Univerzita Hradec Králové
Fakulta informatiky a managementu
Název katedry

Experimentální vyhodnocení frameworku Svelte

Bakalářská práce

Autor: Lukáš Tesař
Studijní obor: Aplikovaná Informatika

Vedoucí práce: Mgr. Daniela Ponce, Ph.D.

Hradec Králové Srpen 2022

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 24.8.2022 Lukáš Tesař

Poděkování:

Děkuji vedoucímu bakalářské práce Mgr. Daniele Ponce, Ph.D. za metodické vedení práce. Děkuji také Vítězslavu Zotovovi a Petru Schmidtovi za gramatickou a stylistickou kontrolu práce.

Anotace

Tato bakalářská práce se zabývá vyhodnocením frameworku Svelte v kontextu vývoje jednostránkových aplikací. Práce se skládá z pěti částí. První část popisuje jednotlivé technologie a principy, na kterých je vývoj jednostránkových aplikací založen. V druhé části je provedena analýza jednotlivých technologií, které jsou v práci pro vývoj aplikací použity a které jsou navzájem v rámci vyhodnocení porovnány, zejména Svelte, Vue.js a React. Analýza se zaměřuje na hlavní vlastnosti každé technologie. Následující část obsahuje popis implementace aplikace s použitím frameworku Svelte. Popis implementace je doprovázen názornými ukázkami kódu přímo z implementované aplikace. Poté je aplikace porovnána se stejnými aplikacemi vyvinutými za použití Reactu a Vue.js, kde je kladen důraz především na podobnosti nebo rozdíly implementací. Na závěr dochází k vyhodnocení frameworku Svelte.

Annotation

Title: Experimental evaluation of Svelte framework

This bachelor thesis deals with the evaluation of the Svelte framework in the context of single-page application development. The thesis consists of five parts. The first part describes the different technologies and principles underlying the development of single-page applications. The second part analyses the different technologies used in the thesis for application development and compares them with each other in the evaluation, notably Svelte, Vue.js and React. The analysis focuses on the main features of each technology. The following section describes the implementation of the application using the Svelte framework. The implementation description is accompanied by illustrative code samples directly from the implemented application. The application is then compared with the same applications developed using React and Vue.js, where the focus is mainly on the similarities or differences of the implementations. Finally, the Svelte framework is evaluated.

Obsah

1 Úvod ... 1
2 Cíl práce ... 3
3 Metodika zpracování ... 4
4 Technologie a principy ... 5
4.1 JavaScript ... 5
4.2 Node.js ... 5
4.3 Module Bundler ... 6
4.3.1 Svazková velikost ... 6
4.4 Správce balíčků ... 6
4.4.1 Npm ... 7
4.4.2 Yarn ... 7
4.5 Jednostránkové aplikace ... 7
4.5.1 Komponenty ... 8
4.5.2 Životní cyklus komponentů (lifecycle) ... 8
4.5.3 Stav (state) ... 8
4.5.4 Koncept směrování (routing) ... 9
5 Analýza technologií ... 10
5.1 React ... 10
5.1.1 Virtuální DOM ... 10
5.1.2 JSX ... 11
5.1.3 React Hooks ... 11
5.1.4 Jednosměrná datová vazba ... 13
5.2 Vue.js ... 14
5.2.1 Základní rysy ... 14
5.2.2 Vue direktivy ... 15

5.2.3 Options API 17

5.3 Svelte 17

5.3.1 Svelte kompilátor..18

5.3.2 Svelte bloky 20

5.3.3 Svelte direktivy 20

6 Návrh a implementace aplikace 22

6.1 Popis aplikace 22

6.2 Vývoj aplikace 23

6.2.1 Založení projektu 23

6.2.2 Struktura aplikace 25

6.2.3 Layout a směrování..27

6.2.4 Manipulace s úkoly..28

6.2.5 Renderování úkolů 31

6.2.6 Formuláře pro přidání úkolů. 32

6.2.7 Odesílání eventů...34

6.2.8 Soustředění se na úkol..36

6.2.9 Zacházení se styly 38

7 Porovnání aplikací 40

7.1 Velikost..40

7.2 Struktura 41

7.3 Layout a směrování 41

7.3.1 React 41

7.3.2 Vue.js 43

7.4 Práce s eventy 45

7.4.1 React 45

7.4.2 Vue.js 47

7.5 Globální stav ... 48
7.5.1 React ... 48
7.5.2 Vue.js ... 51
7.6 Styly ... 52
7.6.1 React ... 52
7.6.2 Vue.js ... 53
8 Vyhodnocení ... 54
8.1 Rozdíly a podobnosti ... 54
8.2 Použití při vývoji ... 54
8.3 Škálovatelnost ... 55
8.4 Výsledek ... 56
9 Shrnutí výsledků ... 57
10 Závěry a doporučení ... 58
11 Seznam použité literatury ... 59
12 Přílohy ... 62

Seznam obrázků

Obrázek 1: Žebříček spokojenosti vývojářů s frontendovými technologiemi. Zdroj [1] ... 1
Obrázek 2: Domovská stránka aplikace. Zdroj autor ... 23
Obrázek 3: Struktura nově vytvořeného projektu. Zdroj autor ... 24
Obrázek 4: Upravená struktura aplikace. Zdroj autor ... 25
Obrázek 5: Hierarchie komponentů. Zdroj autor ... 26
Obrázek 6: Seznam úkolů. Zdroj autor ... 32
Obrázek 7: Úkol s připomínkou. Zdroj autor ... 36
Obrázek 8: Zobrazený úkol na stránce Focus. Zdroj autor ... 36
Obrázek 9: Třída container s unikátním selektorem. Zdroj autor ... 39
Obrázek 10: Graf růstu svazkových velikostí Reactu a Svelte na základně velikosti zdrojového kódu komponentů. Zdroj [37] ... 56

Seznam tabulek

Tabulka 1: Velikosti repozitářů aplikací a jejich svazků. Zdroj autor ... 40

Seznam ukázek kódu

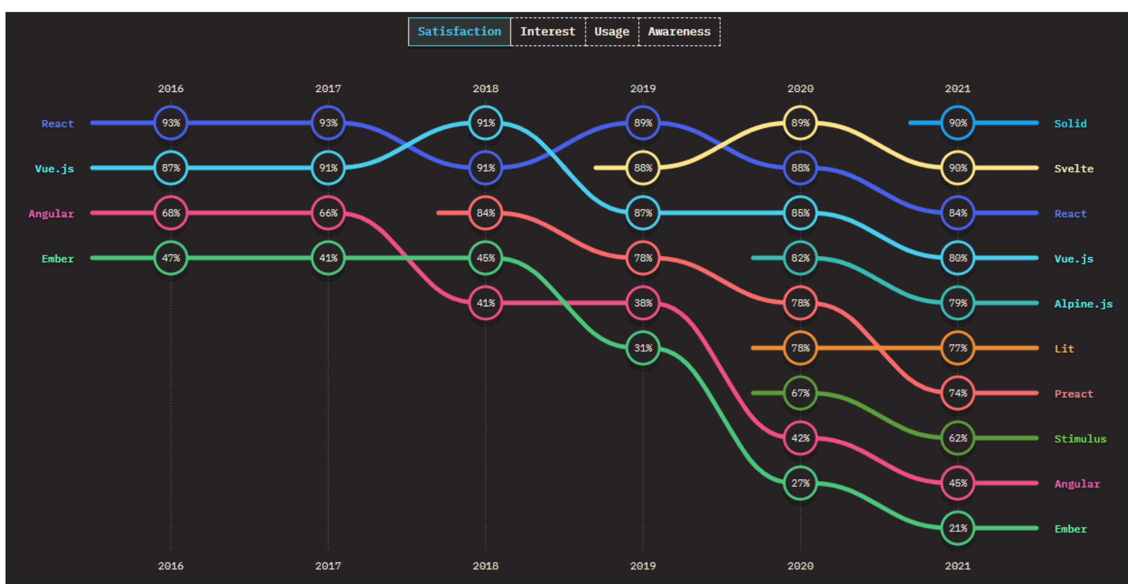
Ukázka kódu 1: Příklad manipulace s DOM. Zdroj autor 10
Ukázka kódu 2: Příklad užití JSX. Zdroj autor 11
Ukázka kódu 3: Použití hooku useState. Zdroj autor 12
Ukázka kódu 4: Použití hooku useEffect. Zdroj autor 12
Ukázka kódu 5: Použití hooku useContext. [22] 13
Ukázka kódu 6: Použití jednosměrné datové vazby. Zdroj autor 14
Ukázka kódu 7: Kódové rozložení Vue.js. Zdroj autor 15
Ukázka kódu 8: Použití v-if a v-else. Zdroj autor 15
Ukázka kódu 9: Použití v-for v kombinaci s :key. Zdroj autor 16
Ukázka kódu 10: Použití direktivy v-bind pro aplikovaní dynamických stylů. Zdroj autor 16
Ukázka kódu 11: Použití direktivy v-on. Zdroj autor 16
Ukázka kódu 12: Použití direktivy v-model. Zdroj autor 17
Ukázka kódu 13: Definování komponentu pomocí Options API. Zdroj autor 17
Ukázka kódu 14: Kódové rozložení Svelte. Zdroj autor 18
Ukázka kódu 15: JavaScript vygenerovaný Svelte kompilátorem. Zdroj autor 19
Ukázka kódu 16: CSS vygenerované Svelte kompilátorem. Zdroj autor 19
Ukázka kódu 17: Použití bloků if, else a else if. Zdroj autor 20
Ukázka kódu 18: Použití bloku each s indexováním. Zdroj autor 20
Ukázka kódu 19: Použití direktivy on. Zdroj autor 21
Ukázka kódu 20: Použití direktivy bind. Zdroj autor 21
Ukázka kódu 21: Použití direktivy style. Zdroj autor 21
Ukázka kódu 22: Sekvence příkazů pro inicializaci Svelte projektu. Zdroj autor 23
Ukázka kódu 23: Implementace App.svelte. Zdroj autor 27
Ukázka kódu 24: Implementace komponentu NavButtons. Zdroj autor 28
Ukázka kódu 25: Inicializace pole úkolů při načtení komponentu Home. Zdroj autor 29
Ukázka kódu 26 Komponent Tasks s atributy. Zdroj autor 29
Ukázka kódu 27: Implementace funkce addTask. Zdroj autor 29
Ukázka kódu 28: Implementace funkcí deleteTask a toggleReminder. Zdroj autor 30

Ukázka kódu 29: Implementace komponentu Tasks. Zdroj autor 31
Ukázka kódu 30: Implementace komponentu Header. Zdroj autor 33
Ukázka kódu 31: Implementace komponentu AddTask. Zdroj autor 34
Ukázka kódu 32: Implementace komponentu Task. Zdroj autor 35
Ukázka kódu 33: Implementace globálního stavu TaskStore. Zdroj autor 37
Ukázka kódu 34: Čtení hodnoty globálního stavu uvnitř komponentu. Zdroj autor 37
Ukázka kódu 35: Alternativní postup čtení hodnoty globálního stavu: Zdroj autor 38
Ukázka kódu 36: Styly uvnitř komponentu Focus. Zdroj autor 39
Ukázka kódu 37: Kořenový komponent App. Zdroj autor 42
Ukázka kódu 38: Kořenový komponent App. Zdroj autor 43
Ukázka kódu 39: Definování cest v souboru router/index.js. Zdroj autor 44
Ukázka kódu 40: Vytvoření Vue instance v souboru main.js. Zdroj autor 45
Ukázka kódu 41: Funkce deleteTask v komponentu Home. Zdroj autor 45
Ukázka kódu 42: Komponent Tasks uvnitř JSX výrazu komponentu Home. Zdroj autor 46
Ukázka kódu 43: Implementace komponentu Tasks. Zdroj autor 46
Ukázka kódu 44: Ikona křížku v komponentu Task s metodou onDelete. Zdroj autor 47
Ukázka kódu 45: Definování funkce handleDelete uvnitř komponentu Task. Zdroj autor 47
Ukázka kódu 46: Přiřazení funkce handleDelete na event click ikony. Zdroj autor 47
Ukázka kódu 47: Odeslání eventu uvnitř komponentu Tasks do komponentu Home. Zdroj autor 48
Ukázka kódu 48: Přiřazení funkcí k příslušným eventům. Zdroj autor 48
Ukázka kódu 49: Volaná funkce deleteTask. Zdroj autor 48
Ukázka kódu 50: Obsah souboru index.js uvnitř adresáře context. Zdroj autor 49
Ukázka kódu 51: Obalení komponentů uvnitř cest komponentem FocusTaskContextProvider. Zdroj autor 49
Ukázka kódu 52: Vytažení stavu z poskytovatele kontextu a jeho čtení uvnitř komponentu Focus. Zdroj autor 50

Ukázka kódu 53: Vytažení setteru stavu z poskytovatele kontextu a jeho použití uvnitř komponentu Task. Zdroj autor 50
Ukázka kódu 54: Definování Vuex instance uvnitř index.js v adresáři store. Zdroj autor..51
Ukázka kódu 55: Funkce handleFocus volající akci setTask uvnitř komponentu Task. Zdroj autor..52
Ukázka kódu 56: Mapování getteru getTask uvnitř možnost computed komponentu Focus. Zdroj autor...52
Ukázka kódu 57: Použití getteru getTask uvnitř šablony komponentu Focus. Zdroj autor..52
Ukázka kódu 58: Definovaný objekt se styly uvnitř komponentu Focus. Zdroj autor 53
Ukázka kódu 59: Definování stylů uvnitř komponentu Focus. Zdroj autor 53

1 Úvod

Od počátku 21. století postupně roste nárok na komplexitu frontend části webových aplikací. Mezi důvody mohou patřit například větší požadavky ze stran uživatelů na způsobilost a uživatelskou přívětivost aplikací nebo udržitelnost a škálovatelnost při jejich vývoji. Bylo zřejmé, že prostředky jako standardní HTML, CSS a JavaScript nemohou s rostoucím nárokem na tuto složitost do budoucna stačit, a proto vznikly jednostránkové aplikace (single-page application, zkráceně SPA). Princip SPA již existuje přes 10 let a ukázal se jako prominentní způsob tvorby webových aplikací. Za poslední léta vznikla řada technologií založených na tomto principu, kdy mezi nejpopulárnější lze řadit framework Vue.js a knihovnu React. Jedním z nejnovějších a nejoblíbenějších frameworků je Svelte. [1]



Obrázek 1: Žebříček spokojenosti vývojářů s frontendovými technologiemi. Zdroj [1]

Svelte přináší nové koncepty mezi stávající standardy a zdá se být na vzestupu. Právě proto se tato práce zabývá experimentálním vyhodnocením tohoto frameworku v rámci vývoje jednostránkových aplikací. K docílení vyhodnocení budou nejprve rozebrány základní koncepty, na kterých vývoj SPA stojí. Následovat bude analýza konkurenčních technologií společně se Svelte. Poté dojde k popisu a implementaci aplikace vytvořené ve Svelte a její porovnání s obdobnými

aplikacemi za použití porovnávaných technologií. Nakonec budou získané poznatky použity pro vyhodnocení.

2 Cíl práce

Cílem této bakalářské práce je vyhodnotit framework Svelte v rámci vývoje jednostránkových aplikací. K tomuto vyhodnocení je zapotřebí nejprve rozebrat základní technologie a principy, na kterých je vývoj jednostránkových aplikací založen. Práce zkoumá framework Svelte a technologie se kterými je porovnáván. Zaměřuje se především na popis základních rysů těchto technologií a uvedení příkladů jejich používání. V práci dochází k návrhu a implementaci aplikace s pomocí Svelte a porovnání s identickými aplikacemi vytvořenými za použití konkurenčních technologií. Poznatky z tohoto zkoumání jsou společně s odkazovanými výzkumy použité k výslednému vyhodnocení. Vyhodnocení se zaměřuje především na otázku kompetentnosti Svelte jako alternativa obdobných technologií pro vývoj SPA.

3 Metodika zpracování

V první řadě dojde k představení důležitých technologií a principů, na kterých je vývoj jednostránkových aplikací založen. Následně budou zanalyzovány frameworky Vue.js, Svelte a knihovna React. V analýze budou vyzdvihnuty jejich základní rysy a příklady jejich použití. Vue.js a React patří mezi nejpoužívanější a nejoblíbenější frontendové technologie. [1] Zároveň aplikují navzájem dostatečně odlišné přístupy a tím slouží jako dobrý vzorek pro porovnání. Dále bude navržena a implementována aplikace s použitím frameworku Svelte. Tato aplikace je následně porovnána s aplikacemi vytvořenými ve Vue.js a Reactu. Všechny tři aplikace jsou z hlediska funkcí identické a liší se pouze v implementacích. Díky tomu budou rozdíly zřejmé. Nakonec dochází k vyhodnocení, ke kterému jsou použity poznatky získané během zkoumání společně s odkazovanými výzkumy.

4 Technologie a principy

K vývoji jednostránkových aplikací jsou používány určité technologie a principy, které je důležité identifikovat. Tato část se zaměřuje na popis těchto technologií a principů. Pozdější kapitoly je dále rozvádějí o praktické ukázky.

4.1 JavaScript

JavaScript (zkráceně JS) je vysokoúrovňový interpretovaný programovací jazyk. Je dynamicky typovaný a podporuje mnoho programovacích přístupů. Mezi významné patří objektově orientované programování (zkráceně OOP), funkcionální programovaní, imperativní (procedurální) programování a programování řízené událostmi (event-driven). Díky jeho vestavěné podpoře v každém moderním prohlížeči je především známý jako skriptovací jazyk pro frontend webových aplikací. S příchodem běhových prostředí (runtime environment), které jsou schopny pracovat mimo internetový prohlížeč, se užití JavaScriptu rozšířilo a přibyly tak možnosti vývoje backendu webových aplikací, desktopových aplikací a mobilních aplikací. Jedním z nejpoužívanějších běhových prostředí je Node.js. [2]

4.2 Node.js

Node.js je multiplatformní JavaScriptové běhové prostředí původně vydané 27. května 2009. [3] Za jeho počinem stojí Ryan Dahl, který jej vydal jako open source projekt a dnes se jedná o nejrozšířenější JavaScript runtime. Jako výchozí správce balíčků (package manager) je používán npm, který má bohatý ekosystém obsahující přes 1 milión balíčků dostupných zcela zdarma. [4] Node.js je postavené na V8 JavaScript enginu od společnosti Google, který je napsán v C++. Díky tomu je Node.js schopno daný JavaScript relativně rychle kompilovat.

Mezi další významné JavaScriptové běhové prostředí patří Deno a Bun. Deno vyvinul Ryan Dahl v roce 2018 za účelem vytvořit prostředí, které napraví to, co Dahl vnímal jako největší slabiny Node.js. [5] Bun je běhové prostředí vyvinuté v roce 2022. Podle zdrojů poskytnutých přímo od jeho vývojářů se jedná o nejrychlejší JavaScriptové běhové prostředí momentálně vyvinuté. [6] V této práci se používá pouze Node.js kvůli jeho stabilitě.

4.3 Module Bundler

Module bundler je nástroj, který slouží ke kompilaci jednotlivých JavaScriptových modulů a jejich závislostí (dependancies) do jednoho souboru, tzv. svazku (bundle), který je posléze použitý prohlížečem k zobrazení produkční verze aplikace. S množstvím závislostí, které dnešní frameworky a knihovny mají, je prakticky nemožné vyvíjet webovou aplikaci bez použití bundleru. [7]

Mezi nejpoužívanější bundlery patří webpack, Rollup a Vite. Webpack je výchozí bundler používaný pro React a Vue.js. Svelte dříve upřednostňoval Rollup, ale dnes doporučuje Vite.

4.3.1 Svazková velikost

Svazková velikost určuje bytovou velikost JavaScriptu, který je potřeba stáhnout na koncové zařízení uživatele při používání aplikace. Větší velikost svazku prodlouží načítání aplikace a při velké velikosti může dojít k opravdu velkému zpomalení, což může vyvolat frustraci ze strany uživatele.

Velikost kódu aplikace při použití v produkci je minifikována. Minifikace slouží k odstranění mezer, komentářů a nepotřebných středníků k redukci bytové délky. Nejedná se o podobu kódu, která by byla žádaná při vývoji, ale u hotové aplikace lze takto docílit odstranění nadbytečných dat, které by jinak muselo zařízení při použití aplikace stahovat. [8]

4.4 Správce balíčků

Při vývoji větší JavaScriptové aplikace je důležité mít k dispozici správce balíčků. Správce balíčků lze vnímat jako software, který se stará o řadu věcí týkajících se jednotlivých knihoven rozšiřující danou aplikaci. Možnosti, které správce balíčků nabízí, jsou např. instalace jednotlivých knihoven, jejich aktualizování a odinstalování. Správců balíčků je více, mezi nejznámější a nejpoužívanější patří npm a Yarn. [9]

4.4.1 Npm

Npm (Node.js Package Manager) je výchozí správce balíčků pro Node.js, který vyvinul Isaac Z. Schlueter a vydal ho v roce 2010. [10] Npm umožňuje sdílení jak vlastního, tak cizího kódu ve formě knihoven, čímž zrychluje celkový proces vyvíjení JavaScriptových aplikací.

Skládá se ze 3 částí: webové stránky, rozhraní příkazové řádku (Command Line Interface, zkráceně CLI) a databáze. [11]

Webové stránky slouží k hledání balíčků, informací o nich a jejich autorech. CLI poté umožňuje přes příkazy interakci s balíčky, jejich stahování a správu. Informace o jednotlivých balíčcích jsou ukládány do souboru package.json, který je uložen v kořenové složce aplikace. Veškeré balíčky jsou ukládány do složky node_modules, nacházející se též v kořenové složce aplikace.

4.4.2 Yarn

Yarn funguje velmi obdobně jako npm. Hlavním rozdílem je v přistupování k správci, kdy každý příkaz začíná slovem yarn namísto npm. Jsou zde také určité rozdíly v organizaci generovaných soubor. Yarn vyvinula společnost Facebook jako rychlejší a bezpečnější alternativu npm. Dnes je tento správce open-source a balíčky jsou s npm vzájemně sdílené, tudíž rozdíl v používání těchto dvou balíčků je nepatrný.

4.5 Jednostránkové aplikace

U běžných webových aplikací je zapotřebí při každé změně obsahu znovu načíst stránky prohlížečem. Tím dochází k neustálému stahování nových dokumentů, což se projeví na rychlosti aplikace. Princip jednostránkových aplikací spočívá v načtení pouze jednoho webové dokumentu, jehož obsah se dynamicky mění na základě interakcí uživatele. Jedná se o druh webových aplikací, které fungují bez jediného znovunačtení prohlížečem. Díky tomu bývají rychlejší a pro uživatele příjemnější. Nevýhodou však je složitější architektura aplikací, která se může projevit na ceně a času potřebného k jejich vývoji. Zároveň je u nich velký problém v případě optimalizace pro vyhledávače (search engine optimization, zkráceně SEO), kdy se

tzv. *crawler* velmi obtížně v aplikaci orientuje a daná stránka se tak může lidem při vyhledávání méně zobrazovat. [12]

4.5.1 Komponenty

Jedním z hlavních konceptů jednostránkových aplikací jsou komponenty. Komponenty lze definovat jako individuální části, ze kterých se aplikace skládají. Může se jednat o komponenty větší, které jsou na úrovni celých sekcí aplikace a jsou funkcionálně robustní, anebo komponenty menší, které mohou sloužit i jako pouhý UI doplněk. Často se jedná o znovupoužitelnou část aplikace a kvůli tomu je osvědčeným postupem komponenty často rozkládat na více menších.

V každé moderní SPA knihovně nebo frameworku komponenty dovolují definovat tzv. *props*. Props mohou být jakákoliv data, která chceme komponentu předat. Předávání připomíná specifikování standardních HTML atributů. Název je odvozen od anglického *property* (vlastnost). [13]

4.5.2 Životní cyklus komponentů (lifecycle)

Komponenty mají podobně jako lidi svůj vlastní životní cyklus. Jsou zrozeny, rostou a umírají. V případě komponentů se jedná o jejich vytvoření (namontování), obnovování a zničení (odmontování). V SPA technologiích se k těmto etapám přistupuje jako k funkcím, které automaticky provádí kód podle fáze, ve které se komponent zrovna nachází. Když je komponent zapotřebí vykreslit (renderovat), dojde k jeho vytvoření. Když se změní jeho stav, dojde k jeho obnovení a když ho už není potřeba vykreslovat, dojde k jeho zničení. Frameworky se k tomuto konceptu staví různě, avšak principiálně podobně. [14]

4.5.3 Stav (state)

Stavem jsou myšleny data, která náleží nějakému komponentu. Nad těmito daty lze provádět všelijaké operace, jako se dají provádět s daty uloženými v běžných proměnných, objektech či jiných datových strukturách. Hlavním rozdílem stavu je možnost detekovat jeho změny. Díky tomu se dají stavět celé algoritmy založené na změně nějakého stavu.

Kromě běžného stavu komponentu existuje také globální (sdílený) stav. Jedná se o stav, který je možno sdílet po celé aplikaci a je tak přístupný více komponentům.

4.5.4 Koncept směrování (routing)

V běžné webové aplikaci dochází ke směrování propojením jednotlivých HTML dokumentů skrze hypertextové odkazy. Ty nelze v případě jednostránkových aplikací použít, protože by došlo k znovunačtení stránky, což není žádané. Právě proto se užívá JavaScriptový směrovač, který organizuje směrování aplikace a přepíná zobrazovaný obsah mezi definovanými pohledy. Individuální SPA frameworky využívají k implementaci směrovače externí knihovny. Směrování je zapotřebí manuálně sestavit. [15]

5 Analýza technologií

V této části dochází k analýze jednotlivých frameworků a knihoven použitých k porovnání se Svelte. Jde především o popsání jejich hlavních prostředků, které nabízí a jaké koncepty uplatňují.

5.1 React

Na rozdíl od Svelte a Vue.js je React knihovna, nikoliv framework, avšak účel plní stejný. React byl vytvořen společností Facebook v květnu 2013 za účelem vyvinout jednoduchý, ale silný nástroj pro tvorbu jednostránkových aplikací. Tento cíl se dá dnes považovat za velký úspěch, protože React je dlouhodobě nejpoužívanější technologií k vývoji SPA. [16] Jednoduchost a flexibilita je bez debat nejsilnější vlastností Reactu a díky obrovskému ekosystému rozšiřujících knihoven dokáže skvěle obstát i oproti robustním frameworkům.

5.1.1 Virtuální DOM

Když webový prohlížeč načte webovou stránku, vygeneruje se tzv. objektový model dokumentu (Document Object Model, zkráceně DOM). Tento dokument reprezentuje obsah webové stránky ve stromové struktuře pro snadné strojové zpracování. Kvůli principu, na kterém jednostránkové aplikace fungují, je zapotřebí DOM dynamicky měnit. Díky JavaScriptu je možné s DOM manipulovat. [17]

```
const element = document.getElementById('hello-world')
element.innerHTML = 'Hello World!'
```

Ukázka kódu 1: Příklad manipulace s DOM. Zdroj autor

Přímá manipulace s DOM má ale dva zásadní problémy. Prvním problémem je rychlost a druhým ještě větším je imperativní povaha kódu. Z dlouhodobého hlediska by takový kód neškáloval vůbec dobře, a proto React používá tzv. virtuální DOM. [17]

Virtuální DOM je odlehčená (lightweight) virtuální kopie DOM. Každá položka, která se nachází v DOM, se nachází zároveň i v jeho virtuální kopii. Díky tomu, že se virtuální DOM nevykresluje při každé změně, jsou manipulace s ním mnohem rychlejší, než z opravdovým DOM. Při každé změně stavu se vytvoří nová kopie

virtuálního DOM, která se porovná s tou předchozí a poznamenají se mezi nimi rozdíly. Ty se pak použijí k nálezu nejrychlejšího způsobu, jak aktualizovat opravdový DOM. [18]

5.1.2 JSX

JSX je rozšíření JavaScriptu specifické pro React. Na rozdíl od frameworků jako Svelte nebo Vue.js, které se vydaly směrem šablon (template), React zůstává v čistém JavaScriptu s možností používat JSX. Ve své podstatě se jedná o možnost psaní výrazů podobných HTML přímo v JavaScriptu. Jeho užití není nutné, ale doporučené. JSX výrazy jsou obsažený v návratové hodnotě funkčně deklarovaného komponentu. Oproti standardnímu HTML se v pár věcech liší. Například atribut class se zapisuje jako className nebo místo dvojitých uvozovek lze používat jednoduché. Pro použití JSX stačí změnit příponu JavaScriptového souboru na „.jsx“. [19]

```
const Error = () => {
  return (
    <div className='page-error'>
      <h1>404</h1>
      <p>Page not found</p>
    </div>
  );
};
```

Ukázka kódu 2: Příklad užití JSX. Zdroj autor

5.1.3 React Hooks

V únoru 2019 se React vydal kompletně jiným směrem s představením verze 16.8. Namísto tradičních komponentů založených na třídách implementoval funkčně definované komponenty a hooky. [20]

Hook lze definovat jako funkci, která dovoluje „se zaháknout“ do stavu a životního cycklu komponentu. Hooky nefungují uvnitř tříd a ani ve vnořených funkcích. Musí být volány přímo ve funkčně definovaném komponentu. React implementuje celkem patnáct hooků. Jejich název zpravidla začíná klíčovým slovem „use“. Mezi nejdůležitější a nejpoužívanější patří useState, useEffect a useContext. [21]

Hook useState je funkce, která vrací pole dvou položek. Jednou je hodnota stavu a druhou je setter stavu. Jako parametr příjímá výchozí hodnotu stavu. K uložení je často používána tzv. destrukturace (*destructuring*).

```
const [tasks, setTasks] = useState([]);
```

Ukázka kódu 3: Použití hooku useState. Zdroj autor

Hook useEffect je funkce, která nahradila tři dřívější metody. Byly jimi componentDidMount, componentDidUpdate a componentWillUnmount**.** Jak už názvy mohou napovídat, jedná se o hook, který pracuje s životním cyklem komponentu. Má celkem dva parametry. Prvním parametrem je funkce, která má být provedena. Druhým parametrem je pole závislostí. Pokud pole závislostí obsahuje stav, jehož hodnota se změní, spustí se tím funkce obsažená v prvním parametru. Pokud je pole závislostí prázdné, hook provede funkci pouze když dojde ke vzniku komponentu. Když pole závislostí chybí úplně, funkce se provádí kdykoliv dojde ke změně stavu. Pokud funkce v prvním parametru vrací funkci, hook zavolá tuto funkci v momentě, kdy dojde k jeho zničení. [21] Příklad užití useEffect se nachází na ukázce kódu níže.

```
useEffect(() => {
    // getTasks and set them into state
    const getTasks = async () => {
      const tasksFromServer = await fetchTasks();
      setTasks(tasksFromServer);
    };

    getTasks();
  }, []);
```

Ukázka kódu 4: Použití hooku useEffect. Zdroj autor

Třetím důležitým hookem je useContext. Tento hook se používá v kombinaci s metodou createContext, která vytvoří kontext.

```
const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{
     background: theme.background, color:  theme.foreground}}
    >
      I am styled by theme context!
    </button>
  );
```

Ukázka kódu 5: Použití hooku useContext. [22]

Na ukázce výše dojde k vytvoření ThemeContext. Poté je obalením ThemeContext.Provider poskytnuta pro komponent Toolbar hodnota themes.dark. Uvnitř komponentu Toolbar se nachází komponent ThemedButton, v kterém se použitím hooku useContext získá daná hodnota. Díky kontextu není zapotřebí provádět tzv. *prop drilling*, kdy by musela být potřebná hodnota předávána přes jednotlivé úrovně komponentů.

5.1.4 Jednosměrná datová vazba

K provázání dat React používá jednosměrnou datovou vazbu (one-way data binding). To znamená, že například u provázání dat uvnitř formuláře je potřeba

provázat jak hodnotu vstupu, tak způsob její změny. K tomu se dá použít například event onChange.

```
return (
  <form onSubmit={onSubmit} className='add-form'>
    <div className='form-control'>
      <label>Task</label>
      <input
        type='text'
        placeholder='Add Task'
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
    </div>
```

Ukázka kódu 6: Použití jednosměrné datové vazby. Zdroj autor

5.2 Vue.js

V roce 2014 byl vydán nový framework pro tvorbu jednostránkových aplikací. Jelikož v té době patřily mezi hlavní technologie pro vývoj jednostránkových aplikací Angular a React, Evan You vytvořil Vue.js, který nabízí to nejlepší z obou světů. Vue.js je dostatečně lightweight na to, aby rychlostí obstál Reactu a zároveň obsahuje řadu vestavěných funkcí, které se musí v případě Reactu importovat z externích knihoven. Od roku 2020 je Vue.js separované na verzi 2 a 3, kdy se v případě verze 3 jedná o kompletní přepsání frameworku s větší inspirací moderního Reactu. Počátkem února 2022 je verze 3 oficiálně brána jako výchozí. [23, 24]. Tato práce se zaměřuje pouze na verzi 2 z důvodu větší odlišnosti od Reactu.

5.2.1 Základní rysy

Vue.js využívá obdobně jako React virtuální DOM. V čem se ale oproti Reactu liší, je vlastním typem souboru „.vue“. Tím je umožněno vývojářům psát běžnou HTML syntax, JavaScript a CSS v jednom souboru. HTML se vkládá do tagu <template>, JavaScript do tagu <script> a CSS do tagu <style>.

```
<template>
  <h1 class="greeting">{{ greeting }}</h1>
</template>

<script>
export default {
  data() {
    return {
      greeting: 'Hello world!',
    };
  },
};
</script>

<style>
.greeting {
  color: aqua;
}
</style>
```

Ukázka kódu 7: Kódové rozložení Vue.js. Zdroj autor

5.2.2 Vue direktivy

K ovlivnění výsledného renderování DOM se používají direktivy. Direktivy jsou atributy začínající písmenem „v“, které lze použít přímo na elementu uvnitř šablony. Mohou přistupovat přímo do stavu komponentu. Vue.js má celkem patnáct direktiv, z toho nejdůležitější jsou v-show, v-if, v-else, v-for, v-bind, v-on a v-model. [25]

Direktivy v-show, v-if a v-else se používají k podmíněnému renderování. Jako hodnotu přijímají boolean, nebo nějaké tvrzení, jehož výsledkem je boolean. Hlavním rozdílem mezi v-show a v-if je ten, že pokud je tvrzení uvnitř v-if nepravdivé, nedojde k vyrenderování daného elementu vůbec, mezitím co u v-show se element vyrenderuje a je schovaný pouze pomocí CSS. Direktiva v-else je použitá pro opačný stav tvrzení uvnitř rodičovského v-if.

```
<div v-if="getTask.text" class="task">
  <h3>{{ getTask.text }}</h3>
  <p>{{ getTask.day }}</p>
</div>
<div v-else class="task text-center">
  <p>There is no task you are focused on</p>
</div>
```

Ukázka kódu 8: Použití v-if a v-else. Zdroj autor

Direktiva v-for slouží k renderování více elementu skrze iteraci. Je dobré ho používat v kombinaci s „:key“ pro ustanovení pořadí elementů.

```
<Task
  v-for="task in tasks"
  :key="task.id"
  @delete-task="$emit('delete-task', task.id)"
  @toggle-reminder="$emit('toggle-reminder', task.id)"
  :task="task"
/>
```

Ukázka kódu 9: Použití v-for v kombinaci s :key. Zdroj autor

Direktiva v-bind slouží k dynamickému provázání atributů elementu nebo props komponentu. Lze v ní využít i tvrzení, které může sloužit například pro dynamické nastavení CSS tříd elementu. Namísto v-bind lze použít znaménko „:“.

```
<button
  class="btn"
  :style="`backgroundColor: ${color}`" @click="onClick">
  {{ text }}
</button>
```

Ukázka kódu 10: Použití direktivy v-bind pro aplikovaní dynamických stylů. Zdroj autor

Direktiva v-on se používá k naslouchání eventů. Pokud nastane event, vykoná se přiřazená funkce. Vue.js má vestavěné eventy jako například v-on:click, ale zároveň je možné použít vlastní eventy. Použití lze zkrátit pomocí znaménka „@“.

```
<i
  class="fas fa-check"
  style="color: blue"
  @click="handleFocus(task)"
/>
```

Ukázka kódu 11: Použití direktivy v-on. Zdroj autor

Direktiva v-model slouží k provázání dat vstupů ve formuláři. Toto provázání je aplikované dvěma směry, což znamená, že změny dat se projeví jak změnou vstupu formuláře, tak změnou ve scriptu. Jedná se o odlišný přístup oproti Reactu, kde existuje provázání dat pouze jedním směrem (viz kapitola 5.1.4).

```
<input type="text" placeholder="Add Task" v-model="text" />
```

Ukázka kódu 12: Použití direktivy v-model. Zdroj autor

5.2.3 Options API

Vue.js implementuje Options API, které slouží k definování komponentů. Název je odvozen od možností (options), které jsou od API k dispozici. Jedná se v podstatě o funkce a objekty, které definují komponent. Mezi tyto možnosti patří například data, methods a created. [26]

- Možnost data umožňuje implementovat stav komponentu. Jedná se o funkci, která vrací objekt s definovanými stavovými proměnnými.
- Možnost methods je objekt, který obsahuje deklarované funkce komponentu.
- Možnost created je lifecycle funkce, která se volá při vytvoření komponentu, tedy ještě před renderováním do DOM.

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!',
    };
  },
  methods: {
    changeGreeting() {
      this.greeting = 'Hello Vue!';
    },
  },
  created() {
    this.changeGreeting();
  },
};
</script>
```

Ukázka kódu 13: Definování komponentu pomocí Options API. Zdroj autor

5.3 Svelte

Svelte je JavaScriptový framework pro tvorbu jednostránkových aplikací vydaný v roce 2016 Richem Harrisem. V mnoha ohledech funguje velice podobně jako obdobné technologie Vue.js a React, avšak hlavní rozdíl je v tom, co se děje na pozadí.

Svelte totiž nepoužívá k renderování virtuální DOM, ale vlastní kompilátor, který kód napsaný vývojářem transformuje do optimálního JavaScriptu. Díky tomu jsou aplikace implementované s jeho pomocí velice výkonné a jejich kód je méně objemný. Od roku 2019 se Svelte řadí mezi nejoblíbenější frontendové frameworky (viz obrázek 1). [27, 28]

5.3.1 Svelte kompilátor

Kód generovaný Svelte kompilátorem generuje JavaScript, který přímo upravuje DOM webové stránky. Obdobně jako Vue.js má Svelte vlastní souborový typ s příponou „.svelte", který umožňuje použití šablony. Do tohoto souboru lze psát HTML, JavaScript a CSS. JavaScript je obsažen v tagu <script>, CSS jsou obsažené v tagu <style> a HTML žádný vyžadovaný tag nemá. Kompilátor vezme obsah souboru se Svelte kódem a převede ho do tzv. abstraktního syntaktického stromu (Abstract Syntax Tree, dále AST). Poté dojde k analýze AST a vygenerování výsledného JavaScriptu a CSS. [29]

```
<script>
  let greeting = 'Hellow World!'
</script>

<h1>{greeting}</h1>

<style>
  h1 {
    color: aqua;
  }
</style>
```

Ukázka kódu 14: Kódové rozložení Svelte. Zdroj autor

```
/* App.svelte generated by Svelte v3.49.0 */
import {
  SvelteComponent,
  attr,
  detach,
  element,
  init,
  insert,
  noop,
  safe_not_equal
} from "svelte/internal";

function create_fragment(ctx) {
  let h1;

  return {
    c() {
      h1 = element("h1");
      h1.textContent = `${greeting}`;
      attr(h1, "class", "svelte-rn3res");
    },
    m(target, anchor) {
      insert(target, h1, anchor);
    },
    p: noop,
    i: noop,
    o: noop,
    d(detaching) {
      if (detaching) detach(h1);
    }
  };
}

let greeting = 'Hellow World!';

class App extends SvelteComponent {
  constructor(options) {
    super();
    init(this, options, null, create_fragment, safe_not_equal, {});
  }
}

export default App;
```

Ukázka kódu 15: JavaScript vygenerovaný Svelte kompilátorem. Zdroj autor

```
h1.svelte-rn3res{color:aqua}
```

Ukázka kódu 16: CSS vygenerované Svelte kompilátorem. Zdroj autor

5.3.2 Svelte bloky

Bloky ovlivňují podobu zobrazeného DOM. Skládají se z otevírací a uzavírací části. Otevírací část začíná znaménky „#{" a končí znaménkem „}". Uzavírací část začíná znaménky „{/" a končí znaménkem „}". K ovlivnění renderování elementu je zapotřebí obalit jej do otevírací a uzavírací části bloku. Výjimkou je blok else, který nemá uzavírací část a element se vkládá pod něj. Místo znaménka „#" je u něj použita „:". Mezi hlavní bloky patři if, else a each. [30]

Blok if slouží k podmíněnému renderování elementů. Uvnitř otevírací části bloku je očekáván boolean, nebo tvrzení výsledkem boolean. Tento blok lze použít společně s bloky else a else if k jeho větvení.

```
<script>
  let greeting = 'Hellow World!';
  let ranNum = Math.floor(Math.random() * 15);
</script>

{#if ranNum > 9}
  <h1>{greeting}</h1>
{:else if ranNum > 4}
  <h1>Hello Svelte!</h1>
{:else}
  <h1>Hello something else!</h1>
{/if}
```

Ukázka kódu 17: Použití bloků if, else a else if. Zdroj autor

Blok each zobrazuje elementy iterací přes seznam hodnot. Pro indexování je možné uvést hodnotu do závorek uvnitř otevírací části bloku.

```
{#each tasks as task (task.id)}
  <Task {task} on:delete-task on:toggle-reminder />
{/each}
```

Ukázka kódu 18: Použití bloku each s indexováním. Zdroj autor

5.3.3 Svelte direktivy

Svelte disponuje direktivami podobně jako Vue.js. Vkládají se jako atribut k elementu a ovlivňují jeho chování. Mezi hlavní patří on, bind, style a class.

Direktiva on umožňuje elementu naslouchání eventu. Jsou podporovány výchozí DOM eventy jako například submit.

```
<form class="add-form" on:submit={handleSubmit}>
```

Ukázka kódu 19: Použití direktivy on. Zdroj autor

Bind je direktiva používaná k provázání dat mezi skriptem a elementy. Používá se v kombinaci s běžným atributem daného elementu, jako například value na elementu input. Jedná se o provázání dvěma směry.

```
<input type="text" placeholder="Add Task" bind:value={text} />
```

Ukázka kódu 20: Použití direktivy bind. Zdroj autor

Direktivy style a class se používají k dynamickému stylování elementu. Umožňují přistupovat k datům uvnitř skriptu. Zápisů těchto direktiv existuje více. Jeden z nich je na následující ukázce.

```
<button
  class="btn"
  style={`background-color: ${color}`}
  on:click={onClick}>
  <slot />
</button>
```

Ukázka kódu 21: Použití direktivy style. Zdroj autor

6 Návrh a implementace aplikace

Struktura a funkce aplikace je inspirována podobnou aplikací napsanou v Reactu. [30] Jelikož implementace backendu aplikace není předmětem této práce, koncové body serveru používané v aplikaci jsou vytvořené knihovnou json-server. Tato knihovna umožňuje simulovat běžné chování REST API bez nutnosti implementace serveru. K ukládání dat používá json-server soubor db.json, který se nachází v kořenové složce projektu. [31]

6.1 Popis aplikace

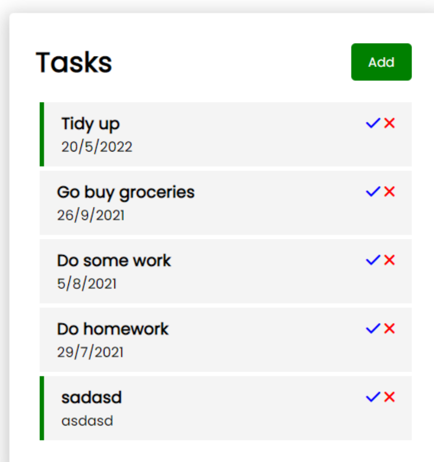
Implementovaná aplikace slouží jako jednoduchý sledovač úkolů. Aplikace má tři stránky:

- Home – domovská stránka
- Focus – stránka s úkolem, na který je uživatel zaměřen
- About – stránka s popisem aplikace

Aplikace je jednostránkovou aplikací, tudíž jsou jednotlivé stránky implementované za pomocí směrovače a nejedná se o separátní dokumenty. Každá z těchto stránek je uvedena v navigaci aplikace a má vlastní URL cestu.

Na domovské stránce se nachází list úkolů. Je zde možné úkoly mazat a přepínat u nich připomínky. Nachází se zde také formulář umožňující přidávat nové úkoly. Uživatel se může na jednotlivé úkoly zaměřit. Pokud je na nějaký úkol zaměřen, ukáže se mu na stránce Focus. Stránka About obsahuje článek s popisem aplikace.

Layout každé stránky je rozdělen na navigaci, tělo stránky a zápatí. Navigace se nachází v horní části aplikace. Pod navigací je tělo stránky, jehož obsah se mění podle stránky, na které se uživatel nachází. Zápatí je umístěné hned pod hlavním obsahem stránky.



Obrázek 2: Domovská stránka aplikace. Zdroj autor

6.2 Vývoj aplikace

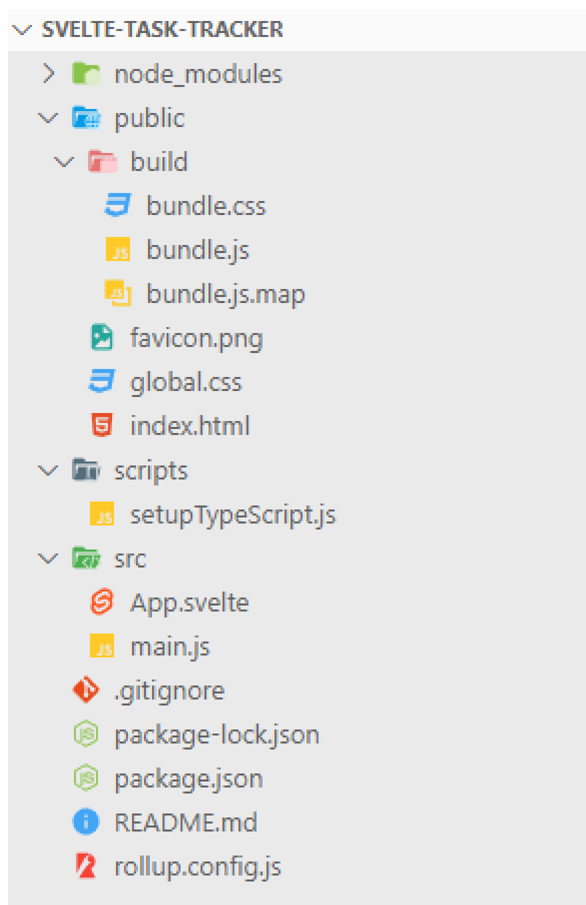
6.2.1 Založení projektu

K založení nového projektu slouží následující sekvence příkazů:

```
npx degit sveltejs/template svelte-task-tracker
cd svelte-task-tracker
npm install
```

Ukázka kódu 22: Sekvence příkazů pro inicializaci Svelte projektu. Zdroj autor

První příkaz použije knihovnu degit, kdy dojde k naklonování uvedeného repozitáře a uložení do složky pod zvoleným názvem. Druhý příkaz změní lokaci terminálu příkaz npm install nainstaluje potřebné balíčky uvedené v package.json. Struktura nově vytvořeného projektu vypadá následovně:



Obrázek 3: Struktura nově vytvořeného projektu. Zdroj autor

V projektu se nachází řada souborů a složek. Jejich názvy a významy jsou následující:

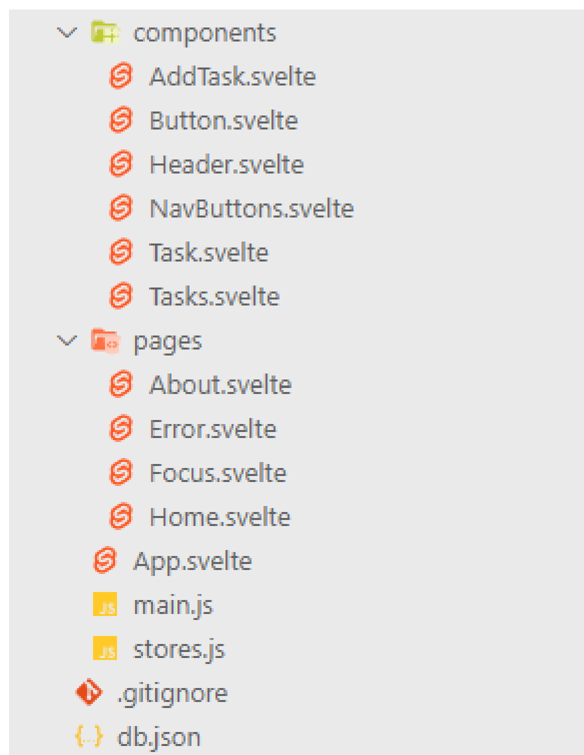
- node_modules – složka s potřebnými knihovnami projektu
- public – složka se statickými soubory, které nejsou zpracovány bundlerem
- build – složka se soubory vygenerovanými bundlerem
- favicon.png – ikona stránky pro prohlížeč
- global.css – globální CSS projektu
- index.html – základní HTML soubor, do kterého se umisťuje JavaScript aplikace
- scripts – složka s JS souborem sloužícím pro nastavení TypeScriptu
- src – hlavní složka obsahující strukturu aplikace ve vývojové fázi
- App.svelte – kořenový komponent, který je rodičem všech komponentů
- main.js – soubor určující cílové místo pro zobrazení JavaScriptu, který vygeneruje Svelte kompilátor

- package.json – JSON soubor obsahující informace o nainstalovaných balíčcích a příkazech relevantní k projektu.
- package-lock.json – automaticky generovaný soubor s odkazy ke stažení knihoven a jejich závislostí pro optimalizaci instalací
- .gitignore – soubor pro verzovací systém Git
- README.md – soubor obsahující dokumentaci projektu
- rollup.config.js – konfigurační soubor pro bundler Rollup

6.2.2 Struktura aplikace

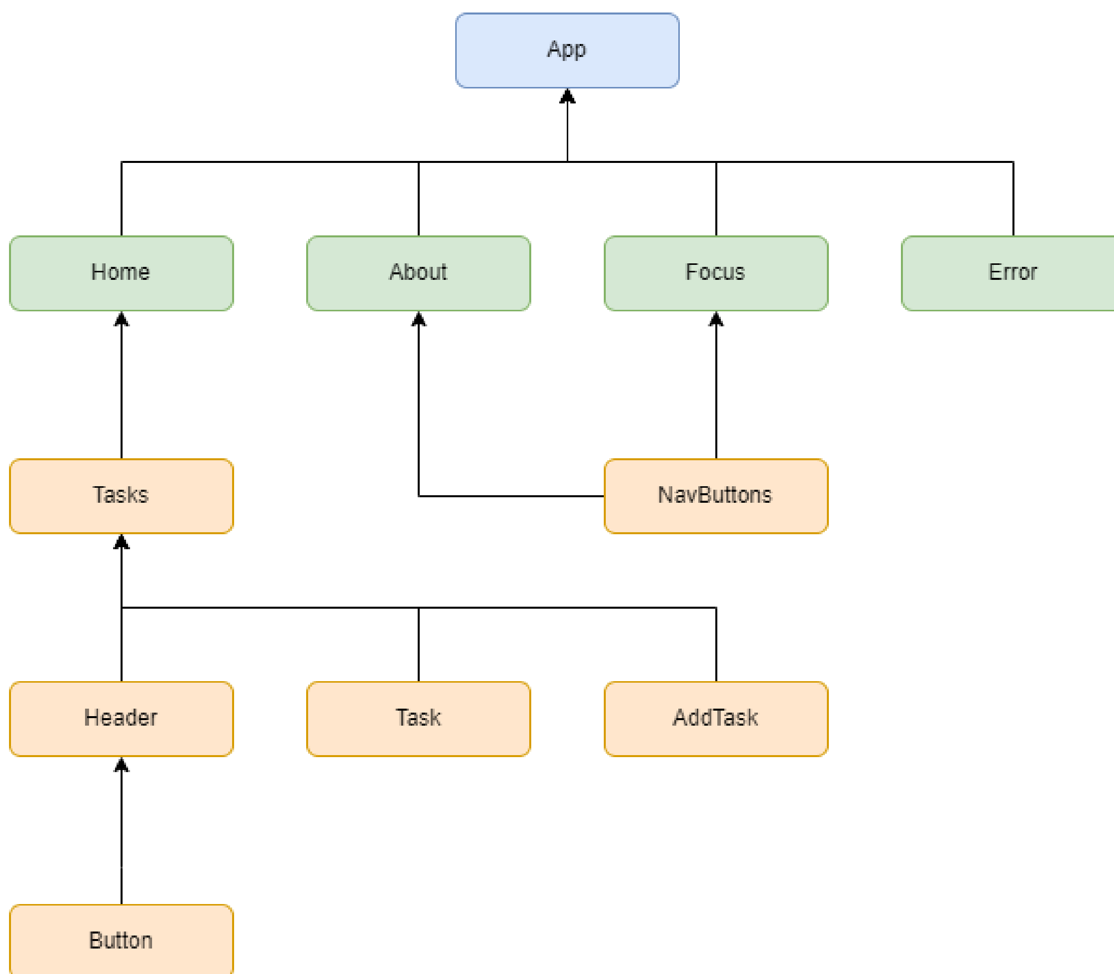
Struktura výchozího projektu není optimální, proto je zapotřebí jí upravit. Do projektu byly přidány následující složky a soubory:

- components – složka pro dílčí komponenty
- pages – složka s většími komponenty sloužící jako stránky
- stores.js – soubor pro globální stav
- db.json – JSON s uloženými úkoly



Obrázek 4: Upravená struktura aplikace. Zdroj autor

Aplikace je složená ze spousty zanořených komponentů. Hierarchii zanoření jednotlivých komponentů lze vidět na následujícím obrázku:



Obrázek 5: Hierarchie komponentů. Zdroj autor

Nachází se zde kořenový komponent App, který obsahuje stránkové komponenty Home, About, Focus a Error. Komponenty About a Focus obsahují komponent NavButtons. Jde zde jasně vidět znovupoužitelnost tohoto komponentu. Home dále obsahuje komponent Tasks, který využívá komponenty Header, Task a AddTask. Header používá komponent Button. Pokud nějaký komponent používá jiný komponent, je zapotřebí jej nejprve importovat. Skrze importování Svelte rozezná komponent, který lze následně použít v šabloně.

6.2.3 Layout a směrování

```
<script>
  import Router from 'svelte-spa-router';
  import Home from './pages/Home.svelte';
  import About from './pages/About.svelte';
  import Error from './pages/Error.svelte';
  import Focus from './pages/Focus.svelte';

  let routes = {
    '/': Home,
    '/about': About,
    '/about/:topic': About,
    '/focus': Focus,

    '*': Error,
  };
</script>

<main>
  <nav>
    <div class="logo">TrackerApp</div>
    <div class="separator">|</div>
    <a href="#/">Home</a>
    <a href="#/about">About</a>
    <a href="#/focus">Focus</a>
  </nav>
  <Router {routes} />
  <footer>© 2022 Lukáš Tesař</footer>
</main>
```

Ukázka kódu 23: Implementace App.svelte. Zdroj autor

Layout a směrování jsou obsaženy v kořenovém komponentu App, který lze vidět na ukázce výše. Router z knihovny svelte-spa-router zobrazuje komponenty, které jsou zrovna aktivní. Na elementy uvedené v šabloně mimo Router se směrování nevztahuje, čímž je docíleno stejného layoutu po celé aplikaci bez nutnosti kopírování navigace a zápatí do každého souboru. Pro Router je zapotřebí definovat objekt, ve kterém jsou obsaženy URL cesty a jejich komponenty. Tento objekt je posléze předán Routeru jako prop. Pokud dojde k přístupu na neexistující cestu, zobrazí se komponent Error, protože znaménko „*“ při deklaraci cest slouží jako označení kterékoliv cesty. Existující cesty tuto deklaraci samozřejmě přepisují. K ovládání, co má zrovna Router zobrazit, jsou použity běžné HTML elementy kotvy (anchor). Namísto běžného uvedení odkazu do atributu href je zde zapotřebí uvést

cestu definovanou v Routeru. Znaménko „#“ je v cestě obsaženo kvůli tomu, že se jedná o hashované směrování, kdy se při směrování kontroluje hash URL.

S tím, co má Router zrovna zobrazit, se dá pracovat i jinak. Jedním ze způsobů je použití funkcí push, pop a replace. Funkce push přesměruje na specifikovanou cestu, stejně jako kotva. Funkce pop slouží ke kroku zpět při navigaci. Funkce replace funguje obdobně jako push, akorát nedojde k vytvoření nové položky do historie prohlížeče. Tyto funkce lze přiřadit k libovolným eventům, jako například v komponentu NavButtons k implementaci ovládání navigace.

```
<script>
  import { push, pop, replace } from 'svelte-spa-router';
</script>

<div class="buttons">
  <button class="btn" on:click={() => pop()}>Back</button>
  <button class="btn" on:click={() => push('/')}>Go Home</button>
</div>
```

Ukázka kódu 24: Implementace komponentu NavButtons. Zdroj autor

6.2.4 Manipulace s úkoly

Hlavní logika aplikace je obsažena v komponentu Home**.** Je zde pole tasks, které je naplněno načtením ze serveru při zobrazení komponentu. Toto zajišťuje lifecycle funkce onMount, která zavolá funkci fetchTasks. Ta přistoupí na koncový bod serveru a vrátí pole úkolů. Pole tasks je předáno komponentu Tasks jako prop. Jelikož název prop je stejný jako název předávaného pole, Svelte umožňuje prop předat pouhým vložením názvu proměnné do složených závorek na komponentu bez nutnosti prop specifikovat.

```
let tasks = [];

onMount(async () => {
  tasks = await fetchTasks();
});

const fetchTasks = async () => {
  const res = await fetch('http://localhost:5001/tasks');
  const data = await res.json();

  return data;
};
```

Ukázka kódu 25: Inicializace pole úkolů při načtení komponentu Home. Zdroj autor

```
<Tasks
  {tasks}
  on:add-task={addTask}
  on:delete-task={deleteTask}
  on:toggle-reminder={toggleReminder}
/>
```

Ukázka kódu 26 Komponent Tasks s atributy. Zdroj autor

O přidání nového úkolu se stará funkce addTask. Funkce očekává parametr „e". Tento parametr je objekt, který má položku detail. V položce se nachází hodnota právě vytvořeného úkolu, která je uložena do proměnné newTask. Následně dojde k přistoupení na koncový bod serveru a předání nového úkolu. Po přijetí odpovědi ze serveru se příchozí data, což je v tuto chvíli nový úkol, uloží do pole tasks.

```
const addTask = async (e) => {
  const newTask = e.detail;

  const res = await fetch('http://localhost:5001/tasks', {
    method: 'POST',
    headers: {
      'Content-type': 'application/json',
    },
    body: JSON.stringify(newTask),
  });
  const data = await res.json();
  tasks = [...tasks, data];};
```

Ukázka kódu 27: Implementace funkce addTask. Zdroj autor

K mazání úkolu je použita funkce deleteTask. Pro přepínání připomínky úkolu slouží funkce toggleReminder. Obě tyto funkce fungují na podobném principu jako již

zmíněná funkce addTask. Každá z těchto tří funkcí je zavolána individuálním příchozím eventem komponentu Tasks (viz Ukázku kódu 26).

```
const deleteTask = async (e) => {
  const taskId = e.detail;

  await fetch(`http://localhost:5001/tasks/${taskId}`, {
    method: 'DELETE',
  });
  tasks = tasks.filter((task) => {
    return task.id !== taskId;
  });
};
const toggleReminder = async (e) => {
  const taskId = e.detail;
  const taskToToggle = await fetchTask(taskId);
  const toggledTask = {
     ...taskToToggle, reminder: !taskToToggle.reminder
  };

  const res = await fetch(`http://localhost:5001/tasks/${taskId}`, {
    method: 'PUT',
    headers: {
      'Content-type': 'application/json',
    },
    body: JSON.stringify(toggledTask),
  });

  const data = await res.json();

  tasks = tasks.map((task) => {
    return task.id !== taskId
      ? task
      : { ...task, reminder: data.reminder };
  });
};
```

Ukázka kódu 28: Implementace funkcí deleteTask a toggleReminder. Zdroj autor

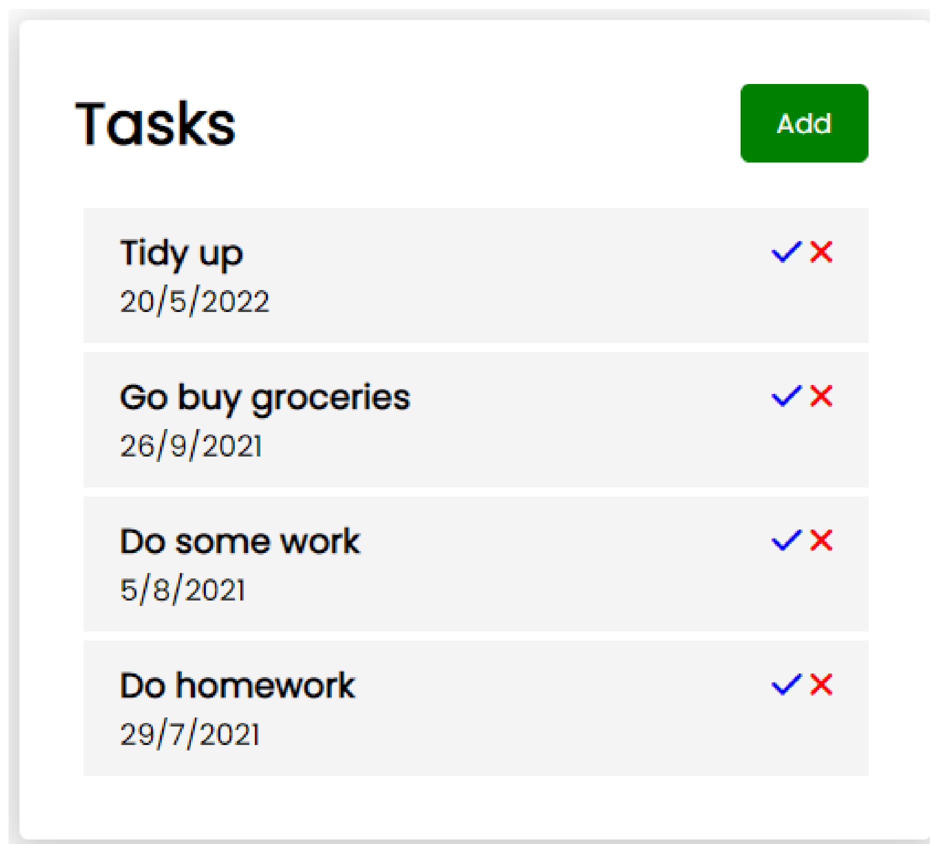
6.2.5 Renderování úkolů

Komponent Tasks definuje prop použitím klíčového slova „export“. Výchozí stav této prop je možné definovat přiřazením hodnoty. V tomto komponentu se odesílané eventy nevytváří. Jsou zde pouze předávány z nižších komponentů.

```
<script>
  import AddTask from './AddTask.svelte';
  import Header from './Header.svelte';
  import Task from './Task.svelte';

  export let tasks = [{ id: 1, text: 'hello' }];

  let showAddTask = false;

  const toggleShow = () => {
    showAddTask = !showAddTask;
  };
</script>

<div class="container">
  <Header {showAddTask} on:toggle-show={toggleShow}>Tasks</Header>
  {#if showAddTask}
    <AddTask on:add-task />
  {/if}
  {#if tasks.length > 0}
    {#each tasks as task (task.id)}
      <Task {task} on:delete-task on:toggle-reminder />
    {/each}
  {:else}
    <p>There are no tasks</p>
  {/if}
</div>
```

Ukázka kódu 29: Implementace komponentu Tasks. Zdroj autor

Použitím bloku each se iteruje přes poskytnuté pole úkolů. Iterace je obalená do bloku if, tudíž pokud pole obsahuje alespoň jeden úkol, k iteraci dojde a úkoly se zobrazí. Pokud však pole neobsahuje žádné úkoly, je zaktivován blok else a zobrazí se pouze paragraf s oznámením, že se žádné úkoly ve sledovači nenachází.



Obrázek 6: Seznam úkolů. Zdroj autor

6.2.6 Formuláře pro přidání úkolů.

Formulář pro vytváření nových úkolu je implementován v komponentu AddTask, který se nachází uvnitř komponentu Tasks (viz Ukázku kódu 29). Z kódu lze vidět, že ve výchozím stavu je komponent s formulářem schovaný. O přepínání se stará funkce toggleShow, která se zavolá v momentě příchodu eventu toggle-show. Event je odesílán z komponentu Header, jenž obsahuje tlačítko s funkcí, která odesílá event. Předáním showAddTask do prop komponentu Header jsou zprostředkovány dynamické styly.

```
<script>
  import { createEventDispatcher } from 'svelte';
  import Button from './Button.svelte';

  export let showAddTask = false;

  const dispatch = new createEventDispatcher();

  const handleToggle = () => {
    dispatch('toggle-show');
  };
</script>

<header>
  <h1><slot /></h1>
  <Button
    onClick={handleToggle}
    color={showAddTask ? 'red' : 'green'}
  >
    {showAddTask ? 'Close' : 'Add'}
  </Button>
</header>
```

Ukázka kódu 30: Implementace komponentu Header. Zdroj autor

Formulář má tři vstupy a každý je provázán přes direktivu bind. Při stisku tlačítka submit dojde ke kontrole vstupu pro text úkolu. Pokud uživatel nezadal text, funkce zde končí. Pokud uživatel text zadal, vytvoří se nový objekt s úkolem a odešle se jako parametr s eventem add-task.

```
<script>
  import { createEventDispatcher } from 'svelte';

  let text = '';
  let day = '';
  let reminder = false;

  const dispatch = new createEventDispatcher();
  const handleSubmit = () => {
    if (!text) {
      alert('Plase add a task');
      return;
    }

    const newTask = { text, day, reminder };
    dispatch('add-task', newTask);
  };
</script>

<form class="add-form" on:submit|preventDefault={handleSubmit}>
  <div class="form-control">
    <label for="">Task</label>
    <input type="text" placeholder="Add Task" bind:value={text} />
  </div>
  <div class="form-control">
    <label for="">Day & Time</label>
    <input type="text" placeholder="Add Day&Time" bind:value={day} />
  </div>
  <div class="form-control form-control-check">
    <label for="">Set Reminder</label>
    <input type="checkbox" checked={reminder} bind:value={reminder}/>
  </div>
  <div class="form-control">
    <input type="submit" class="btn btn-block" value="Save Task" />
  </div>
</form>
```

Ukázka kódu 31: Implementace komponentu AddTask. Zdroj autor

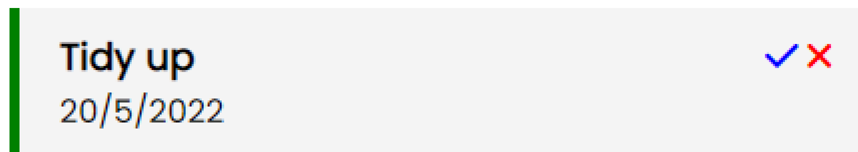
6.2.7 Odesílání eventů

Logika odesílání eventů delete-task a toggle-reminder se nachází v komponentu Task. K práci s eventy je zapotřebí vytvořit tzv. odesílatele (dispatcher). Tímto odesílatelem lze vytvořit event a předat mu odesílaná data. Odesílatel je volán přes korespondující funkci. Pro event delete-task je to funkce handleDelete, která je zavolána při kliknutí na ikonu křížku. V případě eventu toggle-reminder se jedná

o funkci handleToggle, která se zavolá v případě dvojitého kliknutí na celý úkol. Připomínka je v aplikaci znázorněna pomocí zeleného okraje na levé straně úkolu.

```
<script>
  import { createEventDispatcher } from 'svelte';
  import { TaskStore } from '../stores';

  export let task = {};
  const dispatch = new createEventDispatcher();

  const handleDelete = (taskId) => {
    dispatch('delete-task', taskId);
  };
  const handleToggle = (taskId) => {
    dispatch('toggle-reminder', taskId);
  };
  const handleFocus = (task) => {
    TaskStore.set(task);
    alert('Your focus has been set to this task');
  };
</script>

<div
  class={`task ${task.reminder ? 'reminder' : ''}`}
  on:dblclick={handleToggle(task.id)}
>
  <h3>
    {task.text}
    <div>
      <i
        class="fas fa-check"
        style="color: blue"
        on:click={handleFocus(task)}
      />
      <i
        class="fas fa-times"
        style="color: red"
        on:click={handleDelete(task.id)}
      />
    </div>
```

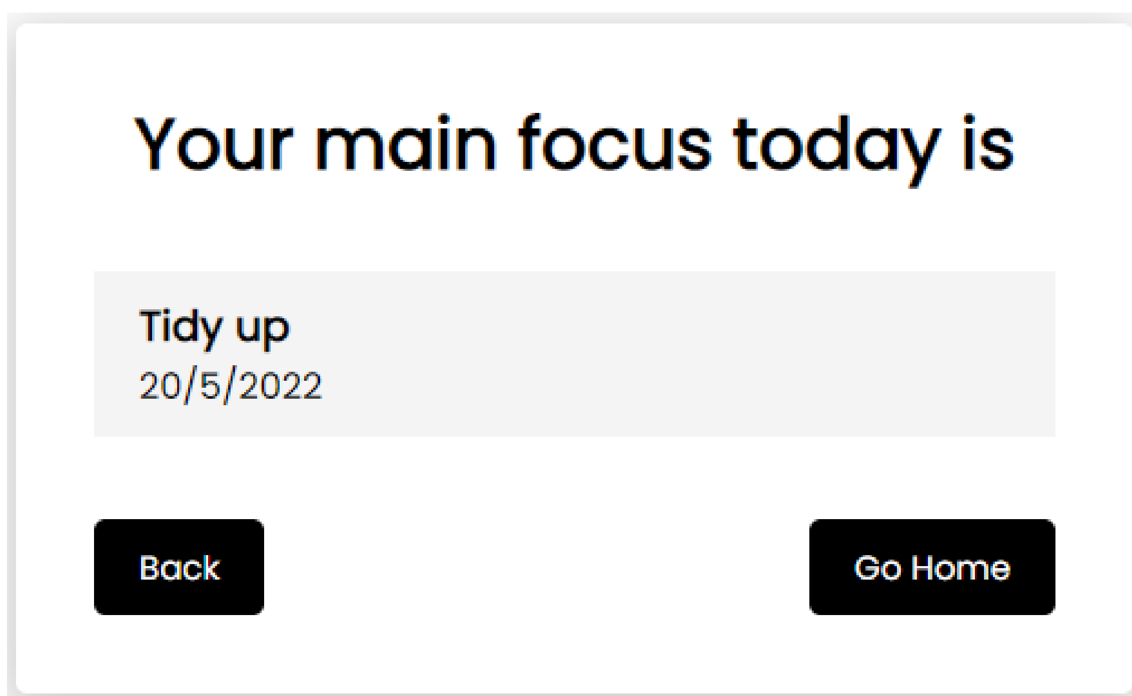
Ukázka kódu 32: Implementace komponentu Task. Zdroj autor



Obrázek 7: Úkol s připomínkou. Zdroj autor

6.2.8 Soustředění se na úkol

Aplikace umožňuje uživateli se na úkol soustředit (focus). V případě, že se na nějaký úkol uživatel soustředí, ukáže se úkol na stránce Focus. K tomuto je použit globální stav. Pro označení úkolu k soustředění slouží ikona zaškrtnutí. Při kliknutí na ikonu se zavolá funkce handleFocus (viz Ukázku kódu 32), která přistoupí ke globálnímu stavu TaskStore a přes funkci set nastaví předaný úkol v parametru do stavu. Následně se zavolá funkce alert, která zašle upozornění do prohlížeče.



Obrázek 8: Zobrazený úkol na stránce Focus. Zdroj autor

Globální stav TaskStore se nachází v souboru stores.js. Implementace globálního stavu se provádí přes funkci writable, která jako parametr přijímá výchozí hodnotu stavu.

```
import { writable } from 'svelte/store';

export const TaskStore = writable({
  id: undefined,
  text: undefined,
  day: undefined,
});
```

Ukázka kódu 33: Implementace globálního stavu TaskStore. Zdroj autor

Mimo funkci set, která slouží k modifikování stavu, lze také nad stavem volat funkci subscribe, jež slouží ke čtení hodnoty stavu. Funkce přijímá jako parametr další funkci, která přiřadí data ze stavu do lokální proměnné, s kterou je poté možné pracovat v šabloně. Čtení hodnoty globální stavu se nachází v komponentu Focus.

```
<script>
  import NavButtons from '../components/NavButtons.svelte';
  import { TaskStore } from '../stores';

  let task = {};

  TaskStore.subscribe((data) => (task = data));
</script>

<div class="container">
  <h1 class="text-center">Your main focus today is</h1>
  <div class="task">
    {#if task.text}
      <h3>{task.text}</h3>
      <p>{task.day}</p>
    {:else}
      <p class="text-center">There is no task you are focused on</p>
    {/if}
  </div>
  <NavButtons />
</div>
```

Ukázka kódu 34: Čtení hodnoty globálního stavu uvnitř komponentu. Zdroj autor

Svelte zároveň umožňuje číst data z globálního stavu alternativním a mnohem jednodušším přístupem s pomocí znaménka „$“. Při užití této notace lze se stavem manipulovat jako s běžnou lokální proměnnou.

```
<script>
  import NavButtons from '../components/NavButtons.svelte';
  import { TaskStore } from '../stores';
</script>

<div class="container">
  <h1 class="text-center">Your main focus today is</h1>
  <div class="task">
    {#if $TaskStore.text}
      <h3>{$TaskStore.text}</h3>
      <p>{$TaskStore.day}</p>
    {:else}
      <p class="text-center">There is no task you are focused on</p>
    {/if}
  </div>
  <NavButtons />
</div>
```

Ukázka kódu 35: Alternativní postup čtení hodnoty globálního stavu: Zdroj autor

6.2.9 Zacházení se styly

Jelikož se nejedná o rozsáhlou aplikaci, styly jsou zde řešeny především globálně v souboru global.css. Tento soubor je posléze vložen do linku uvnitř dokumentu index.html podobně, jako u běžných webových aplikací. Na několika místech se však definují styly lokálně uvnitř komponentu. Jedním z míst je komponent Focus, kde je potřeba aplikovat jiné styly než ty globální. V případě, že jsou některé styly definovány tímto způsobem, Svelte automaticky vytváří tzv. rozsah (scope). Ve své podstatě se jedná o izolaci definovaných stylů od okolí a aplikování pouze na komponent, ve kterém jsou definovány. K takto definovaným stylům Svelte navíc vygeneruje a připojí řetězec jako další selektor. Díky tomu je finální selektor stylu unikátní a hlavně konkrétnější, čímž získá větší prioritu a dojde k přepsání stylů s obecnějším selektorem, který má prioritou menší.

```
<style>
  .container {
    display: flex;
    flex-direction: column;
    justify-content: space-between;
  }

  .task {
    margin: 20px 5px;
  }
</style>
```

Ukázka kódu 36: Styly uvnitř komponentu Focus. Zdroj autor

Obrázek 9: Třída container s unikátním selektorem. Zdroj autor

7 Porovnání aplikací

V této kapitole jsou porovnány implementované aplikace Svelte s obdobnými aplikacemi implementovanými v knihovně React a frameworku Vue.js. Aplikace jsou z hlediska funkcí identické a během vývoje nedošlo k žádnému problému, který by některá z technologií nebyla schopna vyřešit.

Porovnání se zaměřuje především na části aplikací a koncepty, které byly pro účely porovnání uplatněny. Jedná se o části, kde jsou jasně vidět hlavní rozdíly, případně podobnosti mezi technologiemi. Jednotlivé aplikace slouží především k porovnání, tudíž implementace fungují spíše jako demonstrace, co jednotlivé technologie nabízí. Osvědčené postupy (best practices) samozřejmě nebyly zanedbány.

7.1 Velikost

U velikosti aplikace je důležitá především svazková velikost, protože v takové podobě je aplikace stahována webovým prohlížečem při jejím zobrazení, což má vliv na rychlost načítání. Menší velikost znamená rychlejší načítání aplikace, což je pro uživatele lepší. Velikosti aplikací jsou následující:

Technologie aplikace	Velikost repozitáře	Svazková velikost
Vue.js	94.4 MB	944 kB
React	269 MB	669 kB
Svelte	27.9 MB	158 kB

Tabulka 1: Velikosti repozitářů aplikací a jejich svazků. Zdroj autor

Z dat lze vidět, že nejmenší svazkovou velikost má Svelte aplikace, čímž se potvrzuje, efektivita Svelte kompilátoru. Dopad použití Rollupu oproti webpacku je minimální. [32] Na druhém místě je React aplikace a až na třetím Vue.js. Velikosti repozitářů slouží pouze jako uvedení do perspektivy o kolik menší výsledné svazky jsou. Velikost repozitářů ovlivňuje mnoho faktorů. Jedním z nich je obrovské množství závislostí, které projekty mají. Pokud nedojde k jejich použití, bundler je při kompilaci zahodí a na výslednou velikost svazku nemají vliv.

7.2 Struktura

Struktura je u aplikací vesměs stejná, protože všechny tři aplikace jsou rozděleny na stejný počet komponentů, které mají stejnou hierarchii (viz Obrázek 5). U Vue.js je navíc dedikovaný konfigurační soubor směrovače. Obě technologie fungují oproti Svelte o něco odlišněji v kontextu práce s globálním stavem, proto jsou u ovlivněných souborů drobné odlišnosti, avšak tomu se věnuje pozdější kapitola.

7.3 Layout a směrování

7.3.1 React

V případě Reactu jsou layout a směrování implementovány v kořenovém komponentu App stejně jako u Svelte. Knihovna použitá k implementaci směrování se nazývá react-router-dom. Hlavní rozdíl oproti Svelte je ten, že komponent Router zde obaluje celý layout včetně věcí jako navigace či zápatí, které se při navigování po aplikaci nemění. K indikaci, kde se mají požadované komponenty renderovat, slouží komponent Routes. Ten je zapotřebí umístit dovnitř Routeru. Do Routes je zapotřebí dát komponenty Route, které mají atributy path a element. Atribut path specifikuje URL cestu a skrze atribut element je daný komponent předáván. U navigace je zapotřebí používat komponenty Link, nikoliv klasické HTML kotvy. Namísto atributu href má Link atribut „to“, do kterého je vložena odkazovaná cesta. Směrování není hashované, tudíž cesty neobsahují znaménko „#“.

```
import { BrowserRouter as Router, Routes, Route, Link }
  from 'react-router-dom';
import React from 'react';
import { FocusTaskContextProvider } from './context/index';
import Home from './pages/Home';
import About from './pages/About';
import Error from './pages/Error';
import Focus from './pages/Focus';

const App = () => {
  return (
    <Router>
      <div className='app'>
        <nav>
          <div className='logo'>TrackerApp</div>
          <div className='separator'>|</div>
          <Link to='/'>Home</Link>
          <Link to='/about'>About</Link>
          <Link to='/focus'>Focus</Link>
        </nav>
        <Routes>
          <Route
            path='/'
            element={
              <FocusTaskContextProvider>
                <Home />
              </FocusTaskContextProvider>
            }
          />
          <Route path='/about/' element={<About />} />
          <Route path='/about/:topic' element={<About />} />
          <Route
            path='/focus/'
            element={
              <FocusTaskContextProvider>
                <Focus />
              </FocusTaskContextProvider>
            }
          />
          <Route path='*' element={<Error />} />
        </Routes>
        <footer>This is footer</footer>
      </div>
    </Router>
  );
};

export default App;
```

Ukázka kódu 37: Kořenový komponent App. Zdroj autor

7.3.2 Vue.js

U Vue.js je situace komplikovanější. Samotný kořenový komponent App je relativně jednoduchý jako u Svelte. Komponent, ve kterém se požadované komponenty zobrazují, se jmenuje router-view. To, co náleží mimo tento komponent, je při navigování po aplikaci neměnné. K vytváření odkazů pro navigaci se používá komponent router-link, kterému se určí cesta přes atribut „to".

```
<template>
  <div id="app">
    <nav>
      <div class="logo">TrackerApp</div>
      <div class="separator">|</div>
      <router-link to="/">Home</router-link>
      <router-link to="/about">About</router-link>
      <router-link to="/focus">Focus</router-link>
    </nav>
    <router-view />
    <footer>This is footer</footer>
  </div>
</template>
<style>
@import '../public/global.css';
</style>
```

Ukázka kódu 38: Kořenový komponent App. Zdroj autor

Komplexita nastává u definování cest. Ty jsou totiž definované ve vedlejším souboru index.js uvnitř adresáře router. Zde dochází k vytvoření pole objektů, které propojují jednotlivé cesty s odpovídajícími komponenty. Takto konfigurovaný router je posléze importován do main.js k vytvoření Vue instance.

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import Home from '../pages/Home.vue';
import Focus from '../pages/Focus.vue';
import Error from '../pages/Error.vue';
import About from '../pages/About.vue';

Vue.use(VueRouter);

const routes = [
  {
    path: '/',
    name: 'home',
    component: Home,
  },
  {
    path: '/about',
    name: 'about',
    component: About,
  },
  {
    path: '/focus',
    name: 'focus',
    component: Focus,
  },
  {
    path: '*',
    name: 'error',
    component: Error,
  },
];

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes,
});

export default router;
```

Ukázka kódu 39: Definování cest v souboru router/index.js. Zdroj autor

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'

Vue.config.productionTip = false

new Vue({
  router,
  store,
  render: h => h(App)
}).$mount('#app')
```

Ukázka kódu 40: Vytvoření Vue instance v souboru main.js. Zdroj autor

7.4 Práce s eventy

7.4.1 React

React nenabízí podobné vytváření vlastních eventů jako Svelte. Místo toho je tu zapotřebí uplatnit trochu jiný postup.

Stejně jako u Svelte aplikace je i zde v komponentu Home hlavní logika, kde se přistupuje k jednotlivým koncovým bodům a dochází zde k manipulaci se stavem úkolů. Vezměme v potaz například funkci deleteTask. Funkci je potřeba zavolat ve chvíli, kdy uživatel stiskne v aplikaci ikonu s křížkem. K tomu, aby byla funkce zavolána, musí být nejprve v komponentu s ikonou k dispozici. Tudíž je potřeba jí nejprve vložit jako prop komponentu Tasks, odkud jí je potřeba dále vložit jako prop komponentu Task, kde se funkce konečně nastaví do eventu onClick na ikoně.

```
const deleteTask = async (id) => {
  await fetch(`http://localhost:5000/tasks/${id}`, {
    method: 'DELETE'
  });

  setTasks(tasks.filter((task) => task.id !== id));
};
```

Ukázka kódu 41: Funkce deleteTask v komponentu Home. Zdroj autor

```
return (
  <>
    <Tasks
      tasks={tasks}
      onDelete={deleteTask}
      onToggle={toggleReminder}
      onAddTask={addTask}
    />
  </>
);
```

Ukázka kódu 42: Komponent Tasks uvnitř JSX výrazu komponentu Home. Zdroj autor

```
const Tasks = ({ tasks, onDelete, onToggle, onAddTask }) => {
  const [showAddTask, setShowAddTask] = useState(false);

  return (
    <div className='container'>
      <Header
        title='Tasks'
        onAdd={() => setShowAddTask(!showAddTask)}
        showAdd={showAddTask}
      />
      {showAddTask && <AddTask onAddTask={onAddTask} />}
      {tasks.length > 0 ? (
        tasks.map((task) => (
          <Task
            key={task.id}
            task={task}
            onDelete={onDelete}
            onToggle={onToggle}
          ></Task>
        ))
      ) : (
        <h3>There are no tasks</h3>
      )}
    </div>
  );
};
```

Ukázka kódu 43: Implementace komponentu Tasks. Zdroj autor

```
<i
  className='fas fa-times'
  onClick={() => onDelete(task.id)}
  style={{ color: 'red', cursor: 'pointer', marginLeft: '5px' }}
/>
```

Ukázka kódu 44: Ikona křížku v komponentu Task s metodou onDelete. Zdroj autor

7.4.2 Vue.js

Vue.js má velice podobné vytváření eventů jako Svelte. K jejich vytvoření slouží vestavěná metoda $emit, která se volá přímo nad instancí Vue za pomocí klíčového slova „this“. Funkci, která event odešle, je ve Vue.js zapotřebí definovat uvnitř vlastnosti methods. Ve Vue.js není optimální používat lambda funkce, protože by nebyl předán kontext Vue instance. Funkce je posléze volána na stisk ikony uvnitř komponentu Task, čímž je vyslán event do Tasks. Odtud je event dále odeslán do Home, kde dojde k zavolání přirazené funkce eventu.

```
<script>
export default {
  name: 'Task',
  props: {
    task: Object,
  },
  methods: {
    handleDelete(id) {
      this.$emit('delete-task', id);
    },
```

Ukázka kódu 45: Definování funkce handleDelete uvnitř komponentu Task. Zdroj autor

```
<i
  class="fas fa-times"
  style="color: red"
  @click="handleDelete(task.id)"
/>
```

Ukázka kódu 46: Přiřazení funkce handleDelete na event click ikony. Zdroj autor

```
<Task
  v-for="task in tasks"
  :key="task.id"
  @delete-task="$emit('delete-task', task.id)"
  @toggle-reminder="$emit('toggle-reminder', task.id)"
  :task="task"
/>
```

Ukázka kódu 47: Odeslání eventu uvnitř komponentu Tasks do komponentu Home. Zdroj autor

```
<Tasks
  @add-task="addTask"
  @delete-task="deleteTask"
  @toggle-reminder="toggleReminder"
  :tasks="tasks"
/>
```

Ukázka kódu 48: Přiřazení funkcí k příslušným eventům. Zdroj autor

```
async deleteTask(id) {
  await fetch(`http://localhost:5002/tasks/${id}`, {
    method: 'DELETE',
  });

  this.tasks = this.tasks.filter((task) => task.id !== id);
},
```

Ukázka kódu 49: Volaná funkce deleteTask. Zdroj autor

7.5 Globální stav

Globální stav je u aplikací použit pro soustředění se na úkol.

7.5.1 React

React nabízí řadu způsobů, jak sdílet stav napříč aplikací. Jedním ze způsobů je využití hooku useContext. Kontext je definován v adresáři context uvnitř souboru index.js. Společně se stavem, který je sdílen, je definován uvnitř komponentu FocusTaskContextProvider, který je posléze importován na potřebná místa a destrukturací jsou z něj „vytahovány“ potřebné části.

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

export const FocusTaskContext = createContext();

export const FocusTaskContextProvider = ({ children }) => {
  const [focusTask, setFocusTask] = useState({});
  return (
    <FocusTaskContext.Provider value={{ focusTask, setFocusTask }}>
      {children}
    </FocusTaskContext.Provider>
  );
};
```

Ukázka kódu 50: Obsah souboru index.js uvnitř adresáře context. Zdroj autor

```
<Routes>
  <Route
    path='/'
    element={
      <FocusTaskContextProvider>
        <Home />
      </FocusTaskContextProvider>
    }
  />
  <Route path='/about/' element={<About />} />
  <Route path='/about/:topic' element={<About />} />
  <Route
    path='/focus/'
    element={
      <FocusTaskContextProvider>
        <Focus />
      </FocusTaskContextProvider>
    }
  />
```

Ukázka kódu 51: Obalení komponentů uvnitř cest komponentem FocusTaskContextProvider. Zdroj autor

```
const Focus = () => {
  const { focusTask } = useContext(FocusTaskContext);

  const containerStyles = {
    display: 'flex',
    flexDirection: 'column',
    justifyContent: 'space-between',
  };

  return (
    <div className='container' style={containerStyles}>
      <h1 className='text-center'>Your main focus today is</h1>
      <div className='task' style={{ margin: '20px 5px' }}>
        {focusTask.text ? (
          <>
            <h3>{focusTask.text}</h3>
            <p>{focusTask.day}</p>
          </>
        ) : (
          <p className='text-center'>
            There is no task you are focused on
          </p>
        )}
      </div>
      <NavButtons />
    </div>
  );
};
```

Ukázka kódu 52: Vytažení stavu z poskytovatele kontextu a jeho čtení uvnitř komponentu Focus. Zdroj autor

```
const Task = ({ task, onDelete, onToggle }) => {
  const { setFocusTask } = useContext(FocusTaskContext);

  const onFocus = (task) => {
    alert('Your focus has been set to this task');
    setFocusTask(task);
  };
```

Ukázka kódu 53: Vytažení setteru stavu z poskytovatele kontextu a jeho použití uvnitř komponentu Task. Zdroj autor

7.5.2 Vue.js

U Vue.js aplikace je k implementaci globálního stavu použit tzv. Vuex. Vuex je knihovna poskytující vzor řízení stavu (state management pattern). Jedná se o vcelku komplexní zápis, který uplatňuje teorii, kdy dochází k definování tzv. akcí (actions) a mutací (mutations). Akce jsou funkce, které jsou definované uvnitř Vuex instance a jsou přístupné pro okolní komponenty. Při zavolání takové akce dojde ke spuštění mutace, která je také definována uvnitř Vuex instance, avšak není pro vnější komponenty přístupná. Mutace slouží jako jediný způsob, kterým lze stav uvnitř Vuex instance měnit. Díky tomu dojde k centralizaci změn stavu v aplikaci, díky čemuž se snadněji odchytávají potenciální bugy. [33, 34]

Vuex je inspirován React knihovnou Redux, která funguje velice podobně. Důvodem nepoužití Redux v React aplikaci je diverzifikace řešení sdílení stavu u aplikací. Options API u Vue.js nemá nic podobného jako React useContext.

```
Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    task: {},
  },
  getters: {
    GET_TASK(state) {
      return state.task;
    },
  },
  mutations: {
    SET_TASK(state, params) {
      state.task = params;
    },
  },
  actions: {
    setTask({ commit }, params) {
      commit('SET_TASK', params);
    },
  },
  modules: {},
});
```

Ukázka kódu 54: Definování Vuex instance uvnitř index.js v adresáři store. Zdroj autor

```
handleFocus(task) {
  this.$store.dispatch('setTask', task);
  alert('Your focus has been set to this task');
},
```

Ukázka kódu 55: Funkce handleFocus volající akci setTask uvnitř komponentu Task. Zdroj autor

Ke čtení stavu z Vuex instance je zapotřebí přistoupit ke getteru stavu. Getter je zapotřebí namapovat funkcí mapGetters uvnitř možnosti computed daného komponentu. Možnost computed dovoluje definovat položky, u kterých je nejprve zapotřebí provést nějakou logiku před použitím v šabloně.

```
computed: {
  ...mapGetters({ getTask: 'GET_TASK' }),
},
```

Ukázka kódu 56: Mapování getteru getTask uvnitř možnost computed komponentu Focus. Zdroj autor

```
<div v-if="getTask.text" class="task">
  <h3>{{ getTask.text }}</h3>
  <p>{{ getTask.day }}</p>
</div>
```

Ukázka kódu 57: Použití getteru getTask uvnitř šablony komponentu Focus. Zdroj autor

7.6 Styly

7.6.1 React

Stejně jako u Svelte, i zde jsou použity primárně globální styly. React nemá na rozdíl od Svelte oddělený tag pro styly. Je to kvůli tomu, že React je knihovna, která se zaměřuje především na manipulaci se stavem a renderování DOM. Kvůli tomu nemá nijak definovaný výchozí postup, kterým aplikace stylovat. Tento problém lze řešit mnohými možnostmi, které mohou zahrnovat použití knihoven třetích stran jako například Emotion. [35] V aplikaci je uplatněno definování objektu se styly a jeho následné vložení k elementu jako tzv. inline-styly.

```
const containerStyles = {
  display: 'flex',
  flexDirection: 'column',
  justifyContent: 'space-between',
};

return (
  <div className='container' style={containerStyles}>
    <h1 className='text-center'>Your main focus today is</h1>
    <div className='task' style={{ margin: '20px 5px' }}>
```

Ukázka kódu 58: Definovaný objekt se styly uvnitř komponentu Focus. Zdroj autor

7.6.2 Vue.js

Vue.js nabízí podobně provedené stylování jako Svelte. Rozdíl je u scope stylů, kdy ve výchozím stavu scope nemají, tudíž se definované styly komponentu vztahují i na obsahy komponentů uvnitř něj. Ke specifikování scope stylu zde slouží klíčové slovo „scoped“ uvnitř tagu style. Lze také specifikovat rozšíření CSS pomocí atributu lang. [36]

```
<style lang="css" scoped>
.container {
  display: flex;
  flex-direction: column;
  justify-content: space-between;
}

.task {
  margin: 20px 5px;
}
</style>
```

Ukázka kódu 59: Definování stylů uvnitř komponentu Focus. Zdroj autor

8 Vyhodnocení

V této kapitole dochází k využití poznatků nalezených v předchozích kapitolách pro vyhodnocení frameworku Svelte. Výsledkem vyhodnocení je zodpovězení otázky, zda je Svelte dobrou alternativou obdobných technologií pro vývoj SPA. Vyhodnocení je zaměřeno především na podobnosti a rozdíly s porovnávanými technologiemi, obtížnost implementace a efektivitu. V potaz je také brána škálovatelnost u větších aplikací.

8.1 Rozdíly a podobnosti

Na základě analýzy Reactu, Vue.js a Svelte (viz kapitola 5) můžeme vidět, že každá z technologií si je v něčem podobná a v něčem odlišná.

Hlavním rozdílem u Svelte oproti Reactu a Vue.js je užití Svelte kompilátoru k manipulaci s DOM aplikace namísto tradičního virtuálního DOM. To má za následek menší svazkovou velikost aplikace a kratší dobu počátečního načítání. [28] V kapitole 7.1 jsou velikosti jednotlivých implementovaných aplikací srovnány a Svelte tomto ohledu jasně vítězí.

V případě podobností je zjevné, že se Svelte inspiruje z velké části od Vue.js. Mají podobně řešené rozdělení kódu na šablonu, styl a script. V případě Svelte je toto řešené výrazně jednodušeji a čistěji. Není zde nutné specifikovat náležitosti jako například název komponentu nebo tag šablony pro HTML. Dále se u obou frameworků nachází direktivy, které fungují velice podobně. Vue.js direktivy v-if, v-else a v-for jsou ve Svelte pojaty jako bloky. Z funkčního hlediska se bloky od direktiv moc neliší. Ze syntaktického hlediska se liší v užití netradičních znamének u otevírací a uzavírací části bloků (viz kapitola 5.3.2).

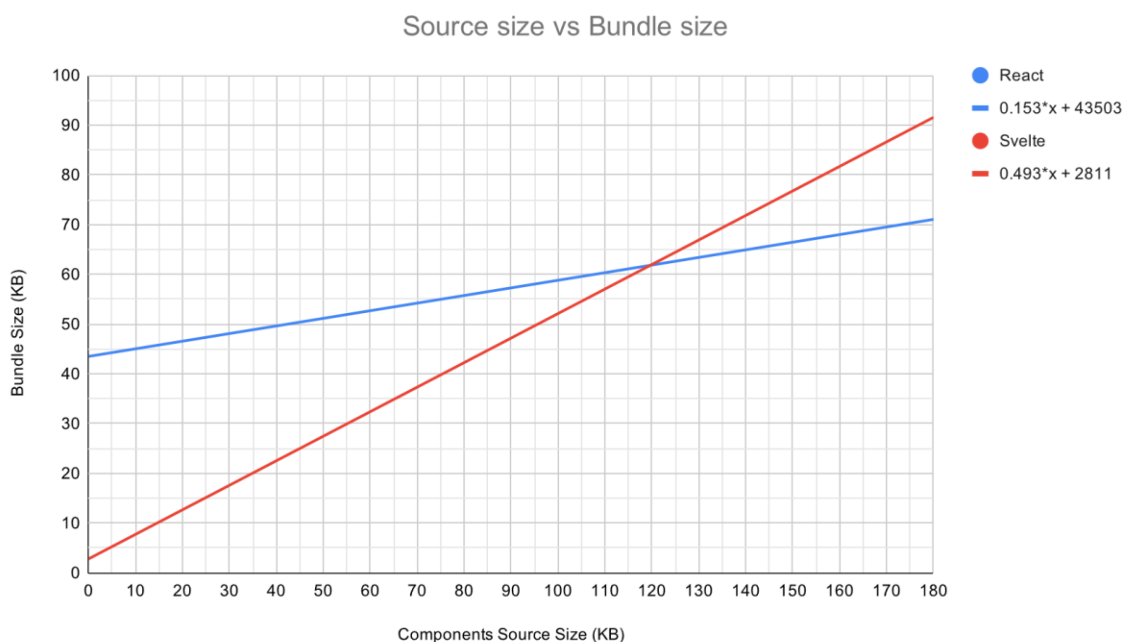
8.2 Použití při vývoji

Aplikaci se s pomocí Svelte podařilo implementovat bez jakýchkoliv obtíží. Na rozdíl od Vue.js a Reactu má Svelte velice jednoduchou a minimalistickou syntax. Vývoj aplikace byl díky tomu snazší a rychlejší. U Reactu a Vue.js je zapotřebí psát hodně standardního kódu (boilerplate code), čímž dochází k častému a zbytečnému opakování (viz kapitoly 7.3 a 7.5). Směrování je u Svelte kódově nenáročné

a přehledné. Největší rozdíl nastává u globálního stavu aplikace, který Svelte implementuje velice elegantně a efektivně, především v případě použití notace s pomocí znaménka „$“ (viz kapitola 6.2.8). Práce se styly a odesílání eventů jsou u Vue.js a Svelte velice podobné, avšak React z důvodu absence eventů a dedikovaných stylů v tomto aspektu pokulhává.

8.3 Škálovatelnost

Implementované aplikace ukázaly, že je Svelte dobrou volbou pro menší aplikace. Otázkou však je, jak si povede v případě větších aplikací. Tímto problémem se zabýval výzkum, kde byly porovnávané aplikace vyvinuté ve Svelte a v Reactu. Konkrétně se jedná o porovnání velikostí zdrojového kódu komponentů a výsledného svazku aplikace. Pomocí základní algebry byl nalezen inflexní bod, který ukazuje, že při velikosti zdrojového kódu komponentů do 120 kB Svelte produkuje menší velikost svazku než React. Od 120 kB zdrojového kódu komponentů však Svelte produkuje větší svazkovou velikost než React. Velikost 120 kB zdrojového kódu komponentů je však daleko od toho, co je často i u větších aplikací dosahováno. Běžné aplikace momentálně vyvíjené ve Svelte této velikosti nedosahují ani bez použití tzv. *code-splittingu*, kdy dochází k rozdělení svazku aplikace na menší části, které jsou dynamicky načítány podle požadavků. [37, 38]



Obrázek 10: Graf růstu svazkových velikostí Reactu a Svelte na základně velikosti zdrojového kódu komponentů. Zdroj [37]

8.4 Výsledek

Framework Svelte aplikuje zcela nový pohled na celkovou reaktivitu aplikací. Jeho jednoduchost a efektivita umožňuje rychlý vývoj aplikací se zárukou stability. Je ovšem důležité brát v potaz, že se jedná stále o vcelku novou technologii. Popularita Svelte od roku 2019 postupně roste a tím se zvětšuje podpora ze strany komunity, avšak jeho použití pro velké projekty je zatím malé. Svelte je tedy rozhodně dobrou alternativou pro tvorbu SPA oproti Reactu nebo Vue.js, nicméně v případě větších SPA je určitě na místě jeho výběr důkladně zvážit.

9 Shrnutí výsledků

Cílem práce bylo vyhodnotit Svelte v rámci vývoje jednostránkových aplikací. K tomu bylo zapotřebí nejprve provést analýzu technologií a principů, na kterých je vývoj jednostránkových aplikací založen. Zde došlo k popisu řady konceptů jako např. module bundler, komponenty, stav nebo směrování. Následně bylo Svelte analyzováno společně s knihovnou React a frameworkem Vue.js. Analýza jednotlivých technologií představila a popsala jejich hlavní vlastnosti. Tímto byl získán základní přehled o porovnávaných technologiích.

V praktické části došlo k implementacím aplikací. Nejprve byla implementována aplikace s použitím frameworku Svelte. Implementace byla podrobně popsána a s pomocí ukázek kódu přímo z aplikace bylo vysvětleno zacházení s prostředky, které Svelte nabízí. Poté došlo k porovnání aplikací implementovaných v Reactu a ve Vue.js. Aplikace byly z hlediska funkcí identické, díky čemuž bylo docíleno jasně viditelných rozdílů mezi implementacemi. Ačkoliv každá z technologii ve vývoji aplikace uspěla, z porovnání je zřejmé, že přístup Svelte je v mnoha případech kratší, čitelnější a šikovnější. Bylo také možné pozorovat, že Svelte je více podobné Vue.js než Reactu.

Nakonec byl framework Svelte úspěšně vyhodnocen, kdy se došlo k závěru, že Svelte je dobrou alternativou pro tvorbu SPA oproti Reactu nebo Vue.js, avšak jeho výběr je důležité zvážit při vyvíjení větších aplikací.

10 Závěry a doporučení

Framework Svelte se podařilo vyhodnotit jako dobrou volbu pro tvorbu jednostránkových aplikací. K vyhodnocení byly využity poznatky z analýzy, implementací aplikací a jejich porovnání. Porovnávané aplikace však nebyly zdaleka tak rozsáhlé a komplexní, jak běžně aplikace bývají. Je tedy otázkou, jak by si Svelte vedlo při vývoji větších aplikací. Zmiňované průzkumy a výzkumy v práci ukazují, že se dá předpokládat dobrá škálovatelnost tohoto frameworku. Ovšem tyto předpoklady jsou především teoretické a kvůli relativně krátké existenci Svelte se tato otázka momentálně nedá s jistotou zodpovědět.

11 Seznam použité literatury

[1] *The State of JS 2021: Front-end Frameworks* [online]. [vid. 2022-08-01]. Dostupné z: https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/

[2] *JavaScript | MDN* [online]. [vid. 2022-08-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[3] Tags · nodejs/node-v0.x-archive. *GitHub* [online]. [vid. 2022-08-02]. Dostupné z: https://github.com/nodejs/node-v0.x-archive

[4] Introduction to Node.js. *Introduction to Node.js* [online]. [vid. 2022-08-02]. Dostupné z: https://nodejs.dev/learn

[5] How Node.js Is Addressing the Challenge of Ryan Dahl's Deno. *The New Stack* [online]. 2. listopad 2020 [vid. 2022-08-02]. Dostupné z: https://thenewstack.io/how-node-js-is-addressing-the-challenge-of-ryan-dahls-deno/

[6] *Bun is a fast all-in-one JavaScript runtime* [online]. [vid. 2022-08-02]. Dostupné z: https://bun.sh/

[7] Let's learn how module bundlers work and then write one ourselves. *freeCodeCamp.org* [online]. 23. červenec 2018 [vid. 2022-08-02]. Dostupné z: https://www.freecodecamp.org/news/lets-learn-how-module-bundlers-work-and-then-write-one-ourselves-b2e3fe6c88ae/

[8] The Difference Between Minification and Gzipping. *CSS-Tricks* [online]. 27. červenec 2015 [vid. 2021-07-12]. Dostupné z: https://css-tricks.com/the-difference-between-minification-and-gzipping/

[9] 5 Best JavaScript Package Managers. *Dunebook* [online]. 5. únor 2020 [vid. 2021-07-12]. Dostupné z: https://www.dunebook.com/best-javascript-package-managers/

[10] Tags · npm/npm. *GitHub* [online]. [vid. 2022-08-02]. Dostupné z: https://github.com/npm/npm

[11] *About npm | npm Docs* [online]. [vid. 2021-07-12]. Dostupné z: https://docs.npmjs.com/about-npm

[12] *SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. [vid. 2022-08-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/SPA

[13] *React Props* [online]. [vid. 2022-08-15]. Dostupné z: https://www.w3schools.com/react/react_props.asp

[14] How to understand a component's lifecycle methods in ReactJS. *freeCodeCamp.org* [online]. 18. březen 2019 [vid. 2022-08-03]. Dostupné

z: https://www.freecodecamp.org/news/how-to-understand-a-components-lifecycle-methods-in-reactjs-e1a609840630/

[15] ESTEBAN. Routing in Javascript. *Medium* [online]. 5. srpen 2017 [vid. 2022-08-03]. Dostupné z: https://medium.com/@fro_g/routing-in-javascript-d552ff4d2921

[16] Stack Overflow Developer Survey 2021. *Stack Overflow* [online]. [vid. 2022-08-23]. Dostupné z: https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021

[17] FEDOSEJEV, Artemij. *React.js Essentials*. B.m.: Packt Publishing Ltd, 2015. ISBN 978-1-78217-462-2.

[18] ReactJS | Virtual DOM. *GeeksforGeeks* [online]. 5. duben 2019 [vid. 2022-08-04]. Dostupné z: https://www.geeksforgeeks.org/reactjs-virtual-dom/

[19] *Introducing JSX – React* [online]. [vid. 2022-08-04]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html

[20] *Introducing Hooks – React* [online]. [vid. 2022-08-04]. Dostupné z: https://reactjs.org/docs/hooks-intro.html

[21] *Hooks at a Glance – React* [online]. [vid. 2022-08-04]. Dostupné z: https://reactjs.org/docs/hooks-overview.html

[22] *Hooks API Reference – React* [online]. [vid. 2022-08-18]. Dostupné z: https://reactjs.org/docs/hooks-reference.html

[23] PRASAD, Madhusha. Vuejs history. *SLIIT FOSS Community* [online]. 25. prosinec 2021 [vid. 2022-08-06]. Dostupné z: https://medium.com/sliit-foss/vuejs-history-865eb1bba386

[24] *Vue 3 as the New Default | The Vue Point* [online]. [vid. 2022-08-06]. Dostupné z: https://blog.vuejs.org/posts/vue-3-as-the-new-default.html

[25] *Built-in Directives | Vue.js* [online]. [vid. 2022-08-06]. Dostupné z: https://vuejs.org/api/built-in-directives.html#v-for

[26] *API — Vue.js* [online]. [vid. 2022-08-24]. Dostupné z: https://v2.vuejs.org/v2/api/#Options-Data

[27] *Introduction / Basics • Svelte Tutorial* [online]. [vid. 2022-08-07]. Dostupné z: https://svelte.dev/tutorial/basics

[28] HARRIS, Rich. *Frameworks without the framework: why didn't we think of this sooner?* [online]. [vid. 2022-08-07]. Dostupné z: https://svelte.dev/blog/frameworks-without-the-framework

[29] The Svelte compiler: How it works. *DEV Community* [online]. [vid. 2022-08-07]. Dostupné z: https://dev.to/joshnuss/svelte-compiler-under-the-hood-4j20

[30] *Docs • Svelte* [online]. [vid. 2022-08-24]. Dostupné z: https://svelte.dev/docs

[31] TRAVERSY, Brad. *React Crash Course 2021 (Task Tracker App)* [online]. JavaScript. 8. srpen 2022 [vid. 2022-08-08]. Dostupné z: https://github.com/bradtraversy/react-crash-2021

[32] TYPICODE. *JSON Server* [online]. JavaScript. 9. srpen 2022 [vid. 2022-08-09]. Dostupné z: https://github.com/typicode/json-server

[33] SAJJAD, Zain. Benchmarking bundlers 2020: Rollup vs. Parcel vs. webpack. *LogRocket Blog* [online]. 7. říjen 2020 [vid. 2022-08-13]. Dostupné z: https://blog.logrocket.com/benchmarking-bundlers-2020-rollup-parcel-webpack/

[34] *What is Vuex? | Vuex* [online]. [vid. 2022-08-16]. Dostupné z: https://vuex.vuejs.org/#what-is-a-state-management-pattern

[35] *Core Concepts | Redux* [online]. [vid. 2022-08-16]. Dostupné z: https://redux.js.org/introduction/core-concepts

[36] *Styling and CSS – React* [online]. [vid. 2022-08-16]. Dostupné z: https://reactjs.org/docs/faq-styling.html

[37] *SFC CSS Features | Vue.js* [online]. [vid. 2022-08-18]. Dostupné z: https://vuejs.org/api/sfc-css-features.html#v-bind-in-css

[38] HALFNELSON. *Will it Scale? - Finding Svelte's Inflection Point* [online]. JavaScript. 3. srpen 2022 [vid. 2022-08-20]. Dostupné z: https://github.com/halfnelson/svelte-it-will-scale/blob/4df4c2af6ac22f10410c417415e48667b33577de/README.md

[39] *Code-Splitting – React* [online]. [vid. 2022-08-20]. Dostupné z: https://reactjs.org/docs/code-splitting.html

12 Přílohy

- Repozitář React aplikace: https://github.com/tesarlukas/svelte-task-tracker
- Repozitář Vue.js aplikace: https://github.com/tesarlukas/react-task-tracker
- Repozitář Svelte aplikace: https://github.com/tesarlukas/svelte-task-tracker
- Zdrojové kódy aplikací

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Lukáš Tesař**
Studium:	I1900263
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Experimentální vyhodnocení frameworku Svelte**
Název bakalářské práce AJ:	Experimental evaluation of Svelte framework

Cíl, metody, literatura, předpoklady:

Cílem bakalářské práce je analyzovat, vyhodnotit a porovnat framework Svelte v rámci vývoje jednostránkových aplikací.

1. Úvod
2. Cíl práce
3. Metodika zpracování
4. **Seznámení s technologiemi a principy**
5. **Analýza použitých frameworků**
6. **Návrh a implementace aplikací**
7. **Porovnání implementace a funkce aplikací**
8. **Vyhodnocení technologií**
9. Shrnutí výsledků
10. Závěr a doporučení

HARRIS, Rich. *Svelte 3: Rethinking reactivity* [online]. [vid. 2021-07-12]. Dostupné z: https://svelte.dev/blog/svelte-3-rethinking-reactivity

State of JS 2020: Front-end Frameworks [online]. [map]. nedatováno [vid. 2021-07-12]. Dostupné z: https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/

YORK, Richard. *Beginning JavaScript and CSS Development with jQuery*. B.m.: John Wiley & Sons, 2011. ISBN 978-1-118-05950-0.

Between the Wires interview with Evan You. | Between the Wires [online]. 3. červen 2017 [vid. 2021-07-12]. Dostupné z: https://web.archive.org/web/20170603052649/https://betweenthewires.org/2016/11/03/evan-you/

Zadávající pracoviště:	Katedra informačních technologií, Fakulta informatiky a managementu
Vedoucí práce:	Mgr. Daniela Ponce, Ph.D.
Oponent:	Ing. Martina Husáková, Ph.D.
Datum zadání závěrečné práce:	15.10.2021