F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Analýza state management v React

Lukáš Majer
Softwarové inženýrství a technologie

Květen 2023
Vedoucí práce: Ing. Jiří Šebek

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Majer** Jméno: **Lukáš** Osobní číslo: **478202**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Analýza state management v React

Název bakalářské práce anglicky:

Analysis of state management in React

Pokyny pro vypracování:

V Reactu jsou aplikace tvořeny pomocí pomocí struktury komponent, které si obstarávají svůj stav interně.
Cíle práce:
Seznamte se s problematikou vývoje front end aplikací a jejich přístupu řešení stavů. Vytvořte detailní rešerši jak problematiky tak technologií. Z dané rešerše vyberte vhodné knihovny pro Vámi definované řešení. Obsahem této práce bude návrh a implementace knihovny pro správu stavů v prostředí Javascript React. Zhodnoťte Vaši implementaci.

Seznam doporučené literatury:

[1] Publications and standards [online]. ECMA international, 2021 [cit. 2021-12-26]. Dostupné z: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/
[2] React state management [online]. Login Radius, 2021 [cit. 2021-12-24]. Dostupné z: https://www.loginradius.com/blog/async/react-state-management/
[3] React states and events [online]. Microsoft, 2021 [cit. 2021-12-24]. Dostupné z: https://docs.microsoft.com/cs-cz/learn/modules/react-states-events/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **11.02.2022** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **30.09.2023**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Poděkování / Prohlášení

Chtěl bych poděkovat Ing. Jiřímu Šebkovi za ochotu a velkou trpělivost při vedení této práce. Dále bych chtěl poděkovat své přítelkyni, rodině a přátelům, hlavně Jiřímu za podporu při celém studiu.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 26. 5. 2023

.....

Abstrakt / Abstract

Obsahem této práce je analýza a rozbor technologií a knihoven pro správu stavů ve frameworku React používaném v jazyce JavaScript pro dynamické webové aplikace. Z této analýzy jsou vybrány klíčové nástroje pro návrh a implementaci vlastních knihoven pro správu stavů. Výstupem práce jsou dvě knihovny, které jsou zaměřeny na zlepšení předávání stavů mezi komponentami a přenášení dat mezi různými aplikacemi. Tato knihovna je poté implementována a otestována na příkladu testovací aplikace.

Klíčová slova: správa stavu; webové aplikace; React; JavaScript.

The scope of this thesis is the research and analysis of technologies and libraries for state management in the React framework used in JavaScript for dynamic web applications. From this analysis, key tools for designing and implementing custom state management libraries are selected. The output of this work is a library that is aimed at improving state transfer between components and data transfer between different applications. This library is then implemented and tested on an example test application.

Keywords: state management; web applications; React; JavaScript.

Title translation: Analysis of state management in React

Obsah /

Ukázky kódů / Obrázky

Kapitola 1
Úvod

Když se v únoru roku 1992 Československo připojilo na internet z místnosti 256 na Fakultě elektrotechnické ČVUT, jen málokdo si dokázal představit, jakým směrem se internet vyvine za dalších 31 let [1]. Dnes už internet k našemu životu neodmyslitelně patří, a to nejen v Evropě, ale po celém světě. V červenci roku 2022 bylo na internet připojeno 63.1 % světové populace [2]. Každý den využíváme připojení k internetu pro komunikaci, práci, bankovnictví, sledování světového dění, nakupování, sdílení zážitků, sociální sítě a nespočet různorodých aktivit. S rozvojem internetových aktivit začaly vznikat internetové stránky prezentující a nabízející tyto služby. Popularita stránek začala rychle růst a s tím přišla potřeba rozvíjet grafické a uživatelské rozhraní a rozvíjet i schopnosti a možnosti použití internetových stránek pomocí webových aplikací.

Abychom se mohli bavit o webových stránkách a aplikacích, musíme se nejdříve podívat, jak takový Web funguje. Základem je značkovací jazyk Hypertext Markup Language (HTML), který zajišťuje především obsah a strukturu stránek, které jsou zobrazovány uživateli [3]. Dále je nutné získávat soubory, ve kterých je HTML zapsáno, aby se mohly zobrazit uživateli. Tyto soubory jsou uloženy na webových serverech, ze kterých je potřeba je získat. To zařizuje protokol Hypertext Transfer Protocol (HTTP) [3].

HTTP je založen na principu client-server, tedy uživatel (pomocí prohlížeče) pošle dotaz na server, a ten mu vrátí odpověď v podobě souboru, který se vykreslí. Celý tento proces je bezstavový, což znamená, že tato komunikace není ukládána a je okamžitě zapomenuta [3]. Toto má samozřejmě své výhody i nevýhody. Výhodou je především výkonnost. Na nevýhody narazíme zejména v případě, kdy potřebujeme dynamicky interagovat s uživatelem nebo při sdílení dat mezi různými stránkami.

Mít ale pouze strukturu stránky nestačí. Je třeba taky zařídit vzhled a rozložení stránek. Z tohoto důvodu vznikly Kaskádové styly značené jako CSS (Cascading Style Sheets). V dnešní době, kdy chceme navštěvovaný a použitelný web, je dáván důraz na kvalitní obrázky, barevnost a uživatelskou přívětivost [4].

Data nebo hodnoty, které se v průběhu životního cyklu aplikace mohou měnit, se nazývají stavy aplikace a jsou výsledkem všech akcí, které na uživatelské straně aplikace (frontend) probíhají. Jedná se tedy o ukládání vstupů a změn provedených uživatelem za běhu aplikace [5]. Se stavem webové aplikace se například setkáme v případě, kdy je naše aplikace přístupná pouze přihlášeným uživatelům. Je totiž uživatelsky žádoucí, aby autentizace proběhla jen jednou a následně byla tato informace k dispozici ostatním částem aplikace.

Stavy, respektive data, se na frontend udržují jen dočasně. Často je potřebujeme předávat mezi jinými aplikacemi nebo na serverovou stranu aplikace (backend). Pro předávání dat mezi jednotlivými aplikacemi se používá rozhraní pro programování aplikací (Application Programming Interface - API). Pravidla, protokoly a syntax přenášených dat dohromady určují API architekturu. V dnešní době se nejčastěji používají 4 architektury: REST, SOAP, GraphQL a RPC [6].

S rostoucí potřebou ukládání více stavů a větší dynamičností webu vznikl nástroj JavaScript (JS). JavaScript se v průběhu vývoje naučil upravovat a překreslovat HTML za běhu aplikace. Vznikaly pro něj průběžně různé knihovny a frameworky jako je React nebo Angular [7].

V Reactu jsou aplikace tvořeny pomocí struktury komponent, které si obstarávají svůj stav interně. To je přístup, který může fungovat pouze pro malé aplikace. Pro komplexnější aplikace se pak může při jedné akci měnit velké množství stavů komponent. Jejich následné sledování pak začne být velmi složité. Z tohoto důvodu je vhodné použít nějakou správu stavu aplikace [8].

V první části této práce budou popsány technologie, které souvisejí s tématem této práce. Dále budou rozebrány populární knihovny a frameworky JavaScriptu včetně frameworku React. Budou představeny techniky a postupy při vývoji. Další kapitola bude věnována nejpoužívanějším knihovnám pro správu stavu aplikace. Rozbor se zde bude zabývat knihovnami použitelnými pro React. Z těchto poznatků vyplynou požadavky, které by měla knihovna pro správu stavu v reactu splňovat. V následující kapitole bude představen návrh vlastní knihovny na základě těchto požadavků. V páté kapitole bude popis její implementace. Prezentovány zde budou nejdůležitější části kódu a popis toho, jak tato knihovna bude řešit správu stavu. V předposlední kapitole bude popsáno testování s důrazem na funkčnost a použitelnost. Poslední kapitola bude věnováná závěru a hodnocení celé práce.

Kapitola 2
Současné technologie

Vývoj webových frontend aplikací, především těch moderních, aby splňovaly veškeré požadavky uživatelů v co nejrychlejším čase, je v dnešní době poměrně komplexní záležitost. Programátor musí obsáhnout hned několik programovacích jazyků jako HTML, CSS a JavaScript. K těmto jazykům navíc existuje velmi velké množství frameworků, knihoven a pomocných nástrojů a služeb.

2.1 JavaScript

JavaScript je jeden z nejpoužívanějších jazyků pro tvorbu webových aplikací. Jeho nasazení napříč webem je přes 97 % u webových stránek používajících webové aplikace [9].

Samozřejmě se nejedná pouze o JavaScript jako takový, ale o všechny jeho verze s knihovnami či frameworky jako například Angular, JQuery a React. Dále je do toho počítán TypeScript, což není verze JavaScriptu, ale je do JavaScriptu překládán pomocí kompilátoru, protože prohlížeče jazyk TypeScript neumí interpretovat [10].

JavaScript je skriptovací jazyk, který vznikl v roce 1995. V současnosti je JavaScript implementací standardizovaného skriptovacího jazyka ECMAScript. V roce 2015 vyšla 6. verze tohoto jazyka od zastřešující společnosti ECMA International [11]. Tato verze přinesla spoustu nových vlastností, například byly zavedeny třídy a arrow funkce. Dále bylo upuštěno od značení proměnných pomocí klíčového slova *var*, které značilo jak konstanty, tak proměnné dohromady. Byla totiž zavedena klíčová slova *let* pro proměnné a *const* pro konstanty. Protože byly zavedeny takto razantní změny ve standardu jazyka, je tato verze považována za milník mezi starým a novým JavaScriptem. Od tohoto roku vychází nová verze pravidelně každý rok [7].

V dnešní době je kladen důraz na to, aby byl web dynamický, což znamená, že bude umět reagovat na nejrůznější události, a to především vstupy od uživatele. Tyto události samotné HTML není schopno zvládat, protože je statické. Z tohoto důvodu vznikla potřeba jazyka, který bude schopen na tyto události reagovat. JavaScript dnes umí mnohem víc než jen reagovat na vstupy od uživatele. Umí například přímo generovat HTML stránky a překreslovat je. Nicméně pořád je reakce na uživatelské události nejčastějším užitím.

JavaScript je programovací jazyk interpretovaný, neprochází tedy kompilátorem jako tomu je například u Javy nebo C++, ale je interpretován interpretem podobně jako Python. Interpret funguje tak, že čte jednotlivé instrukce kódu řádek po řádku a interpretuje ho. Javascriptový interpret je obsažen v naprosté většině moderních webových prohlížečů.

Kód JavaScriptu se běžně vkládá do HTML pomocí tagu *script*, a to buďto napřímo, anebo, což je častější varianta, je uložen v jiném souboru, který je pak v tomto tagu referencován pomocí odkazu na tento soubor. Výhoda externího souboru je zřejmá,

znovupoužitelnost, tedy možnost použití stejného kódu ve více webových aplikacích [12]. Pro tento jazyk, jak již bylo zmíněno výše, existují různé knihovny a frameworky, které zrychlují vývoj, dělají kód přehlednější nebo umožňují jednodušší testování.

2.2 TypeScript

TypeScript je programovací jazyk založený na JavaScriptu vyvinutý společností Microsoft. Jedná o rozšíření JavaScriptu především o statické typování, tedy na rozdíl od JavaScriptu mají proměnné předem známý datový typ jako je integer nebo boolean. Nalezneme zde prvky objektově orientovaných jazyků, jako jsou třídy a moduly. Je určený hlavně pro aplikace a programy velkých rozměrů, které potřebují tisíce řádků JavaScriptu. Informace o typech proměnných mohou být uloženy v samostatných hlavičkových souborech pro následnou integraci s jinými jazyky a knihovnami [10]. Typování je v některých směrech výhodné pro programátora, jelikož se s typováním lépe odchytávají chyby nebo pokud pracujeme v integrovaném vývojovém prostředí (IDE), může tento nástroj lépe napovídat při psaní kódu.

Velkou výhodou TypeScriptu je, že je kompilován do klasického JavaScriptu. Můžeme tedy psát aplikaci s jistotou, že bude kompatibilní všude tam, kde bude fungovat JavaScript. Jedná se tedy o většinu webových prohlížečů, viz kapitola 2.1.

Tento programovací jazyk má některé nativní frameworky jako je Angular, ale vzhledem k tomu, jak je blízký JavaScriptu, velmi dobře spolupracuje i se spoustou knihoven a frameworků původně pouze pro JavaScript jako je React, JQuery, nebo Vue.js.

2.3 Balíčkování

Pomocí balíčkování jde především snižovat počet dotazů na server, kdy se více javascriptových (někdy i jiných) souborů zabalí do jednoho souboru. Tím se dokáže snížit počet dotazů až na jeden jediný [13]. Poté se balíčky dají publikovat ve veřejných registrech, aby si je mohli vývojáři instalovat do svých aplikací.

V kontextu této práce je potřeba balíčkování ze dvou důvodů. Prvním důvodem je, že React jako takový je zabalen v již zmíněném balíčku. Za druhé chceme vytvořit novou knihovnu, která, aby byla pro uživatele co nejjednodušší na použití, by měla být taky zabalena v balíčku. Pro balíčkování jsou potřeba následující nástroje. Správce balíčků pro jejich správu, vyhledávání a instalaci, a dále generátor balíčku, který z našeho kódu vytvoří balíček a následně ho publikuje.

2.3.1 Node.js package manager

Node.js package manager (NPM) je správce balíčků pro JavaScript. Původně byl vytvořen pouze pro prostředí Node.js, ale v současnosti se využívá i pro JavaScript jako celek. NPM také umožňuje sdílet své balíčky nebo využívat veřejné balíčky, které sdílel kdokoliv jiný. Konfigurace toho, jaké balíčky budou v dané aplikaci používány, jsou obsaženy v souboru *package.json*. Taktéž umožňuje sdílet privátní balíčky, ale to mohou jen uživatelé s placeným účtem. Tento systém se skládá ze tří částí, které mohou uživatelé využívat [14]:

- Web - slouží pro správu účtu, hledání a využívání veřejných balíčků
- Command Line Interface (CLI) - hlavní způsob interakce s NPM, spouštění balíčků, jejich sdílení a hledání
- Registr - databáze napojená na NPM

2.3.2 Yarn

Yarn je správce balíčku pro JavaScript podobně jako NPM. Tento balíčkovací systém vznikl později a měl být v zásadě řešením problémů s NPM [15]. Především se jednalo o problém s rychlostí a částečně některé bezpečnostní problémy. Je to dáno tím že Yarn umí instalovat několik balíčku najdenou. Yarn stejně jako NPM umožňuje vytvářet a sdílet balíčky. Konfigurační soubor pro instalaci balíčků v aplikaci se jmenuje *package.json* stejně jako u NPM.

2.3.3 Rollup

Rollup je systém pro tvorbu balíčků. Je vhodný, pokud naše aplikace importuje jen málo knihoven třetích stran. Je zaměřen na ECMAScript modules, což je novější způsob balíčkování.

2.3.4 Webpack

Webpack je jedním z nejstarších systémů pro tvorbu balíčků. Je vhodný, pokud je naše aplikace velká, importuje velké množství knihoven třetích stran nebo potřebuje CommonJs, což je původní způsob balíčkování.

2.4 JQuery

JQuery je menší knihovna pro JavaScript, která je zaměřená na zjednodušení práce s dynamickým webem. Tato knihovna se zabývá především tím, že často opakující se operace, které bývají dost často složité a na velký počet řádku, se snaží zjednodušit. Pro zjednodušení práce využívá znalost selectorů, které jsou běžně využívané v CSS [16].

2.5 React

React je jedna z nejpoužívanějších knihoven pro JavaScript. Tato knihovna byla vytvořena společností Facebook. Hlavní rozdíl oproti JavaScriptu spočívá v tom, že se renderování HTML stránek dělá v objektech JavaScriptu. Tedy HTML notace je zapisována do JS dokumentů. React je vhodný pro Single-page aplikace (SPA), případně pro mobilní aplikace. Pro reakci na uživatele využívají překreslování jednotlivých objektů místo opakovaného načítání nových stránek, takže je stránka schopna rychle a efektivně reagovat na uživatele [17].

React je založen na deklarativním přístupu programování. Postup je takový, že se popisuje, co se má udělat, ale nepopisuje jak. Opakem k deklarativnímu přístupu v programování je imperativní programování [17].

2.5.1 Komponenta

Základním stavebním kamenem Reactu je komponenta. Komponenta může obsahovat různé funkce, ale především obsahuje část *render* viz ukázka kódu 2.1. Jednotlivé komponenty jsou do sebe zanořeny a předávají si mezi sebou data.

```
function Toy(args) {
  const { t, b } = args
  return (
   <div className="toy-container">
        <h3>{t}</h3>
         <div>{b}</div>
   </div>
  )
}

function App() {
  return (
   <div>
     <Toy title={t} body={b} />
   </div>
   );
}
```

Ukázka kódu 2.1. Základní vnoření komponent

2.5.2 React Document Object Model

React Document Object Model (DOM) je virtuální DOM, který aktualizuje HTML DOM, aby elementy v tomto DOMu odpovídaly React elementům. React elementy jsou získávány z komponent podobně jako ve Vue.js, viz kapitola 2.7.1. Díky tomu může být aplikace dynamická a jednostránková, respektive nemusí se načítat nové HTML dokumenty. V běžné React aplikaci jsou totiž prakticky všechny elementy generovány do stránky tímto způsobem, tedy nejsou tam žádné elementy statické [18].

2.5.3 Hooks

Hooks je poměrně nový koncept přidaný do Reactu s verzí 16.8. Tento koncept nám v zásadě umožňuje používat stavovou logiku nebo například generování elementů bez využívání třídních komponent [19]. Tyto vlastnosti jsou vyčleněny do jednotlivých funkcí, které se následně dají používat v jiných funkcích, viz ukázka kódu 2.2.

```
import React, { useState } from 'react';
function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Ukázka kódu 2.2. Nejčastěji používaným Hookem je useState, kterým můžeme používat stav ve funkci. Tento Hook dělá v podstatě to stejné jako setState ve třídních komponentách [20]

Výhody tohoto konceptu oproti předchozímu konceptu Reactu je především oddělení stavové logiky od tříd a následné opětovné použití ve více komponentách napříč celou aplikací. Dále rozložení kódu do menších funkčních komponent namísto velkých a komplikovaných třídních komponent umožňuje lepší porozumění kódu. Rozdělení do menších komponent také umožňuje snazší testování, jelikož testujeme menší celky.

2.6 Kombinace React s jiným frameworkem

Z kapitoly 2.5 víme, že React jako takový není kompletním frameworkem, ale pouhou knihovnou, která se tváří jako framework. Díky této vlastnosti se nabízí varianta kombinace Reactu s nějakým frameworkem, např. Angular.

Využití této varianty je z mého pohledu jen v krajních případech. Například pokud nějakou dobu existuje aplikace v jednom z použitých frameworků a je do ní potřeba přidat nový modul, který bude psán v druhém z nich. Myslím si, že ve většině případů se vyplatí jednu z těchto částí aplikace přepsat, aby měli jednotný framework a výsledná aplikace měla méně závislostí, než je kombinovat dohromady.

2.7 Vue.js

Vue.js je podobně jako React framework pro vývoj frontendových webových aplikací. Stejně jako React používá deklarativní přístup k programování a je založen na komponentách, upravuje DOM a renderuje do něj elementy. Tato vlastnost je vhodná pro Single-page aplikace, tedy aplikace, které využívají pouze jednu stránku místo několika. Nejčastěji se aplikace skládá z jednosouborových komponent. Podobně jako další frameworky, Vue.js zvládá pokrýt většinu úloh potřebných pro tvorbu frontend části webových aplikací [21]. Jelikož případy užití jsou napříč webem hodně různorodé, musí být Vue.js a jiné frameworky schopny držet krok s vývojem trendů a nároků uživatelů na web.

2.7.1 Komponenta

Jednosouborová komponenta (Single-file component - SFC) je tedy základním stavebním kamenem Vue.js. Soubory vypadají podobně jako HTML s tím rozdílem, že se jedná o HTML, CSS a JavaScript všechny dohromady v jednom souboru viz ukázka kódu 2.3.

```
<script>
export default {
  data() {
    return {
      count: 0
    }
  }
}
</script>
<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>
<style scoped>
button {
  font-weight: bold;
}
</style>
```

Ukázka kódu 2.3. Složení option stylu komponenty ve frameworku Vue.js [22]

Komponenta se tedy skládá z části *script*, kde se implementuje chování. Udržuje se tu také stav dané komponenty. Další částí komponenty je *template*, kde se předává HTML, které se má vykreslit. Stylování, tedy kaskádové styly, je uloženo pod tagem *style* v dané komponentě.

SFC se normálně vytváří ve dvou API stylech - *option* a *composition*. V ukázce kódu 2.3 je použit *option* API styl. Kdybychom tuto komponentu přepsali do *composition* stylu, vypadalo by jako v ukázce kódu 2.4.

```
<script setup>
import { ref, onMounted } from 'vue'

// reactive state
const count = ref(0)

// functions that mutate state and trigger updates
function increment() {
  count.value++
}

// lifecycle hooks
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

Ukázka kódu 2.4. Složení composition stylu komponenty ve frameworku Vue.js [22]

Pokud tyto styly porovnáme, tak *option* styl je více zaměřený na třídní programování, respektive na Objektově orientované programování (OOP), zatímco *composition* styl deklaruje stav přímo do funkcí, je tedy více reaktivní. Při porovnání s React je *option* styl velmi podobný třídám, zatímco *componstion* styl je spíše podobný využití Hooku viz kapitola 2.5.

2.8 Angular

Angular je další z velmi používaných frameworků pro JavaScript. Narozdíl od Reactu je více spjatý s TypeScriptem. Součástí Angular frameworku je balíček nástrojů pro kompletní vývoj aplikace. Základ celé aplikace tvoří Moduly. Moduly jsou kontejnery, které mohou obsahovat různé bloky kódu. Mohou to být services, komponenty nebo jiné další soubory. Komponenta je stavebním kamenem aplikace.

Angular je určený, podobně jako React, především pro single-page aplikace, jelikož taky využívá komponenty, které se následně renderují místo znovu načtení celé stránky. Proti Reactu je ale složitější na pochopení a naučení [23].

Narozdíl od Reactu se jedná o robustní framework, který má jasně daná pravidla tvoření webu [24]. Navíc má koncepty podobné spíše backendu, tedy je lepé pochopitelný pro programátora, který má už zkušenosti s jazyky Java nebo C#.

2.8.1 Komponenta

Každá komponenta v Angularu obsahuje CSS selektor, viz tag *@Component* v ukázce kódu 2.5. Tento selektor nám určuje upravovat instance jednotlivých objektů v HTML stránce. Například abychom se dostali ke komponentě použijeme

<hello-world></hello-world> tag. Dále obsahuje Template, který určuje jak se daná komponenta renderuje. Výsledné DOM vypadá jako v ukázce kódu 2.6.

```
import { Component } from '@angular/core';

@Component({
  selector: 'hello-world',
  template: `
    <h2>Hello World</h2>
    <p>This is my first component!</p>
  `
})
export class HelloWorldComponent {
  // The code in this class drives the component's behavior.
}
```

Ukázka kódu 2.5. Minimální komponenta ve frameworku Angular [23]

V části export class probíhá udržování stavu, a taky metod které se dají vyvolat například v template. Celkově se tedy jedná o chování a udržování dat souvicejích s danou komponentou.

```
<hello-world>
    <h2>Hello World</h2>
    <p>This is my first component!</p>
</hello-world>
```

Ukázka kódu 2.6. Vygenerovaná DOM ve frameworku Angular [23]

Kapitola 3
Správa stavů v Reactu

Všeobecně správa stavu je způsob, jakým jsou řešeny změny stavu za běhu aplikace. V nativním JavaScriptu probíhá změna stavu událostmi. Správa tedy v tomto případě je, jak se tyto události odchytávají, jak se na tyto události reaguje a kam se data z těchto událostí ukládají. V této kapitole budu rozebírat vybrané známé způsoby a knihovny pro správu stavů v Reactu.

3.1 Stavy a vlastnosti komponent

Stavy a vlastnosti komponent slouží k uchovávání informací o aktuálním stavu komponent, a tedy jejich následném vykreslování. Oba tyto objekty jsou jednoduchými JavaScript objekty a při změně dat v nich dochází k následnému překreslování výsledného HTML. Rozdíl mezi nimi spočívá v tom, že data do vlastností komponenty jsou předávány hlavně z rodičovských komponent, zatímco stavy jsou upravovány v rámci dané komponenty pomocí funkce setState [25].

3.2 Context

Běžně jsou vlastnosti komponenty předávány z rodičovské komponenty. Některé vlastnosti jsou ale potřeba ve velkém množství komponent naší aplikace, příkladem může být, zda je uživatel online nebo offline, případně zda je uživatel přihlášený. To může být nadbytečné pokud tato data potřebuje většina koncových komponent a v mezivrstvách se tyto data pouze předávají od kořenové komponenty a nijak se s nimi nepracuje. Context je tedy považován za globální proměnnou sdílenou mezi všemi komponentami bez nutnosti předávat je mezi všemi úrovněmi komponentového stromu [26].

3.3 Způsoby cachování v Reactu

Jedním z hlavních problémů, kterému se v Reactu čelí, je zbytečné překreslování komponent. Děje se to například, když musíme předávat data z jedné komponenty do druhé, která ale s těmito daty nepracuje, ale pouze je předává dál. I když tato data nemění danou komponentu, překreslí se. Toto zbytečné překreslování má vliv na rychlost aplikace.

Pro optimalizaci překreslování, a tedy zlepšení výkonnosti aplikace, se používá cachování. Cachování je ukládání dat z původního zdroje do RAM paměti, což vede k

rychlejšímu přístupu k datůmu, než kdybychom pokaždé museli získávat data z původního zdroje.

V reactu je několik způsobů, jak se dají data cachovat, a tak snižovat zbytečná překreslování, což vede ke zlepšení výkonnosti celé aplikace. Existují varianty cachování, kdy se použije metoda, například z React-Redux stavové knihovny. Dále je možné cachovat data za pomoci backendu, což může být v některých případech výhodnější. Například při dočasné nefunkčnosti frontendu máme některé důležité části uloženy v paměti backendu.

V případě, že nechceme používat další knihovny, případně cachovat pomocí backendu, můžeme použít následující metody [27].

3.3.1 UseMemo

UseMemo je React Hook, který ukládá do sebe již spočítané hodnoty. Nepřekresluje se dokud se vstupní hodnoty nezmění. Například pokud budeme mít funkci uvnitř komponenty, pak se tato funkce zavolá vždy, když bude překreslena komponenta. Pokud tuto funkci ale vložíme do tohoto Hooku, tak se tato funkce bude vykonávat jen tehdy, pokud se změní některá ze vstupních hodnot dané funkce [27].

3.3.2 UseCallback

UseCallback je velmi podobný Hook jako je useMemo. Rozdíl je v tom, jaká je návratová hodnota daného Hooku. Oba tyto Hooky berou funkci s nějakými parametry a oba se zavolají, jen tehdy pokud se změní jejich parametry. UseCallback vrací funkci s novými parametry, zatímco useMemo zavolá danou funkci a vrací spočítanou hodnotu [27].

3.3.3 Lazy Loading

Jedná se o princip, kde řekneme, kdy se má jaký obsah načítat a vykreslovat. Například, když třeba bude na stránce obrázek až dole, že se k němu uživatel dostane až když použije posuvník. V tento moment se dá na tuto komponentu s obrázkem použít lazy load, tedy obrázek se načte a vykreslí pouze tehdy, dokud k němu uživatel skutečně nedojde s použitím posuvníku [27].

3.3.4 UseRef

Další z možností Hook Reactu, který slouží pro ukládání dat nebo stavu, ale snižuje počet překreslování je funkce *useRef*. Funguje prakticky stejně jako useState, tedy základní state v Reactu, ale na rozdíl od něj je ideální v situacích, kdy dochází ke změně dat v něm uložených a nedochází překreslení stránky [27].

3.4 React knihovny pro správu stavu

Použijeme-li správu stavu pomocí metod uvedených v kapitolách 3.1 a 3.2, může to u některých aplikací vytvářet jisté problémy. Problém, kterému čelí stavy a vlastnosti je především předávání dat mezi komponentami, který ale dokážeme řešit použitím Contextu. Dalším problémem je, že při změně dat dojde k překreslení všech komponent, kterým jsou tato data předávána. V případě použití Contextu není obtížné předávat data

mezi vrstvami aplikace, ale bohužel se všechny tyto vrstvy překreslí. Toto je problém především u stavů nebo dat, které se často mění, a tedy dochází k častému překreslování elementů, které s tím prakticky nesouvisí. Tyto problémy mohou mít vliv na výkonnost aplikace, a taky na přehlednost a čistotu kódu jako takového. Tyto problémy se dají řešit pomocí knihoven. V následujících kapitolách rozeberu nejpoužívanější z nich [28]. Existující knihovny pro správu stavů se dají rozdělit na dva typy.

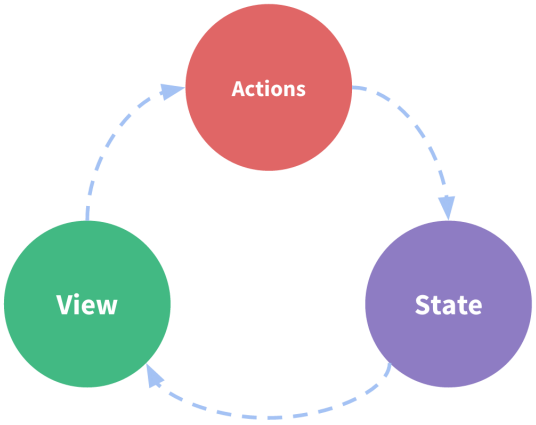
První typ je založený na globálním stavu, který je přístupný napříč celou aplikací podobně jako Context. Komponenty z tohoto stavu čtou anebo tomuto stavu upravují jen konkrétní části, které pro ně mají význam. Stav je tady uložen mimo stromovou strukturu komponent v react aplikaci.

Druhý typ jsou takzvané atomické knihovny, které jsou založeny na tom, že stavy jednotlivých komponent jsou umístěny v jednotlivých atomech. Atom je úložiště stavu, které je přístupné napříč aplikací stejně jako globální stav. Rozdíl je v tom, že každý atom je samostatný objekt. K těmto atomům se dá přistupovat pomocí takzvaných selektorů bez toho, aniž bychom museli procházet stromovou strukturu React aplikace [29].

3.4.1 Redux

Redux je jedna z nejčastěji používaných knihoven pro správu stavů. I přestože je s Reactem jako takovým často spojována, tak jsou propojeny pouze přes UI vrstvu a jsou na sobě nezávislé. Redux tedy může být použit, a dost často je využíván i jinými javascriptovými frameworky jako je Angular nebo Vue.js. Zde se ale budu zaměřovat na Redux pro React [30].

Redux je založen na konceptu toho, že existuje globální *stav* aplikace (State), tedy data využitelná napříč celou aplikací viz obrázek 3.1. Na části tohoto stavu jsou jednotlivé komponenty napojeny a vykreslují se podle něj. Stav se následně mění na základě událostí nazývaných *akce* (Actions). Akce jsou objekty popisující, co se stalo v aplikaci, jako jsou interakce od uživatele nebo vnitřní změna dat [30]. Na základě akcí se aplikace dostane do stavu *překreslení* (View) a všechny napojené komponenty se znovu vykreslí s aktuálními daty.



Obrázek 3.1. Schéma změny stavu aplikace při reakci na událost [31].

Jednou z důležitých částí Reduxu je immutabilita. Imutabilita znamená, že při změně se místo úpravy stávajícího stavu vytvoří nová kopie stavu se zanesenými změnami.

Použití Reduxu je vhodné a výhodné pokud aplikace má:

- Velké množství aplikačních stavů (globálních dat) používaných ve většině částí aplikace
- Aplikace je, co se týče kódu, hodně velká a komplikovaná
- Aplikační stav je neustále upravován [31]

Velkou výhodou Reduxu je jeho výkonnost. Běžně React při překreslování mění všechny komponenty v daném stromu komponent, takže jsou překreslovány také komponenty, pro které se data nijak neměnila. Komponenty se při použití Reduxu překreslují pouze tehdy, když se změní část stavu, na který jsou tyto komponenty napojeny, takže se překreslují pouze komponenty, pro které se data skutečně změnila [30].

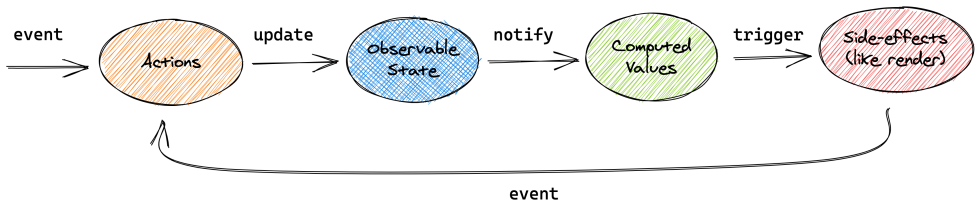
Nevýhodou je, že se programátor musí naučit hned několik konceptů. Také kód je delší, jelikož je potřeba popsat všechny akce a jejich poslouchání. Kód taky není tolik přímočarý. Z těchto důvodů není vhodný pro malé aplikace [31]. Další nevýhodou je, že obsahuje hodně boilerplate kódu, což jsou podobné, nebo stejné funkce na více místech v kódu. Každá úprava kódu musí být zanesena ve více souborech.

3.4.2 MobX

Další z často používaných knihoven pro správu stavů je MobX. Stejně jako Redux není závislá na Reactu a může být použita i pro jiné modifikace JavaScriptu, ale často je s Reactem spojována.

MobX je možné z pohledu návrhového vzoru považovat za pozorovatele. Každá komponenta má svůj stav nebo data. Aby mohl MobX fungovat správně musí být stav komponent nastaven na pozorovatelný. Následně je možné tyto komponenty trackovat a následně re-renderovat každou z nich.

Jak je vidět na obrázku 3.2, tak cyklus v MobX funguje následovně. Každá událost, například kliknutí myši, spustí akci (*action*), kterou může být reset časovače. Tento reset updatuje pozorovatelné stavy (*observable state*).Tento stav může být počet uběhnutých vteřin. Změny v pozorovatelných stavech jsou předány do všech výpočtů (*computed values*), tyto změny mohou spouštět vedlejší efekty (*side-effects*) jako například renderování. Změna v renderování vyvolá další událost a celý cyklus běží znovu [32].



Obrázek 3.2. Cyklus MobX knihovny [33]

Mezi výhodami MobX jsou jeho jednoduchost na naučení oproti Reduxu a méně boilerplate kódu. Nevýhodou je, že není tolik striktní jako jiné knihovny, což může vést k větší chybovosti, která vede k problematičtějšímu debugování.

3.4.3 Recoil

Recoil je nová knihovna pro správu stavu v Reactu. Narozdíl od výše zmiňovaných knihoven je tato knihovna spjatá s Reactem, tedy nelze tuto knihovnu použít bez Reactu. Byla vytvořena Facebookem stejně jako React. Pro osvojení této knihovny je

potřeba být seznámen s konceptem takzvaných Hooku v Reactu. Koncept Hooku byl přidán do Reactu s verzí 16.8, viz kapitola 2.5.3.

Narozdíl od předchozích knihoven umí řešit komplexní a složité problémy za použití minima boilerplate kódu. Výhodou je i kompatibilita s nejnovějšími funkcemi Reactu. Nevýhodou je svázanost s Reactem, tudíž ji nelze použít s jiným frameworkem nebo čistým JavaScriptem. Zatím se jedná o experimentální knihovnu, takže může mít nedostatky z hlediska stability a funkčnosti. Navíc obsahuje jen slabou dokumentaci, tudíž je složitější na použití.

```
const fontSizeState = atom({
  key: 'fontSizeState',
  default: 14,
});
```

Ukázka kódu 3.1. Stav atomu s unikátním klíčem a výchozí hodnotou [34]

Recoil používá jenom dva koncepty Atomy a Selektory [34]. Atomy jsou malé části stavu aplikace, které je možné upravovat, a taky na ně napojovat komponenty. Pokud se Atom upraví, tak se podle toho změní všechny napojené komponenty. Mohou také vznikat za běhu programu. Atom vždy obsahuje unikátní identifikátor a, stejně jako klasický stav, defaultní hodnotu, viz ukázka kódu 3.1. Pro přístup k Atomu se používá Hook *useRecoilState*, viz ukázka kódu 3.2. Je to velmi podobné jako *useState*, akorát se jako argument funkce používá Atom místo defaultní hodnoty.

```
function FontButton() {
  const [fontSize, setFontSize] = useRecoilState(fontSizeState);
  return (
    <button onClick={() => setFontSize((size) => size + 1)}
     style={{fontSize}}>
      Click to Enlarge
    </button>
  );
}
```

Ukázka kódu 3.2. Přístup k atomu pomocí funkce *useRecoilState* [34]

Selektor je čistá funkce, která na vstupu bere Atom nebo jiný selektor. Pokud se vstupní selektor nebo atom změní, je tato funkce znovu přepočítána. Stejně jako u atomů se můžou na selektory napojovat komponenty, a tedy se aktualizovat kdykoliv se selektor změní. Selektory se používají na výpočet dat ze stavu (atomů).

Jako se u atomu pro přístup používá funkce useRecoilState, tak se u selektorů používá useRecoilValue [34], viz ukázky kódů 3.3 a 3.4.

```
const fontSizeLabelState = selector({
  key: 'fontSizeLabelState',
  get: ({get}) => {
    const fontSize = get(fontSizeState);
    const unit = 'px';

    return `${fontSize}${unit}`;
  },
});
```

Ukázka kódu 3.3. Stav selektoru definovaného pomocí atomu [34]

```
function FontButton() {
  const [fontSize, setFontSize] = useRecoilState(fontSizeState);
  const fontSizeLabel = useRecoilValue(fontSizeLabelState);

  return (
    <>
      <div>Current font size: {fontSizeLabel}</div>
      <button onClick={() => setFontSize(fontSize + 1)}
       style={{fontSize}}>
        Click to Enlarge
      </button>
    </>
  );
}
```

Ukázka kódu 3.4. Přístup k selektoru pomocí funkce *useRecoilValue* [34]

3.4.4 Jotai

Jotai je další z nových knihoven pro správu stavu v Reactu. Je velmi podobná knihovně Recoil, protože je také založena na atomickém principu. Hlavní rozdíl mezi těmito knihovnami je, že v Jotai se Atom i selektor vytvářejí stejným způsobem. Jestli je daný konstrukt Atom nebo selektor rozlišuje pouze vstupní argument. Pokud je vstupnim argumentem funkce, jedná se o selektor, pokud ne jedná se o Atom [35], viz ukázka kódu 3.5.

Atom slouží k udržování stavu. Pro přístup k Atomu se používá funkce *useAtom*. Dále oproti Recoilu tu je objekt poskytovatel (Provider).

Provider je něco podobného jako Context u základního Reactu, tedy může dodávat stav pro jednotlivé React podstromy. Pokud Atom není v žádném podstromu spravovaném providerem, nastaví se stav na přednastavenou hodnotu Atomu. Používat provider je dobré především pro zjednodušení debuggingu.

Oproti Recoilu nabízí Jotai menší množství služeb a vlastností. Je tedy vhodnější pro menší projekty, které nemají na svojí implementaci velké a různorodé nároky [36]. Další výhodou může být, že narozdíl od recoilu není experimentální, takže je ověřená a lépe otestovaná. Nevýhodou je, že nelze jednoduše sdílet všechny atomy, respektive celý aplikační stav najednou.

```
import { Provider, atom, useAtom } from 'jotai'
const textAtom = atom('hello')
const Input = () => {
 const [text, setText] = useAtom(textAtom)
 return <input value={text} onChange={(e) => setText(e.target.value)} />}
const App = () => (
 <Provider>
   <Input />
 </Provider>
)
export default App
```

Ukázka kódu 3.5. Příklad definování a užití Atomu v knihovně Jotai [37]

3.4.5 Zustand

Podobně jako Jotai nebo Recoil je Zustand poměrně nová, malá a jednoduchá knihovna pro správu stavů v Reactu. Svým stylem je ale spíše podobná knihovně Redux. Podobně jako Redux využívá pro správu stavu jeden globální stav mimo strom komponent. Na jednotlivé části stavu se napojí jednotlivé komponenty, a ty zároveň upravují globální stav. Zároveň se komponenty updatují podle toho, zda byla část, na kterou jsou napojeny, upravena [38], viz ukázky kódů 3.6 a 3.7.

Výhodou použití Zustand knihovny je především v její jednoduchosti a velikosti, která neobsahuje tolik nadbytečného a opakovaného kódu. Na druhou stranu Zustand nemá moc velkou komunitu uživatelů a vývojářů, a navíc integrace s TypeScriptem není natolik přímočará jako u jiných knihoven [38].

```
import create from 'zustand'

const useStore = create((set) => ({
  bears: 0,
  increasePopulation: () => set((state) => ({ bears: state.bears + 1 })),
  removeAllBears: () => set({ bears: 0 }),
}))
```

Ukázka kódu 3.6. Vytvoření globálního stavu v knihovně Zustand [39]

```
function BearCounter() {
  const bears = useStore((state) => state.bears)
  return <h1>{bears} around here ...</h1>
}

function Controls() {
  const increasePopulation = useStore((state) =>
    state.increasePopulation)
  return <button onClick={increasePopulation}>one up</button>
}
```

Ukázka kódu 3.7. Napojení komponent na části globálního stavu v knihovně Zustand [39]

3.5 Analýza požadavků

Během vytváření jakéhokoliv softwaru je zapotřebí určit, co by mohl a měl umožňovat uživateli (v tomto případě programátorovi). Tyto umožnění jsou označovány jako funkční požadavky (FR). Dále je důležité určit jaké by měl mít software vlastnosti. Tyto vlastnostni jsou označovány jako nefunkční požadavky (NFR). V této práci se požadavky vztahují na navrženou knihovnu. Pro ověření, že byly požadavky naplněny, je zapotřebí vytvořit testovací aplikaci. Aplikace bude rozebrána v kapitole testovaní.

3.5.1 Funkční požadavky

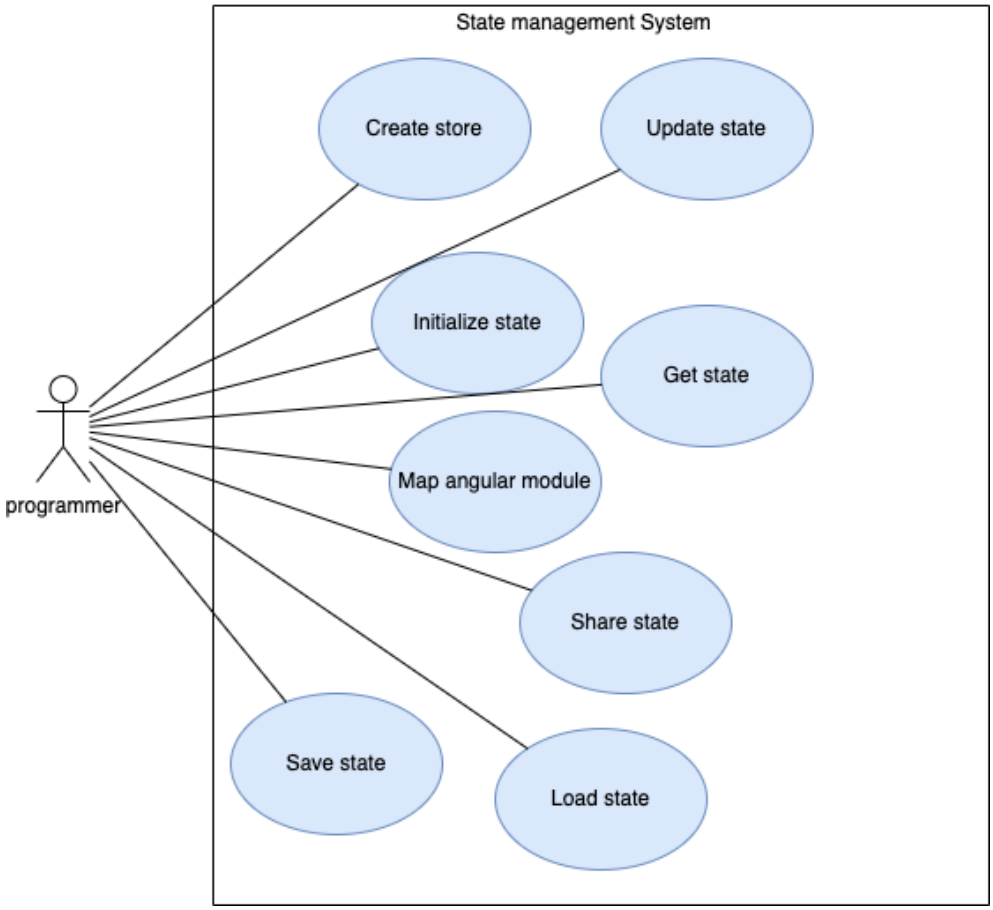
- FR1: knihovna správy stavu poskytne programátorovi mechanismus pro inicializaci stavu komponent React.
- FR2: knihovna by měla umožnit programátorovi měnit a aktualizovat stav React component dle potřeby a případně komponentu překreslit, pokud je to vyžadováno.
- FR3: Komponenty React by měly mít možnost přistupovat k aktuálnímu stavu a načítat jej ze systému správy stavu.
- FR4: Knihovna bude umožňovat, aby různé React komponenty sdílely a synchronizovaly svůj stav prostřednictvím systému správy stavu.
- FR5: Knihovna bude nějakým způsobem ukládat stav do cache (mezipaměť), aby se optimalizoval výkon, a aby se omezily zbytečné aktualizace stavu.
- FR6: Bude možné zachovat stav napříč relacemi nebo restarty, například pomocí ukladání do databáze.
- FR7: Knihovna umožní React komponentám aby mohly skládat svůj stav z více zdrojů.
- FR8: Komponenty React by měly mít možnost obnovit svůj stav na hodnoty, které byly uloženy v minulosti.
- FR9: Knihovna bude umožňovat získávání stavu z vložené komponenty psané v jiném frameworku.

3.5.2 Nefunkční požadavky

- NFR1: Knihovna pro správů stavu by měla mít minimální dopad na výkon aplikace.
- NFR2: Knihovna by měla být dobře škálovatelná, tedy měla by umět zpracovávat větší množství požadavků.
- NFR3: Knihovna by měla být snadno použitelná pro vývojáře.
- NFR4: Knihovna by měla být dobře integrovatelná do projektů vytvářených vývojářem.
- NFR5: Knihovna by měla být dobře testovatelná, například Unit testy.
- NFR6: Knihovna by měla být dobře dokumentovaná, aby bylo jasné, co která metoda či funkce dělá.
- NFR7: Aby se knihovna mohla dál rozvíjet, je dobré aby měla nějakou komunitu uživatelů, kteří budou podoporovat její rozvoj nebo radit s problémy, na které se v průběhu používaní narazí.

3.6 Případy užití

Přípday užití se vyjadřují pomocí diagramu případů užití. V diagramu bývají vždy uvedeni aktéři a jednotlivé případy užití. V případě stavové knihovny zde bude vystupovat pouze jeden aktér, a to programátor 3.3.



Obrázek 3.3. Diagram případů užití (Use case diagram)

Programátor vytvoří pomocí knihovy globální uložiště stavů (1). Akutalizuje stav uložený v uložišti (2). Inicializuje stav jednotlivých komponent z předem vytvořeného úložiště stavů (3). Získá hodnotu stavu podle klíče uloženého v úložišti (4). Načtě externí knihovnu pro Angular a předpřipravený Angular modul namapuje na React (5). Přenese stav mezi několika komponentami, a to nejen v rámci aplikace, ale i mezi aplikacemi (6). Uloží stav mimo aplikaci, aby bylo možné ho získat i po opětovném zapnutí aplikace, například po pádu dané aplikakce (7). Získá uložený stav mimo aplikaci a namapuje ho na komponentu (8).

3.7 Výběr technologií

V předchozích kapitolách 2 a 3 bylo popsáno, jak funguje frontend webových aplikací demonstrovaný na nejznámějších jazycích pro dynamický web. Rovněž jsou představeny klíčové vlastnosti nástrojů, jež jsou potřeba pro tvorbu těchto aplikací. Zejména bylo popsáno, co je to stav aplikace a komponent, a proč je potřeba jej spravovat. Dále byly rozebrány nejčastěji používané knihovny a jejich případné výhody a nevýhody. Pro další pokračování v práci byl jako základní nástroj, na kterém postavit knihovnu pro správu stavů, zvolen framework React z důvodů jeho velké škálovatelnosti a množství dostupných podpůrných nástrojů. Pro části knihovny, které budou potřebovat využívat i serverovou část aplikace, například pro ukládání stavu, se bude využívat Java pro její dlouhodobou existenci, vysokou podporu a robustnost pro vývoj serverových částí aplikace. Tento jazyk byl zvolen také z důvodu osobní zkušenosti s tímto jazykem. Jako nezbytnou součást byly balíčkovací nástroje, protože bez jejich používání je moderní vývoj klientských (frontend) aplikací skoro nemyslitelný. Jako balíčkovací systém byl v implementaci zvolen nástroj NPM s ohledem na jeho široké rozšíření a oblíbenost mezi uživateli.

Během analýzy knihoven a stavu v reactu byly také identifikovány požadavky na knihovnu pro správu stavu a její případy užití. Z těchto poznatků se následně vychází při návrhu vlastní knihovny, která se bude reflektovat tyto požadavky.

Kapitola 4
Návrh knihovny

V této kapitole je popsán návrh vlastní knihovny, která je rozdělena do několika částí podle funkcionalit. První část bude řešit především funkcionalitu uchovávání stavů v rámci jedné aplikace. Další část bude řešit funkcionalitu ukládání stavu do cache a do backendu a předávání stavu mezi aplikacemi. Poslední část bude řešit funkcionalitu předávání stavů mezi objekty ze dvou různých frameworků. Díky této funkcionalitě bude možné vkládat Angular komponenty do React projectu. Budou postupně představeny návrhy jednotlivých funkcionalit, které budou nejdříve rozebrány z pohledu propojení jednotlivých komponent s objekty uchovávající stav, a následně bude nastíněn reakční cyklus na změny stavu aplikace.

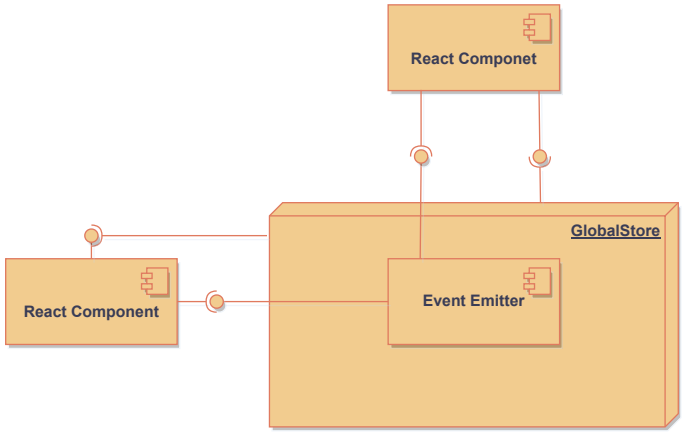
První část vlastní kihovny bude řešit problém přenášení stavů v rámci jedné aplikace pomocí globálního stavu. Jedná se o možnost jednoduše přistupovat ke stavům jednotlivých komponent napříč celou aplikací a zvýšení výkonnosti pomocí zamezení zbytečného překreslování. V návrhu druhé části knihovny bude řešen problém přenášení stavu mezi jednotlivými webovými aplikacemi a ukládáním stavu na backendové straně. To bude probíhat přes prostředníka, *proxy*. Proxy bude řešit kromě předávání dat také zrychlení získávání stavu, a to pomocí cachování dat. Následně se rozebere přenášení stavu mezi komponentami psanými v různých frameworcích Javasriptu. Těmito frameworky budou React a Angular

4.1 Správa stavů uvnitř jedné aplikace - Global store

Nejdříve bude popsán návrh první ze zmíněných částí knihovy, tedy nástroj pro správu stavů pouze uvnitř jedné aplikace. Stav je pomocí knihovny v aplikaci uchováván v konstruktu *global store*. To nám zajistí přímé navázání jednotlivých komponent na příslušné části stavu. Tím se při vyvolání změny stavu předají data jen vybraným komponentám, které se následně překreslí, a díky tomu zajistíme vyšší reaktivnost aplikace, protože se aktualizuje jen malá část oproti celé aplikaci. Rovněž bude tímto návrhem zajištěna vysoká kompaktnost této části knihovny.

4.1.1 Diagram komponent

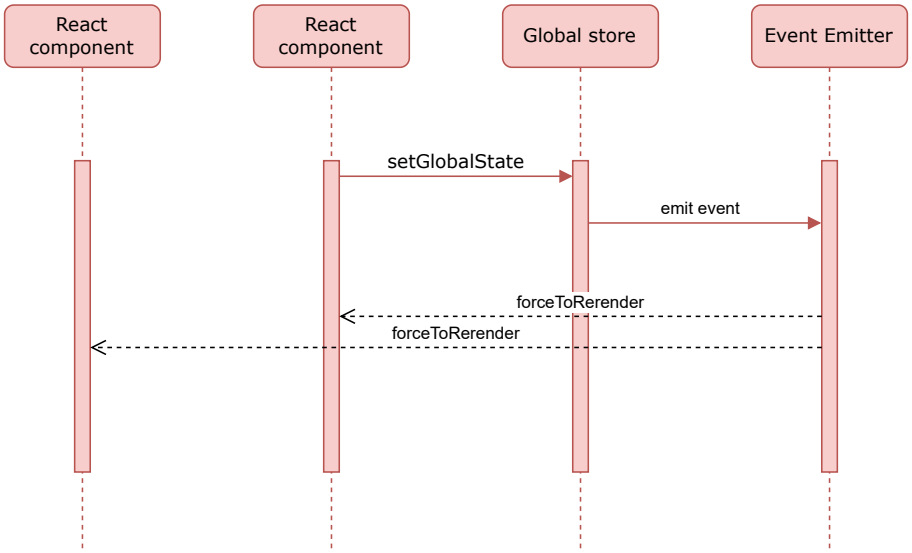
Pomocí diagramu na obrázku 4.1 si rozebereme napojení komponent v rámci jedné aplikace na globální stav. V tomto případě máme dvě komponenty, které jsou navázány na příslušné části globálního stavu. Jakmile nastane jakákoliv změna kterékoliv části globálního stavu, tak se upozorní pouze ty komponenty, které jsou na tyto části napojeny. O upozornění, že tyto změny nastaly, se stará *Event Emitter*. Poté, co jsou komponenty upozorněny, jsou překresleny.



Obrázek 4.1. Diagram napojení komponent na global store v jedné aplikaci

4.1.2 Sekvenční diagram

Sekvenční diagram znázorněný na obrázku 4.2 ukazuje jednotlivé kroky vyhodnocení události. Tedy jak se stav pomocí mé knihovny upravuje a přenáší na jednotlivé komponenty. Nejprve si komponenta zavolá funkci pro provedení úpravy na globálním stavu pomocí *setGlobalState*. Globální stav se následně změní a o této změně je notifikován *Event Emitter* pomocí vyvolání události (*emit event*). Ten se stará o propagaci změn a vyvolání překreslení všech navázaných React komponent pomocí volání *forceToRender*.



Obrázek 4.2. Diagram sekvenčního vyhodnocení události v jedné aplikaci

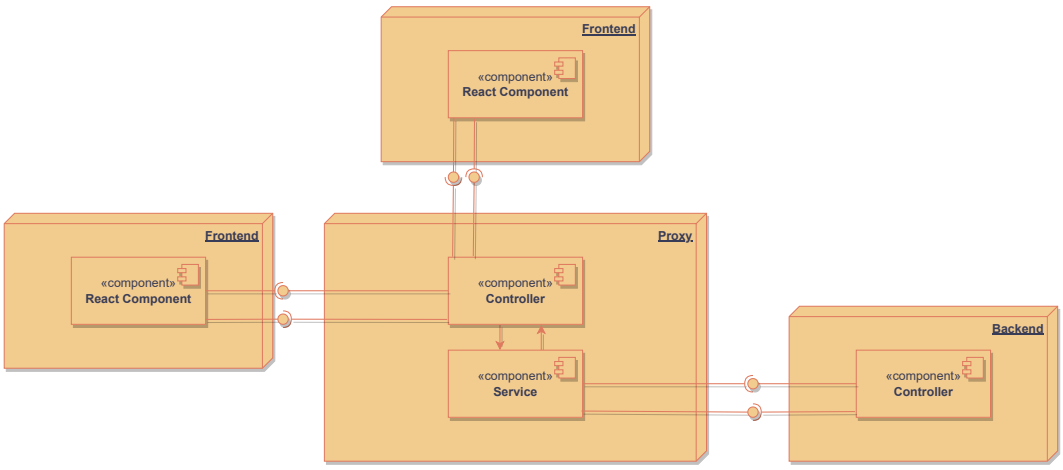
4.2 Správa stavů napříč aplikacemi - Proxy

Tato část knihovny se bude využívat ke správě a synchronizaci globálního stavu, který se ale bude nacházet mimo aplikaci. Stav je mimo aplikaci, protože k němu chceme přistupovat z více aplikací, takže není vhodné, aby byl součástí některé z aplikací. Dalším důvodem, proč mít stav uložen mimo běžící aplikaci, je možnost ho získávat i v případě restartu či pádu aplikace. Stav se tedy může nacházet jak na straně uživatele,

frontend, tak na straně webového serveru, *backend*. Navržená verze této funkcionality se bude nacházet mezi tímto stavem a aplikacemi, takže se ke stavu bude přistupovat pomocí interface prostředníka, *proxy*. Proxy bude v sobě obsahovat cache paměť. Tím bude díky snížení počtu dotazů zvýšena rychlost přístupu ke stavu na backendu.

4.2.1 Diagram komponent

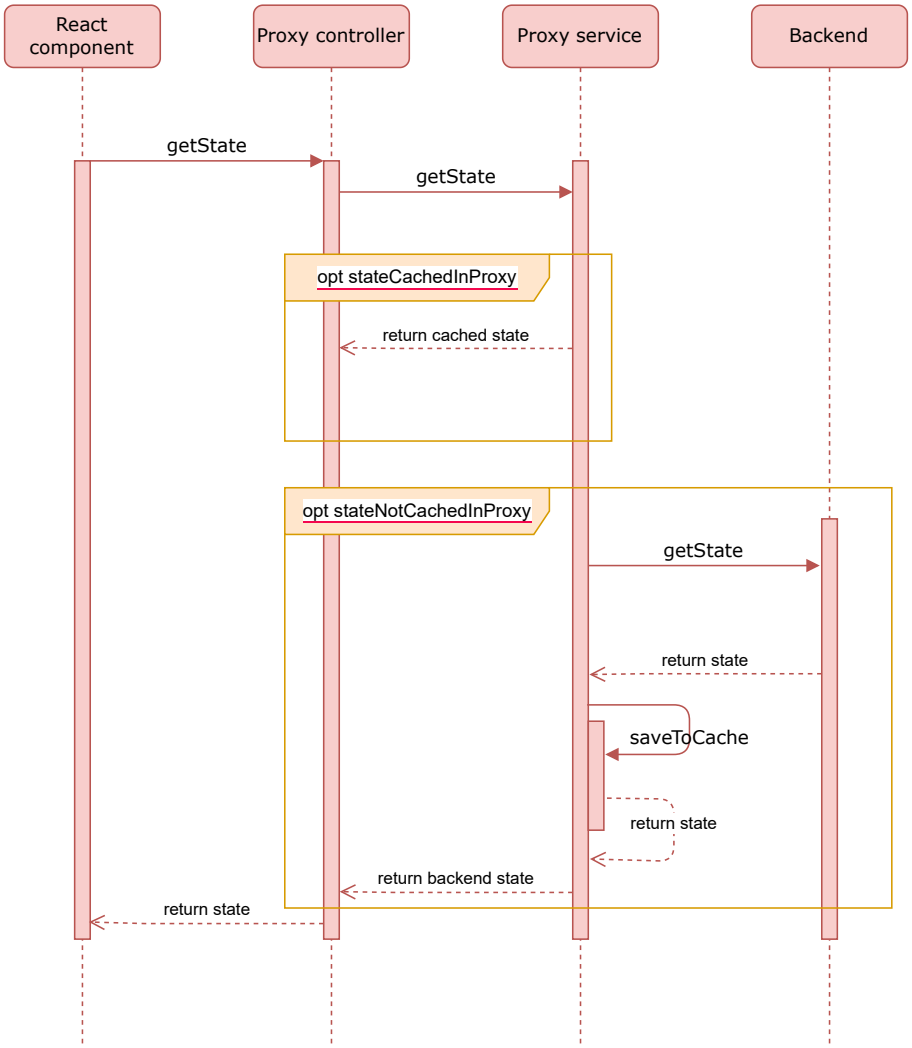
Jednotlivé komponenty spolu budou komunikovat pomocí REpresentational State Transfer (REST) interface [40], viz obrázek 4.3. Moduly označené jako *frontend* představují různé aplikace, se kterými interagují uživatelé. Například se jedná o *React komponenty*, které vykreslují elementy podle svého aktuálního stavu. V našem diagramu máme znázorněné dvě komponenty napojené na *controller* knihovny uvnitř *proxy*. Přes *controller* následně sdílejí svůj stav druhé aplikaci. Konstrukt *controller* figuruje jako interface, který umí komunikovat s *frontend* částí aplikace a dotaz převádí do jazyka knihovny. Logika knihovny se nachází v komponentě *service*. Knihovna je propojena s globálním stavem na *backend* části přes service, a to pomocí REST interface.



Obrázek 4.3. Diagram napojení komponent na sdílený stav mezi aplikacemi

4.2.2 Sekvenční diagram

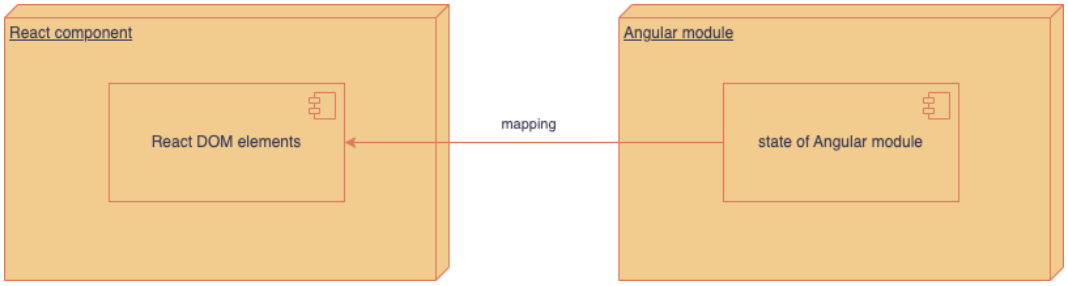
Chování knihovny při vyhodnocení události je schématicky znázorněno na obrázku 4.4. Je zde popsáno, jak probíhá komunikace mezi jednotlivými částmi knihovny a celé aplikace. Komponenta v jedné z aplikací, *React component*, si zažádá o aktuální stav pomocí HTTP požadavku skrze volání *getState*, který je předán pomocí REST API. *Proxy controller* knihovny následně upraví dotaz, aby odpovídal jazyku knihovny a předá ho na *proxy service*. Service zjistí, jestli má požadovaný stav uložený ve své interní paměti (cache), a pokud ho má, tak ho obratem vrátí. Znázorněno v alternativním fragmentu *stateCached*. Pokud ale stav v paměti není (alternativní fragment *stateNotCached*) proběhne komunikace s globálním úložištěm stavu, který se v našem případě nachází v *backend* části aplikace. Při vracení dat si service data uloží i k sobě do paměti a dále je poskytne *controlleru* pro potřeby dotazující komponenty.



Obrázek 4.4. Diagram sekvenčního vyhodnocení události v mezi aplikacemi

4.3 Stavy mezi frameworky

Touto částí knihovy budeme řešit přenos stavů mezi moduly a komponetami z frameworku AngularJS do React. Knihovna bude vytvářet mapování ze stavu modulu z frameworku Angularu do stavu React komponenty. Toto mapovaní bude fungovat tak, že kdykoliv se změní stav angulářího modulu, projeví se to na namapovaných elementech z react DOMu, který se v moment změny bude překreslovat. viz 4.5. V tomto DOMu jsou odkazy na stav angulářího modulu pomocí direktiv. Direktivy mohou odkazovat i na funkce, jež stav modulu upravují. Tedy je možné tatkto namapovaný stav upravovat, pokud budou v daném modulu existovat funkce, které mění stav a React element bude na tuto funkci namapován. Mapovaní v tomto návrhu funguje pouze jednosměrně, takže tuto funkcionalitu bude možná využít pro vkládání Angular modulů do Reactu, nikoliv obráceně.



Obrázek 4.5. Diagram mapování stavu z Angular modulu do React DOM elementů

Kapitola 5
Implementace

Na základě návrhu jednotlivých částí knihovny v předchozí kapitole 4 jsou představeny přístupy k řešení. Při vytváření nového softwaru je potřeba si vybrat integrované vývojové prostředí (IDE). Dá se vyvíjet pouze v textovém editoru, ale IDE má velké množství výhod, jako jsou integrace s jinými systémy, podpora verzovacích nástrojů, automatické doplňování textu a mnoho dalších. Těchto prostředí existuje velké množství, ale v tomto případě se vybíralo pouze ze dvou, Visual Studio (VS) Code a IntelliJ Idea, protože obě tato prostředí jsou robustní, všeobecně známé a mají vysokou podporu pro programování v Jave i Reactu. VS Code je dobré prostředí prakticky pro všechny různé druhy vývoje, jelikož umí stáhnout podporu a toolkity na velké množství programovacích jazyků. IntelliJ Idea je také velmi kvalitní prostředí, které ale má nižší rozmanitost programovacích jazyků. Je především určené pro vývoj v jazyce Java. Má podporu i pro JavaScriptové frameworky, jako je Angular nebo React. Z uvedených možností bylo použito IntelliJ, a to hlavně z důvodu větších osobních zkušeností s tímto prostředím.

5.1 Global store

Tato část je psána v Reactu. Pro instalaci frameworku bylo potřeba stáhnout si vybraný balíčkovací systém. V tomto případě byl použit NPM. Naimplementovaná knihovna byla zabalena a publikována veřejně pod názvem *lmajer-bp-state-library*[1]. K zabalení byl použit Webpack a pro publikaci opět NPM.

První část knihovny, tedy správa stavu komponent a její přístupnosti napříč aplikací, je psána za pomoci funkčních komponent a React Hooku. Pro tuto část bylo důležité vytvořit funkci, která bude generovat globální úložiště stavů. Zde se vycházelo především ze základního Hooku useState.

[1] `https://www.npmjs.com/package/lmajer-bp-state-library`

```
export const createGlobalStore = (initialState = {}) => {
   const EventEmitter = require('events');
   const eventEmitter = new EventEmitter();
   return (stateKey) => {
       const [, setCount] = useState(0);
   (. . .)
       useMemo(() => {
           eventEmitter.on(stateKey, () => {forceUpdate();});
       }, [stateKey]);
       return [
           initialState[stateKey], (setStoreState) => {
               initialState[stateKey] =
                    setStoreState(initialState[stateKey]);
               eventEmitter.emit(stateKey);
           }]}
```

Ukázka kódu 5.1. Výňatek z funkce vytvářející úložiště *Global store* se základním stavem *initialState*. Vrací dvojici hodnot, stav a *event* funkci

Funkce uvedená v ukázce kódu 5.1 při vytvoření bere základní stav, tedy objekt, kde jsou všechny stavy ve tvaru klíč-hodnota. Návratová hodnota funkce je množinou parametrů, které jsou velmi podobné jako u základního Hooku *useState*.

V návratové hodně je ještě přidána instance třídy *EventEmitter*, která umí registrovat posluchače a vyvolávat akce. V mé implementaci je použita k tomu, aby při změně některé části stavu došlo k překreslení všech komponent. *EventEmitter* je vytvořen pomocí konstruktu *node*, a to tak, že do funkce *require* je vložen modul *events*. Takto vytvořený *eventEmitter* je třída obsahující dvě metody - *emit* a *on*. Metoda *on* slouží k registraci posluchače a metoda *emit* slouží k vyvolání akce. Funkce *useMemo* slouží k uložení vnitřního stavu.

Hlavní výhodou knihovny je jednoduché sdílení stavů mezi komponentami napříč celou aplikací. Dále není potřeba psát velké množství kódu, aby byl tento stav sdílen, například v porovnání s knihovnou Redux, zde není prakticky žádný opakovaný kód. Na rozdíl od knihovny využívající proxy si komponenty nemusí žádat o aktualizaci stavu, která probíhá automaticky. Nevýhodou může být jednostranné zaměření funkcionality knihovny, která pouze sdílí stavy mezi komponentami v rámci jedné aplikace.

5.2 Proxy

Část knihovny, která se zabývá správou stavů napříč aplikacemi a jejich ukládání, je napsána v programovacím jazyce Java ve frameworku Spring. Tento framework je zvolen, protože je vhodný pro řešení komunikace mezi frontendem a backendem. Spring byl zvolen z důvodu osobních zkušeností s tímto frameworkem. Proxy funkcionalita knihovny se nachází na backendové straně aplikace. Pro sestavení a správu závislostí a konfigurace aplikace byl zvolen nástroj Maven.

Vstupní bodem je metoda, která nejprve přijímá HTTP Get dotaz z frontendu, v tomto případě z React aplikace, a to pomocí kontroleru, viz ukázka kódu 5.2. Ke kontroleru se dostává pomocí HTTP žádosti, kterou tento kontroler umí číst a odpovídat na ni. Kontroler předává data servisní vrstvě. Ze servisní vrstvy poté přichází odpověď

v podobě žádaných dat. Následně jsou zasílány zpět do React aplikace pomocí HTTP odpovědi.

```
@PostMapping(value = "/get-state",
consumes = MediaType.APPLICATION_JSON_VALUE,
produces = MediaType.APPLICATION_JSON_VALUE)
    public JSONObject getState(@RequestBody Key key) {
        String key1 = key.getKey();
        LOG.info("controller received" + key1);
        return service.getObject(key1);
    }
```

Ukázka kódu 5.2. Kontroler, který přijímá dotazy z frontendu a předává je na servisu. Po zpracování je odpověď vrácena na frontend

Servisní vrstva řeší logiku samotného předávání dat a jejich případné ukládání v interní paměti (cache), viz ukázka kódu 5.3. Pokud tedy přichází dotaz z kontroleru a servisní vrstva najde požadovaná data v cache paměti, načte je a odtud je vrací zpět do kontroleru. Data v cache mají také svou expirační dobu, takže pokud je při zpracování dotazu zjištěno, že je tato doba překročena, jsou data z cache smazána. Druhou variantou je, že data v cache nejsou nalezena. Pokud data nebyla nalezena nebo jsou expirovaná, tak se dotazuje backendu pomocí REST Template. Odpověď z backendu je uložena do cache a vrácena na kontroler. V případě požadavku na změnu stavu jsou data uložena jak na backend, tak do mezipaměti servisy.

```
  if (!objectMap.containsKey(key)) {
// TTL has expired, invoke another endpoint to store the object again
            String endpointUrl = "http://localhost:8081/backend/get";
            URI uri = UriComponentsBuilder.fromHttpUrl(endpointUrl)
                    .queryParam("key", key)
                    .build()
                    .toUri();

            JSONObject response =
                restTemplate.getForObject(uri, JSONObject.class);
            objectMap.put(key, response);
            return response;
        } else {
            return jsonObject;
        }
```

Ukázka kódu 5.3. Ukázka ze servisy, kde se zjišťuje, zda je stav uložen v cache, a následné volání backendu

Tato část knihovny je tedy především určena k přenášení dat mezi aplikacemi, a ne pouze mezi komponentami v rámci aplikace. Toto přenášení je navíc zrychleno cachovaním dat uvnitř této knihovny, takže nemusí všechny dotazy směřovat přímo na úložiště stavů. Další funkcionalitou je zálohování stavu jednotlivých kompnent. Teoreticky se dá využít i pro zrychlení přenášení jakýkoliv dat mezi aplikacemi, a ne pouze pro předávání stavu. Nevýhodu je, že jednotlivé komponenty si musí svůj stav sami aktualizovat, respektive získávat jeho aktuální podobu z úložiště.

5.3 Přenos mezi frameworky

V rámci knihovny určené k usnadnění přenosu stavu mezi komponentami různých frameworků, konkrétně mezi AngularJS a ReactJS, je funkce Angularu hlavním mechanismem pro tento přenos. Konkrétně se jedná o funkci *bootstrap*. Obecně se pod pojmem bootstraping rozumí proces inicializace nebo spuštění frameworku či aplikace, který vytváří nezbytné základy pro její bezproblémový běh. V kontextu této knihovny přebírá funkce *bootstrap* kritickou odpovědnost za inicializaci samotného frameworku Angular.V této části knihovny je funkce bootstrap použita ke spuštění Angular modulu v Reactu a mapoování na React komponenty.

Druhou důležitou částí je Skriptový prvek. Tento prvek je vytvořen dynamicky pomocí metody *document.createElement* a slouží k načítání externích zdrojů do aplikace. V případě této knihovny se načítá knihovna AngularJS, aby bylo možné v Reactu aplikaci pracovat s moduly z Angularu. Tomuto skriptovacímu prvku jsou nastaveny parametry, aby vše probíhalo správně, viz kód 5.4. Při použití je pak potřeba ještě do tohoto scriptovacího prvku natáhnout Angular modul.

```
const script = document.createElement('script');
script.src =
'https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js';
            script.async = true;
            scriptRef.current = script;
            document.body.appendChild(script);
```

Ukázka kódu 5.4. Ukázka z incializace skriptu a dotahování knihovny Angular

5.4 Dokumentace

Součástí implementace knihovny je také dokumentace v kódu. Dokumentace v kódu formou komentářů je duležitá pro lepší orientaci v tom, co se v které části kódu děje. Dokumentace tedy zvyšuje přehlednost a zrychluje pochopení samotné implementace. Komentáře v Javě v této práci jsou psány tak, aby se z nich pomocí nástroje JavaDoc dala generovat i textová dokumentace, která popisuje aplikaci. Takže se nemusí chodit přímo do zdrojového kódu, aby se zjistilo, jak jednotlivé funkce fungují. V Reactu je dokumentace kódu psána stylem pro TypeDoc.

Kapitola 6
Testování

Testovaní je nedílnou součástí vývoje jakéhokoliv softwaru. Testování vyvinuté knihovny probíhalo dvěma způsoby. Uživatelské testování (v tomto případě byl uživatelem programátor) probíhalo tak, že knihovna byla použita ve vytvořené testovací aplikaci. Druhý způsob bylo Unit testovaní, jak na knihovně, tak na vytvořené testovací aplikaci. Testovací aplikace obsahuje několik komponent, ve kterých jsou postupně otestovány všechny části knihovny. Jsou zde testovány všechny funkce, které by měla knihovna poskytovat, jako je tvorba globálního uložiště. Jsou ozkoušeny základní operace se stavem, tedy tvorba, čtení, uprava a mazání (create, read, update, delete - CRUD). Bylo testováno, zda se tyto úpravy projeví i na jiných komponentách, které k tomuto stavu byly registrovány. Další částí testovací aplikace bylo testováno ukládání stavu na backend, a také přenos do jiné aplikace. Pro tento účel bylo vytvořeno pomocí springu testovací úložiště. V následujících kapitolách bude přiblížena integrace navržených částí knihovny do testovací aplikace. V neposlední řadě bylo testováno, zda knihovna umí mapovat stav z Angluar modulu do React komponety. Nejprve bude popsáno uživatelské testování, a poté budou popsány systémové testy.

6.1 Uživatelské testování

Během uživatelského testovaní byly postupně testovány všechny části knihovny. Testovalo se, jestli knihovna celkově dokáže naplnit požadavky, které byly definovány během analýzy, viz 3.5. Nyní tedy bude postupně popsáno jak testovaní probíhalo na jednotlivých částech knihovny.

6.1.1 Global store

Testování této části knihovny probíhalo v rámci komponenty nazvané *Crud*. Jak již ale bylo zmíněno, nejprve v rámci testovací aplikace bylo potřeba vytvořit globální uložiště (*store*) všech stavů komponent, co jsou v aplikaci, viz ukázka kódu 6.1. V rámci Crud komponenty se používali první 4 parametry z ukázky. Pomocí proměnné *items* se uchovávají do pole všechny vytvořené záznamy o uživatelích. Atribut *name*, jak název napovídá, ukládá jméno uživatele, se kterým se aktuálně pracuje v textovém poli. *Editing* je indikátor pro to, zda je nějaký záznam v items aktuálně editován, protože podle toho se mění text tlačítka, kterým se přidává, respektive potvrzuje editace. *EditingId* je potřeba pro určování toho, který záznam v items je aktuálně editován.

```
export const useStore = createGlobalStore({
    items: [],
    name: "",
    editing: false,
    editingId: null,
    chartData: {
        labels: [],
        datasets: [
            {
                label: '',
                data: [],
                backgroundColor: '',
                borderColor: '',
                borderWidth: 0,
            },
        ],
    }
})
```

Ukázka kódu 6.1. Vytvoření iniciálního úložiště v testovací aplikaci, kdy objekt obsahuje dva stavy *counter* a *userName*

Následně takto vytvořené úložiště bylo použito v komponentách prakticky stejně, jako by se použil Hook *useState*. Rozdíl je v tom, že se do *useStore* při inicializaci vkládá reference na klíč dané části globálního storu. Jak je vidět v kódu 6.2, vytváření stavů má 2 složky. Samotnou konstantu, která je navázaná na globální uložiště a funkci set, která aktualizuje globální uložiště a tím pádem i konstatnu. V tomto konkrétním případě jsou zde namapovány na globální stav 4 parametry z lokálního stavu. *Items*, což je pole objektů, které se vykreslují. Dále je zde *name*, což je stav vstupního pole. V tomto stavu jsou uloženy znaky, které uživatel zadal do pole se jménem. Potom, aby se rozlišovalo, zda se přidává nový objekt do pole, či edituje stávající, je zde *editing*.

```
    const [items, setItems] = useStore('items');
    const [name, setName] = useStore('name');
    const [editing, setEditing] = useStore('editing');
    const [editingId, setEditingId] = useStore('editingId');
```

Ukázka kódu 6.2. Mapování lokálních stavů komponenty na části globálního úložiště podle klíčů v testovací aplikaci

V ukázce kódu 6.3 je vidět, jak se pracuje s globálním stavem, a po odesílání dat z formulářovém pole. Pokud tedy chceme nějak stav upravovat, je zapotřebí využít funkce *set*, ve které se musí definovat, jakým způsobem bude upravovat globální stav. V tomto případě, pokud je *editing* nastaveno na true, tak se ve funkci *setItems* najde podle *id* položka, která se nasledně v globálním stavu upraví. V jiném případě je prvek přidán do pole s unikátním *id*. Na závěr je nastaven stav *name* na prázdnou hodnotu, což vyčistí pole *input*, aby bylo možné zadávat další jména.

```
const handleSubmit = event => {
    event.preventDefault();

    if (editing) {
        setItems(() => items.map(item => {
            if (item.id === editingId) {
                return { ...item, name };
            } else {
                return item;
            }
        }));
        setEditing(() => false);
        setEditingId(() => null);
    } else {
        console.log(items);
        setItems(() => [...items, { id: Date.now(), name }]);
    }

    setName(() => '');
};
```

Ukázka kódu 6.3. Ukazka přidávání/editace uživatelů do pole items

Dále je v testovací aplikaci i funkce *delete*, která opět upravuje globální stav, a to tak, že jednotlivé záznamy podle *id* maže. Je také testováno, že stav je sdílený mezi komponentami. Pro tyto účely byla vytvořena druhá komponenta, která má (podobně jako komponenta editující globální stav) svůj vnitřní stav registrovaný na stav *name* a *items* z globálního stavu a všechny úpravy se jí auktualizují a jsou následně vykreslovány.

Pro testování, zda knihovna zvládá synchronizaci stavu mezi komponentami, byla vytvořena druhá komponenta, která má svůj lokální stav registrovaný na stejné části globálního stavu, jako výše uvedená knihovna, jež tento globální stav upravuje. V této komponentě je vidět, že při změně globálního stavu si tato komponenta změní svůj vnitřní stav a změny vykreslí.

6.1.2 Proxy

Část knihovny nazvaná Proxy je na použití podobně jednoduchá jako předchozí část. V rámci testovaní ukládaní stavu mimo aplikaci a loadování zpět bylo vytvořeno mimo aplikaci globální úložiště stavů. Na toto úložiště je napojena proxy, která odbavuje dotazy z frontendu. Tato proxy má v sobě pamět cache, kdy jednotlivě uložené stavy mají dobu životnosti, po které jsou záznamy z cache smazány. Úložiště obsahovalo jako stav text zobrazovaný uživateli v testovacích aplikacích. V konkrétním případě máme tedy nastaveno tento stav takto, viz 6.4. V této testovací komponentě vytváříme graf dat v průběhu měsíců. Názvy parametrů jsou specificky dané knihovnou *react-chartjs-2*, díky které je tento graf vykreslován. Pole *labels* nám určuje časový horizont ve vykresolovném grafu. Pole *datasets* obsahuje objekty nastavení tohoto grafu. Můžeme nastavovat hodnoty, barvy a podobně.

```
const [chartData, setChartData] = useState({
        labels: [
        'January', 'February', 'March', 'April', 'May', 'June', 'July'
        ],
        datasets: [
        {
            label: 'Data 1',
            data: [65, 59, 80, 81, 56, 55, 40],
            fill: false,
            borderColor: 'red',
            tension: 0.1
        }
    ]});
```

Ukázka kódu 6.4. Stav, který bude ukládán na backend

Jelikož je tato část knihovny umístěná na backendu, aplikace pro uložení či získání stavu volá api pomocí HTTP požadavku, viz 6.5. Následná odpověď z proxy se jednoduše načetla do stavu komponenty aplikace, která se potom překreslila.

```
const getState = async () => {
    try {
        const response = await axios.get('/proxy/get-state?key=chart');
        setChartData(response.data);
    } catch (error) {
        console.error(error);
    }
};
```

Ukázka kódu 6.5. Ukázka testovacího backend kontroleru, který bere dotaz z proxy a vrací požadovaný stav

V testovací komponentě je pomocí tlačítka *update* upravována pouze barva grafu, ale na backend je posílán celý objekt včetně pole objeků *datasets*. Mohl by se tedy upravit jakýkoliv jiný parametr a ukládání na backendu by fungovalo stejně.

Pro testovaní požadavku přenášení stavu mezi aplikacemi byla vytvořena nová aplikace, která má stejné parametry, jako komponenta zmíněná výše. Volá stejné api, tedy proxy, a může tedy svůj stav aktulizovat z backendu, který tam byl uložen předtím z jiné aplikace. Opět tedy vidíme vykreslený stav, kterým je graf. V této druhé testovací aplikaci je ale možné upravovat lokálně číselné hodnoty viditelné v grafu místo barvy. Může uložit změněné hodnoty do backendu a první apliakce si může změněné hodnoty z backendu opět získat.

6.1.3 Anuglar

Pro testování přenosu stavu z Angular modulu do React aplikace, respektive do nějaké komponenty v této aplikaci, bylo zapotřebí nejprve si vytvořit Angular modul. Uvnitř tohoto modulu se vytvořily zkladní proměné, a jejich aktualizující funkce viz 6.6

```
const app = angular.module('angular-module', []);

app.controller('angular-ctrl', function($scope) {
    $scope.firstName = 'Karel';
    $scope.lastName = 'Novák';
    $scope.age = 25;

    $scope.updateFirstName = function(newFirstName) {
        $scope.firstName = newFirstName;
    };

    $scope.updateLastName = function(newLastName) {
        $scope.lastName = newLastName;
    };

    $scope.incrementAge = function() {
        $scope.age += 1;
    };
});
```

Ukázka kódu 6.6. Testovací Angular modul

AngularComponent vykresluje strukturu React elementů, které jsou napojeny na modul. Unvitř komponenty je použit hook *useEffect*, v němž je použita část knihovny pro přenos stavu. Tedy je v něm nainicializován skript, který zavádí Angular do Reactu. Následně je namapován předem vytvtořený modul na React element dané komponenty pomocí boostrapovací funkce přes identifikátor *angular-root*, jak je vidět v ukázce 6.7.

```
 script.onload = () => {
    angular.bootstrap(document.getElementById(
    'angular-root')
    ['angular-module']);
            };
```

Ukázka kódu 6.7. Script pro mapovaní modulu

Následně jsou využívány jednotlivé části modulu v React komponentě pomocí direktiv z Angularu, které jsou napsány jako tagy u jedntlivých elementů 6.8. Nejprve je pomocí direktivy definován kontroler modulu. Uvnitř tohoto modulu jsou pak použity další direktivy pro svázaní s jednotlivými atributy a funkcemi v kontroleru. Například pomocí *ng-bind* se u zobrazují hodnoty v stavu modulu. Pomocí *ng-model* jsou navázány příslušné funkce upravující hodnoty stavu. Direktiva *ng-click* určuje, co se má stát, pokud se klikne na dané tlačítko.

```
<div className="angular-container" ng-controller="angular-ctrl">
    <input type="text" ng-model="firstName" placeholder="First Name" />
    <input type="text" ng-model="lastName" placeholder="Last Name" />
    <p className="name-output" ng-bind="firstName"></p>
    <p className="name-output" ng-bind="lastName"></p>
    <p className="age-output" ng-bind="age"></p>
    <button className="increment-button"
    ng-click="incrementAge()">Increment Age</button>
</div>
```

Ukázka kódu 6.8. Mapovní na React elementy a využití direktiv z Angularu

6.2 Systémové testy

Testovací aplikace a části knihovny byly testovány pomocí unit testů. Unit testy testují jednotlivé dílčí funkcionality jednotlivých komponent či tříd aplikace. Společně s unit testy je v práci také využíváno mockování, což znamená, že jsou objekty či metody potřebné k testování nahrazeny jejich mocky. V zásadě se jedná o náhražku, která imituje danou metodu bez jejího reálného volání. V praxi se to používá v případě, že potřebujeme pouze vědět, zda byla metoda zavolána. Dále v případě, kdy v testu potřebujeme, aby nám vrátila data při zavolání s těmito parametry, ale nechcemene nebo nepotřebujeme ji volat napřímo [41].

6.2.1 Frontend

Pro testovaní frontendu psaného v React je použita knihovna *@testing-library/react* společně s testovacím frameworkem Jest. Testy jsou uloženy v adresáři *___test___*. Podle standardu se vždycky testy pro danou komponentu jmenují stejně jako se jmenuje javascriptový soubor s příponou *.test*. Příklad testů, které řeší testovací případy pro Komponentu CRUD, tedy testy frontednu:

- Test: `vykreslí komponentu s položkami`
  Tento test ověřuje, zda se komponenta Crud vykresluje správně, pokud jsou v ní přítomny položky. K vykreslení komponenty se používá funkce render z testovací knihovny. Test pak kontroluje, zda vykreslená komponenta obsahuje konkrétní textové prvky představující názvy položek.
- Test: `vykreslí komponentu bez položek`
  Tento test ověřuje, zda se komponenta Crud vykresluje správně, pokud nejsou přítomny žádné položky. Zde je vytvořen *mock* z hooku *useStore* pro globální stav, který vrací prázdné pole položek. Test ověřuje, zda vykreslená komponenta neobsahuje žádné textové prvky reprezentující názvy položek.
- Test: `přidá položku`
  Tento test ověřuje, zda lze do seznamu přidat novou položku. Použije opět *mock* pro inicializaci globálního stavu z hook *useStore*. Test interaguje s komponentou změnou vstupní hodnoty a kliknutím na tlačítko přidat. Nakonec zkontroluje, zda byla zavolána funkce *setItems*, což znamená, že položka byla úspěšně přidána.
- Test: `upravuje položku`

Tento test ověřuje, zda lze existující položku upravit. Stejně jako u předchozích testů a potřebné funkce pro aktualizaci stavu jsou získávány z *mock* hooku *useStore*. Test komunikuje s komponentou kliknutím na tlačítko editace, změnou vstupní hodnoty a kliknutím na tlačítko uložit. Zkontroluje, zda byla zavolána funkce setEditing, což znamená, že položka byla úspěšně upravena.

- Test: `odstraní položku`

  Tento test ověřuje, zda lze položku ze seznamu odstranit. Je zde funkce *setItems* z *mock* hooku *useStore*. Test komunikuje s komponentou kliknutím na tlačítko delete. Nakonec zkontroluje, zda byla zavolána funkce *setItems*, což znamená, že položka byla úspěšně odstraněna.

Při testovaní unit testy se často zjišťuje, jaké je pokrytí aplikace těmito testy (*coverage*). Pro měření tohoto pokrytí je na frontendu použití přepínače *–coverage* za příkazem *npm test* v příkazové řádce, který je součástí testovací knihovny pro React. V této práci dopadlo pokrytí následovně, viz 6.1. Testování nemá 100% pokrytí, protože testování je zaměřeno především na logiku aplikace. Z těchto důvodů nebyly některé komponenty, například *App*, vůbec testovány a některé testovány jen z části.

File	% Stmts	% Branch	% Funcs	% Lines
All files	63.24	36.36	48.93	70.58
src	0	100	0	0
App.js	0	100	0	0
index.js	0	100	100	0
src/components	67.36	40	52.63	75.6
AngularComponent.jsx	91.66	50	66.66	91.66
ChartComponent.jsx	91.66	50	100	91.3
Crud.jsx	61.9	33.33	45.45	83.33
CrudView.jsx	100	100	100	100
NavBar.jsx	0	100	0	0
Store.jsx	0	100	100	0
angular-module.js	18.18	100	0	18.18
src/components/GlobalStoreLibrary	55.55	0	37.5	62.5
GlobalStore.jsx	55.55	0	37.5	62.5

Obrázek 6.1. Tabulka frontend coverage

6.2.2 Backend

Unit testování probíhalo i na backedové části knihovny a aplikace. Pro testovaní zde je použit známý testovací framework JUnit. Obshauje jak anotace pro běhy testů *@Test*, tak metody pro ověřování funkcionalit jako je *assert* a další funkcionality, ale v tomto projektu byli využívány především tyto. Pro mockování byla využívána knihovna Mockito, která je pro mockovaní v Javě podobně typická jako JUnit pro testovaní. Při testování Javy se testovací třídy Javy jmenují stejně jako testované třídy, mají ale příponu *Test*. Příklad testů pro třídu *StateServiceImpl* v proxy části knihovny:

- `saveObject_NewKey_ObjectSaved`:

  Při tomto testu se zjišťuje, zda při ukládání objektu s novým klíčem metoda *saveObject* správně vyvolá api backendu, do kterého uloží objekt, který vrátí.
- `saveObject_ExistingKey_ObjectReplaced`:

  Tento test ověřuje, zda se při ukládání objektu s klíčem, který je již uložen v mapě, nahradí existující objekt novým objektem.

- `getObject_ObjectExistsInMap_ReturnsObject`:
  Tento test je zaměřen na volání metody *getObject*. Testuje se volání při existujícím klíči v proxy mapě, zda je vrácen tento objekt a zda backendová služba pro ukládání stavu není volána.
- `getObject_ObjectDoesNotExistInMap_InvokesEndpointAndReturnsObject`:
  Tento test zajišťuje, zda při získávání objektu, který v mapě neexistuje, metoda getObject vyvolá backnedové api, získá objekt a vrátí jej.
- `scheduleExpiration_ObjectExists_ObjectRemovedAfterTTL`:
  V průběhu tohot testu se kontroluje chování metody *scheduleExpiration* tak, že čeká na vypršení TTL (Time to Live) a poté ověřuje, zda je objekt z mapy odstraněn.

Tyto testy pokrývají různé scénáře ukládání a načítání objektů, a také vypršení platnosti objektů na základě TTL.

Podobně jako u frontendové části aplikace je dobré mít představu o *coverage*, tedy pokrytí testy testované aplikace. Na backendové části aplikace byl pro měření *coverage* použit Maven plugin *JaCoCo* (Java Code Coverage). Výsledek měření dopadl takto, viz 6.2. Zde je pokrytí vyšší oproti frontendu, jelikož se víceméně v celém kódu backendu vyskytovaly pouze logické funkce.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
cz.cvut.fel.majerlu4.bp.proxyspring		58 %		n/a
cz.cvut.fel.majerlu4.bp.proxyspring.serviceImpl		100 %		100 %
cz.cvut.fel.majerlu4.bp.proxyspring.controller		100 %		100 %
Total	5 of 248	97 %	0 of 6	100 %

Obrázek 6.2. Tabulka backend *coverage*

6.3 Hodnocení výsledků testování

Jak bylo popsáno v rešerši a analýze, práce se stavy při tvorbě react aplikace je klíčová. React sám o sobě nabízí základní přístup ke stavům. Ten se ale zdá být neohrabaný, jakmile je aplikace složitější. Z tohoto důvodu začaly vznikat knihovny které tuto správu zjednodušují.

Navržená knihovna v této práci přistupuje ke správě stavů jednoduchým způsobem a vizuálně vychází ze základního hooku *useState*, aby při jejím používání byly minimální rozdíly oproti nativní správě stavu. Pro správu využívá principu globálního stavu, a tedy jednoho místa pravdy v celé aplikaci.

Z rešerše a analýzy vyplynuly zakládní funkční a nefunkční požadavky, které má navržená knihova pro správu stavu splňovat, viz 3.5. Při návhu a implemetaci byl hlavně kladen důraz na to, aby byla knihovna jednoduchá na použití, a zároveň zvládla všechny funkční požadavky

Během uživatelského testování bylo potvrzeno, že knihovna je schopna tyto požadavky naplnit.

Kromě uživatelského testovaní se vytvořily také systémové testy, a sice Unit testy. Ty testovaly funkčnost jednotlivých funkcí a metod. Toto testovaní bylo zaměřené na všechy řádky kódu, ale především na logiku aplikace a knihovny. Všechny tyto systémové testy prošly.

Celkově je tedy knihovna otestovaná dostatečně, jelikož byly vyzkoušeny všechny základní funkce včetně používání složitějších datových struktur pro stavy, jako je objekt obsahující pole objektů. Také všechny vytvořené unit testy procházejí.

Z těchto testování tedy vyšlo, že knihovna je schopná odbavit základní definované funkční požadavky a je jednoduchá na použití. Má však svá omezení, není například schopná automaticky aktualizovat lokální stav na frontednové části aplikace z backendového úložiště. O tento stav se vždycky musí z frontendu požádat. Další omezení je například v tom, že neukládá historii změn stavu, která by mohla být dobrá například při debugování.

Kapitola 7
Závěr

Cílem této práce bylo seznámit se s problematikou vývoje frontendových webových aplikací, dále analyzovat správu stavu webových aplikací, a to především ve frameworku React. Následně bylo cílem zjistit, jaké problémy je třeba řešit při této správě. Dále představit a popsat některé z nejznámějších knihoven pro framework React. Poté navrhnout a implementovat vlastní verzi knihovny, která bude některé z těchto problémů řešit.

V teoretické části bylo popsán JavaScript, a proč je důležitý pro tvorbu dynamických webových aplikací. Dále byly popsány současné nástroje a knihovny zvyšující použitelnost JavaScriptu, bez nichž se vývoj většiny moderních aplikací v tomto jazyce neobejde.

Následně byly popsány možnosti správy stavu aplikace v JavaScriptovém frameworku React. Jedním z těchto způsobů bylo zapouzdření stavu do knihovny. Nejznámější knihovny byly rozebrány v jednotlivých kapitolách, bylo pospáno jak fungují a jaké mají výhody či nevýhody.

Při návrhu knihovy se vycházelo z poznatků zjištěných v analýze, a to především z funkčních požadavků. Navržená knihovna byla rozdělelna do tří částí, kdy každá z nich řešila některé požadavky identifikované v analýze. První řešila problém sdílení stavů pomocí globálního stavu mezi komponentami jedné aplikace, incializaci stavu komponent z globálního stavu a úpravu stavu. Druhá část řešila problém sdílení stavů mezi jednotlivými aplikacemi, ukládání stavu mimo aplikaci, aby se z tohoto stavu mohla komponenta znovu obnovit i po případném restartu. Poslední část řešila předávání stavu z Angular modulu do React komponenty.

Následně byla navržená knihovna implementována a její funkcionalita byla otestována pomocí testovacích aplikací a unit testů.

Do budoucna by se dal vytvořit balíček nejběžnějších komponent, které by již využívaly tyto knihovny, a tím pádem by bylo použití ještě přívětivější. Také by se díky tomu snížila chybovost, jelikož by je programátor nevytvářel sám. Dále by se mohla přidat rozšíření, která by zjednodušovala debugovaní, například pomocí historizace všech změn ve všech stavech.

Literatura

[1] ČVUT FEL. *13. 2. 1992. byl to ten slavný den, kdy k nám byl zaveden internet.* https://intranet.fel.cvut.cz/cz/aktuality/2022/30_let_internetu. Navštíveno 2023-01-07.

[2] Statista Research Department. *Internet and social media users in the world 2022.* 2022. https://www.statista.com/statistics/617136/digital-population-worldwide/v. Navštíveno 2023-01-07.

[3] Jiří Kosek. *Základy protokolu HTTP.* https://www.kosek.cz/clanky/iweb/05.html. Navštíveno 2023-01-07.

[4] David Powers. *Beginning CSS3.* Apress, 2012.

[5] GeekTrainer. *REACT STAV a události - training.* https://learn.microsoft.com/cs-cz/training/modules/react-states-events/. Navštíveno 2023-01-07.

[6] Wsantos. *Which API types and architectural styles are most used?* 2018. https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26. Navštíveno 2023-01-07.

[7] Ondřej Žára. *JavaScript.* Computer press, 2021.

[8] *React state management: What is it and why to use it?* https://www.loginradius.com/blog/async/react-state-management/. Navštíveno 2023-01-07.

[9] *Historical trends in the usage statistics of client-side programming languages for websites.* https://w3techs.com/technologies/history_overview/client_side_language/all. Navštíveno 2023-01-07.

[10] *JavaScript with syntax for types.* https://www.typescriptlang.org/. Navštíveno 2023-01-07.

[11] *ECMA-262 standard.* 2022. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/. Navštíveno 2023-01-07.

[12] Paul Wilton a Jeremy McPeak. *Beginning JavaScript.* Wiley, 2010.

[13] Frank Zammetti. *Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker.* Springer, 2020.

[14] *About NPM.* https://docs.npmjs.com/about-npm. Navštíveno 2023-01-07.

[15] Adam Murray Alfrick Opidi. *NPM vs. Yarn: Which package manager should you choose?* 2022.

https://www.whitesourcesoftware.com/free-developer-tools/blog/npm-vs-yarn-which-should-you-choose/. Navštíveno 2023-01-07.

[16] Bibeault, De_Rosa a Katz. *JQuery in action*. Manning Publications, 2015.

[17] Cory Gackenheimer. *Introduction to react: Using react to build scalable and efficient user interfaces*. Apress, 2015.

[18] *Virtual dom and Internals*. https://reactjs.org/docs/faq-internals.html. Navštíveno 2023-01-07.

[19] *Introducing hooks*. https://reactjs.org/docs/hooks-intro.html. Navštíveno 2023-01-07.

[20] *Hooks at a glance*. https://reactjs.org/docs/hooks-overview.html. Navštíveno 2023-01-07.

[21] Olga Filipova. *Learning Vue. JS*. PACKT Publishing Limited, 2016.

[22] *Introduction Vue.js*. https://vuejs.org/guide/introduction.html#single-file-components. Navštíveno 2023-01-07.

[23] *What is Angular?* https://angular.io/guide/what-is-angular. Navštíveno 2023-01-07.

[24] *Angular vs react – a complete comparison* . https://brainhub.eu/library/angular-vs-react-comparison/. Navštíveno 2023-01-07.

[25] *Component state*. https://reactjs.org/docs/faq-state.html. Navštíveno 2023-01-07.

[26] *Context*. https://reactjs.org/docs/context.html. Navštíveno 2023-01-07.

[27] Kasra Khosravi. *Options for optimizing caching in react*. 2021. https://blog.logrocket.com/options-caching-react/. Navštíveno 2023-01-07.

[28] Bruno Carmine. *Do you really need a react state management library?* 2022. https://blog.bitsrc.io/react-do-you-really-need-an-external-library-for-state-management-28f67d03ebe5. Navštíveno 2023-01-07.

[29] Author Rob Callaghan. *React State Management Libraries: Which one?* 2022. https://www.matillion.com/resources/blog/react-state-management-libraries-which-one. Navštíveno 2023-01-07.

[30] *Why use react redux?* 2021. https://react-redux.js.org/introduction/why-use-react-redux. Navštíveno 2023-01-07.

[31] *Redux Essentials, part 1: Redux overview and concepts*. 2022. https://redux.js.org/tutorials/essentials/part-1-overview-concepts. Navštíveno 2023-01-07.

[32] *The gist of MobX*. https://mobx.js.org/the-gist-of-mobx.html. Navštíveno 2023-01-07.

[33] *Readme - MobX*. https://mobx.js.org/README.html. Navštíveno 2023-01-07.

[34] *Core concepts: Recoil*. https://recoiljs.org/docs/introduction/core-concepts. Navštíveno 2023-01-07.

[35] *Concepts - jotai, primitive and flexible state management for react.* `https://jotai.org/docs/basics/concepts`. Navštíveno 2023-01-07.

[36] *Comparison Jotai.* `https://jotai.org/docs/basics/comparison`. Navštíveno 2023-01-07.

[37] *Jotai - text length example.* 2023. `https://codesandbox.io/s/github/pmndrs/jotai/tree/main/examples/text_length`. Navštíveno 2023-01-07.

[38] *Comparison.* `https://docs.pmnd.rs/zustand/getting-started/comparison`. Navštíveno 2023-01-07.

[39] Zustand Documentation. *Introduction.* `https://docs.pmnd.rs/zustand/getting-started/introduction`. Navštíveno 2023-01-07.

[40] Harihara Subramanian a Pethuru Raj. *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs.* Packt Publishing Ltd, 2019.

[41] *Software Testing - Mock Testing.* `https://www.geeksforgeeks.org/software-testing-mock-testing/`. Navštíveno 2023-04-20.