FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Systém pro správu obsahu s využitím Next.js
Student:	Matouš Škoda
Vedoucí:	Ing. Filip Glazar
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat systém pro správu obsahu. Systém bude nabízet základní funkce pro práci s obsahem jako je správa stránek pomocí WYSIWYG editoru, správa obrázků a řízení přístupu uživatelů. Tento systém bude možné použít pro snažší vývoj a tvorbu webových stránek. Pro implementaci využijte programovací jazyk TypeScript a reaktivní framework Next.js. Ideálně postupujte dle následujících kroků:

1) Proveďte analýzu existujících řešení
2) Specifikujte požadavky na systém
3) Navrhněte systém dle správných softwarově-inženýrských postupů. Zaměřte se na jednoduchou rozšiřitelnost a údržbu softwaru
4) Implementujte prototyp navrženého řešení
5) Vzhledem k plánovanému dalšímu rozvoji implementujte vhodné automatizované testy
6) Zhodnoťte výsledek dosavadní práce a navrhněte potřebná vylepšení, či další úpravy systému

---

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 22. února 2023 v Praze.

Bakalářská práce

SYSTÉM PRO SPRÁVU OBSAHU S VYUŽITÍM NEXT.JS

Matouš Škoda

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: Ing. Filip Glazar
11. května 2023

Odkaz na tuto práci: Škoda Matouš. *Systém pro správu obsahu s využitím Next.js*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především Ing. Filipu Glazarovi za vedení a konzultace při tvorbě této práce. Dále bych rád poděkoval rodině za podporu během psaní práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 11. května 2023 .....

Abstrakt

Bakalářská práce se zabývá analýzou, návrhem a implementací prototypu systému pro správu obsahu, založeného na technologiích React a Next.js. Cílem práce je navrhnout systém pro správu obsahu takový, který by co nejvíce zjednodušoval vývoj webových aplikací a prezentací, tedy se zaměřením na snadnou rozšiřitelnost, vysoku flexibilitu a snadnout automatizaci a nasazení. V teoretické části práce jsou popsány všechny technologie týkající se následné analýzy a implementace. V praktické části je nejdříve popsána analýza zadání a stávajících řešení, dále implementace automatizace, samotného prototypu a automatizovaného testování. Nakonec je provedena analýza prototypu a jsou navrženy jeho další rozšíření.

Klíčová slova systém pro správu obsahu, webová aplikace, front-end, React, Next.js

Abstract

The bachelor's thesis presents analysis, design and implementation of a prototype of a content management system using React and Next.js. The goal of this thesis is a system that simplifies development of web applications and presentations in the mentioned technologies, providing high flexibility and extensibility and that allows for straightforward automation and deployment. The theoretical chapter describes utilized technologies and their features, while the practical chapter presents analysis of available solutions, implementation of the prototype and it's automated testing. Lastly the thesis analyzes the implemented prototype and presents suggestions for the continuation of it's development.

Keywords content management system, web application, front-end, React, Next.js

Seznam zkratek

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CLI	Command Line Interface
UI	User Interface
CI/CD	Continuous Integration and Continuous Delivery
XML	Extensible Markup Language
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
SVG	Scalable Vector Graphics
DOM	Document Object Model
JSON	JavaScript Object Notation
SSG	Server Side Generation
SSR	Server Side Rendering
IDE	Integrated Development Environment
PAAS	Platform As A Service
SAAS	Software As A Service
ORM	Object-Relational Mapping
CRUD	Create, Read, Update, Delete

Kapitola 1
Úvod

Ve vývoji webových prezentací a stránek se postupem času zvyšují nároky zákazníků a koncových uživatelů na poskytovanou funkcionalitu aplikací. Zejména díky vzestupu systémů pro správu obsahu, zaměřených na koncové uživatele, jako je například Wordpress a Drupal, je už teď standardem poskytovat administrační rozhraní pro tvorbu a úpravy nejen dynamického (databázového) obsahu stránek, ale i toho statického.

Nejpoužívanější systémy pro správu obsahu (dále jen CMS) jsou ale z drtivé většiny postaveny na frameworcích napsaných v jazyce PHP. Zároveň jsou částečně nebo téměř zcela (v případě WordPress, který byl původně určen zejména pro tvorbu blogů) cíleny na koncové uživatele a laiky, což může zesložiťovat vývoj (např. ztížené možnosti verzování a automatizace nasazení).

Pokud chceme z jakéhokoliv důvodu používat jinou technologii – v případě této práce jde o jazyk TypeScript a reaktivní framework Next.js – často zjistíme, že pro ni neexistuje ekvivalent zmíněných CMS, respektive jde často o online platformy, nebo tzv. *headless* CMS, které znamenají obtížnější nasazení (nasazujeme více aplikací, ne jeden monolit) a složitější úpravy. Při vývoji malých aplikací a prezentací tak tato řešení, případně tvorba celého nového administračního rozhraní, často nedává finančně smysl kvůli časové náročnosti vývoje.

Technologie pro implementaci CMS byly zvoleny zejména z následujících důvodů, týkajících se (termíny popsány dále v teoretické části práce):

- Jazyk TypeScript poskytuje statickou kontrolu typů a propojení s editorem, čímž se výrazně zlepšuje tzv. *developer experience* a snižuje se množství chyb při vývoji. Zároveň je možné zpřístupňovat vývojářům rozhraní, které je díky typům *samopopisné*, čímž se dále omezují nároky na vývoj.
- Knihovna React umožňuje díky vývoji pomocí komponent a reaktivitě snáze vyvíjet plnohodnotné aplikace, nejen klasické webové stránky.
- Framework Next.js řeší problémy knihovny React (například *SEO* stránek) a poskytuje řadu funkcionalit jako je *routing* a řada optimalizací a dává celé aplikaci strukturu.

Díky zvoleným technologiím by vyvíjené CMS mělo výrazně snižovat nároky na vývoj, automatizaci a nasazení projektů. Zároveň by mělo být snadno upravitelné a rozšiřitelné tak, aby umožňovalo i vývoj větších projektů, prvotním zaměřením jsou ale menší webové aplikace a prezentace.

Kapitola 2
Cíl práce

Hlavním cílem bakalářské práce je návrh a implementace systému pro správu obsahu, určeného zejména pro vývojáře, v jazyce TypeScript s využitím frameworku Next.js. Tento systém by měl zjednodušovat a urychlovat tvorbu webových stránek a menších projektů tím, že bude poskytovat funkcionalitu pro správu obsahu stránek, konfiguraci aplikace, zjednodušení napojení na databázi a další.

Dílčími cíli práce jsou:

- Analýza existujících řešení
- Specifikace detailních požadavků na implementovaný systém
- Vytvoření návrhu systému se zaměřením na rozšiřitelnost
- Implementace prototypu návrhu
- Implementace automatizovaných testů
- Zhodnocení implementovaného prototypu a návrh potřebných rozšíření a úprav systému

Implementovaný systém bude v budoucnu upraven a rozšířen dle výsledného zhodnocení tak, aby ho bylo možné využívat pro vývoj a správu webových aplikací a prezentací.

Kapitola 3
Teoretická část

3.1 Základní frontend technologie, React a Next.js

V této kapitole popisuji nejprve základní webové technologie a pojmy s nimi spojené a dále rozebírám hlavní knihovny využité při implementaci systému.

3.1.1 HTML, CSS a JavaScript

HTML, CSS a JavaScript jsou základními technologiemi všech webových stránek a aplikací.

HTML

HTML je *značkovací* jazyk, sloužící pro popis obsahu a struktury dokumentů. *Značkovací* jazyky, jako XML a HTML, používají pro popsání obsahu značky (v angličtině „tag“), které dokumentům poskytují sémantický význam. Cílem je pak výsledný dokument, který by v ideálním případě měl být významově pochopitelný nejen pro stroj, ale i pro člověka. Dalším základním prvkem jazyka HTML jsou *hypertextové* odkazy, umožňující přecházení mezi částmi dokumentu, nebo do jiných dokumentů, z tohoto vychází zkratka HTML – HyperText Markup Language. [1]

Bloky obsahu jsou v HTML dokumentech vloženy do tzv. *elementů*, sestávajících z počáteční značky, obsahu a ukončující značky (v případě elementů obsahujících děti, například `<p>Odstavec</p>`), nebo se může jednat o samoukončující značky, v případě elementů bez dětí (např. formulářové prvky typu `<input type="text"/>`). Značky se dělí na *sémantické*, které obalením obsahu přidávají specifický význam (např. `<p></p>`, znamenající odstavec, z angličtiny „paragraph“), nebo na značky bez sémantického významu (např. `<span></span>`), které obsah pouze shlukují do bloků nebo řádek. [1, 2]

Pokud samotný HTML dokument otevřeme například ve webovém prohlížeči, je na základě použitých sémantických značek schopný dokument interpretovat, přiřadit elementům formátování a vykreslit jej tak, aby byl prezentovatelný koncovému uživateli. Zároveň sémantické značky umožňují interpretaci obsahu asistivním technologiím, jako jsou například čtečky obrazovek pro uživatele se zrakovým postižením, a mimo jiné také nástrojům vyhledávačů (v angličtině *Web crawler*), které umožňují automaticky indexovat a hodnotit obsah webových stránek. [2]

Dalším důležitým prvkem HTML dokumentů jsou *atributy* elementů. Ty dále buď rozšiřují sémantický význam elementů, mění jejich zpracování prohlížečem, nebo i umožňují další změny chování určené vývojářem. Například atributy `id` a `class` přidávají elementu identifikátor a třídu a umožňují programátorovi odkazovat se na tento obsah. Element `<h1 id="nadpis-1">Nadpis</h1>` tak může následně sloužit jako cíl hypertextového odkazu (`<a href="#nadpis-1">Odkaz</a>`). [1]

Elementy v dokumentu jsou strukturovány do stromové struktury, tedy tak, že kořenem dokumentu je povinně element `<html>...</html>` a následně každý element musí mít právě jednoho rodiče a libovolné množství dětí. Od toho se dále odvíjí práce s elementy v jazycích CSS a JavaScript (DOM). Základní chování a vykreslení HTML dokumentů v prohlížečích je pro moderní webové prezentace a aplikace nedostačující a je dále nutné využít zmíněných technologií CSS a JavaScript. [1]

CSS

CSS, anglicky zkratka pro „Cascading Style Sheets“, je jazyk určený k popisu vzhledu obsahu HTML (ale i například XML a SVG) dokumentů. Na CSS soubory se buď odkazujeme v HTML dokumentu pomocí elementu `link` v hlavičce, nebo je CSS kód přímo vložen v atributu `style`, nebo jako obsah stejně pojmenovaného elementu. [3]

CSS popisuje dokument pomocí **`selektorů`**, které vybírají elementy z dokumentu na základě značek, atributů, nebo struktury (předci, sourozenci, děti ve stromové struktuře elementů). Následně je těmto elementům přiřazeno pravidlo, podle kterého se změní jejich vykreslení, nebo chování. Mění se tak ale i chování pro děti těchto elementů. Například následující kód upraví nadpis (tag `h1`) s identifikátorem `heading-1` a třídou `blue` tak, že se jeho text vykreslí jako modrý. Pokud by šlo o element s dětmi, pak by se modře vykreslilo i jejich písmo, vzhled a chování v CSS se dědí. [3]

Výpis kódu 3.1 Ukázka CSS kódu

```
1   h1#heading-1.blue {
2       color: blue;
3   }
```

HTML elementy nicméně mohou být vybrány a upraveny (například změna barvy) více selektory zároveň. Z toho důvodu je v CSS zavedena *specificita* selektorů. Selektor pro ID (`#`) tak má větší „váhu“ než selektor pro třídu (`.`) a je ve výsledku použit ten s vyšší specificitou. V případě, že by více pravidel mělo specificitu stejnou, použije se poslední, což může způsobovat řadu špatně odhalitelných chyb (kvůli pořadí vložených CSS souborů), zejména když nejsou vkládány ručně, ale nástrojem. [4]

Ručně psaný CSS kód dnes už není pro větší aplikace běžný, zejména kvůli náročnější udržitelnosti, a rozšířilo se využití knihoven a *preprocesorů*. Přístupů je více, nicméně CSS knihovny typicky poskytují připravená pravidla pro základní ale i komplikovanější chování elementů (například TailwindCSS a Bootstrap). Preprocesory jsou pak většinou nadstavbou jazyka CSS (například Sass), poskytující novou syntaxi a funkcionalitu. [5]

JavaScript

JavaScript je dynamický skriptovací jazyk primárně určený pro spuštění v prohlížeči, kde umožňuje rozšíření interaktivity HTML dokumentů. Díky vzniku serverového runtime Node.js se ale rozšířil i do dalších využití. V prohlížečích je udržován dle standardu EcmaScript. [6]

Ve webových prohlížečích JavaScript poskytuje možnosti upravovat interní reprezentaci HTML dokumentů pomocí *DOM* (z anglického Document Object Model). V kombinaci s API poskytnutým prohlížečem je tak možné dynamicky měnit obsah na základě interakce uživatele, nebo na základě komunikace se serverem. [6]

Jedním ze zásadních omezení JavaScriptu je, že jde o dynamický jazyk, který je slabě typovaný, což zejména u větších projektů klade vyšší nároky na code review, kvalitu kódu a testování, zároveň se zhoršuje udržitelnost kódu. Z těchto důvodů vznikl jazyk TypeScript. [7]

3.1.2 TypeScript

TypeScript je kompilovanou nadstavbou jazyka JavaScript, která ho rozšiřuje o statické typování a další syntaxi. To je zároveň možné i u projektů napsaných v samotném JavaScriptu, díky zabudování TypeScript serveru v editoru. Samotný kód v TypeScriptu se pak pomocí nástroje transformuje do JavaScriptu, který je pak spustitelný všude, kde bychom normálně spouštěli JavaScript samotný. [7]

Typový systém TypeScriptu je sám o sobě velmi flexibilní a díky podpoře generických typů (obdobně jako `templates` u jazyka C++) a *type inference* (do češtiny by šlo přeložit jako „dedukce“ typů), poskytuje výborné prostředky pro zlepšení API knihoven pro uživatele.

Například při použití knihovny `react-hook-form` pomocí typu definuji „tvar“ formuláře a díky tomu získám v editoru nápovědy a kontrolu, jestli vstupní pole opravdu získávají správná data. Při následném zpracování dat si mohu být jistý, že získaná data budou ve tvaru, který jsem definoval. Dále rozvedeno v implementační části práce.

3.1.3 React

React je deklarativní JavaScriptovou knihovnou pro tvorbu uživatelských rozhraní, založenou na *komponentách*. Části aplikací jsou děleny do většího množství komponent, které se *komponují* do stromové struktury. React se zabývá pouze vývojem uživatelských rozhraní a vykreslování obsahu do DOM – neobsahuje žádnou další funkcionalitu, jako například UI frameworky Vue a Ember, je tedy navíc nutné využít další knihovny a nástroje. [8]

K zápisu vykreslovaného obsahu se nepoužívají HTML dokumenty, ale technologie *JSX* (JavaScript XML), která umožňuje psát kód obdobný HTML uvnitř JavaScript souborů (většinou s koncovkou `.jsx`, případně `.tsx` pokud je použit TypeScript), JSX se ale v mnohém liší. Syntax je upravena tak, aby žádným způsobem nekolidovala s existující syntaxí JavaScriptu, například atributy jako `class` a `for` musely být přejmenovány na `className` a `htmlFor`. React následně při vykreslování aplikace převede JSX na tvorbu elementů přes API prohlížeče. [9]

Způsoby zápisu komponent Reactu jsou dva, a to pomocí tříd nebo funkcí (*třídní a funkcionální komponenty*). S vývojem knihovny se ale v posledních letech ustálil jako preferovaný zápis funkcionálních komponent, na třídní komponenty narazíme spíše u starších projektů.

K použití funkcionálních komponent se váže koncept `hooks`, speciálních funkcí, které umožňují komponentám držet vnitřní stav a další možnosti. Interní stav aplikace se mezi komponentami předává pomocí atributů v JSX (*props*), nebo pomocí `kontextu` přístupného přes hook. Kontext stav zpřístupňuje všem potomkům ve stromu komponent, bez nutnosti předávat pomocí atributů ve všech úrovních podstromu. [10]

Výpis kódu 3.2 Ukázka komponenty napsané v Reactu

```
01  const Counter = () => {
02      const [count, setCount] = useState(0)
03      useEffect(() => {
04          console.log("Count is: ", count)
05      }, [count])
06      return (
07          <div>
08              <p>You clicked {count} times</p>
09              <button onClick={() => setCount(count + 1)}>
10                  Click me
11              </button>
12          </div>
13      ) }
```

Komponenta `Counter` v ukázce kódu vykreslí HTML tlačítko s popiskem. Ve funkcionálních komponentách se pro vykreslení používá návratová hodnota. Ta musí být validním JSX, tedy

pokud nechceme vykreslit nic, musíme vrátit například `null`. V JSX můžeme znovu vnořit JavaScript pomocí složených závorek, v příkladu na řádcích 5 a 6 pro výpis hodnoty a předání funkce.

Pro interní stav (počet stisknutí tlačítka) je na řádce 2 použit hook `useState`, přičemž `count` je aktuální stav a `setCount` je funkce pro jeho změnu. Pokud dojde k volání `setCount`, v tomto případě při kliknutí na tlačítko, dojde na překreslení komponenty, tedy celá funkce komponenty se znovu vykoná včetně překreslení celého podstromu. Na řádce 3-5 je další příklad požití hooku `useEffect`, který umožňuje provést vloženou funkci na základě změn závislostí v poli v druhém argumentu.

React při změnách stromu komponent na pozadí provádí proces v angličtině pojmenovaný *reconciliation*. V paměti drží datovou strukturu reprezentující aktuální vykreslený strom komponent, tzv. *Virtuální DOM* (VDOM), na základě změn komponent hledá rozdíly ve VDOM a následně do DOM prohlížeče propisuje pouze nutné změny, kvůli efektivitě. Některé jiné UI knihovny, například *SolidJS a Svelte*, jsou díky kompilaci a přímější integraci s DOM prohlížečů schopné VDOM vypustit a získat tak výrazně lepší efektivitu v některých ohledech (typicky při vykreslení velkého množství prvků), blížící se pak použití samotného JavaScriptu. [11, 12]

3.1.4 Next.js

Jak jsem zmínil v předchozí části o knihovně React, sama o sobě kromě tvorby UI neposkytuje nic dalšího, jako je například *routing* a další nástroje pro vývoj. Z tohoto důvodu vznikly frameworky, jakým je například Next.js.

Next.js poskytuje celou řadu funkcí a nástrojů, které je jinak při použití samotného Reactu nutné konfigurovat ručně. Například jde o zmíněný routing, streaming dat, optimalizace obrázků, fontů a skriptů, podporu CSS knihoven a preprocesorů, možnosti vytváření API endpointů a zejména technologie *SSG a SSR* (z angličtiny Static Site Generation a Server Side Rendering). [13]

Poslední dvě zmíněné technologie umožňují obejít jeden z nedostatků aplikací psaných v Reactu, kterým je zhoršená schopnost robotů indexovat jejich obsah. Toho se dosahuje tím, že se při sestavení aplikace (SSG) nebo při každém requestu (SSR) vygeneruje statický balíček (HTML, CSS, JS), který se odešle klientovi a po spuštění v prohlížeči se provede proces nazvaný *hydration*, který znovu obnoví interaktivitu aplikace. Díky tomu je prvotní odpovědí na požadavek statické HTML, které je možné indexovat. [14]

Next.js také aplikaci udává strukturu. Ve verzích Next.js do verze 13 se převážně používá koncept stránek. Adresář `pages/` slouží jako kořen stránek a jeho adresářová struktura a komponenty v něm určují výsledné stránky aplikace (a případně API endpointy). [15]

```
pages/ ..... kořen aplikace
├── index.tsx ..... kořenová stránka
├── articles/
│   └── [slug].tsx ..... stránky s výsledným URL /articles/<slug>
└── api/
    └── create-user.tsx ..... příklad API endpointu s URL /api/create-user
```

Zároveň je k možné použít komponentu v souboru `_app.tsx`, která automaticky obaluje každou komponentu stránky, tedy není nutné ji ručně přidávat do stránek. Tato obalová komponenta ale může existovat pouze jedna, nejde tedy například mít různé `_app` komponenty pro klientskou a administrační část aplikace. Toto je jeden z nedostatků, který řeší betaverze adresáře `app/` v Next.js od verze 13. [16]

Next.js 13 a adresář `app/`

Betaverze adresáře `app/` zpřístupňuje v Next.js kompletně novou funkcionalitu a nový přístup k tvorbě aplikací. Obdobně jako u adresáře `pages/` adresářem `app/` udáváme strukturu výsledné aplikace, včetně API endpointů. Tam, kde se ale předchozí koncept stránek omezoval na pouze jediný sdílený obalový blok a komponenty pro samotné stránky, je nový přístup výrazně širší. [16]

Základem aplikace je, stejně jako předtím adresářová struktura, komponenty v ní jsou ale pojmenovány jinak. Stavebními bloky jsou nově komponenty v souborech se speciálními názvy: [16]

- `page.tsx` nebo `route.tsx` – Veřejně přístupná komponenta stránky nebo API endpointu
- `layout.tsx` nebo `template.tsx` – Komponenta, kterou se automaticky obalí všechny další komponenty ve stejném adresáři, nebo níže v podstromu komponent. Nahrazuje původní `_app.tsx` s tím rozdílem, že nijak neomezuje počet vnoření
- `loading.tsx` – UI použité, pokud je aktivní *React Suspense*. Může být použito při načítání celých „stránek“ aplikace, ale i pro načítání pouze částí UI
- `error.tsx` – UI použité v případě, že se v podstromu komponent neodchytí JS *Error*
- `not-found.tsx` – UI zobrazené, pokud cesta není nalezena

Celý koncept je založen na principu *klientských* a *serverových* komponent. Klientské komponenty jsou klasické React komponenty, využívající *prerendering* a *hydration* na serveru, stejným způsobem jako předtím celé stránky používající SSR, s tím rozdílem, že je proces prováděn na bázi jednotlivých komponent, ne celých stránek. Tyto komponenty umožňují normální uživatelskou interakci a rozlišíme je podle direktivy `"use client"` na začátku souboru. [17]

Serverové komponenty jsou všechny ostatní komponenty použité v adresáři `app/`, nebo i takové, které jsou importované do komponent tam umístěných. Tyto komponenty se od klientských zásadně liší tím, že jsou `vyhodnoceny` na serveru (respektive v Next.js runtime). S tím souvisí i nutná omezení na jejich kód. Například nelze v nich udržovat vnitřní stav, používat jakékoliv React hooks nebo zpracovávat interakce uživatelem. [17]

Kód v serverových komponentách je prováděn čistě v chráněném prostředí a nikdy se nedostane ke klientovi. Tedy v nich můžeme například bezpečně získávat data z databáze, bez toho, aniž by se přístupové údaje mohly propsat do prohlížeče. Ve výsledku je kód serverových komponent vyhodnocen, UI staticky vygenerováno do HTML obdobou SSG a odesláno na klienta, bez nutnosti přidání JavaScriptu, čímž se výrazně snižuje množství odesílaných dat. [17]

Tato technologie umožňuje Next.js runtime využívat kešování jednotlivých komponent a zároveň zpřístupňuje *streaming* komponent. Data aplikace se uživateli odesílají postupně po jednotlivých komponentách a dále se zmenšuje potřebný balíček JavaScriptu. Nezměněné UI se nijak nepřekresluje, vnitřní stav komponent se zachovává a aplikace zůstává pro uživatele načítání dat interaktivní. [17]

3.1.5 Material UI

Material UI je knihovna vytvořená společností Google, poskytující připravené komponenty, za použití *Material Design*. Knihovna je dostupná nejen pro React, ale i další UI frameworky, jako Angular, a je využívána řadou projektů společnosti Google.

Jednou z hlavních výhod použití této knihovny je, že díky poskytnuté komplexní funkcionalitě komponent výrazně zrychluje vývoj aplikací na ní postavených. Všechny komponenty mají pro uživatele konzistentní chování, vzhled a je možné je různě upravovat bez rozbití chování nebo responzivity. [18]

Vzhled komponent je možné upravit pomocí konfiguračního objektu `theme`, kterým se mění vizuální vlastnosti všech komponent zároveň. V případě potřeby upravení pouze konkrétní komponenty se používá atribut `sx`, kterým se upraví konkrétní položka CSS (například `fontSize`

pro velikost textu), s tím, že se typicky mění na základě daného `theme` (např. u velikosti textu se použije násobek původní velikosti písma). [19, 20]

Projekt je rozdělen do několika balkíků které poskytují různé části Material UI. Knihovna `@mui/material` obsahuje všechny komponenty Material UI, v knihovně `@mui/base` najdeme tytéž komponenty, ale bez implementace Material Design, tedy slouží zejména pro tvorbu nadstavby nad základními komponenty. Další z podstatných balíků je MUI X – sada knihoven obsahujících komplexnější komponenty. Příkladem je knihovna `@mui/x-data-grid` s komponenty tabulek se složitějším chováním, jako je filtrování, řazení, stránkování, apod. [18, 21]

Material UI je ideální pro rychlé vytváření prototypů, zejména aplikací, kde se dá očekávat velký výskyt interaktivních prvků, jako jsou tlačítka, formulářové pole a tabulky. Zároveň není problém komponenty upravovat a měnit jejich vzhled a chování.

3.2 Nástroje pro vývoj a testování

Vývoj aplikací zdaleka nekončí použitím daných technologií a naprogramováním aplikace samotné. V této kapitole popisuji nástroje, které se využívají k usnadnění naprogramování aplikace, jejímu sestavení a automatickému testování.

3.2.1 Npm

Npm (z anglického názvu *Node Package Manager*, psáno malými písmeny) je *package manager* (česky správce balíčků) pro JavaScript runtime Node.js, zároveň jde o největší repozitář JavaScriptových balíčků a knihoven. Registr umožňuje stahovat a sdílet software nejen veřejně, ale i v rámci soukromých organizací. [22]

Npm pro příkazovou řádku je automaticky přiložen k instalaci Node.js a poskytuje možnosti konfigurace projektu a jeho závislostí, spouštění nástrojů na základě skriptů a sestavování a publikování balíčků do repozitáře npm, nebo do libovolného jiného repozitáře. Na pozadí se vždy používá runtime Node.js, tedy jeho instalace musí být vždy přítomná v prostředí, kde chceme spouštět skripty a sestavení (například v rámci CI/CD). Alternativami k nástroji npm jsou například Yarn a pnpm, které se liší například v přístupu k ukládání závislostí. [23]

Konfigurace nástroje se provádí pomocí souboru `package.json` ve formátu JSON. Ten obsahuje nastavení jako název aplikace, autora a její verzi, hlavními položkami jsou ale položky pro skripty (`scripts`) a pro závislosti (`depencencies` a `devDependecies`). Níže je příklad skriptu pro spuštění vývojového serveru Next.js. [24]

Výpis kódu 3.3 Příklad skriptu npm

```
01  ...
02  "dev": "env TAILWIND_MODE=watch NODE_ENV=development next dev",
03  ...
```

Skripty mají název (v ukázce `dev`) a k němu přiřazený skript, který se vykoná v příkazové řádce při jeho zavolání. Typicky jsou skripty napsané v jazyce *Bash* a bývají buď velmi krátké a jednoduché, nebo se může jednat o spuštění jiného skriptu v souboru, zejména pokud potřebujeme zpracovávat argumenty. Pro předání argumentů skriptu je nutné je ve volání oddělit od argumentů nástroje npm, a to dvojitým minus (`--`). Například takto: [25]

```
$   npm run dev -- --argument
```

Npm má řadu zabudovaných skriptů a dále mají určité názvy skriptů specifický význam. Příkladem jsou libovolné skripty s předponou `pre` nebo `post`, které se vždy automaticky zavolají před, respektive po, vykonání skriptu s názvem bez předpony. Těmto skriptům se říká skripty životního cyklu a zpravidla se používají pro vykonání něčeho před nebo po instalaci nebo publikování projektu. [25]

Pro instalaci závislostí a knihoven je určen příkaz `npm install`, případně s přepínačem `--save-dev` pro vývojové závislosti (`devDependencies`). Do `package.json` se následně přidá název a verze knihovny instalované, využívající *sémantické verzování*. Zároveň je vytvořen soubor `package-lock.json`, který obsahuje strom nainstalovaných závislostí s pevně zafixovanými verzemi z důvodu opakovatelnosti instalace. [26]

Instalace všech závislostí se pak provádí dvěma příkazy, a to `npm install` bez argumentů, který se ale vždy pokusí o povýšení závislostí, nebo pomocí `npm ci`, který provádí „čistou“ instalaci s využitím `package-lock.json` a měl by tak být preferován téměř ve všech případech. [26, 27]

Za zmínku ještě stojí pomocné skripty `npm outdated` a `npm audit`. První z nich zkontroluje verze všech závislostí a vypíše ty, u kterých existují nové verze. Druhý skript projde databázi bezpečnostních zranitelností a vypíše hlášky, pokud jsou v některých nainstalovaných verzích balíčků bezpečnostní chyby. [28, 29]

3.2.2 Nástroje ESLint, Prettier, Husky

Při vývoji softwarových projektů, na kterých se podílí více vývojářů, je nutné udržovat kód konzistentní (v angličtině *code style*), zejména pokud jde o *open-source* projekty. Výhodou je pak snazší čitelnost a orientace v kódu pro ostatní vývojáře, kteří daný kód sami nepsali, nebo se na projektu nepodílí dlouhodobě. Snižuje se zároveň časová náročnost při hledání chyb a *code review*. Dodržování code style je pak jednodušší, pokud je automatizováno nástroji, jako jsou například ESLint a Prettier. [30]

ESLint

ESLint je takzvaný *linter*, tedy nástroj, který poskytuje chybové hlášky (případně rovnou i možnosti oprav) v případě, že napsaný kód sice může být validní z pohledu kompilátoru, ale z určitého důvodu neodpovídá správným zásadám pro kód v daném jazyce. [31]

V případě jazyka TypeScript může jít například o využití typu `any`. Tomuto typu (a dalším podobným) se v angličtině říká tzv. *escape hatch* (což by šlo přeložit jako „zadní vrátka“) a jeho používání bychom se pokud možno měli vyvarovat. Je to z toho důvodu, že tento typ v zásadě vypíná kontrolu typů pro danou proměnnou a má tendenci se v aplikaci postupně šířit. [32]

Protože je v ukázce níže proměnná `obj` typu `any` (a tím jsou i všechny položky na ní `any`), je z hlediska kompilátoru TypeScript v pořádku, že se snažíme přistoupit k jeho položce `prop`, ačkoliv neexistuje a měla by být typu `undefined`. Zároveň se odhadnutý typ proměnné `property` vyhodnotí jako `any`.

Výpis kódu 3.4 Příklad šíření typu `any`

```
01  const obj: any = {}
02  let property = obj.prop
```

Předchozí příklad není nahodilý. K obdobnému dojde, pokud se snažíme transformovat serializovaný *JSON* (anglicky „JavaScript Object Notation“) z textového řetězce na objekt pomocí funkce `JSON.parse()`. Výsledný objekt je pak typu `any` a při naivním přístupu dojde k šíření typu v projektu. Vyvarovat se těmto problémům se můžeme používáním typu `unknown`, který nás například nutí zkontrolovat existenci položky na objektu před přístupem k ní. [32]

Prettier

Prettier je nástroj, který nám pomáhá v projektech udržovat konzistentní styl kódu. Toho dosahuje tím, že automaticky transformuje specifikované soubory tak, aby odpovídaly nadefinovanému stylu. Například může jít o používání tabulátorů nebo mezer, nebo konzistentní pojmenování proměnných a funkcí (*camel case* nebo *snake case*, atp.). [33, 34]

Většinou bývá Prettier používán tak, že má vývojář nástroj nebo alespoň podporu pro něj zabudovánu v editoru nebo IDE. Typicky se nástroj spouští při každém uložení souboru, může se ale například také pomocí pluginu integrovat do ESLint, který potom ukazuje chyby formátování.

Husky

Ideálně bychom chtěli použití vývojových nástrojů automatizovat. U softwarových projektů se pro verzování a sdílení kódu mezi vývojáři typicky používá verzovací systém (nejrozšířenějším je *Git*). Git poskytuje tzv. *Git hooks*, které umožňují automatické spouštění skriptů například při vytváření změn (`precommit`, `postcommit`), odesílání změn na server (`prepush`, `precommit`) a další. [35]

Skripty pro Git hooks jsou ale lokální, takže se musí mezi vývojáři ručně sdílet a přidávat. Tento problém u JavaScriptových projektů řeší nástroj Husky, který nastavené Git hooks do lokálního repozitáře automaticky přidává při instalaci projektu (`npm ci`, `npm install`). Skripty

pak využívají nástroje jako ESLint, TypeScript kompilátor a ostatní npm skripty k automatizaci všech kontrol. [36]

Tímto způsobem se pak výrazně omezuje šance, že se do sdíleného Git repozitáře propíší případné chyby. V případě, že potřebujeme v Git provést nahrání změn bez toho, aniž by nás omezovaly kontroly (například u velmi drobných změn jako jsou překlepy v textu), je možné použít přepínač `--no-verify`.

3.2.3 Testování frontend aplikací

K vývoji netriviálních softwarových projektů neoddělitelně patří jejich testování. Testování aplikací je možné rozdělit na tři nejpoužívanější přístupy, a to na *unit* testy (překládáno jako jednotkové testy), *integrační* testování a *end-to-end* testování (překlad nenalezen). [37]

Unit testy ověřují funkčnost „jednotek“ kódu, zpravidla tedy samotných funkcí nebo tříd, a zejména jejich chování při chybových nebo nevalidních vstupních datech (například uživatelské vstupy, nebo data získávaná z API). Integrační testy mají za úkol kontrolovat chování interakce více jednotek dohromady, ve frontend aplikacích může jít třeba o správnou kompozici komponent. Frontend end-to-end testy pak ověřují chování aplikace jako celku při interakcích uživatele. [37]

Testy jsou typicky psány v testovacích frameworcích ve stejném jazyce, jako je napsána samotná aplikace, a zpravidla je píší ti samí programátoři, kteří vytvářeli danou funkcionalitu. Výjimkou je u frontend aplikací často end-to-end testování, které mohou provádět specializovaní *testeři* přiřazení k projektu. Vytvořené testy jsou simulací interakce reálného koncového uživatele s aplikací nebo systémem, jde tedy často o ověření reálného průchodu aplikací dle připravených scénářů. [37]

Hlavním důvodem pro psaní testů je ověření funkcionality a kvality kódu. Automatizované testování aplikací také výrazně zjednodušuje vyhledávání chyb a časovou náročnost *refactoringu*, tedy údržby kódu. Můžeme se totiž alespoň částečně spolehnout na to, že úpravou kódu nevznikla chyba někde, kde bychom ji nečekali, zvláště u větších projektů, kde mohou být části kódu více provázané (v tomto případě pak pomáhají zejména integrační a end-to-end testy).

Aby bylo testování aplikace dobře proveditelné, tak ale klade určité nároky na psaný kód. Aby byl projekt dobře *testovatelný*, je nutné dělit kód do samostatných modulů, vyvarovat se globálním stavům a přebytečné provázanosti modulů, a omezit vedlejší účinky funkcí. To ale odpovídá správným zásadám vývoje software, tedy testování samotné nás zároveň může vést k psaní kvalitního kódu.

Jest

Jest je jedním z nejpoužívanějších testovacích frameworků pro JavaScript. Kromě prostředí pro spouštění testů poskytuje řadu pomocných funkcí pro zjednodušení testování, jako jsou například funkce pro kontrolu ekvivalence, kontrolu volání funkcí s porovnáním argumentů atp. Dalším nástrojem, který díky frameworku Jest získáváme, je tzv. *mocking*, díky kterému můžeme nasimulovat chování funkcí nebo celých modulů kódu. [38]

Typickým příkladem využití mockingu je komponenta formuláře, kdy chceme testovat odeslání dat a následné chování UI po obdržení odpovědi ze serveru. Funkci, která odesílá data na server je nutné nasimulovat, jelikož nechceme, aby byl test závislý na serveru. Testy pak můžeme spouštět bez připojení k internetu a nejsou závislé na funkčnosti API (tedy netestujeme i API, ale pouze požadovanou funkcionalitu). Alternativou může být rovnou simulování API, například pomocí nástroje Mirage JS, to je ale zpravidla časově výrazně náročnější na implementaci a následnou údržbu. [39]

Jest se používá zejména při psaní unit a integračních testů, k end-to-end testování pouze v omezené míře. To je dané tím, že se testy v Jest spouští v omezeném prostředí. Pokud chceme testovat React komponenty a jejich chování při manipulaci uživatelem (manipulace s DOM), je nutné využít rozšíření, jako je třeba knihovna `react-testing-library`, ale ani potom nemáme

jistotu, že chování bude odpovídat webovému prohlížeči. [40]

Cypress

Zmíněný problém s Jest řeší testovací framework Cypress, který se zaměřuje přímo na end-to-end testování frontend JavaScript aplikací. Zásadním rozdílem oproti Jest je prostředí, ve kterém se napsané testy spouští, jde totiž o plnohodnotný webový prohlížeč. Při spuštění Cypress si zároveň můžeme vybrat, v jakém konkrétním prohlížeči se testy budou spouštět, takže můžeme i otestovat případné rozdíly mezi jednotlivými prohlížeči. [41]

Testy se v Cypress dělí na dva typy. Prvním je end-to-end testování, kdy je aplikace spuštěna celá a můžeme přecházet mezi jednotlivými stránkami, jako při reálné návštěvě uživatelem. Druhým typem jsou *komponentové* testy, které testují pouze jednotlivé komponenty, do určité míry se tak jedná o frontend integrační testy. [41]

Nevýhodou Cypress může být pomalejší spouštění testů, jelikož je prostředí, ve kterém jsou spouštěny, náročnější. Dalším specifikem je přístup k psaní testů, kdy jsou všechny pomocné funkce pro interakce a testování asynchronní. Při vykonávání testů se všechny příkazy po určitou dobu opakují, dokud se například neustálí DOM, což dále může zpomalovat vykonání testů. To umožňuje Cypress být flexibilní (manipulace elementu v DOM po kliknutí na tlačítko může nějakou dobu trvat), ale znamená to zároveň nutnou změnu přístupu programátora k psaní testů.

3.3 Principy a nástroje CI/CD

CI/CD je zkratka pro kombinaci procesů, anglicky *Continuous Integration* a *Continuous Delivery/Deployment*, termíny se do češtiny překládají jako průběžná integrace a nasazení. Integrace se týká části vývoje, kdy je nutné, aby programátor po provedení změn aplikaci sestavil, otestoval a změny zapojil do sdíleného repozitáře. Průběžné nasazení je pak proces, při kterém se provedou případné kontroly změn, aplikace je sestavena do tzv. *artefaktu* a následně je nasazena její nová verze na prostředí dostupné koncovým uživatelem. [42]

Celý proces, respektive jeho automatizaci, umožňuje kombinace nástrojů, jako jsou například IDE, zmíněné npm a Husky. Pokud se ale bavíme o CI/CD nástrojích, míníme tím zejména nástroje pro vytváření automatických CI/CD *pipeline*. Příkladem takových samostatných nástrojů může být open-source nástroj Jenkins, CI/CD může ale být integrováno i do platform, jako jsou GitLab a GitHub. [43, 44, 45]

CI/CD pipeline se vytváří pomocí konfiguračních souborů ve formátech specifických pro daný nástroj nebo platformu a často tak nejsou přenosné, například nástroj Jenkins využívá omezenou část jazyka Groovy a platformy GitLab a GitHub používají rozdílné konfigurace ve formátu YAML. Pipeline se spouští buď ručně, nebo na základě změn ve sdílených repozitářích projektů (případně dalších událostí), a sestávají ze sledu příkazů které se mají vykonat. [43, 44, 45]

3.3.1 Github Actions

GitHub Actions jsou integrací nástroje pro CI/CD v platformě GitHub. Pipelines se konfigurují v tzv. *workflow* souborech, které udávají jeden konkrétní proces CI/CD. Tyto konfigurační soubory musí být v repozitáři uloženy ve skrytém adresáři `.github/workflows` a musí být ve formátu YAML. Detaily konfigurace dále popisuji v praktické části práce. [45]

Workflow se dále dělí na podprocesy nazvané *jobs* a ty sestávají z řady kroků, anglicky *step*. Kroky v rámci jednoho job běží v tomtéž prostředí, tzv. *runner*, ale jednotlivé jobs se spouští v oddělených prostředích a bez dodatečné konfigurace i paralelně. Pokud je pro jeden job nutný výsledek jiného, je nutné mezi nimi specifikovat závislost (pomocí položky `needs`) a případné artefakty explicitně předat. [45]

Platforma GitHub Actions je silně provázaná s platformou GitHub jako takovou, tedy spouštění workflows se může provádět na základě řady událostí týkajících se *issues*, nebo *pull requests* a můžeme z nich například vytvářet *release*, nebo provádět další změny v repozitáři (přidávání tagů, atp.). Workflows jsou dále rozšiřitelné o *Actions*, menší aplikace, které je možné dále vyvolávat v rámci workflow. Zmíněné předávání artefaktů mezi jobs provedeme použitím akce `upload-artifact` a `download-artifact`, kešování závislostí pak pomocí akce `cache`. [45]

3.4 Platform as a service (PaaS) a Firebase

Platform as a Service je anglické označení pro cloudovou platformu, poskytující vše nutné k vývoji, nasazení a obsluze aplikací a systémů. Místo nutnosti udržování hardwaru, operačních systémů a softwaru určenému pro nasazení, hosting a údržbu nám je poskytuje platforma sama. Nejvyužívanější z těchto platform jsou například Google Cloud a Amazon AWS, zároveň obě poskytují bezplatnou verzi platformy pro malé projekty. [46, 47, 48]

To je výhodné zejména pro menší projekty, u kterých by bylo finančně neschůdné celou infrastrukturu vytvářet a udržovat *in-house* (projekt by narostl o zmíněnou údržbu hardwaru a softwaru, museli bychom najmout vývojáře specializované na infrastrukturu, atp.). V momentě, kdy má společnost již větší množství vývojářů a aktivních projektů, se mohou platformy přestávat vyplácet, jelikož je cena za služby škálována dle zpracovávaného objemu dat (počet obsloužených požadavků, množství uložených dat v databázi, atd.), a je dále nutné provést analýzu, jestli by nebylo vhodnější infrastrukturu vytvářet a udržovat v rámci společnosti.

Využívání cloudových platform může vést k uvázání se k určitému poskytovateli, tzv. *vendor lock-in*, je tedy nutná určitá obezřetnost při využívání služeb a je lepší počítat s případným přechodem na jinou platformu. Služby jsou typicky modulární a je možné například využívat pouze hosting a úložiště platformy Google Cloud a zároveň používat databáze nasazené jinde, to může ale způsobit vyšší odezvu celého systému, než při využívání jedné samotné platformy (požadavky musí procházet přes více serverů). [49]

3.4.1 Firebase

Firebase je nadstavbou platformy Google Cloud, která odstiňuje její komplexitu. Při založení nového projektu ve Firebase se nám zároveň založí propojený Google Cloud projekt, je tedy možné zároveň používat všechny služby s ním spojené. Možnosti využití se dělí na dva „plány“, a to bezplatný *Spark* a placený *Blaze*. [50, 51]

Plán Spark má kvóty pro využití všech služeb v rámci Firebase a dále má omezení přístupnosti některých služeb, například Cloud Functions a většinu služeb Google Cloud není možné používat vůbec. Po dosažení kvót se přístup k dané službě uzamkne a zpřístupní se až po obnovení limitů (ty jsou většinou denní nebo měsíční), plán je ale kompletně zdarma. [51]

Plán Blaze obsahuje tytéž kvóty jako Spark, do kterých je využití služeb bezplatné. V případě jejich překročení se ale přístup ke službám nezablokuje a dále se cena jejich využívání řídí množstvím zpracovaných dat. Také je možné využívat kompletně všech služeb poskytovaných platformou Google Cloud.

Používání neomezeného plánu vyžaduje obezřetnost, jelikož v případě určitého typu chyby v aplikaci (odesílání požadavků na server v nekonečné smyčce, apod.) se může stát, že se čerpání služeb výrazně prodraží. Z tohoto důvodu je možné nastavit uživatelské limity čerpání, po jejichž překročení nás platforma buď upozorní, nebo dokonce zamezí dalšímu využívání služeb. [51]

Projekty ve Firebase sdružují registrované aplikace (maximálně třicet), přičemž všechny mají stejný přístup ke všem službám v rámci projektu. Tyto služby jsou zejména *Authentication*, *Cloud Storage*, *Realtime Database*, *Cloud Firestore*, *Cloud Functions*, *Analytics*, *Crashlytics*, *A/B Testing* a další. [50]

Firebase Authentication

Firebase Authentication je kompletním řešením pro správu uživatelů, umožňující řadu způsobů autentizace. Služba je následně propojena s dalšími, jako jsou Cloud Firestore a Realtime Database, u kterých umožňuje nastavení přístupů (autorizace) k datům pomocí konfigurace tzv. *Security Rules*. Například tedy omezíme přístup ke kolekci dat na přihlášeného uživatele s ID stejným, jako je ID kolekce, a tím zaručíme, že data může číst pouze jejich vlastník. Pokud chceme rozlišovat mezi kategoriemi uživatelů (administrátor, editor, apod.), je nutné využít značek, tzv.

claims, které se uživatelům musí přiřazovat z chráněného prostředí, tedy buď z backendu, nebo v prostředí Cloud Functions. [52]

Metody přihlašování jsou rozsáhlé a zahrnují kromě základních (e-mail a heslo, SMS) využívat přihlášení pomocí identity jiného poskytovatele (účty Google, Apple, Facebook, Twitter, GitHub) nebo umožnit vytvářet anonymní účty s omezeným přístupem, které je pak možné pomocí registrace povýšit na plnohodnotné. Specialitou je i umožnění napojení již existujícího řešení autentizace proto, abychom mohli využívat zmíněné nastavení autorizace pro databáze. [52]

Funkcionalita Authentication může být volitelně rozšířena pomocí služby *Identity Platform*, to zahrnuje ale i zpřísnění kvót pro aktivní plán. Rozšíření zahrnuje například možnosti vícefaktorového ověření, přihlášení pomocí služeb SAML a OpenID Connect, logování aktivity uživatelů a zpřístupnění synchronních Cloud Functions reagujících na události registrace / přihlášení uživatele (události `beforeCreate` a `beforeSignIn`), díky kterým můžeme modifikovat založené účty během jejich vytváření (například pro přidání označení oprávnění uživatele). [52]

Cloud Firestore

Firestore je flexibilní *NoSQL* databáze s volnou hierarchickou strukturou, díky čemuž poskytuje dobrou automatickou škálovatelnost. Struktura databáze sestává ze dvou hlavních konceptů. Prvním z nich jsou *dokumenty*, které obsahují položky s vnořenými daty různých typů (od primitivních typů jako jsou čísla a text až po vnořené objekty a reference na jiné dokumenty). Druhým konceptem jsou *kolekce*, které seskupují dokumenty a mohou do nich i být vnořené. Zároveň Firestore dodává i možnosti dávkových změn a transakcí, které fungují obdobně, jako v klasických SQL databázích. [53]

Nad dokumenty a kolekcemi se pak můžeme dotazovat pomocí tzv. *queries*, které umožňují jednoduché, mělké dotazy, ale i komplexní dotazy spojené s tříděním, seřazováním a stránkováním. Data pak můžeme získávat buď jednorázově pomocí *snapshots*, tedy jakýchsi „snímků“ dat, nebo je možné získávat změny v databázi v reálném čase použitím takzvaných *realtime listeners*, díky kterým pak dostává klient informace pouze o změnách dat a snižuje se tak datový přenos a počet požadavků. [53]

Firestore nám nicméně přes veškeré výhody neposkytuje nic, co by nám zjednodušovalo modelování vazeb mezi daty. Vytvoření modelu a přidávání vazeb (typicky pomocí ID nebo referencí na dokument nebo kolekci) musíme provádět ručně, a díky volnosti struktury nám není diktován žádný konkrétní přístup.

Cloud Storage

Storage je jednoduché úložiště dat, umožňující ukládat uživateli vytvářená média, a je tedy uzpůsobeno i pro větší datové soubory. K souborům ve Storage se přistupuje obdobně jako k datům ve Firestore pomocí reference, což je cesta k dané položce, například `images/picture.png`. Jak je vidět z příkladu, pokud chceme sdružovat soubory ve Storage, jednoduše v cestě uvedeme sdružující segment (tedy `images`) oddělený lomítkem. [54]

Dále pro Storage platí, stejně jako pro Firestore, že můžeme provádět autorizaci přístupu k datům pomocí bezpečnostních pravidel. To je možné i na základě metadat daných souborů, tedy třeba velikosti, typu, atp. Pokud se chceme v rámci dokumentů ve Firestore odkazovat na soubory ve Storage, přidáme jejich cestu jako textový řetězec do položky dokumentu. [54]

Kapitola 4
Praktická část

4.1 Analýza zadání a existujících řešení

Hlavním cílem této bakalářské práce je vytvoření návrhu a prototypu systému pro správu obsahu. Pro vytvoření návrhu je ale nutné vytyčit, co vlastně takový systém je a co poskytuje.

4.1.1 Systémy pro správu obsahu (CMS)

Tato kapitola částečně čerpá z [55]. *CMS* (z angličtiny Content Management System) jsou systémy využívané pro správu digitálního obsahu typicky webových stránek nebo aplikací, také bývají označovány jako redakční systémy. Typicky tyto systémy umožňují:

- správu dokumentů, obsahu stránek, článků, například formou WYSIWYG editoru
- správu souborů a obrázků
- správu uživatelů a jejich oprávnění
- případné úpravy funkcionality pomocí rozšíření třetích stran

CMS se podle implementace a způsobu jejich použití dělí na několik kategorií. Nejpoužívanějšími z nich jsou tzv. *tradiční*, například jde o systém Wordpress (ve starších verzích). Jsou postavené monoliticky a přímo poskytují možnosti zobrazení spravovaného obsahu. Tyto CMS jsou nedílnou součástí vytvářených webových stránek, tedy při nasazení stránek nebo aplikace je systém přímo jejich součástí, což také znamená, že prostředí, do kterého nasazujeme, musí podporovat i vyvíjenou aplikaci i CMS.

Druhou kategorií systémů pro správu obsahu jsou tzv. *decoupled* (do češtiny by se přeložilo jako „oddělené“) CMS. Tyto sice obdobně jako tradiční systémy zároveň poskytují možnosti zobrazení obsahu i správy, nicméně nejen napřímo, ale i prostřednictvím API. Příkladem může být také již zmíněný Wordpress, který od určité verze poskytuje pro přístup k veškerému obsahu REST API. Tím je výrazně zvýšena flexibilita řešení, protože můžeme buď využívat přímé zobrazení obsahu, nebo poskytovat obsah dalším aplikacím. Také se tímto přístupem zvyšují možnosti případných rozšíření systému, například může jít o rozšíření administračního rozhraní samotného.

Dále existují takzvané *headless* CMS (do češtiny přeloženo jako „bezhlavé“), které neposkytují žádné možnosti zobrazení obsahu, ale pouze zpřístupňují spravovaný obsah pomocí API. Příkladem může být open-source systém Strapi. Výhodou těchto CMS je obrovská flexibilita možností vytváření aplikací s využitím spravovaného obsahu a jeho sdílení mezi více aplikacemi.

Negativem pak může být to, že poskytují pouze správu obsahu a nic dalšího, tedy nijak nám nepomáhají se samotným vývojem aplikací.

V poslední kategorii se nachází *SaaS* CMS (z angličtiny zkratka pro „Software as a Service“), tedy cloudové CMS služby. V zásadě nám umožňují správu digitálního obsahu s nejnižšími nároky, protože není nutné se starat o nasazení systému nebo jeho údržbu, to za nás provádí poskytovatel. Napojení na vyvíjené aplikace je téměř stejné jako u headless CMS, rozdílem je ale to, že služba musí být nutně přístupná přímo z internetu. Headless CMS mohou být potenciálně přístupné pouze v rámci nasazených kontejnerů s aplikacemi. Dále nám pochopitelně není umožněno do SaaS CMS nijak zasahovat nebo je upravovat.

4.1.2 October CMS

October CMS je systém pro správu obsahu, postavený na PHP frameworku Laravel, spadající do kategorie tradičních CMS. Je zaměřený přímo na vývojáře, je tedy velmi flexibilní, co se týče úprav a rozšíření funkcionality, a zároveň poskytuje dostatečnou pomoc při vývoji, například umožňuje využívat správce balíčků Composer i pro rozšíření, takže nasazení aplikací může být snadno automatizováno. Systém byl původně publikován v roce 2013 pod MIT licencí, vyvíjen komunitou a zpřístupněn bezplatně, v roce 2021 ale přešel na proprietární licenci a placený model.

Pro zobrazení obsahu používá October CMS šablony (anglicky „templates“) využívající rozšíření jazyka Twig. Tyto šablony se dělí na čtyři typy, a to stránky („pages“), obalové bloky stránek („layouts“), bloky pro opakované použití („partials“) a statické obsahové bloky („content“). Stránky, layouty a partials mohou obsahovat tři sekce oddělené dvojitým rovnítkem (`==`). Z toho první umožňuje konfiguraci speciálních proměnných v INI formátu. Následuje sekce pro omezený PHP kód – definice funkcí, které je následně i možné používat jako AJAX handlery pro částečnou reaktivitu stránek. Poslední sekcí je samotná Twig šablona obsahující HTML.

Tento systém šablon je pro většinu použití dostačující, má ale určitá specifika:

- Twig používá jinou syntaxi než PHP, orientovanou na šablony. To může být výhoda i nevýhoda, musíme se naučit chování obou jazyků.
- Twig je oproti PHP omezený, v komplikovanějších případech pak musíme kombinovat PHP kód (v druhé sekci šablony) s Twig kódem.
- Každá stránka má definovaný pouze jeden layout, není umožněno vnoření obalových bloků. To může vést k nutnosti opakovat kód v šablonách, pokud se liší pouze menší částí.
- October CMS poskytuje AJAX framework, umožňující získávání dat z backendu a následné obnovení části stránky (partial) u klienta bez načítání celé stránky. V případě, že ale potřebujeme silnější reaktivitu nebo udržovat nějaký stav na klientovi, je nutné dále využít JavaScript. Tedy je pak možné, že pro zobrazení obsahu kombinujeme v jedné šabloně až čtyři jazyky (PHP, HTML, Twig, JavaScript), což může komplikovat orientaci v kódu.

Práce s daty v databázi je zprostředkována pomocí nadstavby *Eloquent*, což je implementace patternu *Active Record* frameworku Laravel. Jde o *ORM* (z angl. „Object–Relational Mapping“) poskytující nejen základní *CRUD* (z angl. „Create, Read, Update, Delete“) operace ale i funkce pro následující práci s daty.

Definice typů dat se provádí pomocí definice tzv. *modelu* – PHP třídy, jejíž instance reprezentují řádky databázové tabulky. Každý model tak odpovídá jedné tabulce v databázi. Zobrazení těchto dat v administračním rozhraní a formuláře k jejich vytváření/upravování se pak konfigurují pomocí souborů `columns.yaml` a `fields.yaml`.

Díky přístupu k šablonám a definicím typů dat je snadné obojí verzovat a nasazovat, například oproti CMS WordPress, kde jsou definice typů dat uloženy v databázi.

4.2 Návrh řešení systému

Navrhované CMS by mělo být primárně určeno pro vývojáře, mělo by tedy poskytovat zejména nástroje pro zjednodušení vývoje. Zároveň by mělo být snadné verzovat a nasazovat aplikace napsané s jeho použitím (z mého pohledu je toto jeden z nejzásadnějších problémů rozšířenějších CMS typu Wordpress).

Dle zadání bude systém vyvíjen v jazyce TypeScript a frameworku Next.js, zvolil jsem tedy jako jeho typ CMS tradiční s případným rozšířením o API, zejména kvůli rychlosti vývoje a možnostem úprav, Next.js také přímo poskytuje způsoby zobrazení dat. Pokud by se při implementaci prototypu ukázalo, že je touto volbou vývoj výrazně omezen, byl by do budoucna případně přepsán jako headless CMS (to by ale pak komplikovalo nasazení systému).

Zásadním rozhodnutím byla dále volba backendu, respektive jestli psát vlastní napojení na databázi (a případně jakou zvolit), nebo využít nějaké platformy. V zájmu zjednodušení a urychlení vývoje jsem pro tento účel zvolil platformu Firebase. Systém tak zároveň získává obrovskou flexibilitu tím, že API Firebase je přístupné přímo z frontendu klienta. Je tedy velmi snadné v případě potřeby provádět libovolné dotazování nad databází.

Firebase také řeší téměř všechny potřeby systému, ať už jde o hosting (Firebase Hosting), databázi (Firestore), autentizaci a přihlašování (Authentication), nebo úložiště uživatelských dat (Storage). Jelikož se počítá zejména s vývojem menších projektů, neměly by být zásadním omezením kvóty.

Navrhované CMS by dále mělo plně využít toho, že spojuje správu dat s možnostmi jejich zobrazení a zároveň využívá jazyk TypeScript, využití všech poskytovaných funkcionalit by mělo být typově bezpečné. To by bylo zásadní výhodou oproti většině CMS postavených na jazyce PHP, včetně October CMS.

4.2.1 Administrační rozhraní

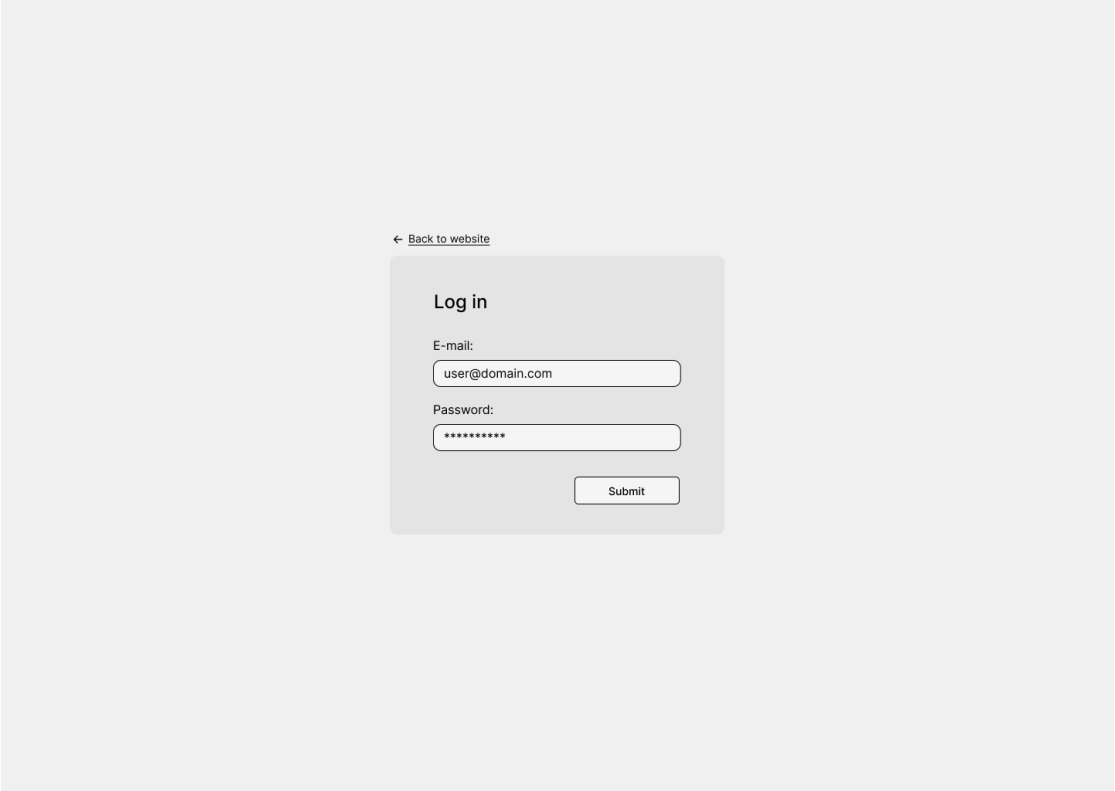
Jelikož je CMS určené pro vývojáře, administrační rozhraní by mělo zejména poskytovat uživatelsky přívětivý ekvivalent CRUD operací nad daty. Tímto způsobem by ho pak případně bylo možné snáze využít i jako administraci pro koncové uživatele, které nechceme zbytečně zahlcovat konfigurací. S využitím Next.js serverových komponent by následně mělo jít omezit přístup k částem administračního rozhraní různým skupinám uživatelů (administrátoři, uživatelé, atd.).

Pro účely následného vytvoření prototypu aplikace jsem vytvořil skicy (dále anglicky *wireframe*) tří stránek, a to stránky pro přihlášení, základní CMS stránky pro zobrazení a správu obsahu a stránky (modálního okna) pro vytvoření a úpravu konkrétní položky. Tímto jsou pokryty v zásadě všechny potřebné případy využití administračního rozhraní.

Přihlašovací stránka

Návrh stránky obsahuje pouze jednoduchý formulář pro přihlášení e-mailem a heslem. Tuto stránku by mělo být do budoucna snadné rozšířit o případné další možnosti přihlášení.

Jelikož byla pro autentizaci zvolena služba Firebase Authentication, je přihlašovacích způsobů relativně mnoho. Muselo by být rozhodnuto, které z přihlašovacích způsobů kromě jména a hesla mají pro účely CMS smysl. Například použití přihlášení pomocí Google účtu je sice výrazně bezpečnější, než uživatelem zvolené heslo, a zároveň jde o jednodušší způsob přihlášení (typicky stačí kliknout na tlačítko a případně potvrdit), ale omezuje nás to počtem vytvořených účtů na jednoho uživatele (jako identifikátor je brán e-mail spojený s Google účtem) a pokud bychom chtěli přiřazovat uživateli scope (viz kapitola o Firebase Authentication v teoretické části), nutilo by nás to využít Cloud Functions a zmíněné autentizační hooky.



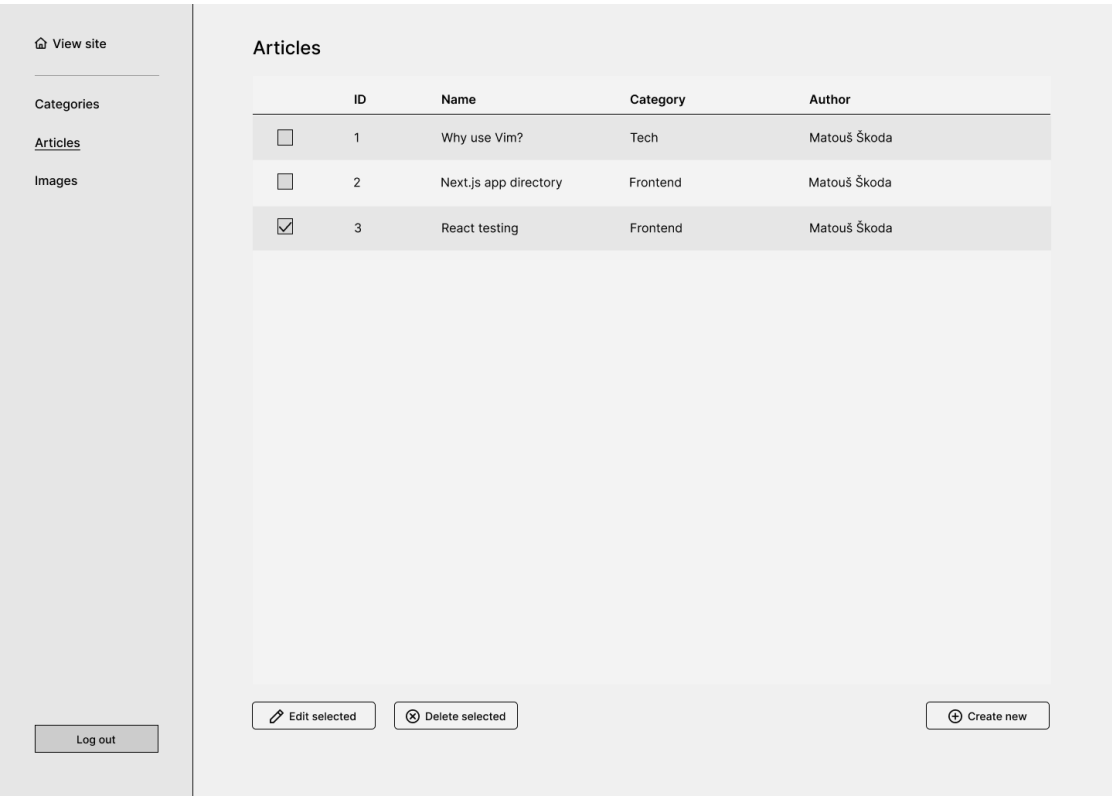
Obrázek 4.1 Wireframe stránky pro přihlášení

Pokud bychom se rozhodli podporovat zmíněné další přihlašovací způsoby, jednalo by se v zásadě o pouhé přidání dalších tlačítek do stránky. Případně, abychom nemátli uživatele, by bylo možné přidat druhou „úroveň“ přihlášení. Tedy nejprve uživatele nechat vybrat z tlačítek pro různé přihlašovací způsoby a až po výběru kombinace e-mailu a hesla následně zobrazit formulář.

Základní CMS stránka

Wireframe základní stránky CMS se dělí na dvě základní komponenty, a to navigační menu vlevo, které by mělo být viditelné na všech stránkách CMS, a obsah CMS stránky. Menu obsahuje popořadě seshora odkaz pro zobrazení samotných webových stránek, odkazy na jednotlivé stránky datových kolekcí a jako poslední tlačítko k odhlášení z CMS.

V případě odkazů na kolekce v menu jsem se rozhodoval mezi jednoduchým jednoúrovňovým způsobem, který jsem ve výsledku zvolil, a vícestupňovým přístupem, kdy by v CMS byla navíc samostatná stránka agregující všechny kolekce a dále by obsahovala odkazy na jejich stránky. Druhý způsob by byl méně přehledný pro uživatele a vyžadoval by implementaci *drobečkové navigace*, nicméně by byl výhodnější v případě, že by kolekcí dat bylo větší množství nebo kdyby v menu mělo být více dalších odkazů (konfigurace, uživatelé, soubory, média, atp.).



Obrázek 4.2 Wireframe základní CMS stránky

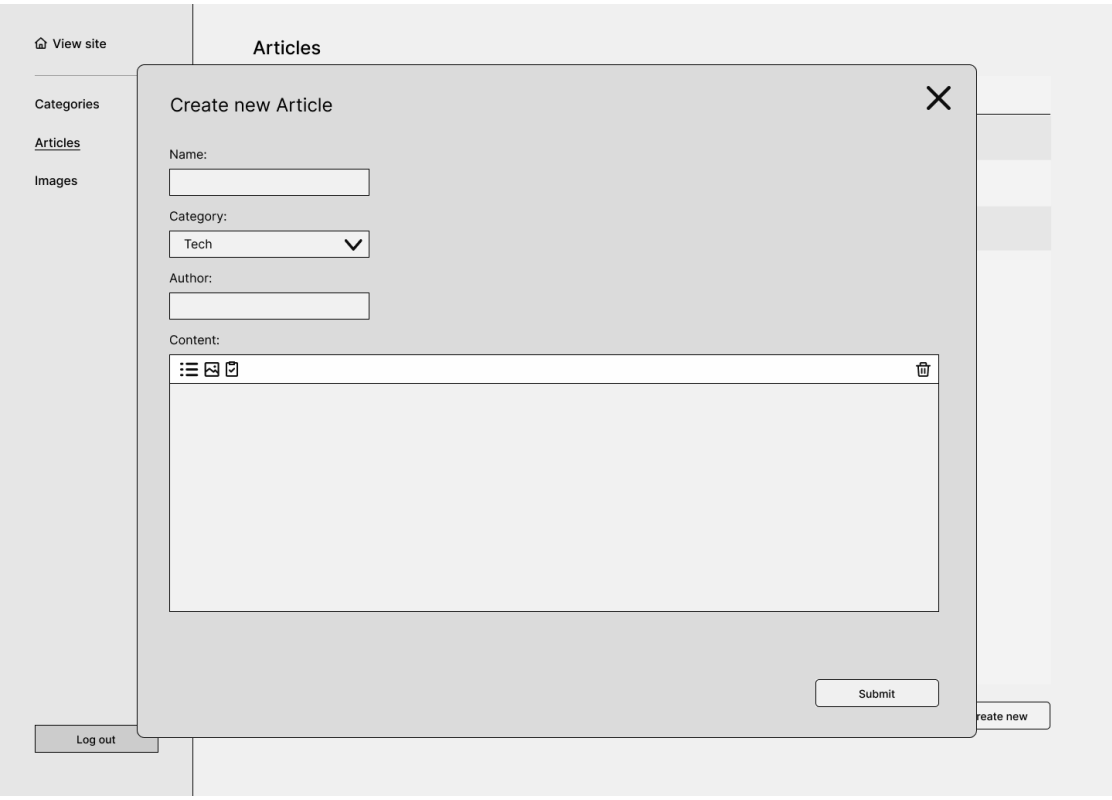
Jako ukázku obsahu CMS stránky jsem zvolil kolekci dat, jelikož jde o nejzásadnější funkcionalitu celé aplikace, jedná se zejména o tabulku zobrazující data položek kolekce a tlačítka pro práci s daty.

Tabulka by měla fungovat tak, že zaškrtávacími tlačítky vlevo vedle položek můžeme označit buď jednu pro editování (jinak je tlačítko pro editaci neaktivní a zašedlé) nebo více položek, které pak můžeme mazat druhým tlačítkem (vyžadující potvrzení). Třetí tlačítko vpravo pak slouží pro přidání nového datového záznamu do kolekce. Dále by zobrazení mělo podporovat *stránkování* dat a případné řazení a úpravy zobrazení (například zobrazovat pouze určité sloupce atp.).

Stránka vytváření nové položky v kolekci

Pro vytváření a úpravu datových položek jsem místo klasické stránky zvolil tzv. *modální okno* (někdy také dialogové okno), tedy blok částečně překrývající již zobrazenou stránku. Důvodem pro to bylo to, že přechod na novou stránku může uživatele do určité míry zmást, zároveň je při správném použití (například stále viditelný nadpis stránky pod oknem) uživateli jasné, jak se do aktuálního stavu dostal. Dále tak odpadá nutnost drobečkové navigace a aplikace je dle mého názoru přehlednější (působí „mělčeji“).

Modální okna často bývají uzavíratelná kliknutím mimo okno (na stránku „pod oknem“). Vzhledem k povaze obsahu (formulář pro vytvoření nebo úpravu položky) je tedy nutné při zavření okna buď vynutit potvrzení nebo vyžadovat přesnější kliknutí na ikonu křížku vpravo nahoře, aby uživatel okno nezavřel náhodným kliknutím vedle formulářového pole a tím si neresetoval stav formulář.



Obrázek 4.3 Wireframe stránky pro vytváření/editaci položky kolekce

Obsah formuláře samotný se skládá z formulářových prvků s popiskami. V ukázce jsou načrtnuté prvky pro jednoduchý textový vstup, volbu (navázání na datovou položku jiného typu) a WYSIWYG editor. Tyto formulářové prvky a další bude nutné vytvořit tak, aby se mohly genericky použít u každé kolekce dat.

V případě úpravy položky kolekce se tentýž formulář předem vyplní existujícími daty a následně se pomocí API Firestore odešlou a přepíší pouze změněné části položky. To je nutné zejména kvůli formulářovým prvkům jako jsou soubory, jelikož data která uživatel zadává (reálně cesta k souboru v lokálním prostředí) jsou jiná, než ta uložená v kolekci (cesta k souboru v rámci Storage).

4.3 Nastavení nástrojů a CI/CD pipeline

V této kapitole popisuji založení projektu, konfiguraci nástrojů nutných pro usnadnění vývoje prototypu, vytvoření CI/CD pipeline v platformě GitHub a dále nastavení projektu v platformě Firebase a nasazení do služby Firebase Hosting.

4.3.1 Nastavení projektu a nástrojů pro vývoj

Založení projektu začíná jednoduše vytvořením Git repozitáře. Vzdálený Git repozitář ale sám o sobě není nijak zabezpečený, co se týče přepisování větví. Základní chování je tedy takové, že povoluje operaci `git push` nad libovolnou větví a bez problémů si tak můžeme vymazat celou historii v hlavní větvi.

To samozřejmě nechceme aby bylo možné, a tak platforma GitHub poskytuje nastavení ochrany větví pomocí pravidel v záložce *Settings, Branches*. Pro každou požadovanou větev přidáme pravidlo specifikující název větve a nastaveními zapneme její ochranu.

Pro práci v týmech jsou nejdůležitějšími nastaveními následující:

- *Require a pull request before merging* – nutí vývojáře nejdříve změny vytvářet ve vedlejší větvi a následně pomocí *Pull requestu* (dále PR) změny zavést do chráněné větve
- *Require approvals* – PR musí před zavedením změn schválit jiní vývojáři
- *Require conversation resolution before merging* – komentáře jiných vývojářů k PR musí být nejdříve vyřešeny, jinak není dovoleno zavést změny

Vývoj projektu by pak měl probíhat tak, že vývojář si nejdříve vytvoří tzv. *feature* větev, tedy větev pro lokální změny vyvíjené funkcionality nebo oprav. Tuto větev symbolicky vlastní a vyvíjí v ní danou část projektu.

Po dokončení vývoje vytvoří PR cílící na větev, ze které se daná vedlejší větev oddělila. PR zkontrolují a schválí další vývojáři a vlastník projektu a případně jsou dodatečně zavedeny změny, opravující nalezené chyby nebo nedostatky. Nakonec jsou změny zavedeny do hlavní vývojové větve.

Výhodou tohoto přístupu k vývoji je to, že vývojáři mají volnost ve vedlejších vývojových větvích, které „vlastní“, a to včetně případného použití *rebase* pro „uklizení“ historie. Kontroly v rámci CI/CD se pak provádí typicky pro každý vytvořený pull request. Díky tomu je potom kód v hlavních vývojových větvích vždy korektní a otestovaný, ale zároveň můžeme do vedlejších větví nahrát i nekorektní kód v případě, že potřebujeme spolupracovat na opravách chyb.

Drobností je při založení Git repozitáře ještě vytvoření skrytého souboru `.gitignore`, podle kterého Git ignoruje specifikované soubory při zařazení k verzování. Přidávají se do něj adresáře závislostí projektu, logy a typicky `.env` soubory, které často obsahují API klíče. Pro účely vytvářeného prototypu jsem do něj zařadil zejména následující položky:

- `node_modules/`, obsahující instalované závislosti npm
- `.next/` a `out/`, adresáře pro mezipaměť a sestavení Next.js
- soubory `firebasePublic.json` a `firebaseAdmin.json`, které poskytují přístup k Firebase

npm

Konfigurace těchto nástrojů je poměrně přímočará, může mít ale určitá úskalí. Například nástroje TypeScript a ESLint kvůli zpětné kompatibilitě nemění způsob konfigurace a mají tak řadu nastavení, které se musí změnit ze základního chování.

Pro nástroj npm jsem pomocí příkazu `npm init` vytvořil základní `package.json`, a dále jsem ho upravil následujícím způsobem (pouze zajímavější konfigurace).

Výpis kódu 4.1 Konfigurace nástroje npm

```
    ...
09  "type": "module",
10  "scripts": {
11    "dev": "env TAILWIND_MODE=watch NODE_ENV=development next dev",
12    "build": "next build",
13    "start": "next start",
14    "lint": "eslint .",
15    "lint:fix": "eslint --fix .",
16    "lint-staged": "lint-staged",
17    "tsc": "tsc",
18    "prettier": "prettier --write .",
19    "test": "jest",
20    "test:watch": "jest --watch"
21    "check-all": "npm run lint && npm run tsc && npm run test",
22    "generate-helpers": "tsc --project './tsconfig.scripts.json' &&
        node ./compiled-scripts/src/admin/scripts/generate-helpers.mjs &&
        prettier --write ./src/admin/__generated__",
23    "postinstall": "npm run generate-db-api",
24    "prepare": "husky install"
25  },
    ...
```

Položka `type` na řádce 9 specifikuje způsob zpracování souborů s koncovkou `.js` runtimem Node.js. V základu se předpokládá, že tyto soubory využívají starší modulový systém *CommonJS*, při nastavení na hodnotu `module` se zpracování změní na novější *EcmaScript* (dále jen ES) moduly, podporované prohlížeči.

Některé nástroje podporují zpracování konfiguračních souborů pouze formou CommonJS modulů, pak je s tímto nastavením nutné používat pro tyto soubory koncovku `.cjs`, případně pro ES moduly koncovku `.mjs`. Pokud je nutné kombinovat v projektu obě formy modulů, je lepší být explicitní v koncovkách kvůli rozlišitelnosti.

Skripty `dev`, `build` a `start` každý spouští různý vývojový mód Next.js. Dále následují skripty pro použití ESLint, na řádce 16 s využitím nástroje *lint-staged*, který jej spouští pouze na změněné verzované Gitem.

Na řádce 21 je definován skript agregující všechny kontroly kódu a automatické testy, typicky ho využijeme před použitím operace `git push`, abychom nemuseli každý nástroj spouštět zvlášť.

Dále je definován skript `generate-helpers`, který pomocí z konfigurace CMS vygeneruje TypeScript typy a pomocné funkce pro práci s daty, napojené na Firebase. Aby mohl být skript napsaný v jazyce TypeScript a spustitelný nezávisle na celé aplikaci je nutné ho nejdříve zkompilovat pomocí nástroje `tsc`. Následně se spustí Node.js se zkompilovaným skriptem a nakonec se vygenerovaný kód zformátuje nástrojem Prettier.

Dále za zmínku stojí lifecycle skripty `postinstall` a `prepare`, které po instalaci závislostí projektu dále nastaví Git hooks podle konfigurace Husky a vygenerují skriptem typy.

TypeScript

Konfigurace TypeScript kompilátoru se provádí, jak jsem psal v teoretické části práce, v souboru `tsconfig.json`. Je ale možné při spouštění CLI pomocí přepínače `--project` specifikovat jiný soubor jako konfiguraci projektu. Toho jsem využil pro kompilaci skriptu pro generování typů.

Výpis kódu 4.2 Zajímavější část konfigurace kompilátoru TypeScript

```
01  {
02    "compilerOptions": {
03      ...
04      "allowJs": true,
05      "checkJs": true,
06      "noEmit": true,
07      "jsx": "preserve",
08      "plugins": [
09          { "name": "next" }
10      ]
11    },
12    ...
13  }
```

Pomocí nastavení `"allowJs"` a `"checkJs"` povolíme zavést do TypeScript projektu soubory psané v samotném JavaScriptu a zapneme kontroly. Toto nastavení umožňuje konfiguraci aplikace psanou v JS s kontrolou typů konfiguračního objektu, bez nutnosti kompilovat projekt.

Na 6. řádku je přepnuta položka `"noEmit"`, to umožňuje používat kompilátor pro kontrolu chyb bez generování přeloženého kódu. Řádky 7 a 9 jsou specifické pro využití frameworku Next.js. JSX kód je nutné při kompilaci zachovat, jelikož Next.js na pozadí kód zpracovává, dále je přidán plugin pro framework.

Pro účely kompilace skriptu pro generování dat jsem vytvořil další konfigurační soubor `tsconfig.scripts.json`, který rozšiřuje ten základní:

Výpis kódu 4.3 Rošíření základního `tsconfig.json` pro kompilaci skriptu

```
01  {
02    "extends": "./tsconfig.json",
03    "compilerOptions": {
04      "noEmit": false,
05      "outDir": "compiled-scripts"
06    },
07    "include": ["src/admin/scripts/generate-helpers.ts"]
08  }
```

Při rozšíření základní konfigurace pomocí klíčového slova `"extends"` je chování takové, že všechna specifikovaná nastavení přepisují ty původní. Zejména jde o změnu v `"noEmit"`, jelikož chceme vytvořit spustitelný kód pro Node.js, a změna cílových souborů pomocí `"include"`.

ESLint, Prettier

Oba nástroje znovu vyžadují specifikaci ignorovaných souborů a adresářů. V případě Prettier jde o soubor `.prettierignore` a u ESLint se jedná o pole `"ignorePatterns"` v konfiguraci nástroje v `".eslintrc.json"`.

Nástroj ESLint je nutné pro podporu jazyka TypeScript a Reactu rozšířit o pluginy, konkrétně jde o pluginy `@typescript-eslint/eslint-plugin` a `eslint-plugin-react`. Ty a další je nutné nejdříve nainstalovat do projektu jako vývojové závislosti. Pro kompatibilitu s nástrojem Prettier je dále nutné nainstalovat ještě plugin pro něj.

Tyto pluginy se pak přidají do pole `"extends"` v `.eslintrc.json`. Plugin pro TS je ještě nutné přidat do `"plugins"` a dále nainstalovat a nastavit TS parser `@typescript-eslint/parser`, aby ESLint byl vůbec schopný soubory zpracovávat.

Kromě přidání celé řady pluginů je dále nutné upravit určitá pravidla. Pro účely sdílení konfigurace mezi projekty je možné vytvořit npm balíček, který se poté nainstaluje do projektu a použije se v poli `"extends"` (viz https://eslint.org/docs/latest/extend/shareable-configs), což je vhodné zejména u týmů s více projekty.

4.3.2 CI/CD

Pro konfiguraci GitHub Actions jsem vytvořil soubor `".github/workflows/ci.yml"`, ve kterém jsem definoval workflow pro kontroly kódu. V této podkapitole postupně popíši jeho části.

Nejprve jsem specifikoval název a události, na které se workflow spouští, podle způsobu práce s repozitářem popsaným v předchozí podkapitole o nastavení projektu. Workflow se tedy spouští na každý pull request a dále na push do větve `main`.

Výpis kódu 4.4 CI/CD - Definice spouštění workflow

```
01  name: CI
02
03  on:
04    pull_request:
05    push:
06      branches:
07        - main
```

Dále jsem vytvořil definice jednotlivých jobs, konkrétně `setup`, `lint`, `typecheck`, `unit-test` a `build`.

Setup

Tento job připraví prostředí a instaluje závislosti projektu, jednoduché spuštění `npm ci` při každém běhu by ale bylo neefektivní. Pokud se totiž nezmění `package-lock.json` tedy definice závislostí, není je nutné znovu instalovat, ale můžeme je kešovat. Aby bylo možné kešování v dalších jobs, je nutné klíče přiřadit do výstupu:

Výpis kódu 4.5 CI/CD - Setup job - základ nastavení

```
01  setup:
02    name: Setup
03    runs-on: ubuntu-latest
04    outputs:
05      cache-npm-key: ${{ steps.npm-cache-key.outputs.key }}
06      cache-node-modules-key:
            ${{ steps.node-modules-cache-key.outputs.key }}
```

Aby vůbec v prostředí jobu byly soubory repozitáře, je nutné použít akci `checkout`. Dále se na řádcích 6 a 9 vytvoří klíče pro kešování globálního `.npm`, ve kterém samotné npm kešuje závislosti, a `node_modules`, kam jsou instalovány:

Výpis kódu 4.6 CI/CD - Setup job - vytvoření klíčů pro kešování

```
01  steps:
02    - uses: actions/checkout@v3
03
04    # Output cache keys
05    - id: npm-cache-key
06      run: echo "key=${{ runner.os }}-node-${{ github.ref }}-
            ${{ hashFiles('**/package-lock.json') }}" >> $GITHUB_OUTPUT
07
08    - id: node-modules-cache-key
09      run: echo "key=${{ runner.os }}-node-modules-${{ github.ref }}-
            ${{ hashFiles('**/package-lock.json') }}" >> $GITHUB_OUTPUT
```

Cache se potom využije tak, že se nejdříve pokusíme obnovit globální `.npm` a lokální `node_modules`, a následně se pokusíme nainstalovat závislosti. V případě, že není podle klíče nalezena uložená

verze lokálních modulů, se npm pokusí využít komprimované závislosti v `.npm`, aby nebylo nutné je stahovat. V ukázce dále se využije akce `cache`, takže je adresář buď obnoven, nebo je na konci jobu vytvořena nová cache.

Výpis kódu 4.7 CI/CD - Setup job - použití cache

```
      ...
01    - name: Cache local node_modules
02      id: cache-node-modules
03      uses: actions/cache@v3
04      with:
05        path: node_modules
06        key: ${{ steps.node-modules-cache-key.outputs.key }}
07
08    # Run install if there wasn't a cache hit
09    - run: npm ci
10      if: steps.cache-node-modules.outputs.cache-hit != 'true'
```

Dále pokračují joby `lint`, `typecheck`, `unit-test` a `build`, které závisí na jobu `setup`. To se specifikuje nastavením `needs: setup`. Každý z nich znovu musí využít akce `checkout` a `cache` pro obnovení nainstalovaných závislostí, jelikož běží v samostatných prostředích, a následně každý spustí svůj konkrétní příkaz pro vyvolání nástroje. Tyto joby jsou na sobě nezávislé a běží tedy paralelně, čímž ušetříme čas.

Výpis kódu 4.8 CI/CD - Job pro kontrolu typů

```
01  typecheck:
02    name: Typecheck
03    runs-on: ubuntu-latest
04    needs: setup
05
06    steps:
07      - uses: actions/checkout@v3
08      - uses: actions/cache@v3
09        with:
11          path: node_modules
12          key: ${{ needs.setup.outputs.cache-node-modules-key }}
13
14      - run: npm run ts
15        timeout-minutes: 2
```

4.4 Implementace prototypu CMS

V této kapitole popisuji implementaci prototypu systému podle výsledků analýzy existujících řešení a vytyčených požadavků na něj.

4.4.1 Konfigurace aplikace a definice dat

Nejdříve bylo nutné implementovat možnosti konfigurace CMS tak, aby tuto konfiguraci bylo možné verzovat a aby zároveň poskytovala typy pro jazyk TypeScript. Nakonec jsem se rozhodl pro přístup s jedním konfiguračním souborem, ze kterého se následně skriptem generují typy a pomocné funkce pro práci s databází. Tato nastavení vývojář provádí v souboru `/config/configuration.mjs` a je napsán v samotném JS proto, aby ho nebylo nutné kompilovat. Je ale rozšířen o TS anotace ve formátu *JSDoc* a díky nastavení TS kompilátoru jsou i v něm kontrolovány případné chyby.

První nutnou konfigurací je pole ID kolekcí definováno následovně:

Výpis kódu 4.9 Implementace - Konfigurace aplikace 1

```
01  const collectionIds = /** @type {const} */ ([
02    "File",
03    "Article",
04    "ArticleCategory",
05  ])
```

Díky použití syntaxi JSDoc s typem `const` je pak při dalším využití pole zpracováno jako tzv. *tuple*, tedy pole konstantí délky s konstantními položkami konkrétních hodnot (*string literal*). Obalové závorky okolo pole jsou zde nutné, jelikož jde o *type cast*, tedy změnu typu. Bez této konstrukce by typ proměnné bylo pole řetězců a ztratili bychom informaci o jejich hodnotách.

Další konfigurační položkou je samotná `appConfiguration`, nejdříve ale popíšu vytvořený typ pro kontrolu validity zadávaných hodnot. Nejdříve na řádce 1 definuji typ pro ID kolekcí. Protože je pole `collectionIds` konstantní, když vytvoříme typ jeho položek při přístupu číselným indexem, je výsledný typ `"File"| "Article"| "ArticleCategory"`, což je *union* řetězců.

Výpis kódu 4.10 Implementace - Konfigurace aplikace 2

```
01  export type CollectionId = (typeof collectionIds)[number]
02
03  export type AppConfiguration = Readonly<{
04    constants: Record<string, unknown>
05    collections: { [K in CollectionId]: CollectionConfig }
06  }>
```

První částí `AppConfiguration` je položka `constants`, umožňující libovolné vnořené položky, sloužící pro jednoduchou konfiguraci konstant, jako je například název stránek. Následuje položka `collections`, využívající tzv. *mapped type*. Syntax v hranatých závorkách iteruje přes union a každému poli objektu přiřadí typ `CollectionConfig`. Výsledný generovaný typ `collections` je potom nutně následující a nutí vývojáře, aby v konfiguraci specifikoval přesně ty kolekce, které definoval v poli.

```
{
  "File": CollectionConfig
  "Article": CollectionConfig
  "ArticleCategory": CollectionConfig
}
```

Typ `CollectionConfig` obsahuje některé jednodušší položky jako je `displayName`, `slug` a další, které z ukázky vynechávám. Jde o *generický* typ tak, aby nebylo nutné typ `T` zbytečně opakovat. Chtěl bych upozornit na použití typu `any`, které je zde v pořádku, kvůli tomu, že je konfigurace dále transformována a ve výsledku z ní vznikají explicitní typy.

Pro řadu položek je použit typ funkce z toho důvodu, že je transformace potom výrazně jednodušší, jelikož implementaci funkce je možné „zkopírovat“ pomocí metody `toString()`.

Výpis kódu 4.11 Implementace - Konfigurace kolekcí

```
01  export type CollectionConfig<
02    T extends Record<string, any> = Record<string, any>
03  > = {
      ...
04    type: Record<keyof T, string>
05    getSelectLabel: (document: T) => string
06    getSchema: () => ZodSchema
07    getDefaultValues: () => T
08    formConfig: FormConfig
09    listConfig: ListConfig
10    additionalHelpers?: Record<
        string,
        (...args: unknown[]) => unknown
      >
11  }
```

Položka `type` na řádce 4 definuje samotný typ pro dokumenty v dané kolekci a je následně transformována na TS typ. Bohužel jsem nepřišel na lepší způsob konfigurace než pomocí řetězců, všechny ostatní způsoby, které jsem vyzkoušel, vedly na cyklické závislosti modulů.

Na řádce 6 je definováno schéma validace formulářů daného typu dokumentů. Pro tento účel jsem využil knihovny *Zod*, která umožňuje z vytvořeného schématu dále generovat TS typy. Dále pak položka `formConfig` definuje položky formuláře, respektive jejich popisky a použitou komponentu.

Nastavení komponenty formulářového prvku je opět formou jejího názvu, nicméně vytvořil jsem pro tento účel typ, díky kterému vývojář dostává nápovědu v editoru. Většina unionu je napsaná ručně, podle implementovaných komponent, a na řádce 5 jsem využil typ *template literal*, který obdobně jako u typu pro kolekce vytvoří union interpolovaných řetězců.

Výpis kódu 4.12 Implementace - Typ pro konfiguraci komponenty formulářového pole

```
01 export type FormInput =
02   | "TextInput"
03   | "PasswordInput"
     ...
05   | `DocumentSelectInput<${CollectionId}>`
```

Dále se pomocí `listConfig` nastavuje chování pro vykreslování jednotlivých polí dokumentu v tabulkách, konkrétně jde o `header` pro hlavičku nebo popisek a funkci `render`, která jako parametr dostane daný dokument a je možné ji využít například pro formátování času nebo data.

Tato konfigurace je dobře verzovatelná, umožňuje kontrolu chyb a pomocí IDE nápovědy pro programátory. Ve výsledku se celá konfigurace zpracuje ve skriptu `generate-helpers.ts`, který konfiguraci projde a vygeneruje typy a funkce do souboru `src/admin/_generated_/index.ts`. Jelikož je generovaný kód zahrnutý do zpracování TS kompilátorem, tak ihned po použití skriptu a tsc dostaneme případné hlášky chyb, způsobených změnami v konfiguraci.

Zároveň nás skript upozorní, když už generovaný soubor existuje a umožní nám ho vytvořit s koncovkou `.new`, abychom mohli porovnat změny.

Pomocné funkce generované z konfigurace

Generování kódu nám umožňuje udržet informace, které by s použitím samotných generických typů a funkcí TS bylo velmi obtížné zachovat. Do definice vygenerovaných typů dokumentů se pak přidává ještě položka `"_typeId"`, abychom mohli vyhodnocovat typ dokumentů za běhu v generických funkcích. Oproti instancím tříd jinak není u objektů možné explicitně rozlišovat mezi typy obdobně jako operátorem `instanceof`.

Výpis kódu 4.13 Implementace - Příklad vygenerovaného typu

```
export type ArticleCategory = {
  _typeId: "ArticleCategory"
  displayName: string
  slug: string
}
```

Dále jsou veškeré pomocné funkce a objekty přiřazeny, obdobně jako u konfiguračního objektu, na objekt `collectionHelpers`. Ten je možné libovolně upravovat a měnit jeho položky, jelikož jeho typ není explicitně definován.

Pro přístup k datům ve Firebase se používají reference na kolekce, které ale samy o sobě nemají informace o typech dokumentů. Pro to jsem vytvořil následující pomocnou funkci:

Výpis kódu 4.14 Implementace - Typovaný přístup k Firebase kolekcím

```
const dataCollection = <T = DocumentData>(
    collectionId: CollectionId
) => {
  return collection(firestore, collectionId)
                        as CollectionReference<T>
}
```

Funkce dostane jako generický parametr typ dokumentu a jako argument `collectionName` název kolekce ve které jsou dokumenty uloženy ve Firestore. Tato funkce je použitá v zásadě pouze interně a vrací referenci na danou kolekci, přetypovanou na daný typ dokumentu. Tuto referenci pak můžeme libovolně používat k dotazování nad Firestore. Ve vygenerovaném kódu je funkce použita následujícím způsobem:

Výpis kódu 4.15 Implementace - Ukázka generovaného kódu 1

```
displayName: "File",
  slug: "file",
  collection: dataCollection<File>("File"),
  ...
```

Pomocná funkce pro přidání nového dokumentu do kolekce pak musí explicitně přidat zmíněné pole `_typeId`, je proto definována takto:

Výpis kódu 4.16 Implementace - Ukázka generovaného kódu 2

```
addDoc: (file: File) => {
  return addDoc(collectionHelpers.File.collection, {
    ...file,
    _typeId: "File",
  })
},
```

Nakonec jsou ještě generovány pomocné typy, funkce a objekty pro práci s dokumenty obecně. Jsou to například explicitní typy pro ID a typ dokumentů a další.

Výpis kódu 4.17 Implementace - Ukázka generovaného kódu 3

```
01  export type CollectionId = "File" | "Article" | "ArticleCategory"
02  export type DocumentType = File | Article | ArticleCategory
03  export type DocumentWithID<T extends DocumentType> = T & {id: string}
```

4.4.2 Základní struktura projektu

V projektu jsem vytvořil následující základní adresářovou strukturu:

```
├── README.md
├── package.json
├── tsconfig.json
├── next.config.mjs
├── ...konfigurační soubory dalších nástrojů
├── public/ ..... statické soubory, obrázky, atd.
├── config/
│   ├── configuration.mjs/ ..... konfigurační soubor pro CMS
│   └── ...konfigurační soubory Firebase
└── src
    ├── admin/ ..... zdrojové soubory CMS
    └── app/ ..... zdrojové soubory Next.js aplikace
        ├── (site)/ ..... veřejné stránky
        ├── login/ ..... stránka pro přihlášení
        ├── admin/ ..... stránky administračního rozhraní
        └── ... ..... zdrojové soubory Next.js aplikace
```

Struktura odděluje zdrojové a konfigurační soubory tak, aby se zjednodušila orientace v projektu. V adresáři `/src/app` jsem soubory a adresáři definoval strukturu výsledných stránek. Použití závorek u adresáře `(site)` je speciální tím, že vynechává adresář z této struktury, respektive segment je vynechán z výsledných URL, takže stránka `(site)/page.tsx` se mapuje na kořenovou url. Konečné mapování URL stránek v adresáři `app` je znázorněno zde:

```
├── layout.tsx ..... globální obalový blok
├── global-error.tsx ..... fallback pro chyby
├── loading.tsx
├── (site)/
│   ├── layout.tsx ..... obecný layout veřejných stránek
│   └── page.tsx ..... /
├── login/
│   ├── layout.tsx ..... layout přihlašovací stránky
│   └── page.tsx ..... /login
└── admin/
    ├── layout.tsx ..... obecný layout administrace
    ├── page.tsx ..... /admin
    └── collections/
        ├── page.tsx ..... /collections
        └── [colId]/
            └── page.tsx ..... /collections/<colId>
```

4.4.3 Implementace administračního rozhraní

Díky možnostem vnoření obalových bloků `layout.tsx` je možné definovat jiný layout pro veřejné stránky a pro administraci, v globálním layoutu je tak zejména jen obalení nutnými HTML tagy a blokem pro zpřístupnění Firebase SDK. To nám umožňuje používat i různé knihovny pro každou z částí aplikace, například v administrační části jsem využil knihovny Material UI, ve veřejné části CMS jsem přidal knihovnu Tailwind CSS. Nemělo by se nám pak stát, že by spolu knihovny kolidovaly. Takto vypadá implementace layoutu administrační části:

Výpis kódu 4.18 Implementace - Layout administrační části

```
01 const AdminLayout = ({ children }: Props) => {
02   return (
03     <AdminProviders>
04       <AuthGuard>
05         <PageLayout>{children}</PageLayout>
06       </AuthGuard>
07     </AdminProviders>
08   )
09 }
```

Obsah stránek je nejdřív obalen komponentou `AdminProviders`, která zpřístupňuje nutný React kontext pro další knihovny jako je Material UI. Komponenta `AuthGuard` je určena pro kontrolu aktuálně přihlášeného uživatele a případně přesměrovává na přihlašovací stránku. Jde o pouhou kontrolu v prohlížeči uživatele, tedy reálně neposkytuje žádné zabezpečení. Jelikož je ale autentizace a autorizace řešena přímo v jednotlivých službách Firebase, uživatel by mohl přejít na stránky administrace bez přihlášení a viděl by pouze rozbitý obsah, komponenta je tedy přidána zejména kvůli UX.

Samotné rozvržení administračního rozhraní je implementováno v komponentě `PageLayout` a vypadá následovně. Komponenty `Stack` a `Container` poskytuje Material UI a jde o obalové bloky, které využívají CSS *flexbox*. Na řádce 5 je vidět změna stylů pomocí rozšíření aktuálního `theme`, který je poskytován pomocí kontextu. Okno je takto rozděleno na levou část s menu (řádka 2) a zbylý obsah vycentrovaný pomocí `Container`.

Výpis kódu 4.19 Implementace - Rozvržení stránek administrace

```
01  <Stack direction="row" sx={{ width: "100vw", height: "100vh" }}>
02    <SideMenu />
03    <Container
04      component="main"
05      sx={(theme) => ({ paddingY: theme.spacing(2) })}
06    >
07      {children}
08    </Container>
09  </Stack>
```

Implementace menu byla relativně přímočará s pomocí Material UI, zejména s využitím komponent `Drawer` pro blok navigace, `List`, `ListItem` a `ListItemButton` pro seznam tlačítek. Pro tlačítka jsem využil vestavěnou komponentu Next.js `Link`, jelikož se reálně nejedná o tlačítka, ale o odkazy, a protože podporuje *preloading* stránek, díky kterému se obsah začne načítat už při přejetí kurzorem nad tlačítko.

Implementace generické stránky pro kolekce

Obecná stránka pro kolekci je vytvořena s dynamickou URL `/collections/<colId>`. Aby framework Next.js nemusel vyhodnocovat validitu daného parametru `colId` při každém dotazu, je poskytnuta možnost vygenerovat tyto hodnoty staticky už při sestavení projektu pomocí funkce `generateStaticParams` exportované ze souboru stránky následujícím způsobem:

Výpis kódu 4.20 Implementace - Generování statických parametrů URL

```
01  export async function generateStaticParams() {
02    return collectionIds
03      .filter((k) => k !== "File")
04      .map((k) => ({ colId: k }))
05  }
```

Jelikož pro kolekci se soubory `File` je nutné jiná stránka než obecná, nejdříve se ID odfiltrují na řádce 3 a pak se vrátí jako pole objektů. Důvod pro tento tvar je ten, že tyto vygenerované objekty se pak přímo předávají jako parametry komponentám samotných stránek.

V komponentě stránky pak mohu ID využít pro získání pomocného objektu pro danou kolekci a jeho položek a dále s nimi pracuji. Na řádce 7 využívám hook pro získání dat kolekce.

Výpis kódu 4.21 Implementace - Využití pomocného objektu

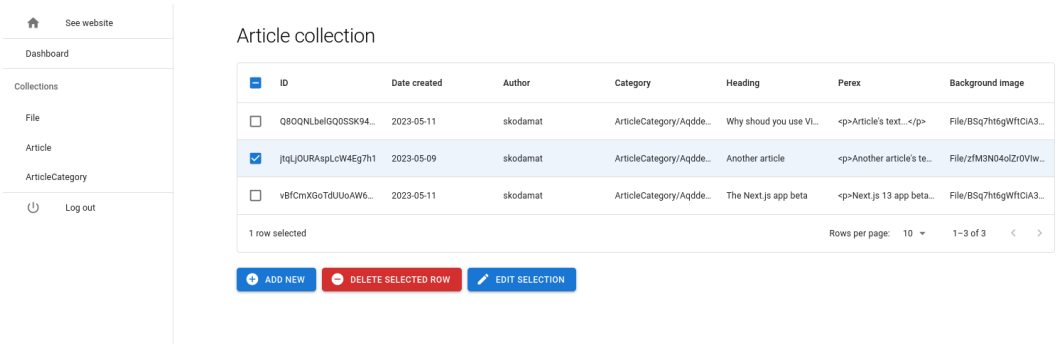
```
01  const {
02    listConfig,
03    useCollectionData,
04    getDocRef
05  } = collectionHelpers[colId]
06
07  const { status, data } = useCollectionData()
```

Pro zobrazení dat v tabulce jsem použil knihovnu `@mui/x-data-grid`, která rozšiřuje základní Material UI o složitější komponenty tabulek. Nejdříve musel vytvořit definici sloupců za použití položky `listConfig` konfigurace. Zároveň jsem využil pomocné funkce pro vykreslení jednotlivých polí.

Výpis kódu 4.22 Implementace - Definice sloupců pro tabulky

```
01  const columns = useMemo(() => {
02    const res: GridColDef[] = [
03      {
04        field: "id",
05        headerName: "ID",
06        flex: 1,
07      },
08    ]
09    Object.entries(listConfig).forEach(([k, item]) => {
10      res.push({
11        field: k,
12        headerName: item.header,
13        flex: 1,
14        valueGetter: (params: GridValueGetterParams<DocumentType>) =>
15          renderDocumentField(
16            params.row,
17            params.field as DocumentField<DocumentType>
18          ),
19      })
21    })
22
23    return res
24  }, [listConfig])
```

Poskládání samotných komponent pro zobrazení stránky bylo už přímočaré a výsledná stránka je vidět na následujícím snímku. Díky použité knihovně tabulka automaticky podporuje řazení, konfiguraci zobrazených sloupců i stránkování.



Obrázek 4.4 Snímek implementované generické stránky pro kolekce

Dále bylo nutné vytvořit dialogová okna pro vytvoření a editaci dokumentů. Pro tento účel jsem vytvořil komponenty `NewDocumentModal` a `EditDocumentModal`, popisovat budu ten první z nich.

Komponenta pro vytvoření dokumentu obdobně jako generická stránka dostane v props ID kolekce dat, které využije pro získání pomocných funkcí a definic. Dále je s využitím knihovny `react-hook-form` vytvořen formulář s validací dle schématu. Na řádce 5 jsem využil již zmíněné možnosti ze schématu vytvořit typ získávaných dat.

Výpis kódu 4.23 Implementace - Využití validačního schématu ve formuláři a vytvoření polí

```
01  const { schema, formConfig, addDoc, defaultValues } =
02    collectionHelpers[collectionId]
03
04  const { control, register, handleSubmit, reset } = useForm<
05    z.infer<typeof schema>
06  >({
07    defaultValues,
08    resolver: zodResolver(schema),
09  })
10
11  const formInputs = useMemo(() => {
11    const res = Object.entries(formConfig).map(([key, field]) => {
12      return (
13        <GenericFormInput
14          key={key}
15          name={key as keyof typeof formConfig}
16          label={field.label}
17          control={control}
18          register={register}
19          inputType={field.inputComponent}
20        />
21      )
22    })
23    return res
24  }, [control, register, formConfig])
```

Aby bylo vůbec možné takto genericky vytvářet formulářová pole, musel jsem nejdříve vytvořit komponentu `GenericFormInput` použitou na řádce 13. Ta v zásadě dostane parametr `inputType` a na vykreslí něj základě nějakou z implementovaných specializovaných komponent. Pro příklad popíši komponentu `TextInput`, kterou jsem použil pro vstupy jako text, e-mail, hesla, ale i čísla, datum a čas.

Kvůli použití `react-hook-form` je nutné typy props komponent pro vstupní pole definovat s generickými parametry. Protože většina těchto komponent dostává stejné parametry, vytvořil jsem pro ty základní následující typ. Generický parametr `TForm` reprezentuje typ formuláře, `TName` pak název pole.

Výpis kódu 4.24 Implementace - Základní typ pro parametry komponent vstupních polí

```
01  export type BaseInputProps<
02    TForm extends FieldValues,
03    TName extends FieldPath<TForm>
04  > = {
05    name: TName
06    label: string
07    validation?: Omit<
08      RegisterOptions<TForm, TName>,
09      "valueAsNumber" | "valueAsDate" | "setValueAs" | "disabled"
10    >
11    control: Control<TForm>
12  }
```

Samotná komponenta pro formulářové pole pak využívá dodanou komponentu `Controller` formulářové knihovny a komponentu `TextField` z Material UI.

Výpis kódu 4.25 Implementace - Komponenta pro textový vstup

```
01  function TextInput<TForm extends FieldValues, TName extends ...>({
02    ...
03  }: TextInputProps<TForm, TName>) {
04    return (
05      <Controller
06        control={control}
07        name={name}
08        rules={validation}
09        render={({ field, fieldState: { error } }) => (
10          <TextField
11            {...field}
12            ...
13          />
14        )}
15      />
16    )
17  }
```

Dále jsem implementoval obdobným způsobem dialogové okno pro úpravu obecného dokumentu. Následně jsem vytvořil specifické dialogy a stránku pro práci se soubory (kolekce `File`), které navíc pracují s Firebase Storage, a nakonec i stránku pro přihlašování uživatele. Implementovaný prototyp tedy pokrývá CRUD operace nad daty, umožňuje vazby mezi dokumenty i vkládání souborů, a poskytuje pomocné funkce pro práci s těmito daty.

4.5 Implementace automatizovaného testování

Pro nastavení nástroje Jest je nutná instalace řada dalších závislostí, aby byl vůbec schopný zpracovávat kód v jazyce TypeScript a s knihovnou React, jde zejména o překladač *Babel* a pluginy pro něj. Zásadními jsou dále prostředí *js-dom*, které při běhu testu simuluje DOM prohlížeče, a knihovna `react-testing-library`, která poskytuje řadu pomocných funkcí pro testování nad DOM. Nastavení pro tyto nástroje pak vypadají takto:

Výpis kódu 4.26 Testování - Konfigurace Babel

```
01  module.exports = {
02    presets: [
03      "@babel/preset-env",
04      ["@babel/preset-react", { runtime: "automatic" }],
05      "@babel/preset-typescript",
06    ],
07  }
```

Výpis kódu 4.27 Testování - Konfigurace Jest

```
01  export default {
02    clearMocks: true,
03    setupFilesAfterEnv: ["src/admin/setupTests.ts"],
04    testEnvironment: "jsdom",
05  }
```

Dále je nutné vytvořit soubor `setupTests.js`, ve kterém naimportujeme do testovacího prostředí vše ze zmíněné `react-testing-library`. Dále pak už můžeme psát unit testy. Pro ukázku popíši vytvořený unit test pro formulářové pole `TextInput`. Nejprve musíme vytvořit obalový formulář, pro to ale potřebujeme schéma a počáteční hodnoty formuláře.

Výpis kódu 4.28 Testování - Schéma formuláře

```
01  const schema = z
02  .object({ testInput: z.string().min(1, "Test input is required") })
03  .required()
04
05  const defaultValues = { testInput: "" }
```

Dále vytvoříme minimální komponentu pro formulář, kterou následně můžeme renderovat v testu. Pro vyzkoušení funkcionality vytvoříme dvě *mock* funkce, které se zavolají při odeslání formuláře.

Výpis kódu 4.29 Testování - Mocking

```
01  const mockSubmit = jest.fn()
02  const mockError = jest.fn()
03
04  const { control, handleSubmit } = useForm<z.infer<typeof schema>>({
05    defaultValues,
06    resolver: zodResolver(schema),
07    mode: "onChange",
08  })
09  const onSubmit = handleSubmit(mockSubmit, mockError)
```

Komponenta formuláře je pak pouhé obalení `TextInput` HTML formulářem `form` s přidaným tlačítkem pro odeslání. Nejdříve otestujeme, že se komponenta renderuje, a že předáním položky `label` se opravdu vytvoří HTML label s textem.

Výpis kódu 4.30 Testování - Test popisky pole formuláře

```
01  it("renders with label", () => {
02    const { getByLabelText } = render(<FormComponent />)
03
04    const element = getByLabelText("Test input label")
05    expect(element).toBeInTheDocument()
06  })
```

V dalším testu ověříme, že kliknutím na tlačítko se formulář odešle s chybou, jelikož ve schématu požadujeme vstup delší než 1.

Výpis kódu 4.31 Testování - Test chování formulářového pole 1

```
01  it("submits correctly", async () => {
02      const user = userEvent.setup()
03      const { getByTestId, ... } = render(
04        <FormComponent />
05      )
06
07      const button = getByTestId("submitButton")
08      await user.click(button)
09      expect(mockError).toHaveBeenCalled()
        ...
```

Dále zkontrolujeme, že je zobrazena chybová nápověda a po vepsání delšího vstupu zmizí a že se následně už formulář odešle se správnými daty. Pro simulaci interakcí uživatele jsem použil knihovnu `@testing-library/user-event`.

Výpis kódu 4.32 Testování - Test chování formulářového pole 2

```
01    ...
02    const error = getByText("Test input is required")
03    expect(error).toBeInTheDocument()
04
05    // Check if label is gone
06    await user.type(input, "Input value")
07    const errLabelAfter = queryByText("Test input is required")
08    expect(errLabelAfter).toBeNull()
09
10    // Calls submit with correct values
11    await user.click(button)
12    expect(mockSubmit).toHaveBeenCalledWith(
13      { testInput: "Input value" },
14      expect.anything()
15    )
16  })
```

4.6 Analýza výsledku a návrhy rozšíření

Implementovaný prototyp systému pro správu obsahu pokrývá základní požadovanou funkcionalitu, tedy možnosti konfigurace typů dat v databázi, administrační rozhraní poskytující možnosti práce s daty (CRUD operace) a poskytuje vývojářům rozhraní pro libovolnou práci s daty v rámci aplikace.

Vytvořená funkcionalita a vývojové rozhraní jsou pouze prototypem, nicméně pro vývoj opravdu malých prezentací a projektů by v aktuálním stavu bylo CMS možné použít, znamenalo by to ale pravděpodobně nutnost dalších rozšíření. Dále by bylo nutné pokrýt pokud možno většinu funkcionality unit testy, které jsem bohužel nestihl vytvořit, nicméně projekt je nastaven tak, aby nebyla nutná žádná další konfigurace.

Dalším problémem při použití implementovaného prototypu by byl samotný Next.js, respektive fakt, že použitý přístup s adresářem `app/` je prozatím pouze betaverze. Každá aktualizace knihovny tak potenciálně může představovat tzv. *breaking changes*, změny, které zásadním způsobem mění (a pravděpodobně rozbíjí) využití stávající funkcionality nebo rozhraní. Bylo by tedy nutné nejdříve počkat, až se vývoj projektu do určité míry ustálí.

Beta verze Next.js má dopad i na samotné nasazení projektu, jelikož při konfiguraci Firebase Hosting jsem narazil na chybu, která aktuálně znemožňuje využití dynamických cest v URL. Zkoušel jsem několik metod pro obejití této chyby, ale bohužel se mi nepodařilo projekt bez problémů nasadit. Dynamické cesty vždy vracejí chybu 404, ačkoliv zbytek CMS funguje.

4.6.1 Návrhy rozšíření prototypu

Jak jsem již zmínil, do budoucna by bylo nutné prototyp projektu výrazně rozšířit tak, aby podporoval větší množství případů použití. Zejména jde o vytvoření další funkcionality administračního rozhraní.

Díky struktuře projektu by neměl být problém rozšíření implementovat a zakomponovat do stávajícího prototypu.

Rozšíření administračního rozhraní

Tato část rozšíření by vylepšovala UX koncových uživatelů, ale také přidávala funkcionalitu, kterou aktuální prototyp nemá.

Prvním z nutných rozšíření by byla implementace a umožnění použití dalších formulářových polí pro typy dat. Příkladem může být pole pro zadání GPS souřadnic, které by aktuálně bylo možné simulovat pouze použitím dvou textových polí pro zeměpisnou délku a šířku. Dalším by mohl být tzv. *multiselect*, tedy volba z několika prvků s možností výběru více prvků.

Dalším ze zásadních rozšíření by byla implementace stránky pro detail dokumentu, zobrazující data jedné konkrétní položky databáze. Datové položky implementované v prototypu je možné snadno zobrazit v tabulce, v případě, že bychom ale chtěli například zobrazit zmíněné GPS souřadnice na mapce, už bychom pro tento účel potřebovali komplikovanější komponenty. Samozřejmě by dále bylo nutné tyto komponenty pro zobrazení položek implementovat a zpřístupnit vývojáři při konfiguraci.

Dále by větším rozšířením administračního rozhraní měla být stránka pro správu uživatelů, jelikož v prototypu jsem tuto funkcionalitu implementoval pouze využitím Firebase Authentication a jejího rozhraní pro přidání a odebrání uživatelů. Pro použití prototypu v projektech, kde je nutné mít pouze administrátory projektu, je tato implementace dostačující, ale v případě, že potřebujeme vytvářet účty pro klienty, už ne.

Jeden ze způsobů implementace stránky by mohl být následující:

- Implementovali bychom stránku pro správu uživatelů obdobnou jako pro správu obsahu (tabulka, CRUD operace). Přidání nového uživatele by bylo prostřednictvím zadání jeho e-mailové adresy, na kterou by se odeslala pozvánka, zároveň by se e-mail přidal do kolekce autorizovaných adres.
- Vytvořili bychom registrační formulář s e-mailem a heslem.
- Data by se odesílala na nový API endpoint, který by ověřil autorizaci dané adresy v databázi, čímž bychom omezili možnost vytváření účtů na pouze pozvané adresy. Při vytváření účtu by se přiřadil požadovaný scope (role).

Rozšíření funkcionality pro vývoj

Následující vylepšení by rozšiřovaly funkcionalitu pro vývoj a údržbu vyvíjených projektů:

- Přidání *hooků* pro provádění funkcí na základě událostí souvisejících s CRUD operacemi s daty. Například by bylo umožněno kaskádování vymazání vazeb mezi dokumenty. Tuto funkcionalitu by bylo možné přidat jak do konfigurace typů dat, tak i pro vestavěné typy jako jsou uživatelé.
- Umožnění tzv. *overrides* (tedy přepsání) funkcionality. To je u prototypu možné přímým zasahováním do kódu CMS, mělo by to být ale umožněno přívětivěji, tedy tak, aby kód CMS zůstal zachován, ale použila se vývojářem vytvořená funkce. U jazyka JavaScript toto není nijak obtížné, funkce mohou být přiřazené proměnným, které je jednoduché změnit na jinou implementaci. Problém by ale mohl nastat u správných typů takovýchto funkcí.
- Implementace *logování* chyb a událostí. Pro toto by bylo nutné přidat nový API endpoint, který by logoval události na serveru a zprávy by se zapisovaly ve službě Cloud Logging. V prototypu je služba automaticky aktivní, ale loguje pouze události na serveru, tzn. odchytává chyby pouze v API endpointech a zpracování stránek (kompilace, renderování atp.).
- Vylepšit způsob verzování CMS a instalaci nových verzí. Momentálně by bylo nutné ručně *patchovat* novou funkcionalitu, což není uživatelsky schůdné.
- Vytvořit nástroj, který by umožňoval jednoduše založit nový projekt CMS s výběrem předem připravené šablony (např. základní blog).
- Vzhledem ke způsobu konfigurace typů dat a následnému generování souborů s typy a funkcemi by mělo být možné v případě potřeby CMS rozšířit o generování API endpointů pro práci s daty. Tím by se mohlo potenciálně i rozšířit využití celého CMS, jelikož by dále bylo umožněno API využívat libovolnou další aplikací.

Kapitola 5
Závěr

Bakalářská práce popisuje tvorbu systému pro správu obsahu, se zaměřením na usnadnění a zrychlení vývoje webových aplikací a prezentací, s využitím technologií React a Next.js. Jejím hlavním cílem byla analýza existujících řešení, návrh systému a následná implementace prototypu. Analýzu jsem provedl, na základě ní vytvořil návrh systému a implementoval jeho prototyp, který je ve výsledném stavu možné použít jako základ pro vývoj aplikací. Dále jsem specifikoval návrhy pro další rozšíření prototypu tak, aby CMS bylo možné využívat pro vývoj větších produkčních aplikací.

V první a druhé kapitole praktické části práce popisuji analýzu zadání a existující řešení systémů pro správu obsahu. Jako předlohu pro návrh výsledného systému jsem zvolil a popsal projekt October CMS, který nejvíce odpovídá požadavkům zadání (usnadnění vývoje, atd.). Následně jsem vytvořil návrh řešení a funkční požadavky na systém, zejména týkající se administračního rozhraní.

V další kapitole charakterizuji konfiguraci nástrojů pro vývoj a automatizaci, nastavení CI/CD pipeline v platformě GitHub Actions a poté automatizaci nasazení implementovaného prototypu ve službě Firebase Hosting.

Dále popisuji detaily vývoje samotného prototypu systému, a to nejdříve implementaci možností konfigurace aplikace, typů dat a pomocných funkcí pro vývojáře. V druhé části pak píši o vývoji administračního rozhraní a jeho komponent, s využitím knihovny Material UI.

V páté kapitole píši o detailech automatizovaného testování prototypu, s cílem usnadnění následného vývoje projektu.

Na konci praktické části práce provádím analýzu vytvořeného prototypu systému a specifikuji návrhy, dle kterých by dále bylo vhodné pokračovat ve vývoji systému.

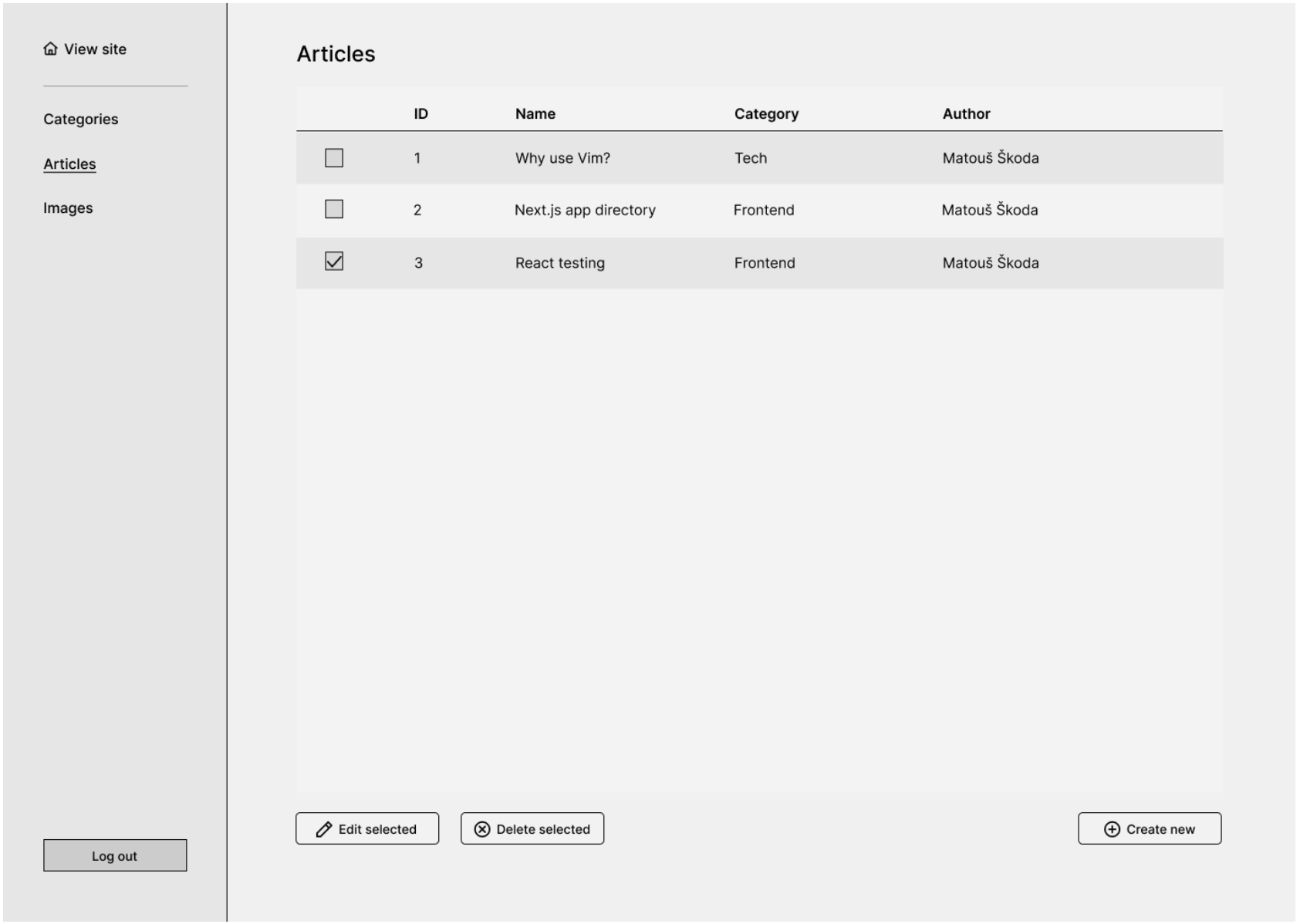
Jak jsem již popisoval v kapitole o analýze vytvořeného prototypu, v aktuálním stavu by CMS bylo možné použít jako základ pro vývoj aplikací, nicméně by to potenciálně mohlo být náročné. Důvodem pro to je probíhající vývoj zvolených technologií (zejména beta verze frameworku Next.js), které nejsou ještě ve stavu ustáleném pro produkční prostředí a vyžadovaly by časté úpravy projektu. Dále by bylo nutné obejít chybu při nasazení ve Firebase Hosting způsobenou beta verzí Next.js, nebo počkat na její opravu.

V případě ustálení vývoje zmíněných technologií a implementace navrhovaných rozšíření prototypu by výsledné CMS splňovalo všechny prvky zadání práce i požadavky vytyčené v návrhu řešení, dále by jej pak bylo možné využívat k výraznému zjednodušení vývoje aplikací v technologiích React a Next.js.

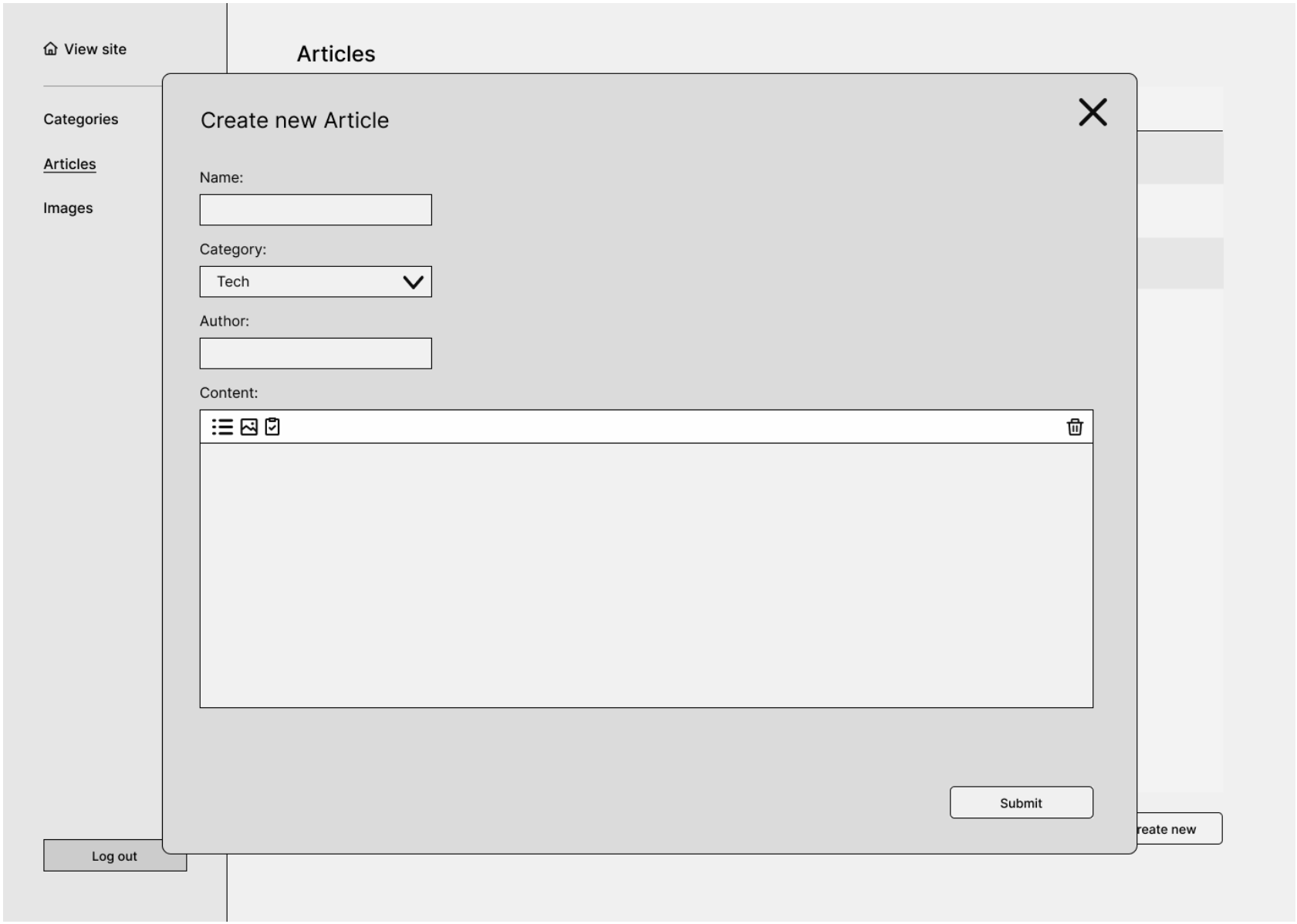
Příloha A

Příloha

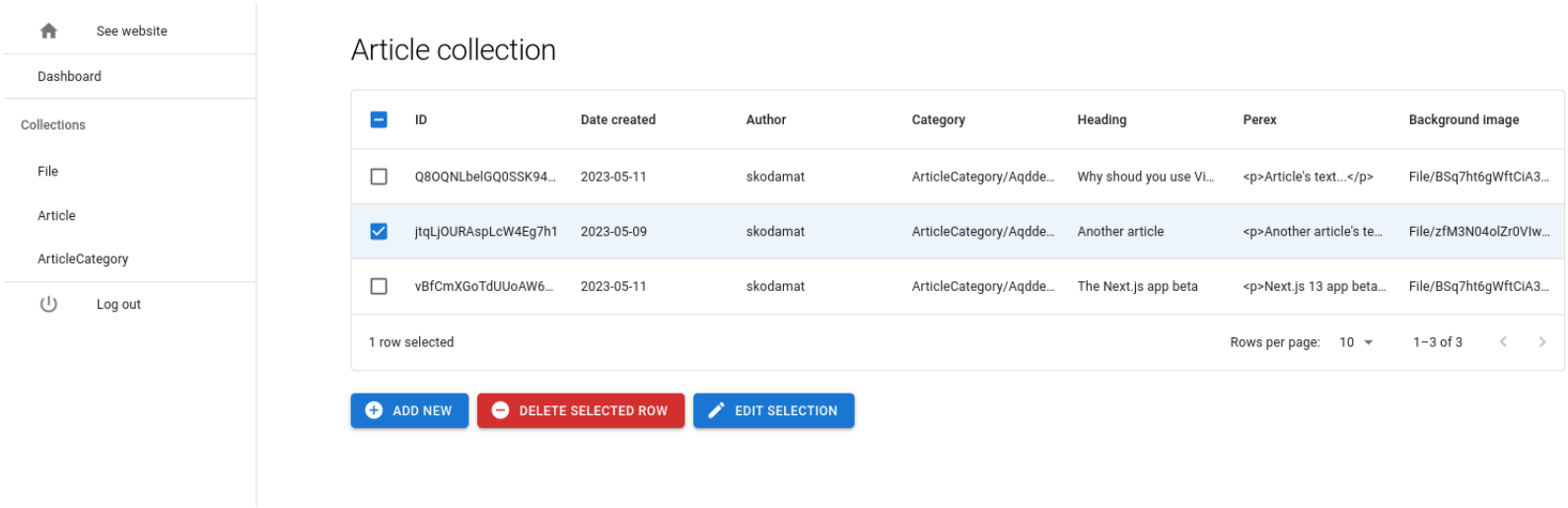
V této kapitole se nachází přílohy práce, zejména snímky implementace obrazovek CMS.



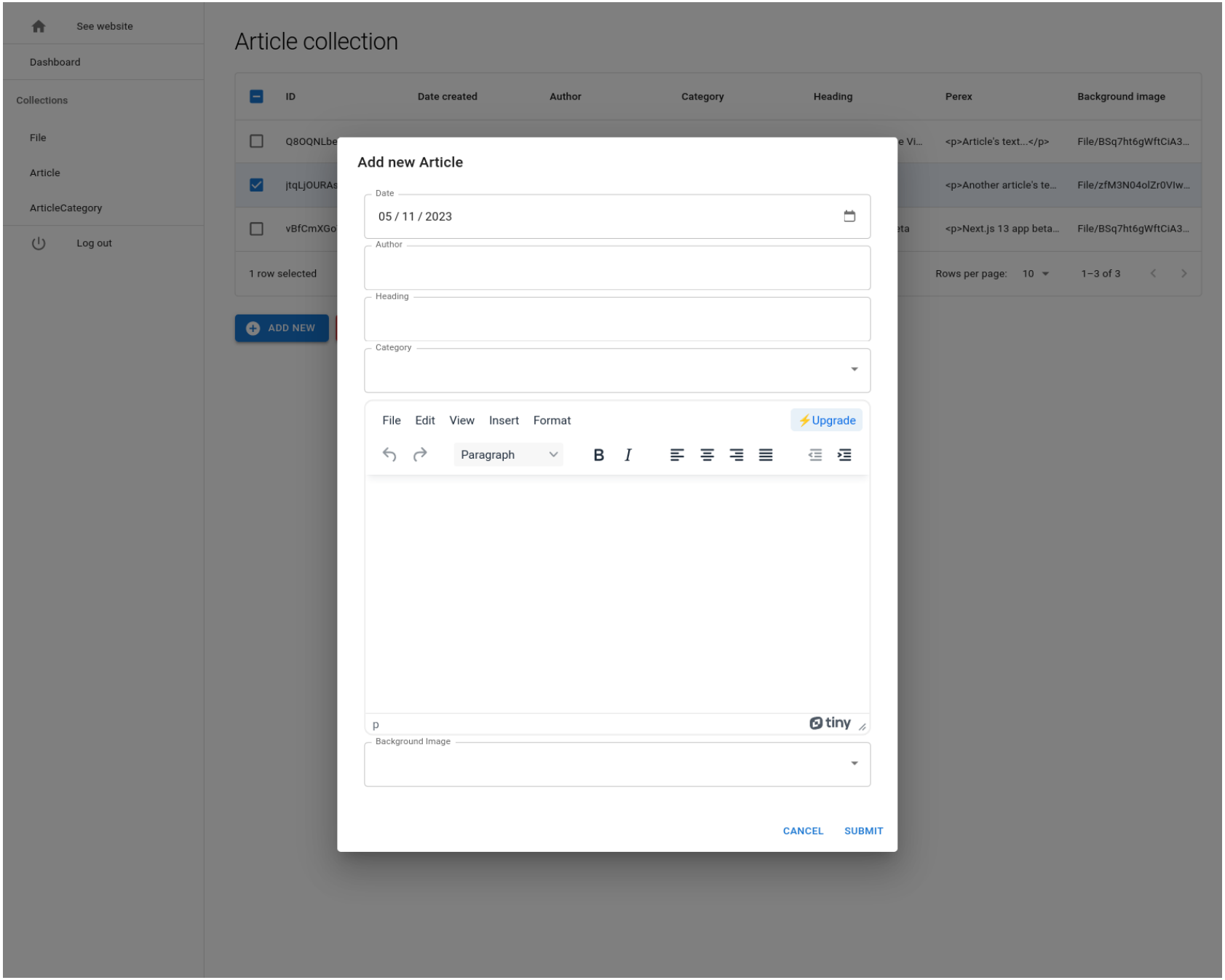
Obrázek A.1 Wireframe základní CMS stránky



Obrázek A.2 Wireframe stránky pro vytváření/editaci položky kolekce



Obrázek A.3 Snímek implementované generické stránky pro kolekce



Obrázek A.4 Snímek implementovaného dialogu pro vytvoření dokumentu

Bibliografie

1. CONTRIBUTORS, MDN. *HTML basics* [online]. Mozilla, 2023-02-24 [cit. 2023-04-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics.
2. MEIERT, Jens O. *On web semantics* [online]. Google LLC, 2012-07-16 [cit. 2023-04-24]. Dostupné z: https://developers.google.com/search/blog/2012/07/on-web-semantics.
3. CONTRIBUTORS, MDN. *What is CSS?* [online]. Mozilla, 2023-02-24 [cit. 2023-05-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS.
4. CONTRIBUTORS, MDN. *How CSS is structured* [online]. Mozilla, 2023-04-02 [cit. 2023-05-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_is_structured.
5. *CSS Preprocessors Explained* [online]. freeCodeCamp, 2020-01-17 [cit. 2023-05-03]. Dostupné z: https://www.freecodecamp.org/news/css-preprocessors/.
6. KANTOR, Ilya. *An Introduction to JavaScript* [online]. 2022-08-08. [cit. 2023-05-03]. Dostupné z: https://javascript.info/intro.
7. *TypeScript for the New Programmer* [online]. Microsoft, 2023-05-01 [cit. 2023-05-03]. Dostupné z: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html.
8. *React* [online]. Meta Open Source, 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://react.dev/.
9. *Writing Markup with JSX* [online]. Meta Open Source, 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://react.dev/learn/writing-markup-with-jsx.
10. *Built-in React Hooks* [online]. Meta Open Source, 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://react.dev/reference/react.
11. *Virtual DOM and Internals* [online]. Meta Platforms, Inc., 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://legacy.reactjs.org/docs/faq-internals.html.
12. KRAUSE, Stefan. *JS framework benchmark* [online]. 2023-04-25. [cit. 2023-04-25]. Dostupné z: https://github.com/krausest/js-framework-benchmark.
13. *What is Next.js?* [online]. Vercel, Inc., 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://nextjs.org/learn/foundations/about-nextjs/what-is-nextjs.
14. *Pre-rendering* [online]. Vercel, Inc., 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://nextjs.org/docs/basic-features/pages#pre-rendering.

15. *Routing* [online]. Vercel, Inc., 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://nextjs.org/docs/routing/introduction.
16. *Routing Fundamentals* [online]. Vercel, Inc., 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://beta.nextjs.org/docs/routing/fundamentals.
17. *Server and Client Components* [online]. Vercel, Inc., 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://beta.nextjs.org/docs/rendering/server-and-client-components.
18. *Material UI - Overview* [online]. Material UI SAS, 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://mui.com/material-ui/getting-started/overview/.
19. *Theming* [online]. Material UI SAS, 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://mui.com/material-ui/customization/theming/.
20. *How to customize* [online]. Material UI SAS, 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://mui.com/material-ui/customization/how-to-customize/.
21. *MUI X - Overview* [online]. Material UI SAS, 2023-05-03 [cit. 2023-05-03]. Dostupné z: https://mui.com/x/introduction/.
22. *About npm* [online]. npm, Inc., 2022-10-27 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/about-npm.
23. *npm* [online]. npm, Inc., 2023-01-12 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/cli/v9/commands/npm.
24. *package.json* [online]. npm, Inc., 2023-02-21 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/cli/v9/configuring-npm/package-json.
25. *scripts* [online]. npm, Inc., 2022-12-31 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/cli/v9/using-npm/scripts.
26. *npm-install* [online]. npm, Inc., 2022-10-05 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/cli/v9/commands/npm-install.
27. *npm-ci* [online]. npm, Inc., 2022-10-05 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/cli/v9/commands/npm-ci.
28. *npm-audit* [online]. npm, Inc., 2022-10-05 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/cli/v9/commands/npm-audit.
29. *npm-outdated* [online]. npm, Inc., 2022-10-05 [cit. 2023-05-03]. Dostupné z: https://docs.npmjs.com/cli/v9/commands/npm-outdated.
30. KANTOR, Ilya. *Coding Style* [online]. 2022-06-26. [cit. 2023-05-04]. Dostupné z: https://javascript.info/coding-style.
31. ESLINT CONTRIBUTORS. *Core Concepts* [online]. OpenJS Foundation, 2023-01-28 [cit. 2023-05-03]. Dostupné z: https://eslint.org/docs/latest/use/core-concepts.
32. ALEJANDRO DUSTET, Wil Hall. *TypeScript: Stop Using 'any', There's a Type For That* [online]. 2020-10-13. [cit. 2023-05-04]. Dostupné z: https://thoughtbot.com/blog/typescript-stop-using-any-there-s-a-type-for-that.
33. *What is Prettier?* [online]. prettier.io, 2022-04-13 [cit. 2023-05-03]. Dostupné z: https://prettier.io/docs/en/index.html.
34. *What is Prettier?* [online]. prettier.io, 2020-06-28 [cit. 2023-05-03]. Dostupné z: https://prettier.io/docs/en/why-prettier.html.
35. *Customizing Git - Git Hooks* [online]. git-scm.com, 2023-05-04 [cit. 2023-05-04]. Dostupné z: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks1.
36. TYPICODE. *What is Prettier?* [online]. 2023-05-04. [cit. 2023-05-04]. Dostupné z: https://typicode.github.io/husky/.

37. THE REFLECT TEAM. *Mapping the Testing Pyramid to Automated Testing Tools* [online]. Reflect Software Inc., 2020-05-07 [cit. 2023-04-30]. Dostupné z: https://reflect.run/articles/automated-testing-tools/.
38. *Jest* [online]. Meta Platforms, Inc., 2023-04-30 [cit. 2023-04-30]. Dostupné z: https://jestjs.io/.
39. BEKKHUS, Simen. *Mock Functions* [online]. Meta Platforms, Inc., 2023-04-06 [cit. 2023-04-30]. Dostupné z: https://jestjs.io/docs/mock-functions.
40. BEKKHUS, Simen. *Testing React Apps* [online]. Meta Platforms, Inc., 2023-04-06 [cit. 2023-04-30]. Dostupné z: https://jestjs.io/docs/tutorial-react.
41. *Why Cypress?* [online]. Cypress.io, 2023-04-30 [cit. 2023-04-30]. Dostupné z: https://docs.cypress.io/guides/overview/why-cypress.
42. *What is CI/CD?* [online]. Red Hat, Inc., 2022-05-11 [cit. 2023-04-29]. Dostupné z: https://www.redhat.com/en/topics/devops/what-is-ci-cd.
43. *Jenkins* [online]. 2023-04-29. [cit. 2023-04-29]. Dostupné z: https://www.jenkins.io/.
44. *GitLab CI/CD* [online]. GitLab B.V., 2023-04-29 [cit. 2023-04-29]. Dostupné z: https://docs.gitlab.com/ee/ci/.
45. *Understanding GitHub Actions* [online]. GitHub, Inc., 2023-04-29 [cit. 2023-04-29]. Dostupné z: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.
46. *What is Platform as a Service (PaaS)?* [online]. Google LLC, 2023-04-30 [cit. 2023-04-30]. Dostupné z: https://cloud.google.com/learn/what-is-paas.
47. *Why Google Cloud* [online]. Google LLC, 2023-04-30 [cit. 2023-04-30]. Dostupné z: https://cloud.google.com/why-google-cloud.
48. *Cloud Computing Services - Amazon Web Services (AWS)* [online]. Amazon.com, Inc., 2023-04-30 [cit. 2023-04-30]. Dostupné z: https://aws.amazon.com/.
49. KOGUT, Oliver Sejling. *How to Avoid Cloud Vendor Lock-In* [online]. plainenglish.io, 2021-11-09 [cit. 2023-04-30]. Dostupné z: https://aws.plainenglish.io/how-to-avoid-cloud-vendor-lock-in-1616b4932d11.
50. *Understand Firebase projects* [online]. Google LLC, 2023-04-27 [cit. 2023-04-30]. Dostupné z: https://firebase.google.com/docs/projects/learn-more.
51. *Firebase Pricing* [online]. Google LLC, 2023-04-27 [cit. 2023-04-30]. Dostupné z: https://firebase.google.com/pricing.
52. *Firebase Authentication* [online]. Google LLC, 2023-04-25 [cit. 2023-04-30]. Dostupné z: https://firebase.google.com/docs/auth.
53. *Cloud Firestore* [online]. Google LLC, 2023-04-27 [cit. 2023-04-30]. Dostupné z: https://firebase.google.com/docs/firestore.
54. *Cloud Storage* [online]. Google LLC, 2023-04-25 [cit. 2023-04-30]. Dostupné z: https://firebase.google.com/docs/storage.
55. *What is a content management system (CMS)?* [online]. Oracle Corporation, 2023-05-02 [cit. 2023-05-02]. Dostupné z: https://www.oracle.com/cz/content-management/what-is-cms/.

Obsah přiloženého média

- `src`
  - `implementation` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF