

Aplikace pro spolupráci studentů

Bc. Marek Novák



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Marek Novák
Osobní číslo: A23769
Studijní program: N0613A140022 Informační technologie
Specializace: Softwarové inženýrství
Forma studia: Kombinovaná
Téma práce: Aplikace pro spolupráci studentů
Téma práce anglicky: Application for Student Collaboration

Zásady pro vypracování

- Provedte průzkum trhu v oblasti stávajících nástrojů pro spolupráci a vytvořte souhrn jejich funkcí, výhod a nedostatků.
- V teoretické části stručně představte technologie, které budou využity k implementaci vlastního řešení nástroje pro spolupráci studentů.
- Navrhněte funkcionální a nefunkcionální požadavky pro vývoj aplikace a specifikujte jednotlivé procesy pomocí user stories či uživatelských scénářů.
- Vytvořte návrh uživatelského rozhraní aplikace, které poskytuje snadnou navigaci a intuitivní ovládání aplikace.
- Implementujte prototyp online kolaboračního nástroje s důrazem na responzivní uživatelské rozhraní, použitelnost a výkon.
- Otestujte aplikaci na různých typech zařízení tak, aby se zajistila kompatibilita s ohledem na responzivitě, uživatelskou přívětivost a celkovou použitelnost aplikace.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. BANKS, Alex a PORCELLO, Eve. *Learning react: functional web development with react and redux*. Sebastopol ; O'Reilly, 2017. ISBN 978-1-491-95462-1.
2. CHODOROW, Kristina. *MongoDB: the definitive guide*. 2nd ed. Sebastopol: O'Reilly, 2013. ISBN 978-1-449-34468-9.
3. VERCEL, INC. The React Framework for the Web. Online. C2024. Dostupné z: <https://nextjs.org/>. [cit. 2024-10-26].
4. META PLATFORMS. *React: The library for web and native user interfaces* [online]. c2023 [cit. 2024-10-26]. Dostupné z: <https://react.dev/>
5. MONGODB, INC. *MongoDB Documentation* [online]. c2023 [cit. 2024-10-26]. Dostupné z: <https://www.mongodb.com/docs/>
6. TIDWELL, Jenifer, Charles BREWER a Aynne VALENCIA. *Designing interfaces: patterns for effective interaction design*. Third edition. Beijing: O'Reilly, 2020. ISBN 978-149-2051-961.
7. RIVA, Michele. *Real-World Next.js: Build scalable, high-performance, and modern web applications using Next.js, the React framework for production*. Packt Publishing, 2022. ISBN 9781801073493.

Vedoucí diplomové práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**

Termín odevzdání diplomové práce: **2. června 2025**



doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

L.S.

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato diplomová práce se zabývá vývojem webové aplikace pro podporu spolupráce studentů při týmových projektech. Cílem je vytvořit nástroj, který umožní správu úkolů, sdílení poznámek a studijních materiálů. Teoretická část se věnuje webovým technologiím, zejména technologiím, jako je React, Next.js, TypeScript a MongoDB. Praktická část popisuje návrh, implementaci a testování aplikace. Výsledná aplikace je přizpůsobena potřebám studentů a podporuje efektivní týmovou práci ve studijním prostředí.

Klíčová slova: React, Next.js, TypeScript, vývoj webové aplikace, kolaborace

ABSTRACT

This thesis focuses on the development of a web application designed to support student collaboration on team projects. The goal is to create a tool that facilitates task management, note sharing, and access to study materials. The theoretical part discusses web technologies, especially tools such as React, Next.js, TypeScript, and MongoDB. The practical part describes the design, implementation, and testing of the application. The resulting application is tailored to student the needs and supports effective teamwork in an academic environment.

Keywords: React, Next.js, TypeScript, web application development, collaborationism

Tímto bych chtěl poděkovat panu Ing. Radku Valovi, Ph.D. za odborné vedení práce, přínosné rady a také ochotu.

OBSAH

ÚVOD.....	13
I TEORETICKÁ ČÁST.....	14
1 WEBOVÉ APLIKACE.....	15
1.1 ROZDÍL MEZI WEBOVOU APLIKACÍ A WEBOVOU STRÁNKOU.....	15
1.2 ARCHITEKTURA WEBOVÝCH APLIKACÍ.....	15
1.3 TECHNOLOGIE PRO VÝVOJ WEBOVÝCH APLIKACÍ	16
1.3.1 Frontend	16
1.3.2 HTML	16
1.3.3 CSS.....	17
1.3.4 JavaScript.....	18
1.3.5 Backend.....	18
1.4 DATABÁZE	19
1.5 RESTFUL APLIKAČNÍ PROGRAMOVÉ ROZHRANÍ	19
1.5.1 HTTP metody.....	20
1.5.2 Metoda GET.....	20
1.5.3 Metoda POST.....	20
1.5.4 Metoda PUT	21
1.5.5 Metoda PATCH	21
1.5.6 Metoda DELETE	22
1.5.7 API stavové kódy a jejich význam.....	22
1.5.8 2xx – Úspěšné stavové kódy	22
1.5.9 3xx – Stavové kódy pro přesměrování.....	23
1.5.10 4xx – Stavové kódy chyb na straně klienta.....	23
1.5.11 5xx – Stavové kódy chyb na straně serveru	24
2 POUŽITÉ TECHNOLOGIE	25
2.1 REACT.....	25
2.1.1 DOM vs VDOM.....	25
2.1.2 JSX	26
2.1.3 Komponenty	26
2.1.4 Životní cyklus komponent.....	27
2.1.5 Kdy dochází k renderování komponent	27
2.1.6 Předávání informací mezi komponenty – props.....	28
2.1.7 Stavy.....	29
2.1.8 React Hooks	29
2.1.9 useState	29
2.1.10 useEffect.....	30
2.1.11 useContext.....	30
2.1.12 useMemo	32
2.1.13 useCallback	32
2.1.14 useReducer	32
2.1.15 useRef.....	33
2.1.16 Optimalizace vykreslení komponent pomocí memo.....	34

2.2	NEXT.JS	35
2.2.1	Routing	35
2.2.2	Nested routing	36
2.2.3	Dynamic routes.....	36
2.2.4	Route handlers.....	37
2.2.5	Renderování.....	38
2.2.6	Serverové komponenty.....	38
2.2.7	Klientské komponenty.....	40
2.2.8	Kdy použít server a kdy klientské komponenty	40
2.2.9	Získávání dat	41
2.2.10	Server actions	43
2.3	TYPESCRIPT	44
2.4	MONGODB.....	46
II	PRAKTICKÁ ČÁST	47
3	IMPLEMENTACE KOLABORAČNÍHO NÁSTROJE PRO STUDENTY	48
3.1	EXISTUJÍCÍ NÁSTROJE A JEJICH ANALÝZA	48
3.1.1	Jira	48
3.1.2	Miro	49
3.1.3	Notion.....	49
3.2	POŽADAVKY NA IMPLEMENTACI.....	50
3.2.1	Funkcionální požadavky.....	50
3.2.2	Nefunkcionální požadavky.....	51
3.2.3	User stories	51
3.3	UŽIVATELSKÉ ROZHRANÍ.....	53
3.3.1	Layout – rozvržení stránky.....	53
3.3.2	Stránky.....	54
3.3.3	Komponenty	54
3.3.4	Návrhy uživatelského rozhraní pomocí Figma	54
3.4	IMPLEMENTACE APLIKACE.....	57
3.4.1	Struktura projektu.....	57
3.4.2	Material UI komponenty	59
3.4.3	Dialog	60
3.4.4	Formuláře	63
3.4.5	MongoDB	67
3.4.6	MongoDB schema.....	69
3.4.7	Server actions	70
3.4.8	Získávání dat a jejich zobrazení	72
3.4.9	Next Auth	74
3.4.10	Registrace nového uživatele.....	77
3.4.11	Bezpečné uchovávání hesel.....	79
3.4.12	Protected routes	80
3.4.13	Editor.....	80
3.4.14	Responzivita	84
3.4.15	Implementace úvodní webové stránky	87

3.5	KOMPLETNÍ UKÁZKA ROZHRANÍ APLIKACE	88
3.5.1	Přihlášení.....	88
3.5.2	Registrace.....	88
3.5.3	Přehled skupin.....	89
3.5.4	Tasks overview.....	90
3.5.5	Detailu úkolu.....	91
3.5.6	Wiki.....	93
3.5.7	Vytváření zdrojů.....	94
3.5.8	Správa členů	96
3.6	TESTOVÁNÍ A NASAZENÍ IMPLEMENTOVANÉ APLIKACE	97
3.6.1	Testování aplikace.....	97
3.6.2	Nasazení aplikace na Vercel	100
ZÁVĚR		102
SEZNAM POUŽITÉ LITERATURY.....		105
SEZNAM OBRÁZKŮ		110
SEZNAM TABULEK.....		111
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....		112
SEZNAM PŘÍLOH		113

ÚVOD

V současné době hraje efektivní spolupráce klíčovou roli, a to nejen v pracovním prostředí, ale také v tom akademickém. Na trhu je hned několik nástrojů v podobě webových aplikací, které fungují jako nástroje na kolaboraci. Mezi tyto nástroje patří například Jira, Miro nebo Notion. Ačkoliv jsou tyto nástroje užitečné, nejsou konkrétně mířeny na potřeby studentů, mohou být příliš komplexní či pro studenty cenově náročné.

Cílem této práce je navrhnout a implementovat webovou aplikaci, která slouží právě jako kolaborační nástroj pro studenty. Aplikace integruje osvědčené postupy z oblasti projektového řízení, jako je podrobný management úkolů, sdílení znalostí pomocí implementované wiki či rychlé poznámky. V teoretické části je poskytnut přehled o webovém vývoji se zaměřením na webové aplikace. Jsou zde popsány postupy či často použité technologie pro vývoj. Dále jsou rozebrány témata jako architektura aplikací, principy RESTful API, dále teoretická část rozebírá technologie použity pro vlastní implementaci kolaborační aplikace pro studenty. Mezi tyto technologie patří především React, Next.js, TypeScript či databáze MongoDB.

Praktická část práce zahrnuje popis procesu vývoje celé aplikace, a to od analýzy existujících řešení, přes definici funkčních a nefunkčních požadavků, následný návrh uživatelského rozhraní, až po samotnou implementaci a následné testování této aplikace. Analýza obsahuje informace o existujících řešení pro kolaboraci, shrnutí jejích výhod a nevýhod. Specifika funkčních a nefunkčních požadavků nastavuje jasnou definici funkcí a charakteristik, které by měla implementovaná aplikace splňovat. Návrh uživatelského rozhraní pak definuje základní designové prvky a pravidla aplikace, které jsou pak využity při implementaci aplikace. Implementace pak seznamuje s každou klíčovou částí, které jsou nutné pro funkčnost a pochopení aplikace. Aplikace je navržena a implementována tak, aby šla rozšířit či jinak modifikovat na základě rozšíření či změny požadavků. Díky těmto prvkům uživatelé (především studenti) mohou využívat postupy běžné v praxi – sdílení a úprava obsahu ve skupině, delegování úkolů a průběžně tak vyhodnocovat dosažené výsledky. Aplikace tak může studenty efektivně připravit na týmovou spolupráci v reálném pracovním prostředí již během studia.

I TEORETICKÁ ČÁST

1 WEBOVÉ APLIKACE

Webové aplikace dnes tvoří velkou část veškerého internetového obsahu. Tyto aplikace jsou uloženy na vzdáleném serveru a uživatelé k nim mohou přistupovat pomocí rozhraní webového prohlížeče, jako je například Chrome, Edge, Safari nebo Opera. Není tak nutná žádná instalace, jako je tomu u desktopových aplikací, které ke svému použití vyžadují instalaci přímo do zařízení uživatele.

1.1 Rozdíl mezi webovou aplikací a webovou stránkou

Ačkoliv jsou webové aplikace i webové stránky přístupné pomocí webového prohlížeče, jejich účel a architektura se výrazně liší. Zatímco webové stránky jsou především využívány k distribuci informací, webové aplikace jsou naopak zaměřeny na interakci s uživatelem a obsahem samotné aplikace. Tyto interakce mohou vykonávat požadavky, jako je ukládání dat, zpracování dat nebo přihlášení a registrace uživatele. Tradiční webové stránky tak můžeme popsat jako statické, což znamená, že jejich obsah se příliš nemění. Z tohoto pohledu jsou webové aplikace představují opak webových stránek, jejichž obsah je dynamický, tedy měnící se v průběhu používání této aplikace. [1]

Typickým příkladem webové stránky může být osobní portfolio, univerzitní stránky nebo případně webové stránky restaurace. Všechny tyto příklady mají obsahově informační charakter. Pokud se zaměříme na příklady webových aplikací, dostaneme například sociální sítě (*Facebook*, *YouTube*), emailové klienty (*Gmail*, *Outlook*) nebo aplikace zaměřené na řešení konkrétní problematiky, jako je *Jira* (projektový management) nebo *Canva* (tvorba grafických vizuálů). [1]

1.2 Architektura webových aplikací

Jak již bylo zmíněno, tak pro využívání online nástrojů v podobě webových aplikací je nutné, aby měl uživatel nainstalovaný webový prohlížeč. Pro správnou funkčnost těchto aplikací se však na pozadí skrývá mnohem více. Celá aplikace by tak šla rozdělit do tří základních částí. První z těchto částí a zároveň část, která je k uživateli nejbližší, se nazývá frontend. Frontend má za úkol vykreslení vizuálních prvků a interaktivních komponent v aplikaci. Dalo by se také říct, že je to jakýsi most mezi uživatelem a serverovou částí aplikace. Další část se říká backend, který tvoří základní část serveru. Slouží ke zpracování požadavků z klientské části (frontend) na serverovou část, jako je například přístup nebo

manipulace s daty. Třetí a poslední část aplikace tvoří databáze. Ta slouží k organizovanému ukládání dat, které jsou využívány napříč celou aplikací. [2]

1.3 Technologie pro vývoj webových aplikací

Výběr vhodných technologií pro vývoj webové aplikace je klíčovou součástí vývojového procesu a většinou se jedná o jednu z prvních součástí při plánování. Optimální výběr technologií nám může zajistit hned několik pozitivních aspektů projektu, a naopak eliminovat ty negativní, jako je rychlost celého vývojového procesu, výkon a škálovatelnost, náklady na vývoj, provoz a údržbu či čerpání dalších zdrojů, jako je vytíženost a množství vývojářů. Z tohoto důvodu je důležité zaměřením na klíčové technologie a přizpůsobit je tak, aby reflektovaly požadavky na aplikaci samotnou.

1.3.1 Frontend

Frontend je část aplikace, která slouží jako vrstva rozhraní obsahující prvky, jako jsou tlačítka, obrázky, texty a jiné komponenty, se kterými může uživatel provádět určitou řadu interakcí. Pokud je součástí vývoje také grafický návrh aplikace, frontend je tvořen tak, aby byl co nejvíce podobný předem definovanému návrhu a případně dalším aditivním pravidlům. Vizuální vrstva následně komunikuje s backendem, který zpracovává požadavky uživatele. Vývojáři samozřejmě neřeší pouze problematiku správné implementace designové vrstvy aplikace, ale také její výkon na straně klienta a bezpečnost. Výkonem se v tomto případě rozumí rychlost načítání částí aplikace, optimalizace pro konkrétní prohlížeče a zařízení nebo také rychlost zpracování požadavků, kde hraje roli nejen frontend, ale je nedílnou součástí. Je také důležité se zaměřit na zmíněnou bezpečnost, která by měla být brána v potaz napříč celou architekturou aplikace. U frontendu to může znamenat nejen obranu před kybernetickými útoky, ale také validace dat, které uživatel zasílá v rámci požadavku na server, aby se zachovala nejen bezpečnost, ale také konzistence dat, které jsou nutné pro správný chod aplikace. Ze všech těchto charakteristik vyplývá, že front-end je nedílnou součástí webových aplikací, jejich plynulý chod a uživatelský požitek z pohledu UX/UI implementace. [3]

1.3.2 HTML

HyperText Markup Language neboli HTML je základním stavebním prvkem veškerého webového obsahu, tedy i webových aplikací. Jedná se o značkovací jazyk, který definuje strukturu obsahu a popisuje obsah samotný. HTML využívá tzv. značky, které charakterizují

jednotlivé elementy, které jsou následně vykresleny díky webovému prohlížeči. Mezi tyto elementy patří například texty, které můžeme rozdělit do nadpisů či odstavců, tlačítka, textové pole, obrázky nebo jiný mediální obsah. Dále existují značky, které jsou použity pro seskupení těchto elementů nebo definování chování HTML souboru. Tyto značky mají pevně daný zápis, tedy jsou tvořeny pomocí názvu konkrétní značky, který je zaobalen do hranatých závorek. Tímto definujeme, co obsah znamená a jak ho má prohlížeč vnímat a vykreslit. Značky také dělíme na párové a nepárové. Párové jsou dvě stejné značky, kde jedna tvoří začátek a druhá konec elementu. Mezi tyto značky se následně umístí obsah. Níže se nachází příklad obou těchto variant. [4], [54]

```
// Párové značky
```

```
// Nadpis
```

```
<h1>Tohle je nadpis velikosti h1</h1>
```

```
// Odstavec
```

```
<p>Tato značka definuje odstavec s textem</p>
```

```
// Rozdělení a seskupení obsahu
```

```
<div>
```

```
<p>Odstavec s textem</p>
```

```
</div>
```

```
// Nepárové značky
```

```
// Obrázek
```

```
<img />
```

```
// Zalomení textu
```

```
<br />
```

1.3.3 CSS

Pro stylování elementů na stránce lze využít CSS, tedy *Cascading Style Sheets*. CSS nám umožňuje vytvářet nejen přívětivě vypadající elementy na stránce, ale také zajišťuje responzivitu, což je možnost adaptace designu pro různé typy rozlišení. Vývojáři tak navrhnu design aplikace, který se přizpůsobí jak na mobilní zařízení, tablety a počítače. Tohle je dnes nedílnou součástí a standardem při vývoji frontendu. Prohlížeč aplikuje na webový obsah základní sadu stylů, aby bylo možné i bez dalšího stylování rozpoznat jednotlivé HTML elementy. CSS styly lze aplikovat na jednotlivé elementy pomocí třech základních postupů. První z nich je přiřazení názvu třídy konkrétnímu elementu a následná definice stylů této třídy v odděleném souboru pro styly (soubory s koncovkou *.css*). Výhodou tohoto přístupu je znovu použitelnost stylů. Na každý element s touto třídou budou aplikovány styly této třídy. Dalším přístupem může být zvolení zápisu stylů přímo jako

atribut elementu nebo zápis stylů do hlavičky HTML dokumentu pomocí značek `<style>`. [5]

```
// Stylování pomocí tříd a typu HTML značky

// Třída
.MojeTrida {
    color: red;
    font-size: 15px;
}

// Stylování podle typu HTML značky, v tomto případě definice stylu pro všechny
nápisy typu h1
h1 {
    color: red;
    font-size: 15px;
}

// Stylování jako atribut elementu
<h1 style="color: red; font-size: 15px">Nadpis h1</h1>
```

1.3.4 JavaScript

JavaScript je skriptovací, ale také programovací jazyk, který umožňuje vytvářet komplexní funkce webu a webových aplikací. Pokud tedy stránka obsahuje dynamické aktualizace obsahu nebo například nějaké animace, jde o implementaci právě JavaScriptu. Jedná se tak o další vrstvu, která umožňuje vytvářet webový obsah. JavaScript je jedním z nejpopulárnějších programovacích jazyků, a to zejména díky frameworkům, jako je React, Angular nebo Vue.js. Tyto frameworky umožňují efektivní vývoj webových aplikací, a právě React je velkou součástí této práce, a proto je podrobněji popsán v dalších kapitolách. JavaScript nemusíme však využívat pouze při práci na webových aplikacích či stránkách, ale taky při vývoji mobilních aplikací, kde lze vyvíjet tyto aplikace pomocí JavaScriptu a další sady technologií, mezi které patří například React Native, Ionic nebo Capacitor. [6]

1.3.5 Backend

Je to část softwaru, který pracuje na straně serveru a se kterou uživatel komunikuje prostřednictvím frontendu. Backend se zaměřuje na implementaci logiky celé aplikace, databáze, *application programming interface (API)* nebo práci se serverem. Pro správnou implementaci backendu je důležité, aby vývojáři backendu úzce spolupracovali s týmem zaměřený na frontend. Backend se stará o požadavky z klienta a jejich zpracování. Mezi zpracování patří validace dat, modifikace dat nebo ukládání dat do databáze. Backend tak

nabízí možnost komunikaci s databází, což je další nedílnou součástí webových aplikací a samotné databáze jsou dále rozebrány v této práci. Výběr technologie pro implementaci backendu závisí na několika proměnných, jako jsou obecné požadavky na projekt, technologická náročnost projektu, škálovatelnost nebo také zkušenosti programátorů, kteří jsou součástí vývojového týmu. Pro vývoj backendu se dnes využívá mnoho programovacích jazyků a mezi ty populární patří například PHP, Python, JavaScript, TypeScript, Ruby, C# nebo Java. Dále se pro tyto programovací jazyky vybírá některý z frameworků, které jsou určené pro specifický jazyk. Příkladem mohou být Nette, Laravel, Node.js, Express.js, Django, Flask, Spring Boot, ASP.NET Core nebo .NET framework. [7]

1.4 Databáze

Databáze je strukturovaná a organizovaná kolekce dat, které jsou uloženy na nějakém počítači. Abychom mohli s databází pracovat, je potřeba zahrnout *database management system* (DBMS), který společně tvoří databázový systém. Data, která obsahují dnes běžné typy, se ukládají ve formě řádků a sloupců v sadě několika tabulek. Tyto data pak mohou být snadno zpřístupněny, modifikovány, aktualizovány nebo jinak organizovány. Většina databází využívá pro tyto úkony strukturovaný dotazovací jazyk SQL. Existuje několik typů databází a jejich výběr se liší dle požadavků na projekt. Mezi tyto typy patří například relační databáze, které jsou strukturovány na principu sloupců a řádků. Vyznačují se především tím, že obsahují a zprostředkovávají data, které mají mezi sebou nějakou vazbu. Objektově orientované databáze obsahují informace ve formě objektů, stejně jako je tomu u objektově orientovaného programování. NoSQL databáze neboli nerelační databáze jsou opakem relačních databází a umožňuje ukládat data ve flexibilnějším způsobem než pouze do tabulek, čímž umožňuje ukládat nestrukturované informace. [8], [9], [10]

1.5 RESTful Aplikační programové rozhraní

Toto rozhraní zajišťuje komunikaci mezi dvěma počítačovými systémy, čímž dochází k bezpečné výměně informací mezi nimi. Většina webových aplikací musí komunikovat s další externí aplikací nebo aplikací třetích stran k dosažení nějakých funkcí. Příkladem může být komunikace klientské části aplikace bankovníctví s interním systémem banky nebo také komunikace s internetovým obchodem při online nákupu, kdy je veškerý obsah vykreslován na základě komunikace se systémem, který například uchovává veškeré informace o produktech, zpřístupňuje platební brány a podobně. Aplikační programové

rozhraní (anglicky *Application programming interface*) tedy můžeme popsat jako sadu pravidel, které je nutné dodržet při komunikaci s dalšími softwarovými systémy. [11]

REST neboli anglicky *Representational State Transfer* je softwarová architektura, která stanovuje podmínky pro fungování API rozhraní. Vývojáři API mohou navrhnout takové API, které využívá různé typy architektury. API, které dodržují REST architekturu se přezdívá REST API. Webové služby implementující REST architekturu se anglicky říká *RESTful web services*. Co se týká terminologie týká, pojmy REST API a RESTful API se obecně používají zaměnitelně pro vyjádření API využívající architekturu REST a popisují stejnou problematiku. [11]

1.5.1 HTTP metody

HTTP metody určují typ akce, která se má provést na zdroji. Každá metoda je specifikuje konkrétní operaci, mezi které patří vytvoření, čtení, aktualizace nebo smazání zdroje a každá z těchto metod musí být součástí požadavku na REST API. Tyto operace se také označují zkratkou CRUD, která vychází z. anglického označení těchto metod, tedy *Create, Read, Update a Delete*. [12]

1.5.2 Metoda GET

Metoda GET se využívá k získání dat ze strany serveru. Klient může využít tuto metodu k získání jednoho nebo hned několika zdrojů. Při použití GET metody typicky nezasíláme serveru žádný obsah v těle samotného požadavku. Server nám po zpracování požadavku zašle buď požadované data nebo chybový stav, jejichž rozdělení je rozebráno dále v této práci. [12]

Jednoduchým příkladem může být endpoint pro získání všech uživatelů. Takový endpoint pro metodu GET by vypadal následovně: */users* a naopak endpoint, který vrátí konkrétního uživatele bude vytvořen ve formátu: */users/123*, kdy *123* označuje konkrétní *ID* zdroje, v tomto případě uživatele, který má server vrátit.

1.5.3 Metoda POST

Metodu POST využijeme v případě, že chceme vytvořit nový zdroj. Narozdíl od metody GET, metoda POST typicky vyžaduje zaslání těla požadavku, které obsahuje potřebné atributy, které se následně použijí pro vytvoření nového zdroje. Pro zachování integrity je dobré, aby tyto atributy procházely validací, která proběhne ještě před vytvořením tohoto

zdroje. V případě, že validace neproběhne úspěšně, server nepokračuje s akcí vytváření, ale naopak informuje klienta o chybě pomocí jednoho ze stavů charakterizující tuto chybu. [12]

Pokud si vezmeme zmíněný příklad pro metodu GET, kdy jsme pracovali s uživatelem, můžeme zde rovněž aplikovat metodu *POST*. Jestliže chceme vytvořit nového uživatele, využijeme endpoint ve formátu: */users* a doplníme požadavek o jeho tělo, které bude obsahovat atributy pro vytvoření nového uživatele. Mezi tyto atributy může patřit například *jméno*, *email* a *věk*.

```
#Tělo požadavku pro vytvoření uživatele obsahující potřebné atributy
{
  "name": "Jan Novák",
  "email": "jan.novak@email.cz",
  "age": 35
}
```

1.5.4 Metoda PUT

Metoda PUT se používá ke kompletnímu nahrazení existujícího zdroje, a to jeho aktualizovanou verzí. Metoda tak musí obsahovat tělo požadavku, kde se budou nacházet atributy pro tento nový zdroj. Pokud měl původní zdroj některé atributy, které nový zdroj neobsahuje, tyto atributy budou vymazány a nebudou ta pro novou verzi zdroje dostupné. [12]

Ukázkou tak může být aktualizace uživatele, který již je uložen na serveru. Endpoint pro tuto akci by mohl být vytvořen v této formě: */users/123*, kdy opět použijeme *123* jako *ID* uživatele. Pokud se v databázi nachází uživatel s tímto *ID* a atributy těla požadavku projdou případnou validací, bude tak původní zdroj nahrazen a dostaneme tak aktualizovaného uživatele.

1.5.5 Metoda PATCH

PATCH metoda zajišťuje aktualizací existujícího zdroje. Metoda je podobná předchozí metodě *PUT*, ale liší se v tom, že nenahrazuje celý zdroj, ale pouze jeho vybrané atributy bez nutnosti nahrazení zbývajících, které aktualizaci nevyžadují. [12]

Opět ukázkou může být aktualizace vybraných atributů existujícího uživatele. Stejně jako u metody PUT, i zde musíme zprostředkovat *ID* existujícího zdroje: */users/123*. Do těla požadavku poté doplníme požadované atributy k aktualizaci. Tělo požadavku tak nemusí

obsahovat všechny atributy zdroje, což zajišťuje větší flexibilitu a efektivitu než metoda PUT.

1.5.6 Metoda DELETE

Pro smazání zdroje zde existuje metoda DELETE, která zajišťuje, že dojde k úplnému odstranění zdroje z databáze. Před smazáním zdroje je vždy dobré se přesvědčit, že zdroj může být smazán, jelikož jeho odstranění je nenávratné. [12]

Pokud bychom tedy chtěli smazat konkrétního uživatele reprezentující požadovaný zdroj k odstranění, musíme opět zprostředkovat jeho *ID*: */users/123*. Pokud tento uživatel existuje a zdroj projde případnou validací pro kontrolu, zda může být smazán, je tento zdroj nenávratně odstraněn a server pouze zašle informaci o tom, jestli byla akce úspěšná či nikoliv.

1.5.7 API stavové kódy a jejich význam

HTTP stavové kódy jsou generovány na straně serveru a jsou zasílány jako odpověď na vytvořený požadavek ze strany klienta. Každý z těchto stavových kódů má specifický význam a klient na základě těchto stavů může informovat uživatele o každé akci, kterou uživatel provádí a která nějakým způsobem pracuje se serverem. Níže jsou rozebrány základní druhy těchto kódů. **Chyba! Nenalezen zdroj odkazů.**

1.5.8 2xx – Úspěšné stavové kódy

Při úspěšném zpracování požadavku na server je na stranu klienta zaslána odpověď s jedním z těchto stavových kódů, které se mohou lišit na základě typu požadavku. [13], [14]

Tabulka 1 Úspěšné stavové kódy

Kód	Stav	Popis
200	OK	Požadavek vyhodnocen jako úspěšný
201	Created	Úspěšný požadavek a nově vytvořený zdroj
202	Accepted	Požadavek přijat, ale nebyl zatím zpracován
204	No Content	Úspěšný požadavek s prázdným tělem odpovědi

(zdroj: vlastní zpracování)

1.5.9 3xx – Stavové kódy pro přesměrování

Pokud stavový kód začíná číslem 3, znamená to, že klient musí vynaložit další akci, aby byl požadavek zpracován. Většinou to znamená, že se zdroj nachází na jiném místě, byl změněn nebo jsou vyžadovány další kroky. [13], [14]

Tabulka 2 Přesměrování stavové kódy

Kód	Stav	Popis
301	Moved Permanently	URL požadovaného zdroje je permanentně změněna
302	Found	URL požadovaného zdroje je dočasně změněna
303	See Other	Požadovaný zdroj je na jiné URL adrese
304	Not Modified	Zdroj nebyl modifikován a lze využít ten co je v cache

(zdroj: vlastní zpracování)

1.5.10 4xx – Stavové kódy chyb na straně klienta

Ne všechny požadavky na server jsou zpracovány úspěšně. Pokud se jedná o chybu na straně klienta, tak jsou použity stavové kódy začínající číslem 4. Tyto chyby mohou vzniknout například tak, že požadavek neprojde validací nebo že k úspěšnému požadavku ne nutná autorizace. Nejčastěji používané kódy pro tyto chybové identifikátory se nachází v tabulce níže. [13], [14]

Tabulka 3 Chyba na straně klienta stavové kódy

Kód	Stav	Popis
400	Bad Request	Server nerozumí požadavku
401	Unauthorized	Požadavek vyžaduje ověření uživatele
403	Forbidden	Klient nemá přístup k dokončení požadavku
404	Not Found	Server nemohl najít požadovaný zdroj
405	Method Not Allowed	Server metodu zná, ale nelze použít pro požadovaný zdroj
406	Not Acceptable	Zdroj existuje v jiném než pořadovém formátu

409	Conflict	Požadavek nelze dokončit díky aktuálnímu stavu zdroje
410	Gone	Požadovaný zdroj již není dostupný
413	Request Entity Too Large	Požadavek je větší než limit definovaný na straně serveru
415	Unsupported Media Type	Tělo požadavku obsahuje data ve formátu, kterému server nerozumí

(zdroj: vlastní zpracování)

1.5.11 5xx – Stavové kódy chyb na straně serveru

Chyby však nemusí vznikat pouze na straně klienta, ale také na straně serveru. Tyto chyby je důležité odlišit od chyb klientských. Toto odlišení je důležité nejen při vývoji, kdy vývojáři zjistí, zda se jedná o špatnou implementaci frontendové části nebo backendové části aplikace, ale také pro informování uživatelů. Díky tomuto můžeme uživatele informovat o tom, zda se jedná o chybu zapříčiněnou jejich akcí, nebo chybou na straně serveru, na jehož opravu je třeba vyčkat. [13], [14]

Tabulka 4 Chyba na straně serveru stavové kódy

Kód	Stav	Popis
500	Internal Server Error	Server z neznámého nedokáže zpracovat požadavek
501	Not Implemented	HTTP metoda není se strany serveru implementována
502	Bad Gateway	Server fungující jako brána nebo proxy, obdržel chybnou odpověď
503	Service Unavailable	Požadavek nyní nelze serverem zpracovat

(zdroj: vlastní zpracování)

2 POUŽITÉ TECHNOLOGIE

Práce využívá řadu technologií a také postupů, které jsou popsány v následující kapitole. Všechny tyto technologie tvoří celek, díky kterému je možné vytvářet webové aplikace. Jak již bylo zmíněno, pro implementaci webových aplikací je vhodné, aby se pečlivě zvážil výběr technologií, a to hned dle několika kritérií. Při výběru technologií pro tuto práci byla klíčová následující kritéria: výběr moderních technologií, rychlost vývoje, škálovatelnost a možnost nasazení aplikace pro přístup veřejnosti.

2.1 React

React lze popsat jako JavaScriptovou knihovnu pro vytváření uživatelského rozhraní. Knihovna byla vytvořena v roce 2013 společností Meta, která je stále vlastníkem této technologie, a i nadále se stará o její vývoj a údržbu. Díky Reactu lze vytvořit jednostránkové (anglicky *single-page*) aplikace nebo také serverově renderovaných (anglicky *server-rendered*) aplikací s využitím frameworků, jako je Next.js nebo Remix. Celý princip Reactu je založen na tvorbě a efektivním využití komponent, které jsou základním stavebním blokem uživatelského rozhraní aplikací. React je závislý na dalších knihovnách, které napomáhají přidávat funkce do samotné aplikace, jako jsou například knihovny pro směrování, tedy možnosti přesměrovat uživatele mezi stránkami. [15]

2.1.1 DOM vs VDOM

DOM (anglicky *Document Object Model*) je programové rozhraní pro webový obsah. Jedná se o reprezentaci XML nebo HTML dokumentů ve stromové struktuře. V momentě, kdy se webový obsah načte, prohlížeč vytvoří DOM v podobně stromu, který obsahuje jednotlivé HTML elementy na stránce. Tyto elementy mohou být modifikovány nebo zpřístupněny pomocí JavaScriptu. Příkladem může být scénář, kdy chceme změnit text tlačítka. Tento proces by zahrnoval kroky, kdy právě pomocí JavaScriptu vybereme element tlačítka a aktualizujeme jeho text. React využívá VDOM, tedy virtuální DOM, který optimalizuje proces vykreslování elementů na stránce. Jedná se o odlehčenou verzi skutečného DOM, což umožňuje vývojářům provádět změny bez přímé manipulace skutečného DOM, který uživatel vidí na obrazovce. React následně rozhodne, jako změny musí provést na základě porovnání virtuálního DOM a klasického DOM a znovu vykresluje pouze ty části aplikace, které se liší. [16], [17]

2.1.2 JSX

JSX vychází spojení pojmů JavaScript a XML, což lze popsat jako rozšíření JavaScriptové syntaxe, což umožňuje psaní HTML a JavaScriptu v jednom dokumentu s příponou .jsx. Místo toho, abychom oddělovali UI, tedy značkovací jazyk v podobě HTML elementů od logiky do oddělených souborů, React umožňuje psaní tohoto kódu v jednom souboru a ponechat tak logiku pro konkrétní elementy u sebe. Díky tomuto vytvoříme tzv. komponent, které jsou dále rozepsány v další kapitole. Ačkoliv není použití JSX při práci s Reactem vyžadováno, dnes se tento princip vývoje stal standardem a málo kdy se setkáme s odlišným přístupem, kdy píšeme HTML a JavaScript do oddělených souborů. [18], [19]

2.1.3 Komponenty

Jak již bylo zmíněno, komponenty jsou základním stavebním blokem všech aplikací vytvořené právě pomocí Reactu. Komponenty jsou jednotlivé části uživatelského rozhraní, které obsahují logiku a HTML elementy. Komponenty mohou nabývat různých rozměrů, ať už se jedná o menší části aplikace, jako jsou tlačítka, tak části velké jako jsou například celé stránky obsahující hned několik vnořených komponent v podobě potomků. React komponenty jsou tedy jednoduše JavaScriptové funkce, které vrací HTML element. Níže můžeme vidět jednoduchou ukázkou komponenty tlačítka a následné využití této komponenty v jedné z dalších komponent, která tak reprezentuje rodiče vnořené komponenty, tedy potomka. [20]

```
const MyButton = () => {  
  const handleClick = () => { console.log("Button clicked"); };  
  
  return <button onClick={handleClick}>Button</button>;  
};  
  
export const ParentComponent = () => {  
  return (  
    <div>  
      <h1>Parent Component</h1>  
      <MyButton />  
    </div>  
  );  
};
```

Jak si můžeme všimnout, tak komponenta obsahující tlačítko, tedy `<MyButton />` začíná velkým písmenem, čímž můžeme v Reactu identifikovat komponenty. V Reactu vždy musí všechny komponenty začínat velkým písmenem, zatímco HTML tagy začínají písmenem malým.

2.1.4 Životní cyklus komponent

Všechny komponenty v aplikacích vytvořené pomocí Reactu mají životní cyklus. Tento životní cyklus můžeme popsat pomocí tří fází každé komponenty. Jedná se o *mounting*, *updating* a *unmounting*. [21]

Každou z těchto fází můžeme popsat následovně: [21]

- **Mounting:** fáze ve které je komponenta vytvořena a následně umístěna do příslušné části DOM objektu.
- **Updating:** tato fáze nastává, pokud je nutné komponentu znovu vykreslit na základě změny stavu nebo hodnotám (props), které jsou nutné pro správné fungování komponenty.
- **Unmounting:** konečná fáze každé z komponent a dá se popsat jako odstranění komponenty z DOM.

2.1.5 Kdy dochází k renderování komponent

Jak můžeme vidět v předchozí kapitole, tak ta naznačuje, že k renderování dochází ve dvou případech. K prvnímu vykreslení komponenty dochází při jejím prvním mountu do DOM, aby byla zobrazena uživateli. K dalšímu vykreslení této komponenty dojde v momentě, kdy se změní její stav, props, context nebo při explicitním volání `forceUpdate`. Stejně tak každý re-render rodičovské komponenty bez další optimalizace způsobí i render svých potomků. Ne vždy však potřebujeme, aby se komponenta vykreslila, jelikož v ní například ani nemuselo dojít k žádné změně a UI by i tak bylo stejné jak před opakovaným vykreslením. [22], [23]

Abychom těmto nevyžádaným vykreslením mohli předejít, můžeme komponentu uložit do mezipaměti pomocí `memo`, které pomocí mělkého porovnání (anglicky *shallow comparison*) kontroluje změny props (primitivní typy podle hodnoty, objekty/funkce podle reference). Pro zlepšení výkonu při práci s hodnotami nám React nabízí `useMemo` a pro zapamatování funkcí `useCallback`. Tato technika, nazývaná anglicky *memoizace*, optimalizuje výkonu aplikace, a to tak, že ukládá výsledek a zabraňuje zbytečnému přepočítávání nebo překreslování komponent. Můžeme však definovat pole závislostí (*dependency array*), které když se změní, tak dojde k novému přepočítání a hodnota či komponenta se může aktualizovat. Je však dobré podotknout, že *dependency array* porovnává každou položku také pouze pomocí *shallow comparison*, takže u primitivních typů porovná

podle hodnoty. Objekty, pole a funkce porovnává podle reference. Pokud se reference nezmění, funkce se nespustí, a to i v případě, že se uvnitř objektu změnila data. [22], [23]

- **Ukládání primitivních typů:** mezi primitivní typy patří například *number*, *string* nebo *boolean*. Tyto hodnoty jsou přiřazovány a předávány podle hodnoty, což znamená, že se vytváří kopie samotné hodnoty. Primitivní hodnoty jsou neměnné, takže jakákoliv změna vede k vytvoření nové hodnoty. [24]
- **Ukládání objektových typů:** do této kategorie patří například *object*, *array* nebo *function*. Při přiřazení nebo předání těchto hodnot se nepředává samotný objekt, ale pouze reference (ukazatel) na paměť, kde je skutečně uložena. V praxi to také znamená, že více proměnných může ukazovat na stejný objekt. [24]

2.1.6 Předávání informací mezi komponenty – props

Komponenty využívají pro komunikaci props (properties). Každá rodičovská komponenta může předávat informace všem svým potomkům právě pomocí props. Props mohou připomínat atributy z tradičního HTML, ale liší se v tom, že props mohou přijímat jakékoliv hodnoty nabízené JavaScriptem a to včetně objektů, polí nebo dokonce funkcí. Ukázkou může být opět komponenta tlačítka, které může přijímat hodnoty jako text tlačítka nebo samotnou funkci, která se má vykonat v případě kliknutí na tlačítko. [25]

```
type MyButtonProps = {
  onClick: () => void;
  label: string;
};

const MyButton = ({ onClick, label }: MyButtonProps) => {
  return <button onClick={onClick}>{label}</button>;
};

export const ParentComponent = () => {
  const handleClick = () => { console.log("Button clicked"); };
  return (
    <div>
      <h1>Parent Component</h1>
      <MyButton onClick={handleClick} label="Click me" />
    </div>
  );
};
```

Jak můžeme na ukázce vidět, jedná se o stejný typ komponenty jako na ukázce v kapitole popisující komponenty. Komponenta tlačítka se nám však změnila, a to tak, že nyní je schopna přijmout props v podobě `onClick` funkce a `label` reprezentující text tlačítka. Díky

tomuto se komponenta stává více flexibilní a to tak, že lze využít na více místech, a to zcela pro jiné účely. Komponenty jdou tímto přístupem rozšiřovat a stále zachovat jejich flexibilitu. V ukázce lze také vypořádat, že se změnil i způsob, jak rodičovská komponenta pracuje s tímto tlačítkem, tedy potomkem. Funkce, která byla dříve deklarována přímo v komponentě tlačítka se nyní přesunula do rodiče, kde se společně s textem tlačítka využije jako prop pro tuto komponentu. V ukázce si dále můžeme všimnout, že komponenta `<MyButton />` obsahuje také type `MyButtonProps`, což definuje typ všech props pro danou komponentu. Tento vzor je dobré využít pouze v případě, že pracujeme s TypeScriptem a nikoliv s čistým JavaScriptem. TypeScript je také součástí této práce a je dále popsán v jedné z dalších kapitol.

2.1.7 Stavy

Aplikace často potřebuje změnit jednu nebo více ze svých komponent a vykreslit tak jiný obsah. Příkladem může být psaní do textového vstupu formuláře nebo tlačítko, které po stisknutí změní nějaký obsah a nahradí ho novým. Komponenty si tak musí ukládat všechny tyto akce do paměti, které se říká stav. Stav tedy slouží pro uložení různých typů dat do paměti, čímž dojde k novému vykreslení této komponenty, a to na základě změny stavu. Každá komponenta může mít hned několik stavů a může tyto stavy dále předávat svým potomkům v podobě props. Tyto stavy lze vytvářet a manipulovat pomocí `useState`, který je součástí React Hooks popsaný v další kapitole. [26], [53]

2.1.8 React Hooks

Hooks umožňují využívat různé typy React funkcí přímo z komponent. Při jejich využití lze buď použít vestavěné funkce (*hooks*) poskytované přímo Reactem, nebo můžeme vytvořit vlastní znovupoužitelné funkce (anglicky *custom hooks*) v kombinaci s těmi existujícími. [27]

2.1.9 useState

`useState` je jedna ze základních funkcí, které React nabízí a který umožňuje komponentám zapamatovat si informace, jako je například vstup uživatele nebo změnit svůj stav podle určitých akcí, jako může být například kliknutí a tlačítko. Stavy se většinou inicializují s výchozí hodnotou a v případě, že využijeme TypeScript, určíme také typ, které state přijímá. State může přijímat primitivní typy, jako je string, number, boolean nebo array, ale

také vlastní definované typy a objekty. Je nutné podotknout, že komponenta využívající `useState` se znovu vykreslí v případě, že dojde ke změně stavu. [28]

```
const [counter, setCounter] = React.useState(0);
```

2.1.10 `useEffect`

Pro práci s vedlejšími efekty využíváme v komponentách `useEffect`. Tyto vedlejší efekty mohou být například získávání dat z API, manipulace DOM či práce s časovači. Tento hook lze také využít při všech fázích životního cyklu komponenty, tedy při `mounting`, `updating` a `unmount`. Můžeme tak specifikovat, co se má v každé fázi tohoto cyklu provést. Při fázi `mounting`, tedy prvním vykreslení komponenty se vždy provede akce v těle `useEffectu`. Akce se následně vykoná v případě, že `useEffect` obsahuje závislosti a jedna z nich se změní. Závislosti se v `useEffectu` definují jako pole. Pokud `useEffect` obsahuje prázdné pole závislostí, vykoná se kód v jeho těle pouze při prvním vykreslení komponenty, a naopak pokud `useEffect` neobsahuje pole vůbec, vykoná se při každém vykreslení. V případě, že `useEffect` vrací nějakou implementovanou logiku pomocí syntaxe `return`, je tato část kódu vykonána při každé fázi `unmount`, tedy při odstranění komponenty z DOM struktury. [29]

```
useEffect(() => {  
  console.log("Component mounted");  
  return () => {  
    console.log("Component unmounted");  
  };  
}, [dependencies]);
```

2.1.11 `useContext`

Abychom předešli tak zvanému *prop drillingu*, tedy přenášení props z jedné komponenty rodiče skrze několika potomků, můžeme využít `context`. `Context` implementujeme pomocí hooku `useContext`, který umožňuje komponentám přistupovat k datům uložené v `contextu`. Aby `context` fungoval a abychom mohli přistupovat k datům `contextu`, je nutné, abychom vytvořili také `provider`. `Provider` je komponenta, která zaobaluje aplikaci nebo část naší aplikace, která má mít přístup k datům `kontextu` a následně se stará o zprostředkování dat `kontextu`. `Provider` také většinou obsahuje funkce spojené s `contextem`. Jednoduchá implementace může vypadat následovně. [30]

```
// Definujeme interface pro User a UserContextValue
interface User {
  name: string;
  loggedIn: boolean;
}

interface UserContextValue {
  user: User | null;
  login: (username: string) => void;
  logout: () => void;
}

// Vytvoříme context
export const UserContext = createContext<UserContextValue |
undefined>(undefined);

interface UserProviderProps {
  children: ReactNode;
}

// Vytvoříme UserProvider
export const UserProvider: React.FC<UserProviderProps> = ({ children }) => {
  // Vytvoříme state pro user, který bude přístupný pomocí contextu
  const [user, setUser] = useState<User | null>(null);

  const login = (username: string) => {
    setUser({ name: username, loggedIn: true });
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <UserContext.Provider value={{ user, login, logout }}>
      {children}
    </UserContext.Provider>
  );
};
```

Pro použití tohoto contextu stačí importovat useContext přímo z Reactu. Takové použití může vypadat následovně.

```
const context = useContext(UserContext);
```

2.1.12 useMemo

useMemo je funkce (React hook), která ukládá mezivýsledek výpočtů za účelem optimalizace výkonu aplikace. Přepočítává hodnotu jen v momentě, kdy dojde ke změně jedné ze závislostí. Díky tomuto se vyhneme zbytečným výpočtům během opakovaného vykreslení komponent. Funkce useMemo se často využívá v kombinaci s memo, tedy komponentou s uloženými výsledky vykreslení, která přijímá právě hodnotu, u které dojde k přepočítávání jen na základě výsledku useMemo. Funkce stejně jako další React funkce určené na optimalizaci, je dobré využívat jen do jisté míry. Pokud máme hodnotu, která lze získat bez náročné operace, není dobré ji ukládat do mezipaměti, protože porovnávání samotné hodnoty vůči nově spočítané hodnotě může být náročnější než samotné přepočítání v průběhu vykreslování. [31]

```
const result = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

2.1.13 useCallback

useCallback je další funkce z řady optimalizačních funkcí Reactu. Naopak od useMemo, který vrací uloženou hodnotu, useCallback vrací celou funkci. Funkcionalita je podobná jako u useMemo, tedy dochází k nové inicializaci této funkce jen v momentě, kdy se změní alespoň jedna ze závislostí. Opět je dobré využívat useCallback společně s komponentou, jejíž výsledek vykreslení je uložen v paměti a která přijímá tuto funkci jako props, čímž se vyhneme zbytečnému opakovanému vykreslení. [32]

```
const increment = useCallback(() => {  
  setCount((prevCount) => prevCount + 1);  
}, []);
```

2.1.14 useReducer

Používá se pro správu stavu v komponentě pomocí reducer funkce. Je vhodný pro složitější logiku, kdy stav závisí na různých akcích, nebo také když je třeba zajistit předvídatelné změny stavu. Místo toho, aby se hodnoty stavu nastavovaly přímo, všechny změny se provádějí odesláním akcí do reduceru, který na základě typu akce vrací nový stav. Pro správnou implementaci nejdříve vytvoříme reducer, tedy funkci, které vezmou aktuální stav a akci, z čehož vrátí nový stav na základě typu vybrané akce. Pro použití těchto akcí použijeme dispatch. [33]

```
type State = {count: number};

type Action = {type: 'increment'} | {type: 'decrement'};

const reducer = (state: State, action: Action): State => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error('Unknown action type');
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({type: 'increment'})}></button>
      <button onClick={() => dispatch({type: 'decrement'})}></button>
    </div>
  )
};

export default Counter;
```

2.1.15 useRef

`useRef` je hook, díky kterému máme možnost získat referenci na nějaký element nebo hodnotu. Na rozdíl od `useState`, tento hook nespouští nové vykreslení komponent při své aktualizaci. Běžně a nejčastěji se používá právě pro přístup k elementům nacházejícím se na stránce v DOM struktuře. Díky tomuto se vyhneme používání `document.getElementById`, které nabízí čistý JavaScript. Díky získané referenci na objekt můžeme zjistit informace o něm, jako je například jeho výška nebo šířka. [34]

```
const MyComponent = () => {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleFocus = () => {
    if (inputRef.current) {
      inputRef.current.focus();
    }
  };

  return (
    <div>
      <h1>MyComponent</h1>
      <input type="text" ref={inputRef} />
      <button onClick={handleFocus}>Focus Input</button>
    </div>
  );
};

export default MyComponent;
```

2.1.16 Optimalizace vykreslení komponent pomocí memo

React nabízí nejen ukládání mezivýsledku hodnot a funkcí, ale také celých komponent. To znamená, že je možné přeskočit opakované vykreslení komponent, pokud to není nutné. Abychom docílili ukládání mezivýsledku komponenty, stačí ji zaobalit do memo, které lze importovat z Reactu. Jak již víme, tak k přerenderování komponenty dojde vždy, kdykoliv se změní její stav nebo kdykoliv se vyrenderuje rodičovská komponenta. Komponenta, která tedy využívá memo se tak renderuje jen v momentě, když se změní její props. V případě, že se rodičovská komponenta znovu vykreslí, tak potomek, který využívá memo porovná, zda se její props liší a v případě že ano, dojde k přerenderování. Pokud se však props nezmění, komponenta se nebude znovu renderovat a celý proces opakovaného vykreslení se tak přeskočí. Tímto docílíme toho, že nebudeme zbytečně znovu vykreslovat komponenty, kterých se změna netýká a nové vykreslení není nutné, protože by komponenta vypadala totožně. Je také dobré zmínit, že memo bychom měli využívat pouze pro komponenty, které mají “pure rendering logic“, což znamená, že komponenta vždy vrátí stejný výstup, pokud její props, stav a context zůstanou neměnné. V momentě, kdy použijeme memo, říkáme tím Reactu, že naše komponenta dodržuje toto pravidlo. [35], [36]

Uložení komponenty do paměti se často využívá společně s funkcemi useMemo a useCallback, aby nedocházelo k neustálému renderování. Je však dobré zvážit, zda je vůbec vhodné využívat memo pro některé komponenty, protože v některých případech nebude mít

tato funkce vliv na výkon. Může se například jednat o komponenty, které vždy dostanou odlišné props, a tak dojde vždy k novému vykreslení této komponenty i přestože využívá memo. Některé týmy vývojářů však nechtějí řešit, které komponenty ukládat do paměti a které ne, a proto zaobalují všechny komponenty do memo. Tímto se nijak nezhorší výkon aplikací, ale může dojít ke zbytečné komplexitě kódu. [35]

2.2 Next.js

Next.js je moderní *React framework* vytvořený společností Vercel. Lze ho využít pro vytváření full-stack (jak backend, tak frontend) webových aplikací. Při práci s Next.js se využívá React a jeho principy pro vytvoření komponent a celého rozhraní. Next.js se následně využije pro další funkce a optimalizaci, jak je popsáno v dalších kapitolách této práce. Next.js také automaticky konfiguruje potřebné nástroje pro React, jako bundlování, kompilace a podobně, což usnadňuje práci vývojářům, kteří se mohou namísto konfigurací soustředit na vývoj samotný. Mezi funkce tohoto frameworku patří renderování obsahu na straně serveru (anglicky *server components*), generování statického obsahu pomocí `generateStaticParams` (anglicky *static site generation*) a hybridní přístup renderování, který kombinuje serverové a klientské komponenty. Na rozdíl od tradičních aplikace tvořené pomocí Reactu, které se spoléhají především na renderování na straně klienta, umožňuje Next.js generování stránek a jejich obsahu již na straně serveru. Tím zlepšuje výkon, a dokonce zvyšuje optimalizaci pro vyhledavače (SEO). [37], [38], [39]

Kromě toho Next.js také podporuje řadu dalších funkcí, jako je dynamické směrování, vytváření API bez nutnosti backendového frameworku nebo jazyka, jako například .NET nebo Spring Boot. Získávání dat z API přímo v komponentách, které jsou renderovány na straně serveru nebo optimalizace multimédií, jako jsou obrázky nebo písma.

Díky těmto vlastnostem se Next.js stal velice populárním mezi vývojáři a rychle si našel cestu do několika projektů. Dnes se Next.js stává standardem při vývoji aplikací založené na Reactu a které cílí na výkon nebo škálovatelnost.

2.2.1 Routing

Next.js využívá tzv. *file-system based routing*, což znamená, že při práci můžeme použít složky a soubory k definování route. *Route* jsou vlastně stránky, na které se může uživatel dostat, respektive kam ho aplikace přesměruje. [40]

Stránka (anglicky *page*) je uživatelské rozhraní, které je vykresleno na určité routě. Pro vytvoření stránky stačí přidat nový soubor s příponou *.jsx* nebo *.tsx* uvnitř složky *app* a vytvořit *default export* pro tuto komponentu stránky. Například, pokud bychom chtěli vytvořit první stránku aplikace, na kterou se dostaneme pomocí lomítka v URL adrese hned za doménou, bude struktura projektu a složky *app* vypadat následovně. [41]



Obrázek 1 Vytvoření stránky pomocí App router

(zdroj: vlastní)

2.2.2 Nested routing

Nested route jsou URL adresy složené z několika segmentů. Příkladem může být stránka obsahující všechny produkty nějakého internetového obchodu. Abychom toho docílili, stačí vytvořit novou složku *products* a do ní opět vložit novou stránku. Taková cesta by mohla náš existující příklad rozšířit následovně. [41]



Obrázek 2 Nested routing

(zdroj: vlastní)

2.2.3 Dynamic routes

Dynamické routování je vhodné využít v momentě, kdy neznáme přesné názvy všech segmentů URL dopředu. Pro vytvoření dynamické URL stačí vytvořit jeden nebo více

dynamických segmentů této adresy. V praxi to funguje tak, že do názvu stránky použijeme identifikátor, který následně můžeme získat z URL. Tento identifikátor se píše do hranatých závorek a může vypadat například takhle: [id] nebo [slug]. Můžeme tak opět rozšířit náš příklad o získání produktu na základě jeho ID, které získáme pomocí dynamického segmentu URL adresy. [41]



Obrázek 3 Dynamic routing

(zdroj: vlastní)

2.2.4 Route handlers

Route handlers umožňují vytvářet požadavky na routy pro danou cestu pomocí API. Abychom mohli použít route handlers, je nutné, abychom je definovali v souboru s názvem `route.ts` nebo `.js` a to uvnitř složky `app`.

Http metody, které můžeme při práci s route handlers využít jsou následující: GET, POST, PUT, PATCH, DELETE, HEAD a OPTION. Pokud bude použita jiná metoda, která není podporována, Next.js vrátí odpověď *405 Method Not Allowed*.

Díky této funkci vytvářet požadavky získáváme možnost implementovat backend naší aplikace, a to pomocí stejné technologie, kterou tvoříme frontendovou část. Route handlers lze přirovnat k technologii Node.js, která rovněž umožňuje implementaci backendu. Právě možnost vytvářet fullstack aplikace pomocí jedné technologie podporují popularitu Next.js.

Jednoduchou ukázkou může být využití metody GET pro získání uživatele. Ve skutečné implementaci nějaké aplikace je samozřejmé, že bychom uživatele získávali z databáze, což je samozřejmě v případě Route handlers možné. [42]

```
export async function GET() {
  try {
    const data = {
      id: 1,
      name: "John Doe",
    };
    return new Response(JSON.stringify(data), { status: 200 });
  } catch {
    return new Response("Error fetching users", { status: 500 });
  }
}
```

2.2.5 Renderování

Renderování je proces, který konvertuje napsaný kód na uživatelské rozhraní. Next.js umožňuje vytvářet tzv. hybridní aplikace, kde jednotlivé části mohou být renderovány buď na straně serveru nebo na straně klienta.

2.2.6 Serverové komponenty

Renderování na straně serveru neboli již zmíněné SSR je princip renderování obsahu již na serveru. Pokaždé, když je na stránku vytvořený požadavek, je tato stránka s celým svým obsahem znovu renderována a následně zaslána na stranu klienta. Tento koncept má významný dopad na výkon webových aplikací a tím i na uživatelskou přívětivost. V kontextu Next.js hraje SSR klíčovou roli při vývoji aplikací zaměřené na výkon a také optimalizací SEO. [39]

Jak již bylo zmíněno, SSR je aplikován vždy, když je na stránku vytvořen požadavek. Tímto požadavkem začíná celý tento cyklus SSR obsahu. Uživatel vytvoří požadavek k získání webového obsahu, který se dostane až na serveru. Server tento požadavek zpracuje a zašle zpět celou stránku, a to včetně všech dat, které by se v opačném případě získávaly až na straně klienta. Tímto se snižuje doba, za kterou je stránka interaktivní, respektive čas, kdy na ni uživatel může provádět nějaké akce, jelikož celý obsah je již připravený ze strany serveru. [39]

V *Next.js* jsou stránky a komponenty ve výchozím nastavení renderovány na serveru, což znamená, že se *HTML* vygeneruje na straně serveru a je odesláno do prohlížeče. Pokud

chceme, aby byla komponenta renderování na straně klienta, musíme explicitně deklarovat direktivu *use client* v horní části dokumentu obsahující tyto komponentu. Tím zajistíme, že daná komponenta se vykreslí pouze na straně klienta. To je užitečné zejména u komponent, které využívají stav nebo efekty specifické pro klienta, například sloužící pro manipulaci s *DOM* nebo získávání dat přímo z klienta. [39]

Pro lepší pochopení se nyní podívejme na jednoduchou ukázkou SSR komponenty a její rozdíl oproti komponentě renderované až na straně klienta. V obou příkladech máme identický kód, který se však liší definicí *use client* v případě klientské komponenty. Jak můžeme vidět, obě ukázky obsahují `console.log`, tedy nějaký výpis textu do konzole. Může se zdát, že kromě renderování komponenty se také v konzoli prohlížeče bude nacházet již zmíněný výpis s obsahem *Hello world!*. Skutečně tomu tak je u komponenty, která se renderuje až na straně klienta, jelikož veškerý kód, který komponenta obsahuje, se vykoná až v prohlížeči. Naopak u komponenty renderované již na serveru zjistíme, že v konzoli prohlížeče se tento výpis nenachází. Je tomu právě proto, že tento výpis již proběhl na serveru a Next.js nám tak zaslal pouze zkompleťovanou stránku, kdy veškerá logika této komponenty již proběhla. Níže můžeme vidět ukázkou serverové komponenty. [39]

```
const MyComponent = () => {  
  console.log("Hello world!");  
  return (  
    <div>MyComponent</div>  
  )  
}  
export default MyComponent
```

Klientská komponenta má totožnou implementaci, jen obsahuje definici, že se jedná o klientskou komponentu pomocí *use client*.

"use client"

```
const MyComponent = () => {  
  console.log("Hello world!");  
  return (  
    <div>MyComponent</div>  
  )  
}
```

2.2.7 Klientské komponenty

Klientské komponenty umožňují vytvářet interaktivní obsah, jehož základní struktura je vykreslena na serveru, ale skutečný obsah se vykreslí až na straně klienta. Tyto komponenty mohou využívat React funkce jako `useState`, `useEffect` což znamená, že dokážou zprostředkovat okamžitou odezvu uživateli a aktualizovat své rozhraní. Dále je díky těmto komponentám možné využít tzv. Browser API, tedy přístup k funkcím prohlížeče jako geolokace nebo lokální paměť. Jak bylo zmíněno v předchozí kapitole, tak abychom definovali, zda se jedná o klientskou komponentu či nikoliv, stačí v horní části dokumentu (před jednotlivé `importy`) definovat *use client*. Na začátku této kapitoly byl zmíněn pojem předrenderování na serveru. Aby byl tento pojem a vlastně i celý proces klientský komponent podrobněji vysvětlen, níže se nachází jednotlivé kroky tohoto procesu. [43]

Kroky na straně serveru: [43]

1. React vykreslí serverové komponenty do speciálního formátu nazývaný *React Server Component Payload* neboli *RSC Payload*, který obsahuje reference na klientské komponenty.
2. Next.js použije tento *RSC Payload* a *JavaScriptové* instrukce klientských komponent pro vyrenderování *HTML* pro konkrétní stránku na serveru.

Kroky na straně klienta: [43]

1. Vyrenderované *HTML* je použito ihned pro zobrazení náhledu stránky.
2. *Payload serverových komponent Reactu* se používá k synchronizaci klientských a serverových komponent a k aktualizaci *DOM*.
3. *JavaScriptové* instrukce se používají k hydrataci klientských komponent a zajištění jejich interaktivity.

2.2.8 Kdy použít server a kdy klientské komponenty

Abychom při práci s Next.js získali veškeré jeho výhody, vhodná architektura aplikace je důležitým faktorem. Při implementaci jednotlivých komponent je dobré pečlivě promyslet vhodný přístup a vybrat tak optimální variantu na základě technických požadavků a požadavků na komponentu samotnou. Pro lepší orientaci se níže nachází tabulka, která slouží jako přehled případů užití pro serverové a klientské komponenty. [44]

Tabulka 5 Vhodné využití serverových a klientských komponent

Příklad použití	Serverové komponenty	Klientské komponenty
Získání dat	Ano	Ne
Přímý přístup ke zdrojům na serveru	Ano	Ne
Uchování senzitivních informací na serveru (<i>tokens, API klíče, atd.</i>)	Ano	Ne
Práce s náročnými závislostmi na serveru ke snížení velikosti <i>JavaScriptu</i> na straně klienta	Ano	Ne
Přidání interaktivity (<i>onClick, onChange, atd.</i>)	Ne	Ano
Použití stavů a efektů životních cyklů (<i>useState, useReducer, useEffect, atd.</i>)	Ne	Ano
Použití <i>API</i> prohlížeče (<i>lokální paměť, geolokace</i>)	Ne	Ano
Využití tzv. vlastních <i>hooks</i> , které jsou závislé na stavech, efektech nebo <i>API</i> prohlížeče.	Ne	Ano
Použití <i>React Class</i> komponent	Ne	Ano

(zdroj: vlastní zpracování)

2.2.9 Získávání dat

Získávání dat se liší dle toho, jakou verzi Next.js používáme a také podle toho, jestli používáme na našem projektu tzv. App router nebo Pages routes. Pokud tedy budeme pracovat s novější verzí Next.js, budeme používat App router, tedy trochu odlišný přístup směřování (anglicky *routing*) jako tomu je u starších verzí. Jelikož je v praktické části této práce využit přístup App router, bude dále popsán jen přístup získávání dat pro App router, a nikoliv pro Pages router. App router zajišťuje získávání dat přímo v React komponentách renderovaných na straně serveru. Tento přístup nám umožňuje zasílat méně JavaScriptu na stranu klienta, přičemž zachováme vykreslený obsah získaný ze serveru. Níže se můžeme podívat je jednoduchý příklad získávání dat. [45]

V této ukázce si můžeme všimnout, že získávání dat je nyní implementováno prostřednictvím samostatné asynchronní funkce. Funkce je dále volána přímo v rámci komponenty, kdy jsou data, které funkce `getData` vrací, přiřazena do proměnné `data`. Díky této proměnné můžeme získat obsah pro vykreslení na stránce. Abychom mohli využít funkci `getData`, museli jsme také z komponenty vytvořit komponentu asynchronní, což

umožňuje provedení asynchronních operací, jako je právě načítání dat. Je také nutné podotknout, že tento princip získání dat funguje pouze v případě, že použijeme serverovou komponentu. [45]

```
import { Data } from "@models/data";

interface MyPageProps {
  data: Data;
}

async function getData() {
  const response = await fetch("https://api.example.com/data");
  const data: Data = await response.json();
  return data;
}

const MyPage: FC<MyPageProps> = async () => {
  const data = await getData();
  return (
    <div>
      <h1>{data.title}</h1>
    </div>
  );
};
```

Získávání dat přímo na serveru se také bude lišit v případě, že budeme využívat ORM nebo databázi. Pokud vytvoříme stejnou ukázkou, jako v předchozím příkladě, ale nyní s použitím ORM. Opět se jedná o serverovou komponentu, takže na stranu klienta se dostane pouze již vykreslený obsah, a to včetně veškerých dat, které jsou potřebné. [45]

```
import { db, posts } from '@lib/db'

export default async function Page() {
  const allPosts = await db.select().from(posts)
  return (
    <ul>
      {allPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

Mohou však nastat také případy, kdy budeme potřebovat získání dat na straně klienta. Dokumentace Next.js však nejdříve doporučuje vyzkoušet přístup získání dat na straně serveru. Pokud jsme však v situaci, kdy toto není možné nebo vhodné, můžeme využít klientskou komponentu a přístupy k získání dat, jako jsme zvyklí v Reactu. Můžeme například využít fetch v kombinaci s useEffect. Tohle však není doporučeno, takže je lepší

zvolit jednu z React knihoven, jako je třeba Tanstack query. Pro jednoduchou ukázkou však můžeme použít useEffect. Opět si v ukázce můžeme všimnout použití React hooks, jako je useState nebo useEffect, které lze využít pouze v klientských komponentách. Pokud bychom chtěli oddělit logiku získávání dat od komponenty, a to například s použitím vlastních hooks, opět bychom museli z komponenty vytvořit klientskou. Příkladem, kdy můžeme chtít využít tradiční získání dat v klientských komponentách oproti získání dat na serveru, může být ten, kdy komponenta obsahuje nějaké interaktivní prvky nebo kdy nechceme, aby server znovu vykresloval celou stránku, ale chceme aktualizovat pouze část stránky, která obsahuje tyto získaná data. [45]

```
'use client'

import { useState, useEffect } from 'react'

export default function Data() {
  const [allData, setData] = useState(null)

  useEffect(() => {
    async function fetchData() {
      const res = await fetch('http://localhost:3000/api/user')
      const data = await res.json()
      setData(data)
    }
    fetchData()
  }, [])

  if (!allData) return <div>Loading...</div>

  return (
    <ul>
      {allData.map((item) => (
        <li key={item.id}>
          {item.name}
        </li>
      ))}
    </ul>
  )
}
```

2.2.10 Server actions

Server actions jsou asynchronní funkce, které se provádějí na straně serveru. Mohou být však volány jak v serverových, tak klientských komponentách, a to například k mutaci dat nebo při práci s formuláři. Abychom z funkce vytvořili *server action*, stačí deklarovat *use server* v těle této funkce nebo v případě, že je funkce v samotném souboru, stačí využít

stejnou deklaraci v horní části souboru, čímž zajistíme, že všechny funkce exportované z tohoto souboru budou definovány jako server actions. Server actions následně využíváme jako jakékoliv jiné funkce a můžeme je také předávat mezi komponenty v podobě props. [46]

```
"use server"
```

```
export async function create() {  
  // Mutace dat  
}
```

Použití této funkce v případě komponenty tlačítka, by mohlo vypadat například následovně.

```
"use client";
```

```
import { create } from "@actions/actions";  
  
const Button = () => {  
  return <button onClick={create}>Create</button>;  
};  
  
export default Button;
```

2.3 TypeScript

TypeScript je open-source programovací jazyk vyvíjený firmou Microsoft, který je superset (nadstavba) jazyka JavaScript, který ho rozšiřuje o volitelné statické typování pomocí definování typů a zároveň zachovává kompatibilitu s knihovnamí JavaScriptu. Díky transpilaci TypeScriptu na JavaScript je zachována možnost spuštění v JavaScriptových prostředích, webových prohlížečů nebo Node.js. Velkou výhodnou použitím TypeScriptu je odhalení chyb již během vývoje, a nikoliv až za běhu aplikace, což celkově napomáhá zachování integrity, čitelnosti a udržitelnosti kódu. [47], [48]

- **Staticky typované jazyky:** Typy jsou známy a kontrolovány při kompilaci (*compile-time*), takže chyby v typech se odhalí před spuštěním aplikace. Pro některé programovací jazyky to znamená, že musíme specifikovat co za datový typ daná proměnná vlastně je. [49]
- **Dynamicky typované jazyky:** typová kontrola probíhá až za běhu programu (*runtime*) a proměnné mohou měnit svůj typ za běhu programu. Není nutné specifikovat typ při deklaraci proměnné, ale jazyk dokáže sám detekovat o jaký typ se jedná. [49]

Ačkoliv jsou typy v TypeScriptu volitelné, je dobré je využít, aby se předešlo chybám v implementaci. TypeScript obsahuje několik základních typů, které jsou popsány v tabulkách níže. [50]

Tabulka 6 TypeScript – přehled primitivních datových typů

Typ	Definice
number	Číselný datový typ zahrnující celí i desetinná čísla
string	Sekvence znaků
boolean	Datový typ pro hodnoty true/false
null	Záměrně prázdná nebo neexistující hodnota
undefined	Reprezentuje hodnotu, která není inicializována
symbol	Unikátní a neměnná hodnota, často jako klíč objektu

(zdroj: vlastní zpracování)

Tabulka 7 TypeScript – přehled objektových datových typů

Typ	Definice
Object	Reprezentuje libovolný nepřimitivní typ
Array	Reprezentuje kolekci prvků určitého typu
Tuple	Reprezentuje pole s pevně daným počtem prvků specifických typů
Enum	Reprezentuje sadu pojmenovaných konstant
Function	Volitelná entita, může definovat typ parametrů a návratovou hodnotu
Class	Definuje šablonu pro vytváření objektů s konkrétními vlastnostmi a metodami
Interface	Struktura objektu

(zdroj: vlastní zpracování)

2.4 MongoDB

MongoDB je NoSQL databáze navržená pro ukládání velkého objemu dat, které mají různorodé typy. Na rozdíl od tradičních relačních databází, které mají strukturu tabulek a řádků, MongoDB ukládá data do flexibilních dokumentů, které připomínají formát JSON. Díky tomuto je tento typ databáze ideální pro aplikace, které mohou vyžadovat škálovatelnost, a právě zmíněnou flexibilitu. Příkladem dokumentu v této databázi může být následující dokument uloženého uživatele. [51], [52]

```
#Dokument uloženého uživatele v MongoDB
{
  _id: "678fb970405f87b94b6c2c46",
  firstName: "Jon",
  lastName: "Doe",
  address: {
    street: "1 Circle Rd",
    city: "Los Angeles",
    state: "CA",
    zip: "90404"
  }
}
```

Dokumenty v této nerelační databázi můžeme vnímat stejně jako řádky tabulek v databázích relačních. Tyto dokumenty jsou pak uloženy v kolekcích (anglicky *collections*), které jsou právě analogií pro tabulky vyskytující se v relačních databázích. Příkladem kolekce tak mohou být všechny dokumenty ukládající data uživatelů, které dohromady spadají pod kolekci s názvem *Users*, nebo podobného jmenného ekvivalentu, které je možné zvolit. [52]

II PRAKTICKÁ ČÁST

3 IMPLEMENTACE KOLABORAČNÍHO NÁSTROJE PRO STUDENTY

V praktické části této práce je popsán kompletní proces vývoje kolaboračního nástroje pro studenty. Mezi tyto procesy patří stručná analýza existujících nástrojů, návrh funkčních a nefunkčních požadavků, návrh uživatelského rozhraní a samotná implementace této aplikace. Samotná aplikace využívá praktiky z firemních postupů, které lze také aplikovat na spolupráci mezi studenty a která může potenciálně zlepšit kolaboraci a produktivitu studentů na společných projektech. Mezi tyto procesy patří například plánování a rozdělení úkolů, rychlé poznámky a tvorba vlastního obsahu ve formě sdílení znalostí a zdrojů.

3.1 Existující nástroje a jejich analýza

Na trhu již existuje hned několik nástrojů, které se zaměřují na spolupráci, produktivitu a správu projektů. Avšak většina těchto nástrojů nejsou specificky navrženy právě pro studenty, ale spíše pro firmy. Zmíněné nástroje mohou být pro studenty příliš komplexní, a proto se svým obsahem stávají pro studenty nerelevantní a studenti zkrátka nevyužijí veškeré funkce těchto aplikací. Níže jsou rozebrány tyto existující nástroje, jejich výhody a případné nedostatky.

3.1.1 Jira

Jira je poměrně robustní nástroj co se funkcí a změřením týká. Tento nástroj vznikl díky firmě Atlassian a jeho primární zaměření je týmový vývoj softwaru. Nabízí funkce jako je plánování sprintů, správu backlogu či podrobné reportování. Jira také umožňuje týmům sledovat chyby, spravovat úkoly či závislosti těchto úkolů a monitorovat pokrok v celém životním cyklu softwarového vývoje. Vývoj software však není jediné zaměření Jiry a lze ji také využít pro týmy zaměřené na marketing nebo design.

Výhody:

- Přizpůsobitelný dle potřeb týmu
- Podrobné reportování
- Funguje v rámci ekosystému Atlassian, který nabízí další kolaborační nástroje

Nevýhody:

- Komplexní nastavení pro nové uživatele a menší týmy

- S rostoucím počtem uživatelů může růst také cena za služby Jiry
- Pro studenty či menší týmy může Jira obsahovat nepotřebné funkce

3.1.2 Miro

Miro je další nástroj na týmovou kolaboraci se zaměřením na vizuální reprezentaci informací. Uživatelé si v této aplikaci mohou vytvářet vizuální obsah ve formě sticky notes (nalepovacích štítků), které jsou umístěny na canvas (plátno, tabule). Firmy a další týmy tento nástroj také využívají na vytváření diagramů, kolaboraci v jednom prostředí v reálném čase, takže všichni připojení uživatelé vidí, co ostatní dělají na stejném plátně.

Výhody:

- Intuitivní i pro nové uživatele
- Řešení pro vizuální reprezentaci informací
- Kolaborace v reálném čase

Nevýhody:

- Může být nevhodné pro uživatele, kteří hledají jednoduchý nástroj pro správu projektu
- Mimo vizuální reprezentaci informací nenabízí žádné další klíčové funkce

3.1.3 Notion

Notion je velice populární nástroj, a to především u studentů. K této popularitě pomohlo to, že Notion obsahuje hned několik kolaboračních funkcí, které by jinak uživatel musel nahrazovat kombinací několika odlišných nástrojů. Mezi tyto nástroje patří kombinace textového editoru, správa úkolů, kalendář či plánování. Je především využíván právě jako textový editor pro vytváření textového obsahu, na kterém mohou uživatelé kolaborovat.

Výhody:

- Využití předpřipravených šablon pro vytvářený obsah
- Jednoduché použití ať už pro jednotlivce nebo větší týmy
- Kombinace několika nástrojů do jedné platformy

Nevýhody:

- Obsahuje klávesové zkratky pro ovládání prostředí a obsahu, které mohou pro uživatele matoucí nebo nevhodné
- Uživatelé mohou od tohoto nástroje odejít ke konkurenci díky špatně nastavené struktuře prostředí, díky čemuž se obsah může stát matoucí a nepřehledný

3.2 Požadavky na implementaci

V následující kapitole jsou rozebrány funkční a nefunkční požadavky implementované aplikace se zaměřením na kolaboraci mezi studenty. Tyto požadavky jsou následně využity při definici a implementaci jednotlivých funkcí, která by měla aplikace splňovat.

3.2.1 Funkcionální požadavky

Definují, co by měla implementovaná aplikace umět. Popisují specifické funkce a chování aplikace. V následující tabulce je uvedeno několik těchto požadavků na implementovanou aplikaci.

Tabulka 8 Přehled funkcionálních požadavků

ID	Název	Popis
<i>FP-01</i>	Registrace	Uživatel se může registrovat pomocí emailu a hesla
<i>FP-02</i>	Přihlášení	Uživatel se může přihlásit pomocí emailu a hesla
<i>FP-03</i>	Vytvoření skupiny	Uživatel může založit novou skupinu
<i>FP-04</i>	Správa skupin	Admin skupiny ji může editovat
<i>FP-05</i>	Pozvání uživatelů do skupiny	Admin skupiny může pozvat či odebrat členy skupiny
<i>FP-06</i>	Vytváření úkolů	V rámci skupiny může uživatel vytvářet úkoly
<i>FP-07</i>	Přiřazení úkolu	Uživatelé mohou mezi sebou přiřazovat úkoly ostatním členům skupiny
<i>FP-08</i>	Editace úkolů	Uživatelé mohou editovat či mazat úkoly
<i>FP-09</i>	Komentáře úkolů	Každý uživatel má možnost přidat komentář ke konkrétnímu úkolu
<i>FP-10</i>	Stav úkolů	Uživatelé mohou měnit stav úkolů

<i>FP-11</i>	Rychlé poznámky	Uživatelé mohou vytvářet rychlé poznámky
<i>FP-12</i>	Editace poznámek	Uživatelé mohou spravovat rychlé poznámky
<i>FP-13</i>	Sdílení znalostí	Možnost sdílení znalostí formou skupinové Wiki
<i>FP-14</i>	Sdílení odkazů	Uživatelé mohou sdílet užitečné odkazy
<i>FP-15</i>	Textový editor	Uživatelé mohou tvořit obsah pomocí textového editoru

(zdroj: vlastní zpracování)

3.2.2 Nefunkcionální požadavky

Požadavky definující, jak by se měla implementovaná aplikace chovat. Tyto požadavky mohou pokrývat aspekty aplikace jako je výkon, bezpečnost, škálovatelnost či použitelnost.

Tabulka 9 Přehled nefunkcionálních požadavků

ID	Název	Popis
<i>NP-01</i>	Design	Intuitivní design se zaměřením na uživatelskou přívětivost
<i>NP-02</i>	SSR	Aplikace bude využívat serverové komponenty díky Next.js
<i>NP-03</i>	Škálovatelnost	aplikace bude připravena pro případnou škálovatelnost, a to nejen z hlediska uživatelského rozhraní, ale také z hlediska implementace kódu a celé struktury projektu
<i>NP-04</i>	Validace	Aplikace bude validovat veškeré vstupy uživatele
<i>NP-05</i>	Bezpečnost	Aplikace bude obsahovat autentifikaci za cílem zachování integrity celého systému
<i>NP-06</i>	Kompatibilita	Aplikace bude plně responzivní pro všechna zařízení

(zdroj: vlastní zpracování)

3.2.3 User stories

User stories je sada krátkých popisů funkcí a funkcionalit v rámci aplikace, které jsou popsány z perspektivy koncového uživatele. Tyto popisy jsou pro implementovanou aplikaci popsány formou tabulky níže.

Registrace a přihlášení:

- Jako uživatel se chci registrovat pomocí emailu a hesla za cílem bezpečného přístupu do aplikace.
- Jako uživatel se chci přihlásit pomocí emailu a hesla za cílem přístupu do aplikace

Vytváření a správa skupin:

- Jako uživatel chci vytvořit novou skupinu za cílem kolaborace s ostatními uživateli
- Jako admin skupiny chci pozvat další členy do této skupiny
- Jako admin skupiny chci spravovat členy této skupiny
- Jako admin skupiny chci spravovat název a popis skupiny s možností odstranit skupinu celkově

Vytváření a správa úkolů:

- Jako uživatel chci vytvářet úkoly v konkrétní skupině a možnost spravovat tento úkol
- Jako uživatel chci přiřadit úkol konkrétnímu uživateli s možností nastavení data splnění, obtížnosti a stavu úkolu
- Jako uživatel chci přidat komentář ke konkrétnímu úkolu
- Jako uživatel chci přiřadit užitečné zdroje k úkolu

Sticky notes:

- Jako uživatel chci vytvářet sticky notes za účelem rychlého zápisu krátkých informací
- Jako uživatel chci editovat a odstranit tyto sticky notes

Wiki:

- Jako uživatel chci vytvořit složky na stránce wiki pro sdílení znalostí a informací
- Jako uživatel chci editovat a odstranit složky na stránce wiki
- Jako uživatel chci vytvářet dokumenty uvnitř složky na stránce wiki
- Jako uživatel chci editovat a odstranit dokumenty uvnitř složek
- Jako uživatel chci možnost vytvořit dokument se zdroji ve formě odkazů a spravovat tyto odkazy

- Jako uživatel chci vytvářet obsah dokumentu v textové formě pomocí textového editoru

3.3 Uživatelské rozhraní

Při návrhu uživatelského rozhraní byl kladen důraz především na uživatelskou přívětivost (*UX, user experience*) a splnění požadavků na aplikaci samotnou. Cílem návrhu uživatelského rozhraní nebylo vytvořit návrh pro všechny části aplikace, ale vytvořit strukturovanou, a především vizuální sadu pravidel pro implementaci aplikace. Byla tak navržena sada znovupoužitelných komponent, které se následně využívají napříč celou aplikací a tvoří tak základní prvky UI. Následně byla navržena řada několika klíčových stránek, od kterých se následně opět odvíjí návrh a rozvržení zbytku aplikace. Pro vytvoření těchto návrhů byla zvolena aplikace Figma, díky které je možné vytvářet uživatelské rozhraní a obohatit tyto návrhy o prototypování.

Návrhy můžeme rozdělit do tří sekcí, které společně tvoří aplikaci. Jedná se o *layout*, tedy rozvržení samotné aplikace, kde bylo cílem vytvořit komponenty, které jsou využity globálně napříč aplikací a tvoří strukturu stránky, která je neměnná čímž se zachová vizuální konzistence a podporuje uživatelskou přívětivost. Další sekcí těchto návrhů můžeme kategorizovat jako stránky samotné. Stránky jsou potomkem rozvržení a obsahují jednotlivé komponenty, které ve výsledku tvoří obsah. Poslední sekcí jsou zmíněné komponenty, mezi které patří různé typy, ať už se jedná od interaktivní komponenty typu tlačítek, vstupních polí nebo statické komponenty typu nadpisy, popisy, dialogy až po komplexní návrhy rozhraní, jako jsou například formuláře, které jsou tvořeny z několika menších komponent.

3.3.1 Layout – rozvržení stránky

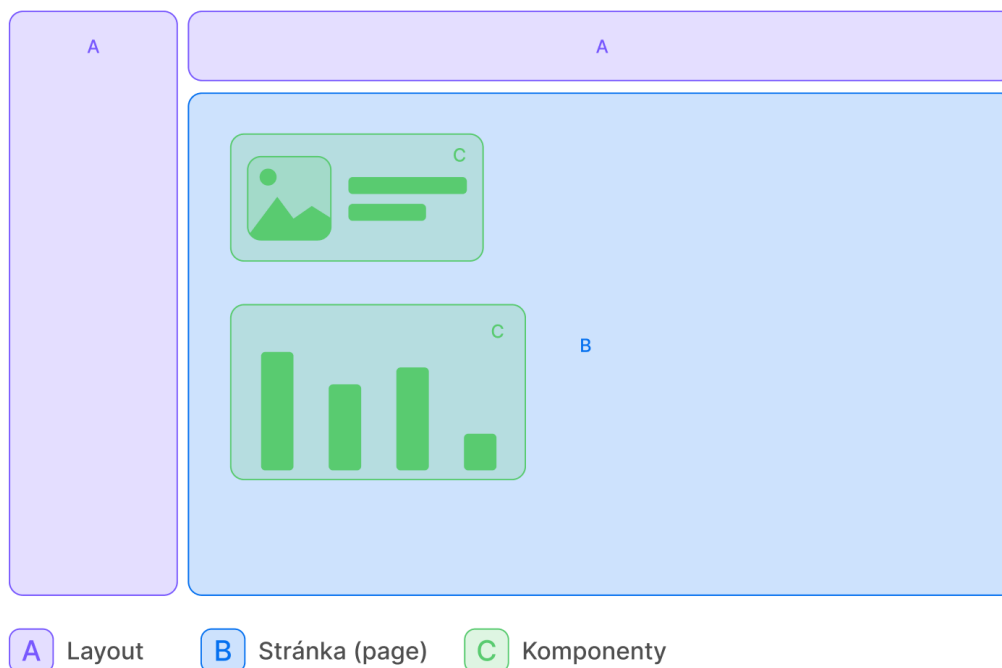
V rámci rozvržení stránky bylo vytvořeno hned několik komponent, které jsou ve struktuře celého DOM stromu nejvýše. Tyto komponenty je možné vidět na každé stránce, jelikož mezi ně patří boční navigace či lišta, která je umístěna v horní části obrazovky a která obsahuje další komponenty, jako je například vstup pro výběr skupiny uživatele či informace o aktuálně přihlášeném uživateli, jako je jméno a email.

3.3.2 Stránky

Aplikace obsahuje řadu stránek, které jsou umístěny přímo v rozložení aplikace. Tyto stránky jsou z hlediska designu poměrně jednoduché a jejich cílem je správné zobrazení komponent v požadované struktuře. Slouží tedy jako rodič pro komponenty a vymezuje tak dynamický obsah od toho globálního, kdy globálním obsahem je myšleno již zmíněné rozložení, které je stejné napříč celou aplikací.

3.3.3 Komponenty

Komponenty jsou navrženy tak, aby bylo možné je implementovat na straně frontendu, kde se využívá knihovna Material UI. V rámci designu jsou navrženy základní komponenty, které jsou potřebné pro implementaci uživatelského rozhraní aplikace a design těchto komponent je navržen tak, aby bylo možné ho modifikovat či případně celý design systém rozšířit o další komponenty s korespondujícím designem.



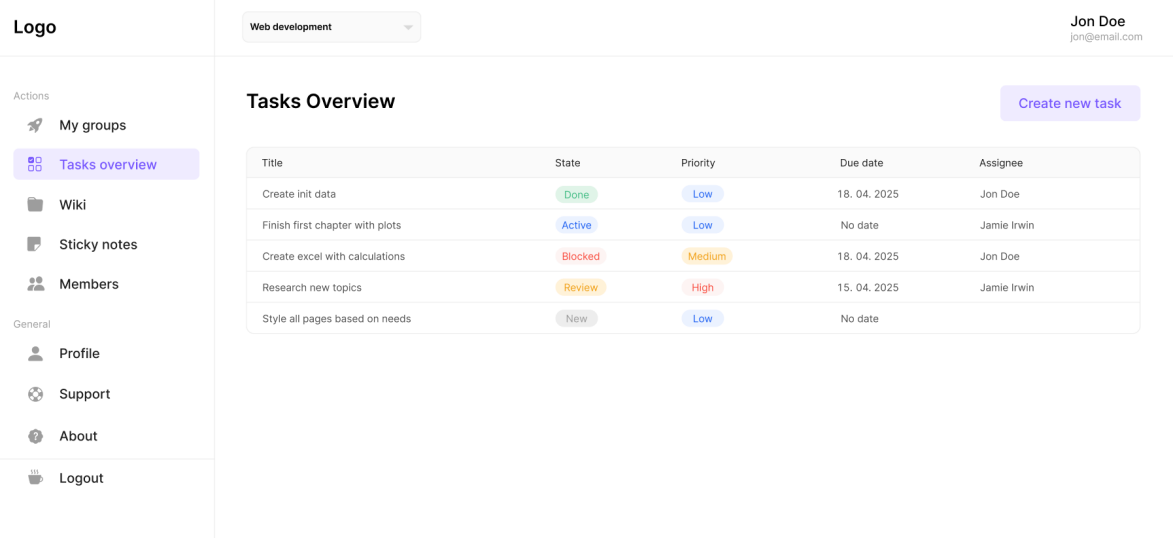
Obrázek 4 Vizuální ukázky struktury aplikace

(zdroj: vlastní)

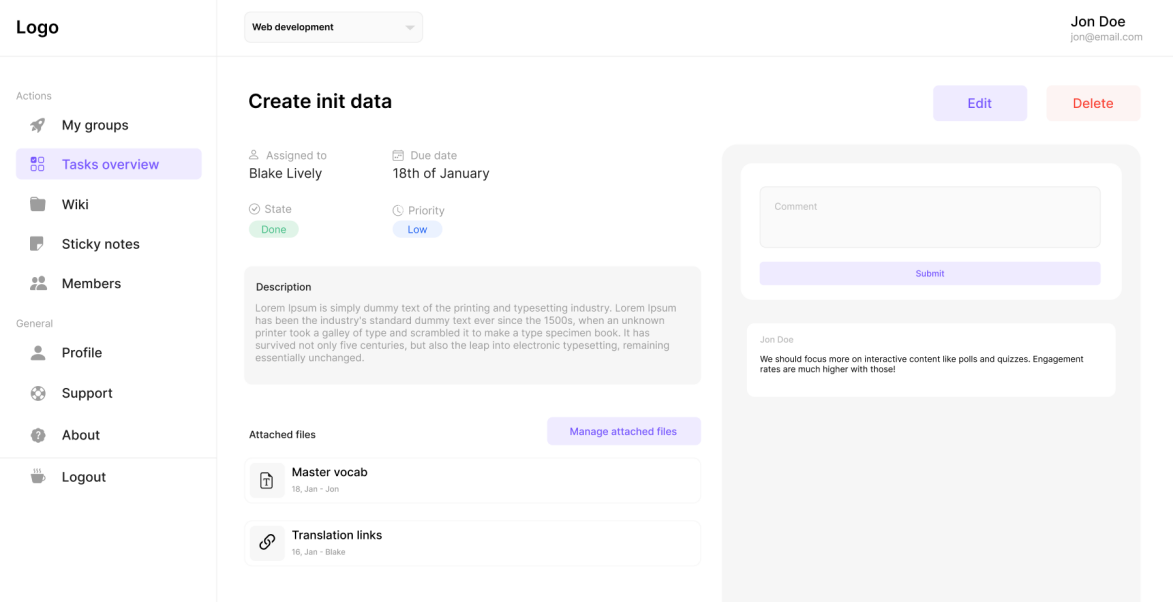
3.3.4 Návrhy uživatelského rozhraní pomocí Figma

V této části jsou ukázky návrhů uživatelského rozhraní, kterým se pak řídí implementace skutečné aplikace. Tyto návrhy neslouží jako kompletní příručka pro všechny části aplikace,

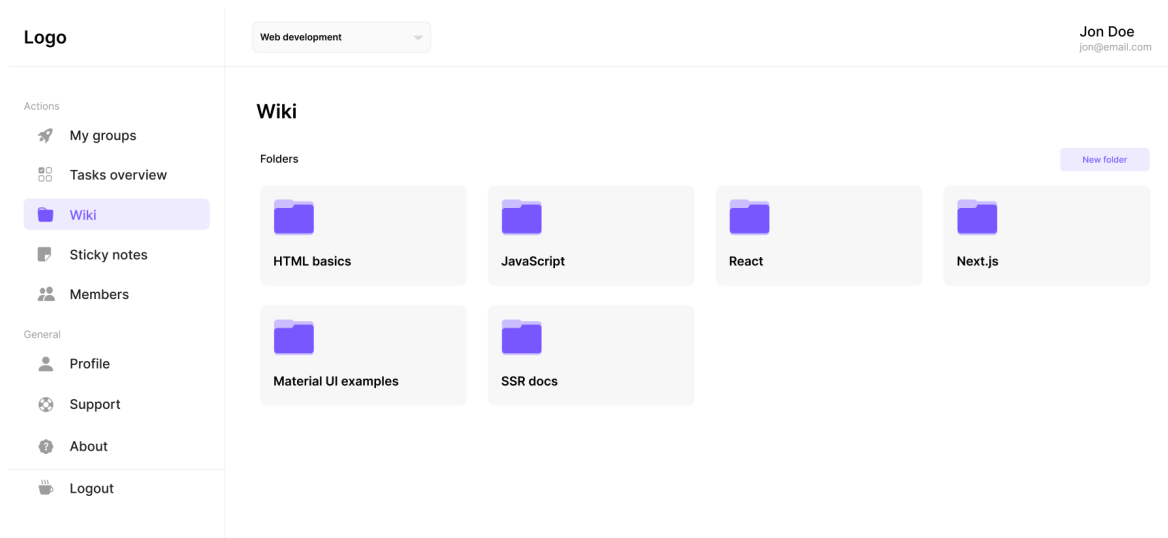
ale pro určení směru, co se týká základního rozhraní aplikace, kterým se pak řídí implementace částí, které nejsou navrženy.



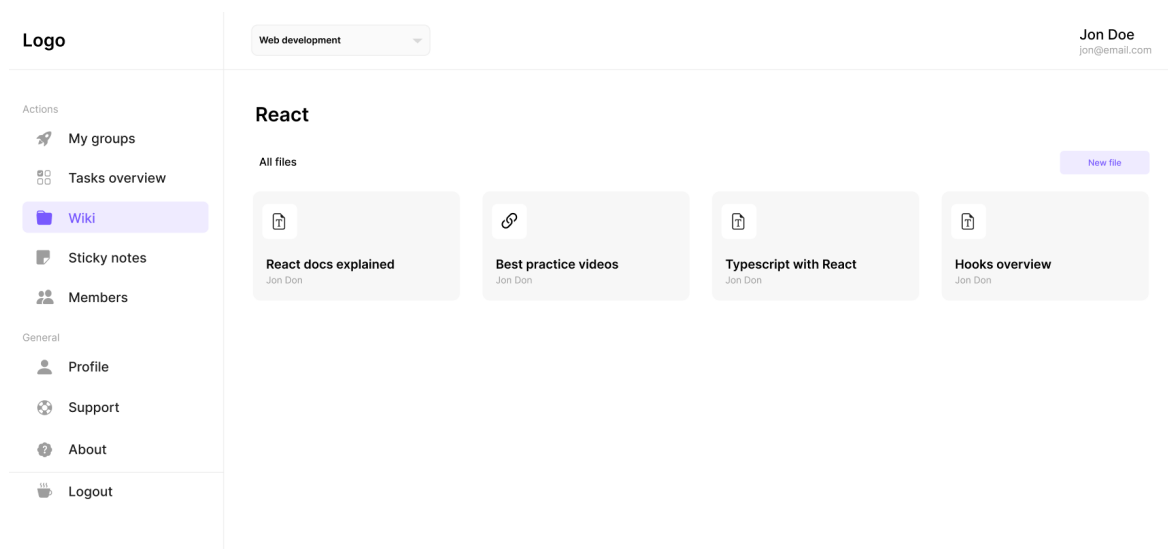
Obrázek 5 Návrh stránky s úkoly ve skupině
(zdroj: vlastní)



Obrázek 6 Návrh stránky detail úkolu
(zdroj: vlastní)



Obrázek 7 Návrh stránky Wiki skupiny obsahující složky

(zdroj: vlastní)

Obrázek 8 Návrh stránky detailu složky obsahující soubory

(zdroj: vlastní)

Tyto návrhy jsou vytvořeny pomocí aplikace Figma, kde nejdříve byla navržena základní struktura aplikace, návrhy komponent a určení barevnosti. Dále byla vytvořena řada variant této aplikace, která se postupně upravovala na základě specifik a požadavků této aplikace. Při návrhu byl také kladen důraz na to, aby se zachovala konzistence v designu, uživatelská přívětivost a také proveditelnost v rámci samotné implementace aplikace.

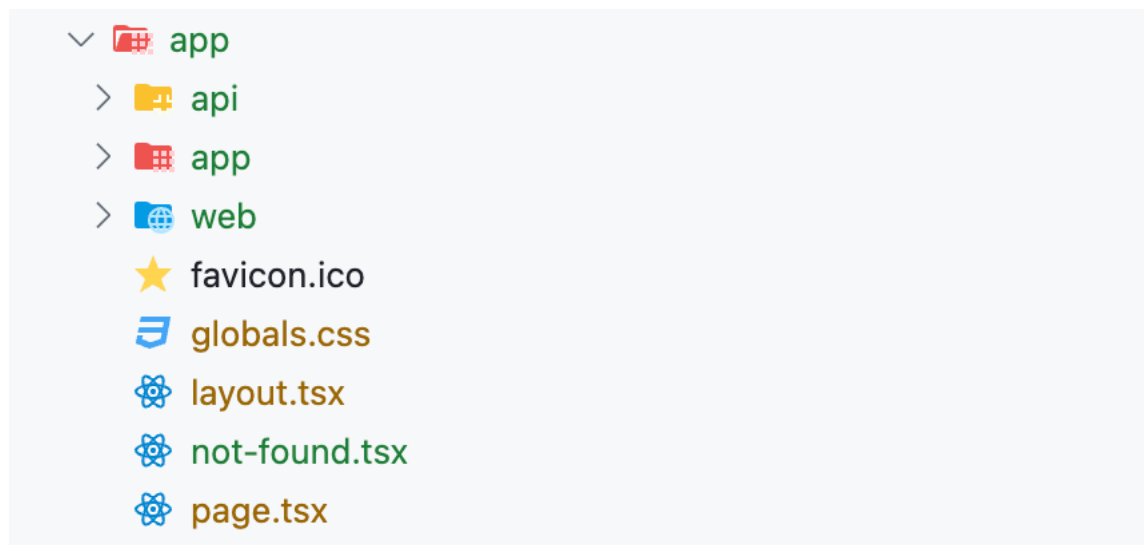
3.4 Implementace aplikace

V této kapitole praktické části jsou vysvětleny postupy a celý proces implementace navržené aplikace. Je zde vysvětleno, jak byla implementována frontendová část aplikace a také implementace backendové části a komunikace s databází pomocí server actions.

3.4.1 Struktura projektu

Projekt využívá standardní strukturu Next.js s app router. To znamená, že veškeré stránky (routy) jsou umístěny ve složce app. Složka je následně rozdělena do dalších tří složek, které reprezentují jednotlivé části aplikace. První složkou je api, což je složka obsahující endpointy pro komunikaci se serverem. I přes to, že v této aplikaci jsou především využívány serverové funkce (*server actions*) pro komunikaci se serverem a databází, pro správnou funkci autentizace bylo vytvořena funkce pro správu přihlášení a registrace pomocí Next Auth. Funkce je umístěna právě ve složce api. Dále se zde nachází složka app, což se může stát na první pohled jako redundantní složka, ale je to právě naopak. Rodičovská složka app není součástí žádné z route a pouze obsahuje další řadu složek a souborů určující routy aplikace. Potomek této složky, tedy zmíněná složka app se již stává součástí route a obsahuje části aplikace, která je přístupná pouze pro přihlášené uživatele. Naopak složka web obsahuje webovou část celého tohoto projektu, která je přístupná i bez přihlášení, která obsahuje statický obsah a funguje jako prezenční část aplikace, tedy webová stránka aplikace za účelem prezentace a případného marketingu.

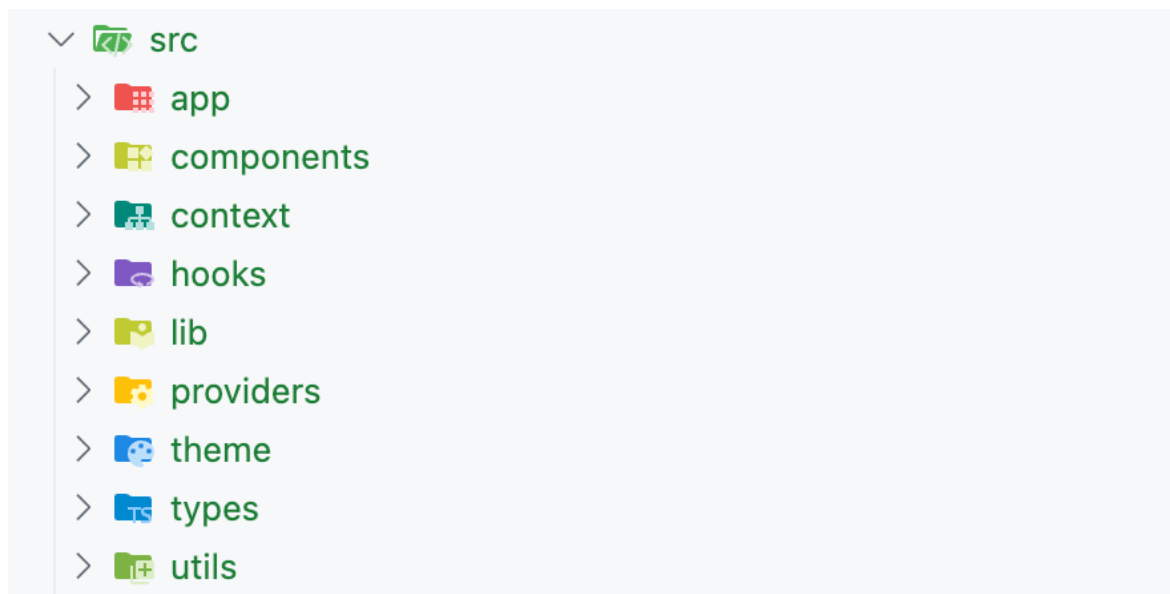
Rodičovská složka app také obsahuje vygenerované soubory, jako je například základní rozvržení, domovskou stránku či globální CSS. Do stejného místa můžeme také přidat stránky, které Next.js automaticky podporuje a usnadňuje tak práci. Je tím například *not-found*, což je stránka zobrazující obsah v případě erroru 404, tedy nenalezeného zdroje.



Obrázek 9 Struktura složky app

(zdroj: vlastní)

Celá souborová struktura pak vypadá jako na obrázku níže. Mimo popsané složky app se zde nachází například složka components, která obsahuje veškeré komponenty v rámci celé aplikace, které jdou sdílet jak napříč aplikací, tak prezenční částí webu. Komponenty jsou dále rozděleny do složky ui, kde se nachází menší komponenty, jako jsou textové vstupy, tlačítka nebo chips. Dále se zde nachází implementace komplexnějších komponent, jako jsou například formuláře či dialogy. Složka context obsahující veškeré použité contexty napříč aplikací, jako je například context pro správu dialogu. Na tuto složku také navazuje složka providers, která je klíčová pro logiku, která je sdílená skrze své potomky v aplikaci. Obsahuje tak například logiku pro správné zobrazení dialogů či jejich odstranění z DOM. Hooks je složka pro globální hooky, které lze využít napříč celou aplikací a které oddělují logiku od komponent. Ve složce lib můžeme najít hned několik užitečných implementací, jako je například připojení se k databázi, serverové funkce nebo validace formulářů. Složka theme je domovskou složkou pro vlastní definici stylů díky Material UI, takže v této složce jsou modifikovat či rozšířit globální definice stylů, jako jsou barvy či typografie. Types složka obsahuje definice typů a schémat v rámci zachování integrity aplikace, a to ať na straně klienta, tak na straně serveru. Ve složce utils jsou umístěny různé typy pomocných funkcí, díky kterým omezíme redundantní kód napříč aplikací. Příkladem mohou být funkce pro práci s datem, které je nutné využít hned na několika místech v celé aplikaci. Právě díky těmto funkcím nemusíme psát stejný kód stále dokola kdykoliv potřebujeme například formátovat datum.



Obrázek 10 Struktura složky src

(zdroj: vlastní)

3.4.2 Material UI komponenty

Pro implementaci rozhraní aplikace bylo využito knihovny Material UI, která nabízí řadu předpřipravených komponent, které je možné přizpůsobit svým vzhledem. Pro vytvoření vlastního stylu komponent z Material UI, bylo využito procesu, kdy se přepisují jednotlivé třídy základních komponent. Následně v aplikaci využijeme tuto nastylovanou komponentu.

Jednoduchou ukázkou může být implementace vlastního vzhledu tlačítka dle předlohy z figmy. Nejdříve můžeme vidět definici komponenty s názvem *PrimaryButton*, který volá *Styled(Button)< ButtonProps>()*, díky čemuž říkáme TypeScriptu, že tato komponenta bude přijímat všechny props (properties) jako standardní Material UI Button. Komponenta využívá objekt *theme*, což je sada definovaných stylů, čímž se zajistí konzistence rozhraní v každé z komponent využívající tuto sadu stylů. Následuje definice stylů samotné komponenty, kde můžeme využít tříd, které jsou zprostředkované přímo Material UI pro všechny komponenty, čímž zajistíme kompletní stylování každé z komponent, a to včetně všech jejich stavů či *pseudo class*. Výslednou komponentu následně využíváme jako všechny standardní komponenty v Reactu nebo Material UI.

```
import { Button, ButtonProps } from "@mui/material";
import { styled } from "@mui/system";

const PrimaryButton = styled(Button)<ButtonProps>(({ theme }) => ({
  backgroundColor: theme.palette.primary.light,
  color: theme.palette.primary.main,
  borderRadius: "8px",
  padding: "8px 16px",
  "&:hover": {
    backgroundColor: theme.palette.primary.main,
    color: theme.palette.common.white,
  },
  "&:disabled": {
    backgroundColor: theme.palette.grey[400],
    color: theme.palette.common.white,
  },
  "&.MuiButton-contained": {
    boxShadow: "none",
  },
  "&.MuiButton-containedError": {
    backgroundColor: "#FDF4F3",
    color: "#FA4B3D",
    boxShadow: "none",
    "&:hover": {
      backgroundColor: "#FA4B3D",
      color: theme.palette.common.white,
    },
  },
}));
```

Stejným způsobem bylo vytvořeno několik základních komponent, které jsou součástí aplikace a které Material UI nabízí.

3.4.3 Dialog

Pro vytvoření dialogu bylo důležité nejen implementovat samotnou komponentu zobrazující dialog a jeho obsah, ale především funkcionalitu pro správu toho dialogu. Za tímto účelem bylo využito contextu a provideru, díky čemuž se docílilo centralizované logiky a sdílení stavů dialogu napříč celou aplikací. Důležitá byla implementace zobrazení a také zavření dialogu. Implementace tohoto contextu pro dialog vypadá následovně.

```
"use client";

import { ReactNode, createContext } from "react";

interface DialogContextType {
  showDialog: (content: ReactNode) => void;
  closeDialog: () => void;
}

export const DialogContext = createContext<DialogContextType>({
  showDialog: () => {},
  closeDialog: () => {},
});
```

Jak si můžeme všimnout, nejdříve definujeme interface, který pevně specifikuje strukturu hodnot contextu pro správu dialogu. Interface tedy obsahuje dvě funkce, jedna pro zobrazení dialogu (*showDialog*) a druhá pro jeho zavření (*closeDialog*). Funkce *showDialog* vyžaduje parametr s názvem *content*, který je typově definován jako *ReactNode*. Funkce *closeDialog* nepřebírá žádné parametry. Následuje vytvoření a zároveň export tohoto contextu.

Pro úplnou funkci dialogu byla potřeba vytvořit také již zmíněný provider, který obsahuje veškerou logiku pro tento dialog. Jeho implementace vypadá následovně.

```
"use client";

import React, { ReactNode, useCallback, useEffect, useState } from "react";
import { DialogContext } from "../context/DialogContext";
import { createPortal } from "react-dom";

interface DialogProviderProps {
  children?: ReactNode;
}

const DialogProvider = ({ children }: DialogProviderProps) => {
  const [dialog, setDialog] = useState<ReactNode | null>(null);
  const [dialogPortalNode, setDialogPortalNode] = useState<HTMLElement | null>(
    null
  );

  useEffect(() => {
    let portalNode = document.getElementById("dialog-portal");

    if (!portalNode) {
      portalNode = document.createElement("div");
      portalNode.setAttribute("id", "dialog-portal");
      document.body.appendChild(portalNode);
    }
    setDialogPortalNode(portalNode);
  }, [])
```

```
const showDialog = useCallback((dialogComponent: ReactNode) => {
  setDialog(dialogComponent);
}, []);

const closeDialog = useCallback(() => {
  setDialog(null);
}, []);

return (
  <DialogContext.Provider value={{ showDialog, closeDialog }}>
    {dialogPortalNode && dialog && createPortal(dialog, dialogPortalNode)}
    {children}
  </DialogContext.Provider>
);
};

export default DialogProvider;
```

Jak si můžeme všimnout, nejdříve je definovaná direktiva, která zajistí, že komponenta je renderována na straně klienta. Je to zejména z toho důvodu, že celý provider využívá hooky, které lze použít pouze v klientských komponentách. Také deklarujeme interface, který definuje properties pro provider, které přijímá. V tomto případě je to children, která je typově definován jako ReactNode. Children reprezentuje veškeré komponenty, který zaobaluje tento provider. V případě této aplikace se jedná o veškerý obsah aplikace, jelikož context je využitý v celkovém rozvržení aplikace. Provider dále definuje stavy pomocí useState, kde první z nich spravuje obsah dialog (a dialog samotný). Druhý useState využitý pro uložení reference na DOM element, kde bude dialog vykreslen. V provideru následuje useEffect, který při inicializaci komponenty kontroluje, zda v DOMu existuje element, který disponuje ID dialog-portal. Pokud tento element neexistuje, vytvoří *<div>* element, kterému je přiřazeno právě toto ID. Následně aktualizuje stav dialogPortalNode vytvořeným elementem. Tímto zajistíme, že v DOMu bude vždy dedikovaný prostor pro renderování dialogu, a to zcela nezávisle na tom, kde v aplikaci bude vyvoláno renderování dialogu. Provider dále definuje dvě funkce, kterou je funkce pro zobrazení a funkce pro zavření dialogu. Jako poslední část tohoto provideru je samotné renderování tohoto dialogu. Pro správné zobrazení je definována funkce createPortal, která zajistí vykreslení dialogu v předem vytvořeném elementu. Implementace je efektivní zejména proto, že odděluje již vykreslený obsah a samotná dialog. Díky tomuto zajistíme správné vykreslení a konzistenci stylování, jelikož stylování potomka a dialogu na sebe nemají vliv. Například při využití z-index, kdy by mohlo dojít k překrytí dialogu obsahem z potomka.

```
▼ <div>
  ▶ <main> ... </main>
</div>
<div id="dialog-portal"></div>
```

Obrázek 11 Div element pro zobrazení dialogu v prohlížeči

(zdroj: vlastní)

Pokud si zobrazíme HTML v prohlížeči, uvidíme main element, který obsahuje vykreslený obsah stránky a prázdný div, kde bude umístěn dialog, pokud dojde k vyžádání jeho vykreslení.

3.4.4 Formuláře

Formuláře jsou nedílnou součástí této aplikace, jelikož slouží k vytváření obsahu. Jelikož formuláře komunikují s backendem a databází, je důležité, jakým způsobem jsou tyto formuláře navrženy a validovány, a to jak na straně klienta, tak následná validace dat na straně serveru ještě před samotným zpracováním, čímž se zajistí integrita. Aby byl celý vývojový proces jednodušší a kód přehlednější či škálovatelnější, byla navržena řada komponent, které jsou specifické přímo pro formulář. Příkladem může být komponenta pro textový vstup uživatele.

```
import React, { FC } from "react";

import { StandardTextFieldProps } from "@mui/material";
import { Controller, useFormContext } from "react-hook-form";
import CustomTextField from "../CustomTextField/CustomTextField";

interface FormTextFieldProps {
  name: string;
  defaultValue?: unknown;
}

const FormTextField: FC<FormTextFieldProps & StandardTextFieldProps> = ({
  name,
  defaultValue,
  ...props
}) => {
  const { control } = useFormContext();

  return (
    <Controller
      control={control}
```

```

name={name}
defaultValue={defaultValue}
render={({ field, fieldState: { error } }) => (
  <CustomTextField
    {...field}
    {...props}
    value={field.value ?? ""}
    error={!!error}
    helperText={error?.message as string}
  />
)}
/>
);
};

export default FormTextField;

```

Ukázka kódu ukazuje implementaci znovupoužitelné komponenty pro textový vstup ve formuláři, který byl implementován za pomoci Material UI a React Hook Form (RHF). Díky tomuto zajistíme, že tento vstup je tzv. řízen (controlled) RHF, které spravuje stav formuláře, validaci a odesílání dat. RHF je knihovna pro správu stavu formulářů a validaci v Reactu. Komponenta přebírá jako properties name, což je unikátní identifikátor vstupu ve formuláři a defaultValue, což je výchozí hodnota, která může, ale také nemusí být předána do komponenty. Díky tomuto se komponenta stává flexibilní a můžeme ji využít jak pro vytvoření, tak editaci. Můžeme si také všimnout, že komponenta přebírá properties StandardTextFieldProps, což jsou veškeré props, které také přebírá výchozí Material UI Text field, díky čemuž můžeme do komponenty předat aditivní props, jako je například přidání stylování, k již definovaným stylům komponenty. V komponentě dále využíváme useFormContext, což je hook z RHF, který umožňuje přístup ke kontextu formuláře bez nutnosti předávat data pomocí props. Tohle je užitečné zejména ve složitějších formulářích, kde dochází k nutnosti přístupu k metodám formuláře, jako je například register, handleSubmit nebo control a to z několika komponent najednou.

Celá funkce (komponenta) vrací Controller, který spravuje vstup. Controller je speciální komponenta z RHF, která propojí externí komponentu vstupu (v tomto případě textový vstup) s formulářem. Controller tedy funguje jako most mezi komponentou a formulářem. Controller přijímá následující props:

- **Control** – objekt, který zprostředkovává RHF, který spravuje všechny vstupy formuláře. Sleduje hodnotu vstupu, stav validace a také aktualizuje stav formuláře

při jakékoliv změně vstupu. Control je vždy vyžadován při použití Controller, který potřebuje zaregistrovat vstup do RHF.

- **Name** – propojuje vstupní pole s formulářem pomocí definovaného názvu.
- **defaultValue** – výchozí hodnota vstupu.
- **Render** – definuje, jak by se měl vstup chovat.

```
interface CreateEditFileFormProps {
  defaultValues?: Partial<FileFormValues>;
  onSubmit: (data: FileFormValues) => void;
  editMode?: boolean;
}

const CreateEditFileForm = ({
  defaultValues,
  onSubmit,
  editMode,
}: CreateEditFileFormProps) => {
  const { closeDialog } = useDialogContext();
  const form = useForm<FileFormValues>({
    resolver: yupResolver(schema),
    defaultValues,
  });

  return (
    <FormProvider {...form}>
      <form onSubmit={form.handleSubmit(onSubmit)}>
        <FormLabel label="Name" />
        <FormTextField name="name" placeholder="Enter name" fullWidth />
        <FormLabel label="File type" sx={{ mt: 2.5 }} />
        <FormSelect
          name="fileType"
          options={getFileTypeOptions()}
          fullWidth
          disabled={editMode}
        />

        <FormDialogBtn
          primaryBtnLabel={editMode ? "Save changes" : "Create file"}
          secondaryBtnLabel="Close"
          secondaryBtnAction={closeDialog}
        />
      </form>
    </FormProvider>
  );
};

export default CreateEditFileForm;
```

Všechny komponenty, které jsou určeny pro použití ve formulářích, jsou pojmenovány dle konvence, která v názvu definuje, že se jedná o formulářovou komponentu, tedy použití slova *Form*. Tyto komponenty jsou tedy využity ve formulářích, jako můžeme vidět na obrázku výše. Tento formulář slouží pro vytvoření a editaci složek. V interface je definované, že props které komponenta přijme jsou `defaultValues`, což je objekt výchozích hodnot, které je možné do ve formuláři automaticky vyplnit. Toto řešení je využito při editaci existující složky. Dále funkce `onSubmit`, což je funkce, která se má provést v případě, že uživatel potvrdí formulář. Potvrzením je myšleno kliknutí na tlačítko pro vytvoření či uložení nové složky. Jako poslední prop, kterou tento formulář přijímá je hodnota `editMode`, která může nabývat hodnot `true` nebo `false`, kdy na základě této hodnoty změníme text v primárním tlačítku formuláře. Komponenta dále využívá již dříve zmíněný `context` pro správu dialogu, kdy v tomto případě pracujeme se zavřením formuláře, což je funkce, která se provede v momentě, kdy uživatel klikne na sekundární tlačítko formuláře, který je umístěn v dialogu, čímž tento dialog zavře. Následuje inicializace RHF pomocí `useForm`, která obsahuje resolver, který umožňuje využití externích knihoven pro validaci, jako `Yup` nebo `Zod`. Následuje předání výchozích hodnot pomocí `defaultValues`, které vyplní formulář, pokud jsou tyto hodnoty zprostředkovány.

Komponenta samotná pak vrací celý formulář, který je zaobalen díky `FormProvider`, což je komponenta opět z RHF, díky které můžeme sdílet stav formuláře se všemi potomky. Díky tomuto můžeme využít `useFormContext`, který můžeme vidět v individuálních komponentách (vstupech) formuláře, jako bylo v předchozí ukázce textového pole.

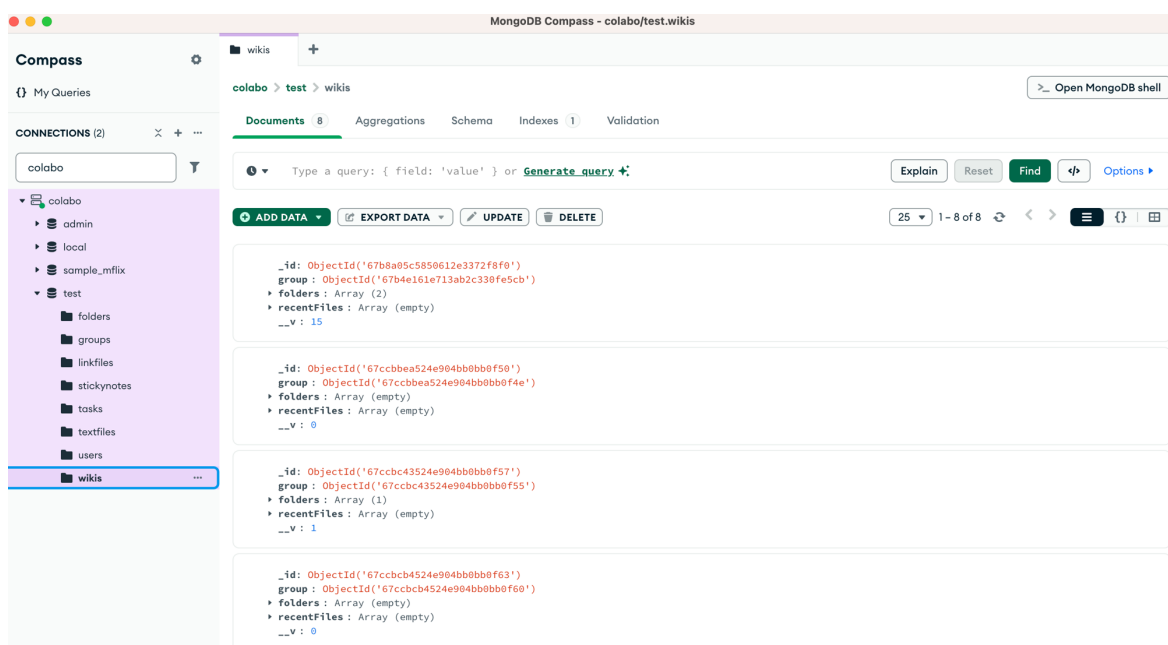
```
import { object, ObjectSchema, string, mixed } from "yup";
import { FileFormValues } from "@components/forms/File/models/FileForm";
import { FileType } from "@types/enums/fileType";

export const schema: ObjectSchema<FileFormValues> = object({
  name: string().required("Title is required"),
  fileType: mixed<FileType>().required("File type is required"),
});
```

K zajištění validace na straně klienta je využito `Yup` knihovny pro validaci. Díky tomuto zajistíme, že data formuláře splňují předem definované požadavky ještě před zasláním data na stranu serveru. V tomto případě definujeme, že složka název (`name`) složky musí být `string`, tedy řetězec znaků a také to, že je vyžadován, tedy že vstup pro `name` musí být vyplněn. Následuje `fileType`, který vyžaduje, aby se hodnota vstupu rovnala jedné z hodnot `FileType`, což je definovaný jako typ `enum`.

3.4.5 MongoDB

Jako databáze bylo pro tento projekt zvoleno MongoDB. Za tímto účelem bylo nutné vytvořit účet přímo na stránce MongoDB a zvolit variantu Atlas, což znamená že databáze projektu bude fungovat na cloudu společnosti MongoDB. Kdybychom chtěli projekt škálovat, mohli bychom MongoDB mít na svém vlastním serveru, takže bychom nemuseli za databázi platit. Nicméně tento projekt nepřekročí nároky varianty zdarma, takže můžeme klidně využít Atlas. Při založení účtu na MongoDB dostaneme tzv. connection string, která využijeme pro připojení našeho projektu na náš cluster, kde běží databáze projektu. Abychom mohli vidět obsah databáze a případně s ním i manuálně manipulovat, že možné využít aplikace MongoDB Compass.



Obrázek 12 MongoDB Compass

(zdroj: vlastní)

Pro připojení databáze je tedy nutné využít vygenerovaný connection string, která umístíme do *.env.local* nebo ekvivalentního souboru, který je součástí *gitignore*, díky čemuž nebo náš connection string součástí souborů, které lze umístit na GitHub nebo jiný verzovací systém. Abychom opět zlepšili kvalitu kódu a nepsali redundantní kód, je důležité vytvořit funkci, díky které se k databázi připojíme. Tuto funkci umístíme složky lib.

Připojení k databázi

V této části je ukázka praktického připojení k MongoDB, které lze následně využít v Next.js. Pro připojení využíváme knihovnu Mongoose a po úspěšném připojení k databázi můžeme začít komunikovat s databází pomocí server actions a tím vytvářet, číst či mutovat zdroje, které jsou v databázi uloženy. Takové připojení může vypadat následovně.

Implementace nejdříve kontroluje, zda projekt jsou v projektu definovány proměnné prostředí. Konkrétně dochází ke kontrole proměnné `MONGO_URI`, což je řetězec vygenerovaný přímo z MongoDB při vytváření databáze. Tento řetězec také obsahuje hesla a další potřebné atributy pro připojení. V případě, že proměnná není inicializována, připojení k databázi neproběhne a zobrazíme error.

```
const MONGO_URI = process.env.MONGO_URI;
if (!MONGO_URI) {
  throw new Error(
    "Please define the MONGO_URI environment variable in .env.local"
  );
}
```

Inicializujeme cache, která v sobě potenciálně drží již existující připojení k databázi. Jelikož `global.mongoose` bude při prvním spuštění aplikace undefined, proměnná `cached` bude obsahovat objekt, která má `conn` a `promise` nastavené na `null`.

```
const cached = global.mongoose || { conn: null, promise: null };
```

V implementaci následuje funkce `dbConnect`, což je funkce, která se využívá pro připojení k databázi. V této funkci nejdříve dojde ke kontrole, zda je aplikace připojena k databázi. Pokud ano, funkce se v tento moment zastaví a vrátí existující připojení. V případě, že však připojení neexistuje (například při spuštění aplikace), je toto spojení vytvořeno.

Při připojení je možné definovat řadu nastavení, které jsou definované v proměnné `opts`. Pro tuto aplikaci je například nastaven maximální počet připojení nebo minimální počet připojení. Funkce následně uloží toto připojení a je využito v následujícím požadavku pro připojení k databázi.

```
async function dbConnect() {
  if (cached.conn) {
    return cached.conn;
  }
  if (!cached.promise) {
    const opts = {
      bufferCommands: false,
      maxPoolSize: 5,
      minPoolSize: 1,
      serverSelectionTimeoutMS: 5000,
      socketTimeoutMS: 45000,
    };

    cached.promise = mongoose
      .connect(MONGO_URI!, opts)
      .then((mongoose) => mongoose.connection);
  }

  cached.conn = await cached.promise;
  return cached.conn;
}

export default dbConnect;
```

3.4.6 MongoDB schema

Pro vytvoření struktury dokumentů v MongoDB je využit Mongoose. Jedná se o ODM (*Object Data Modeling*) knihovnu, která zprostředkovává schéma pro práci s MongoDB. Díky tomuto zajistíme validaci dat ještě před jejich uložením.

```
import mongoose, { Schema, Document } from "mongoose";

export interface ITextFile extends Document {
  _id: mongoose.Types.ObjectId;
  name: string;
  slug: string;
  groupId: mongoose.Types.ObjectId;
  folderSlug: string;
  content?: string;
  createdBy: mongoose.Types.ObjectId;
  createdAt: Date;
}

const TextFileSchema = new Schema<ITextFile>({
  name: { type: String, required: true },
  slug: { type: String, required: true },
  groupId: { type: Schema.Types.ObjectId, required: true },
  folderSlug: { type: String, required: true },
  content: { type: String },
  createdBy: {
```

```
    type: mongoose.Schema.Types.ObjectId,  
    ref: "User",  
    required: true,  
  },  
  createdAt: { type: Date, default: Date.now },  
});  
  
const TextFile = mongoose.models.TextFile ||  
mongoose.model<ITextFile>("TextFile", TextFileSchema);  
  
export default TextFile;
```

V této ukázce byl nejdříve definován interface `ITextFile`, díky kterému definujeme, v jakém tvaru očekávat data. Následuje vytvoření schématu, kde je definováno, v jaké struktuře bude dokument uložený v MongoDB. V tomto schématu definujeme, jaké části dokumentu jsou vyžadovány a v jakém formátu je může databáze očekávat.

3.4.7 Server actions

Jelikož je celý projekt postavený na frameworku Next.js, backendová integrace může být implementována díky server actions. V rámci tohoto přístupu nemusíme implementovat separátní aplikaci pro backend, ale stačí implementovat serverové funkce, které Next.js nabízí a díky kterým dosáhneme stejného výsledku, a to s podstatně méně napsaného kódu. Všechny server actions jsou umístěny ve složce `actions`, která je umístěna ve složce `lib`. Všechny tyto server actions jsou umístěny dle jejich domény. To znamená, že všechny serverové funkce spojené s úkoly ve skupinách, jako je čtení, vytváření, editace či odstranění zdroje z databáze jsou v jednom souboru. Každý soubor obsahující tyto serverové funkce obsahuje direktivu *use server*, čímž definujeme, že veškerý obsah tohoto souboru bude možné spustit pouze na straně serveru.

```
export async function createGroupAction( formData: FormData ): Promise<{ success:  
boolean; error?: string }> {  
  try {  
    await dbConnect();  
  
    const session = await getUserFromSession();  
  
    if (!session) {  
      return {  
        success: false,  
        error: "You must be logged in to perform this action",  
      };  
    }  
  }  
}
```

```
const user = {
  _id: session.id,
  username: session.name,
};

const name = formData.get("name").toString();
const description = formData.get("description").toString();

if (!name || !description) {
  return { success: false, error: "Name and description are required" };
}

const newGroup = new Group({
  name,
  description,
  admin: { id: user._id, username: user.username },
  members: [
    {
      member: user._id,
      role: "admin",
    },
  ],
});
const createdGroup = await newGroup.save();

await User.updateOne(
  { _id: user._id },
  { $push: { groups: createdGroup._id } }
);

const wiki = new Wiki({ group: createdGroup._id });
await wiki.save();

await createdGroup.updateOne({ wiki: wiki._id });

revalidatePath("/app/groups");

return { success: true };
} catch (error) {
  console.error("Error creating group:", error);
  return { success: false, error: "Error creating group" };
}
}
```

Tato serverová funkce je ukázkou toho, jak jsou implementovány všechny ostatní funkce, které nahrazují tradiční API. Tato konkrétní funkce slouží pro vytvoření nové skupiny. Jako parametr této funkce jsou formData, které reprezentují data z formuláře. Abychom mohli komunikovat s databází, je nutné se k ní nejdříve připojit, a to díky funkci

dbConnect. Funkce dále funguje jako standardní implementace backendu, takže zde proběhne získání informací o uživateli, validace vstupních dat z formuláře a následuje zápis do databáze. Abychom na straně klienta získali nejnovější data, je nutné udělat revalidaci cesty, kde se právě uživatel nachází. K tomuto využijeme funkci `revalidatePath`, kterou zprostředkovává `Next.js` a díky které uděláme nový požadavek na serveru, aby aktualizoval definovanou stránku. Revalidace nám umožní ukázat uživateli aktualizovaná data, a to vše v případě, že celý proces vytváření proběhne úspěšně. Abychom pokryli i případy, kdy může dojít k chybě, celá funkce vrací objekt, který informuje, zda funkce proběhla úspěšně v případě, že tato funkce v nějaké části selže, vrátíme klientovi i konkrétní chybu a proces vytváření nové skupiny selže.

3.4.8 Získávání dat a jejich zobrazení

Jak bylo zmíněno v předchozí kapitole, tak pro veškeré CRUD operace jsou využity server actions. To znamená, že stejným způsobem můžeme získat data z databáze a ty následně zobrazit uživateli. Tuto serverovou funkci pak můžeme využít přímo v komponentně samotné a na základě výsledku této akce zobrazit příslušné UI. Ukázkou může být následující ukázka ze stránky, která vykresluje skupiny uživatele. Importujeme funkci `getGroups` a tu následně použijeme pro získání dat na stránce. Výsledek této funkce pak budeme mít v proměnné `response`. Jelikož funkce `getGroups` vrací informaci o tom, zda získání dat bylo úspěšné pomocí identifikátoru `success`, který je typu `boolean` a to znamená, že je buď `true` nebo `false`. Tímto způsobem budeme vědět, že data se nepodařilo z nějakého důvodu získat a měli bychom tak podle toho upravit rozhraní a tím informovat uživatele. `Success` tak můžeme využít v podmínce, která situaci ošetří a případně zajistí vykreslení chybové hlášky. Dále můžeme v ukázce vidět, že opět vytváříme novou proměnnou, do které budou vložena data, které reprezentují skupiny uživatele nebo v případě jakékoliv jiné serverové akce to znamenají příslušná data, která se snažíme získat. Vytváření nové proměnné není nutné, nic méně v rámci zachování lepší čitelnosti kódu je lepší tuto proměnnou ponechat. Pokud data nejsou k dispozici, znamená to, že uživatel nemá žádné skupiny k dispozici. V tento moment se opět naplní podmínka, která tuto skutečnost zaznamená a zobrazení uživateli komponentu informující o tom, že není součástí žádných skupin.

```
const GroupsPage = async () => {
  const response = await getGroups();

  if (!response.success) {
    return notFound();
  }

  const groups = response.data;

  if (!groups) {
    return <NoContentInfoBox message="You don't have any groups yet." />;
  }
}
```

V případě, že uživatel je součástí nějakých skupin a tyto skupiny jsou získány z databáze úspěšně, můžeme je vykreslit. Pro vykreslení je využito mapování, což je proces, kdy dojde k iteraci napříč celým polem dat, které získáme a pro každou jednu položku v tomto poli je vykreslena specifická komponenta.

```
return (
  <Container>
    <Header
      title="Your Groups"
      button={
        <OpenDialogButton
          label="Create new group"
          dialog={<CreateGroupDialog />}
        />
      }
    />
    <Grid container spacing={2}>
      {groups.map((group) => (
        <Grid size={{ xs: 12, sm: 6, md: 4, lg: 4, xl: 3 }} key={group.id}>
          <GroupCard
            id={group.id.toString()}
            name={group.name}
            description={group.description}
            admin={group.admin}
          />
        </Grid>
      ))}
    </Grid>
  </Container>
);
};

export default GroupsPage;
```

Těto přístup je využit napříč celou aplikací v každém místě, kde je potřeba získat nějaké data z databáze. Ukázka je příkladem implementace, kterou tak můžeme najít v samotné aplikaci a která díky stejnému přístupu zachovává přehlednost v kódu a nabízí snadnou úpravu v případě změn ať už ze strany klientské části při úpravě uživatelského rozhraní nebo na straně serveru v případě změny server actions.

3.4.9 Next Auth

K autentizaci uživatelů tento projekt využívá Next Auth, který velmi dobře pracuje s Next.js. Díky Next Auth nemusíme implementovat vlastní logiku pro autentizaci. Konfigurace této knihovny se nachází v souboru, který je umístěn v `/api/auth/[...nextauth]`, kde definujeme poskytovatele a všechny strategie či další konfigurace pro přihlášení do naší aplikace. Next Auth umožňuje řadu poskytovatelů k ověření uživatelů. Patří mezi ně například přihlášení pomocí Google nebo Apple účtu. Pro tento projekt byl však vybrán způsob přihlášení pomocí emailu a hesla, tedy poskytnutím tzv. credentials. V implementaci tak definujeme pole formuláře, které je nutné vyplnit v rámci přihlášení. Ačkoliv nemusíme pro autentizaci pomocí Next Auth implementovat vlastní backend, stále musíme implementovat alespoň základní logiku pro přihlášení, která bude komunikovat s naší databází a ověřovat uživatele. Logika se implementuje ve funkci `authorize`, což je asynchronní funkce, která ověří zadané údaje a vrátí nalezeného uživatele. Funkce v první řadě ověří, zda je zprostředkován email a heslo. Následně se připojíme k databázi a provedeme kontrolu, zda existuje uživatel se stejným emailem jako ten, se kterým se snažíme přihlásit. Pokud ano, ověříme také heslo tohoto uživatele s heslem poskytnutým skrze přihlašovací formulář. V případě, že uživatel s tímto emailem existuje a hesla se shodují, vrací tato funkce objekt reprezentující uživatele. V tomto objektu vracíme ID, jméno a email uživatele, který pak Next Auth serializuje do tokenu.

```
export const authOptions: NextAuthOptions = {
  providers: [
    CredentialsProvider({
      name: "Credentials",
      credentials: {
        email: {
          label: "Email",
          type: "text",
          placeholder: "example@example.com",
        },
        password: { label: "Password", type: "password" },
      },
      async authorize(credentials) {
```

```
    if (!credentials?.email || !credentials?.password) return null;

    await dbConnect();

    const user = await User.findOne({ email: credentials.email });
    if (!user) {
        return null;
    }

    const isValid = await bcrypt.compare(
        credentials.password,
        user.password
    );
    if (!isValid) {
        return null;
    }

    return { id: user._id.toString(), name: user.name, email: user.email };
  },
  }),
],
```

Dále konfiguruje způsob, jak pracovat se session aktuálně přihlášeného uživatele. *Strategy jwt* zajišťuje, že session se nebude ukládat v databázi, ale v zašifrovaném tokenu. Pro tento token dále definujeme *maxAge*, což definuje životnost token, tedy po jak dlouhé době přestane být token validní a uživatel se bude muset znovu přihlásit.

```
session: {
  strategy: "jwt",
  maxAge: 1 * 24 * 60 * 60,
},
```

Next Auth nám také dává možnost vytvořit tzv. *callbacks*, což jsou asynchronní funkce, které zajišťují, co se má stát, pokud se vykoná nějaká z akcí. V implementaci této aplikace se využívá JWT a session callback. JWT callback se zavolá vždy, když je token vytvořen nebo aktualizován a můžeme do něj ukládat libovolná data, které potřebujeme v naší aplikaci. Tyto data dále zprostředkujeme právě pomocí session callback.

```

callbacks: {
  async jwt({ token, user }) {
    if (user) {
      token.id = user.id;
      token.name = user.name;
      token.email = user.email;
    }
    return token;
  },
  async session({ session, token }) {
    if (token) {
      session.user = {
        id: token.id as string,
        name: token.name as string,
        email: token.email as string,
      };
    }
    return session;
  },
},
};

```

Abychom mohli získat session, a tak i její obsah v podobně informací o uživateli, můžeme využít přístupu buď na straně klienta, ale také na straně serveru, což je užitečné z důvodu používání server actions. Pro účely získání session na straně serveru byla vytvořena funkce, která využívá `getSession` funkci přímo z Next Auth a pokud tato funkce vrátí session obsahující data uživatele, vrátíme objekt uživatele, který je definován ve stejném souboru.

```

import { getSession } from "next-auth/next";
import { authOptions } from "../authOptions";

export interface User {
  id: string;
  name: string;
  email: string;
}

export async function getUserFromSession(): Promise<User | null> {
  const session = await getSession(authOptions);
  return session?.user as User | null;
}

```

Abychom mohli získat data uživatele, je nutné zaobalit celou aplikaci nebo její části, kde chceme přístup k session a to pomocí tzv. *SessionProvideru* z Next Auth.

```
"use client";

import { SessionProvider } from "next-auth/react";

export default function AuthProvider({ children }: {
  children: React.ReactNode;
}) {
  return <SessionProvider>{children}</SessionProvider>;
}
```

Díky tomuto pak můžeme využít `useSession`, která zprostředkuje status, ve kterém se session nachází a také samotná data která poskytuje. Ukázkou může být získání jména a emailu právě přihlášeného uživatele za účelem zobrazení těchto dat v uživatelském rozhraní aplikace.

```
const { data: session, status } = useSession();

if (status === "loading") {
  return <div>Loading session...</div>;
}

if (!session) {
  return null;
}

const user = {
  name: session.user.name,
  email: session.user.email,
};
```

3.4.10 Registrace nového uživatele

Aby bylo možné uživatele přihlásit, je nedíve nutné, aby uživatel provedl registraci. Jedná se o standardní přístup ve všech webových aplikacích, kde je nutné, aby každý uživatel měl založený svůj účet a byl tak zaznamenán v databázi. V této aplikaci to není výjimkou a registrace je vytvořena pomocí server actions a ukládání uživatelů probíhá přímo do projektové databáze (MongoDB). Při registraci nového uživatele je nutné, aby při registraci vyplnil do formuláře své jméno, email a heslo. Email musí být unikátní, což znamená, že v databázi může být pouze jeden uživatel s konkrétním emailem. V případě, že uživatel při registraci vyplní email, který je již využíván jiným uživatelem, registrace neproběhne úspěšně.

```
export async function signUp(formData: FormData): Promise<{ success?: boolean;
error?: string }> {
  try {
    await dbConnect();

    const data = {
      email: formData.get("email")?.toString(),
      password: formData.get("password")?.toString(),
      name: formData.get("name")?.toString(),
    };

    if (!data.email || !data.password || !data.name) {
      return { success: false, error: "All fields are required." };
    }

    const checkUser = await User.findOne({ email: data.email });
    if (checkUser) {
      return { success: false, error: "User already exists." };
    }

    const hashedPassword = await bcrypt.hash(data.password, 10);

    const newUser = new User({
      ...data,
      password: hashedPassword,
      createdAt: new Date(),
    });
    await newUser.save();

    return { success: true };
  } catch (error) {
    console.error("Error signing up:", error);
    return { success: false, error: "An error occurred during sign up." };
  }
}
```

V této implementaci serverové funkce můžeme vidět konkrétní logiku obsluhující registraci. Funkce očekává jako vstup FormData. V samotné funkci pak dojde k připojení k databázi prostřednictvím funkce dbConnect. Následně proběhne kontrola, jestli data z formuláře obsahují jméno, email a heslo. Pokud ne, funkce v této podmínce skončí a informuje o neúspěšném chodu této funkce. V případě, že jsou všechny data k dispozici, dojde ke kontrole unikátnosti emailu. To znamená, že zkontrolujeme záznamy v databázi pro tabulku uživatelů vůči zprostředkovanému emailu. Pokud se tento email nachází v jednom ze záznamů, dojde opět k přerušení chodu funkce s informací, že tento uživatel již existuje. Pokud je však email unikátní, funkce pokračuje k hashování hesla. Následně jsou všechny data uživatele uložena do databáze společně s datumem registrace a funkce vrací success s hodnotou true. V tento moment dojde k úspěšné registraci nového uživatele, který

se může přihlásit. Pro zachycení dalších neočekávaných chyb je logika funkce obalena pomocí try/catch.

3.4.11 Bezpečné uchovávání hesel

Pro zachování bezpečnosti a integrity celé aplikace je důležité, jak se ukládají data uživatelů, a to zejména jejich hesla. Obecně se nedoporučuje, aby se hesla ukládala bez nějaké další modifikace. Šifrování (přesněji hashování) hesel před jejich uložením je kritickým faktorem pro ochranu dat uživatele v případě úniku dat z databáze. Hesla v databázi tak nejsou uložena ve formátu, v jakém je uživatel vytvořil, ale je zde pouze jejich šifrovaná verze, která je pro běžného člověka nečitelná. V tomto projektu je využita knihovna bcrypt.js, která se stará o hashování vstupních dat. Při vytváření nového uživatele, respektive u registrace uživatele je tato knihovna využita pro šifrování hesla, které uživatel vyplní v registračním formuláři. Heslo je nejdříve z formuláře převzato ve stejném formátu, jako ho uživatel vytvořil a následně je použito jako vstup do metody hash, kterou získáme právě z knihovny bcrypt.js. Výstupem této funkce bude šifrovaná verze tohoto hesla, která je následně uložena do databáze společně se zbytkem dat uživatele.

```
const hashedPassword = await bcrypt.hash(data.password, 10);
```

Při přihlášení uživatele je opět využita tato knihovna a její metoda compare, která porovnává heslo z formuláře s hashovanou verzí hesla v databázi. Tímto způsobem ověříme, zda uživatel zadal při přihlášení správné heslo.

```
const isValid = await bcrypt.compare(credentials.password, user.password);
```

Praktickou ukázkou může být registrace uživatele, který zvolil jako heslo řetězec heslo123. Podíváme-li se do databáze na uložený záznam tohoto uživatele, uvidíme heslo uložené v následujícím šifrovaném formátu.

```
password : "$2b$10$woIeZmz6HcXQ.palia1nyOtWWGgThIYdsl1xPp0RAIUQWvulaRnQ0"
```

Obrázek 13 Ukázka formátu hesla po šifrování pomocí hash

(zdroj: vlastní)

3.4.12 Protected routes

Abychom docílili toho, že se do aplikace dostanou jen přihlášení uživatelé, musíme implementovat logiku, která bude část aplikace, která vyžadující autentizaci spravovat. Jelikož víme, že část aplikace, která vyžaduje přihlášení se nachází ve složce /app, můžeme tak specifikovat, že všechny stránky naší aplikace, které se nachází na adrese začínající jako /app, budou vyžadovat přihlášení. Způsob, který využívá implementovaná aplikace je tzv. middleware. Next.js umožňuje vytvořit tyto middlewary, které umožňují spuštění nějakého kódu ještě před splněním požadavku. Díky odpovědi z middlewaru, pak dokážeme ovlivnit chování aplikace. Next.js automaticky vyhledává soubor s názvem middleware.js (nebo middleware.ts) v src adresáři projektu a přistupuje k tomuto souboru jako k middlewaru, a to zcela bez nutnosti jakékoliv další konfigurace. Tento projekt využívá middleware pouze pro kontrolu přihlášení uživatele a vypadá následovně.

```
export { default } from "next-auth/middleware";

export const config = {
  matcher: ["/app/:path*"],
};
```

Na začátku souboru exportujeme middleware, který zprostředkovává Next Auth. Tento middleware kontroluje příchozí požadavky a validaci JWT tokenu, který je uložen v cookies prohlížeče. V případě, že je token validní, umožní uživateli pokračovat. Pokud však validní není, je uživatel přesměrován na přihlašovací stránku aplikace. Dále můžeme využít tzv. matcher, který obsahuje pole adres, které vyžadují přihlášení. Abychom však nemuseli manuálně doplňovat toto pole kdykoliv, kdy vytvoříme novou stránku, můžeme využít vzory, které například v tomto případě definují, že všechny cesty, které začínají jako /app jsou chráněny a uživatel tak musí mít platný JWT token. Na tomto základě byl tedy rozdělen projekt a například domovská stránka celého projektu či stránky pro přihlášení a registraci jsou mimo adresář /app a tak nevyžadují přihlášení, čímž umožníme uživateli přístup i bez tokenu a zajistíme tak, že se uživatel může registrovat, přihlásit nebo navštívit domovskou stránku projektu, která slouží jako prezenční část projektu.

3.4.13 Editor

Jednou z velkých součástí této aplikace a je sdílení informací mezi studenty v daných skupinách. Mezi způsoby tohoto sdílení patří textová forma, kde uživatelé mohou využít implementovaného textového editoru. Tento obsahuje řadu základních textových úprav a je

implementován pomocí knihovny Titap. Knihovna umožňuje vytvoření plnohodnotného editoru a možnost konfigurovat tento editor dle potřeb vývojáře.

```
const editor = useEditor({
  extensions: [
    StarterKit,
    TextStyle,
    TiptapUnderline,
    Code,
    Subscript,
    Superscript,
    Heading,
    Color,
    Highlight.configure({ multicolor: true }),
    Color.configure({
      types: ["textStyle"],
    }),
    Heading.configure({
      levels: [1, 2, 3],
    }),
    HorizontalRule.configure({
      HTMLAttributes: {
        class: "editor-divider",
      },
    }),
    TextAlign.configure({
      types: ["heading", "paragraph"],
      alignments: ["left", "center", "right", "justify"],
    }),
    CharacterCount,
    Placeholder.configure({
      placeholder: "Write something ...",
    }),
  ],
  content: textFile.content,
});

if (!editor) {
  return null;
}
```

Nejdříve bylo nutné vytvořit instanci editoru díky funkci, která importujeme přímo z knihovny titap. Do této instance dále konfiguruje všechna rozšíření, které reprezentují chování editor. Mezi tyto rozšíření patří například nastavení nadpisů či zarovnání textu.

```

<EditorContent
  editor={editor}
  className="editor"
  style={{
    padding: "2rem 3rem",
    boxShadow: "0 0 8px rgba(0, 0, 0, 0.1)",
    marginTop: "1rem",
    width: "100%",
    height: "100%",
    minHeight: "297mm",
  }}
/>

```

V projektu poté využijeme komponentu `EditorContent`, do které předáme jako props instanci editoru. Komponenta představuje interaktivní oblast pro psaní a úpravu vytvářeného obsahu. Pro použití této komponenty ji opět stačí importovat z knihovny `Tiptap` a případně nastylvat pomocí vlastních stylů. Obsah tohoto editoru je ve formátu HTML, tedy zprostředkovává formu tohoto značkovacího jazyka, čímž je možné obsah jednoduše uložit do databáze ve formátu textového řetězce (string). Abychom mohli formátovat obsah editoru, je bylo nutné vytvořit tyto funkce. Pro tento požadavek tak aplikace obsahuje komponentu s názvem `EditorControlPanel`, který následně vykresluje jednotlivé sekce obsahující tlačítka pro různé typy akcí.

```

<EditorControlPanel>
  <HeadingControlSection editor={editor} />
  <Divider orientation="vertical" flexItem />
  <TextControlSection editor={editor} />
  <SubAndSuperScriptControlSection editor={editor} />
  <Divider orientation="vertical" flexItem />
  <ListsControlSection editor={editor} />
  <AlignmentControlSection editor={editor} />
  <Divider orientation="vertical" flexItem />
  <ColorControlSection editor={editor} />
  <HighlightControlSection editor={editor} />
</EditorControlPanel>

```

Pokud se podíváme na ukázkou konkrétní sekce ovládající editor, uvidíme téměř ve všech těchto sekcích podobnou implementaci, jako je v kódu níže. Díky tomuto se omezi variabilita jednotlivých částí a kód tak bude přehlednější. Každá z těchto komponent tak funguje prakticky na stejné logice a struktuře, kterou lze opakovaně využívat při implementaci potenciálně dalších těchto sekcí.

```
interface HeadingControlSectionProps {
  editor: Editor;
}

const HeadingControlSection = ({ editor }: HeadingControlSectionProps) => {
  const headingControls: Control[] = [
    {
      tooltip: "Heading 1",
      onClick: () => editor.chain().focus().setHeading({ level: 1 }).run(),
      disabled: false,
      icon: <BsTypeH1 style={{ fontSize: "1.2rem" }} />,
      isToggled: editor.isActive("heading", { level: 1 }),
    },
    {
      tooltip: "Heading 2",
      onClick: () => editor.chain().focus().setHeading({ level: 2 }).run(),
      disabled: false,
      icon: <BsTypeH2 style={{ fontSize: "1.2rem" }} />,
      isToggled: editor.isActive("heading", { level: 2 }),
    },
    {
      tooltip: "Heading 3",
      onClick: () => editor.chain().focus().setHeading({ level: 3 }).run(),
      disabled: false,
      icon: <BsTypeH3 style={{ fontSize: "1.2rem" }} />,
      isToggled: editor.isActive("heading", { level: 3 }),
    },
  ];
};
```

Komponenta sekce přebírá jako prop editor z Tiptap, díky kterému můžeme napojit funkce na jednotlivá tlačítka. Pro účely definování struktury tlačítek a funkcí slouží pole objektů, které lze vidět v ukázce a které obsahuje tooltip, který definuje text a který se zobrazí v momentě, kdy uživatel ukáže myši na tlačítko. Tento tooltip slouží pro zlepšení uživatelské přívětivosti. Dále objekt obsahuje onClick funkci, která využívá předem definované chování tlačítka pomocí editoru. Parametr disabled definuje, zda je možné na tlačítko kliknout. Ikona, která je zobrazena v tlačítku a která koresponduje s funkcí tlačítka. Následuje identifikátor isToggled, který obsahuje boolean hodnotu, jestli je tato funkce pro vybranou část textu aplikována. Díky tomuto identifikátoru můžeme měnit uživatelské rozhraní a informovat tak uživatele o změnách ve textu. Uživatel tak jasně uvidí, jestli je pro vybraný text aplikováno nějaké stylování nebo ne. Toto napomáhá k zajištění intuitivního návrhu a funkčnosti editoru.

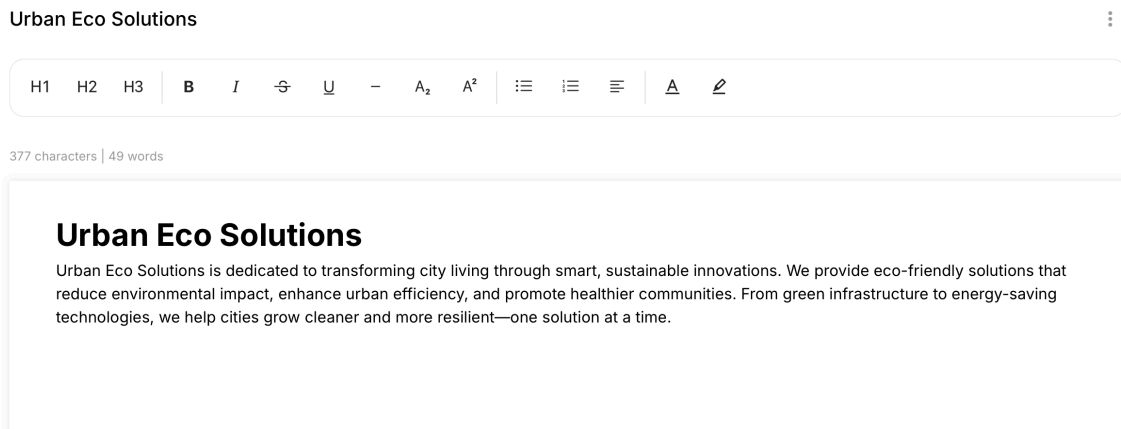
```

return (
  <>
    {headingControls.map((control, index) => {
      return (
        <ControlButton
          key={index}
          tooltip={control.tooltip}
          onClick={control.onClick}
          disabled={control.disabled}
          icon={control.icon}
          isToggled={control.isToggled}
        />
      );
    })}
  </>
);
};

export default HeadingControlSection;

```

V komponentě sekce pak následuje vykreslení jednotlivých tlačítek z pole. Díky tomuto dostaneme sadu tlačítek, jejichž struktura a chování je definována předem. Implementace je vhodná zejména proto, že komponenta a celý kód je přehledný a snadno rozšiřitelný o další tlačítka do konkrétní sekce.



Obrázek 14 Implementovaný editor

(zdroj: vlastní)

3.4.14 Responzivita

Abychom zajistili, že aplikace bude přístupná na různých typech zařízení, bylo důležité přizpůsobit design odlišné variace velikosti displejů. To znamená, že cílem bylo vytvořit tzv. responzivní design. Toho můžeme docílit několika způsoby, ale jelikož tento projekt využívá

Material UI jako knihovnu pro uživatelské rozhraní, můžeme využít funkce této knihovny a poměrně jednoduše tak vytvořit design, které se přizpůsobí velikosti obrazovky. Za účelem responzivního designu bylo také celé uživatelské rozhraní navrženo tak, aby se celé rozložení stránky dalo poměrně jednoduše přizpůsobit. To znamená, že celá aplikace využívá například grid, tedy rozložení elementu do mřížky, kterou lze následně upravit tak, abychom definovali, jak se má tato mřížka zobrazovat podle velikosti obrazovky.

```
<Grid container spacing={2}>
  {groups.map((group) => (
    <Grid size={{ xs: 12, sm: 6, md: 4, lg: 4, xl: 3 }} key={group.id}>
      <GroupCard
        id={group.id.toString()}
        name={group.name}
        description={group.description}
        admin={group.admin}
      />
    </Grid>
  ))}
</Grid>
```

V této ukázce je implementován grid pro zobrazení skupin uživatelů. Vnější část gridu je definována jako container, čímž definujeme, že se jedná o tzv. grid layout, který obsahuje jednotlivé položky této mřížky. Jednotlivé položky tohoto gridu jsou opět definovány komponentou z Material UI jako Grid, jen již nespecifikujeme container, ale naopak definujeme size. Tento parametr přebírá objekt, který obsahuje breakpoints, neboli body zlomu, což určuje velikost obrazovky. Základní definici těchto bodů zlomu můžeme vidět v tabulce níže.

Tabulka 10 Material UI breakpoints

Velikost	Velikost obrazovky (px)
xs	0px až 600px
sm	600px až 900px
md	900px až 1200px
lg	1200px až 1536px
xl	1536px a více

(zdroj: vlastní zpracování)

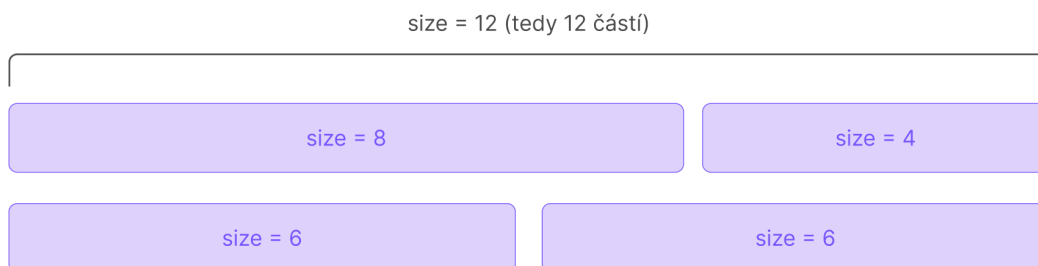
Tyto body lze upravit a případně i přidat další dle preferencí a potřeb projektu. Toho docílíme tak, že upravíme theme Material UI pomocí `createTheme` a defnujeme breakpoints. Taková implementace může vypadat následovně.

```
import { createTheme } from "@mui/material";

const theme = createTheme({
  breakpoints: {
    values: {
      mobile: 0,
      tablet: 640,
      laptop: 1024,
      desktop: 1200,
    },
  },
});
```

Tento projekt však nevyžaduje úpravu či definici vlastních bodů, a proto využíváme ty, které nabízí přímo Material UI.

Pokud se opět podíváme na Grid a definici `size`, uvidíme tyto breakpoints společně s přiřazenou hodnotou. Material UI rozděluje Grid container na 12 částí a my následně můžeme definovat kolik těchto částí má jednotlivá položka v této mřížce zabírat. Pokud bychom tedy chtěli zobrazit 3 položky vedle sebe a vytvořit tak 3 sloupce, vydělíme zmíněných 12 částí hodnotou reprezentující požadované množství sloupců, tedy 3. Tímto dostaneme hodnotu 4 a víme, že `size` nastavíme na hodnotu 4, čímž dostaneme 3 sloupce. Breakpoints nám tak pomohou definovat, kolik sloupců se vleze v jedné řadě tohoto gridu. Například pro velikost XS máme definováno, že velikost jednoho sloupce bude 12, to znamená, že každá jedna položka v tomto gridu zabere celou šířku containeru (vnější grid). Naopak velikost SM specifikuje, že při této velikosti obrazovky má každá z položek v gridu zabírat 6 částí, a tak dokážeme na obrazovce vykreslit dva sloupce vedle sebe. Tímto přístupem můžeme snadno a efektivně vytvořit responzivní uživatelské rozhraní, které je přehledné a kód je snadno rozšiřitelný o další body zlomu dle potřeb. Na tomto ukázku můžeme vidět vizuální ukázku určení velikosti sloupce pomocí `size`.



Obrázek 15 Ukázka parametru size při implementaci gridu z MUI

(zdroj: vlastní)

Využití breakpointů však lze i během stylování elementů na stránce k dosažení responzivního designu. Ukázkou může být přizpůsobení textu dle velikosti obrazovky na základě velikosti obrazovky. Definujeme velikost písma pomocí `fontSize` a na místo toho, abychom definovali pouze jednu hodnotu, nastavíme jako velikost písma objekt obsahující pod zlomu definující velikost obrazovky zařízení a následně velikost textu, která se aplikuje v případě, že obrazovka odpovídá konkrétnímu bodu.

```
<Typography
  variant="h1"
  component="h1"
  gutterBottom
  sx={{
    fontSize: { xs: "2rem", sm: "3rem", md: "4rem", lg: "5rem", xl: "6rem" },
  }}
>
```

3.4.15 Implementace úvodní webové stránky

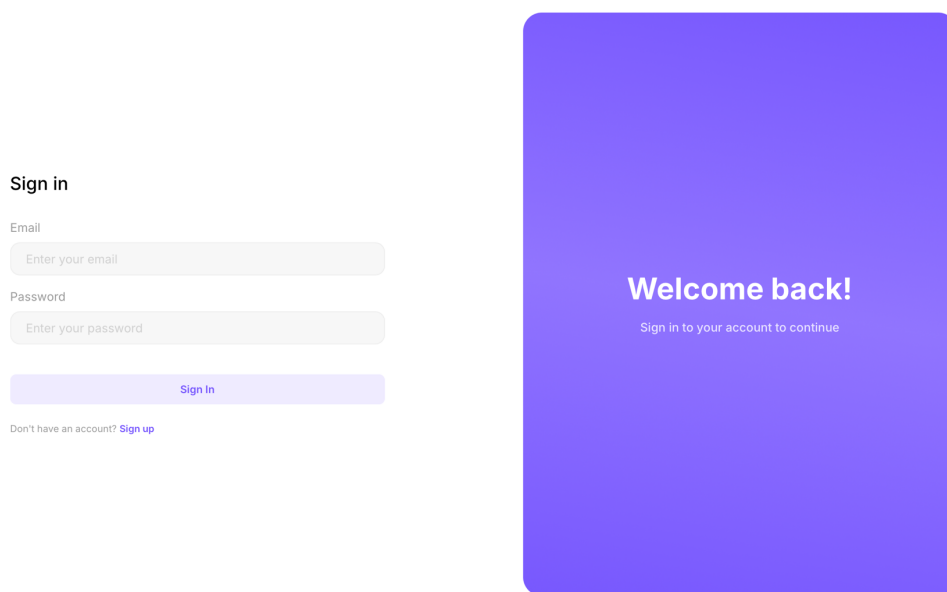
Aby aplikace neobsahovala pouze část, do které se musí člověk přihlásit a formuláře pro přihlášení či registraci, byl implementován jednoduchý web, který slouží jako prezenční web této aplikace. U této části byl kladen důraz na základní responzivní prvky a možnost přesměrování na přihlášení do aplikace. Tato část tedy nevyžaduje žádné přihlášení, a proto byla část aplikace umístěna přímo do složky `/app`, kterou zprostředkovává Next.js, čímž docílíme toho, že stránka bude první stránkou, kterou uživatel uvidí a ze které může navigovat dále do aplikace. Díky tomuto tuto část také oddělíme od zmíněného middleware, který kontroluje URL cestu, která vyžaduje přihlášení (validní JWT token).

3.5 Kompletní ukázka rozhraní aplikace

V této kapitole je ukázána kompletní implementovaná aplikace a stručný popis těchto částí. Každá z těchto částí je důležitým prvkem celé aplikace a jejich design se odvíjí od návrhů, které byly předem připraveny.

3.5.1 Přihlášení

Tato stránka slouží jako stránka pro přihlášení uživatele. Obsahuje jednoduchý formulář, který je nutné vyplnit pro pokračování do aplikace. Z této stránky je také možné přesměrovat uživatele na stránku pro registraci v případě, že uživatel ještě není zaregistrován. Pro přihlášení uživatel používá email a heslo, které jsou ověřeny a v případě úspěšného přihlášení je uživatel přesměrován dále do aplikace, která je brána jako chráněna v tom smyslu, že uživatel musí být přihlášen, aby mohl aplikaci používat.



Obrázek 16 Ukázka implementované stránky pro přihlášení

(zdroj: vlastní)

3.5.2 Registrace

Stránka pro registraci slouží pro nové uživatele, kteří si chtějí založit účet, čímž se jim zpřístupní možnost přihlášení a využívání aplikace. Na stránce se nachází registrační formulář, stejně jako je tomu u stránky s přihlášením. Z této stránky se uživatel může přesměrovat zpět na stránku pro přihlášení.

Obrázek 17 Ukázka implementované stránky pro registraci

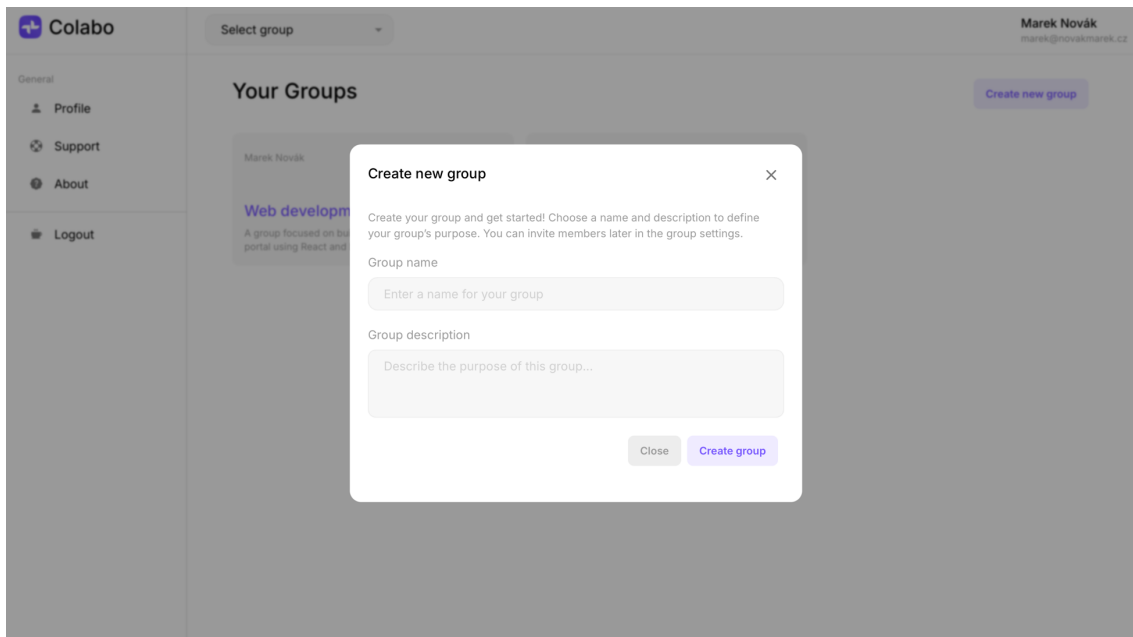
(zdroj: vlastní)

3.5.3 Přehled skupin

Přehled skupin je stránka, na kterou je uživatel přesměrován ihned po přihlášení. Slouží jako rozcestník mezi skupinami, kterých je uživatel součástí. Na této stránce může také tyto skupiny vytvářet či spravovat, a to v případě, že je jejich adminem, tedy uživatelem, který danou skupinu vytvořil.

Obrázek 18 Ukázka implementované stránky se skupinami

(zdroj: vlastní)

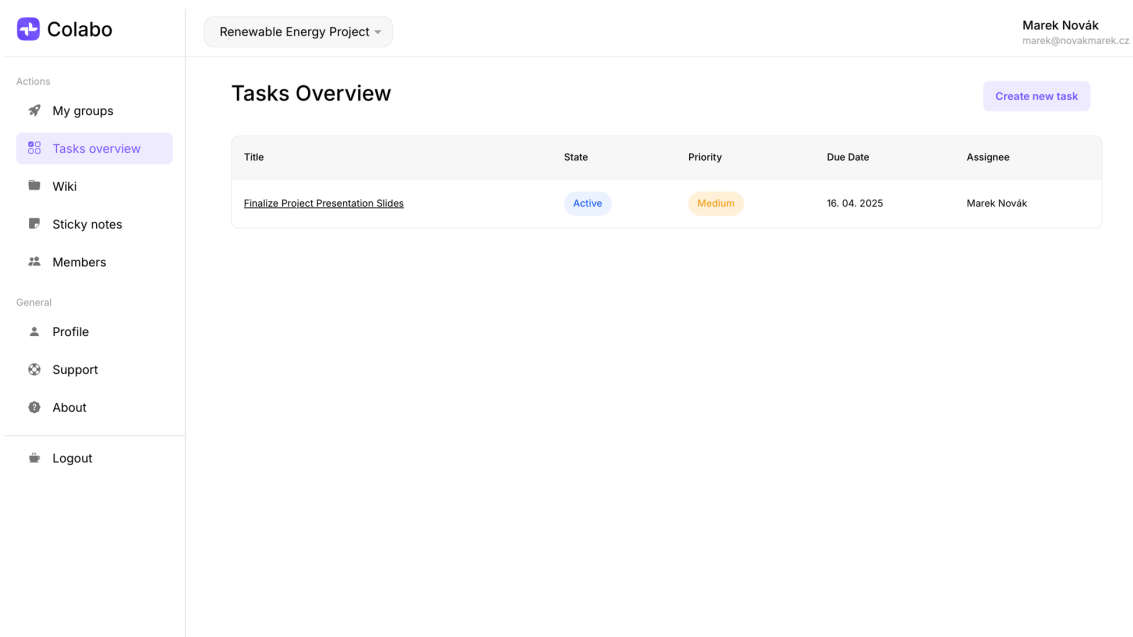


Obrázek 19 Ukázka implementace vytváření skupin

(zdroj: vlastní)

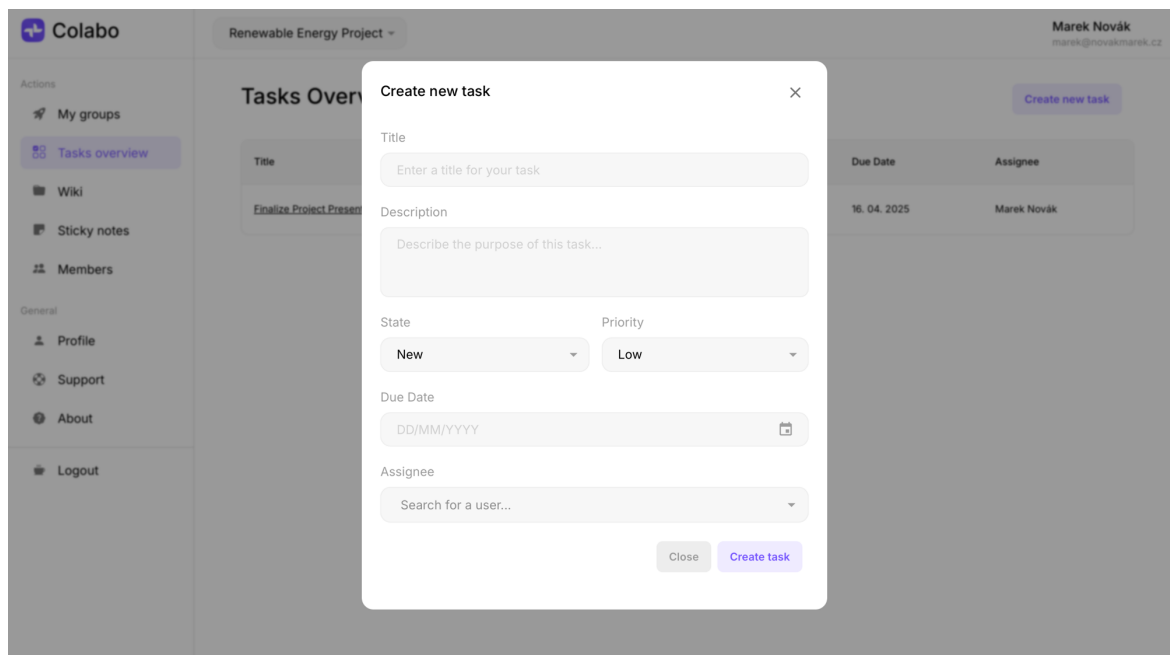
3.5.4 Tasks overview

Stránka Tasks overview se zaměřuje na zobrazení jednotlivých úkolů v dané skupině. Na této stránce se také nachází možnost vytvářet tyto úkoly a v případě, že uživatel klikne na název tohoto úkolu, bude přesměrován na jeho detail.



Obrázek 20 Ukázka implementované stránky s úkoly

(zdroj: vlastní)

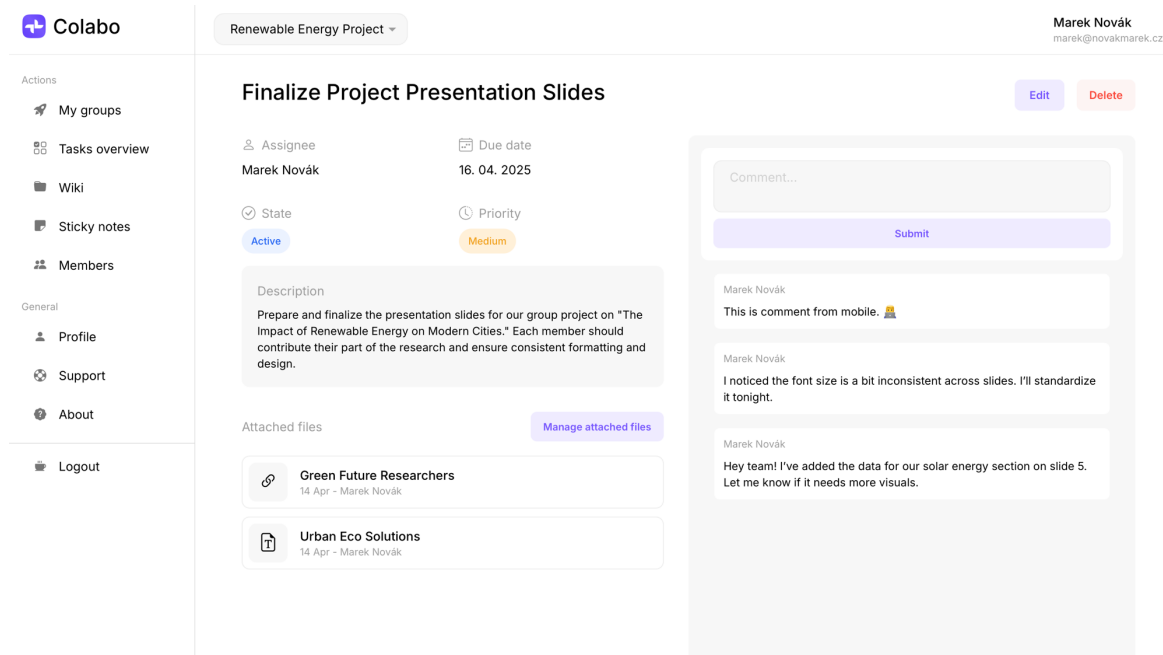


Obrázek 21 Ukázka implementace vytváření úkolů

(zdroj: vlastní)

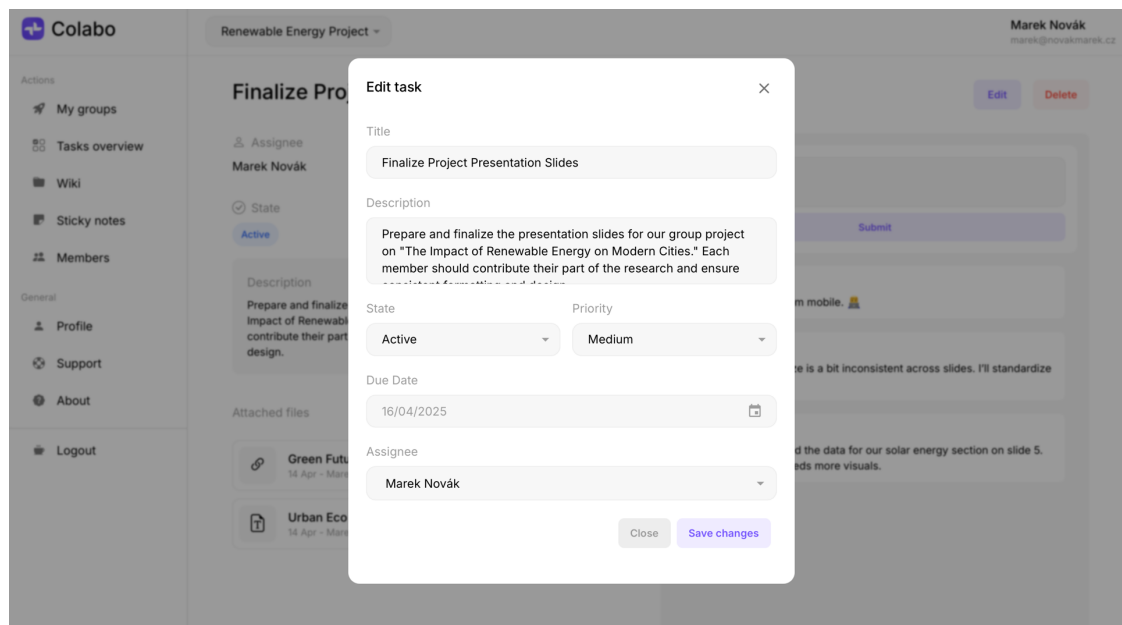
3.5.5 Detailu úkolu

Po přesměrování na detail úkolu se uživatel dostane na následující stránku, kde vidí veškeré informace o daném úkolu, jako je stav, priorita, datum splnění, kdo má úkol vypracovat či jeho podrobný popis. K tomuto úkolu lze také připojit soubory ze stránky Wiki, která je rozebrána v jedné z následujících kapitol. Rovněž se na této stránce nachází sekce s komentáři, které jsou spjaty s tímto úkolem. Nechybí také možnost editace či smazání tohoto úkolu. Stránka tak nabízí veškeré informace ke konkrétnímu úkolu na jednom místě. V případě, že uživatel chce zobrazit jeden z připojených souborů, může se na něj přesměrovat pomocí kliknutí na požadovaný soubor.



Obrázek 22 Ukázka implementace stránky pro detail úkolu

(zdroj: vlastní)



Obrázek 23 Ukázka implementace editace úkolu

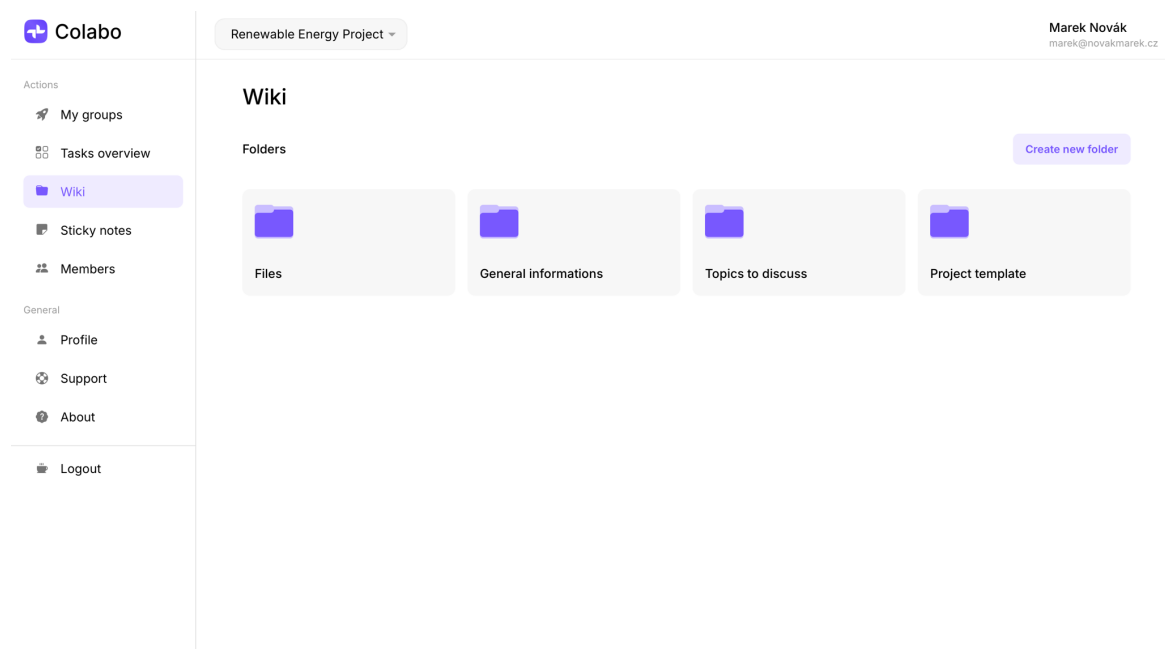
(zdroj: vlastní)

3.5.6 Wiki

Stránka Wiki se zaměřuje na sdílení informace mezi studenty ve skupině. Mohou si tak vytvářet vlastní zdroje a sdílet užitečné informace, které jim mohou pomoci při spolupráci či celkově sdílet užitečné zdroje a informace mezi ostatními členy skupiny. Stránka nabízí možnost vytvářet složky a rozdělit tak zdroje například podle tématu. Složky jsou pak zobrazeny právě na stránce Wiki. Kliknutím na jednu ze složek se dostane na stránku reprezentující obsah této složky, kde se nachází jednotlivé soubory.

Forma složek byla vybrána především z toho důvodu, že uživatelé s tímto principem budou obeznámeni díky standardnímu přístupu, který lze například nalézt v operačních systémech, kde jsou složky běžně užívány. Složky lze pak editovat v momentě, kdy je uživatel přesměrován do jedné z nich.

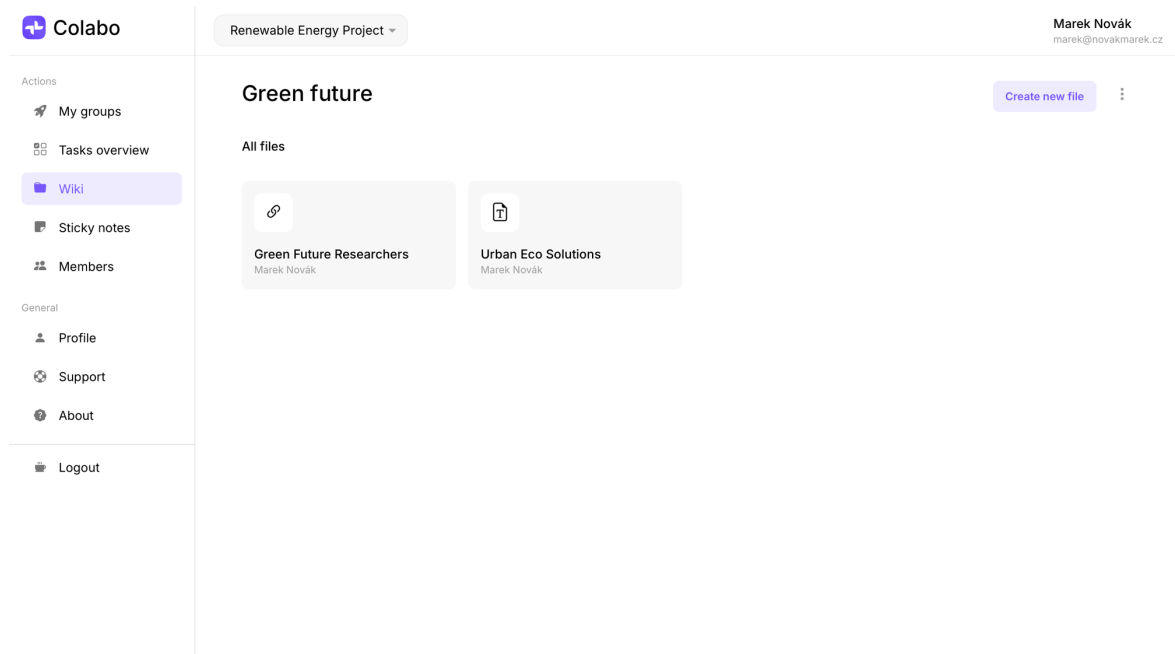
Celou stránku Wiki nemusí uživatelé vytvářet manuálně, ale je vytvořena automaticky v momentě vytvoření nové skupiny. Každá skupina tak disponuje právě jednou Wiki stránkou. Obsah následně tvoří sami uživatelé a rozhodují, jakou strukturu pro vytváření obsahu zvolí.



Obrázek 24 Ukázka implementace stránky Wiki

(zdroj: vlastní)

Stránka se soubory pak vypadá následovně. Na této stránce se dá editovat samotná složka, ale také přesměrovat na konkrétní soubory.



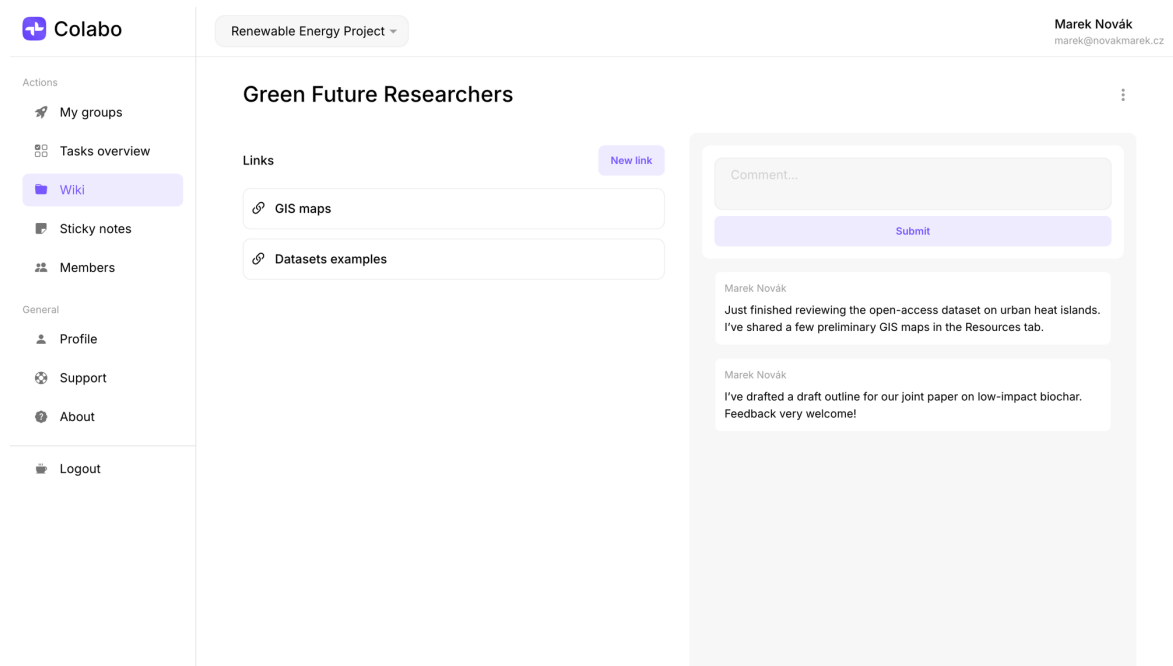
Obrázek 25 Ukázka implementace stránky s obsahem složky

(zdroj: vlastní)

3.5.7 Vytváření zdrojů

Aplikace umožňuje vytvářet dva typy sdílených zdrojů. Mezi tyto zdroje patří sdílení užitečných odkazů a sdílení zdrojů textovou formou v implementovaném editoru.

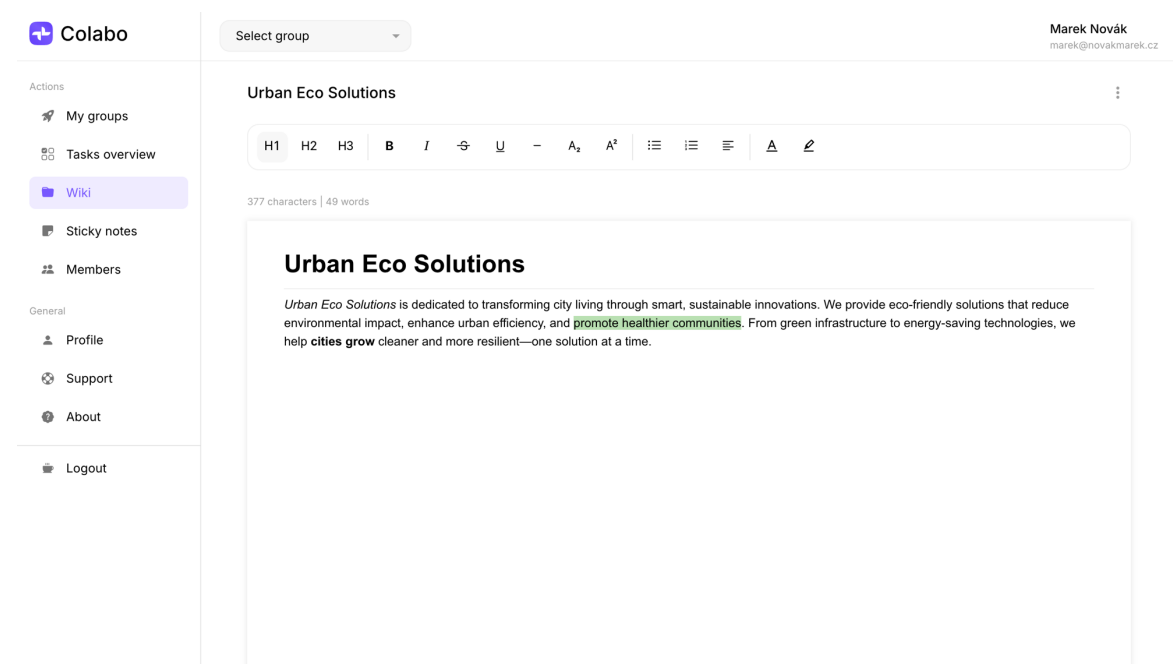
Sdílení odkazů funguje formou, kdy uživatel vloží při vytváření odkaz a také popis o jaký zdroj se jedná. Uživatelé následně mohou využívat tyto odkazy a přidávat další. Tímto způsobem se vytvoří jeden soubor, který obsahuje sadu odkazů, které lze tímto způsobem seskupovat například podle tématu, priority nebo jiným způsobem. V tomto odkazovém souboru lze také vytvářet komentáře, což nabízí možnost komentovat přidané odkazy.



Obrázek 26 Ukázka implementace stránky s odkazovými zdroji

(zdroj: vlastní)

Sdílení textových zdrojů funguje prostřednictvím implementovaného editoru, který poskytuje základní funkce běžných editorů. Nabízí funkce jako vytváření nadpisů, tučné písmo, možnost vložení seznamu/listů, zarovnání textu nebo také nastavení barev textu či jeho pozadí.

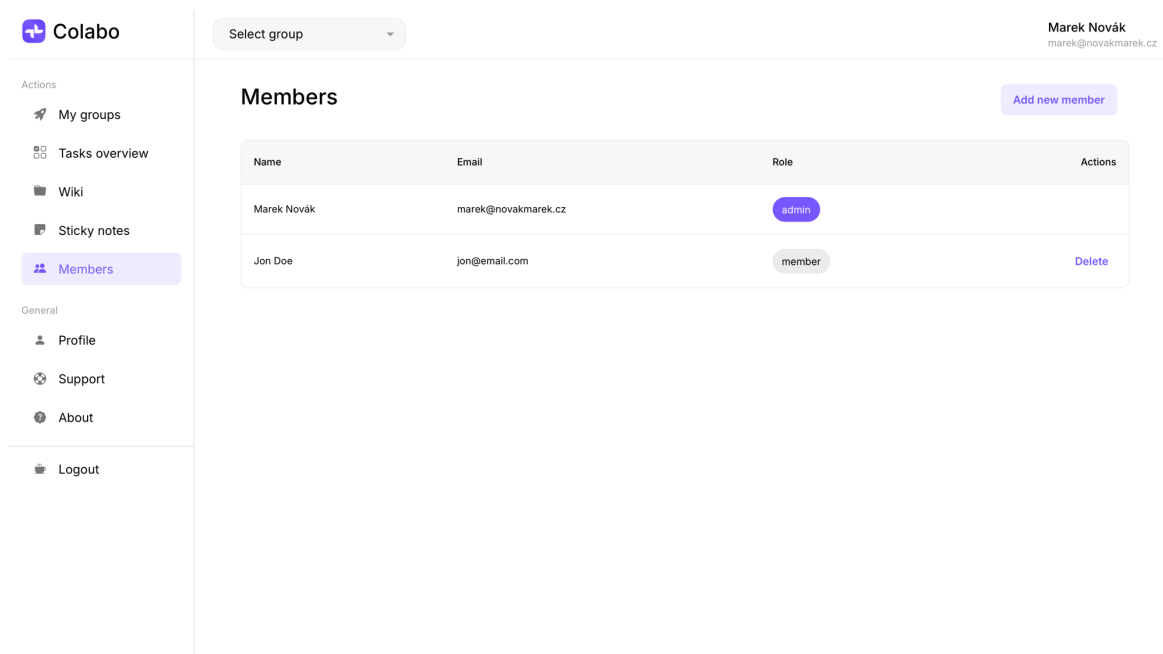


Obrázek 27 Ukázka implementace textového editoru pro vytváření textových zdrojů

(zdroj: vlastní)

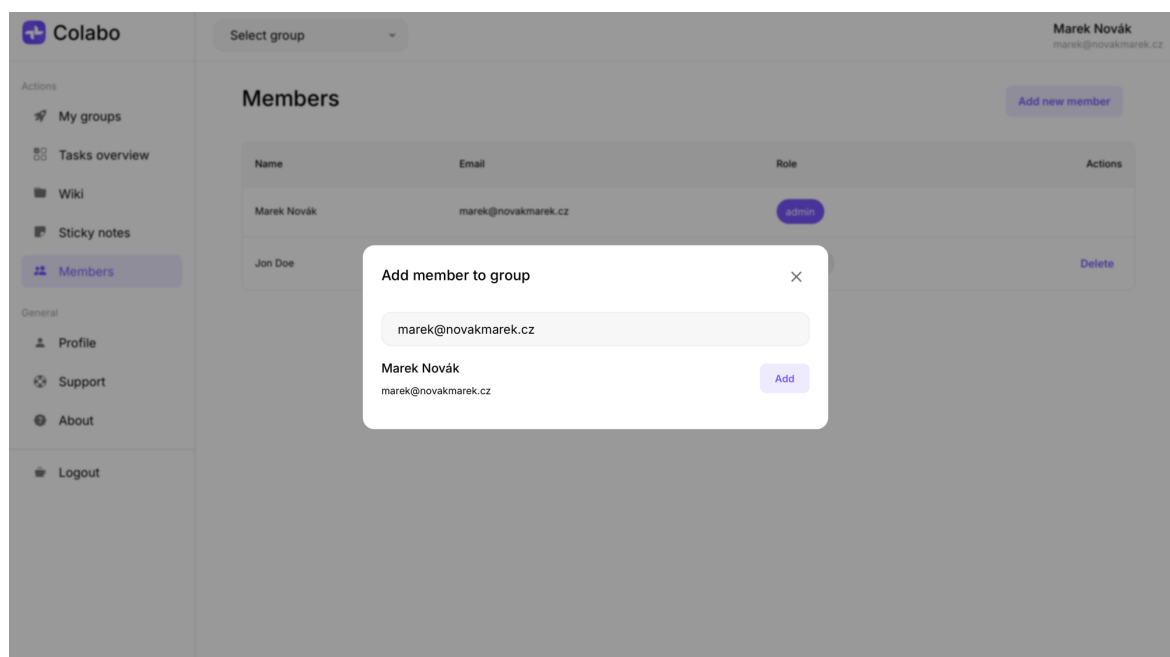
3.5.8 Správa členů

Aby bylo možné spravovat členy skupiny, byla implementována stránka, kde je možné členy přidávat nebo odebírat ze skupiny. Právo odebírat nebo přidávat nové členy má pouze admin skupiny. Členové jsou na stránce zobrazeni v tabulce.



Obrázek 28 Ukázka implementace stránky se seznamem členů skupiny

(zdroj: vlastní)



Obrázek 29 Ukázka implementace přidání nového člena do skupiny

(zdroj: vlastní)

3.6 Testování a nasazení implementované aplikace

V této kapitole jsou popsány postupy, jak byla implementovaná aplikace nasazena do produkčního prostředí a jak byla testována.

3.6.1 Testování aplikace

V následující kapitole je popsáno, jakým způsobem byla implementovaná aplikace testována. Testování pomáhá odhalit chyby, čímž se zvyšuje kvalita aplikace. Testované aplikace jsou většinou stabilnější, předvídatelné a splňují tak očekávání uživatelů. Pro účely této aplikace jsou dva druhy testů. Prvním test je na úrovni implementování kódu, kdy pomocí knihovny Jest testujeme vybrané části aplikace. Těmito částmi jsou tzv. pomocné funkce, které lze najít ve složce `utils` a které mají za úkol manipulovat s daty. Další vrstvou testů bylo manuální testování již implementované aplikace, a to v lokálním prostředí při vývoji aplikace, ale také verze aplikace, která se v produkčním prostředí přistupná běžným uživatelům. Tento druh manuálního testování byl prováděn na více zařízeních, aby se zajistila kompatibilita.

Jest – testování funkcí

Abychom se ujistili, že pomocné funkce, které jsou v aplikaci implementovány a které mohou ovlivnit správnou funkčnost aplikace, je nutné se přesvědčit, zda tyto funkce fungují tak, jak se od nich očekává. Špatná implementace by případně mohla narušit funkčnost aplikace. Ukázkou může být funkce `slugify`, která se nachází ve složce `utils`. Jedná se o funkci, která vrací tzv. slug z požadovaného řetězce. Tento slug je využit jako identifikátor některých záznamů v databázi. Příkladem může být identifikátor vytvořený z názvu složky. Ten se pak v aplikaci využívá pro získání složky z databáze. Nahrazuje tak tradiční získávání záznamu pomocí ID. Stejně tak je tento unikátní identifikátor využit v URL adrese, takže se adresa stává pro uživatele čitelnější.

```
import { slugify } from "../slugify";

describe("slugify", () => {
  test("should convert basic text to a slug", () => {
    expect(slugify("Hello World")).toBe("hello-world");
  });
});
```

Tato ukázka testu pomocí knihovny Jest ukazuje, jakým způsobem jsou funkce testovány. Describe seskupuje skupinu testů, v tomto případě skupinu testů pro funkci

slugify. Následuje popis testu, kde je stručně popsáno, co se od tohoto testu očekává. Samotný test pak obsahuje expect, kde máme výsledek funkce a toBe, kam vložíme hodnotu, kterou očekáváme, že funkce slugify vrátí. Tímto zajistíme, že funkce bude obsahovat testy, které pak můžeme kdykoliv spustit a otestovat, zda vše funguje správně. V momentě, kdy tedy upravíme námi implementovanou funkci, testy stále ověří, že funkce i po své modifikaci zajišťuje správné chování a výsledek.

```
PASS src/utils/__tests__/slugify.test.ts
slugify
  ✓ should convert basic text to a slug (1 ms)
  ✓ should remove special characters
  ✓ should replace spaces and underscores with hyphens (1 ms)
  ✓ should trim leading and trailing spaces
  ✓ should handle diacritics correctly
  ✓ should replace multiple consecutive hyphens/spaces with a single hyphen
  ✓ should return an empty string for an empty input
  ✓ should return an empty string for input with only special characters or spaces
  ✓ should handle a complex combination of cases

Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        0.214 s, estimated 1 s
```

Obrázek 30 Výsledek jednotkových testů v konzoli

(zdroj: vlastní)

Po spuštění testu pak dostaneme kompletní výpis do konzole, kde můžeme vidět veškerý přehled všech testů a jejich výsledek.

Manuální testování

Pro otestování implementované aplikace došlo k manuálnímu testování, a to na několika zařízeních a prohlížečích. Cílem manuálního testování bylo najít chyby, a to nejen ve funkčnosti aplikace, ale také implementovaného designu, čímž se otestuje uživatelská přívětivost. Pro tyto účely bylo nutné využít různé druhy zařízení a webových prohlížečů, čímž zjistíme, zda je aplikace kompatibilní pro všechna zařízení. Kompatibilita opět nezahrnuje pouze funkcionalitu, ale také třeba responzivní design, který se může chovat jinak na každém zařízení.

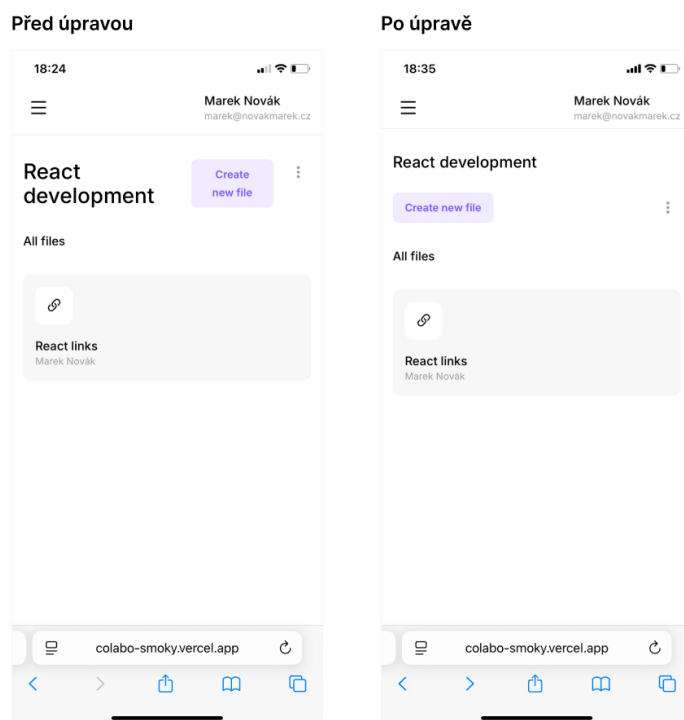
Při manuálním testování aplikace na počítači bylo využito zařízení jak s operačním systémem MacOS, tak Windows. Co se týká webových prohlížečů, tak pro MacOS bylo využito Safari a Google Chrome. Pro operační systém Windows to byl pouze Google Chrome. Testování probíhalo registrací nového uživatele, přihlášením a následném

vytváření obsahu. Byla tak vytvořena nová skupina a v ní pak byl tvořen obsah v podobě úkolů, rychlých poznámek či sdílení zdrojů pomocí skupinové wiki. V rámci těchto testů aplikace fungovalo dle očekávání a definice, která byla stanovena ve funkčních a nefunkčních požadavcích.

Pro testování aplikace na mobilních zařízeních bylo využito zařízení s operačním systémem iOS a Android. Na mobilních zařízeních se opět očekává, že aplikace bude fungovat dle předem stanovených specifikací. Pro mobilní platformu je však specifické přizpůsobení designu formou responzivity. Jak byl zmíněno v předchozích kapitolách, tak responzivita je důležitou součástí této aplikace, aby se docílilo přívětivého designu, který se adaptuje na základě velikosti obrazovky.

Při tomto testování nebylo na mobilních zařízeních zjištěna žádná chybovost či neočekávané chování aplikace. Nicméně aplikace musela být mírně upravena, co se týká responzivity, jelikož při zobrazení některých částí aplikace se některé části UI jevily ne příliš konzistentně a byly například pro menší obrazovky příliš velké. Mezi tyto prvky patří především texty. Na základě tohoto pozorování byl design mírně upraven, aby lépe splňoval požadavky na přizpůsobivý design a zlepšila se tak i uživatelská přívětivost.

Jak můžeme vidět na následujícím obrázku, kde jsou vedle sebe dvě ukázky obrazovky před a po úpravě responzivity, tak došlo k drobným změnám, díky kterým aplikace působí celkově lépe co se týká designového hlediska. Na levé straně (před úpravou) můžeme vidět, že název složky se zalamoval na dva řádky, v případě, že byl název příliš dlouhý. To stejné platí pro text v tlačítku. Za tímto účelem došlo k několika úpravám v rámci responzivity. Jak je vidět na pravé ukázce na obrázku (po úpravě), došlo ke zmenšení textu vykreslující název složky a také ke změně rozložení této hlavičky obrazovky. Text a tlačítka jsou tak v aktualizované verzi umístěny do dvou oddělených řádků, čímž se design stává nejen přehlednějším, ale také se texty budou lépe zobrazovat, a to i v případě, že budou delší. Podobné drobné změny je dobré ošetřit ještě předtím, než se aplikace dostane k uživatelům v produkčním prostředí. Tyto nedostatky mohou negativně ovlivnit používání aplikace, uživatelskou přívětivost nebo celkově vytvořit nekonzistenci designu rozhraní.



Obrázek 31 Ukázka změny designu na základě manuálních testů

(zdroj: vlastní)

3.6.2 Nasazení aplikace na Vercel

Po kompletní implementaci aplikace došlo k nasazení do produkce. Tímto se rozumí zpřístupnění aplikace veřejnosti. Pro tyto účely byla vybrána platforma Vercel, která umožňuje jednoduché a automatizované nasazení z Git repozitáře. Vercel je zároveň společnost, která stojí za vývojem frameworku Next.js, na kterém je postavena celá aplikace.

V první fázi byl na GitHubu vytvořen nový repozitář, který byl následně propojen s účtem na Vercelu. Po autorizaci repozitáře proběhla konfigurace projektu, a to především formou doplnění nezbytných proměnných prostředí, které jsou nutné pro správný chod celého projektu. V rámci tohoto nasazení byly nastaveny potřebné proměnné pro MongoDB nebo Next Auth. Jakmile byly proměnné prostředí nastaveny, Vercel zahájil automatické spouštění buildů a testů, které zároveň probíhají při jakékoliv aktualizaci v hlavní (main) větvi na GitHubu. V případě nasazení nové verze aplikace na GitHub se okamžitě spustí automatizovaný proces nasazení aplikace a výsledná aktualizovaná verze je publikována do produkčního prostředí. Díky této integraci je možné flexibilně vydávat aktualizace, opravovat chyby nebo rozšiřovat aplikaci o nové funkce, aniž by bylo nutné manuálně

spravovat server. Tímto způsobem se Vercel postará o doručení nejnovější verze všem uživatelům.

▼ Environment Variables

Key	Value i	
EXAMPLE_NAME	I9JU23NF394R6HH	—
+ Add More		

Obrázek 32 Vložení proměnných prostředí na platformě Vercel

(zdroj: vlastní)

Implementovaná aplikace je dostupná na následující adrese a slouží jako praktická ukázka.

<https://colabo-smoky.vercel.app>

ZÁVĚR

Cílem této práce bylo navrhnout a následně implementovat webovou aplikaci pro podporu skupinové spolupráce mezi studenty, která kombinuje řízení a delegaci úkolů, sdílení znalostí a tvorbu rychlých poznámek v jednom integrovaném prostředí. Práce současně představuje použité technologie nezbytných pro vývoj této aplikace a dále seznamuje čtenáře s celkovou problematikou vývoje webových aplikací a jejich základní architekturou.

V teoretické části byl nejdříve popsán celý koncept webového obsahu, a to především se zaměřením na webové aplikace. Byla popsána architektura, základní technologie pro vývoj webových aplikací a také forma komunikace mezi klientem a serverem v podobě RESTful API. Dále jsou v této části rozebrány technologie, které byly využity pro implementaci vlastního řešení kolaboračního nástroje. Mezi tyto technologie patří například React, který tvoří celé jádro klientské části aplikace a díky němuž bylo možné vytvořit celé uživatelské rozhraní aplikace. Byly popsány všechny základní koncepty Reactu, které jsou nutné pro pochopení praktické části této práce. Další klíčovou technologií, která je v této části pospána, je Next.js, což je framework pro React. Díky Next.js máme možnost implementovat komunikaci mezi klientem a serverem, a to bez nutné implementace backendové části aplikace. Tento framework nám také umožňuje renderovat obsah na straně serveru, čímž se výrazně zvýší výkon a uživatelská přívětivost. Jsou tak pospány základní koncepty Next.js, které jsou v této práci využívány a které jsou specifické pro tento framework.

Praktická část popisuje klíčové body implementace kolaboračního nástroje v podobě webové aplikace, kde jsou využity technologie popsány v praktické části. Nejdříve je popsána krátká analýza již existujících nástrojů, kde jsou zmíněny jejich výhody a nedostatky. Dále je vytvořena řada specifik v podobě funkčních a nefunkčních požadavků, kterými se pak řídil celý vývoj aplikace a které určují její směr z pohledu základních požadavků a funkčnosti. Následoval základní návrh uživatelského rozhraní, kterým se pak řídila implementace tohoto rozhraní do skutečné aplikace. Byl tak udán směr a řada pravidel co se týká návrhu základních komponent a rozložení aplikace. Následně je popsána samotná implementace aplikace, kde je popsáno, jakým způsobem se aplikace programovala. Jsou zde popsány klíčové části, jako je přípojný celého projektu k databázi, využití server actions pro komunikaci mezi serverem a klientem. Další důležitou částí byl popis zvoleného přístupu pro přihlášení a registraci uživatelů do samotné aplikace nebo popis, jakým způsobem bylo dosaženo responzivního rozhraní pomocí knihovny Material UI. U této knihovny bylo také

popsáno, jak byly jednotlivé komponenty stylovány, aby se dosáhlo výsledku, dle již zmíněných předloh designu aplikace. Po kompletní implementaci byly jednotlivé části aplikace popsány a zprostředkovány ukázky formou obrázků výsledné aplikace.

V poslední kapitole je popsána důležitost testování aplikací a jakou formou byla aplikace testována. Jednalo se především o jednotkové testy pomocných funkcí aplikace, na kterých je aplikace závislá. Dále byla aplikace testována manuálně, kdy se konečná verze aplikace manuálně testovala na několika zařízeních a v odlišných webových prohlížečích. Tento typ testů odhalil drobné nedostatky v rámci responzivního designu. Následuje tak popis, jakým způsobem byla tato problematika řešena a jsou zprostředkovány obrázky před úpravou a po úpravě.

Práce tak může čtenářům vnést zajímavý vhled do použitých technologií a přístupů. Stejně tak je možné tento návrh aplikace rozšířit či jinak modifikovat a aplikace společně s celou prací může být základní koncept skutečné aplikace v produkčním prostředí s cílem získání skutečných uživatelů a vytvoření tak platformy se zaměřením čistě na studentskou kolaboraci.

SEZNAM POUŽITÉ LITERATURY

- [1] YASAR, Kinza. What is web application (web apps) and its benefits? Online. Newton (Massachusetts), C2006–2025, This was last updated in November 2024. Dostupné z: <https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app>. [cit. 2025-05-03].
- [2] Web application. Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, C2001-2025, This page was last edited on 31 March 2025. Dostupné z: https://en.wikipedia.org/wiki/Web_application. [cit. 2025-05-04].
- [3] HEWLETT PACKARD ENTERPRISE DEVELOPMENT LP. What is a Front-End Application? Online. C2025. Dostupné z: <https://www.hpe.com/cz/en/what-is/front-end-applications.html>. [cit. 2025-05-04].
- [4] MOZILLA CORPORATION. HTML: HyperText Markup Language. Online. C1998–2025. Dostupné z: <https://developer.mozilla.org/en-US/docs/Web/HTML>. [cit. 2025-05-04].
- [5] MOZILLA CORPORATION. What is CSS? Online. C1998–2025. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Styling_basics/What_is_CSS. [cit. 2025-05-04].
- [6] MOZILLA CORPORATION. What is JavaScript? Online. C1998–2025. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/What_is_JavaScript. [cit. 2025-05-04].
- [7] COURSERA INC. What Does a Back-End Developer Do? Online. C2025. Updated on Jan 27, 2025. Dostupné z: <https://www.coursera.org/articles/back-end-developer>. [cit. 2025-05-04].
- [8] ORACLE CORPORATION. What Is a Database? Online. C2025. Dostupné z: <https://www.oracle.com/database/what-is-database/>. [cit. 2025-05-04].
- [9] ORACLE CORPORATION. What Is a Relational Database? (RDBMS)? Online. C2025. Dostupné z: <https://www.oracle.com/database/what-is-a-relational-database/>. [cit. 2025-05-04].

- [10] ORACLE CORPORATION. NoSQL Database Cloud Service. Online. C2025.
Dostupné z: <https://www.oracle.com/database/nosql/>. [cit. 2025-05-04].
- [11] AMAZON WEB SERVICES, INC. What is a RESTful API? Online. C2025.
Dostupné z: <https://aws.amazon.com/what-is/restful-api/>. [cit. 2025-05-04].
- [12] POSTMAN, INC. What are HTTP methods? Online. C2025. Dostupné z:
<https://blog.postman.com/what-are-http-methods/>. [cit. 2025-05-04].
- [13] LOKESH GUPTA. HTTP Status Codes. Online. C2023, Last Updated: August 9, 2024. Dostupné z: <https://restfulapi.net/http-status-codes/>. [cit. 2025-05-04].
- [14] REFSNES DATA AS. HTTP Status Messages. Online. C1999-2025. Dostupné z:
https://www.w3schools.com/tags/ref_httpmessages.asp. [cit. 2025-05-04].
- [15] React (software). Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, C2001-2025, This page was last edited on 15 April 2025.
Dostupné z: [https://en.wikipedia.org/wiki/React_\(software\)](https://en.wikipedia.org/wiki/React_(software)). [cit. 2025-05-04].
- [16] Document Object Model. Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, C2001-2025, Stránka byla naposledy editována 4. 8. 2023. Dostupné z: https://cs.wikipedia.org/wiki/Document_Object_Model. [cit. 2025-05-04].
- [17] META PLATFORMS, INC. Virtual DOM and Internals. Online. C2025. Dostupné z:
<https://legacy.reactjs.org/docs/faq-internals.html>. [cit. 2025-05-04].
- [18] META PLATFORMS, INC. Writing Markup with JSX. Online. C2025. Dostupné z:
<https://react.dev/learn/writing-markup-with-jsx>. [cit. 2025-05-04].
- [19] META PLATFORMS, INC. Introducing JSX. Online. C2025. Dostupné z:
<https://legacy.reactjs.org/docs/introducing-jsx.html>. [cit. 2025-05-04].
- [20] META PLATFORMS, INC. Your First Component. Online. C2025. Dostupné z:
<https://react.dev/learn/your-first-component>. [cit. 2025-05-04].
- [21] GEEKSFORGEES. React Lifecycle. Online. [n.d.], Last Updated: 08 Mar, 2025.
Dostupné z: <https://www.geeksforgeeks.org/reactjs-lifecycle-components/>. [cit. 2025-05-04].
- [22] META PLATFORMS, INC. Render and Commit. Online. [n.d.]. Dostupné z:
<https://react.dev/learn/render-and-commit>. [cit. 2025-05-04].

- [23] MAKAREVICH, Nadia. React re-renders guide: everything, all at once. Online. [n.d.], 02-08-2022. Dostupné z: <https://www.developerway.com/posts/react-re-renders-guide>. [cit. 2025-05-04].
- [24] DEV COMMUNITY. Mastering JavaScript Memory: A Beginner's Guide to Stack and Heap. Online. C2016 - 2025. Dostupné z: <https://dev.to/chintanonweb/mastering-javascript-memory-a-beginners-guide-to-stack-and-heap-57nj>. [cit. 2025-05-04].
- [25] META PLATFORMS, INC. Passing Props to a Component. Online. [n.d.]. Dostupné z: <https://react.dev/learn/passing-props-to-a-component>. [cit. 2025-05-04].
- [26] META PLATFORMS, INC. State: A Component's Memory. Online. [n.d.]. Dostupné z: <https://react.dev/learn/state-a-components-memory>. [cit. 2025-05-04].
- [27] MODI, Mohammed. React Hooks: A Beginners Guide. Online. C2013-2015. Dostupné z: <https://www.loginradius.com/blog/engineering/react-hooks-guide>. [cit. 2025-05-04].
- [28] META PLATFORMS, INC. useState. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/useState>. [cit. 2025-05-04].
- [29] META PLATFORMS, INC. useEffect. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/useEffect>. [cit. 2025-05-04].
- [30] META PLATFORMS, INC. useContext. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/useContext>. [cit. 2025-05-04].
- [31] META PLATFORMS, INC. useMemo. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/useMemo>. [cit. 2025-05-04].
- [32] META PLATFORMS, INC. useCallback. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/useCallback>. [cit. 2025-05-04].
- [33] META PLATFORMS, INC. useReducer. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/useReducer>. [cit. 2025-05-04].
- [34] META PLATFORMS, INC. useRef. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/useRef>. [cit. 2025-05-04].
- [35] META PLATFORMS, INC. memo. Online. [n.d.]. Dostupné z: <https://react.dev/reference/react/memo>. [cit. 2025-05-04].

- [36] META PLATFORMS, INC. Keeping Components Pure. Online. [n.d.]. Dostupné z: <https://react.dev/learn/keeping-components-pure>. [cit. 2025-05-04].
- [37] VERCEL, INC. Introduction. Online. C2025. Dostupné z: <https://nextjs.org/docs>. [cit. 2025-05-04].
- [38] MEDIUM. Understanding SSR in Next.js and Its Benefits. Online. [n.d.]. Dostupné z: <https://medium.com/@markminj/understanding-ssr-in-next-js-and-its-benefits-e54ffed48294>. [cit. 2025-05-04].
- [39] VERCEL, INC. Server Components. Online. C2025. Dostupné z: <https://nextjs.org/docs/app/building-your-application/rendering/server-components>. [cit. 2025-05-04].
- [40] VERCEL, INC. Routing. Online. C2025. Dostupné z: <https://nextjs.org/docs/pages/building-your-application/routing>. [cit. 2025-05-04].
- [41] VERCEL, INC. Pages and Layouts. Online. C2025. Dostupné z: <https://nextjs.org/docs/pages/building-your-application/routing/pages-and-layouts>. [cit. 2025-05-04].
- [42] VERCEL, INC. Route Handlers. Online. C2025. Dostupné z: <https://nextjs.org/docs/app/building-your-application/routing/route-handlers>. [cit. 2025-05-04].
- [43] VERCEL, INC. Client Components. Online. C2025. Dostupné z: <https://nextjs.org/docs/app/building-your-application/rendering/client-components>. [cit. 2025-05-04].
- [44] VERCEL, INC. Server and Client Composition Patterns. Online. C2025. Dostupné z: <https://nextjs.org/docs/app/building-your-application/rendering/composition-patterns>. [cit. 2025-05-04].
- [45] VERCEL, INC. Data Fetching and Caching. Online. C2025. Dostupné z: <https://nextjs.org/docs/app/building-your-application/data-fetching/fetching>. [cit. 2025-05-04].
- [46] VERCEL, INC. Server Actions and Mutations. Online. C2025. Dostupné z: <https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations>. [cit. 2025-05-04].

- [47] MICROSOFT. TypeScript for the New Programmer. Online. C2012-2025. Dostupné z: <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>. [cit. 2025-05-04].
- [48] MICROSOFT. TypeScript for JavaScript Programmers. Online. C2012-2025. Dostupné z: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>. [cit. 2025-05-04].
- [49] BAELDUNG. Statically Typed vs Dynamically Typed Languages. Online. [n.d.], Last updated: May 17, 2023. Dostupné z: <https://www.baeldung.com/cs/statically-vs-dynamically-typed-languages>. [cit. 2025-05-04].
- [50] GEEKSFORGEEKS. Data types in TypeScript. Online. [n.d.], Last Updated: 21 Jan, 2025. Dostupné z: https://www.geeksforgeeks.org/data-types-in-typescript/?utm_source=chatgpt.com. [cit. 2025-05-04].
- [51] MONGODB, INC. What Is MongoDB? Online. C2025. Dostupné z: <https://www.mongodb.com/company/what-is-mongodb>. [cit. 2025-05-04].
- [52] CHODOROW, Kristina. MongoDB: the definitive guide. 2nd ed. Sebastopol: O'Reilly, 2013. ISBN 978-1-449-34468-9.
- [53] BANKS, Alex a PORCELLO, Eve. Learning React: functional web development with React and Redux. Beijing: O'Reilly Media, 2017. ISBN 978-1-491-95462-1.
- [54] LAURENČÍK, Marek. Tvorba www stránek v HTML a CSS. Průvodce. Praha: Grada Publishing, 2019. ISBN 978-80-271-2241-7.

SEZNAM OBRÁZKŮ

Obrázek 1 Vytvoření stránky pomocí App router.....	36
Obrázek 2 Nested routing	36
Obrázek 3 Dynamic routing.....	37
Obrázek 4 Vizuální ukázky struktury aplikace	54
Obrázek 5 Návrh stránky s úkoly ve skupině	55
Obrázek 6 Návrh stránky detail úkolu	55
Obrázek 7 Návrh stránky Wiki skupiny obsahující složky.....	56
Obrázek 8 Návrh stránky detailu složky obsahující soubory.....	56
Obrázek 9 Struktura složky app	58
Obrázek 10 Struktura složky src	59
Obrázek 11 Div element pro zobrazení dialogu v prohlížeči.....	63
Obrázek 12 MongoDB Compass	67
Obrázek 13 Ukázka formátu hesla po šifrování pomocí hash	79
Obrázek 14 Implementovaný editor.....	84
Obrázek 15 Ukázka parametru size při implementaci gridu z MUI	87
Obrázek 16 Ukázka implementované stránky pro přihlášení	88
Obrázek 17 Ukázka implementované stránky pro registraci	89
Obrázek 18 Ukázka implementované stránky se skupinami	89
Obrázek 19 Ukázka implementace vytváření skupin.....	90
Obrázek 20 Ukázka implementované stránky s úkoly.....	90
Obrázek 21 Ukázka implementace vytváření úkolů	91
Obrázek 22 Ukázka implementace stránky pro detail úkolu	92
Obrázek 23 Ukázka implementace editace úkolu	92
Obrázek 24 Ukázka implementace stránky Wiki.....	93
Obrázek 25 Ukázka implementace stránky s obsahem složky	94
Obrázek 26 Ukázka implementace stránky s odkazovými zdroji	95
Obrázek 27 Ukázka implementace textového editoru pro vytváření textových zdrojů.....	95
Obrázek 28 Ukázka implementace stránky se seznamem členů skupiny	96
Obrázek 29 Ukázka implementace přidání nového člena do skupiny	96
Obrázek 30 Výsledek jednotkových testů v konzoli.....	98
Obrázek 31 Ukázka změny designu na základě manuálních testů.....	100
Obrázek 32 Vložení proměnných prostředí na platformě Vercel	101

SEZNAM TABULEK

Tabulka 1 Úspěšné stavové kódy.....	22
Tabulka 2 Přesměrování stavové kódy	23
Tabulka 3 Chyba na straně klienta stavové kódy	23
Tabulka 4 Chyba na straně serveru stavové kódy.....	24
Tabulka 5 Vhodné využití serverových a klientských komponent.....	41
Tabulka 6 TypeScript – přehled primitivních datových typů	45
Tabulka 7 TypeScript – přehled objektových datových typů	45
Tabulka 8 Přehled funkcionálních požadavků.....	50
Tabulka 9 Přehled nefunkcionálních požadavků	51
Tabulka 10 Material UI breakpoints	85

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
UX	User Experience
UI	User Interface
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
DBMS	Database Management Systém
SQL	Structured Query Language
CRUD	Create, Read, Update, Delete
DOM	Document Object Model
XML	Extensible Markup Language
SEO	Search Engine Optimization
URL	Uniform Resource Locator
SSR	Server Side Rendering
JSON	JavaScript Object Notation
JWT	JSON Web Token

SEZNAM PŘÍLOH

Příloha P I: přílohy

PŘÍLOHA P I: RILOHY

Přílohy přiložené k diplomové práci zahrnují:

- Zdrojové kódy implementované aplikace z praktické části.