Mendelova univerzita v Brně
Provozně ekonomická fakulta

Využití JavaScriptu pro single-page aplikaci vytvářející dokumenty

Bakalářská práce

Vedoucí práce:
Ing. Petra Čačková, Ph.D.

Šimon Mareš

Brno 2015

Na tomto místě bych rád poděkoval Ing. Petře Čačkové, Ph.D. za rady a podporu při vedení práce. Děkuji také své rodině.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Využití JavaScriptu pro single-page aplikaci vytvářející dokumenty**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 4. ledna 2015 .....

Abstrakt

Mareš, Šimon. Využití JavaScriptu pro single-page aplikaci vytvářející dokumenty. Bakalářská práce. Brno, 2015.

Práce se zabývá aktuálním stavem JavaScriptových technologií a single-page aplikacemi. Okrajově jsou také diskutovány problematiky tvorby dokumentů a webové typografie. Praktickou část tvoří implementování a popis JavaScriptového editoru dokumentů včetně použitých nástrojů a technik. Aplikace je napsána zejména v knihovně React.

Klíčová slova

JavaScript, single-page aplikace, React, Closure Tools, tvorba dokumentu, webová typografie

Abstract

Mareš, Šimon. Use of JavaScript for single-page application creating documents. Bachelor thesis. Brno, 2015.

This thesis deals with current state of JavaScript based technologies and single-page applications. As minor topics, text authoring and web typography were also discussed. A practical part is based on implementation and description of JavaScript document editor including used libraries and techniques. Application is written mainly in React library.

Keywords

JavaScript, single-page applications, React, Closure Tools, document writing, web typography

Obsah

1 Úvod a cíl práce

1.1 Úvod

Oliver Reichenstein v roce 2006 napsal, že webový design je převážně o typografii. Obsah webu a jeho možnosti se od té doby výrazně změnily, ale text zde má stále nezastupitelné místo. Protože publikovat na internetu může každý i bez znalosti typografie, nachází se zde dokumenty, které nejsou čitelné. Zvláštním případem je webová encyklopedie Wikipedia, která poskytuje obsah mnoha lidem, ale roztahuje text na celou šíři stránky a tím snižuje čitelnost.

Tato práce se zabývá zejména JavaScriptem, jazykem běžícím v prohlížečích. Jeho možnosti se od roku 2006 změnily opravdu významně. Už několik let se nepoužívá pouze pro zlepšení uživatelského rozhraní, ale jsou v něm napsány i programy, které dříve byly určeny pouze pro určité operační systémy. Dnes je možné vyvíjet webové stránky a JavaScriptové aplikace v textovém editoru napsaném přímo v tomto jazyce[1]. Z hlediska práce je také zajímavá možnost psát dokumenty v prohlížeči[2]. Vývoj těchto aplikací byl umožněn až v posledních několika letech rychlejším prováděním JavaScriptu v prohlížečích.

Tato práce spojuje problematiku tvorby textů a aplikace typografie pro web s problematikou vývoje tzv. single-page aplikací, které mohou nahradit desktopové. Jako ukázková aplikace bude vytvořen editor HTML dokumentu, který by měl být vhodný pro běžné uživatele a zároveň podpořit tvorbu strukturovaných dokumentů.

JavaScript v této práci je použit ve verzi ECMAScript 5, který je implementován ve všech moderních prohlížečích.

1.2 Cíl práce

Cílem této práce je popsat aktuální stav jazyka JavaScript a technologií na něm založených nebo s ním souvisejících. Po tomto zjištění by měly být vybrány vhodné nástroje či knihovny použité při implementaci single-page aplikace určené pro tvorbu webových dokumentů.

[1] vývojové prostředí Cloud9

[2] textový procesor Google Docs

2 Teorie – JavaScript

2.1 JavaScript jako programovací jazyk

JavaScript je skriptovací jazyk, jehož typický interpret je webový prohlížeč. Mezi jeho vlastnosti patří, že je dynamický a netypový (Flanagan, 2011, s. 1). Jazyk umožňuje programování v mnoha programovacích paradigmatech, např. objektové, funkcionální a procedurální programování.

Pro JavaScriptový kód je typické, že je zpracován asynchronně nebo používá události (Flanagan, 2011, s. 296). Mnoho API pro JavaScript je k tomu přizpůsobeno. Neblokující kód je důležitý v uživatelských rozhraních.

Dynamičnost se projevuje mimo jiné v možnosti přidávat nebo odstraňovat, respektive měnit objektům vlastnosti během běhu programu (Flanagan, 2011, s. 115).

Mezi funkcionální prvky jazyka patří mimo jiné tzv. prvotřídní funkce. Funkce je instance typu Object a je možné s ní manipulovat jako s objektem či jinou hodnotou – např. předávat parametrem nebo vracet ve funkci. Pomocí prototypové dědičnosti a tzv. konstrukčních funkcí je možné programovat do určité míry objektově.

Jazyk byl vytvořen pro validaci údajů zadávaných uživateli v prohlížeči pro omezení požadavků na server (Zakas, 2009, s. 29). Je to jazyk implementovaný ve všech hlavních prohlížečích a je možné jej použít i v jiných prostředích (Flanagan, 2011, s. 2).

ECMAScript je standard organizace Ecma International, ze kterého vychází implementace JavaScriptu. Standard nespecifikuje hostující prostředí jazyka. Standardní knihovna jazyka nemá prostředky pro vstupně-výstupní operace. Ty musejí být zpřístupněny právě v hostujícím prostředí. Mezi hlavní hostující prostředí v dnešní době patří webový prohlížeč a Node.js.

Webový prohlížeč JavaScript v prohlížeči je standardizován kromě ECMAScriptu zejména specifikací DOM (Document Object Model). DOM je standardní rozšíření hostujícího prostředí webového prohlížeče. Nestandardní rozšíření pro skriptování prohlížeče se někdy nazývá BOM (Browser Object Model) (Zakas, 2009, s. 30–31). API pro JavaScript v tomto prostředí tvoří DOM a další standardní API specifikované organizací W3C.

DOM Pro změnu HTML dokumentu v prohlížeči slouží rozhraní DOM pro JavaScript. DOM neboli Document Object Model je standardní, na jazyku nezávislé, API pro čtení a změnu dokumentů typu HTML a XML (Zakas, 2009, s. 345).

Změnit HTML lze pomocí JavaScriptového rozhraní DOM, metod manipulující s uzly. Uzly je možné vytvořit programově. Další možností je použít rozšíření pro DOM innerHTML, což je vlastnost dostupná v každém elementu.

Podstrom DOM je možné také číst a upravit jako HTML řetězec pomocí vlastnosti innerHTML. Při zápisu se výraz analyzuje v nativním kódu prohlížeče a poté se z něj vytvoří DOM. Ten nahradí potomky elementu, nad kterým je vlastnost

innerHTML vyhodnocena (Zakas, 2009, s. 333). Kvůli potřebě inicializace analyzátoru by se innerHTML mělo používat pouze pro větší části HTML, kdy je navíc nativní kód rychlejší. Pokud jsou zaregistrovány obsluhy událostí na některých elementech v nahrazeném podstromu a nejsou odstraněny, pak zbytečně zabírají paměť a nebudou nikdy zavolány (Zakas, 2009, s. 336–337).

Manipulace i čtení pomocí DOM patří mezi *nejpomalejší* operace, které lze v JavaScriptu provádět. Tyto operace a jejich počet by se měly minimalizovat (Zakas, 2009, s. 346).

2.2 Aktuální stav

JavaScript je považován za plnohodnotný a výkonný jazyk již ve starší publikaci (Zakas, 2009, s. 29).

Dřívější implementace JavaScriptu nebyly považovány za dostatečně výkonné pro použití na serveru (Bolin, 2010, s. 17), dnes existuje mnoho aplikací a společností nějakým způsobem využívající Node.js pro běh JavaScriptu na serveru např. pro mobilní web (Node.js, 2014). Patří mezi ně: LinkedIn (server pro mobilní web), ebay nebo The New York Times.

Některé prohlížeče kód JavaScriptu kompilují do nativního kódu počítače, na kterém běží (Glover, 2011). To umožňuje rychlejší vykonávání kódu než při pouhé interpretaci.

2.3 Specifika jazyka

Prováděcí kontext

Kontext či obor platnosti určuje, které proměnné a funkce jsou dostupné v místě provádění. Globální kontext je přístupný v jakémkoliv místě provádění kódu[3]. Kontext je vytvořen pouze při volání funkce. Tím se přidá do řetězce kontextů dále označeným jako scope chain. Po skončení provádění funkce se kontext zruší a tím se proměnné a funkce v něm definované smažou. Kontext v rámci scope chain může přistupovat pouze k aktuálnímu kontextu nebo k vnějším kontextům (Zakas, 2009, s. 108–114).

Využití prototypů

Pro simulaci objektového programování je v ECMAScript 5 možné použít několik technik, z nichž nejefektivnější je využití prototypů (Zakas, 2009, s. 178–182). Prototyp je objekt, který je definovaný na každém objektu a na kterém se hledají vlastnosti, pokud nejsou nalezeny přímo na daném objektu. Pokud je funkce použita s operátorem new, pak se jedná o konstruktor a je vytvořen nový objekt, který má prováděcí kontext jako daná funkce, má tedy v oboru platnosti objekt prototype

[3] v prohlížeči jej tvoří vlastnosti objektu window

patřící konstruktoru. Pokud vlastnost není nalezena na objektu, je hledána v prototypu konstruktoru. Aby se zamezilo definování metod pro každou instanci zvlášť, metody se přiřazují do prototypu, který se sdílí mezi všemi instancemi kontextem.

Metodu hasOwnProperty() lze použít pro rozlišení, zda vlastnost je přítomna přímo na objektu nebo je dostupná přes nějaký prototyp.

Tuto techniku je možné použít i pro dědičnost. Zjednodušeně řečeno, do řetězce kontextů se přidá kontext zděděného konstruktoru (Zakas, 2009, 197-199).

2.4 Node.js

Patrně nejdůležitější technologii pro tuto práci představuje Node.js umožňující spuštění JavaScriptového kódu mimo prohlížeč.

Node.js je platforma pro JavaScript a kromě jeho spouštění obsahuje API jako každé jiné hostující prostředí JavaScriptu. Toto API je knihovnou zpřístupňující např. vstupně-výstupní operace nebo vytváření modulů. Platforma byla navrhnuta pro škálovatelné síťové aplikace, ve kterých se využije souběžnost pomocí neblokujícího respektive asynchronního zpracování kódu v jednom vlákně. Také je možné jej využít pro serverovou část webové aplikace, např. s použitím webového frameworku Express (Glover, 2011).

Platforma je založená na JavaScriptovém enginu V8, který byl uvolněn jako open-source z prohlížeče Google Chrome. V8 kompiluje JavaScriptový kód do nativního (Glover, 2011).

Node.js používá událostmi řízené vstupně-výstupní operace. Tyto časově náročné operace jsou zpracovány asynchronně, a proto neblokují další kód (Glover, 2011). Když skončí I/O operace, zavolá se callback funkce s výsledkem.

Node.js umožňuje, aby nástroje pro webové aplikace byly vytvořeny přímo v JavaScriptu. Pomocí platformy byl vytvořen transpilátor pro CoffeeScript, běží na něm buildovací nástroj gulp. Také lze v něm použít knihovny používané i na straně klienta – např. React a PouchDB (použité v této práci).

2.5 Single-page aplikace

Single-page aplikace, dále označené jako SPA, je webová aplikace, která běží na straně klienta v prohlížeči. Podle (Osmani, 2013) lze definovat SPA jako aplikaci, která se načte v prohlížeči, a poté se stará o vykreslování a navigaci aplikace. Tím se SPA odlišují od tradičního modelu, ve kterém jsou pohledy uživatelského rozhraní vykresleny na straně serveru.

Se serverem se po inicializaci komunikuje pouze, pokud jsou potřeba dodatečná data, nebo je nutné uložit stav aplikace.

Serverově založené aplikace Tradiční webové aplikace jsou tvořeny jednotlivými stránkami, které odkazují na zdroj serveru, na kterém běží. Při přístupu na stránku

dojde k jejímu vykreslení na serveru, který vrátí HTML. Informace mohou být získány z databáze a předány šabloně generující HTML (Anderson, 2010b). Navigace v aplikaci mezi stránkami vyžaduje kompletní načtení nové stránky. Prohlížeč teoreticky musí zpracovat CSS, JavaScript a HTML pro každou stránku (Osmani, 2013). Ta se přitom nemusí výrazně změnit.

Pro interaktivní uživatelská rozhraní, jako jsou přesuny elementů se většinou použije knihovna typu jQuery. Problém je odlišnost aktuálního stavu DOM od dokumentu, který byl vykreslen na serveru, synchronizace dvou stavů (DOM v prohlížeči a např. data v databázi serveru) může být velmi náročná a kód náchylný k chybám (Osmani, 2013). Navíc se duplikuje pohledová logika, která je již implementovaná na serveru.

Využití SPA Pro zlepšení rychlosti odezvy, zvyšující použití aplikací, bývají vytvářeny single-page aplikace (Osmani, 2013). SPA komunikují se serverem asynchronně, což umožňuje překreslení pohledu až v době, kdy jsou data připravena. Stránka se nemusí překreslovat celá, ale pouze potřebná část. Před přijetím dat je možné aplikaci dále používat.

Požadavky na SPA

Pokud budeme vycházet z výše uvedené definice, pak každá single-page aplikace musí mít vyřešené, jak bude vykreslovat uživatelská rozhraní, a jak bude probíhat navigace. V této souvislosti vznikají problémy, které v tradičních webových aplikacích neexistovaly, popsané níže.

Navigace Navigace v serverově založených webových aplikací je tvořena odlišnými URL. Přechod na nové URL vytvoří bod v historii prohlížení. Single-page aplikace může změnit obsah stránky, aniž by se změnilo URL nebo historie prohlížeče. Stránky v rámci SPA pak mají jednu URL a není možné se na ně odkázat. Protože nemění historii, není možné jí procházet.

Pro vyřešení těchto dvou problémů je nutné použít router pro prohlížeč. Router je možný implementovat pomocí HTML5 rozhraní History a Location. Po inicializaci router v prohlížeči zachycuje kliknutí na URL, které by normálně vytvořilo požadavek na server, a volá příslušnou callback funkci, která by měla aplikaci změnit do stavu pro dané URL.

Vykreslování Další problém nastává při vykreslování v prohlížeči, k čemuž nelze přistupovat stejně jako na serveru. Server generuje HTML jako text, ke kterému se v prohlížeči při načtení stránky vytvoří DOM. Do něj je postupně uložen stav jednotlivých elementů (např. aktivní element v dokumentu, scroll pozice nebo označení textu) a elementům mohou být přiřazeny obsluhy událostí.

Celý podstrom elementu je možné změnit pomocí řetězce reprezentující HTML předaného do jeho vlastnosti innerHTML. I když je to možné, není to v mnoha pří-

padech vhodné, protože podstrom bude zcela nahrazen a tím se ztratí stav a obsluhy událostí uložené v nahrazeném podstromu.

Protože jsou informace vykresleny až v prohlížeči, nejsou zpravidla indexovatelné vyhledávači a dostupné v programech zpracovávající text stránky (Anderson, 2010b). Vykreslování v prohlížeči proto není vhodné pro webové stránky a webové aplikace poskytující textový obsah.

Protože aplikace v prohlížeči je stavová, knihovny a frameworky musí nějakým způsobem sledovat změny, které probíhají v aplikačních datech.

2.6 Knihovny a frameworky

Pro programování single-page aplikací existuje mnoho existujících řešení ve formě knihoven či frameworků. Dobrý přehled poskytuje web (TodoMVC, 2014), kde je možné nalézt mimo jiné knihovny Backbone.js, React a Ractive.js nebo frameworky AngularJS a Ember.js.

Frameworky se v tomto případě od knihoven liší v tom, že dodávají kompletní řešení pro většinu single-page aplikací, knihovny pouze řeší určitou problematiku nebo mají na sobě nezávislé komponenty. Byla snaha se frameworkům vyhnout, protože v porovnání s frameworky na straně serveru jsou nevyspělé – AngularJS byl vydán v roce 2009, Backbone.js 2010 (Shaked, 2014). Navíc je kód aplikace na frameworku závislý a méně flexibilní pro další změny.

Pro účely implementace byly evaluovány Backbone.js, Ractive.js a React.

HTML řetězec a DOM Jak bylo uvedeno výše, DOM umožňuje měnit části podstromu pomocí metod rozhraní DOM, nebo přiřazením řetězce do vlastnosti innerHTML. První způsob je vhodný pro malé změny, jako je změna textu nadpisu nebo přemístění elementu. Druhý je vhodný pro vytvoření velkých částí HTML.

V JavaScriptu bylo implementováno velké množství šablonovacích systémů, z nichž většina z předaných dat vytváří HTML řetězec. Při změně dat je nutné překreslit celou šablonu a nahradit celý podstrom DOM, který je jí reprezentován. To svádí k jakési optimalizaci rozdělením větších částí uživatelského rozhraní do více šablon, aniž by to bylo logicky vhodné (Harris, 2014b).

Ideální je vytvořit HTML ze šablony pomocí innerHTML a poté změny zanášet přes manipulaci existujícího DOM. Kombinování šablon a manuální DOM manipulace kromě nejednotnosti připomíná problémy uvedené v odstavci o serverově založených aplikacích. Užitečnou knihovnu pro tvorbu single-page aplikací lze tedy považovat takovou, která umí vytvářet UI oběma způsoby.

Backbone.js

Backbone.js má router, umožňuje strukturovat aplikaci (rozdělením na modely a pohledy), sledovat změny v modelech nebo navázat obsluze události objekt this na daný

pohled. Přínos knihovny pro SPA je poměrně nízký, protože neřeší tvorbu uživatelského rozhraní, což bylo výše definováno jako základní požadavek všech SPA.

Každý pohled má metodu render, ve které dochází k manuálnímu vykreslení dat pomocí nějakého šablonovacího systému. Pohled je nutné spojit s modelem také ručně (Osmani, 2013). Protože pohledy manipulují přímo s DOM, činí je to méně znovupoužitelné, např. pro jednotkové testování (Shaked, 2014).

HTML je strukturovaný dokument, který je tvořen zanořenými elementy. Protože se pohledy vytvářejí ručně, je nutné je také ručně propojit (Osmani, 2013). Z poměrně rozsáhlé části v (Osmani, 2013) věnované problémům se zanořenými pohledy a modely lze vyvodit, že Backbone.js není vhodná knihovna pro aplikace s hierarchickými daty.

Ember.js

Ember.js je podobný Backbone.js, pouze je to framework. Vyžaduje, aby data byla zapouzdřena pomocí modelů frameworku. To umožňuje sledovat změny, které se promítnou do pohledů. Komponenty jsou zde na rozdíl od Backbone.js propojeny. Pohledy vytvářejí *nevhodný* HTML řetězec (Shaked, 2014).

AngularJS

AngularJS je framework, který rozšiřuje jazyk HTML tak, aby byl dynamický a vhodný pro aplikace (Google, 2010). Abstrakcí nad DOM/HTML je vlastní šablonovací systém, který rozšiřuje syntaxi HTML. Ve většině případů není nutné manipulovat s DOM. Hlavní funkcionalitou frameworku je oboustranné navázání dat (two-way data binding), na kterém je postavena synchronizace modelů a DOM. Změny jsou detekovány automaticky a není nutné sledovat změny v modelech. Framework je určen pro CRUD či AJAX aplikace. Není naopak určen pro aplikace intenzivně nebo netypicky využívající DOM.

Projekt je vyvíjen a podporován společností Google a je velmi úspěšný (Shaked, 2014). Součástí frameworku je prakticky vše, co je nutné pro tvorbu aplikací, pro které je určený: propojení aplikačních dat s DOM, šablonování, routování, či znovupoužitelnost komponent. Kromě toho obsahuje i správu závislostí pomocí DI[4], asynchronní zpracování s Promises, validaci formulářů nebo nástroje pro testování (Google, 2010). AngularJS byl navrhnut pro snadné testování. Ke snadnějšímu testování pomáhá rozdělení aplikace do několika částí podle (Shaked, 2014) na: „Controllers, Directives, Factories, Filters, Services and Templates.“

Šablonovací systém není jen deklarativní. Šablony ve výrazech umožňují např. přiřazování, ale není to plnohodnotný JavaScript. Pro navázání dat je nutné zadávat informace o modelech nebo obsluh událostí pomocí řetězce, což je náchylné k zbytečným chybám při refactoringu.

[4]Dependency Injection viz s. 28

React a Ractive.js

Následující porovnání vychází z článku (Harris, 2014a) autora Ractive.js. V něm jsou společné vlastnosti knihoven vystihnuty takto:

> Finally, both libraries believe that the way to help developers build complex apps is to give them tools that encourage simplicity and composability and then get out of their way.

Obě knihovny byly vyzkoušeny a lze potvrdit, že je jednoduché s nimi začít. Principy lze efektivně aplikovat i poté, co se aplikace rozrůstá.

React a Ractive.js mají společné následující:

- slouží pouze pro vytvoření UI
- abstrakce nad DOM implementovaná v JavaScriptu, minimální manipulace s DOM
- vykreslení na serveru v Node.js
- propagování změny do všech závislých míst (reactive programming)

Poslední položku je vhodné doplnit o informaci, co je to reactive system / programming (Harris, 2014a):

> Put simply, in a reactive system where the value of b depends on the value of a, if a changes then b will also change.

Odlišnosti

Vytvoření pohledů V React se HTML vytváří pomocí volání funkcí, kód pohledu je celý v JavaScriptu. To umožňuje používat všechny jazykové prostředky, není potřeba se učit nic nového a kód je flexibilní. Takovéto pohledy jsou staticky analyzovatelné, mohou být odladěny a optimalizovány jako jiný JavaScriptový kód, existují zvýrazňovače syntaxe v editorech apod. Pomocné funkce pro vykreslení v komponentě jsou metodami komponenty a lze je testovat stejně jako ostatní kód. Ractive.js je nutné funkce předávat do dat pohledu, což víceméně snižuje flexibilitu kódu. Ractive.js používá šablony, které vycházejí z Mustache, ale přidávají vlastní funkcionalitu. Výhodou Mustache syntaxe je, že šablonu tvoří HTML, do kterého jsou přidány výrazy. Je možné zkopírovat existující části HTML kódu a přidat k tomu funkcionalitu pomocí prostředků šablony.

Změny dat Při změně stavu se pohled v React rekurzivně překreslí ve virtuálním DOM a pouze změny jsou promítnuty do skutečného DOM. Ractive.js také mění DOM pouze, pokud je to nutné, ale data aplikace rekurzivně nepřekresluje.

Používá abstrakci nad DOM pro aplikovaní nejmenší možné změny zjištěné přes sledovací mechanismus změn. Knihovna ví, kde se změněná data používají a vykreslí pouze tyto části aplikace. Tento přístup, na implementaci složitější, by měl být podle (Harris, 2014a) zpravidla rychlejší než React.

Vyhodnocení

AngularJS, React i Ractive.js umějí efektivně pracovat s DOM, a proto jsou vhodné pro single-page aplikace. Ve všech lze použít normální JavaScriptové objekty, čímž neomezují modelovou logiku. AngularJS nebyl vybrán pro svoji složitost. Výhodou Ractive.js vůči React je podpora pro animace a SVG, což knihovnu činí ideální pro vizualizaci dat. Měla by být také rychlejší.

Pro implementaci byla vybrána knihovna React. Nejvíce váhy mělo používání JavaScriptu pro definování uživatelského rozhraní. Méně důležité bylo, ale ve svém důsledku hodně užitečné, že React je vyvíjena velkou softwarovou firmou, která se o projekt stará. Praktickým užitkem je rozšíření React Developer Tools pro prohlížeč Google Chrome.

2.7 Závěr

Interpretace JavaScriptu je v dnešní době velmi výkonná. Pro tvorbu velkých aplikací však není sám o sobě ECMAScript 5 moc vhodný, protože nemá správu závislostí a jednoduchou syntaxi pro tvorbu metod[5] s možností prototypové dědičnosti.

Kvůli tomu vznikají nové jazyky přinášející novou syntaxi, jmenovitě CoffeeScript nebo TypeScript (Bolin, 2014).

[5] Nejméně problémový zápis jedné metody je: Person.prototype.sayName = function(){} (Zakas, 2009)

3 Teorie – typografie a tvorba textů

V následující kapitole budou popsána pravidla a doporučení, která se vztahují k webu. Webová typografie zahrnuje pouze část počítačové typografie, což je dáno tím, že webový dokument nemá určený formát stránky. Šířku lze omezit, ale tím se zhorší použitelnost napříč zařízeními.

Kvůli různé délce stránky se neřeší problematika týkající se zalomení stránek, umístění poznámek pod čarou na stránce nebo již zmíněný sázecí formát. Kvůli různé šířce stránky a absence algoritmu pro dělení slov v prohlížečích, se nesází do bloku a neřeší se jmenovitě tzv. řeky nebo vdovy.

3.1 Typografické pojmy

V následujícím textu budou použity tyto pojmy, převzaté z publikace o počítačové typografii (Rybička, 2011).

Typografický bod – absolutní jednotka, jejíž skutečná velikost je dána použitým písmem a typem bodu (monotypové, anglické).

Čtverčík – dvojrozměrná relativní jednotka s oběma hranami o velikosti stupně písma aktuálního textu. Používá se pro relativní vyjádření vzdálenosti.

Rodina písma – slučuje jednotlivá písma s odlišnými řezy do jedné skupiny.

Písmo – množina znaků s konkrétním řezem.

Pravý prapor – pokud je odstavec zarovnaný na pravý prapor, pak je text zarovnán na levé straně.

Řez písma – modifikuje základní tvar písma.

Základní písmo – je jím napsán obsah dokumentu, je tvořen převážně odstavci či seznamy.

3.2 Přehlednost a struktura dokumentů

Dlouhý souvislý text odrazuje od čtení, je lepší jej rozdělit do kratších celků (Golombisky, 2010, s. 97–99). Nejzákladnějším prvkem pro zpřehlednění je rozlišení odstavců, např. vertikálním odsazením. I při tomto zpřehlednění by odstavce neměly být příliš dlouhé.

Nadpisy, jako velmi důležitý prvek, by měly být vůči okolnímu textu kontrastní. Kromě samotných nadpisů jsou důležité i mezery kolem nich. Mezera před nadpisem by měla být větší než za ním, protože se nadpis předchozího textu netýká a vyčleňuje se z něj. Nadpisy zpřehledňují dokument i tím, že podávají informaci o hierarchickém uspořádání informací.

Posledním, ale neméně důležitým, prvkem jsou odrážkové seznamy. Jednotlivá položka je velmi rychle přečtena, zvláště pokud je nastaveno vhodně velké řádkování.

Oproti odstavci se hodí pro stručné fráze. Seznam může být od obklopujícího textu odlišen odsazením.

3.3 Typografická pravidla a doporučení

Marginálie Jsou to krátké poznámky, které shrnují nebo doplňují odstavec, vůči kterému jsou vysázeny na okrajích stránek. Text marginálie je zarovnán vůči odstavci. Pokud jsou napravo od textu, tak je jejich text zarovnán na levé straně.

Velikost písma by měla být menší než základní, ke kterému se vztahují. Umístění účaří by mělo být ve stejné výšce jako účaří prvního řádku příslušného odstavce (Kočička, 2004, s. 178).

Nadpisy Slouží pro přehlednost a vyznačení hierarchie informací v dokumentu pomocí jejich úrovní. Jednotlivé úrovně by měly být odlišeny stupněm písma. Od základního textu jsou odlišeny stupněm písma, případně jiným písmem, které se základním harmonuje. Všechny nadpisy jedné úrovně by měly mít jednotnou úpravu (Kočička, 2004, s. 176).

Obrázky Mohou být v rámečku, pokud část obrázku má stejnou barvu jako pozadí dokumentu. Pokud je jednomu obrázku přidán rámeček, měl by být z důvodu konzistence dokumentu přidán všem. Okraje rámečku by neměly být tlusté (Golombisky, 2010, s. 34–36). Při formátování vůči odstavci je možné obrázek zobrazit na celou šíři sazebního obrazce nebo jej nechat obtékat odstavcem zleva nebo zprava. Pokud je obrázek obtékán, měl by být obtékán jednotně na všech místech dokumentu. Popisek obrázku je možné zarovnat na střed nebo na prapor k jednomu z okrajů obrázku. Je možné popisek napsat i do marginálie k obrázku (Kočička, 2004, s. 182).

Odsazení odstavce Jednotlivé texty odstavců je nutné od sebe rozlišit, k čemuž slouží mimo jiné jejich vertikální odsazení. Je to mezera mezi dvěma odstavci. Velikost mezery má být stejná nebo poloviční jako je řádkování (Rybička, 2011, s. 43–44).

Odstavce Samotný obsah dokumentu je tvořen převážně odstavci. Zarovnat odstavec je možné na levý či pravý prapor, nebo do bloku. Do bloku není doporučeno pro texty, které nejsou sázeny pro tisk, protože mohou vznikat dlouhé mezery mezi slovy a tím i řeky v odstavci (Golombisky, 2010, s. 42).

Písmo V rámci dokumentu by se měla použít jednotná písma. Většinou stačí na celý dokument jedno nebo dvě. Je možné použít serifové písmo pro základní text a odpovídající bezserifové pro nadpis a poznámky (Kočička, 2004, s. 184).

Písmo je množina konkrétních znaků a může se stát, že některé písmo nebude mít potřebné znaky např. české (Rybička, 2011, s. 22).

Řádkování Velikost se určuje stejně jako u stupně písma, na kterém je řádkování závislé, v typografických bodech. Je to vzdálenost mezi účařími[6] dvou řádků. Vhodné řádkování kolem 1,2násobku stupně písma zlepšuje čitelnost textu.

Řádkování u velkých nadpisů je potřeba hlídat, protože s větším písmem se příliš zvětšuje mezera mezi účařími, pokud se jeho text nevejde na jeden řádek. V takovém případě je možné řádkování zmenšit (Golombisky, 2010, s. 96).

Stupeň písma Veličina je vyjádřena v absolutní jednotce typografický bod (Kočička, 2004, s. 184). Pokud se mění stupeň písma, měl by se měnit alespoň o 20 % (Rybička, 2011, s. 25), jinak není změna dostatečně rozlišitelná.

Jednotlivá písma mohou zabírat různé skutečné vertikální a horizontální místo pro znaky stejného stupně. Pro základní text je proto nutné manuálně vybrat velikost vhodnou pro dané písmo (Golombisky, 2010, s. 94).

3.4 Webová typografie

Jak je zmíněno v publikaci (Golombisky, 2010, s. 216), navrhnutá podoba stránky nemusí na konec v prohlížeči vypadat stejně. Kromě různých prohlížečů se webové stránky také zobrazují na odlišných zařízeních.

Velikost webové stránky

Stránka, a dokument v ní, může mít šířku buď fixní, nebo fluidní. Fluidní velikost se přizpůsobuje velikosti okna prohlížeče, a proto je nutné ve stránce mít prvky, které zaplní různou velikost stránky a umí se zalomit, pokud jsou větší než šíře stránky. S textem problém v tomto ohledu není.

Fixní velikost stránky znamená, že je stejně velká při každé velikosti okna. To umožňuje větší kontrolu nad tím, jak text bude vypadat (Golombisky, 2010, s. 212). Jednotlivé prohlížeče ale interpretují standardy s menšími nebo většími rozdíly, stránky nemusí vždy být zobrazeny tak, jak bylo zamýšleno (Golombisky, 2010, s. 216).

Protože se stránky stejně mohou zobrazit v různých prohlížečích jinak, může být lepší přístup nemít kontrolu nad velikostí stránky a umožnit přehlednější zobrazení na různých velikostech obrazovek pomocí fluidní velikosti stránky a dokumentu.

Obrázky

Hlavní požadavek na obrázek pro web je jeho velikost. Rychleji se načte a nižší rozlišení není na obrazovce dostatečně rozpoznatelné. Typicky stačí rozlišení o velikosti 72 dpi (Golombisky, 2010, s. 134–135). Mezi hlavní formáty pro web patří JPG pro fotografie a GIF nebo PNG pro grafiku (Castro, 2012).

[6] linka, na které jsou posazena písmena (Rybička, 2011)

Typy písma

Písmo textu ovlivňuje jeho čitelnost. Použití špatného písma může i odradit čtenáře. Volba písma by měla odpovídat danému textu (Golombisky, 2010, s. 86), proto je to odpovědnost autora či designéra dokumentu. Rodiny písma lze rozdělit do mnoha různých kategorií, pro webovou typografii je vhodné rozdělit rodiny písma jednoduše na serifová a bezserifová.

Bezserifová písma nemají serify a tloušťka čar bývá většinou jednotná ve všech místech znaku. Při zobrazení na obrazovce se používají přímo pro text dokumentu, jsou na obrazovkách čitelnější než serifová. Mezi nejčitelnější běžné písmové rodiny patří Helvetica, Verdana a Arial (Golombisky, 2010, s. 89).

Serifová písma mají serify a mohou mít dynamickou tloušťku čar napříč znakem. Serify v těchto písmech nejsou dostatečně zřetelné na obrazovkách, a proto by se měla používat spíše pro nadpisy a nejlépe tučné (Golombisky, 2010, s. 88).

3.5 Možnosti tvorby textu

Tvorba dokumentů je možná pomocí textových procesorů, které během psaní text formátují, nebo textových editorů, ve kterých se edituje zdrojový text standardního textového formátu. Mezi vhodné standardní textové formáty pro tvorbu elektronických dokumentů lze zařadit LaTeX, HTML5 a DocBook. DocBook svým principem nejvíce odpovídá problematice práce, a proto bude popsán detailně.

Webové dokumenty

Tvorba webových dokumentů znamená vytvořit HTML. Vhodná verze tohoto jazyka je HTML5, která přidává sémantické elementy vhodné pro dokumenty (*article*, *section*). Kromě přímého použití HTML se používají jiné syntaxe, které se do HTML převádí, např. Markdown. Existují také textové procesory, které formátují přímo do HTML, ale uživatel text vytváří v náhledu. Patří mezi ně například CKEditor. Obecně platí, že do HTML je možné převést, ovšem s různou kvalitou kódu, většinu formátů pro text.

LaTeX

LaTeXje systém pro tvorbu textů určených pro tisk nebo elektronické publikování. Je rozšířením multiplatformního typografického systému TeX. Jeho cílem je usnadnění tvorby textů v tomto systému rozdělením procesu strukturování a formátování dokumentu zavedením strukturních příkazů (Přichystal, 2006, s. 23). Dokumenty vytvořené v LaTeX slouží především pro tisk nebo se exportují do PDF. Lze je ale převést rozumně i do HTML, protože jsou tvořeny strukturními značkami.

Textové procesory

Programy tohoto typu automaticky formátují text během jeho psaní. Uživatel vidí, jak by měl dokument po publikování vypadat. Zpravidla jsou v nich dokumenty vytvářeny pro tisk, lze je ale exportovat do jiných výstupních formátů, jako je HTML nebo PDF.

Mezi tyto programy patří Microsoft Word, který patří k nejrozšířenějším produktům pro pořizování textů (Přichystal, 2006, s. 21). Jeho alternativou je též desktopový program Writer z kancelářského balíku OpenOffice nebo webová aplikace Google Docs.

Používání těchto produktů je rozšířeno hlavně mezi běžnými uživateli (Rybička, 2011, s. 98) a pro profesionální sazbu nejsou vhodné (Přichystal, 2006, s. 21). Přestože jsou určeny pro běžné uživatele, mají někdy příliš mnoho funkcí (Přichystal, 2006, s. 13, s. 18), jimž tito uživatelé nemusí rozumět. Svým principem nevedou k tvorbě strukturovaných a typograficky správných dokumentů (Přichystal, 2006, s. 21). Formátování během psaní odvádí autora od tvorby samotného obsahu k určování jeho vzhledu (Přichystal, 2006, s. 13).

Mezi další nevýhody patří:

- nekompatibilita formátů jednotlivých verzí (Přichystal, 2006, s. 15)
- nízká kontrola nad skutečným obsahem dokumentu
- většina příkazů pro formátování je před uživatelem skryta (Přichystal, 2006, s. 13)
- nejasné hranice formátování mohou způsobit těžko opravitelné chyby (Přichystal, 2006, s. 12)
- automatické formátování způsobuje triviální chyby, jako je vyznačení pouze části slova (Rybička, 2011, s. 94).

XML dokumentové systémy

Na XML jsou postaveny dokumentové systémy, mezi které patří zejména DocBook. XML je standardní textový formát umožňující vytvářet strukturovaná data zvaná dokumenty. V souvislosti s ním existuje mnoho různých XML technologií. Například data v uložená XML dokumentu je možné transformovat do jiného formátu pomocí XSL.

S možností strukturování je formát vhodný pro hierarchicky založená data či data s daným pořadím. Podle způsobu použití mají XML dokumenty různou úroveň strukturovanosti (Mlýnková, 2008, s. 11–13). Příkladem může být explicitní označkování odstavce textu značkou p (v HTML) oproti vyjádření konce odstavce prázdným řádkem (LaTeX).

Formát XML byl původně vytvořen pro přenos dat mezi aplikacemi. XML dokumenty jsou k tomuto účelu vhodné i proto, že je lze tzv. validovat vůči XML schématu. Ten explicitně vyjadřuje, jakou strukturu a data by měl dokument mít (Mlýnková, 2008, s. 11–13).

DocBook

DocBook je standardní XML formát pro tvorbu textu. Systém tvoří XML schémata, která validují strukturu dokumentu pomocí povolených elementů. Protože je DocBook standardní formát, je možné použít existující řešení pro převod textu do výstupních formátů (PDF, HTML, epub).

Primární použití byla technická dokumentace, ale dnes je možné DocBook využít i pro psaní článků a knih (Mlýnková, 2008).

Struktura je, stejně jako v XML, vytvářena pomocí značek, které obalují text a tím mu dávají význam. Značky jsou převážně párové, které naprosto jasně vymezují začátek a konec části textu, např. sekce.

DocBook obsahuje generické elementy vytvářející strukturu, aby bylo možné je použít na jiném místě (i v jiném dokumentu) bez narušení osnovy. V DocBook lze zapsat nadpis sekce pouze pomocí značky *title* bez explicitního vyjádření úrovně, kterou lze zjistit ze struktury dokumentu (Von Hagen, 2009). Tento přístup je flexibilnější pro změny ve struktuře dokumentu. Např. sekci (či jiný samostatný element) je možné získat z dokumentu některou XML technologií pro dotazování a programově ji použít na jiném místě.

HTML5

HTML5 je nová verze jazyka HTML, jehož standard není ve finální verzi. Tato verze klade důraz na sémantiku a odstraňuje formátovací značky (Castro, 2012, s. 27). Proto je vhodnější pro značkování textového obsahu než starší verze HTML. Mezi nové elementy vhodné pro značkování dokumentů patří header, footer a elementy, které rozdělují obsah.

Elementy rozdělující obsah V těchto elementech může být hierarchie nadpisů H1–H6, které se vztahují k nim a ne k celému dokumentu. Mezi ně patří elementy article, nav, aside a section (s. 59). Výhodou je nezávislost elementu včetně jeho obsahu na osnově dokumentu. To umožňuje je použít na jiné stránce, aniž by se narušila její struktura (s. 62). Konkrétně značka article je určena pro článek nebo jiný *samostatný obsah* stránky (s. 69).

Strukturu dokumentu lze vytvořit podobně jako v DocBook pomocí značek *section* pro vyznačení sekce a *h1* pro její nadpis. Takový dokument by bylo možné vložit i na jiné stránky, než pro které byl určen. Této praktice brání skutečnost, že některé programy zpracovávající obsah stránek neumí zpracovat takovouto osnovu a nejsou k tomu připraveny CSS styly, které by správně vyznačily nadpisy podle jejich důležitosti.

Závěr

V textových procesorech je možné příkazy dohledat v grafickém uživatelském rozhraní, ale pro efektivní práci je stejně nutné daný program dobře znát. Většinou je nutné používat myš (Přichystal, 2006, s. 18). Kromě znalosti příkazů vyžaduje LaTeX také netriviální instalaci a konfiguraci sázecího systému (Přichystal, 2006, s. 90–91), a proto je méně přístupný pro běžné uživatele. DocBook nijak neřeší prezentaci textu, a proto je pro běžné uživatele ještě méně přístupný než LaTeX.

Kvůli těmto nedostatkům by bylo pro tvorbu webových dokumentů ideální použít jazyk HTML5. Ten ale není v dnešní době zcela adaptován a standardizován. V případě jednoduchých článků je možné použít syntaxí odlišný jazyk, který je určen pro konverzi do HTML, může být ale omezující.

4 Použité knihovny a techniky

V této kapitole budou popsány techniky a knihovny, které jsou použity, respektive aplikovány v kódu aplikace. Zvláštní pozornost je věnována knihovně React, která byla pro aplikaci stěžejní.

4.1 Návrhové vzory

Observer

Návrhový vzor Observer (Pozorovatel) je používán, pokud je potřeba sledovat změny na nějaké instanci (Subject).

To je nutné, pokud pozorovatel využívá stav pozorované instance a jeho data mají zůstat konzistentní. Výhodou je, že Subject nemusí vědět nic o svých pozorovatelích. Těch může být různý počet. Zpráva o změně se v terminologii vzoru nazývá Notify. Subject může mít několik takových událostí. Pozorovatelé se mohou k notifikaci o změně přihlašovat nebo odhlašovat, pomocí metod pozorované instance Attach a Detach, které předají funkci, která má být při změně zavolána. Vzor neomezuje jakým způsobem pozorovatel získá aktuální data – zda přímo v metodě zaregistrované pomocí Attach (model push), nebo si je získá sám (model pull) (Bishop, 2010, s. 236–239).

Composite

Tento návrhový vzor je možné použít v mnoha případech, zejména v programech mající strukturovaná aplikační data, protože aplikací vzoru se vytváří strukturované hierarchie. Vzor tvoří dva typy objektů, prvním je komponenta (Component) a druhým je komponenta složená z jiných komponent (Composite). Oba typy mají společné rozhraní, Composite pouze v některých operacích využívá volání operací definovaných na komponentách, které ji tvoří.

4.2 Techniky v JavaScriptu

Delegování událostí

Delegování událostí je technika, ve které se přiřazuje jeden posluchač události nadřazenému EventTarget objektu místo toho, aby byl přiřazen každému jeho potomku. To snižuje potřebné množství paměti (posluchač se musí uchovat) a čas nutný k přiřazení posluchače událostí do DOM. Technika využívá probublávání události z potomka do rodiče. Událost probublává až do nejvyššího EventTarget objektu, kterým je window. Probublávání může být zastaveno voláním metody stopPropagation() na objektu události.

Posluchač, kterému jsou události delegovány, musí z objektu události zjistit, kde událost vznikla a provést patřičné operace (Zakas, 2009, 455–457).

DI kontejner

Dependency Injection (zkráceně DI) je technika správy závislostí, která přesunuje odpovědnost inicializace závislosti mimo samotný objekt, který ji používá. Ten definuje své závislosti např. jako argumenty v konstruktoru. Inicializace probíhá v hlavní funkci programu nebo v rámci DI kontejneru.

DI kontejner je objekt, který se stará o inicializaci požadovaných objektů a všech jejích závislostí tam, kde je to potřeba. To je typicky v hlavní funkci programu. Konkrétní závislosti je možné definovat v konfiguračním souboru nebo za běhu programu. (Martin, 2009, s. 170–171)

Promises

Promise je objekt zlepšující práci s asynchronním prováděním kódu. Také využívá callback funkce, ale umožňuje koordinovat jejich volání. Protože ECMAScript 5 nemá Promise specifikován jako konstrukt jazyka, jedná se v této verzi o techniku či návrhový vzor.

Kvůli rozličnosti implementací vznikl otevřený standard Promises/`A+` (Promises/A+, 2014). Pokud knihovna vrací Promise podle tohoto standardu, práce s ním je stejná, bez ohledu na implementaci, přes metodu then().

Promise umožňuje synchronně vrátit objekt reprezentující hodnotu, která ještě nemusí existovat, protože je získána asynchronně. K výsledku Promise lze zaregistrovat libovolné množství callback funkcí pomocí metody then(). Hodnota výsledku je neměnná, proto každá z těchto funkcí ji získá stejnou.

Metoda then() vrací vždy další Promise, což umožňuje jejich řetězení. Hodnota je vyřešená, pokud je ve splněném (fulfilled) nebo zamítnutém (rejected) stavu. V případě, že je hodnota zamítnutá a metodě then() není předaná funkce pro zachycení chyby (onRejected), jsou zamítnuty i následující Promise v řetězci. To umožňuje zpracování chyb až na konci řetězce.

4.3 Uložení dat

Persistentně uložit data v prohlížeči je možné pomocí Web Storage API nebo IndexedDB API.

IndexedDB je databázový systém, který umožňuje ukládání celých JavaScriptových objektů (Flanagan, 2011, s. 705–706). Rozhraní k databázi je asynchronní. Při porovnání s Web Storage API je efektivnější a lze uložit větší množství dat.

PouchDB

PouchDB (PouchDB, 2014) je JavaScriptový databázový systém, který implementuje API databázového systému CouchDB, aby jej bylo možné použít jak v prohlížeči, tak na serveru s Node.js. Systém je nezávislý na CouchDB, ale je možné

vzájemně databáze replikovat. S databázovým serverem CouchDB se komunikuje pomocí HTTP.

Databáze k uložení v prohlížeči využívá IndexedDB API. Protože PouchDB vychází z CouchDB, následující text se věnuje tomuto databázovému systému.

CouchDB

CouchDB patří mezi tzv. dokumentové databázové systémy. Databázový systém zahrnuje JavaScriptový interpret. Pro dotazování dat se používá programovací paradigma MapReduce, kde funkce jsou psány v JavaScriptu (Anderson, 2010c).

Modelování dat Databáze nemají definovaná schémata, strukturu dat lze vynutit pouze validačními funkcemi (Anderson, 2010b). Jednotka dat, která se ukládá, je nazývána pojmem dokument, dále odkazovaným jako JSON dokument kvůli podobnosti s tématem této práce.

JSON je textový formát vycházející z podmnožiny syntaxe objektových literálů JavaScriptu. Datový formát tvoří klíč-hodnota, kde hodnotou může být mimo jiné i další klíč-hodnota, proto je libovolně strukturovaný.

JSON dokument je tak zvaný podle skutečnosti, že model obsahuje většinu dat, která s ním souvisí, přímo v sobě (Anderson, 2010a). Tedy podobně jako skutečný dokument, kde nejsou identifikační čísla osob, ale jejich jména nebo adresy. JSON dokument částečně nebo úplně kopíruje objekt, který se vytváří v JavaScriptové aplikaci (Anderson, 2010b).

Co je uloženo v rámci jednoho dokumentu a co je uloženo jako zvláštní dokument, záleží na dané aplikaci.

Do vlastní databáze by se měly ukládat JSON dokumenty, kterých je příliš mnoho na to, aby byly součástí jiného (Anderson, 2010b). Zvlášť by se měly ukládat i takové, se kterými se manipuluje samostatně.

Ukládání dat Při každém uložení do databáze je předán celý JSON dokument. Po uložení se změní vlastnost _rev, jejíž aktuální hodnotu je nutné použít při dalším uložení. Tím se kontroluje, zda se ukládá změna aktuálně uloženého JSON dokumentu. Revize JSON dokumentu slouží pro replikaci a ukládání, ne k jeho verzování (Anderson, 2010a).

Závěr

Aplikace není postavena na relační databázi, nemá logiku postavenou na zpracováním nebo zobrazováním takovýchto dat. Uložení zde má vedlejší funkci – dočasné uložení umožňující tvorbu i po zavření okna prohlížeče. Z tohoto důvodu je databázový systém bez explicitního schématu, jako je PouchDB, vhodný.

4.4 Closure Tools

Closure Tools je sada nástrojů, která vznikla ve společnosti Google. Součástí je zejména rozsáhlá knihovna Closure Library nebo Closure Compiler optimalizující kód. Aplikace je postavená zejména na správě závislostí Closure Library, a proto je popsána níže.

Správa závislostí v Closure Library

Protože ECMAScript 5 nemá definované moduly, jsou funkce buď globální (jako vlastnost objektu window v prohlížeči) nebo vlastností nějakého globálního objektu. V knihovně Closure je jediný globální objekt goog, kterému jsou přiřazeny vlastnosti představující různé jmenné prostory, např. goog.array je jmenný prostor, do kterého se přiřazují jednotlivé funkce pro práci s poli. (Bolin, 2010, s. 49)

Součástí knihovny Closure Library je systém, který simuluje správu závislostí mezi jednotlivými JavaScriptovými soubory. Závislosti jsou vyjádřeny voláním funkcí goog.provide() a goog.require(). Funkce goog.provide() vytváří jmenný prostor podle předaného řetězce. Správa závislostí funguje na základě zvláštního skriptu (calcdeps.py), který zapíše graf závislostí do JavaScriptového souboru jako volání funkce goog.addDependency(). Tento skript může být připojen na stránku, čímž je do globálního objektu goog zanesen graf závislosti aplikace.

Když je během vykonávání kódu aplikace zavoláno goog.require(), systém přidá do dokumentu HTML značku *script* s atributem *src* odkazující se na soubor, kde je požadovaný jmenný prostor. Tato značka způsobí zpracování souboru a vyhodnocení synchronně (Flanagan, 2011, s. 323), tedy dříve než se vykonávání vrátí na místo zavolání původního goog.require(). HTML dokument, pokud nebyl zcela načten, může být pozměněn v místě volání pomocí metody document.write() (Flanagan, 2011, s. 406). Tím je docíleno, že požadovaný jmenný prostor bude dále v kódu dostupný. Pokud je v tomto vyžádaném souboru požadován jiný, proces se opakuje.

Výhodou výše zmíněného přístupu je, že se nemusí použít server pro načtení závislostí, což zrychluje zpracování stránky. Umístění souboru je vygenerované, takže se může kdykoliv změnit a poté stačí znovu spustit skript. Je to ale kompenzováno nutností buildovacího kroku, kdy se musí vytvořit soubor závislostí. Dynamické přidávání mnoha souborů již není vhodné pro produkci, skript ale umí spojit všechny soubory ve správném pořadí do jednoho souboru nebo spolupracuje s Closure Compiler (Bolin, 2010, s. 46).

4.5 CoffeeScript

CoffeeScript je jazyk, ve kterém je napsán zdrojový kód implementované aplikace. Jazyk se tzv. transpiluje do JavaScriptu. Transpilátor je napsán v CoffeeScriptu, respektive v JavaScriptu. Kompilace během implementace aplikace působila rychle a byla dostatečně rychlá např. pro okamžité spouštění testů. CoffeeScript se od JavaScriptu liší jak syntaxí (inspirována jazyky Ruby či Python), tak vyjadřovacími

možnostmi jazyka. Některé konstrukty jazyka JavaScript není možné v CoffeeScriptu zapsat. CoffeeScript se nicméně překládá přímo do JavaScriptu bez dalších knihoven.

Pomocí syntaxe zpřístupňuje konstrukce např. pro vytvoření tříd a dědičnosti, které v ECMAScript 5 k definování vyžadují relativně rozsáhlý kód. CoffeeScript patří mezi nejvyspělejší transpilátory JavaScriptu (Bolin, 2014).

Mezi výhody jazyka CoffeeScript patří:

- omezuje některé vlastnosti jazyka, které bývají příčinou chyb při nesprávném použití (např. odstranění klíčového slova var)
- přidání konstrukcí, které jinak musí být v JavaScriptu vytvořeny kódem
- syntaktická správnost výsledného souboru po transpilaci

Mezi nevýhody patří:

- nutnost kompilace
- nekonzistence se standardním JavaScriptem, což může být problém např. pro vývoj open-source (Bolin, 2014)
- některá nová funkcionalita (např. ECMAScript 6) nemusí být dostupná, pokud není implementována v transpilátoru
- odlišná syntaxe, nelze kopírovat existující JavaScriptové zdrojové kódy
- relativně méně čitelný vygenerovaný kód, než kdyby byl napsán přímo v JavaScriptu

Pro implementaci byl vybrán kvůli jednoduššímu zápisu objektů respektive metod. Vytváření objektů v CoffeeScriptu je implementováno nejefektivnější technikou viz s. 13 včetně dědičnosti a nerozbíjí operátor instanceOf (Zakas, 2009, s. 190).

V kontextu této práce nevadila nekonzistence s normálním JavaScriptem, protože aplikace byla napsána jedním člověkem. Jediný zásadní nedostatek byl ten, že řádky kódů vyhozených chyb na příkazovém řádku při testování neodpovídaly zdrojovému textu.

4.6 Este framework

Pro vývoj byl použit Este framework, který tvoří projekty *este* (Steigerwald, 2014b), *este-library* (Steigerwald, 2014c) a další, jejichž autorem je český programátor Daniel Steigerwald. Projekt *este* tvoří nástroje pro automatizaci, z kterých byly použity hlavně tyto:

- automatická transpilace při změně pro CoffeeScript a Stylus
- kompilace a minifikace výsledného souboru pomocí Closure Compiler

- automatické spouštění jednotkových testů při změně jednotky
- vytvoření konfiguračního souboru pro DI kontejner

Z knihovny este-library byl použit router, do kterého je možné přidat routovací cesty kompatibilní s webovým frameworkem Express (určeným pro Node.js).

closure-dicontainer

Projekt closure-dicontainer zpřístupňuje DI kontejner v JavaScriptovém projektu využívající správu závislostí z Closure Library. Kontejner využívá anotace pro tzv. silné typy pro Closure Compiler, díky čemuž není nutné kontejner ručně konfigurovat pro každý objekt. Mezi tyto anotace patří @constructor značící konstruktor vytvářející instance daného typu a @interface, který slouží jako typové rozhraní při kompilaci. Tyto typy jsou registrovány automaticky, a proto objekty může kontejner vytvořit sám. Pokud jsou požadovány typy, které nejsou vytvořeny pomocí těchto anotací, musejí být vyřešeny konfigurací kontejneru za běhu programu. Konfigurace probíhá předáváním objektů se specifickými vlastnostmi (with, by) podle vzoru z dokumentace:

> There is a pattern: Resolve A as B with C by D.

A je typ objektu, který má být vytvořen. Pokud je to rozhraní, tak je možné vyřešit závislost určením konkrétního typu B. Pokud již máme existující objekt, můžeme jej předat jako C, opačně předáme konstrukční funkci D (Steigerwald, 2014a).

4.7 React

React je knihovna pro vytváření uživatelských rozhraní.

Principy knihovny

Komponenty Komponenty slouží pro vykreslování dat. Mohou se libovolně zanořovat podle vzoru popsaném v podsekci 4.1 na straně 27. Komponenta je vlastníkem, pokud vytváří jiné (vlastněné) komponenty v její metodě render(). V této metodě je deklarativně nadefinované uživatelské rozhraní. UI je tvořeno existujícími komponentami reprezentující elementy HTML (*React.DOM.div*, *React.DOM.input*) nebo uživatelem definované.

Při vykreslování komponenty se chovají jako funkce, které přebírají *vlastnosti* (viz níže) a vygenerují HTML. Na rozdíl od jiných šablonovacích řešení viz sekce 2.5, vygenerované HTML je pouze virtuální.

Virtual DOM Autoři React považují DOM za pomalý, zvláště ve srovnání s interpretací JavaScriptu (Facebook, 2014a). Proto byl implementován virtuální DOM,

který reprezentuje skutečný. Po virtuálním vykreslení (sestavení JavaScriptového objektu reprezentující nový stav DOM) probíhají porovnávací operace mezi JavaScriptovými objekty knihovny.

Knihovna po porovnání nového a starého virtuálního DOM vypočítá minimální změnu skutečného DOM (Facebook, 2014a). Změna je minimalizována z důvodu již zmíněné rychlosti DOM a neporušení stavu v něm uloženém[7].

Překreslení aplikace Aplikace je celá překreslena ve virtuálním DOM při každé změně stavu.

Vývojáři React zastávají názor, že JavaScript je dostatečně výkonný pro překreslování při každé změně pro většinu aplikací (Facebook, 2014a), natož aby se musela data předem vypočítávat a dočasně ukládat. Veškeré výpočty z dat by měly být prováděny v metodě render(), aby se předešlo nekonzistenci dat. Překreslení podstromu komponenty je možné pro zlepšení výkonu omezit implementováním metody životního cyklu shouldComponentUpdate(), ve kterém by se měly porovnat aktuální a nové vlastnosti komponenty a vrátit pravdivostní hodnotu, zda se má podstrom překreslit.

Rozdělení dat Knihovna rozlišuje data komponent na dva typy. Prvním je **stav** (state), kterého tvoří aktuální stav komponenty a zpravidla nebývá trvale ukládána do databáze. Stav vlastní komponenta a může jej kdykoliv změnit.

Druhým typem jsou **vlastnosti** (properties, zkráceně props), které reprezentují podmnožinu dat aplikace a komponenta je nevlastní. Vlastnosti jsou neměnné a musí být komponentě předány (Facebook, 2014b).

Toto rozdělení by mělo zjednodušit uvažování nad chováním aplikace a není samoúčelné. Nad jedním typem jsou definovány operace, které není možné provádět nad druhým. Komponenta si může změnit stav a tím vynutit překreslení, což není možné pro vlastnosti.

Mezi stav patří zpravidla hodnoty, které se mění v čase. Jsou to hlavně reakce na uživatelský vstup. Ke stavu také patří komunikace se serverem nebo akce závislé na čase.

Jednosměrný tok dat V React je tok dat jednosměrný – od vlastníka do vlastněné komponenty. S libovolně zanořenými komponenty se proces opakuje rekurzivně (Facebook, 2014a). Ve výsledku jsou data aktualizována na všech místech. Při správném použití knihovny se nestane, že by data a z nich odvozené informace byly zobrazeny na jednom místě s jinými hodnotami než na druhém.

Komponentě jsou její data předána právě a pouze při jejím vykreslování. Když se data změní, jsou jí předána nová a dojde k opakovanému vykreslení. Tento přístup má podle autorů knihovny zjednodušit čitelnost předávání dat v aplikaci (Facebook, 2014b).

[7]Psaní ve formulářovém poli by nemělo měnit např. scroll pozici stránky.

Popis komponent

Mezi základní rysy knihovny patří komponentový návrh. Nutnost používat komponenty a skládat je z jiných vychází z toho, že komponenta své vlastnosti nevlastní, ale musí jí být předány odjinud. Komponenta by měla mít dobře navržené rozhraní, které zvýší znovupoužitelnost a omezí opakování kódu (Facebook, 2014a).

Tvorba komponent probíhá pomocí tříd, které definují chování svých instancí. K vytvoření tříd slouží pomocná funkce React.createClass, která přebírá objekt s minimálně jednou metodou render. Tato metoda vrací vždy jednu komponentu knihovny, např. React.DOM.div. Komponenta může implementovat metody životního cyklu, které ovlivňují její chování nebo zpřístupňují DOM, který není v deklarativní metodě render přístupný. Všechny funkce lze volat jako metody na objektu tvořící prototyp komponenty, včetně obsluh událostí díky technice popsané na straně 34.

Doporučené postupy

Vytváření komponent Komponenty by měly být znovu použitelné a měly by dělat jednu věc. Většina komponent by měla být bezstavová. Stav by měl být pouze v té komponentě, kam logicky patří a případně předán do dalších míst jako vlastnost. Vlastnosti by neměly být duplicitní, měly by být vypočítány až když je to potřeba, což je zpravidla v metodě render() (Facebook, 2014b).

Statická verze aplikace Protože je aplikace vykreslena rekurzivně, je možné předat objekt se všemi daty aplikace komponentě na nejvyšší úrovni. Tímto způsobem je možné implementovat velkou část aplikace jako neinteraktivní pouze pomocí jednoho objektového literálu, komponent tvořící HTML a CSS (Facebook, 2014b). Taková implementace vykresluje podmnožinu možných aplikačních dat, další typy dat by měly být nejlépe otestovány. Komponenty je navíc možné použít i samostatně, a proto mohou být dobře otestovány, ať už manuálně prohlížením výsledků v prohlížeči či spouštěním jednotkových testů.

Implementační detaily

Události jsou normalizovány napříč prohlížeči implementováním vlastního systému událostí podle W3C specifikace DOM Level 3 Events[8] (Facebook, 2014c).

Autobinding Knihovna pozměňuje nativní chování událostí a jejich zpracování. V prvním případě automaticky definuje všem posluchačům událostí v komponentě objekt this na hodnotu this dané komponenty. Nativně by při přiřazení posluchače pomocí EventTarget.addEventListener() či definování vlastnosti posluchače elementu měl objekt this být pro daný element (Zakas, 2009, s. 400–401). Protože

[8] http://www.w3.org/TR/DOM-Level-3-Events/

v posluchačích komponenty zpravidla chceme změnit stav nebo zavolat předaný callback, musíme být v oboru platnosti funkce tvořící komponentu. K elementu můžeme přistoupit pomocí this.getElementNode(), jako i v některých metodách komponenty.

Delegování událostí Další použitou technikou v React je delegování událostí, která zefektivňuje využití paměti a práci s DOM. Ke každé komponentě reprezentující HTML element lze přiřadit obsluhu události, React má ale pouze jednoho skutečného posluchače, připojeného na nejvyšší úrovni DOM. To umožňuje přidat obsluhu události na místě jejího vzniku, aniž by docházelo ke kompromisnímu výkonu. Přidávání nebo ubírání (podle toho, zda je komponenta v DOM nebo není) posluchačů modifikuje pouze interní mapování, takže není nutné modifikovat DOM, který je (podle autorů) pomalý. Automatická správa posluchačů se jeví jako správný přístup. Podle knihy (Zakas, 2009, s. 457), jsou nepotřebné obsluhy, které nebyly odstraněny z DOM, hlavním zdrojem problémů s pamětí a výkonem webových aplikací.

5 Editor webových dokumentů

V této kapitole budou definovány požadavky na aplikaci, popsán vývoj a implementace. Text této kapitoly vychází z informací předcházejících kapitol.

5.1 Požadavky

funkční

- nepoužívat značkování
- neformátovat text během psaní
- podporovat tvorbu strukturovaných dokumentů
- rozdělit proces editování, formátování a prohlížení

Aplikace by měla umožnit tvorbu strukturovaných dokumentů bez použití značek a formátování textu během jeho psaní. Strukturovaní pomocí značek zanáší do textu metainformace, které víceméně snižují čitelnost samotného obsahu. Pro přehlednost je nutné obsah dokumentu odsadit a je nutné správně párové značky ukončit. Tyto činnosti odvádějí pozornost od psaní. Formátování v textových procesorech bylo kritizováno na straně 24.

nefunkční

- použít ECMAScript 5
- použít vhodná API prohlížeče
- cílit pro Google Chrome

ECMAScript 5 je verze jazyka ECMAScript, který je implementován ve všech běžných prohlížečích. Má svoje nedostatky, ale je to poslední standardizovaná verze jazyka ECMAScript. Pro vývoj aplikace by mělo být použito jakékoliv vhodné API bez ohledu na podporu v různých prohlížečích. Kompatibilita napříč prohlížeči není v této práci prioritou. Protože je aplikace vytvořena jedním člověkem, nelze ji testovat všemi prohlížeči. Měla by fungovat v prohlížeči Google Chrome.

Funkčními požadavky by mělo být zajištěno to, že aplikace bude vhodná pro běžné uživatele a zároveň podpoří tvorbu struktury dokumentu. K tomu by mělo pomoci i cílové prostředí pro běh aplikace – webový prohlížeč, který je většinou na počítačích nainstalovaný a pokud není, tak je to pouze jedna závislost.

5.2 Popis vývoje

Nejdříve byla vytvořena data pomocí jednoho objektového literálu reprezentující dokument. K němu byla vytvořena statická verze generující HTML viz s. 34. K tomu

byly vytvořeny kaskádové styly, které se poté už výrazně nezměnily. Styly jsou postavené na frameworku Bootstrap. Po rozhodnutí, zda je navržený model vhodný, byly přidány obsluhy událostí, modely objektů a metody nad nimi operující.

Testy byly vytvářeny průběžně během vývoje aplikace, někdy i před implementací dané funkcionality. Psaní těchto testů je díky Este frameworku velice pohodlné, spouští se v terminálovém okně. Otestovány byly především metody pracující nad modelem dokumentu.

Ukládání dat bylo během vývoje nejdříve vyřešeno přes localStorage, jehož použití je velmi jednoduché. Pro daný klíč se uloží hodnota jako řetězec (Flanagan, 2011, s. 589–593). Tento mechanizmus uložení není určený pro větší data, protože přístup k datům je synchronní. Protože se ukládá řetězec, je nutné pokaždé data serializovat do JSON. Po přidání obrázků do aplikace bylo nutné najít nové řešení. Tím bylo IndexedDB, jehož rozhraní je asynchronní, ale nelze použít techniku Promises. Byly hledány nadstavby nad tímto úložištěm, až byl vybrán specifický databázový systém PouchDB.

Během refactoringu se osvědčila jinak méně používaná kontrola typů pomocí Closure Compiler. Jednalo se zejména o případy, kdy jednotkové testy byly v pořádku, ale propojení aplikace bylo chybné kvůli volání staré metody. Kontrola typů včetně jednoduché optimalizace trvala průměrně 15 s.

5.3 Princip aplikace

Tvorba dokumentu je podle požadavku na aplikaci rozdělena do tří částí. V první se dokument edituje, v další prohlíží a v poslední formátuje.

Editování dokumentu je rozděleno na editování logických celků pojmenované jako *bloky*. Terminologie vychází z blokových elementů v HTML[9].

Obsah dokumentu je možné tvořit pomocí odstavců, seznamů a obrázků. Každý z těchto bloků může mít svůj nadpis. Bloky je možné vložit do sekcí, které vytváří hierarchickou strukturu.

Takovéto rozlišení bylo navrhnuto s tím, že by mělo zjednodušit změny v dokumentech. V jiných programech pro tvorbu dokumentů na sebe odstavce přímo navazují a tím mohou splynout v jeden text. Odstavce by měly tvořit ucelené informace, a proto je vhodné je editovat s vizuálním odstupem od ostatních.

5.4 Návrh modelů

Blok zahrnuje určitou část dokumentu. Rozlišuje se mezi strukturními a ostatními bloky. Strukturní se vyznačují tím, že mají v sobě zanořené další bloky. Strukturní blok tvoří typy *app.document.Section* a *app.document.Document*. Mimo strukturní bloky patří všechny ostatní mající svůj obsah. Jsou to odstavce (*app.document.Paragraph*), seznamy (*app.document.List*) a obrázky (*app.document.Image*).

[9] blokové elementy jsou takové, které začínají na novém řádku (Castro, 2012, s. 26)

Abstraktním typem je *app.document.Block*, jehož následující vlastnosti mají všechny bloky:

- _id – identifikátor
- title – nadpis
- type – typ bloku jako řetězec
- comment – komentář
- authorNote – autorská poznámka

Strukturní bloky mají navíc pole *blocks*, ve kterém mají své zanořené bloky. Těmi mohou být znovu strukturní bloky. Největší možná hierarchie je document>section>section>section>paragraph/image/list. Sekce jsou tedy omezeny tak, aby mohly být zanořené maximálně třikrát v dokumentu.

5.5 Architektura aplikace

Architektura vychází z ukázkové aplikace Este frameworku.

Data aplikace jsou spravována ve dvou procedurálních modulech *app.DocumentStore* a *app.EditorStore*. V *app.DocumentStore* se dokument a jeho části vytvářejí a má vyhledávací funkce nad modelem dokumentu. Modul *app.EditorStore* slouží pro změny v dokumentu.

Tyto moduly jsou podle návrhového vzoru Observer, popsaného v podsekci 4.1 pozorovány příslušnými komponentami, aby mohly být překresleny při změně. Dědí goog.events.EventTarget, aby mohly být zdrojem události.

Modul *app.EditorStore* ukládá stav aplikačních dat při každé změně.

Komunikace v aplikaci

Knihovna React vynucuje jednosměrný přenos dat od vlastníků do jejich podstromů. Protože je aplikace relativně jednoduchá, jsou kromě aplikačních dat přenášeny i callback funkce, které mají být zavolány při uživatelském vstupu měnící předaná data. Změna pak neproběhne v komponentě, která data vykresluje, ale v té, která logicky vlastní data.

Pro složitější aplikaci by bylo vhodnější použít elegantnější architekturu, ve které není nutné předávat callback funkce přes několik komponent. Autoři React doporučují architekturu, kterou nazvali Flux (Avram, 2014). Ta se liší od architektury aplikace tím, že má Dispatcher a komponenty nemusí komunikovat pomocí callback funkcí.

6 Diskuze

Teoretickou část o JavaScriptu bylo možné popsat i z monografií, přestože se tyto technologie rychle mění. Webové typografii se věnuje v této práci pouze (Golombisky, 2010), a proto další informace byly doplněny ze zdrojů o počítačové typografii.

V této práci bylo navrhnuto vlastní řešení, které je odlišné od stávajících. Má ale také své nedostatky. Pro splnění požadavků na aplikaci (nepoužívat značkování) není možné přidat metainformace na úrovni řádku. Proto není možné zvýrazňovat části textu a hlavně přidávat odkazy. Další nedostatky nebo možnosti rozšíření aplikace jsou zmíněny níže.

Webová písma S příchodem standardních formátů pro webová písma je možné definovat vlastní písmo a tedy použít i takové písmo, které není v operačním systému nainstalované (Castro, 2012, s. 269–270). Podpora jednotlivých formátů napříč prohlížeči je různá. Pro začlenění vlastních písem by bylo nutné najít typograficky správná písma s patřičnou licencí.

Uložení a export Vytvoření dokumentu a jeho uložení do databáze je nyní omezeno na jeden prohlížeč. Serverová část aplikace by měla vytvořit správu uživatelů a umožnit uložení dokumentu do serverové databáze. Pak by bylo možné dokument editovat na více počítačích. Aplikace by také mohla umožnit exportovat a importovat data ve formátu DocBook (jeho podmnožině), protože interní reprezentace dokumentu je mu velice podobná. DocBook není určený pro běžné uživatele, a proto lze považovat tuto funkcionalitu za doplňkovou.

Elementy Aplikace umožňuje tvorbu pomocí základních prvků, které tvoří většinu dokumentů. Dalšími myslitelnými prvky by mohly být např. zdrojový kód pro počítačovou dokumentaci nebo element pro sazbu matematických vzorců. Oba elementy by bylo možné v editoru zobrazit, protože jejich výsledná podoba je na rozdíl od textu odstavce výrazně odlišná.

Tabulky zpřehledňují informace a patří mezi zásadní elementy dokumentů. Sazba a tvorba je poměrně problematická, patří k nejsložitějším prvkům dokumentů (Rybička, 2011, s. 48–49). Proto nebyly implementovány a je možné je přidat pouze jako obrázek.

Perzistentní datové struktury Každý editor by měl umožnit procházet provedené změny v dokumentu. K tomu by se hodily perzistentní datové struktury, které jsou v JavaScriptu implementované např. v knihovně Immutable.js (Facebook, 2014d). V případě použití této knihovny by bylo možné výrazně optimalizovat překreslení aplikačních dat implementováním metody životního cyklu komponenty shouldComponentUpdate(). Zjištění, zda se struktura změnila, poté spočívá v pouhém porovnání referencí.

7 Závěr

Cílem práce byl popis aktuálního stavu JavaScriptových technologií a vybrání vhodných nástrojů pro implementaci single-page aplikace. Cíl práce byl splněn.

JavaScript má nenahraditelné místo ve vývoji webových aplikací. Protože je nutné pro skriptování na straně klienta používat právě tento jazyk, vznikají pro něj nové nástroje, některé výrazně měnící přístup k jeho použití, ale i nové jazyky určené pro transpilaci do JavaScriptu. V dnešní době lze doporučit pro vývoj single-page aplikací zejména knihovnu React.

Typografie na webu se vyvíjí spíše pomaleji, v rámci CSS3. Kvůli rozličnosti zařízení, které čtou obsah na webu, nebude webová typografie asi nikdy tak precizní jako počítačová sazba určená pro konkrétní formát. Největším přínosem zde bylo patrně rozšíření webových písem.

V teoretické části byly vyjádřeny nedostatky zejména textových procesorů a poté bylo v praktické části navrhnuto vlastní řešení. Aplikace je spíše jednoduchá, ale dostatečně funkční např. pro webové články. Její nedostatky a možnosti řešení byly zmíněny v diskuzi.

Práce pro mě byla velice přínosná, protože jsem si mohl vyzkoušet moderní rozhraní prohlížečů. Prohlížeče s moderními API jako je IndexedDB, Blob apod. považuji za plnohodnotné prostředí pro běh mnoha typů aplikací. Jediné, co zpomaluje běh a vývoj opravdu velkých aplikací je podle mě netypovost JavaScriptu.

8 Reference

8.1 Knihy

Bishop, J. *C#: návrhové vzory*. Vyd. 1. Brno: Zoner Press, 2010, 328 s. ISBN 9788074130762.

Bolin, M. *Closure: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, 2010. ISBN 978-1-4493-8187-5.

Castro, Elizabeth a Bruce Hyslop. *HTML5 a CSS3: názorný průvodce tvorbou WWW stránek*. 1. vyd. Brno: Computer Press, 2012, ISBN 9788025137338.

Flanagan, D. *JavaScript: The Definitive Guide*. 6th ed. Sebastopol, CA: O'Reilly, 2011, xvi, 1078 p. ISBN 05-968-0552-7.

Golombisky, K. and Hagen, R. *White Space is Not Your Enemy*. UK: Focal Press, 2010. ISBN 978-0240812816.

Kočička, P. a Blažek, F. *Praktická typografie*. Vyd. 2. Brno: Computer Press, 2004, xiv, 288 s. ISBN 8072263854..

Martin, Robert C. *Čistý kód*. Vyd. 1. Brno: Computer Press, 2009, 423 s. ISBN 9788025122853.

Mlýnková, Irena a Jaroslav Pokorný. *XML technologie: principy a aplikace v praxi*. 1. vyd. Praha: Grada, 2008, 267 s. ISBN 9788024727257.

Osmani, Addy. *Developing Backbone.js Applications* [online]. 2013 [cit. 2014-12-15]. ISBN 978-1-4493-2824-5. Dostupné z: http://addyosmani.github.io/backbone-fundamentals/.

Přichystal, Jan. *Moderní přístupy v technologii zpracování textů*. Brno, 2006. Disertační práce. Mendelova zemědělská a lesnická univerzita v Brně.

Rybička, Jiří, Petra Čačková a Jan Přichystal *Průvodce tvorbou dokumentů*. 1. vyd. Bučovice: Martin Stříž, 2011, 222 s. ISBN 9788087106433..

Zakas, Nicholas C. *JavaScript pro webové vývojáře*. Vyd. 1. Brno: Computer Press, 2009, 832 s. ISBN 9788025125090.

8.2 Webové stránky

Node.js Nodejs.org [online]. 2014 [cit. 2014-12-18]. Dostupné z: http://nodejs.org/industry/.

PouchDB, the JavaScript Database that Syncs! [online]. 2014 [cit. 2014-12-21]. Dostupné z: http://pouchdb.com/.

Promises/A+ [online]. 2014 [cit. 2014-12-18]. Dostupné z: https://promisesaplus.com/.

TodoMVC [online]. 2014 [cit. 2014-12-11]. Dostupné z: http://todomvc.com/.

8.3 Webové zdroje

Anderson, J. Chris, Jan Lehnardt a Noah Slater. The Core API. In: CouchDB: The Definitive Guide [online]. 2010a [cit. 2014-12-18]. Dostupné z: http://guide.couchdb.org/editions/1/en/api.html.

Anderson, J. Chris, Jan Lehnardt a Noah Slater. Storing Documents. In: CouchDB: The Definitive Guide [online]. 2010b [cit. 2014-12-18]. Dostupné z: http://guide.couchdb.org/editions/1/en/documents.html.

Anderson, J. Chris, Jan Lehnardt a Noah Slater. Getting Started. In: CouchDB: The Definitive Guide [online]. 2010c [cit. 2014-12-18]. Dostupné z: http://guide.couchdb.org/editions/1/en/tour.html.

Avram, Abel *Facebook: MVC Does Not Scale, Use Flux Instead*. In: InfoQ: Software Development News, Videos & Books [online]. 2014 [cit. 2014-12-07]. Dostupné z: http://www.infoq.com/news/2014/05/facebook-mvc-flux.

Bolin, Michael. *Hacking on Atom Part I: CoffeeScript*. In: Bolinfest Changeblog [online]. 2014 [cit. 2014-12-18]. Dostupné z: http://blog.bolinfest.com/2014/08/hacking-on-atom-part-i-coffeescript.html.

ECMA-262 5th edition. *ECMAScript Language Specification*. Geneva: Ecma International, 2011. Dostupné z: http://www.ecma-international.org/ecma-262/5.1/.

Facebook, Inc. *Multiple Components*. In: React [online]. 2014a [cit. 2014-12-18]. Dostupné z: http://facebook.github.io/react/docs/multiple-components.html.

Facebook, Inc. *Thinking in React*. In: React [online]. 2014b [cit. 2014-12-18]. Dostupné z: http://facebook.github.io/react/docs/thinking-in-react.html.

Facebook, Inc. *Interactivity and Dynamic UIs*. In: React [online]. 2014c [cit. 2014-12-18]. Dostupné z: http://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html.

Facebook, Inc. *Immutable.js*. In: GitHub [online]. 2014d [cit. 2014-12-21]. Dostupné z: http://facebook.github.io/immutable-js/.

Glover, Andrew. *Node.js for Java developers*. In: IBM developerWorks [online]. 2011 [cit. 2014-12-18]. Dostupné z: http://www.ibm.com/developerworks/java/library/j-nodejs/index.html.

Google, Inc. *What Is Angular?* AngularJS: Developer Guide [online]. 2010–2014 [cit. 2014-12-15]. Dostupné z: https://docs.angularjs.org/guide/introduction.

Harris, Rich. *What's the difference between React and Ractive?*. In: Blog | Ractive.js [online]. 2014a [cit. 2014-12-11]. Dostupné z: http://blog.ractivejs.org/posts/whats-the-difference-between-react-and-ractive/.

Harris, Rich. *String Templating Considered Harmful.* In: Modern Web [online]. 2014b [cit. 2014-12-18]. Dostupné z: http://modernweb.com/2014/03/24/string-templating-considered-harmful/.

Reichenstein, Oliver. *Web Design is 95% Typography.* In: Information Architects [online]. 2006 [cit. 2014-12-16]. Dostupné z: https://ia.net/blog/the-web-is-all-about-typography-period/.

Shaked, Uri. *AngularJS vs. Backbone.js vs. Ember.js.* In: AirPair [online]. 2014 [cit. 2014-12-15]. Dostupné z: https://www.airpair.com/js/javascript-framework-comparison.

Steigerwald, D. steida/closure-dicontainer. In: GitHub [online]. 2014 [cit. 2014-12-18]. Dostupné z: https://github.com/steida/closure-dicontainer.

Steigerwald, D. steida/este. In: GitHub [online]. 2014 [cit. 2014-12-18]. Dostupné z: https://github.com/steida/este.

Steigerwald, D. steida/este-library. In: GitHub [online]. 2014 [cit. 2014-12-18]. Dostupné z: https://github.com/steida/este-library.

Von Hagen, William. *Producing documentation and reusing information in XML.* In: IBM developerWorks [online]. 2009 [cit. 2014-12-18]. Dostupné z: http://www.ibm.com/developerworks/library/x-reuseinfo2/.

9 Dodatek

Aplikaci bude možné najít online na `https://akela.mendelu.cz/~xmares8/bp/`. Instrukce budou v souboru README.txt v odevzdané příloze do UIS.

9.1 Použité zkratky

API (Application Program Interface) – rozhraní pro aplikace

DOM (Document Object Model) – objektový model dokumentu

SPA single-page aplikace

SVG (Scalable Vector Graphics) – textový formát pro vektorovou grafiku

UI (User Interface) – uživatelské / uživatelská rozhraní

9.2 API prohlížeče

Aplikace využívá tato API nutná pro její běh:

- Web Storage
- IndexedDB
- data URIs

Aplikace byla testována v prohlížeči Google Chrome verze 37. Vygenerovaný dokument lze zobrazit v prohlížeči podporující data URIs.