Česká zemědělská univerzita v Praze
Provozně ekonomická fakulta
Katedra informačních technologií

Bakalářská práce

Front-end s použitím JavaScript knihovny React

Mikhail Ishutin

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Mikhail Ishutin

Informatika

Název práce

Front-end s použitim JavaScript knihovny React

Název anglicky

Front-end using the JavaScript library React.

Cíle práce

Bakalářská práce je tematicky zaměřena na problematiku front-endových technologií. Hlavním cílempráce je analýza možnosti využiti knihovny React s pilotním nasazením tvorby vybraného front-enduaplikace se zaměřením na návrh a implementaci funkčních komponentů, využiti stavu a vlastnosti komponent, správu dat pomocí knihovny Redux a implementaci navigace pomocí React Routeru.

Metodika

Metodika řešené problematiky bakalářské práce je založena na studiu a analýze odborných informačníchzdrojů. Teoretická část se zabývá architekturou Reactu se zaměřením na jeho základní principy a výhody oproti jiným frameworkům. Dále bude vysvětlena problematika vývoje moderního a responzivního fronten- du a jak React může pomoci s timto problémem. V praktické části práce bude provedena analýza možnostivyužiti knihovny React s následnou implementací responzivního front-endu a budou představeny konkrét-ní postupy a technologie použité při vývoji. Na základě syntézy teoretických poznatků a výsledků praktickéčásti budou formulovány závěry bakalářské práce.

Doporučený rozsah práce

40 – 50 stran textu.

Klíčová slova

JavaScript , React, JSX, front-end, Redux

Doporučené zdroje informací

DOMES, Martin. Tvorba WWW stránek pro úplné začatečníky.1.vyd.Brno: Computer Press, 2012.ISBN 978-80-251-2160-3

HOLZNER, S. JavaScript : profesionálně : [kompletní referenční příručka]. Praha: Mobil Media, 2003. ISBN 80-86593-40-1.

Oficiální dokumentace Reactu eactgirls.medium.com

O'Reilly Media;React: Up & Running: Building Web Applications 1st edition (August 16, 2016).ISBN978-1491931820

PEHLIVANIAN, A., NGUYEN D., Javascript okamžitě. Brno: Computer Press, 2014. ISBN 978-80-251-4163-2.

POWELL, Thomas A. Web design: kompletní průvodce. Vyd. 1. Překlad Petr Matějů. Brno: Computer Press, 2004, 818 s. ISBN 80-722-6949-6.

SUEHRING, S. JavaScript : krok za krokem. Brno: Computer Press, 2008. ISBN 978-80-251-2241-9.

THAU. Velký průvodce JavaScriptem: tvorba interaktivních webových stránek v praxi. Praha: Grada, 2009. ISBN 978-80-247-2211-5.

ZAKAS, Nicholas: JavaScript pro webové vývojáře. Computer Press, 2009. EAN 9788025125090.

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

doc. Ing. Pavel Šimek, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 4. 7. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 10. 03. 2024

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci " Front-end s použitím JavaScript knihovny React" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 14.03.2024

Poděkování

Rád bych poděkovala vedoucímu bakalářské práce Ing. Pavlu Šimkovi, Ph.D za odbornou přípravu a metodologickou pomoc při zpracování mé práce

Front-end s použitím JavaScript knihovny React

Abstrakt

Tato bakalářská práce se zabývá analýzou možností využití populární JavaScriptové knihovny React pro vývoj moderních front-endových aplikací. Hlavní důraz je kladen na návrh a implementaci funkčních komponent, správu aplikačního stavu pomocí Redux knihovny a integraci směrování prostřednictvím React Routeru.

V teoretické části jsou objasněny architektonické principy Reactu, jako jsou komponenty, virtuální DOM nebo JSX syntaxe. Pozornost je věnována také popisu problematiky vývoje responsivních a výkonných uživatelských rozhraní a výhodám, které React v této oblasti poskytuje.

Praktická část se zaměřuje na analýzu konkrétních možností Reactu ve vztahu k návrhu funkčních komponent, zpracování událostí, podmíněnému renderingu nebo předávání dat. Další část se zabývá správou komplexního aplikačního stavu pomocí Redux knihovny a jejími klíčovými koncepty akce-reducer-store. V neposlední řadě je řešena implementace client-side směrování za využití React Router knihovny.

Teoretické poznatky jsou následně aplikovány při vývoji pilotní front-endové aplikace, která demonstruje reálné nasazení zmíněných technik Reactu. Výsledky praktické části jsou diskutovány a zhodnoceny z hlediska vhodnosti využití této knihovny pro daný typ aplikací.

Závěr práce shrnuje dosažené cíle, porovnává přínos Reactu oproti alternativním řešením a nastiňuje možnosti jejího budoucího rozšíření.

Klíčová slova: JavaScript, React, JSX, front-end, Redux

Front-end using the JavaScript library React

Abstract

This bachelor's thesis deals with the analysis of possibilities of using the popular JavaScript library React for developing modern front-end applications. The main emphasis is placed on designing and implementing functional components, managing application state using the Redux library, and integrating routing via React Router.

The theoretical part clarifies the architectural principles of React, such as components, virtual DOM, or JSX syntax. Attention is also paid to describing the development of responsive and performant user interfaces and the advantages that React provides in this area.

The practical part focuses on analyzing React's specific capabilities in relation to designing functional components, handling events, conditional rendering, or passing data. Another part deals with managing complex application state using the Redux library and its key action-reducer-store concepts. Last but not least, the implementation of client-side routing using the React Router library is addressed.

The theoretical knowledge is subsequently applied in the development of a pilot front-end application, which demonstrates the real deployment of the mentioned React techniques. The results of the practical part are discussed and evaluated in terms of the suitability of using this library for the given type of applications.

The conclusion summarizes the achieved objectives, compares the benefits of React with alternative solutions, and outlines possibilities for its future extension.

Keywords: JavaScript, React, JSX, front-end, Redux

Obsah

1. Úvod

Rozvoj webových technologií v posledních letech zažil obrovský růst a přinesl řadu výzev pro vývoj moderních webových aplikací. S rostoucími nároky uživatelů na uživatelský zážitek, výkon a interaktivitu se čím dál tím více klade důraz na efektivní vývoj front-endu. V tomto kontextu se react-ná knihovna React, vyvíjená společností Facebook, stala jednou z nejpopulárnějších a nejvlivnějších knihoven pro tvorbu uživatelských rozhraní.

React, založený na architektuře založené na komponentách, přináší revoluční přístup k vývoji front-endových aplikací. Jeho hlavní předností je schopnost efektivně spravovat a aktualizovat složité uživatelské rozhraní, což vede k vyšší výkonnosti a lepší uživatelské zkušenosti. Díky své modulární a znovupoužitelné povaze umožňuje React vývojářům soustředit se na tvorbu jednotlivých komponent, které lze snadno kombinovat a znovu využívat v rámci celé aplikace.

Kromě samotného Reactu existuje rozsáhlý ekosystém souvisejících knihoven a nástrojů, které dále zvyšují produktivitu a flexibilitu vývoje front-endu. Mezi ně patří Redux pro efektivní správu stavu aplikace, React Router pro implementaci robustní navigace a mnoho dalších.

Tato bakalářská práce se zabývá analýzou možností využití knihovny React s cílem poskytnout komplexní pohled na vývoj moderních front-endových aplikací. Teoretická část práce se zaměřuje na studium architektury Reactu, jeho základních principů a výhod oproti jiným frameworkům. Dále se věnuje problematice vývoje responzivních front-endů a tomu, jak React může pomoci s těmito výzvami.

V praktické části práce je proveden pilotní návrh a implementace vybraného front-endu aplikace s využitím knihovny React. Pozornost je věnována návrhu a implementaci funkčních komponent, využití stavu a vlastností komponent, správě dat pomocí knihovny Redux a implementaci navigace pomocí React Routeru. Cílem je představit konkrétní postupy a technologie používané při vývoji moderních front-endových aplikací s Reactem.

Na základě syntézy teoretických poznatků a výsledků praktické části jsou na závěr formulovány závěry a doporučení pro využití Reactu při vývoji front-endových aplikací. Tato práce má za cíl poskytnout čtenářům komplexní vhled do problematiky front-endových technologií s důrazem na React a přispět k lepšímu pochopení výhod a potenciálu této knihovny.

2. Cíl práce a metodika

2.1 Cíl práce

Bakalářská práce je tematicky zaměřena na problematiku front-endových technologií. Hlavním cílem práce je analýza možností využití knihovny React s pilotním nasazením tvorby vybraného front-endu aplikace se zaměřením na návrh a implementaci funkčních komponentů, využití stavu a vlastností komponent, správu dat pomocí knihovny Redux a implementaci navigace pomocí React Routeru.

2.2 Metodika

Metodika řešené problematiky bakalářské práce je založena na studiu a analýze odborných informačních zdrojů. Teoretická část se zabývá architekturou Reactu se zaměřením na jeho základní principy a výhody oproti jiným frameworkům. Dále bude vysvětlena problematika vývoje moderního a responzivního frontendu a jak React může pomoci s tímto problémem. V praktické části práce bude provedena analýza možností využití knihovny React s následnou implementací responzivního front-endu a budou představeny konkrétní postupy a technologie použité při vývoji. Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry bakalářské práce.

3. Teoretická část

3.1 Koncept Single Page Applications (SPA)

Single Page Applications (SPA) představují moderní přístup k vývoji webových aplikací. Na rozdíl od tradičních webových stránek, které při přechodu mezi jednotlivými částmi aplikace celkově obnovují obsah a přenášejí nová data ze serveru, SPA fungují jako jediná stránka.

Po prvotním načtení SPA ze serveru dochází k vykreslení počátečního stavu aplikace v prohlížeči. Veškeré další interakce a navigace mezi částmi aplikace jsou prováděny na straně klienta (v prohlížeči) bez nutnosti opětovného načítání celé stránky. Pouze potřebná data jsou dynamicky získávána ze serveru pomocí API volání. (1)

Díky tomu SPA poskytují plynulý a nepřerušovaný zážitek podobný desktopovým aplikacím. Odpadá prodleva způsobená zpracováním na serveru a přenosem celé nové stránky. Uživatel vnímá aplikaci jako velmi responzivní.

Architektura SPA typicky zahrnuje oddělenou front-endovou a back-endovou část. Front-end obvykle využívá JavaScriptové frameworky/knihovny jako React, Angular či Vue ke správě stavu aplikace a vykreslování komponent. Back-end zajišťuje podporu pro API rozhraní a případně zpracování mimo prohlížeč.

Výhody SPA jsou lepší uživatelský zážitek, vysoká responzivita, efektivnější využití šířky pásma a možnosti modularizace aplikace. Naopak nevýhodami mohou být inicializační zpoždění při načítání většího množství kódu nebo výzvy spojené s SEO optimalizací. (1)

3.2 Document Object Model

Objektový model dokumentu (DOM) je způsob interakce s HTML a XML dokumenty prostřednictvím programovacího rozhraní (API). DOM umožňuje přistupovat ke struktuře dokumentu a měnit jeho obsah, formátování a organizaci. Dokument v DOM je reprezentován jako strom uzlů a objektů, které mají své atributy a funkce. Tímto způsobem DOM spojuje webovou stránku s jazyky skriptů nebo programování.

Webová stránka je dokument. Dokument lze zobrazit v okně prohlížeče nebo v kódu HTML. Jedná se o stejný dokument, ale s různými způsoby zobrazení. DOM poskytuje další

způsob práce s dokumentem, který je založen na objektově orientovaném přístupu, který umožňuje jeho modifikaci pomocí jazyka skriptů, například JavaScriptu.

Standardy W3C[1] DOM[2] a WHATWG[3] DOM definují základ DOM, který je implementován ve většině moderních prohlížečů. Některé prohlížeče přidávají své rozšíření k tomuto standardu, proto je třeba brát v úvahu kompatibilitu různých možností DOM s každým prohlížečem. (7)

3.2.1 Uzly DOM

Uzly dokumentu[4] jsou vrcholy stromu DOM a jsou rodičovskými uzly pro všechny ostatní prvky stránky. Zahrnují element <html>, který je kořenovým prvkem celého stromu DOM, a další elementy, jako jsou <head> a <body>.

Uzly elementů [5] představují prvky na HTML stránce, jako jsou , <p>, <h1>, <a> a další. Jsou umístěny uvnitř uzlů dokumentu a tvoří hierarchii prvků na stránce. Každý prvek může mít své podřízené prvky, které jsou jeho potomky ve stromu DOM. (5)

Textové uzly[6] představují textový obsah na HTML elementech. Například, pokud je element <p> s textem "Ahoj světe!", pak toto je textový uzel, který je potomkem elementu <p>.

Uzly atributů[7] představují atributy na HTML elementech. Například, element <a> může mít atribut "href", který určuje cílovou adresu odkazu. Tento atribut je reprezentován jako uzel ve stromu DOM.

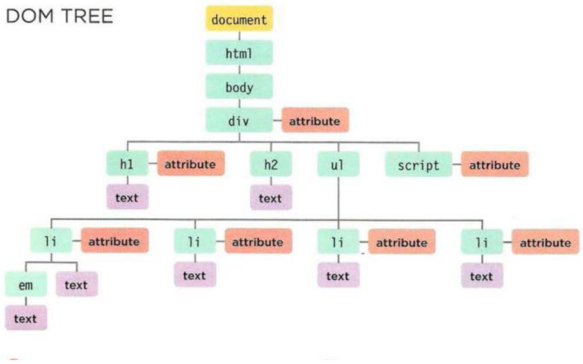
Strom DOM představuje objektový model dokumentu, a každý uzel je objektem s odpovídajícími metodami a vlastnostmi. JavaScript umožňuje programátorům interagovat s prvky stránky a měnit jejich obsah, styly a vlastnosti, včetně textových uzlů a atributů.

Změny, které jsou pomocí JavaScriptu provedeny ve stromu DOM, se okamžitě projeví na stránce, což umožňuje vytvářet interaktivní a dynamická uživatelská rozhraní. Strom DOM také poskytuje pohodlný způsob, jak skripty a programy interagují s webovou stránkou, získávají a aktualizují data, reagují na akce uživatele a vytvářejí animace a efekty.

---

[1] World Wide Web Consortium, W3C)
[2] Document Object Model
[3] Web Hypertext Application Technology Working Group
[4] Document nodes
[5] Element nodes
[6] Text nodes
[7] Attribute nodes

Obrázek 1 DUCKETT,Jon. JavaScript and jQuery:



Obrázek 2 DUCKETT,Jon. JavaScript and jQuery

Je však třeba poznamenat, že časté manipulace se stromem DOM mohou ovlivnit výkon a rychlost načítání stránky, proto se doporučuje být opatrný a optimalizovat kód pro efektivní práci se stromem DOM. (4)

3.2.2 Práce DOM stromem

Vybrání odlišného uzlu elementu

Pro vybrání (vyznačení) konkrétního uzlu (elementu) v DOM stromu lze v jazyce JavaScript použít různé metody a vlastnosti. Vybraný element da se následně upravovat nebo manipulovat s jeho obsahem. Vybrání elementu pomocí identifikátoru: Použijeme metodu `getElementById()`, která vybere element na základě jeho unikátního identifikátoru (ID). Identifikátor elementu je definován pomocí atributu `id`. Vybrání elementu pomocí CSS selektoru:

Použijeme metodu `querySelector()`, která vybere první nalezený element odpovídající zadanému CSS selektoru. Vybrání více elementů pomocí CSS selektoru: Použijeme metodu `querySelectorAll()`, která vybere všechny elementy odpovídající zadanému CSS selektoru. Výsledek bude kolekce elementů, které lze následně procházet např. pomocí cyklu `for` nebo metody `forEach()`.

Práce s vybraným elementem

Po vybrání elementu jej můžeme upravit pomocí různých vlastností a metod. Například měnit jeho textový obsah, nastavovat styly nebo přidávat/odebírat třídy. Výběr a práce s elementy v DOM stromu je klíčovým prvkem pro interakci a manipulaci s webovými stránkami pomocí JavaScriptu. (4)

3.2.3 Vybrání více elementů (seznamy uzlů)

Pro vybrání a práci s několika elementy (skupinou) v DOM stromu lze v jazyce JavaScript použít různé metody a vlastnosti. Zde jsou některé způsoby, jak vybrat a pracovat s více elementy najednou: Vybrání více elementů pomocí třídy (CSS selektor): Použijeme metodu `querySelectorAll()`, která vybere všechny elementy odpovídající zadanému CSS selektoru. Selektor může být založen na třídě, což umožní vybrat skupinu elementů s danou třídou.

Práce s vybranými elementy (skupinou)

Když jsou elementy vybrány, dá se pracovat pomocí cyklu `for`, `forEach()` nebo jiných iterací, abychom prováděli operace na všech elementech najednou. Přidání a odebrání tříd: (2)

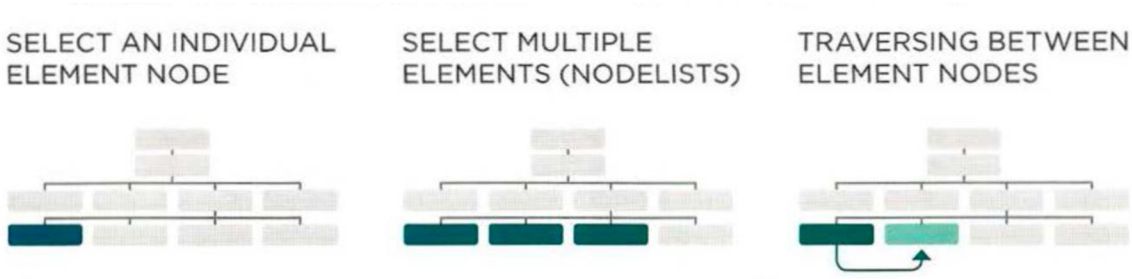
Da se přidávat nebo odebírat třídy ze všech vybraných elementů, abychom změnili jejich vzhled nebo chování. Výběr a práce s několika elementy zároveň je užitečným nástrojem při manipulaci s více prvky na webových stránkách a může pomoci urychlit a zjednodušit proces úprav a změn v DOM stromu. Da se přejít od jednoho uzlu elementu k jinému, který s ním souvisí.

parentNode

Získá rodičovský uzel daného elementu (bude vrácen pouze jeden prvek).

previousSibling / nextSibling

Získá předchozí nebo následující prvek v DOM stromu, který sousedí s tímto prvkem. firstChild / lastChild



Obrázek 3 DUCKETT,Jon. JavaScript Práce s vybranými elementy

3.2.4 Procházení DOM

Pokud je uzel elementu, lze pomocí následujících pěti vlastností vybrat jiný prvek, který s ním souvisí. Tato operace se nazývá procházení DOM.

Vlastnosti `parentNode`, `previousSibling`, `nextSibling`, `firstChild` a `lastChild` z DOM API jsou užitečné při vývoji webových stránek pro navigaci v DOM struktuře, iteraci a procházení prvků, práci s DOM stromem, zpracování událostí a filtrování a vyhledávání prvků na webové stránce. Tyto vlastnosti poskytují flexibilní a pohodlné způsoby, jak pracovat s různými prvky a jejich strukturou, což umožňuje dynamicky spravovat obsah webové stránky a provádět různé operace s prvky.

parentNode:

Toto vlastnictví odkazuje na uzel, který představuje rodičovský (obalovací) prvek v původním HTML.

previousSibling nextSiblin:

Tato vlastnost umožňuje získat předchozí a následující prvky, které jsou sousední s původním (za předpokladu, že existují).

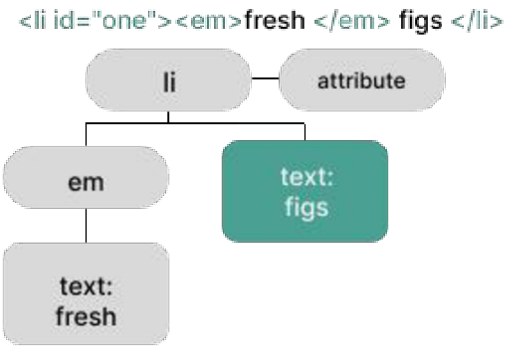
firstChild lastChild:

Tato vlastnost ukazuje na první nebo poslední dětský prvek původního elementu.

nodeValue:

Dá získat nebo změnit obsah vybraného textového uzlu pomocí vlastnosti nodeValue.

Pro získání uzlu prvku li se používá metoda getElementById(). Prvním potomkem prvku li je element em. Textový uzel je následujícím sousedním uzlem tohoto elementu em. Máme textový uzel, ke kterému můžete přistupovat pomocí vlastnosti nodeValue a změnit jeho obsah. Pro použití nodeValue musíte být na textovém uzlu, nikoli na elementu, který obsahuje text (5)



Následující kód ukazuje přístup k druhému textovému uzlu. Vrátí hodnotu "figs".

```
document.getElementById('one').firstChild.nextSibling.nodeValue;
                               ①           ②          ③         ④
```

Obrázek 4 zdroj: DUCKETT,Jon. JavaScript and jQuery:

3.3 React

3.3.1 Představení React knihovny

Od svého vzniku React vyvolal v komunitě vývojářů značný zájem a stal se významnou silou v evoluci webových technologií

S rozvojem technologií, které přineslo první generace SPA frameworků, jako jsou Angular, Ember, Knockout a Backbone, se objevila schopnost vytvářet webové aplikace, které překonávají možnosti standardního JavaScriptu a jQuery. V té době Facebook vydal svůj vlastní variantu řešení pro SPA – framework React, který přinesl s sebou inovativní přístup k vývoji rozhraní.

Na počátku ve webovém vývoji hrál server hlavní roli v renderování a zpracování dat aplikace. Avšak s příchodem Reactu a dalších moderních technologií se akcent přenesl ze serveru na klientskou část aplikace. To umožnilo vytvořit scénář, ve kterém se prohlížeč aktivně podílí na renderování rozhraní a JavaScript se stal ústředním prvkem v řízení uživatelských interakcí.

Současná éra Reactu a komponentové architektury, která se stala široce rozšířenou, změnila paradigmu vývoje webových aplikací. Komponenty se staly klíčovou koncepcí, která definuje strukturu a vizuální styl rozhraní. Tato flexibilní a výkonná architektura umožňuje efektivně vytvářet a škálovat webové aplikace.

Aspekty flexibility a omezení Reactu: hodnocení funkčnosti a výběr nástrojů (8)

React je v oblasti webového vývoje výkonným nástrojem pro vytváření uživatelských rozhraní. Přesto má knihovna flexibilitu i omezení. React je zaměřen na rozhraní a neobsahuje integrovaná řešení pro HTTP, navigaci a další funkce. Díky deklarativnímu přístupu však vývojáři mohou vytvářet jedinečná řešení. Flexibilita Reactu spočívá ve výběru nástrojů, ale znamená to také, že vývojáři musí sami najít řešení pro funkční aspekty. V porovnání s jinými frameworky je React flexibilnější, ale méně funkční. Je důležité vyvážit flexibilitu a omezení s ohledem na potřeby projektu. React, který byl vytvořen s ohledem na potřeby Facebooku, může být pro jiné projekty méně efektivní. Přesto zůstává React důležitým nástrojem pro webové vývojáře a hodnocení funkčnosti a výběr nástrojů jsou klíčové pro dosažení nejlepších výsledků. (8)

Problém, který React řeší, spočívá v efektivním vytváření a správě složitých webových aplikačních rozhraní na straně klienta. V kontextu moderního vývoje webových aplikací, zejména v kontextu vývoje rozsáhlých aplikací, se objevují potíže se správou změn zobrazení v reakci na změny dat.

React byl navržen tak, aby řešil problém vytváření rozsáhlých aplikací s časově proměnlivými daty. Na oficiálních stránkách Reactu se zdůrazňuje, že cílem knihovny je usnadnit vývoj aplikací s dynamickými daty.

Tvůrce Reactu Jordan Walk stál před úkolem aktualizovat pole automatického dokončování pro Facebook, kde data přicházela asynchronně ze serveru. Při řešení tohoto problému přišel na elegantní řešení: generování pohledů jako funkcí, což umožnilo efektivní aktualizaci zobrazení polí.

Knihovna React poskytuje vysoce efektivní, vývojářsky přívětivou a na komponentách založenou architekturu. Díky tomu, že se vyhýbá zbytečné manipulaci s DOM, poskytuje React rychlý výkon, díky čemuž si jej oblíbila nejen společnost Facebook, ale i další velké společnosti, například Instagram, PayPal, Uber, Sberbank a další. V současné době přechází na React mnoho aplikací, které začínaly s jinými technologiemi, a oceňují jeho výhody.

3.3.2 Problém, který React řeší

Problém, který React řeší, spočívá v efektivním vytváření a správě složitých webových aplikačních rozhraní na straně klienta. V kontextu moderního vývoje webových aplikací, zejména v kontextu vývoje rozsáhlých aplikací, se objevují potíže se správou změn zobrazení v reakci na změny dat. React byl navržen tak, aby řešil problém vytváření rozsáhlých aplikací s časově proměnlivými daty. [8]Na oficiálních stránkách Reactu se zdůrazňuje, že cílem knihovny je usnadnit vývoj aplikací s dynamickými daty. Tvůrce Reactu Jordan Walk stál před úkolem aktualizovat pole automatického dokončování pro Facebook, kde data přicházela asynchronně ze serveru. Při řešení tohoto problému přišel na elegantní řešení: generování pohledů jako funkcí, což umožnilo efektivní aktualizaci zobrazení polí. Knihovna React poskytuje vysoce efektivní, vývojářský přívětivou a na komponentách založenou architekturu. Díky tomu, že se vyhýbá zbytečné manipulaci s DOM, poskytuje React rychlý výkon, díky čemuž si jej oblíbila nejen společnost Facebook, ale i další velké společnosti, například Instagram, PayPal, Uber, Sberbank a další. V

[8] Single Page Application

současné době přechází na React mnoho aplikací, které začínaly s jinými technologiemi, a oceňují jeho výhody. (9)

3.3.3 Vhodné použití knihovny React

React, vyvinutý společností Facebook, je široce využívaná javascriptová knihovna pro tvorbu uživatelských rozhraní. Díky své modularitě, výkonu a reaktivnímu přístupu se React stal oblíbenou volbou pro vývoj různých typů webových aplikací. V této odborné studii se zaměříme na případy použití, kde se React jeví jako nejvhodnější řešení, a provedeme analýzu jeho výhod v daných scénářích. (6)

Jednostrankové aplikace (Single Page Applications - SPA)

Jednou z hlavních oblastí, kde React exceluje, jsou jednostránkové aplikace. Tyto aplikace načítají veškerý potřebný kód při prvním načtení a poté dynamicky aktualizují obsah bez nutnosti obnovovat celou stránku. Díky využití virtuálního DOM a efektivního diffovacího algoritmu je React schopen rychle renderovat a aktualizovat komponenty uživatelského rozhraní, což vede k plynulému a responzivnímu zážitku pro uživatele. (7)

React je vhodný pro vývoj komplexních a interaktivních SPA, jako jsou webové aplikace pro správu obsahu, e-commerce platformy, online kancelářské nástroje nebo sociální sítě. Jeho modulární architektura založená na komponentách usnadňuje správu složitých uživatelských rozhraní a umožňuje snadné znovupoužití kódu.

Webové aplikace s bohatým uživatelským rozhraním

Kromě jednostránkových aplikací je React výbornou volbou pro vývoj webových aplikací s bohatým a interaktivním uživatelským rozhraním. Díky své schopnosti efektivně renderovat a aktualizovat komponenty na základě změn stavu aplikace, je React vhodný pro vytváření vysoce responzivních a dynamických uživatelských rozhraní.

Příklady aplikací, kde React vyniká, zahrnují online editory, webové hry, vizualizační nástroje, dashboardy pro správu dat a další aplikace, které vyžadují plynulou interakci s uživatelem a rychlé reakce na vstupy.

Mobilní vývoj s React Native

React ekosystém přesahuje i do oblasti mobilního vývoje díky React Native. Tato opensource knihovna využívá stejné koncepty a design jako React, ale místo renderování komponent do webového prohlížeče je zaměřena na nativní komponenty pro iOS a Android.

Vývojáři mohou pomocí React Native vytvářet nativně vypadající mobilní aplikace s využitím JavaScriptu a React komponent. Tento přístup umožňuje sdílet kód mezi různými platformami, což vede k úsporám času a zdrojů při vývoji multiplatformních aplikací.

React Native je oblíbenou volbou pro vývoj mobilních her, e-commerce aplikací, streamovacích služeb, sociálních sítí a dalších interaktivních mobilních aplikací, které vyžadují vysoký výkon a nativní zážitek.

Rozšířené webové aplikace (Progressive Web Apps - PWA)

S rostoucí popularitou progresivních webových aplikací (PWA) se React jeví jako vhodné řešení i pro tento typ projektů. PWA kombinují výhody webových aplikací s funkcemi nativních aplikací, jako je off-line režim, push notifikace a instalace na plochu zařízení.

Díky své reaktivitě a výkonu je React schopen poskytnout plynulý a rychlý uživatelský zážitek, který je klíčový pro úspěch PWA. Modulární architektura Reactu také usnadňuje integraci různých PWA funkcí, jako je správa Service Workerů a Cache API.

PWA postavené na Reactu nacházejí uplatnění v e-commerce, zábavních aplikacích, produktivitních nástrojích a dalších scénářích, kde je požadována kombinace webového a nativního zážitku.

Integrované uživatelské rozhraní v existujících aplikacích

Kromě vývoje samostatných aplikací lze React využít také pro integraci moderních uživatelských rozhraní do stávajících webových aplikací. Díky své schopnosti renderovat

komponenty nezávisle na zbytku stránky umožňuje React postupnou modernizaci a vylepšování uživatelského rozhraní bez nutnosti přepracovávat celou aplikaci najednou.

Tato strategie je vhodná pro starší webové aplikace, kde je potřeba vylepšit interaktivitu a uživatelský zážitek, ale úplná reimplementace celé aplikace by byla příliš nákladná nebo časově náročná.

3.3.4 Porovnání knihovny React s jinými knihovnami a frameworky

Angular

Výhody Angularu vůči Reactu

Komplexnější řešení: Angular poskytuje kompletnější sadu funkcí a nástrojů pro vývoj webových aplikací, včetně směrování, správy stavu, dependency injection a dalších. Typescriptová podpora: Použití TypeScriptu v Angularu přináší výhody statické typové kontroly, lepších nástrojů pro refaktoring a ladění. Oficiální podpora: Angular je oficiálně podporován společností Google, která zajišťuje jeho aktivní vývoj a údržbu.
Nevýhody Angularu vůči Reactu (7)

Strmější učební křivka: Angular má komplexnější architekturu a vyžaduje hlubší porozumění konceptům jako dependency injection, TypeScript a dalším. Větší velikost balíčku: Angular má tendenci generovat větší balíčky kódu, což může mít dopad na načítací časy aplikace. Výkon: Ačkoli Angular poskytuje dobrou výkonnost, React a Vue.js mohou být v některých případech rychlejší díky svému odlišnému přístupu k virtuálnímu DOM .

Vue.js

Výhody Vue.js vůči Reactu:

Jednodušší integrace: Vue.js komponenty lze snadno začlenit do stávajících projektů, což usnadňuje postupnou migraci. Nižší učební křivka: Vue.js je obecně považován za snazší na pochopení a použití než React. Progresivní přijetí: Vue.js umožňuje postupné přijímání a rozšiřování funkcí podle potřeb projektu. (7)

Nevýhody Vue.js vůči Reactu:

Menší ekosystém: Ačkoli ekosystém Vue.js roste, stále má méně doplňků a nástrojů třetích stran ve srovnání s Reactem. Menší komunita: I když je komunita Vue.js aktivní, stále

má menší základnu uživatelů než React. Omezené zdroje: Dostupnost online zdrojů, kurzů a materiálů pro Vue.js může být omezená ve srovnání s Reactem.

Tabulka 1 Porovnání knihovny React s jinými knihovnami a frameworky

Funkce	Angular	React	Vue.js
Vývojový cyklus	Plný framework, který zahrnuje vše potřebné pro vývoj, jako je směrování, správa stavu a testování.	Bibliotéka pro tvorbu uživatelského rozhraní, která se zaměřuje především na práci s komponentami.	Progresivní framework s možností postupného rozšiřování. Začíná se jako jednoduchá knihovna a lze postupně přidávat další funkce dle potřeby.
Učení	Vyžaduje předchozí znalosti TypeScriptu a velmi specifických konceptů Angularu.	Jednoduchý na pochopení díky svému přístupu k psaní komponent. Vyžaduje znalost JavaScriptu a JSX.	Snadno pochopitelný, protože kombinuje známé koncepty HTML, CSS a JavaScriptu.
Velikost	Obvykle vyšší než React nebo Vue.js, protože obsahuje mnoho vestavěných funkcí.	Může být výrazně menší než Angular, protože se zaměřuje pouze na uživatelské rozhraní.	Velmi malý, když se použije v základní konfiguraci, ale může se zvětšit při použití dalších modulů a funkcí.
Komunita	Velká komunita a podpora od společnosti Google.	Aktivní komunita a mnoho dostupných balíčků a nástrojů.	Rychle rostoucí komunita s mnoha open-source balíčky a doplňky.
Výkon	Obecně dobře optimalizovaný, ale může vyžadovat více paměti a výkonu než React nebo Vue.js.	Velmi výkonný díky virtuálnímu DOM, který minimalizuje náklady na manipulaci s reálným DOM.	Efektivní a rychlý, díky možnosti optimalizace reaktivních komponent.
Dokumentace	Obsáhlá dokumentace s mnoha návody a příklady.	Dobrá dokumentace s bohatou sadou příkladů.	Dobře zdokumentovaný s jednoduchými a jasnými příklady.
Použití	Vhodný pro velké a komplexní aplikace s plnou funkcionalitou.	Ideální pro vývoj dynamických webových stránek a jednostránkových aplikací.	Ideální pro menší až středně velké aplikace, které vyžadují rychlý vývoj.

3.3.5 DOM a Virtuální DOM v React

Objektová model dokumentu (DOM) představuje programový rozhraní pro interakci JavaScriptu s různými typy dokumentů, jako jsou HTML, XML a SVG, v kontextu webového prohlížeče. DOM vytváří hierarchickou stromovou strukturu, která zobrazuje prvky a uzly dokumentu. Zajišťuje strukturovaný přístup, ukládání a manipulaci s daty uvnitř dokumentu.

Virtuální DOM představuje abstraktní vrstvu, která je implementována v mechanismu React, s cílem optimalizace aktualizací rozhraní. Funguje jako prostředník mezi aplikací a skutečným DOM, určeným pro prohlížeče. Implementací této koncepce je React schopen efektivněji zpracovávat složité procesy aktualizací uživatelského rozhraní.

V rámci Reactu popisují vývojáři komponenty v deklarativním stylu, nastavují jejich chování a vzhled v různých stavech. Vnitřní mechanismus React řídí proces aktualizace a modifikace rozhraní na základě virtuálního DOM.

V praxi vytváření velkých webových aplikací s použitím Reactu je interakce s skutečným DOM obvykle prováděna přes virtuální DOM, což přispívá ke zjednodušení řízení složitosti. Přesto je v některých scénářích, jako (10)

3.3.6 Virtuální DOM a React: Jak zlepšit výkon webových aplikací

V rámci webových aplikací, které interagují s webovými dokumenty prostřednictvím webových API v prohlížečích, je jazyk programování JavaScript nezbytný pro operace manipulace s Document Object Model. Nesporně je zde několik výzev, zejména v případě rozsáhlých webových aplikací, kde hrají důležitou roli dynamicky se měnící data. Zde se objevují problémy s efektivní manipulací s změnami spojenými s DOM. Hlavní překážkou je obtížnost detekce a následné optimalizované integrace změn do uživatelského rozhraní při dynamické metamorfóze dat. (7)

K vyřešení tohoto problému byla do knihovny React zavedena koncepce virtuálního objektového modelu dokumentu (Virtual DOM). Nachází své racionální základy v tom, aby se uchýlila k optimalizaci procesů aktualizace DOM v rámci omezení, která se vyskytují při interakci s skutečným DOM. Je důležité zdůraznit, že toto řešení nemá za cíl zrušit důležitost tradičních webových API; naopak, zůstávají nezbytnou součástí funkčnosti Reactu. Místo toho je zaměření virtuálního objektového modelu dokumentu na optimalizaci mechanismů aktualizace DOM v situacích, které jsou vystaveny omezením skutečného světa.

Původní příčina vzniku těchto nedostatků souvisí s metodou, kterou prohlížeče spravují DOM. Při interakci s prvky DOM, ať už se jedná o jejich transformaci nebo instanci, musí prohlížeč prohledat strukturované stromové prvky, aby dosáhl cílového prvku. To je pouze první akt, po kterém následuje nízká míra vizuálních mutací, které jsou zatížené zdroji. V tomto okamžiku přichází virtuální objektový model dokumentu na pomoc a poskytuje racionalizovaný přístup k aktualizaci DOM, vzhledem k těmto omezením.

Při navrhování složitých webových aplikací, které jsou vystaveny častým výkyvům dat a, v důsledku toho, DOM, se nevyhnutelně potýkají s úkolem efektivně řídit tuto dynamiku. V takových rámcích nevyhnutelně vznikají kolize a neoptimalizované manipulace s DOM, což vede ke složitosti struktury systému a obtížím pro vývojáře, spojeným s možnými negativními důsledky pro uživatelský zážitek. V takovém kontextu poskytování adekvátní výkonnosti působí jako významná úloha. Použití virtuálního objektového modelu dokumentu v kontextu knihovny React poskytuje udržitelné řešení pro tuto situaci, zejména vzhledem k jeho působivému výkonu.

Je důležité zdůraznit, že úspěšné zavedení virtuálního objektového modelu dokumentu v Reactu není dáno pouze vysokou výkonností, ale také dalšími detaily. Sem patří stabilní API, intuitivní mentální model pro vývojáře a slučitelnost mezi prohlížeči. Tyto aspekty přidávají větší význam použití virtuálního objektového modelu dokumentu než pouhé úsilí o zlepšení výkonnosti. Je třeba zdůraznit, že virtuální objektový model dokumentu, navzdory své vysoké výkonnosti, není univerzálním řešením pro zajištění vynikajícího výkonu. Ostatní výhody této koncepce zůstávají důležité v rámci spolupráce s knihovnou React.

Virtuální DOM v Reactu: Srovnání s konceptem inkrementálních změn ve vědeckých výpočtech (2)

Mechanismus virtuálního DOM v Reactu má analogii v oblasti vědeckých výpočtů, kde se snaží optimalizovat procesy aktualizace a vykreslování složitých modelů. V této analogii je virtuální DOM podobná metodám inkrementálních změn používaným v numerickém výzkumu a modelování. Podobně jako principy virtuálního DOM, metoda inkrementálních změn ve vědeckých výpočtech funguje následovně: Vytvoření počáteční modelu: Na začátku výzkumu je vytvořen základní model systému. To je analogie virtuálního DOM, která představuje abstrakci skutečného stavu rozhraní. Změny v modelu: V průběhu výzkumu jsou do modelu systému provedeny změny. To je analogie změn stavu v Reactu. Nicméně místo úplného přepočítání modelu se používají inkrementální změny k

aktualizaci pouze těch aspektů, které se změnily. Optimální analýza: Podobně jako React, který určuje optimální změny ve virtuální DOM, metoda inkrementálních změn vybírá optimální sadu výpočtů pro aktualizaci modelu. To umožňuje vyhnout se nadbytečným výpočtům a snižuje výpočetní zátěž.

Použití změn: Nakonec jsou optimalizované změny použity k modelu systému. To je analogie procesu aplikace změn v reálném DOM v Reactu.

Tato analogie pomáhá pochopit, jak virtuální DOM umožňuje optimalizovat aktualizace uživatelského rozhraní, podobně jako inkrementální změny ve vědeckých výpočtech snižují výpočetní náklady při modelování složitých systémů. To zdůrazňuje efektivitu a aktuálnost přístupu virtuálního DOM v Reactu, protože optimalizovaný přístup k aktualizacím může výrazně zlepšit výkon a odezvu webových aplikací.

3.3.7 Komponenty

Tento přístup představuje nejen nový způsob práce s datovými změnami v průběhu času, ale také zdůrazňuje používání komponent jako základní paradigma pro organizaci architektury aplikací. Komponenty slouží jako základní jednotky v React.

Komponent je entita představující část rozsáhlejší struktury. Jako mentální i vizuální nástroj usnadňují komponenty proces návrhu a vývoje uživatelských rozhraní, což zajišťuje logické strukturování a intuitivní interakci v aplikacích. Definice komponenty může být flexibilní, ale ne vše lze zařadit do kategorie komponent. Například považovat celý rozhraní za jediný komponent, bez ohledu na vnořené prvky nebo jiné komponenty, je nesprávné. Doporučuje se rozdělit různé prvky rozhraní na dílčí části, které lze poté sloučit, znovu použít a snadno reorganizovat.

3.3.8 Komponentní přístup v React

Efektivní a opakovaně použitelné

Moderní webová vývojová práce vyžaduje efektivní organizaci, správu a škálovatelnost. Komponentní přístup v React tyto úkoly řeší. Je založen na rozdělení rozhraní na komponenty, z nichž každá plní určitou funkci. Komponenty React inkapsulují data a logiku, což usnadňuje ladění a podporu. Podporují recyklaci kódu, což usnadňuje vytváření flexibilních aplikací. Vytvářením komponent pro opakované použití lze zabránit duplicitě kódu, zlepšit strukturu projektu a zajistit jednotný rozhraní. Tento přístup také usnadňuje týmovou práci.

Komponenty React poskytují metody životního cyklu, které kontrolují chování v různých fázích. To zajišťuje spolehlivost fungování.

Použití komponent v React přispívá k efektivnímu vývoji rozhraní, inkapsulaci dat a recyklaci kódu. (9)

3.3.9 Životní cyklus a život komponenty

Životnost komponenty v Reactu představuje posloupnost fází, kterými komponenta prochází během své existence, od vytvoření až po odstranění. Tyto fáze jsou definovány životním cyklem komponenty, což je sada metod, které jsou automaticky volány knihovnou React v souladu s určitými událostmi a změnami. (10)

Životní cyklus komponenty zahrnuje následující klíčové fáze:

Montáž: V této fázi je komponenta připravena k vložení do DOM. Je vytvořena pomocí konstruktoru, poté dochází k renderování komponenty, a nakonec je komponenta přidána do DOM. Tento krok je charakterizován metodami konstruktor, render a componentDidMount.

Aktualizace: Když komponenta obdrží nové vlastnosti (props) nebo změní své vnitřní stav, dochází k fázi aktualizace. V tomto případě je komponenta překreslena s přihlédnutím k novým datům. Důležité metody této fáze zahrnují render a componentDidMount

Odmontáž: Když komponenta již není potřeba a měla by být odstraněna z DOM, dochází k fázi odmontáže. Tento krok zahrnuje metodu componentWillUnmount, která poskytuje možnost uvolnit zdroje a ukončit další akce před odstraněním komponenty.

Chyba (Error): Pokud během renderování, aktualizace nebo provádění metod životního cyklu dojde k chybě, komponenta přejde do fáze chyby. V této fázi se používá metoda componentDidCatch pro zpracování chyb a udržení správně fungujícího aplikace.

Pochopení životnosti komponenty je důležité pro efektivní řízení stavu, interakce a aktualizací komponent, což vede k spolehlivějšímu a předvídatelnějšímu fungování Reactových aplikací.

3.3.10 Reaktivní stav v Reactu

Reaktivní stav v Reactu představuje data, která se mohou měnit a automaticky aktualizovat související uživatelské rozhraní. React komponenty, zejména funkční, mohou mít reaktivní stav. Při změně reaktivního stavu se komponenta automaticky překreslí, aby

odrážela nové stav bez explicitního zásahu programátora. Tento mechanismus zajišťuje pohodlnou a efektivní zpracování dynamických dat v rozhraní aplikace. (5)

3.3.11 Stav v Reactu: Dynamika a využití

Použití stavu v React aplikacích zajišťuje dynamickou funkčnost. Stav je sada dat, která jsou k dispozici v určitém okamžiku. Jedná se o klíčový prvek moderních webových platforem, včetně Facebooku. V Reactu stav umožňuje ukládat, aktualizovat a přenášet data mezi komponentami, což zajišťuje dynamickou interakci. Použití stavu je způsobeno potřebou zpracovávat dynamická data. To umožňuje vytvářet rozhraní, která reagují na vstup uživatele, a obohacují uživatelský zážitek. Nakonec je použití stavu v React aplikacích nezbytné pro moderní, dynamické webové platformy. (7)

Měnitelné a neměnné stavy

V React aplikacích existují dva hlavní způsoby práce s komponenty: prostřednictvím měnlivého stavu, který podporuje aktualizace, a prostřednictvím neměnného stavu, který zachycuje verze dat v čase. V React lze měnlivý stav aktualizovat, zatímco vlastnosti komponentů by se obvykle neměly měnit. Neměnnost znamená vytváření nových verzí datových struktur při změnách, namísto přímého změny aktuálních dat.

Použití neměnnosti je důležité v React aplikacích, protože to pomáhá předcházet nežádoucím vedlejším účinkům při aktualizaci komponent a zajišťuje efektivnější správu stavu. To také umožňuje React optimalizovat proces aktualizace komponent a udržovat historii stavu aplikace. K udržení neměnnosti lze použít specializované knihovny, jako jsou Immutable.js nebo immer.js, které poskytují pohodlné datové struktury a metody pro práci s nimi. Použití konceptu neměnnosti zlepšuje výkon, předvídatelnost a údržbu kódu v React aplikacích. (6)

3.4 Návrh a implementace funkčních komponent

3.4.1 Zpracování událostí

V Reactu se pro zpracování událostí používají obslužné funkce událostí neboli handlery. Tyto funkce jsou připojené k odpovídajícím událostem v React komponentách prostřednictvím speciálních atributů s camelCase pojmenováním (např. onClick, onSubmit, onChange atd.). (9)

Na rozdíl od běžného DOM zpracování událostí v JavaScriptu, obslužné funkce v Reactu nebudou automaticky předávat nativní události prohlížeče. Místo toho React syntetizuje vlastní cross-browser události obalující skutečné události, které jsou poté předávány obslužným funkcím. Tyto syntetické události mají stejnou rozhraní jako nativní události, ale fungují konzistentně napříč všemi prohlížeči.

Při připojování obslužných funkcí události je důležité nepředávat funkci přímo s závorkami (handleClick()), protože by došlo k okamžitému vyvolání této funkce. Místo toho se předává pouze odkaz na funkci (handleClick). React poté sám zavolá tuto funkci při výskytu dané události a předá ji objekt syntetické události jako první argument. (10)

Další argumenty lze předat obslužné funkci pomocí vyšších řádových funkcí, např: `onClick={(e) => handleClick(id, e)}`. Tímto způsobem můžeme předávat další relevantní data spolu s událostí.

V případě komponent třídy je nutné korektně navázat kontext `this` na obslužné metody událostí, protože vazba `this` v metodách tříd v JavaScriptu není automatická. To lze provést v konstruktoru třídy pomocí `bind(this)` nebo použitím arow funkcí.

Naproti tomu ve funkčních komponentách jsou handlery událostí přímé funkce, takže nevyžadují vázání kontextu `this`. Namísto sledování stavu v instanci `this` se u funkčních komponent používají Hooky jako `useState` či `useReducer`.

Obecně by měly být všechny operace zpracování událostí obsaženy v obslužných funkcích, a ne promítány přímo do JSX kódu. Tím se zvyšuje čitelnost kódu komponent a zjednodušuje se jejich údržba a testování.

3.4.2 Předávání dat komponentám (props, context)

V Reactu existují dva hlavní způsoby, jak předávat data mezi komponentami:

Props

Props (zkratka pro properties) jsou způsob předávání dat od rodičovské komponenty ke komponentě dítě. Data se předávají jako atributy při vykreslování dítěte v JSX. Dítě pak může data ze svých props číst, ale nemůže je přímo měnit, protože jsou jen pro čtení. (11)

Příklad:

```jsx
// Rodičovská komponenta
import DiteKomponenta from './DiteKomponenta';

function Rodic(props) {
  return <DiteKomponenta nazev="React" verze={18} />;
}

// Dítě komponenta
function DiteKomponenta(props) {
  return (
    <div>
      <h1>{props.nazev}</h1>
      <p>Verze: {props.verze}</p>
    </div>
  );
}
```

Obrázek 5 vlastní obrazek vs code props

Context:

Context API umožňuje předávat data hlouběji do stromové struktury komponent, aniž by musela být explicitně předávána přes každou úroveň props. Je vhodný pro globální data aplikace, nastavení apod.

Kontext se vytvoří pomocí React.createContext() a na nejvyšší úrovni, kde jsou data dostupná, je obalí Context. Provider. Poté mohou komponenty kdekoli ve stromu využít Context.Consumer nebo hook useContext k přístupu k těmto datům.

Příklad:

```jsx
// Vytvoření kontextu
const TemaContext = React.createContext();

// Poskytovatel kontextu
<TemaContext.Provider value={{tema: 'svetle'}}>
  <App />
</TemaContext.Provider>

// Komponenta spotřebovávající kontext
import {useContext} from 'react';

function DiteKomponenta() {
  const {tema} = useContext(TemaContext);
  // Použití dat tema...
}
```

Obrázek 6 vlastní obrazek VS Code Context Consumer

Props by měly být upřednostňovány pro přenos dat mezi přímými potomky, zatímco kontext je vhodný pro odstínění některých props od manual komponent nebo pro poskytování globálních dat aplikace.

3.4.3 Podmíněný rendering

Podmíněný rendering v Reactu umožňuje renderovat různé části uživatelského rozhraní na základě nějakého stavu nebo podmínky. Je to velmi užitečný princip pro vytváření dynamických a interaktivních komponent. Existuje několik způsobů, jak v Reactu dosáhnout podmíněného renderingu:

Operátor && v JSX

Umožňuje jednoduše renderovat něco, pokud je daná podmínka pravdivá. Pokud je podmínka nepravdivá, React nic nevyrenderuje.

```
const messages = props.messages;
return (
  <div>
    {messages.length > 0 && <MessageList messages={messages} />}
  </div>
);
```

Obrázek 7 Operátor && v JSX (vlastní obrazek)

Ternární operátor v JSX

Klasický ternární operátor lze použít pro renderování jedné ze dvou možných variant na základě nějaké podmínky.

```
const isLoggedIn = props.isLoggedIn;
return <div>{isLoggedIn ? <UserGreeting /> : <GuestGreeting />}</div>;
```

Obrázek 8 operátor v JSX (vlastni obrazek)

Operátor &&& a ternární operátor společně

Kombinace předchozích dvou technik může řešit i složitější scénáře podmíněného renderingu. Vkládání podmíněného renderingu do vlastních komponent Složitější případy podmíněného renderingu je vhodné vyčlenit do samostatných komponent, které pak lze znovupoužít.

```
function WarningBanner(props) {
  if (!props.warnings.length) {
    return null;
  }
  return <div>{props.warnings}</div>;
}
```

Obrázek 9 ternární operátor společně

Podmíněný rendering pomocí klíčového slova switch

Lze využít switch konstrukci k rozhodování, co renderovat na základě hodnoty určité proměnné.

Příklad komponentu, která zobrazuje tlačítko pro odhlášení pouze v případě, že je uživatel přihlášen. Bez podmíněného renderingu by muselo být tlačítko součástí komponenty za všech okolností, což by bylo matoucí a nežádoucí při nepřihlášeném stavu. Pomocí podmíněného renderingu však můžeme toto tlačítko zobrazit nebo skrýt na základě aktuálního přihlašovacího stavu. (6)

V Reactu existuje několik způsobů, jak provádět podmíněný rendering, od jednoduchého použití logických operátorů až po vyčlenění složitějších podmínek do samostatných komponent. Volba správného přístupu často závisí na konkrétním scénáři a složitosti dané podmínky.

Je důležité zmínit, že podmíněný rendering by měl být využíván pouze tam, kde je to skutečně potřebné. Zbytečné používání může vést ke snížení výkonu aplikace a složitějšímu kódu. React efektivně aktualizuje pouze ty části stromu komponent, které se skutečně změnily, takže podmíněný rendering by měl být aplikován uvážlivě.

Při provádění podmíněného renderingu je také třeba dbát na to, aby nedocházelo k nežádoucím vedlejším účinkům, jako je ztráta stavu komponent nebo neočekávané chování. Správné použití klíčů u renderováních prvků může pomoci Reactu efektivně identifikovat, které prvky mají být aktualizovány a které přidány nebo odstraněny. (6)

3.4.4 Rendering seznamů

Rendering seznamů je jednou ze základních operací, kterou vývojáři v Reactu často provádějí. Ať už se jedná o zobrazení seznamu položek, úkolů, produktů nebo jiných datových struktur, React nabízí elegantní způsob, jak tyto seznamy vykreslit do uživatelského rozhraní. (8)

Klíčovým bodem pro rendering seznamů je použití metody `map` jazyka JavaScript. Tato metoda umožňuje iterovat přes pole prvků a pro každý prvek vytvořit odpovídající React element. Výsledkem je nové pole React elementů, které může být přímo vloženo do výstupu komponenty.

Při renderování seznamů je však nutné dbát na přiřazení unikátních klíčů (`keys`) každému vykreslenému prvku. Klíče umožňují Reactu identifikovat, které položky se změnily, přidaly nebo odstranily při následujících aktualizacích. Pokud tyto klíče nejsou

přiřazeny správně, může dojít k neočekávanému chování a potenciálním problémům s výkonem.

V praxi se často stává, že jednotlivé položky seznamu mají určitý identifikátor, který lze použít jako klíč. V případě, že takovýto identifikátor neexistuje, lze jako klíč použít index položky v poli. Nicméně použití indexu jako klíče se nedoporučuje v případech, kdy se může pořadí položek v poli změnit, protože to může vést k neočekávanému chování.

Pro složitější struktury seznamů je vhodné vyčlenit kód pro renderování jednotlivých položek do samostatných komponent. Tím se zvyšuje čitelnost a udržovatelnost kódu a usnadňuje se znovupoužitelnost těchto komponent v jiných částech aplikace.

Při práci s vnořenými seznamy je nutné přiřadit unikátní klíče na každé úrovni vnořeného seznamu. Tím se zajistí, že React bude moci správně sledovat změny na všech úrovních a efektivně aktualizovat pouze ty části, které se skutečně změnily.

Celkově je rendering seznamů v Reactu poměrně přímočarý a efektivní proces, pokud jsou dodrženy správné zásady pro přiřazování klíčů a strukturování komponent. Díky tomu mohou vývojáři snadno vytvářet dynamická a responzivní uživatelská rozhraní, která se plynule přizpůsobují změnám v datech.

3.5 Správa stavu aplikace

Využijte React k efektivní správě stavu aplikace pomocí stavových proměnných a React hooks. Udržujte stav komponenty aktuální a synchronizovaný s pomocí useState hooku. Pro komplexnější aplikace použijte správu stavu pomocí Context API nebo knihoven jako Redux. Sledujte jednoduchost a efektivitu ve správě stavu, abyste zajistili robustní a snadno udržovatelný kód.

3.5.1 Problematika správy stavu v Reactu

Správa stavu (state management) je jednou z klíčových oblastí, které je potřeba věnovat pozornost při vývoji větších a komplexnějších aplikací v Reactu. Stav reprezentuje aktuální data v aplikaci, která se mohou v průběhu času měnit v reakci na různé uživatelské akce nebo události. Efektivní správa stavu je zásadní pro zajištění konzistence a prediktibility chování aplikace. (12)

V Reactu existují dva hlavní typy stavu:

1. Lokální stav (Local State)

Tento stav je spravován uvnitř jednotlivých komponent pomocí Hooks jako `useState` nebo `useReducer`.

Lokální stav je vhodný pro jednoduché scénáře, kdy data souvisejí pouze s konkrétní komponentou a jejími potomky.

Výhodou je snadná správa a jasné vymezení odpovědností.

Nevýhodou může být obtížnější sdílení dat mezi vzájemně nesouvisejícími komponentami.

2. Globální stav (Global State)

Představuje data, která jsou sdílena v rámci celé aplikace nebo jejích větších částí.

Pro správu globálního stavu se často využívají externí knihovny, jako je Redux, MobX nebo Context API v Reactu.

Globální stav umožňuje snadné sdílení dat mezi komponentami a centralizovanou správu aplikačního stavu.

Nevýhodou může být složitější nastavení a nutnost externích závislostí.

Při rozhodování o správném přístupu ke správě stavu je důležité zvážit velikost a složitost aplikace, požadavky na sdílení dat a potřebu prediktibility a škálovatelnosti.

Velké a komplexní aplikace často vyžadují robustnější řešení pro správu globálního stavu, jako je Redux. Redux zavádí principy jako jednosměrný datový tok, neměnné stavy a čisté funkce, které usnadňují vývoj, testování a ladění aplikací.

Naopak pro menší a jednodušší aplikace může být vhodnější spoléhat se na lokální stav spravovaný uvnitř komponent pomocí Hooks. Tento přístup je přímočařejší a nevyžaduje dodatečné závislosti.

Správná strategie správy stavu je klíčová pro udržení přehlednosti a škálovatelnosti React aplikací. Vhodně zvolený přístup umožňuje vývojářům lépe předvídat chování aplikace, sdílet data mezi komponentami a efektivně reagovat na změny stavu.

3.5.2 Kompozice komponent

Kompozice komponent představuje jeden ze stěžejních principů, na kterých je založen vývoj uživatelských rozhraní v rámci React knihovny. Tento přístup umožňuje budovat komplexní aplikace skládáním menších, izolovaných a znovupoužitelných stavebních bloků, jimiž jsou samotné komponenty. Tím dochází k logickému rozkladu problému na menší, snadněji uchopitelné části, což přináší řadu výhod. (10)

Primárním benefitem kompozice je zvýšení modularity a znovupoužitelnosti kódu. Jednotlivé komponenty lze kombinovat a opětovně využívat v různých částech aplikace, čímž je podporován princip DRY (Don't Repeat Yourself) a AHA (Avoid Horizontal Alignment). Aplikace se tak stává přehlednější a udržovatelnější, neboť logika je rozložena do menších, snáze pochopitelných celků.

Komponenty jsou navíc navrženy jako izolované a nezávislé jednotky, jejichž rozhraní je jasně vymezeno vstupními parametry (props). Tato izolace umožňuje snadnější testovatelnost komponent a částečně eliminuje riziko vzniku nežádoucích vedlejších efektů při provádění změn. Zároveň podporuje principy abstrakce a hierarchického uspořádání, kdy mohou komponenty obsahovat další vnořené komponenty, čímž vzniká stromová struktura odrážející přirozenou dekompozici řešeného problému.

Samotnou kompozici lze v Reactu realizovat několika technikami. Jednou z nich je přímé skládání komponent, kdy je daná komponenta renderování uvnitř jiné komponenty jako její potomek. Další možností je využití props skládání, při kterém jsou komponenty předávány jako vstupy skrze props a následně renderovány uvnitř komponenty nadřazené. V neposlední řadě lze kompozici provádět pomocí rozšiřování, kdy jsou vytvářeny specializované varianty existujících komponent skrze mechanismus dědičnosti.

Vhodná aplikace kompozičního přístupu je klíčovou podmínkou pro dosažení vysoce modulárního a jednoduše udržovatelného kódu v rámci vývoje v React knihovně. Umožňuje vývojářům řídit se zásadou jednotné zodpovědnosti a budovat robustní aplikace skládáním jednodušších, snadno pochopitelných částí, které lze efektivně kombinovat a znovu používat dle aktuálních požadavků.

3.5.3 Akce, reducery a store

V kontextu správy globálního stavu aplikace pomocí Redux knihovny jsou tři hlavní části akce, reducery a store.

Akce (Actions) jsou zdrojem informací popisujících, jaké změny mají být provedeny ve stavech aplikace. Jsou reprezentovány jednoduchými objekty obsahujícími povinný atribut "type" identifikující typ operace, jež má být vykonána. Kromě toho mohou obsahovat libovolná dodatečná data nezbytná pro provedení dané akce. Akce by měly být neměnné a vyjadřovat pouze fakta popisující změny, nikoliv způsob jejich aplikace.

Reducery (Reducers) představují čisté funkce, které na základě přijatých akcí modifikují aktuální stav aplikace a produkují její nový stav. V Reduxu je aplikační stav zastoupen jednou neměnnou datovou strukturou, nad kterou reducery provádějí výpočty čistých funkcí. Pro každý typ akce existuje příslušný reducer, který přijímá aktuální stav a akci jako vstupy a vrací výsledný nový stav aplikace. (6)

Store je centrálním uzlem, jenž uchovává globální stav celé aplikace. Obsahuje kořenový reducer aplikovaný nad celým aplikačním stavem a metody umožňující přistupovat k aktuálnímu stavu, spouštět akce a registrovat callbacky pro zpracování změn stavů. Store zajišťuje udržení neměnnosti aplikačního stavu skrze jeho kompletní nahrazení novým stavem při každé změně.

Správa globálního stavu aplikace pomocí těchto tří komponent funguje následujícím způsobem. Při výskytu události, jako je například uživatelská akce, dochází k vyvolání odpovídající akce informující o nastalé změně. Tato akce je poté předána příslušnému reduceru skrze store, který na jejím základě aktualizuje stav aplikace na novou hodnotu. Komponenty naslouchající změnám stavů jsou poté informovány a jejich příslušné části jsou znovu vykresleny s novými daty.

Princip akce-reducer-store tak zavádí principy jednotného toku dat, neměnnosti stavů a oddělení zodpovědností do správy aplikačního stavu. Umožňuje vytvářet vysoce

prediktivní aplikace s jasnými pravidly pro řízení stavu a poskytuje rozsáhlé možnosti pro ladění, testování a rozšiřitelnost.

3.5.4 Propojení Reduxu s React aplikací

Aby bylo možné využívat Redux knihovnu ke správě globálního stavu v rámci React aplikace, je nezbytné provést jejich vzájemné propojení. K tomuto účelu slouží oficiální balíček react-redux, poskytující vyšší řádové komponenty a Hooky usnadňující integraci. (12)

Proces propojení Reduxu s React aplikací lze shrnout do následujících klíčových kroků:

1. Vytvoření Redux store - Centrální uložiště aplikačního stavu je inicializováno voláním funkce createStore() z Redux knihovny, které je předán kořenový reducer aplikace.

2. Zapouzdření React komponenty - Kořenová komponenta aplikace je obalena komponentou "Provider" z balíčku react-redux. Tato komponenta umožňuje přístup ke globálnímu stavu v jejích potomcích.

3. Připojení komponent ke stavu - Pro propojení konkrétních React komponent s Redux storem existují dva hlavní přístupy. Prvním z nich je použití vyšší řádové komponenty connect() obalující cílovou komponentu a zprostředkovávající přístup ke stavu a akcím. Druhým modernějším přístupem je využití React Hooku useSelector() pro čtení části stavu a useDispatch() pro vyvolávání akcí.

4. Mapování stavu do props - Při využití HOC connect() je potřeba definovat dvě mapovací funkce - mapStateToProps() a mapDispatchToProps(), které určují, jaké části globálního stavu a akce budou v komponentě dostupné.

5. Předávání akcí do komponent - Vyvolávání akcí pro modifikaci globálního stavu probíhá ze samotných React komponent skrze dispatch funkci obdrženou z Reduxu.

Použitím oficiálního balíčku react-redux je zajištěna efektivita a optimální výkon při změnách stavu, neboť jsou aktualizovány pouze komponenty, jejichž mapovaný stav se skutečně změnil. Zároveň je udržena prediktibilita chování zajištěná Reduxem.

Při správě aplikačního stavu v Reduxu často vyvstává potřeba zpracovávat asynchronní akce, jako jsou síťové požadavky, interakce s externími API či vykonávání operací s prodlevou. Nativně však Redux neposkytuje způsob, jak tyto asynchronní operace řešit. K tomuto účelu slouží koncept middleware. (6)

Middleware v Reduxu představuje vrstvu programové logiky, která má možnost zachytit vyvolané akce, provést nad nimi další operace a poté je případně předat dalšímu middleware nebo samotné Redux store. Umožňuje tak zasahovat do životního cyklu zpracování akcí a obohacovat jej o dodatečnou funkcionalitu.

Pro správu asynchronních požadavků se v Redux ekosystému ustálily dva hlavní přístupy využívající middleware:

1. Redux Thunk middleware - Tento middleware umožňuje vyvolat akci jako funkci místo obyčejného objektu. Funkce může obsahovat asynchronní logiku a při svém dokončení vyvolat skutečnou akci předávanou do Reduxu. Thunk akce tak získávají možnost provádět asynchronní operace.

2. Redux Saga middleware - Alternativou je použití middleware Redux Saga využívajícího koncepty známé z generátorů (generator functions). Tento přístup zavádí deklarativní způsob popisování posloupností asynchronních toků pomocí generátorů a efektů definujících asynchronní činnosti.

Oba přístupy umožňují přehlednou správu asynchronních operací oddělením jejich logiky od React komponent. Asynchronní akce jsou tak zpracovávány v dedikované vrstvě middleware, zatímco komponenty pouze reagují na výsledný stav aplikace.

Použití middlewaru přináší kromě řešení asynchronity také další benefity, jako je centralizace logiky napříč aplikací, lepší znovupoužitelnost kódu či možnost snadnějšího sledování a ladění asynchronních toků. Zejména v kontextu komplexních aplikací s rozsáhlými asynchronními operacemi tak middleware představuje vhodný nástroj pro udržení přehledného a škálovatelného řešení.

3.6 Směrování v React aplikacích

Směrování v React aplikacích umožňuje navigaci mezi různými částmi aplikace bez nutnosti načítání nových stránek. K dosažení tohoto cíle můžete využít knihovny jako React Router, která poskytuje jednoduché a efektivní směrování pomocí komponent a definic cest. Díky směrování můžete vytvořit plynulý uživatelský zážitek a zároveň udržet stav aplikace synchronizovaný s URL adresou. (13)

3.6.1 Koncept client-side směrování

Koncept client-side směrování představuje přístup k implementaci navigace a směrování v rámci Single Page Applications (SPA). Na rozdíl od tradičních webových aplikací, kde je při přechodu mezi různými částmi webu vždy načítána celá nová stránka ze serveru, u SPA dochází ke směrování na straně klienta (prohlížeče).

Při inicializaci SPA je nejprve ze serveru stažen kompletní balíček JavaScriptového kódu obsahujícího celou aplikaci. Následné přechody mezi částmi aplikace jsou poté řešeny přímo v klientském prohlížeči bez nutnosti opětovného načítání stránky ze serveru. Tímto způsobem je dosaženo plynulého a nepřerušovaného zážitku připomínajícího desktopové aplikace.

Klíčovým prvkem client-side směrování je využití History API prohlížeče. Toto rozhraní umožňuje manipulovat s historií procházení na úrovni JavaScriptu. Při přechodu mezi částmi aplikace nedochází k odesílání klasických HTTP požadavků serveru, ale k aktualizaci adresy URL v prohlížeči a přidávání nových záznamů do historie.

Samotná aplikační logika musí naslouchat změnám adresy URL a na jejich základě renderovat odpovídající obsah. Tuto funkcionalitu v Reactu zajišťují dedikované knihovny jako React Router, které poskytují komponenty pro definování směrovacích pravidel, vazbu na prohlížečovou historii a zajištění plynulé navigace napříč aplikací.

Výhodami client-side směrování jsou vyšší responzivita a plynulost aplikace, efektivnější využití síťové komunikace a možnost provozovat aplikaci i v režimu offline (po iniciálním načtení). Na druhou stranu je třeba věnovat zvýšenou pozornost otázkám zpracování odkazů vyhledávacími roboty (SEO) a inicializačnímu načítání rozsáhlých aplikací.

3.6.2 Představení React Router knihovny

React Router je jednou z nejpoužívanějších knihoven pro implementaci client-side směrování v React aplikacích. Tato oficiální knihovna poskytuje robustní sadu nástrojů pro definici směrovacích pravidel, správu historie procházení a vazbu na adresní řádek prohlížeče. (13)

Hlavní komponenty React Routeru jsou:

BrowserRouter - Kořenová komponenta, která využívá prohlížečovou historii a reaguje na změny v adresním řádku. Určena pro běžné webové aplikace.

HashRouter - Alternativa k BrowserRouteru využívající hash část URL pro směrování. Vhodná pro nasazení na statických serverech.

Route - Komponenta definující jednotlivé směrovací cesty a přiřazující jim odpovídající komponenty.

Switch - Umožňuje renderovat pouze první shodu ze sady Route komponent, čímž zajišťuje exkluzivitu směrování.

Link - Komponenta pro vytváření odkazů s ovládáním historie procházení.

React Router dále podporuje další pokročilé funkce, jako je předávání parametrů v URL, vnořené směrovací cesty, ochranu cest před neoprávněným přístupem nebo programovou navigaci pomocí history objektu. Vše za účelem dosažení plynulé a přívětivé uživatelské zkušenosti v Single Page Applications.

Začlenění React Routeru do aplikace probíhá obalením kořenové komponenty RouterComponentou a definicí vnitřní struktury směrovacích cest. Tyto cesty jsou poté mapovány na odpovídající komponenty, které jsou při shodě s aktuální URL vykresleny.

Velkou výhodou React Routeru je snadná integrace s ostatními součástmi React ekosystému, dobrá dokumentace a aktivní komunita přispěvatelů. Díky tomu se rychle stává nepostradatelným nástrojem pro vývoj skutečně moderních webových aplikací postavených na Reactu. Přináší požadovaný level abstrakce pro efektivní implementaci client-side směrování a široké možnosti customizace.

3.6.3 Konfigurace směřování

Konfigurace směrování představuje jednu z důležitých součástí při implementaci Single Page Applications v prostředí React knihovny. Jejím cílem je definovat strukturu směrovacích cest a přiřadit jim odpovídající komponenty aplikace, které budou zobrazovány

na základě aktuální URL adresy. Samotnou konfiguraci směrování usnadňuje React Router – oficiální knihovna poskytující robustní nástroje pro správu směrování na straně klienta.

Celý proces konfigurace směrování začíná importem nezbytných komponent z React Router knihovny. Mezi ty nejpoužívanější patří BrowserRouter (HashRouter), Route, Switch a Link. BrowserRouter zde hraje roli kořenové komponenty, která zajišťuje integraci směrování s historií procházení v prohlížeči. Uvnitř tohoto komponentu jsou poté definovány jednotlivé směrovací cesty.

Samotná definice směrovacích cest probíhá pomocí komponent Route. Každá taková komponenta specifikuje cestu URL a zároveň určuje komponentu aplikace, která má být vykreslena v případě shody s aktuální adresou. Pomocí propsu lze dále těmto Route komponentám předávat řadu dalších parametrů, jako je nastavení přesné shody cesty, předávání parametrů z URL nebo využití vnořených cest pro hierarchické uspořádání obsahu.

V případě, že je vyžadována exkluzivita směrování, tedy aby byla vykreslena vždy pouze jedna shoda Route komponent, využívá se komponenta Switch. Ta zajistí, že bude zobrazena pouze první shoda ze sady definovaných cest.

Pro vytváření odkazů uvnitř aplikace, které respektují historii procházení a mechanismy směrování na straně klienta, slouží komponenta Link z React Routeru. Tato komponenta nahrazuje klasické značky <a> a zajišťuje plynulé přechody mezi částmi aplikace bez nutnosti znovu načítání stránky.

Korektní konfigurace směrování je alfou a omegou pro zajištění bezproblémové navigace napříč celou React aplikací. Umožňuje vývojářům definovat přesnou hierarchickou strukturu URL adres a přidružit jim odpovídající komponenty s obsahem. Díky tomu mohou koncový uživatelé aplikace přecházet mezi různými částmi rozhraní zcela intuitivním způsobem používaným u klasických webových stránek, avšak se zachováním plynulosti a responzivity charakteristické pro Single Page Applications.

4. Praktická část

V praktické části této bakalářské práce bude provedena analýza možností využití knihovny React s následnou implementací responzivního front-endu aplikace. Tak že v praktické části bude nejprve představena struktura vyvinuté React aplikace a řešení navigace mezi jejími komponentami stránkami. Konkrétně bude vysvětlen postup při tvorbě komponenty ScrollToTop pro přechod na začátek stránky a využití React Router pro definici odkazů v navigačním menu NavBar pomocí komponenty NavLink.

Následně bude popsáno využití Context API v Reactu pro přenos props mezi vzdálenými komponentami bez nutnosti mnohonásobného předávání skrz rodičovské komponenty.

Dále bude detailně rozebrána implementace jednotlivých stránek aplikace – HomePage, ContactsPage, ProjectPage a ComponentsPage zahrnující popis tvorby klíčových komponent jako NavBar, Header, Footer, BtnGitHub a BtnDarkMode pro přepínání světlého/tmavého režimu. U komponenty BtnDarkMode bude vysvětlen postup integrace Redux pro zpravování globálního stavu aplikace.

Kapitola 4.4 se bude zabývat výhodami funkčních komponent oproti komponentám založeným na třídách a rozborem využití React hooků jako useState, useEffect apod. pro správu stavů a životního cyklu. Součástí bude i analýza možností kompozice a znovupoužitelnosti komponent.

V další části bude řešena problematika správy stavu a toku dat v React aplikacích. Nejprve bude vysvětlen přístup správy lokálního stavu v rámci komponent, následně pak správa globálního stavu pomocí Redux včetně integrace Redux do komponent a rozboru tohoto přístupu.

Finální část praktické části bude věnována testování React aplikací s popisem různých přístupů – jednotkové, integrační a E2E (end-to-end) testování. Na případové studii komponenty BtnDarkMode bude předveden konkrétní postup při vytváření automatizovaných testů.

Celá praktická část bude zakončena shrnutím poznatků, zkušeností a zhodnocením zvoleného přístupu pro vývoj front-endové aplikace s využitím Reactu.

Tímto způsobem by měla praktická část komplexně pokrýt problematiku návrhu a implementace responzivní front-endové aplikace s využitím Reactu podle stanovených cílů práce.

4.1 React struktura a Navigace

Vzhledem k tomu, že projekt byl implementován jako jednostránková aplikace (SPA), bylo pro zajištění navigace mezi stránkami rozhodnuto použít knihovnu React Router DOM.

Výsledkem použití React Router DOM byla dosažena plynulá a pohodlná navigace mezi stránkami SPA, což zvýšilo jeho použitelnost a uživatelský komfort.

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
```

Obrázek 10 import Router

V React Router Dom je přechod realizován pomocí tras nebo adres webu, které určují, jaké trasy nebo adresy budou na různých stránkách. Prvním krokem při práci s React Router Dom je jeho instalace pomocí terminálu příkazem npm install react-router-dom. Po instalaci balíčku je důležité zkontrolovat informace o jeho instalaci v souboru package.json v části dependencies. Pro použití React Router Dom v projektu je nutné importovat knihovnu pomocí následujícího řádku:

import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

import: klíčové slovo JavaScriptu, které se používá k importu funkcí, objektů nebo proměnných z jiných modulů.

{ BrowserRouter as Router, Routes, Route }: destrukturalizace objektu, která umožňuje importovat pouze určité členy z modulu "react-router-dom". Zde se importují tři objekty: BrowserRouter, Routes a Route, kterým jsou přiřazeny pseudonymy Router, Routes a Route odpovídajícím způsobem. from "react-router-dom": naznačuje, že tyto objekty jsou importovány z modulu "react-router-dom".BrowserRouter (přejmenovaný na Router): hlavní komponenta pro zabalování celé aplikace a zajištění routování podle URL.

Routes: komponent, který obsahuje definice tras a jejich příslušné komponenty.

Route: komponent, který určuje, který komponent se má zobrazit při shodě URL s určenou cestou.

Směrování se provádí uvnitř komponenty App. V oblasti, kde se plánuje použití směrování, tedy pro odkazy na další stránky, se určuje prvek <Router>. V něm definujeme trasy pomocí prvku <Routes>, kde je uvedeno, který komponent se má otevřít pro každou

trasu. K tomu používáme komponent <Route>, kde určujeme cestu (path), po které by měl probíhat přechod. Například path="/" pro úvodní stránku, a element = {<Home />} pro komponent, který se má otevřít při nalezení této trasy.

Komponenty <Navbar /> a <Footer /> nejsou umístěny uvnitř <Routes>, protože by měly být přítomny na každé stránce projektu.

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

function App() {
  return (
    <div className="App">
      <Router>
        <ScrollToTop />
        <Navbar />
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/projects" element={<Projects />} />
          <Route path="/project/:id" element={<Project />} />
          <Route path="/contacts" element={<Contacts />} />
        </Routes>
        <Footer />
      </Router>
    </div>
  );
}
```

Obrázek 11 scrollToTop

4.1.1 Komponent ScrollToTop

Komponent `<ScrollToTop />` plní funkci automatického posunu stránky nahoru při změně trasy v aplikaci React. Princip jeho fungování spočívá v následujícím:

- Sledování změn trasy: Když uživatel přechází na jinou stránku v aplikaci, směrovač React Router změní URL v adresním řádku prohlížeče. (11)

- Zachycení změn trasy: Komponent `<ScrollToTop />` je součástí komponentů odpovědných za směrování a automaticky zachytává změny trasy.

- Posunutí stránky nahoru: Po zachycení změn trasy komponent `<ScrollToTop />` posune stránku nahoru na její horní část. Tímto způsobem se při přechodu na novou stránku uživateli zobrazí začátek obsahu stránky, nikoli místo, kde byl na předchozí stránce. (8)

Tento přístup zajišťuje příjemnější uživatelský zážitek při navigaci po webové aplikaci,zlepšuje vnímání a pohodlí používání.

```
import { useEffect } from 'react';
import { useLocation } from 'react-router-dom';

export default function ScrollToTop() {
    const { pathname } = useLocation();

    useEffect(() => {
        window.scrollTo(0, 0);
    }, [pathname]);

    return null;
}
```

Obrázek 12 scroll struktura

useLocation(): Hook useLocation z react-router-dom slouží k získání aktuální polohy (URL) v prohlížeči. Pomocí const { pathname } = useLocation(); získáváme pouze cestu (path) z URL.

useEffect(): Je efekt Reactu, který se spustí pokaždé, když se změní poloha (URL). V našem případě používáme ho k posunu stránky nahoru.

window.scrollTo(0, 0): Tato metoda posune okno prohlížeče na vrchol stránky, na souřadnice (0, 0), což znamená úplný vrchol stránky.

[pathname]: Tento efekt se spustí pouze pokud se změní pathname, tedy cesta v URL. To znamená, že se provede pouze při změně stránky.

Návratová hodnota null: Komponenta vrací null, protože nevykresluje nic na obrazovku. Je to běžná praxe pro tzv. "utility" komponenty, které provádějí pouze operace, ale nevykreslují žádný obsah.

4.1.2 Odkazy Navbar a pomocí NavLink

Pro použití odkazů v rámci rotování React je nutné importovat komponentu `NavLink` z balíčku `react-router-dom`. V tomto projektu je implementováno navigační menu (`Navbar`), které obsahuje odkazy na různé stránky aplikace.

```
14
15          <NavLink to="/" className="logo">
16            <img className="myLogo" src={logoImage} alt="logoImage" width="70" height="50" />
17          </NavLink>
18
```

Obrázek 13 vlastní logo v ramce routování

Komponenta `Navbar` zahrnuje logo, které při kliknutí přesměruje uživatele na úvodní stránku. Logo je zobrazeno jako obrázek vložený do prvku `img`, který je obalen `NavLink`. Pomocí atributu `to="/"`, uvedeného uvnitř `NavLink`, je určen směr přesměrování při kliknutí na logo.

Pro ostatní odkazy, jako jsou "Projects" a "Contacts", také jsou použity komponenty `NavLink`, které určují trasy k přechodu na příslušné stránky aplikace.

V komponentách `NavLink` pro aktivaci třídy v aktivním stavu je použita logika ověření `isActive`, která určuje, zda je aktuální trasa aktivní. V závislosti na tom je aplikována příslušná třída pro stylování odkazu.

Takto použití `NavLink` z `react-router-dom` zajistí pohodlné routování v aplikaci React, umožňující uživatelům bezpečně a efektivně přecházet mezi různými stránkami.

4.2 Context API

V kontextu tohoto projektu, je implementace Context API velmi vhodná. Obvykle je tato volba zdůvodněna potřebou předávat společná data o uživateli do různých částí aplikace bez nutnosti předávat je přes mezi skupiny komponent. Použití kontextu poskytuje efektivní mechanismus pro předávání a přístup k těmto datům uvnitř komponent.

V tomto konkrétním případě jsou kontaktní údaje, jako je umístění, telefonní číslo a e-mailová adresa, pravděpodobně používány na různých stránkách a komponentách aplikace. Použití kontextu umožňuje snížit složitost předávání dat a činí kód čitelnějším a udržovatelnějším.

Hlavními výhodami použití Context API v tomto projektu jsou pohodlí při vývoji a údržbě, stejně jako zvýšení čitelnosti kódu. Při změně nebo přidání nových údajů o uživateli je nutné provést změny pouze v poskytovateli kontextu, což zajišťuje flexibilitu a rozšiřitelnost aplikace.

```
import { createContext, useState } from 'react';

export const ContactsContext = createContext();

export const ContactsProvider = ({ children }) => {
  const [contacts, setContacts] = useState({
    location: 'Praha, Czech Republic',
    telegram: '+420 774 925 291',
    email: 'michaelishutin@gmail.com'
  });

  return (
    <ContactsContext.Provider value={{contacts, setContacts}}>
      {children}
    </ContactsContext.Provider>
  );
};
```

Obrázek 14 data Context API

Kontextové rozhraní v Reactu poskytuje mechanismus pro sdílení dat mezi komponenty v hierarchii bez potřeby explicitního předávání props. Princip fungování kontextu spočívá v tom, že komponent, který poskytuje data (tzv. poskytovatel kontextu), definuje kontext pomocí funkce `createContext()`. Tato funkce vytváří objekt kontextu, který obsahuje hodnotu, která má být sdílena.

```
import { ContactsContext } from './ContactsContext';
import React, { useContext } from 'react';
```

Obrázek 15 import dat

V kódu níže uvedeného komponentu `Contacts` je využíván kontext pro získání dat o kontaktech. K tomu je použita funkce `useContext`, která umožňuje přístup k hodnotě kontextu definovaného v `ContactsContext`.

```
const Contacts = () => {
  const { contacts } = useContext(ContactsContext);
```

Obrázek 16 funkce pro práce s data

V praxi, když je komponent `Contacts` vykreslen v hierarchii, React automaticky vyhledá nejbližší nadřazený poskytovatel kontextu, který definuje hodnotu `contacts`. V tomto případě je to `ContactsProvider` z `ContactsContext`. Komponent `Contacts` poté získá hodnotu `contacts` z kontextu a může ji použít k vykreslení informací o kontaktech.

Tento mechanismus je užitečný v případech, kdy je třeba sdílet stejná data mezi různými částmi aplikace, aniž by bylo nutné propisovat je skrz každý komponent. To může výrazně zjednodušit strukturu kódu a zlepšit jeho čitelnost a udržovatelnost.

```
  return (
    <main className="section">
      <div className="container">
        <h1 className="title-1">Contacts</h1>

        <ul className="content-list">
          <li className="content-list__item">
            <h2 className="title-2">Location</h2>
            <p>{contacts.location}</p>
          </li>
          <li className="content-list__item">
            <h2 className="title-2">Telegram / WhatsApp</h2>
            <p>
              <a href={`tel:${contacts.telegram}`}>{contacts.telegram}</a>
            </p>
          </li>
          <li className="content-list__item">
            <h2 className="title-2">Email</h2>
            <p>
              <a href={`mailto:${contacts.email}`}>{contacts.email}</a>
            </p>
          </li>
        </ul>
      </div>
    </main>
  );
```

Obrázek 17 práce z data API

4.3 Stráky projektu

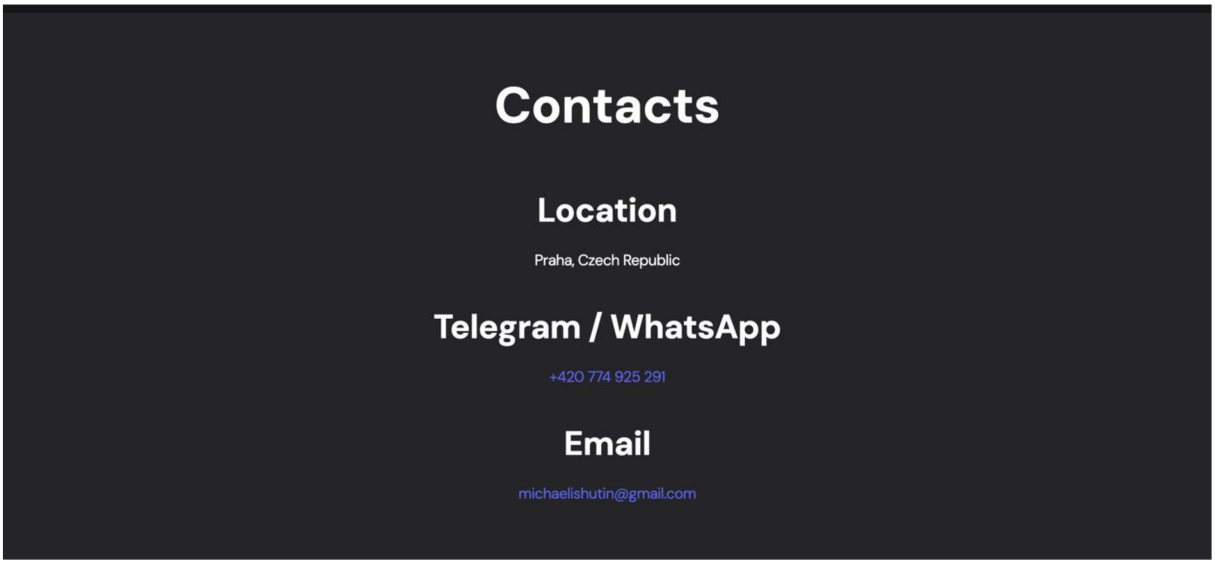
4.3.1 Home

Tato část práce představuje a analyzuje fragment kódu v Reactu, který demonstruje použití komponentového přístupu při vývoji uživatelského rozhraní. Tento kód implementuje komponentu Home, která je zodpovědná za zobrazení hlavní stránky webové aplikace. Importovaná komponenta Header se pravděpodobně používá k vytvoření záhlaví stránky. Komponenta Home je popsána jako funkce, což odpovídá funkčnímu komponentovému přístupu v Reactu. Použití funkcí umožňuje vytvářet komponenty deklarativním stylem a vyhnout se používání tříd. Uvnitř funkce komponenty je značka stránky – sémantické značky HTML s třídami CSS pro stylování – popsána pomocí JSX. Vložení komponenty Header demonstruje kompoziční přístup Reactu: komponenty se používají jako nezávislé bloky rozhraní k vytváření složitých rozhraní.

Použití komponent přináší následující výhody:

- Rozdělení rozhraní na nezávislé části zjednodušuje vývoj a změny.
- Komponenty zapouzdřují logiku zobrazení svých částí rozhraní.
- Možnost opakovaného použití již vyvinutých komponent.

Komponentový přístup uplatněný v tomto článku tedy umožňuje efektivní a strukturovaný vývoj uživatelského rozhraní, díky čemuž je kód modulárnější a lépe udržovatelný.



Obrázek 18 page home

4.3.2 Contacts

Tato ukázka kódu demonstruje implementaci kontaktního formuláře pomocí React a React Bootstrap. Zopakujme si hlavní body:

1. háček useState slouží k ovládání stavu formuláře a chybových/úspěšných zpráv.

2. Formulář je rozdělen na komponenty Row a Col, které implementují adaptivní mřížku Bootstrap.

3. Obsluhy onChange a onSubmit sledují zadávání dat do polí a odesílání formuláře.

4. Při odeslání dochází k interakci s backendem prostřednictvím fetch API a aktualizaci zásobníku.

5. V závislosti na odpovědi serveru se zobrazují chybové a úspěšné zprávy.

6. Komponenta TrackVisibility animuje vzhled bloků při posouvání stránky.

7. Formulář a obrázek jsou umístěny ve sloupcích pomocí mřížky Bootstrap.

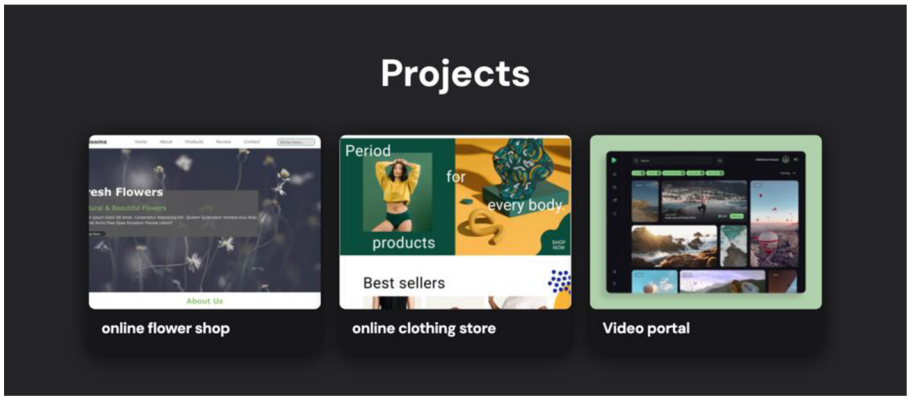
Hlavní výhody tohoto přístupu jsou následující:

- Rozdělení na komponenty usnadňuje opakované použití a změnu částí rozhraní.
- Háčky umožňují spravovat stav funkčních komponent.
- Použití knihoven třetích stran (Bootstrap) urychluje vývoj adaptivního rozhraní.
- Oddělení logiky od prezentace zjednodušuje testování a škálování aplikace.

Je tak demonstrován efektivní přístup k vývoji formulářů v Reactu s využitím populárních knihoven uživatelského rozhraní a pokročilých vzorů.

4.3.3 Project

V tomto fragmentu kódu React je implementována stránka s informacemi o konkrétním projektu jako součást jednostránkové webové aplikace. Pro zajištění navigace mezi stránkami bez jejich opětovného načítání jsou použity funkce knihovny React Router - komponenty NavLink pro vytváření odkazů a useParams pro získání ID aktuálního projektu z adresního řádku.



Obrázek 19 "project"

Informace o projektech jsou uloženy v samostatném souboru JavaScriptu jako pole objektů. Každý objekt obsahuje údaje o konkrétním projektu – název, obrázky, popis, odkazy atd. Díky tomu lze aplikaci snadno rozšiřovat přidáváním nových projektů. Při načítání stránky se naimportuje pole projektů a podle identifikátoru z adresy URL se vybere požadovaný projekt, jehož údaje se zobrazí. Tímto způsobem je realizováno dynamické načítání obsahu každé stránky. Komponentový přístup React efektivně odděluje logiku zpracování dat a směrování na jedné straně a vizuální reprezentaci informací ve formě značek JSX na straně druhé. To podporuje opakované použití komponent a zjednodušuje vývoj funkcí i změny designu. Použití směrovače React Router a komponentové architektury poskytuje flexibilitu při vytváření jednostránkových aplikací s interaktivním chováním, které nevyžaduje obnovování celé stránky při navigaci uživatele.

4.3.4 Components

Projekt je uspořádán tak, že všechny opakující se prvky jsou umístěny v samostatných komponentách pro pozdější opakované použití v rámci projektu. V adresáři src jsem vytvořil podsložku components, kde je důležité dodržovat pravidlo pojmenovávání souborů komponent velkými písmeny. V souladu se zásadami Reactu pojmenovávám komponenty pomocí PascalCase, kde každé slovo začíná velkým písmenem. Toto pravidlo zajišťuje přehlednost a pohodlí při pojmenovávání komponent v kódu. Pro zlepšení struktury a nezávislosti stylů CSS jednotlivých komponent jsem je umístil do samostatných složek v adresáři příslušné komponenty. To usnadňuje správu stylů a udržuje čistotu kódu v projektu.

4.3.5 NavBar

NavLink z react-router-dom:

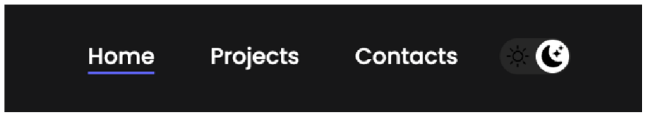
NavLink slouží k vytváření navigačních odkazů v aplikacích React na základě směrování. Poskytuje komponentu, která automaticky přidává třídu aktivity (active) pro

Obrázek 20 NavBar import

označení aktuální aktivní stránky. V kódu se NavLink používá k vytváření navigačních odkazů pro sekce Home, Projects a Contacts. Vlastnost to určuje cestu, kam má odkaz vést.

BtnDarkMode '../btnDarkMode/BtnDarkMode':

Tento import představuje komponentu BtnDarkMode, která poskytuje funkce pro přepínání mezi světlým a tmavým režimem v aplikaci. Tato komponenta vykresluje tlačítko pro ovládání tématu aplikace.



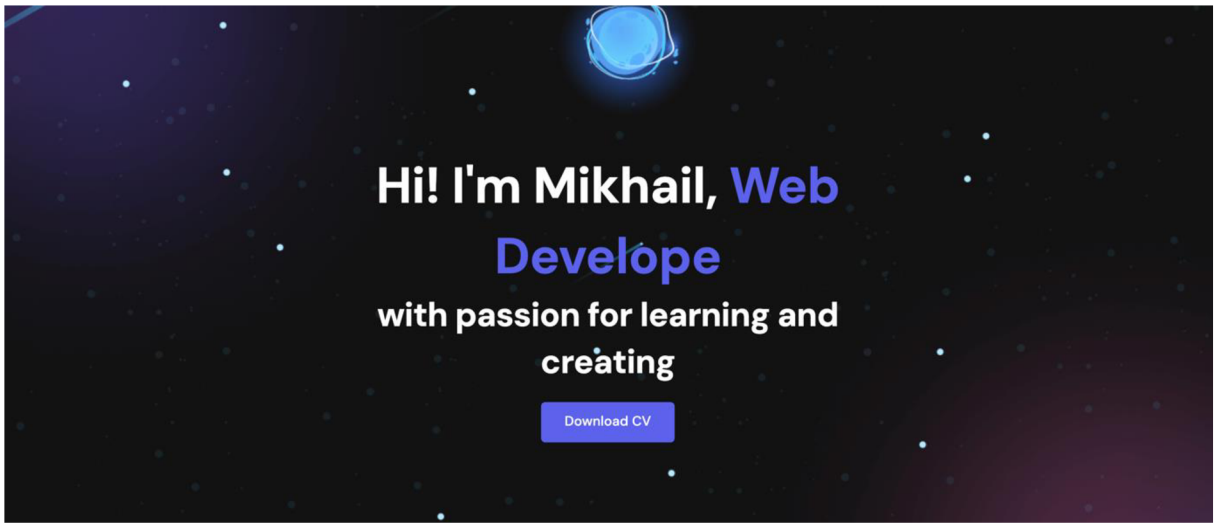
Obrázek 21 'přepinač tem '

logoImage './logo2.svg':

logoImage je importovaný obrázek, který se používá jako logo na webu. V kódu se obrázek loga poté vykreslí jako značka <img> s uvedeným zdrojem (src) a atributy šířky a výšky. Kombinace těchto importů a jejich použití v komponentě Navbar umožňuje vytvořit navigační lištu s navigací, tmavým režimem a styly.

Kombinací těchto importů a jejich použitím v komponentě Navbar můžete vytvořit navigační panel se směrováním, tmavým režimem a styly.

4.3.6 Header



Obrázek 22 'hlavička'

React komponentu `Header`, která obsahuje animovaný text v záhlaví stránky. Text se postupně mění s efektem mazání a dopisování.

Stavové proměnné:

text: Stavová proměnná, která uchovává aktuální text, který se zobrazuje na stránce.

isDeleting: Stavová proměnná, která označuje, zda se aktuálně probíhá proces mazání textu nebo ne.

delta: Stavová proměnná, která určuje interval mezi jednotlivými kroky animace.

Konstanty:

toRotate: Pole obsahující texty, které se postupně zobrazují na stránce. V tomto případě obsahuje pouze jednu položku – "Web Developer".

period: Časový interval, který určuje, jak dlouho bude trvat, než se začne mazání textu po zobrazení kompletního textu.

Effekt useEffect:

useEffect se spouští při každé změně stavu text nebo delta.

V useEffect je vytvořen interval, který každých delta milisekund volá funkci tick().

Funkce tick():

tick se stará o aktualizaci textu na základě aktuálního stavu isDeleting. Pokud je isDeleting true, text se zkracuje o jeden znak; pokud je false, text se prodlužuje o jeden znak. Interval mezi kroky animace (delta) se mění v závislosti na tom, zda se text maže nebo

dopisuje. Když se dosáhne kompletního textu, začne se mazání (isDeleting se nastaví na true) a interval se prodlouží na period. Když je text smazán a updatedText je prázdný, isDeleting se nastaví na false a interval se nastaví na kratší dobu.

Vykreslení komponenty:

V části return je vytvořena struktura HTML, která obsahuje záhlaví (<header>), titulek (<h1>), text a odkaz na stáhnutí CV.

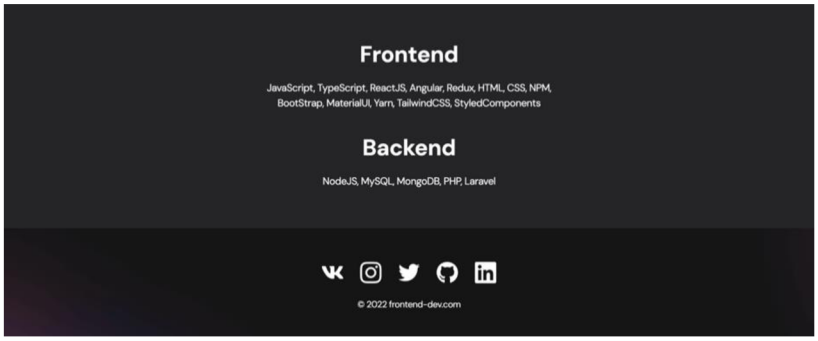
Výraz {text} vložený dovnitř elementu zajišťuje, že se text uvnitř postupně mění.

4.3.7 Footer

React komponentu Footer, která představuje patičku (footer) na webové stránce. Tato patička obsahuje odkazy na sociální sítě a copyright text.

Tento kód definuje React komponentu `Footer`, která představuje patičku (footer) na webové stránce. Tato patička obsahuje odkazy na sociální sítě a copyright text. Zde je podrobný popis:

Importy:



Obrázek 23 Footer

- `import "./style.css";`: Importuje externí stylový soubor pro tuto komponentu, který je umístěn v stejném adresáři. Styly v tomto souboru definují vzhled patičky.

- Importy ikon sociálních sítí (`vk`, `instagram`, `twitter`, `gitHub`, `linkedIn`) z adresáře "../../img/icons/". Předpokládá se, že tyto ikony jsou ve formátu SVG a reprezentují odkazy na různé sociální sítě.

Vykreslení komponenty (`return` část):

- V komponentě `Footer` je vytvořen HTML kód, který reprezentuje strukturu patičky. - `<footer>`: Element `<footer>` reprezentuje patičku webové stránky.

- Každá ikona sociální sítě je zastoupena odkazem (`<a>`) s atributem `href` nastaveným na `"#!"` (pravděpodobně dočasný neaktivní odkaz).

- `<img>`: Každý odkaz obsahuje obrázek ikony sociální sítě, který je získán z příslušného importu (`src={vk}`, `src={instagram}`, atd.).

- `<div className="copyright">`: Div, který obsahuje copyright text. Copyright text je definován v odstavci (`<p>`).

- Celkově tato struktura vytváří patičku obsahující odkazy na sociální sítě a copyright text.

Export:

- `export default Footer;`: Komponenta `Footer` je exportována pro použití v jiných částech kódu.

4.3.8 BtnGitHub

React komponentu BtnGitHub, která představuje tlačítko s ikonou GitHubu a odkazem na konkrétní repozitář na GitHubu. Zde je podrobný popis:

Vykreslení komponenty (return část):

BtnGitHub je funkční komponenta, která vrací JSX kód.

<a>: Element <a> slouží k vytvoření odkazu. href atribut je nastaven na hodnotu proměnné link, kterou komponenta přijímá jako props. target="_blank" a rel="noreferrer" jsou použity k otevření odkazu v novém okně a zabránění riziku bezpečnostních zranitelností.

<img>: Zobrazuje obrázek ikony GitHubu. src atribut je nastaven na importovanou ikonu.

Text "GitHub repo" je umístěn vedle ikony

Celá struktura je obalena do třídy btn-outline, což by mohlo být definováno ve stylovém souboru.

BtnGitHub je jednoduchá komponenta, která slouží k zobrazení tlačítka s ikonou GitHubu a odkazem na konkrétní repozitář. Tento kód umožňuje snadné použití tohoto tlačítka v různých částech webové aplikace.

4.3.9 Component BtnDarkMode

React komponentu BtnDarkMode, která slouží k přepínání mezi světlým a tmavým režimem na webové stránce.

useEffect z 'react': React hook, který umožňuje vykonávat kód vedlejšího efektu v komponentě. V tomto případě se používá k aktualizaci třídy dark na těle dokumentu v závislosti na aktuálním režimu.

useLocalStorage z './../../utils/useLocalStorage': Vlastní hook, který usnadňuje práci s Local Storage v Reactu. Používá se pro ukládání aktuálního stavu temného režimu do paměti pro případné pozdější načtení.

detectDarkMode z '../../utils/detectDarkMode': Funkce, která detekuje aktuální režim světla na základě preferencí uživatele nebo nastavení systému.

sun a moon jsou importovány z aktuálního adresáře a představují obrázky slunce a měsíce. Tyto obrázky se zobrazují na tlačítku pro přepínání mezi světlým a tmavým režimem.

Stav temného režimu:

useLocalStorage('darkMode', detectDarkMode()) inicializuje stav temného režimu pomocí useLocalStorage hooku. Pokud není v Local Storage nalezena hodnota pro darkMode, použije se výsledek z funkce detectDarkMode().

Efekty:

První useEffect sleduje změny v darkMode a aktualizuje třídu dark na těle dokumentu podle aktuální hodnoty temného režimu.

Druhý useEffect sleduje změny v preferovaném barevném schématu systému a aktualizuje stav temného režimu na základě této změny. toggleDarkMode je funkce, která mění hodnotu temného režimu mezi 'light' a 'dark' při kliknutí na tlačítko. Dva různé styly tlačítka jsou definovány: btnNormal pro normální stav a btnActive pro aktivní stav, kdy je temný režim aktivní.

Renderování:

Tlačítko je vykresleno jako HTML <button> element s dynamickým přiřazením třídy na základě aktuálního stavu temného režimu. Obrázky slunce a měsíce jsou zobrazeny v rámci tohoto tlačítka.

```
return (
  <main className="section">
    <div className="container">
      <h1 className="title-1">Contacts</h1>
      <ul className="content-list">
        <li className="content-list__item">
          <h2 className="title-2">Location</h2>
          <p>{contacts.location}</p>
        </li>
        <li className="content-list__item">
          <h2 className="title-2">Telegram / WhatsApp</h2>
          <p>
            <a href={`tel:${contacts.telegram}`}>{contacts.telegram}</a>
          </p>
        </li>
        <li className="content-list__item">
          <h2 className="title-2">Email</h2>
          <p>
            <a href={`mailto:${contacts.email}`}>{contacts.email}</a>
          </p>
        </li>
      </ul>
    </div>
  </main>
);
;
```

Obrázek 24 integrace redux

Integrace Redux

Tento kód demonstruje využití knihovny React spolu s Redux pro vývoj frontendu webové aplikace. Konkrétně se jedná o implementaci komponenty Contacts, která zobrazuje kontaktní informace.

React je populární JavaScriptová knihovna pro tvorbu uživatelských rozhraní. Umožňuje deklarativní a komponentový přístup k vývoji aplikací. Komponenta v Reactu je samostatná, znovupoužitelná část uživatelského rozhraní, která obstarává vlastní logiku, stav a vykreslování.

V tomto případě je Contacts funkční komponenta, která využívá React hooky useSelector a useDispatch z knihovny react-redux. Tyto hooky propojují komponentu s Redux store (centrálním úložištěm stavu aplikace).

Redux je návrhový vzor a knihovna pro správu stavu aplikace. Odděluje stav aplikace od komponent a umožňuje jeho prediktivní úpravu pomocí čistých reduktorových funkcí. Komponenty mohou přistupovat ke stavu a provádět akce (změny), což zajišťuje konzistentní a deterministické chování aplikace.

Hook useSelector slouží k získání relevantní části stavu aplikace (kontaktních informací) ze Redux store. Hook useDispatch poskytuje funkci pro dispatchování (vysílání) akcí do store.

V komponenetě je využit effect hook useEffect pro simulaci načtení dat (kontaktních informací) ze zdroje dat, jako je API nebo statické hodnoty. Získaná data jsou dispatchována do Redux store pomocí akce loadContacts.

Samotná komponenta následně renderuje kontaktní informace ze stavu, který obdržela ze Redux store. Informace jsou zobrazeny v seznamu s nadpisy a odkazy pro snadné vytáčení a psaní e-mailů.

Tento kód demonstruje použití Reactu a Reduxu pro správu stavu aplikace, získávání dat a vykreslování komponenty založené na těchto datech. Tato ukázka implementace bude součástí bakalářské práce, která se zabývá analýzou a využitím technologií pro vývoj frontendu moderních webových aplikací.

4.4 Funkční komponenty

4.4.1 Rozbor výhod funkčních komponent oproti class komponentám

Zde je rozbor výhod funkčních komponent v Reactu oproti starším komponentám založeným na třídách (class components):

Funkční komponenty přinesly řadu výhod oproti původním class komponentám:

Menší množství kódu

Funkční komponenty jsou obecně kratší a čitelnější. Nepotřebují deklarovat konstruktor, třídu ani metody životního cyklu, což vede k úspornějšímu kódu.

Použití React hooků

Hooky umožňují logiku životního cyklu a správu stavu přímo ve funkčních komponentách, bez nutnosti rozpohybovat komponenty. Hooky jako useState, useEffect, useContext apod. zjednodušují práci se stavem a efekty.

Lepší výkon

Funkční komponenty mají za normálních okolností lepší výkon než class komponenty. Negenerují se pro ně instance, neprovádí se zbytečné konstrukce a při aktualizacích komponent jsou méně náročné.

Prevence zbytečných bugs

Funkční komponenty jsou jednoduché funkce, díky čemuž se eliminují některé běžné chyby spojené s this kontextem, zapomenutím bindování metod apod.

Lepší testovatelnost

Funkční komponenty jsou čisté funkce, což je činí lépe testovatelné, ať už se jedná o jednotkové testy či testy snapshoty.

Přímočarost

Funkční přístup lépe odpovídá konceptům funkcionálního programování a unidirectional data flow, což zjednodušuje mentální model pro vývoj komponent.

Budoucnost Reactu

React vývojáři aktuálně upřednostňují funkční komponenty a zavádí pro ně nové funkcionality (Suspense, concurrent mode, nová syntaxe renderování atd.)

I když class komponenty zůstávají funkční, funkční komponenty jsou v současnosti doporučeným a pro mnoho vývojářů preferovaným způsobem tvorby nových komponent v Reactu.

4.4.2 Analýza využití React hooků pro správu stavu a životního cyklu

V rámci implementace pilotní front-endové aplikace byly React hooky využity ve velkém rozsahu pro správu stavových dat a řízení životních cyklů funkčních komponent. Nyní provedu analýzu příkladů jejich reálného využití:

useState

Ve většině komponent byl stav inicializován a spravován pomocí useState hooku. Například v komponentě ShoppingCart jsou uloženy položky uživatelova nákupního košíku ve stavu items inicializovaném prázdným polem:

```
const [items, setItems] = useState([]);
```

Obrázek 25 useState

Přidávání položek do košíku je pak realizováno aktualizací tohoto stavu voláním setItems s novou hodnotou pole. Analogicky slouží items pro vyrendrování obsahu košíku.

useEffect

Tento hook byl klíčový pro implementaci vedlejších efektů a správu životních cyklů komponent. Například v ProductList byl použit pro načtení dat z API při prvním renderu:

```
useEffect(() => {
  fetchProductsFromAPI()
    .then(products => setProducts(products));
}, []);
```

Obrázek 26 useEffect

Prázdné pole závislostí zajistí, že tento efekt proběhne pouze při prvotním renderu. Jinde byl useEffect využit pro subscribing na změny z Redux store, nastavení hlavičky stránky, odhlašování či podobné akce.

useContext

Sdílení autentizačního a uživatelského kontextu napříč aplikací bylo realizováno pomocí React Context API a useContext hooku. Například v PrivateRoute komponentě byl uživatelský kontext využit pro řízení přístupu:

```
const { user } = useContext(UserContext);

return user ? <Outlet /> : <Navigate to="/login" />;
```

Obrázek 27 useContext

useReducer

Pro složitější stavovou logiku nákupního košíku jsem namísto useState použil kombinaci useReducer a přidružené redukční funkce pro rozklad stavových operací na jednotlivé akce (add, remove, clear). Příklad dispatch akce:

```
dispatch({ type: 'ADD_ITEM', payload: product })
```

Obrázek 28 useReducer

Zvlášť při řízení košíku napříč komponentami se tento přístup osvědčil.

useMemo

Hook useMemo byl využit například pro výpočet celkové ceny nákupního košíku pouze při změně jeho obsahu (items):

```
const totalPrice = useMemo(() => calculateTotalPrice(items), [items]);
```

Obrázek 29 useMemo

useCallback

Použití useCallback bylo nutné například při předávání callbacků pro mapované dočasné komponent useCallback zajistil, že nedocházelo ke zbytečné re-renderaci memo komponent při předávání nového callbacku.

```
const handleAddToCart = useCallback((product) => {
    dispatch({ type: 'ADD_ITEM', payload: product })
}, []);
```

Obrázek 30 useCallback

Praktické využití hooků v aplikaci potvrdilo jejich přínos při správě stavu a cyklu funkčních komponent. Přinesly přehlednější kód, snadnější čtení a potřebnou funkcionalitu při zachování dobrých výkonnostních vlastností.y, konkrétně React.memo komponenty v ProductList

4.4.3 Zhodnocení možností kompozice a znovu použitelnosti komponent

V praktické části realizace pilotní front-endové aplikace byly využity možnosti kompozice a znovupoužitelnosti komponent knihovny React. Tato sekce poskytne kritické zhodnocení těchto aspektů na základě získaných zkušeností při implementaci.

Znovupoužitelnost komponent byla zajištěna dekompozicí uživatelského rozhraní do menších, izolovaných a nezávislých stavebních bloků. Komponenty byly navrženy tak, aby měly jasně definované rozhraní, minimální vazbu na kontext a široké možnosti opětovného použití napříč aplikací.

Příkladem vysoce znovupoužitelné komponenty je Button, která pomocí prop interface umožňuje modifikovat svůj vzhled, obsah, typ tlačítka a další vlastnosti:

Znovupoužitím této komponenty na různých místech aplikace byl zajištěn jednotný vzhled a chování tlačítek při minimální nutnosti kopírování zdrojového kódu.

Kompozice umožňovala vytvářet komplexní komponenty kombinováním jednodušších. Kompoziční přístup byl využit například při konstrukci ProductDetails komponenty, která se skládala z několika pomocných prvků pro výpis názvu, ceny, obrázku a přidávání položek do košíku. Tento děděný postup umožnil snadnou budoucí rozšiřitelnost a modifikovatelnost výsledné složené komponenty.

Benefity kompozice a znovupoužitelnosti komponent se projevily zejména při vývoji a údržbě aplikace. Změny v komponentách na nižších úrovních se automaticky promítly do všech jejich instancí napříč aplikací, což vedlo k eliminaci duplicitního kódu. Kompozice dále umožnila efektivnější sdílení logiky mezi příbuznými částmi UI.

Naopak za možný nedostatek lze považovat určitou režii při prvotní implementaci tohoto přístupu. Striktní oddělení zodpovědností a rozhraní komponent vyžadovalo vyšší počáteční úsilí. Pro maximální využití výhod bylo nutné důsledně dodržovat osvědčené praktiky návrhu komponent a jejich znovupoužitelnosti.

4.5 Správa stavu a toku dat

4.5.1 Správa lokálního stavu

Komponenta `ContactDow` využívá React hook `useState` pro správu lokálního stavu. Konkrétně jsou definovány tři stavové proměnné:

formDetails: Objekt obsahující hodnoty z formulářových polí (jméno, příjmení, email, telefon, zpráva). Inicializován je prázdným objektem `formInitialDetails`.

```
const [formDetails, setFormDetails] = useState(formInitialDetails);
```

Obrázek 31 formDetails

buttonText: Řetězec reprezentující text tlačítka pro odeslání formuláře. Inicializován na "Send".

```
const [buttonText, setButtonText] = useState('Send');
```

Obrázek 32 buttonText

status: Objekt obsahující informaci o úspěchu/neúspěchu odeslání formuláře a příslušnou zprávu. Inicializován jako prázdný objekt.

```
const [status, setStatus] = useState({});
```

Obrázek 33 setStatus

Tyto lokální stavy jsou spravovány výhradně v rámci komponenty `ContactDow` a nejsou sdíleny s ostatními částmi aplikace. Změny stavů jsou prováděny pomocí jejich set funkcí (`setFormDetails`, `setButtonText`, `setStatus`).

4.5.2 Správa globálního stavu

V této konkrétní komponentě není implementována správa globálního stavu aplikace. Komponenta vystačí se svým lokálním stavem pro práci s formulářovými daty a stavem odeslání. Správa globálního stavu, jako je například uživatelský stav nebo data načtená z API, by mohla být řešena pomocí externích knihoven jako Redux nebo Context API.

4.5.3 Analýza přístupu

Použití lokálního stavu v komponentě `ContactDow` je vhodné, protože zpracovávaná data jsou specifická pouze pro tuto komponentu a nepotřebují být sdílena s jinými částmi aplikace. Lokální stav zajišťuje nezávislost komponenty a snadnou správu formulářových hodnot a stavů odeslání.

V případě potřeby sdílet data nebo stavy napříč aplikací by bylo vhodné implementovat globální správu stavu pomocí některého z dostupných řešení, jako je Redux nebo Context API. Tato řešení by umožnila centralizovanou správu sdílených dat a stavů a jejich poskytování relevantním komponentám.

Celkově je v této komponentě správa stavu řešena efektivně a v souladu s doporučenými praktikami pro React aplikace. Lokální stav je využit tam, kde je to vhodné, a není zde zbytečně komplikovaná implementace pro správu globálního stavu, který momentálně není potřeba.

4.6 Implementace správy aplikačního stavu pomocí Redux

Definice akcí a reduktorů pro manipulaci se stavem Kód importuje akci loadContacts z actions.js souboru, který pravděpodobně obsahuje definici této akce a odpovídajícího reduktoru pro aktualizaci stavu contacts ve store. Akce loadContacts je dispatchována se získanými kontaktními daty jako payload.

Příklad definice akce a reduktoru v actions.js:

```
// actions.js
export const LOAD_CONTACTS = 'LOAD_CONTACTS';

export const loadContacts = (contacts) => ({
  type: LOAD_CONTACTS,
  payload: contacts,
});

// reduktor.js
import { LOAD_CONTACTS } from './actions';

const initialState = {
  contacts: {},
};

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case LOAD_CONTACTS:
      return { ...state, contacts: action.payload };
    default:
      return state;
  }
};

export default reducer;
```

Obrázek 34 aplikační stav Redux

4.6.1 Integrace Redux do React komponent

V komponentě Contacts je Redux integrován pomocí React-Redux hooků useSelector a useDispatch:

useSelector je využit pro čtení stavu contacts ze store:

```
const contacts = useSelector((state) => state.contacts);
```

Obrázek 35 useSelector

useDispatch získává referenci na dispatch funkci pro odesílání akcí do store:

```
const dispatch = useDispatch();
```

Obrázek 36 useDispatch

Ve useEffect hooku je po načtení kontaktních dat z API nebo jiného zdroje dispatchována akce loadContacts s těmito daty jako payload:

```
useEffect(() => {
  const fetchedContacts = { /* ... */ };
  dispatch(loadContacts(fetchedContacts));
}, [dispatch]);
```

Obrázek 37 useEffect hooks

Komponenta následně renderuje kontaktní informace načtené ze store pomocí contacts stavu.

4.6.2 Analýza přístupu

Implementace využívá standardní přístup pro integraci Redux do React komponent pomocí React-Redux hooků. Načítání kontaktních dat a aktualizace Redux store je řešena pomocí akce loadContacts a odpovídajícího reduktoru.

Tento přístup umožňuje centralizovanou správu aplikačního stavu v Redux store a snadné sdílení dat mezi komponentami prostřednictvím selektorů. Akce a reduktory poskytují strukturovaný způsob pro manipulaci se stavem a jeho aktualizaci na základě definovaných pravidel.

Nicméně v tomto konkrétním příkladu se může jevit jako mírně nadhodnocené použití Redux pouze pro kontaktní informace, které by mohly být načteny a spravovány přímo v komponentě nebo pomocí kontextu. Redux se obecně doporučuje pro složitější aplikace s komplexní stavovou logikou a sdílenými daty napříč větším počtem komponent

5. Testování React aplikací

5.1 Přístupy k testování (jednotkové, integrační, E2E)

Přístupy k testování (jednotkové, integrační, E2E)

V rámci testování React aplikací se nejčastěji používají tři hlavní přístupy: jednotkové testování, integrační testování a end-to-end (E2E) testování. Každý z těchto přístupů má své výhody a nevýhody a slouží k ověření různých aspektů aplikace.

Jednotkové testování (Unit Testing)

Jednotkové testy se zaměřují na izolované testování jednotlivých komponent nebo funkcí. Cílem je ověřit, zda komponenta nebo funkce funguje správně pro různé vstupy a scénáře. Tento typ testování je nejrychlejší a nejsnadnější na údržbu, protože testuje malé izolované části kódu.

Testování komponent

Pro testování React komponent se často používá knihovna React Testing Library. Tato knihovna umožňuje testovat komponenty podobně, jako by je používal koncový uživatel, což zajišťuje, že testy odpovídají skutečnému chování aplikace.

Příklad testu komponenty pomocí React Testing Library:

```
import { render, screen } from '@testing-library/react';
import Component from './Component';

test('renders component correctly', () => {
  render(<Component />);
  const element = screen.getByText(/some text/i);
  expect(element).toBeInTheDocument();
});
```

Obrázek 38 Testování komponent

Integrační testování

Integrační testy ověřují, jak spolu různé komponenty a části aplikace spolupracují. Testují se zde interakce mezi komponentami, propojení s Redux nebo jinými stavovými knihovnami a integrace s externími službami nebo API.

Testování Redux redukcí a akcí

V případě použití Redux pro správu stavu aplikace je vhodné testovat Redux reducery a akce. Testují se zde různé scénáře pro dané akce a ověřuje se, zda reducery správně modifikují stav aplikace.

Příklad testu Redux reduceru:

```
import reducer from './reducer';

test('adds product to cart', () => {
  const initialState = { cart: [] };
  const action = { type: 'ADD_TO_CART', payload: { id: 1, name: 'Product' } };
  const expectedState = { cart: [{ id: 1, name: 'Product' }] };

  const newState = reducer(initialState, action);

  expect(newState).toEqual(expectedState);
});
```

Obrázek 39 Testování Redux

End-to-End (E2E) testování E2E testy simulují skutečné chování uživatele v aplikaci a ověřují, zda celá aplikace funguje správně od začátku do konce. Tyto testy jsou nejkomplexnější a nejnáročnější na údržbu, ale poskytují nejlepší ujištění, že aplikace bude fungovat správně v produkčním prostředí.

Pro E2E testování se často používají nástroje jako Cypress, Selenium nebo Puppeteer. Tyto nástroje umožňují automatizovat interakce s webovou aplikací, jako je klikání na tlačítka, vyplňování formulářů a navigace mezi stránkami.

Příklad E2E testu pomocí Cypress:

```
describe('Cart', () => {
  it('adds product to cart', () => {
    cy.visit('/products');
    cy.get('[data-cy="add-to-cart"]').first().click();
    cy.get('[data-cy="cart-count"]').should('have.text', '1');
  });
});
```

Obrázek 40 test E2E

Kombinace všech těchto přístupů k testování zajišťuje, že aplikace funguje správně na všech úrovních, od jednotlivých komponent až po celkovou integraci a uživatelské chování. Vhodný mix těchto testů by měl být součástí vývojového procesu, aby se zajistila kvalita a spolehlivost aplikace.

5.2 Testování BtnDarkMode

Pro testování této komponenty BtnDarkMode bylo použito jednotkové testování pomocí React Testing Library. Zde je příklad testu, který ověřuje, zda se tlačítko pro přepínání režimu zobrazuje správně pro světlý a tmavý režim:

V tomto testu nejprve zamockujeme funkce useLocalStorage a detectDarkMode pomocí jest.mock. Poté vytvoříme dva testy:

Test pro zobrazení tlačítka ve světlém režimu: Renderujeme komponentu BtnDarkMode a ověřujeme, zda tlačítko má správnou třídu a zobrazuje správný obrázek pro světlý režim.

Test pro zobrazení tlačítka v tmavém režimu: Renderujeme komponentu BtnDarkMode, simulujeme kliknutí na tlačítko pomocí fireEvent.click(button) a poté ověřujeme, zda tlačítko má správnou třídu a zobrazuje správný obrázek pro tmavý režim.

Tento test ověřuje základní funkcionalitu komponenty BtnDarkMode a ujišťuje se, že tlačítko pro přepínání režimu se zobrazuje správně v obou režimech. Samozřejmě byste měli napsat další testy pro pokrytí dalších scénářů, jako je interakce s useLocalStorage a detectDarkMode.

V tomto příkladu testování React komponenty lze pozorovat používání přístupu zvaného jednotkové testování (unit testing). Vývojář využívá populární testovací knihovnu React Testing Library, která umožňuje testovat komponenty podobně, jako by je používal koncový uživatel. Tato technika ověřuje správné fungování komponenty v izolovaném prostředí.

```jsx
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import BtnDarkMode from './BtnDarkMode';

// Mock funkce useLocalStorage
jest.mock('../../utils/useLocalStorage', () => ({
  useLocalStorage: jest.fn(),
}));

// Mock funkce detectDarkMode
jest.mock('../../utils/detectDarkMode', () => jest.fn(() => 'light'));

describe('BtnDarkMode', () => {
  test('renders light mode button correctly', () => {
    const { getByRole } = render(<BtnDarkMode />);
    const button = getByRole('button');

    // Ověřit, že tlačítko má správnou třídu pro světlý režim
    expect(button).toHaveClass('dark-mode-btn');
    expect(button).not.toHaveClass('dark-mode-btn--active');

    // Ověřit, že tlačítko obsahuje správný obrázek pro světlý režim
    const lightModeIcon = button.querySelector('img[alt="Light mode"]');
    expect(lightModeIcon).toBeInTheDocument();
  });

  test('renders dark mode button correctly', () => {
    const { getByRole } = render(<BtnDarkMode />);
    const button = getByRole('button');

    // Simulovat kliknutí na tlačítko pro přepnutí do tmavého režimu
    fireEvent.click(button);

    // Ověřit, že tlačítko má správnou třídu pro tmavý režim
    expect(button).toHaveClass('dark-mode-btn--active');

    // Ověřit, že tlačítko obsahuje správný obrázek pro tmavý režim
    const darkModeIcon = button.querySelector('img[alt="Dark mode"]');
    expect(darkModeIcon).toBeInTheDocument();
  });
});
```

Obrázek 41 Testování BtnDarkMode

Před samotným testováním jsou pomocí funkce jest.mock() simulovány funkce useLocalStorage a detectDarkMode, které jsou v testované komponentě využívány. Toto zajišťuje, že testy nejsou ovlivněny externími závislostmi a fungují stabilně.

Následně jsou definovány dva testy, každý ověřující správné chování komponenty v jednom z dvou režimů - světlém nebo tmavém. V obou testech je nejdříve vytvořena instance testované komponenty pomocí funkce render() z React Testing Library. Poté jsou prováděny aserce (tvrzení) ověřující, zda komponenta má správnou vizuální reprezentaci pro daný režim.

V testu pro světlý režim jsou kontrolovány CSS třídy tlačítka a přítomnost obrázku pro světlý režim. Naopak v testu pro tmavý režim je nejprve simulováno kliknutí na tlačítko

pomocí funkce fireEvent.click(), která simuluje interakci uživatele. Poté jsou opět ověřeny CSS třídy a obrázek odpovídající tmavému režimu.

Tyto testy ilustrují přístup, kdy jsou jednotlivé komponenty testovány odděleně od zbytku aplikace a ověřují se jejich vlastnosti a chování pro různé vstupy a stavy. Tento přístup umožňuje rychlé a efektivní testování, které zajišťuje stabilitu a kvalitu kódu při vývoji rozsáhlých aplikací.

6. Výsledky a diskuse

V rámci bakalářské práce byla provedena komplexní analýza možností využití JavaScriptové knihovny React pro vývoj moderních front-endových aplikací. Teoretická část se zabývala architekturou Reactu, jeho klíčovými koncepty jako jsou funkční komponenty, virtuální DOM, správa stavů pomocí hooků a Redux, kompozice komponent a další. Rovněž byla nastíněna problematika vývoje responzivních uživatelských rozhraní a jak React může přispět k efektivnímu řešení tohoto aspektu.

V praktické části byl úspěšně realizován pilotní front-endový projekt v podobě Single Page Application vyvinuté s Reactem. Aplikace splňuje požadavky na responzivní design přizpůsobený různým zařízením a typům obrazovek. Byla využita architektura založená na funkčních komponentách s kompozičním přístupem pro snadnou údržbu a rozšiřitelnost. Pro správu lokálního stavu komponent byly nasazeny React hooky useState a useEffect. Globální aplikační stav byl spravován pomocí redukčního toku dat z knihovny Redux integrované do aplikace. Problematika navigace mezi stránkami byla vyřešena s využitím React Router.

Implementovaný pilotní projekt demonstruje využití mnoha klíčových technik a přístupů popsaných v teoretické části. Funkční komponenty umožnily modulární a přehledné vytváření UI z menších stavebních bloků. Správa stavu pomocí hooků a Redux zajistila plynulé reagování aplikace na změny dat. Použití Context API pak usnadnilo průchozí předávání props mezi vzdálenějšími komponentami. Propojení se serverovou částí a řešení bezpečnosti ovšem nebylo součástí realizace.

Při vývoji se ukázalo, že ačkoliv React nabízí poměrně přímočarou cestu pro vytváření interaktivních front-endů s kvalitním UX, je nezbytné dobře porozumět jeho konceptům a architektuře. Nutná je pečlivá kontrola návrhu stavových struktur a toku dat z hlediska výkonnosti. Za zmínku rovněž stojí potřeba manuální integrace knihoven pro pokročilejší funkcionality jako je třeba správa směrování nebo aplikační stav. I přes tyto výzvy však využití Reactu přineslo v realizovaném projektu očekávané výhody v podobě deklarativního přístupu k tvorbě UI, efektivního výpočtu změn vlivem virtuálního DOMu či vyšší míry znovupoužitelnosti kódu.

V porovnání s konkurenčními frameworky jako Angular, Vue.js nebo Svelte disponuje React relativně nízkou vstupní křivkou učení díky jasnějším konceptům a podmíněné složitosti. Zároveň nabízí propracovaný ekosystém nástrojů pro pokročilejší vývoj a rozsáhlou komunitu. Na druhou stranu vyžaduje používání dodatečných knihoven, což činí počáteční nastavení složitějším.

Závěrem lze konstatovat, že realizovaný praktický projekt splnil vytyčené cíle bakalářské práce a potvrdil vhodnost nasazení Reactu pro tvorbu moderních, výkonných a responzivních front-endových řešení. Získané zkušenosti s touto knihovnou a souvisejícími postupy budou cenným přínosem pro můj další profesní rozvoj.

7. Závěr

Cílem této bakalářské práce bylo provést detailní analýzu možností, které React knihovna poskytuje pro vývoj moderních webových aplikací. Práce se zaměřila zejména na návrh a implementaci funkčních komponent, správu stavů aplikace pomocí Reduxu a integraci směrování prostřednictvím React Router.

V teoretické části byly objasněny klíčové koncepty knihovny React jako kompozice komponent, podmíněný rendering nebo zpracování událostí. Pozornost byla věnována také životnímu cyklu funkčních komponent a využití React Hooks pro správu stavu a vedlejších efektů. Další část se zabývala problematikou správy komplexních datových toků na úrovni celé aplikace pomocí Redux knihovny, včetně popisu principů akce-reducer-store a integrace s Reactem.

Stěžejní součástí práce byla praktická implementace pilotní front-endové aplikace s využitím nabytých teoretických poznatků. V rámci vývoje byly aplikovány techniky návrhu a kompozice funkčních komponent, propojení se správou globálního stavu v Reduxu a konfigurace směrování pomocí React Router. Výsledný prototyp tak demonstruje komplexní využití zmíněných knihoven a jejich vzájemnou součinnost.

Zhodnocením vyvinuté aplikace lze konstatovat, že React ve spojení s podporujícími knihovnami představuje vysoce efektivní nástroj pro tvorbu robustních a škálovatelných Single Page Applications. Jeho přístup založený nakomponentách, virtuálním DOM a deklarativním vykreslování přináší výrazné benefity v oblasti výkonu, modularity a udržovatelnosti výsledných řešení.

Navzdory své komplexnosti díky rozsáhlé dokumentaci a aktivní komunitě podpory poskytuje React vývojářům přívětivé vývojové prostředí a ucelený ekosystém pro pokrytí všech aspektů vývoje front-endu aplikací. Zejména správa stavu pomocí Reduxu a React Routeru umožňuje efektivně zvládat i rozsáhlé a datově náročné aplikace.

I přes nesporné výhody je však nutno podotknout, že React nepředstavuje všespásný nástroj pro všechny případy užití. Jeho robustnost a vysoká úroveň abstrakce nemusí být vhodná pro jednoduché webové prezentace, kde by mohl být přístup založený na Reactu zbytečně složitý. Je proto na uvážení vývojářů, pro jaký typ aplikace je React vhodnou volbou.

Celkově lze tuto práci považovat za ucelený průvodce možnostmi a principy React knihovny a jejího ekosystému podpůrných knihoven. Uvedené teoretické základy i praktická ukázka jejich aplikace mohou posloužit jako odrazový můstek pro další studium této moderní front-endové technologie.

8. Seznam obrázku

9. Seznam použitých zdrojů

1. **Mikowski, M. S., & Powell, J. K.** *Developing Single Page Web Applications.* . Shelter Island, NY 11964 : Manning Publications Co., 2018. ISBN 9781617290756.
2. **Marini, Joe.** *Document Object Model : Processing Structured Documents* . místo neznámé : McGraw-Hill/Osborne Media; First Edition , July 24, 2002. 978-0072224368.
3. **DUCKETT, JON.** *2014.* Manufactured in the United States of America : John Wiley & Sons, Inc., 2014.
4. ***react.org.*** **[Online] Copyright © 2024 Meta Platforms, Inc., 2024. https://legacy.reactjs.org/docs/getting-started.html.**
5. React.js, Introduction to. https://www.youtube.com/watch?v=XxVg_s8xAms. [Online]
6. 2 React Podcast, «8. React, GraphQL, Immutable & Bow-Ties with Special Guest Lee Byron»,. [Online]
7. Wieruch, Robin. ***The Road to React.*** **Independently published : autor neznámý, September 14, 2018. 978-1720043997.**
8. Narayn, Hari. ***Just React!: Learn React the React Way.*** **New York, NY, USA : Apress ISBN-13: 978-1484282939, September 2021.**
9. Roldán, Carlos Santana. ***React 18 Design Patterns and Best Practices.*** **Birmingham, UK : Packt Publishing ISBN-13: 978-1803233109, August 2023.**
10. Thomas, Mark Tielens. ***React in Action.*** **UA : Manning Publications 978-1617293001, 2017.**
11. https://www.w3schools.com/html/. [Online] 1993- 2023. https://www.w3schools.com/html/.
12. Garreau, Marc. ***Redux in Action*** **. místo neznámé : Manning; First Edition, (June 14, 2018). 978-1617294976.**
13. Ganatra, Sagar. ***React Router Quick Start Guide: Routing in React applications made easy.*** **místo neznámé : Packt Publishing; 1st edition (September 29, 2018) , September 29, 2018. 1789532558.**
14. David, Flanagan. ***Javascript: The Definitive Guide: Master the World's Most-Used Programming Language.*** **USA : 9781491952023 Oreilly Media , 2020 .**
15. habr. ***habr.*** **[Online] 2006. https://habr.com/.**
16. Crockford, Douglas. ***JavaScript: The Good Parts.*** **místo neznámé : O'Reilly Media, 2008.**
17. https://web.archive.org/web/20160324075257/http://facebook.github.io/react/docs/why-react.html. [Online]
18. International, Ecma. https://ecma-international.org/about-ecma/history/. ***ecma-international.*** **[Online] Ecma International, 2023. https://ecma-international.org/.**
19. Zakas, Nicholas C. ***Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers.*** **místo neznámé : No Starch Press; 1st edition (August 16, 2016), August 16, 2016. 978-1593277574.**
20. Nate Murray, Ari Lerner. ***Fullstack React: The Complete Guide to ReactJS and Friends.*** **místo neznámé : Fullstack.IO (September 12, 2017), 2017. 978-0991344628.**
21. Moghe, Sumeet. ***The Async-First Playbook: Remote Collaboration Techniques for Agile Software Teams.*** **místo neznámé : Addison-Wesley Professional , October 7, 2023 . 978-0138187538.**