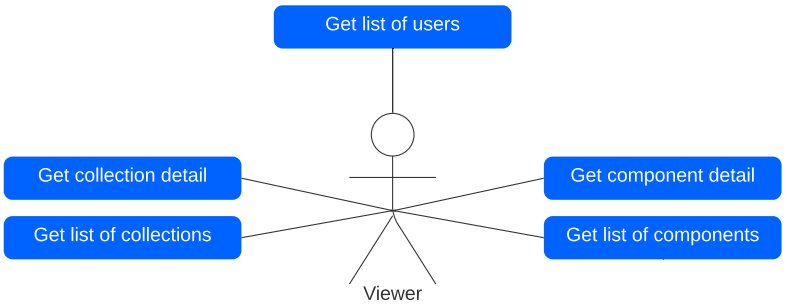
Diagramy prípadov použitia ukazujú možnosti práce s redakčným systémom Slakcify z pohľadu užívateľov s rôznymi prístupovými oprávneniami (rolami) a zariadenia, ktoré sa snaží získať obsah z verejného rozhrania.

5.2.1 Viewer

Každému novému užívateľovi, ktorý prepojí svoj účet v službe Slack so Slackify je automaticky priradená základná užívateľská rola *Viewer*.

Užívateľ s rolou *Viewer* môže vykonávať nasledujúce akcie:

- `Získať detail kolekcie/komponentu` – Zobrazenie všetkých informácií o kolekcií alebo alebo komponente. Dáta je možné iba čítať, nie modifikovať.
- `Získať list kolekcií/komponentov` – Zobrazenie zoznamu kolekcií alebo komponentov. Komponenty je možné zobraziť všetky alebo podľa pridelených kolekcií. Dáta je možné iba čítať, nie modifikovať.



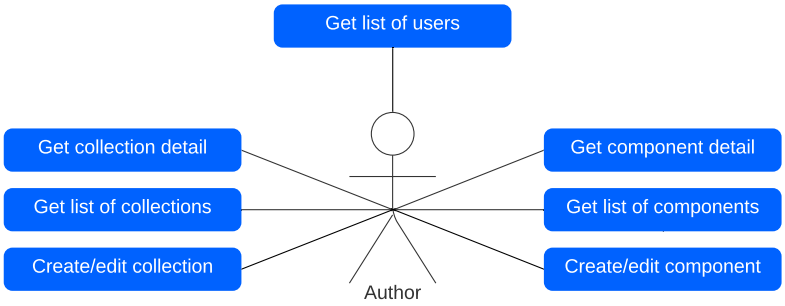
Obr. 5.1: Diagram prípadov použitia – *Viewer*.

5.2.2 Author

Užívateľská rola *Author* zahŕňa všetky právomoci, ktoré mala rola *Viewer* a rozširuje ich o možnosť vykonávať nasledujúce akcie:

- `Vytvoriť kolekciu/komponentu` – Po zadaní a potvrdení vstupných údajov je nová kolekcia alebo komponent pridaná do databázy. Nové komponenty sú inicializované so stavom *hidden* (skrytý, nepublikovaný).

- `Upraviť kolekciu/komponentu` – Kolekcie alebo komponety je možné upraviť v ľubovoľný čas. Jediná hodnota, ktorú nie je možné modifikovať je typ kolekcie alebo komponentu.

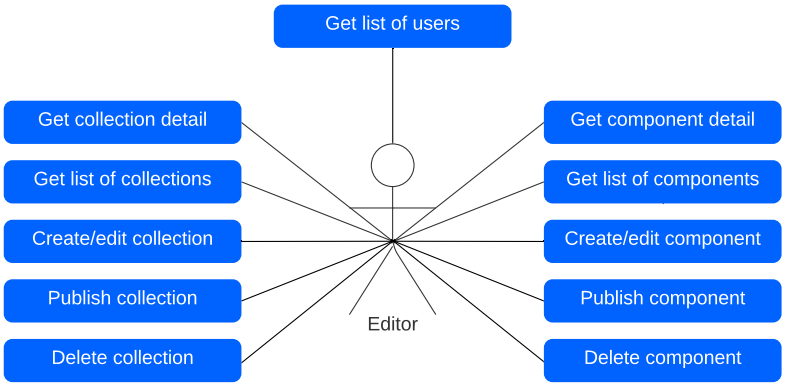


Obr. 5.2: Diagram prípadov použitia – *Autor*.

5.2.3 Editor

Užívateľská rola *Editor* zahŕňa všetky právomoci, ktoré mala rola *Author* a rozširuje ich o možnosť vykonávať nasledujúce akcie:

- `Zmazať kolekciu/komponent` – Kolekcie alebo komponety je možné zmazať v ľubovoľný čas. Pri zmazaní kolekcie sa zmazú aj všetky komponenty k nej priradené.
- `Publikovať kolekciu/komponent` – Publikovaná kolekcia alebo komponent je okamžite dostupná verejne. *Editor* rovnako môže kolekcie alebo komponenty skryť.

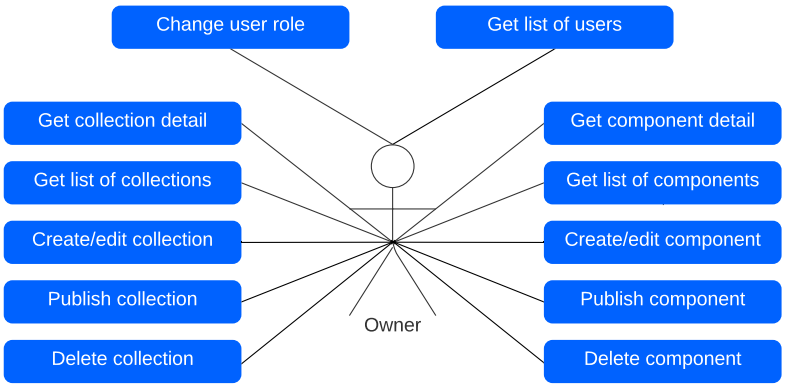


Obr. 5.3: Diagram prípadov použitia – *Editor*.

5.2.4 Owner

Užívateľská rola *Owner* je rola s najvyššími právomocami. Zahŕňa všetky právomoci, ktoré mala rola *Editor* a rozširuje ich o možnosť vykonávať nasledujúcu akciu:

- `Zmeniť rolu užívateľa` – Umožňuje modifikovať užívateľské role ostatných užívateľov. Užívateľ nemôže zmeniť rolu sám sebe.



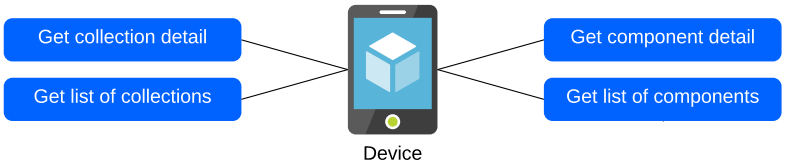
Obr. 5.4: Diagram prípadov použitia – *Owner*.

5.2.5 Zariadenie

Obsah zo Slackify je možné získať z verejného API prostredníctvom akéhokoľvek zariadenia, ktoré dokáže komunikovať cez protokol HTTP(S). Pre ukážku diagramu prípadov použitia je na mobilnom telefóne nainštalovaná aplikácia, ktorá komunikuje s rozhraním Slackify.

Zariadenie môže v prostredí Slackify vykonávať nasledujúce akcie:

- `Vyžiadať detail kolekcie/komponentu` – Informácie o jednej kolekcií alebo komponente je možné vyžiadať od API pomocou ich unikátneho ID.
- `Vyžiadať list kolekcií` – Zariadenie si môže vyžiadať zoznam kolekcií. Zoznam kolekcií je v základnom nastavení zoradený chronologicky podľa času vytvorenia. Výsledky je možné zoradiť alebo filtrovať pomocou parametrov operácie.
- `Vyžiadať list komponentov` – Pomocou unikátneho ID kolekcie je možné vyžiadať k nej priradené komponenty. V prípade neuvedenia ID kolekcie je navrátený list všetkých komponentov v danom Slack pracovnom prostredí (zoradené chronologicky podľa času vytvorenia). Výsledky je možné zoradiť alebo filtrovať pomocou parametrov operácie.



Obr. 5.5: Diagram prípadov použitia – *Zariadenie*.

5.3 Návrh architektúry

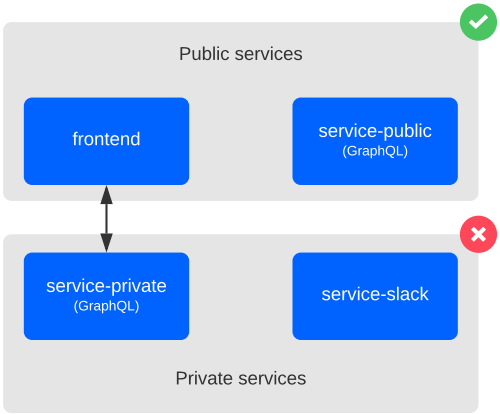
V počiatočných návrhoch architektúry bol systém rozdelený do dvoch častí:

- `frontend` – Webová konzola poskytujúca prístup k obsahu a nastaveniam Slackify.
- `backend` – Služba zložená z troch hlavných súčastí:
  - Skryté GraphQL rozhranie poskytujúce prístup k dátam a logiku pre `frontend`.
  - Verejné GraphQL rozhranie poskytujúce prístup k obsahu redakčného systému.
  - Rozhranie zodpovedné za komunikáciu s verejným API služby Slack.

Neskôr bola však služba `backend` rozdelená do troch menších samostatných služieb. Finálna architektúra redakčného systému Slackify sa teda skladá z týchto služieb:

- `frontend` – Webová konzola poskytujúca prístup k obsahu a nastaveniam Slackify. Táto časť zostala nezmenená.
- `service-slack` – Rozhranie zodpovedné za komunikáciu s verejným API služby Slack. Udalosti a akcie prichádzajúce zo Slack aplikácie sú smerované na túto službu, ktorá ich spracuje a vyhodnotí.
- `service-private` – Skryté GraphQL rozhranie poskytujúce prístup k dátam a logiku pre `frontend`
- `service-public` – Verejné GraphQL rozhranie poskytujúce prístup k obsahu redakčného systému.

Takáto architektúra sa oproti jednej veľkej `backend` službe ukázala výhodnejšia z dvoch hlavných dôvodov. Prvým dôvodom je *jednoduchšia údržba* (napríklad udržovanie logickej súborovej štruktúry bolo pri jednej monolitickej službe obtiažne). Druhým významným dôvodom je možnosť jednotlivé služby *verziovať a vydávať samostatne*, pretože každá služba je nezávislá od ostatných.



Obr. 5.6: Vizualizácia architektúry podľa návrhu.

Všetky služby využívajú *zdieľanú databázovú vrstvu*, ktorá poskytuje prístup k dátam uloženým v relačnej databáze. Celkovo je teda architektúra redakčného systému rozdelená do piatich samostatných súčastí.

5.4 Dátové typy redakčného systému

Slackify je headless redakčný systém obsahujúci dva hlavné dátové typy – *collection* (kolekcia) a *component* (komponent). Pomocou týchto dvoch dátových typov je možné vytvoriť logickú štruktúru obsahu, ktorú je možné následne distribuovať pomocou verejného API. Oba dátové typy uchovávajú okrem svojich vlastných parametrov aj momentálny stav uverejnenosti *published*, repretenzovaný dátovým typom `boolean`.

5.4.1 Component

Component (komponent) je základná dátová struktúra v Slackify. Celý obsah redakčného systému je zložený z komponentov usporiadaných do kolekcií [5.4.2]. Každý komponent má povinnú položku *type* (typ), ktorá určuje ďalšie zloženie štruktúry.

Typ „`Plain Text`“

Najjednoduchší typ obsahujúci jediné povinné textové pole s názvom *text*.

Typ „`Article`“

Typ pre vytvorenie jednoduchého článku. Obsahuje povinné textové položky *title* (titulok), *content* (obsah článku) a nepovinnú textovú položku *lead* (úvodný text).

Typ „`Link`“

Typ pre vytvorenie hypertextového odkazu. Obsahuje povinnú textovú položku *url* (cieľová adresa) a voliteľnú textovú položku *text*.

5.4.2 Collection

Collection (kolekcia) je zoznam slúžiaci pre kategorizovanie jednotlivých komponentov. Každá kolekcia ich môže obsahovať 0 až n. Jedná sa o *homogénny zoznam*, tzn. že pri tvorbe kolekcie sa vždy musí zvoliť práve jeden typ komponentov z ktorých sa môže zoznam skladať. Každá kolekcia má povinnú položku *name* (názov) a nepovinnú *description* (popis).

Kapitola 6

Implementácia

Ako vysvetľuje sekcia 5.3, headless redakčný systém Slackify je rozdelený do spoločnej databázovej vstvy a štyroch samostatných služieb. Táto kapitola popisuje vybrané časti implementácie všetkých piatich súčastí systému.

Sekcia 6.1 je venovaná implementácií spoločnej databázovej vrstvy, sekcia 6.2 implementácií verejnej a skrytej GraphQL služby, sekcia 6.4 implementácií Slack aplikácie. Posledná sekcia 6.3 je venovaná implementácií webovej konzoly.

6.1 Databázová vrstva

Zdieľaná súčasť systému, ktorá poskytuje prístup k databáze službám `service-private`, `service-public` a `service-slack`. Pre zostavenie databázy a prístup k nej je využitá sada nástrojov `prisma`, bližšie popísaná v sekcií 3.3.

6.1.1 Schéma

Hlavný konfiguračný súbor sady nástrojov `prisma`, pomenovaný `schema.prisma`, ktorý sa skladá z troch hlavných súčastí:

- `datasource` – Konfiguračný blok špecifikujúci typ zdroja dát (databázy) a údaje potrebné k nadviazaniu úspešného pripojenia. Slackify využíva PostgreSQL databázu ako jediný zdroj dát. Environmentálna premenná `DATABASE_URL` určuje URL pre pripojenie k danej databáze (viz výpis 6.1).
- `generator` – Konfiguračný blok obsahujúci nastavenia automaticky generovaného databázového klienta. Slackify generuje iba jednoho databázového klienta do zložky `node_modules` v koreňovom adresári systému. Tento databázový klient je zdieľaný ostatnými službami (viac viz 6.1.3).
- `model` – Jednotlivé dátové modely systému, bližšie popísané v sekcií 6.1.2.

Sada nástrojov `prisma` využíva konfiguračný súbor aj pre *migrácie* databázy. Pri každej zmene dátových modelov automaticky zmení schému databázy a vytvorí nový záznam v zložke `migrations`. Tento záznam obsahuje výpis zmien, ktoré migrácia aplikuje, obsah konfiguračného súboru po aplikovaní migrácie a jednotlivé kroky, ktoré migrácia vykoná pre zmenu schémy databázy. Vďaka týmto záznamom je schému databázy možné vrátiť do stavu pred aplikovaním jednej alebo viacerých migrácií.

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
  enabled  = env("DATABASE_URL")
}

generator cient_common {
  provider = "prisma-client-js"
  output   = "../../node_modules/@prisma/client"
}
```

Výpis 6.1: Špecifikácia `datasource` a `generator` v konfiguračnom súbore `prisma`.

6.1.2 Dátové modely

Dátové modely definované v konfiguračnom súbore `prisma` reprezentujú entity využívané v aplikácií. Sada nástrojov `prisma` tieto modely automaticky mapuje do jednotlivých tabuliek v databáze.

Slackify obsahuje dva hlavné modely `Collection` a `Component` reprezentujúce obsah uložený v redakčnom systéme. Ďalšie modely `User` a `Team` služia pre uchovávanie informácií o užívateľoch a tímoch (pracovných prostredí) *prepojených so službou Slack*.

Model `Collection`

Dátový model `Collection` (kolekcia) slúžiaci ako zoznam pre jednotlivé komponenty. Sú to homogénne zoznamy, ktoré môžu obsahovať iba komponenty jednoho typu.

```
model Collection {
  id          String        @default(cuid()) @id
  name        String
  type        ComponentType
  published   Boolean       @default(false)
  description String?
  team        Team          @relation(...)
  teamId      String
  components  Component[]
  createdAt   DateTime      @default(now())
  updatedAt   DateTime      @updatedAt
}
```

Výpis 6.2: Dátový model `Collection` v konfiguračnom súbore `prisma`.

Popis niektorých významných položiek obsiahutých v dátovom modeli:

- `type` – Určuje jediný typ komponentu, ktorý je možné vložiť do danej kolekcie. Typ komponentu je bližšie popísaný v sekcií 6.1.2.
- `team` – Relácia s dátovým modelom `Team` ku ktorému je kolekcia priradená. Každá kolekcia musí byť priradená k práve jednej entite modelu `Team`.
- `components` – Relácia $1 : n$ všetkých komponentov priradených ku kolekcií. Každá kolekcia môže mať 0 až ∞ priradených komponentov.

Model `Component`

Dátový model `Component` (komponent) reprezentujúci malú časť obsahu v Slackify. Každý komponent *musí mať zvolený typ*, ktorý určuje vnútornú štruktúru dát uložených v komponente.

```
model Component {
  id              String                  @default(cuid()) @id
  type            ComponentType           @default(PLAIN_TEXT)
  published       Boolean                 @default(false)
  collection      Collection              @relation(...)
  collectionId    String
  author          User                    @relation(...)
  authorId        String
  team            Team                    @relation(...)
  teamId          String
  plainTextDataId String?
  plainTextData   PlainTextComponentData? @relation(...)
  articleDataId   String?
  articleData     ArticleComponentData?   @relation(...)
  linkDataId      String?
  linkData        LinkComponentData?      @relation(...)
  createdAt       DateTime                @default(now())
  updatedAt       DateTime                @updatedAt
}
```

Výpis 6.3: Dátový model `Component` v konfiguračnom súbore `prisma`.

Popis niektorých významých položiek obsiahutých v dátovom modeli:

- `type` – Typ komponentu nadobúdajúci jednu z hodnôt výčtového typu `ComponentType`.
- `collection` – Relácia s dátovým modelom `Collection`. Každý komponent musí byť priradený k práve jednej kolekcií.
- `author` – Relácia s dátovým modelom `Author`. Každý komponent musí byť priradený k právej jednomu autorovi.
- `team` – Relácia s dátovým modelom `Team`. Každý komponent musí byť priradený k práve jednomu tímu.

Hodnota položky `type` určuje práve jednu z položiek `plainTextData`, `articleData` alebo `linkData` ako povinnú. Tieto položky ukazujú na rôzne špecifické dátové modely, ktoré reprezentujú obsah daných komponentov. Napríklad ak položka `type` má hodnotu `LINK`, položka `linkData` *musí* byť reláciou na dátový model `LinkComponentData`, ktorý obsahuje povinné pole URL a nepovinné pole text.

```
model LinkComponentData {
  id   String  @default(cuid()) @id
  text String?
  url  String
}
```

Výpis 6.4: Dátový model `LinkComponentData` v konfiguračnom súbore `prisma`.

Výčtový typ `ComponentType`

Výčtový typ (enum), ktorý môže nadobúdať jednu z hodnôt `PLAIN_TEXT`, `ARTICLE` alebo `LINK`.

```
enum ComponentType {
  PLAIN_TEXT
  ARTICLE
  LINK
}
```

Výpis 6.5: Výčtový typ `ComponentType` v konfiguračnom súbore `prisma`.

Model `User`

Dátový model `User` (užívateľ) reprezentujúci informácie o užívateľskom účte prepojenom so službou Slack.

```
model User {
  id          String      @id
  email       String      @unique
  name        String
  role        UserRole    @default(VIEWER)
  accessToken String      @unique
  image_24    String?
  image_32    String?
  image_48    String?
  image_72    String?
  image_192   String?
  image_512   String?
  team        Team        @relation(fields: [teamId], references: [id])
  teamId      String
  components  Component[]
}
```

Výpis 6.6: Dátový model `User` v konfiguračnom súbore `prisma`.

Popis niektorých významných položiek obsiahutých v dátovom modeli:

- `id` – Jedinečný identifikátor užívateľa, prevzatý z autentifikačnej API služby Slack.
- `role` – Užívateľská rola nadobúdajúca jednu z hodnôt výčtového typu `UserRole`. Určuje právomoci užívateľa, inicializuje sa s hodnotou `VIEWER`.
- `team` – Relácia s dátovým modelom `Team`. Každý užívateľ musí byť priradený k práve jednomu tímu.
- `components` – Relácia $1 : n$ všetkých komponentov, ktorým je uživateľ autorom. Každý užívateľ môže byť autorom 0 až ∞ komponentov.

Výčtový typ `UserRole`

Výčtový typ (enum), ktorý môže nadobúdať jednu z hodnôt `VIEWER`, `AUTHOR`, `EDITOR` alebo `OWNER`.

```
enum UserRole {
  OWNER
  EDITOR
  AUTHOR
  VIEWER
}
```

Výpis 6.7: Výčtový typ `UserRole` v konfiguračnom súbore `prisma`.

Model `Team`

Dátový model `Team` (tím) reprezentujúci informácie o tíme (pracovnom prostredí, workspace) v službe Slack.

```
model Team {
  id          String       @id
  name        String
  domain      String       @unique
  accessToken String       @unique
  collections Collection[]
  users       User[]
  components  Component[]
}
```

Výpis 6.8: Dátový model `Team` v konfiguračnom súbore `prisma`.

Popis niektorých významných položiek obsiahutých v dátovom modeli:

- `id` – Jedinečný identifikátor tímu, prevzatý z autentifikačnej API služby Slack.
- `accessToken` – Generovaný prístupový kód, ktorý je nutné využiť pre získanie obsahu z redakčného systému cez verejnú api (službu `service-public`).
- `collections` – Relácia $1 : n$ všetkých kolekcií, ktoré sú vytvorené pod daným tímom. Každý tím môže mať priradených 0 až ∞ kolekcií.
- `users` – Relácia $1 : n$ všetkých užívateľov, ktorí sú členmi daného tímu. Každý tím môže mať 0 až ∞ užívateľov.
- `components` – Relácia $1 : n$ všetkých komponentov, ktoré sú vytvorené pod daným tímom. Každý tím môže mať priradených 0 až ∞ komponentov.

6.1.3 Automaticky generovaný databázový klient

Pomocou konfiguračného súboru `prisma` automaticky zostrojí striktne typovaného databázové klienta, ktorý dokáže zostavovať dotazy pre databázu. Pomocou tohto klienta je možné vytvárať, čítať, modifikovať, či mazať dáta v databáze.

Databázový klient je generovaný do zložky `node_modules`, odkiaľ je možné ho importovať ako z modulu `@prisma/client`.

```
import { PrismaClient } from "@prisma/client";

export const prisma = new PrismaClient();
```

Výpis 6.9: Príklad vytvorenia instancie databázového klienta v Slackify.

6.2 GraphQL služby

Ako bolo vysvetlené v sekcií 5.3 o návrhu architektúry, redakčný systém Slackify disponuje dvomi GraphQL službami – skrytou `service-private` a verejnou `service-public`. Obe GraphQL služby sú implementované pomocou balíčka Apollo Server.

6.2.1 Schéma

GraphQL schémy oboch služieb sú implementované pomocou balíčka `Nexus Schema`, ktorý umožňuje vytvoriť GraphQL schému v jazyku JavaScript namiesto SDL[1] definovanom v GraphQL špecifikácií. Vďaka tomu je možné budovať schému „code-first“ a silno typovanú.

```
export const Query = queryType({
  definition(t) {
    /* Users */
    t.crud.users({
      filtering: {
        team: true,
      },
    });
  }
});
```

Výpis 6.10: Časť GraphQL schémy služby `service-private`.

Balíček `Nexus Schema` disponuje aj pluginom, ktorý ho dokáže prepojiť s `prisma` klientom. Toto prepojenie rozširuje triedu `t` (prvý parameter v bloku `definition()`) o dve položky:

- `t.model` – Mapovanie položiek databázových modelov na GraphQL typy.
- `t.crud` – Databázové operácie automaticky generované pre každý databázový model. Operácie podporujú stránkovanie, filtrovanie alebo zoraďovanie výsledkov. Vo výpise 6.10 je znázornená operácia na získanie listu všetkých užívateľov s možnosťou filtrácie podľa tímu.

Plugin pre prepojenie schémy s `prisma` klientom však nepodporuje kontrolu autorizácie užívateľa, tá je riešená použitím „middleware“ funkcií.

6.2.2 Autorizácia

Každá požiadavka a mutácia (s výnimkou mutácie `signIn()` služby `service-private`) musí obsahovať HTTP hlavičku s názvom `Authorization`. Obe GraphQL služby však vyžadujú rozdieľne hodnoty tejto hlavičky, bližšie popísané v sekciách 6.2.3 a 6.2.4.

GraphQL Shield

Skrytá aj verejná GraphQL služba pre kontrolu prístupu využíva aj middleware `GraphQL Shield`, ktorý umožňuje vytvoriť „pravidlá“ pre prístup ku konkrétnym požiadavkám alebo mutáciam.

[1] SDL – Schema Definition Language

```
export const isAuthenticated = rule({ cache: 'contextual' })(
  async (_parent, _args, { user }: Context) => {
    return user !== undefined;
  }
);
```

Výpis 6.11: Pravidlo `GraphQL Shield` kontrolujúce či je užívateľ autentifikovaný.

Podobné pravidlá ako vo výpise 6.11 sú využité pre kontrolu prístupu v celej GraphQL schéme. Ak niektorá z podmienok pravidla nie je splnená, dané pravidlo obvykle vráti instanciu typu `Error` so špecifickou správou, ktorá je zaslaná späť do webovej konzoly. Ak sú však splnené *všetky podmienky*, pravidlo vracia boolean hodnotu `true` a riadenie je prenechané príslušnej GraphQL `resolver` funkcií.

6.2.3 Skrytá služba `service-private`

Skrytá GraphQL služba (`service-private`) poskytuje prístup k dátam pre webové konzolu (službu `frontend`, ktorej implementácia je popísaná v sekcií 6.3). Schéma tejto služby je tvorená prevažne CRUD operáciami nad databázovými modelmi `Collection` a `Component`.

Autorizácia

HTTP hlavička `Authorization` požiadavky alebo mutácie obsahuje JWT konkrétneho užívateľa. Tento token obsahuje okrem iných informácií o užívateľovi aj jeho ID. Vďaka tomu je možné pred vykonaním každej operácie získať všetky dostupné informácie o užívateľovi a jeho tíme, ktoré sú následne dostupné v kontexte každej GraphQL `resolver` funkcií. Ak nebolo možné nájsť užívateľa s daným ID, dáta o ňom a jeho tíme majú v GraphQL `resolver` funkcií hodnotu `undefined`.

6.2.4 Verejná služba `service-public`

Verejná GraphQL služba (`service-public`) slúži ako hlavné aplikačné rozhranie pre headless redakčný systém Slackify. Tento GraphQL server umožňuje vývojárom využívajúcim Slackify vo svojich implementáciach jednoducho získať obsah z redakčného systému.

Autorizácia

Po prepojení tímu v službe Slack s redakčným systémom Slackify je tomuto tímu vygenerovaný *unikátny autorizačný token*. Tento token je následne očakávaný v HTTP hlavičke `Authorization` každej požiadavky smerujúcej na službu `service-public`. V prípade, že autorizačný token chýba alebo je nesprávny, služba požiadavku zahodí a odpovie chybovou správou vysvetlujúcou zlyhanie.

Zjednodušený typ `Component`

Databázový model `Component` slúži pre uchovávanie metadát o komponente. Samotný obsah komponentu sa nachádza v oddelených databázových modeloch s ktorými je daný komponent v relácií (viac viz 6.1.2). Takýto princíp ukladania dát umožňuje jednoduchú modifikáciu typov komponentov a ich obsahu, no nie je optimálny pre vývojárov, ktorí implementujú Slackify do svojich projektov.

```
query {
  component(...) {
    id
    type # ?? - Unknown type
    plainTextData
    articleData
    linkData
    ...
  }
}
```

Výpis 6.12: Príklad získania obsahu komponentu pred optimalizáciou.

Vývojár musí totiž vopred poznať typ komponentu a podľa toho si vyžiadať z GraphQL servera konkrétnu reláciu s obsahom komponentu. Väčši problém však nastane v prípade, že vývojár vopred nepozná typ komponentu, ktorý žiada od GraphQL servera. V takejto situácií je nutné vyžiadať relácie s obsahmi pre *všetky typy komponentov* a po obdržaní dát pristúpiť k správnemu obsahu podľa typu.

```
export const ComponentData = unionType({
  name: 'ComponentData',
  definition(t) {
    t.members('PlainTextComponentData', 'ArticleComponentData', 'LinkComponentData');
    t.resolveType((data) => {
      // ...
    });
  },
});
```

Výpis 6.13: Definícia union typu pre dáta komponentu.

Z tohto dôvodu dôvodu služba `service-public` disponuje upraveným typom `Component`, ktorý má špeciálne pole `data` typu `union` (viz výpis 6.13). Toto pole združuje relácie s obsahom pre všetky typy komponentov. Ak si vývojár vyžiada toto pole, získa obsah komponentu bez ohľadu na to, akého je typu.

Zber štatistík

Verejná GraphQL služba zaznamenáva jednoduché štatistiky prístupu k obsahu jednotlivých komponentov. Pokaždé, keď je nejaký komponent vyžiadaný z tohto rozhrania služba vytvorí nový záznam v databáze. Záznam sa skladá z času a ID komponentu a je vytvorený aj keď je komponent obsiahnutý ako súčasť požiadavky na kolekciu.

6.3 Webová konzola

Webová konzola (služba `frontend`) slúži pre komplexnejšiu správu obsahu v Slackify redakčnom systéme. Samotné uživateľské rozhranie je implementované pomocou JavaScriptovej knižnice React a frameworku Next.js, ktorý zabezpečuje vykresľovanie (render) na strane servera.

6.3.1 Pridanie Slackify do pracovného prostredia

Pred prvým použitím Slackify je nutné redakčný systém (konkrétne jeho Slack aplikáciu) pridať do pracovného prostredia pomocou OAuth 2.0 brány služby Slack.

Proces pridania Slackify do pracovného prostredia je následovný:

1. Užívateľ, ktorý chce pridať Slackify do svojho pracovného prostredia je po kliknutí na tlačidlo „Add to Slack“ presmerovaný na autorizačnú stránku Slacku.
2. Po úspešnej autorizácií aplikácie je užívateľ presmerovaný späť na stránku vo webovej konzole Slackify `/auth/add` s URL parametrom *code* (dočasný prístupový kód).
3. Slackify vymení obsah parametra *code* za informácie o danom pracovnom prostredí a užívateľovi, ktorí aplikáciu do pracovného prostredia pridal.
4. Informácie o pracovnom prostredí sú pridané do databázy spolu s užívateľom, ktorý má automaticky priradenú rolu *Owner*.

Slackify je možné pridať do pracovného prostredia aj v obchode s aplikáciami v Slacku.

6.3.2 Prihlásenie užívateľa

Slackify nemá vlastný proces vytvorenia užívateľského účtu, naopak všetky *dáta o užívateľoch preberá z služby Slack* pomocou OAuth 2.0 brány služby Slack.

Proces príhlásenia užívateľa do Slackify je následovný:

1. Užívateľ, ktorý sa chce prihlásiť do Slackify je po kliknutí na tlačidlo „Sign in with Slack“ presmerovaný na autorizačnú stránku služby Slack.
2. Užívateľ vyplní svoje prihlasovacie údaje pre účet v tom pracovnom prostredí, *ktoré má nainštalovanú aplikáciu Slackify*.
3. Po úspešnom prihlásení užívateľ *povolí Slackify prístup k užívateľským údajom* a je následne presmerovaný na stránku vo webovej konzole Slackify `/auth/redirect` s URL parametrom *code*.
4. Slackify odošle hodnotu URL parametra *code* na OAuth API služby Slack a očakáva odpoveď s *autorizačným kódom pre daného užívateľa* a informáciach o ňom.
5. Autorizačný token je následne možné využiť pre opätovné získanie informácií o užívateľovi alebo k vykonávaniu akcií za užívateľa.

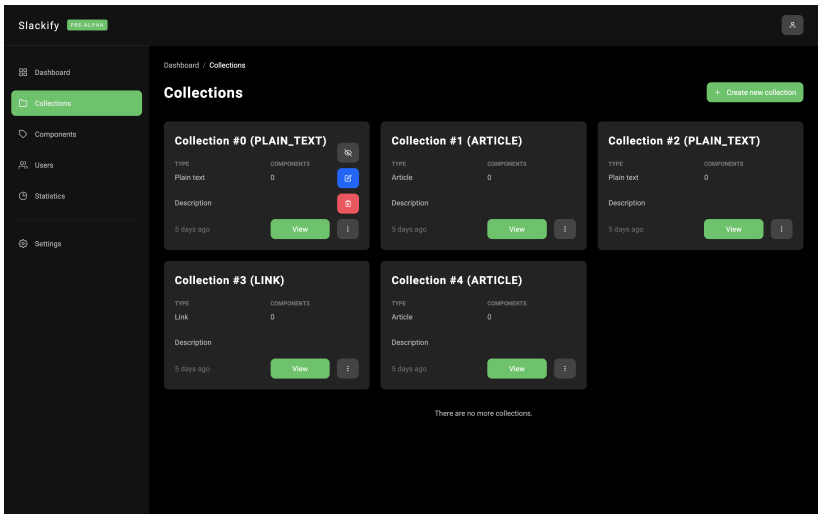
Po úspešnom obdržaní autorizačného tokena a užívateľských dát systém skontroluje, či už daný užívateľ v databáze existuje. V prípade, že užívateľ neexistuje, je záznam o ňom pridaný do databázy. V prípade, že už existuje, informácie o ňom sú len aktualizované.

Vygenerovaný JWT[2] so základnými informáciami o užívateľovi je uložený do cookies prehliadača. Pri návšteve každej stránky prebieha kontrola, či uložený JWT nie je expirovaný. Napríklad pravidlo

[2] JWT – JSON Web Token

6.3.3 Správa kolekcií

Hlavnou súčasťou správy kolekcií je zoznam všetkých kolekcií, ktoré boli vytvorené v tíme aktuálne prihláseného užívateľa. Na tejto stránke je možné kolekcie vytvoriť, upraviť alebo zmazať, ak má užívateľ na tieto akcie dostatočné práva.



Obr. 6.1: Stránka správy kolekcií vo webovej konzole.

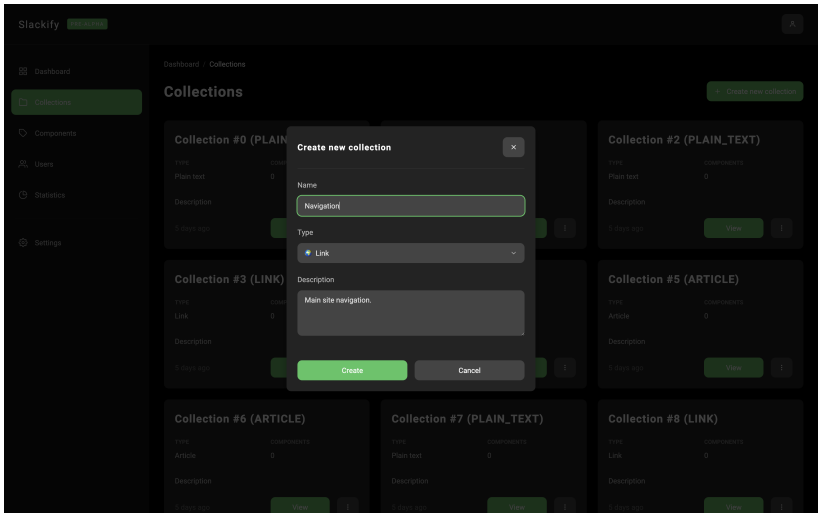
Pri úvodnom načítaní stránky je odoslaná GraphQL požiadavka (query) `collections()`, ktorá získa prvých 40 kolekcií zo služby `service-private`. Ak chce užívateľ zobraziť viac kolekcií, musí zísť na koniec zoznamu. Na jeho konci sa nachádza element, ktorý ak je viditeľný v okne prehliadača odošle znova požiadavku `collections()`, ale tentoraz s priloženými parametrami `skip` a `first`. Získané položky sú pridané na koniec zoznamu a takto je vytvorený mechanizmus tzv. „nekonečného scrollovania“ (infinite scrolling).

Vytvorenie a úprava kolekcie

Vytvorenie novej kolekcie alebo úprava už existujúcej je možná pomocou dialógového okna, ktoré je dostupné na stránkach zoznamu a detailu kolekcií. Objekt uložený v globálnom stave aplikácie pod názvom `createUpdateModal` uchováva kontext dialógového okna, ktorý sa obsahuje položky:

- `mode` – Mód zobrazenia, môže nadobudnúť hodnoty „create“ alebo „update“.
- `collection` – V prípade, že mód zobrazenia má hodnotu „update“, očakáva sa, že hodnota tejto položky obsahuje objekt popisujúci upravovanú kolekciu.

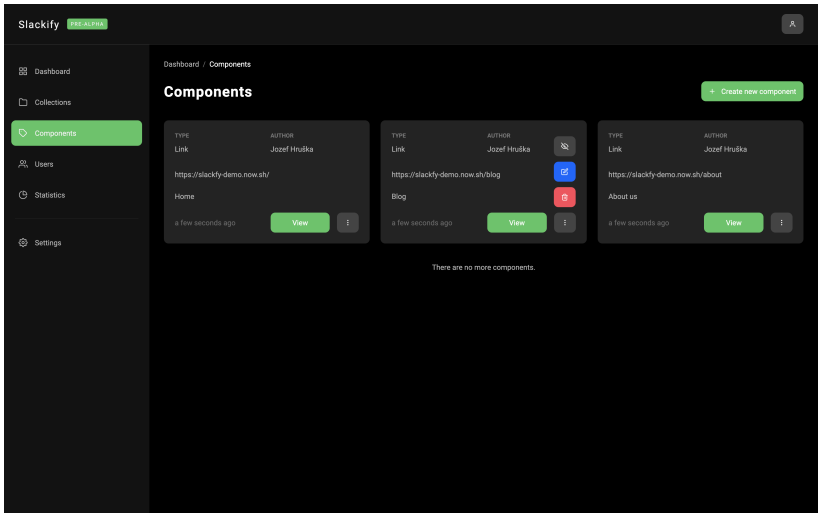
Po potvrdení formulára vytvorenia alebo úpravy kolekcie je na službu `service-private` odoslaná mutácia `createOneCollection()`, resp. `updateOneCollection()` s dátami z formulára. Ak bola mutácia úspešná, v odpovedi od služby `service-private` by sa mali nachádzať dáta novej, resp. aktualizovanej kolekcie. Tieto dáta sú následne použité pre aktualizovanie cache webovej aplikácie.



Obr. 6.2: Dialógové okno pre vytvorenie a úpravu kolekcií.

6.3.4 Správa komponentov

Podobne ako u kolekcií je hlavnou súčasťou správy komponentov zoznam všetkých komponentov, ktoré boli vytvorené v tíme aktuálne prihláseného užívateľa. Podmienkou vytvorenia nového komponentu je už *aspoň jedna existujúca kolekcia*, ku ktorej môže byť daný komponent priradený. Na tejto stránke je možné okrem vytvorenia nového komponentu aj upraviť alebo zmazať už existujúce komponenty.



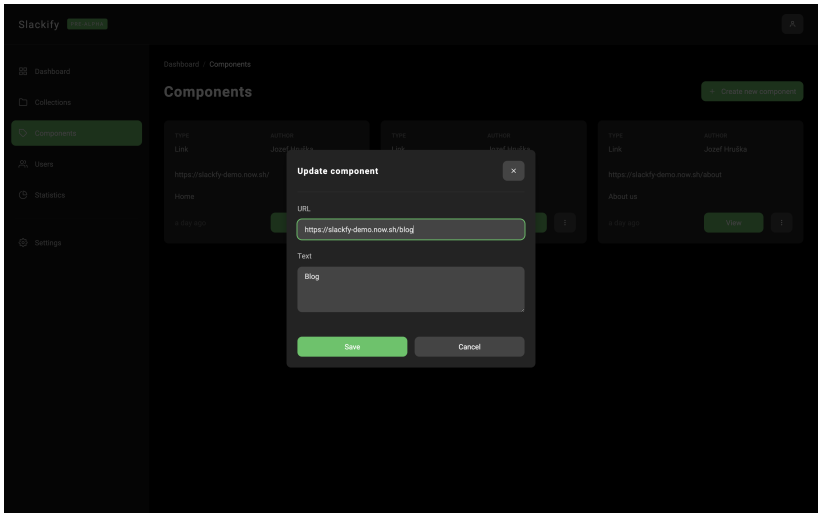
Obr. 6.3: Stránka správy komponentov vo webovej konzole.

Rovnako ako pri zozname kolekcií je pri úvodnom načítaní odoslaná požiadavka pre získanie z ktorých sa zoznam zostaví. Pre komponenty je táto požiadavka pomenovaná `components()`. Zoznam komponentov takisto podporuje aj infinite scrolling mechanizmus.

Vytvorenie a úprava komponentov

Rovnaký princíp ako pri tvorbe a úprave kolekcií. Jedná sa o dialógové okno, ktoré svoj kontext uchováva v globálnom stave aplikácie, avšak mierne pozmenený:

- `mode` – Mód zobrazenia, rovnako ako u kolekcií môže nadobudnúť hodnoty „create“ alebo „update“.
- `collection` – Obsahuje informácie o aktuálne zvolenej kolekcií pri tvorbe nového komponentu. Typ kolekcie určuje typ k *nej priradených komponentov*, a preto je táto kolekcia využitá pre zobrazenie relevantného formulára pre konkrétny typ komponentov.
- `component` – V prípade, že mód zobrazenia má hodnotu „update“, očakáva sa, že hodnota tejto položky obsahuje objekt poposijúci upravovaný komponent.



Obr. 6.4: Dialógové okno pre vytvorenie a úpravu komponentov.

Po povrdení formulára je pre vytvorenie nového komponentu alebo úpravu už existujúceho odoslaná mutácia `createOneComponent()`, resp. `updateOneComponent()` na službu `service-private`.

6.3.5 Zmena role užívateľa

Každý užívateľ môže navštíviť stránku so zoznamom všetkých užívateľov tímu, ktorého je členom. V prípade, že má aktuálne prihlásený užívateľ rolu „Owner“, môže navyše meniť role ostatných užívateľov v rámci tímu. U každého užívateľa v zozname sa nachádza HTML element `<Select />`, ktorý ukazuje aktuálnu rolu daného užívateľa. Pri zmene na inú hodnotu tento element odošle mutáciu `updateOneUser()` so zvolenou hodnotou na službu `service-private`, ktorá aktualizuje rolu daného užívateľa.

6.4 Slack aplikácia

Služba `service-slack` je Slack aplikácia umožňujúca *plnohodnotnú správu obsahu* v redakčnom systéme Slackify. Jedná sa o HTTP server implementovaný pomocu balíčka Bolt

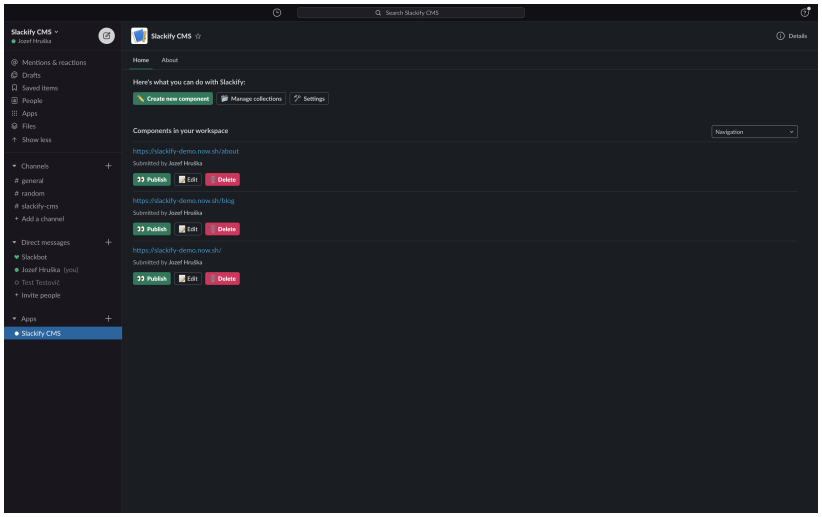
od spoločnosti Slack, ktorý reaguje na udalosti (events) a akcie (actions) prichádzajúce z verejnej API služby Slack.

6.4.1 Autorizácia tímov

Slack aplikácia sa musí pri každej akcií autorizovať špeciálnym tokenom „bot token“ unikátnym pre každý tím (pracovné prostredie) prepojený s aplikáciou. Na túto autorizáciu slúži metóda `authorize()`, ktorá vyhľadá „bot token“ konkrétneho tímu podľa jeho ID, ktoré je obsiahnuté v každej prichádzajúcej akcií alebo udalosti.

6.4.2 Domovská stránka

Každá Slack aplikácia má k dispozícií jednu stránku, ktorej obsah môže prispôsobiť danému užívateľovi. Táto stránka má názov „App Home“ a v Slackify slúži ako jediné miesto, kde môže užívateľ spravovať obsah redakčného systému. Pri každej návšteve tejto stránky užívateľom Slack API odošle udalosť `app_home_opened` na službu `slackify-slack`. Po obrdžaní udalosti služba vygeneruje rohranie pre daného užvateľa a odošle ho späť na Slack API, ktoré sa postará aby bolo správne zobrazené užívateľovi v aplikácií. Rozhranie domovskej stránky je *perzistentné*.



Obr. 6.5: Domovská stránka v Slack aplikácií.

Na domovskej stránke sa nachádza hlavička so všetkými akciami, ktoré môže užívateľ v redakčnom systéme vykonávať (ak užívateľ nemá dostatočné práva, akcie sú skryté). Pod touto hlavičkou sa nachádza zoznam komponentov. Zoznam zobrazuje komponenty priradené k zvolenej kolekcií pomocou elementu `Select` v záhlaví zoznamu. Ak nie je zvolená žiadna kolekcia, zobrazuje komponenty priradené k prvej dostupnej kolekcií.

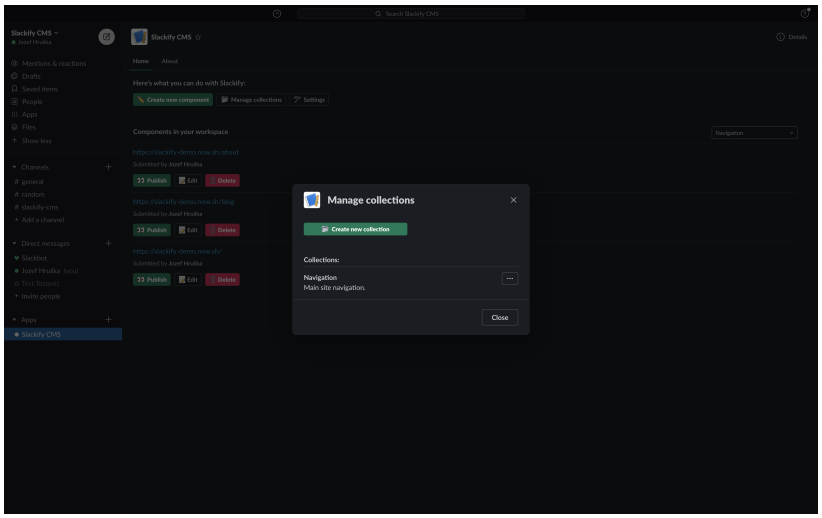
```
async function compose_app_home_view(
  teamId: string,
  userId: string,
  initialCollectionId?: string
): Promise<View | undefined>
```

Výpis 6.14: Funkcia zodpovedná za generovanie rozhrania domovskej stránky.

Domovská stránka je aktualizovaná po akciách užívateľa. Ak užívateľ vytvorí, odstráni alebo upraví kolekciu alebo komponent, rozhranie domovskej stránky musí byť aktualizované s novými zmenami.

6.4.3 Správa kolekcií a komponentov

Užívateľské rozhranie správy kolekcií a komponentov kopíruje webovú konzolu. Užívateľ má možnosť zobraziť, publikovať, skryť, upraviť a zmazať kolekcie alebo komponenty.



Obr. 6.6: Dialógové okno so zoznamom kolekcií v Slack aplikácií.

Zoznam kolekcií sa nachádza v dialógovom okne prístupnom z domovskej stránky. Dialógové okná v Slack aplikácií slúžia pre zobrazovanie informácií, ktoré sa nenachádzajú na domovskej stránke alebo pre získanie dát od užívateľa pomocou formulárov (napríklad vytvorenie novej kolekcie).

Kapitola 7

Testovanie

Podstatnou časťou tejto práce bolo testovanie systému. Tak ako end to end a jednotkové testy zabezpečujú správnu funkcionalitu implementácie, tak užívateľské testy pomáhajú odhalovať nedostatky v samotnom používaní systému.

Táto kapitola je venovaná užívateľskému testovaniu headless redakčného systému Slackify. Prvá sekcia 7.1 popisuje princíp a závery z testovania počas implementovania samotným vývojárom a druhá sekcia 7.2 sa venuje užívateľskému testovaniu osobami, ktoré sa na vývoji nepodielali.

7.1 Testovanie počas vývoja

Testovanie počas vývoja je prirodzeným procesom pri navrhovaní a implementovaní softvéru. Pri každom zásahu do návrhu alebo implementácie bola každá zmena užívateľsky testovaná vývojárom. Ak neboli splnené požiadavky na funkčnosť, postupnými iteráciami nad návrhom alebo implementáciou boli problémy identifikované a odstránené.

7.2 Užívateľské testovanie

Headless redakčné systémy vyžadujú pre ich používanie pokročilejšie technické znalosti. Z tohto dôvodu bola pre testovanie zvolená osoba pôsobiaca v oblasti IT a vývoja.

7.2.1 Testovacie scenáre

Testovacie scenáre boli zostavené z najbežnejších úkonov, ktoré by typický užívateľ v systéme vykonával. Testovanie prebiehalo za minimálneho zásahu osôb, ktoré neboli jeho súčasťou. Z dôvodu minimálnej užívateľskej dokumentácie však malé zásahy boli potrebné.

Úspech každého testovacieho scenáru bol vyhodnotený jednou z troch možností:

- `ÁNO` – Úspech
- `NIE` – Neúspech
- ∅ – Čiastočný úspech / Čiastočný neúspech

Webová konzola		
	Názov	Úspech
1.	**Pridanie Slackify do pracovného prostredia v Slacku** Poznámka: Chýbajúce zdôvodnenie požadovaných oprávnení.	ÁNO
2.	**Prihlásenie sa do užívateľského účtu**	ÁNO
3.	**Vytvorenie kolekcie** Poznámka: Chýbajúce označenie voliteľných polí.	ÁNO
4.	**Vytvorenie komponentu** Poznámka: Chýbajúca možnosť vytvoriť komponent priamo v detaile kolekcie	ÁNO
5.	**Odstránenie komponentu**	ÁNO
5.	**Odstránenie kolekcie**	ÁNO

Slack aplikácia		
	Názov	Úspech
1.	**Vytvorenie kolekcie** Poznámka: Odhalená chyba v zobrazení zoznamu komponentov typu `Article` s nevyplneným polom „lead“. Chýbajúce upozornenie pri duplicite názvov kolekcií.	ÁNO
2.	**Vytvorenie komponentu**	ÁNO
3.	**Odstránenie komponentu** Poznámka: Chýbajúce potvrdenie pri zmazaní komponentu.	ÁNO
4.	**Odstránenie kolekcie** Poznámka: Chýbajúce potvrdenie pri zmazaní komponentu.	ÁNO

Verejné GraphQL rozhranie		
	Názov	Úspech
1.	**Získanie všetkých kolekcií**	ÁNO
2.	**Získanie jednej kolekcie**	ÁNO
3.	**Získanie všetkých komponentov v kolekcií**	ÁNO
4.	**Získanie jedného komponentu**	ÁNO

7.2.2 Záver užívateľského testovania

Všetky scenáre užívateľského testovania boli *úspešné* aj vďaka silnému dôrazu na prívetivosť užívateľského rozhrania pri vývoji. Pri testovaní boli odhalené malé nedostatký návrhu alebo implementácie systému uvedené v poznámke pri danom testovacom scenári.

Kapitola 8

Záver

Cieľom práce bolo vytvoriť headless redakčný systém s možnosťou správy obsahu v službe Slack. Výsledné riešenie, redakčný systém Slackify, splňuje všetky body formálneho zadania s výnimkou využitia aplikácie Slack pre ukladanie dát. Už pri návrhu riešenia som odhalil niekoľko prekážok, pre ktoré som sa rozhodol, že obsah redakčného systému bude ukladaný v konvenčnej databáze.

Na začiatku práce som analyzoval existujúce headless redakčné systémy a nedostatky v ich nasadení, používaní alebo architektúre. Slackify adresuje a rieši tieto nedostatky. Umožňuje užívateľom spravovať a distribuovať obsah bez zbytočnej komplexity súčasných riešní, čo považujem za hlavný úspech tejto práce.

Vďaka tejto práci som sa naučil budovať robustné systémy a serverové aplikácie. Oblasť, v ktorej som nemal pred touto prácou žiadne skúsenosti a minimálne znalosti. Toto rozhodnutie spravilo prácu náročnejšou, ale o to viac odmeňujúcu.

Jedným z možných rozšírení, ktoré som sám niekoľkokrát zvažoval implementovať, je možnosť definovať vlastné typy komponentov. Toto rozšírenie by značne zvýšilo flexibilitu systému, no zároveň by si rozporovalo s hlavným cieľom tejto práce – jednoduchosťou použitia.

Literatúra

[1] Cambridge University. *Cambridge Dictionary* [online]. [cit. 2020-04-03]. Dostupné z: https://dictionary.cambridge.org/.

[2] Facebook Inc.. *React* [online]. [cit. 2020-04-05]. Dostupné z: https://reactjs.org/.

[3] Fielding, R., UC Irvine, Gettys, J., Compaq/W3C, J. Mogul et al. *Hypertext Transfer Protocol – HTTP/1.1* [Internet Requests for Comments]. RFC. 1997.

[4] Komunita Netlify CMS. *Dokumentácia Netlify CMS* [online]. [cit. 2020-04-03]. Dostupné z: https://strapi.io/documentation/.

[5] Komunita Strapi.io. *Dokumentácia Strapi.io* [online]. [cit. 2020-04-03]. Dostupné z: https://strapi.io/documentation/.

[6] Meteor Development Group Inc.. *Dokumentácia Apollo GraphQL* [online]. [cit. 2020-04-05]. Dostupné z: https://www.apollographql.com/docs/.

[7] Microsoft Corporation. *TypeScript Language* [online]. [cit. 2020-04-04]. Dostupné z: https://www.typescriptlang.org.

[8] Mozilla and individual contributors. *Mozilla MDN web docs* [online]. [cit. 2020-04-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/.

[9] npm, Inc.. *Node Package Manager* [online]. [cit. 2020-04-04]. Dostupné z: https://www.npmjs.com/.

[10] OpenJS Foundation a Joyent. *Node.js* [online]. [cit. 2020-04-04]. Dostupné z: https://nodejs.org/en/.

[11] Prisma. *Prisma* [online]. [cit. 2020-04-05]. Dostupné z: https://www.prisma.io/.

[12] Slack. *Slack API* [online]. [cit. 2020-04-05]. Dostupné z: https://api.slack.com/.

[13] The GraphQL Foundation. *GraphQL* [online]. [cit. 2020-04-05]. Dostupné z: https://graphql.org/.

[14] The PostgreSQL Global Development Group. *PostgreSQL* [online]. [cit. 2020-04-05]. Dostupné z: https://www.postgresql.org/.

[15] ZEIT, Inc.. *Next.js* [online]. [cit. 2020-04-05]. Dostupné z: https://nextjs.org/.