Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Nástroj pro validaci zápisu kaskádových stylů

Martin Krištof

© 2017 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Martin Krištof

Informatika

Název práce

Nástroj pro validaci zápisu kaskádových stylů

Název anglicky

Cascading style sheets validation tool

Cíle práce

Práce je zaměřena na problematiku zpracování CSS. Cílem práce je navrhnout a implementovat nástroj pro validaci zápisu kaskádových stylů využívající pravidla definovaná podle metodiky BEM (Block Element Modifier). Dílčím cílem bude poskytnout přehled použitých technologií.

Metodika

Teoretická východiska práce budou zpracována analyticko-syntetickým přístupem. Na základě analýzy odborných informačních zdrojů budou popsány technologie a postupy nezbytné pro zpracování aplikace. Při zpracování praktické části budou využity standardní metody softwarového inženýrství.

S využitím znalostí získaných v teoretické části práce bude navržen a implementován nástroj sloužící k validaci zápisu kaskádových stylů na základě pravidel definovaných podle metodiky BEM (Block Element Modifier). Nástroj bude otestován, postup jeho vývoje popsán a zhodnocen a budou navrženy případné možnosti jeho dalšího rozvoje.

Doporučený rozsah práce

35-40 stran

Klíčová slova

CSS HMTL BEM Linter Javascript Module NPM

Doporučené zdroje informací

DOMES, M. *Tvorba internetových stránek pomocí HTML, CSS a JavaScriptu.* Kralice: Computer Media, 2005. ISBN 80-86686-39-6.

FLANAGAN, D. *JavaScript : the definitive guide*. Sebastopol, CA: O'Reilly, 2002. ISBN 0-596-00048-0.

HOGAN, B P. *HTML5 a CSS3 : výukový kurz webového vývojáře.* Brno: Computer Press, 2011. ISBN 978-80-251-3576-1.

MEYER, E A. – MEYER, E A. *CSS : the definitive guide.* Beijing ; Sebastopol, CA: O'Reilly, 2007. ISBN 0596527330.

REYNOLDS, M C. *JavaScript : profesionální řešení.* Brno: Unis Publishing, 1996. ISBN 0789707896.

Předběžný termín obhajoby

2016/17 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 1. 11. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 1. 11. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 21. 11. 2016

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Nástroj pro validaci zápisu kaskádových stylů" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce.
Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 14.3.2017 ___________________________

Poděkování

Poděkování patří panu Ing. Jiřímu Brožkovi, Ph.D. za vedení práce a Robinu Pokornému za ideu a koordinaci metodiky

Obsah

1 ÚVOD 1
2 CÍL A METODIKA 2
2.1 HLAVNÍ CÍL 2
2.2 DÍLČÍ CÍLE 2
2.3 METODIKA 3
2.4 VOLBA TÉMATU 3
3 TEORETICKÁ VÝCHODISKA 4
3.1 HYPERTEXT MARKUP LANGUAGE – HTML 4
3.2 STRUKTURA HTML5 4
3.3 BOX MODEL 5
3.4 KASKÁDOVÉ STYLY – CSS 6
3.4.1 CSS Vlastnosti: 6
3.4.2 Zápis CSS 7
3.4.3 Zápis pomocí obecných HTML značek 7
3.4.4 Zápis pomocí unikátního identifikátoru 8
3.5 BLOCK ELEMENT MODIFIER – BEM 9
3.6 JAVASCRIPT 11
3.7 TYPESCRIPT 12
3.8 JEDNOTKOVÉ TESTOVÁNÍ – UNIT TESTING 13
3.8.1 Data provider 14
3.9 AVA TESTOVACÍ FRAMEWORK 14
3.10 MIDDLEWARE 15
3.11 LINTING 16
3.12 VERZOVACÍ SYSTÉM GIT 16
3.13 GITHUB 17
3.14 PUBLIKOVÁNÍ BALÍČKU DO NPM (NODE PACKAGE MANAGER) 17
3.15 NÁVRH FUNKCIONALITY 18
4 PRAKTICKÁ ČÁST 20
4.1 ZVOLENÉ TECHNOLOGIE 20
4.2 PŘÍPRAVA PROSTŘEDÍ 20
4.2.1 Instalace Node.js 20
4.2.2 Instalace NPM 21
4.3 PACKAGE.JSON 21
4.4 KOMPILÁTOR BABEL 22
4.5 INSTALACE TYPESCRIPTU 22
4.6 AVA 23
4.7 VYTVOŘENÍ TASKŮ 23
4.8 NÁVRH IMPLEMENTACE 25
4.9 ROZDĚLNÍ NA TŘÍDY DLE OOP 25
4.9.1 Třída BemParser a jednotkový test 25
4.9.2 Třida RulesResolver a jednotkové testy 28
4.9.3 Soubor index.ts a jednotkový test 30
4.10 DEMONSTRACE FUNKČNOSTI 31
4.10.1 Ukázka linting procesu bez chyb ve vstupním souboru 32
4.10.2 Ukázka linting procesu s chybami ve vstupním souboru 33
4.10.3 Ukázka doplnění BEM deklarací bez chyb ve vstupním souboru 34
4.10.4 Ukázka doplnění BEM deklarací s chybami ve vstupním souboru 36
5 DISKUSE 37

6 ZÁVĚR **39**
POUŽITÁ LITERATURA **40**
SEZNAM PŘÍLOH **41**

Souhrn

Závěrečná práce se zabývá vytvořením validačního nástroje pro zápis kaskádových stylů (dále CSS) dle metodiky zvané Block Element Modifier (dále BEM).

Obsahuje pak i teoretický úvod do problematiky BEM metodiky, HTML, CSS a použitých technologií při tvorbě tohoto nástroje.

Pro vývoj byl zvolen programovací jazyk Javascript (Node.js) v podobě ECMAScript 2015 a TypeScriptu. Funkční kód je otestován pomocí testovacího Framework AVA. Tyto technologie jsou v práci popsány.

Výsledkem této práce je modul pro aplikaci **CSS-should**, který je pod licencí MIT a vytvořil jej můj bývalý kolega Robin Pokorný.

Samotná aplikace obsahuje exemplární příklady jejího využití a je pojmenována jako CSS-should-plugin-bem.

Klíčová slova

Linter, CSS, BEM, HTML, JS, Nodejs, Javascript, Typescript, Ava

Summary

This bachelor thesis deals with creation of validation tools for writing cascading styles (CSS here in below) in accordance with a methodology called Block Element Modifier (the BEM).

It also contains theoretical introduction to the methodology of BEM, HTML, CSS and technologies used to create this tool.

Programming language Javascript (Node.js) was chosen for the development of the mentioned tool, in the form of ECMAScript 2015 and TypeScript. Function code is tested using the test Framework AVA. These technologies are described in the thesis.

The result of all is a module for application CSS-Should, which is licenced under the MIT and made by my former colleague Robin Pokorný.

The application itself contains typical examples of its use and is named as the CSS-should-plugin-bem.

Keywords

Linter, CSS, BEM, HTML, JS, Nodejs, Javascript, Typescript, Ava

1 Úvod

Výsledkem této závěrečná práce s názvem „***Nástroj pro validaci zápisu kaskádových stylů***“ je projekt pro veřejnou publikaci pojmenován jako **CSS-should-plugin-bem**.
Tento projekt je modulem pod MIT licencí pro aplikaci **CSS-should**.

Tato aplikace slouží pro zpracování a porovnávání pravidel či vlastností kaskádových stylů v jazyce Javascript.

Umožňuje tudíž definovat libovolná pravidla, následně pak jakoukoliv implementaci CSS kontrolovat a hledat chyby. Modul **CSS-should-plugin-bem** definuje pravidla pro zápis kaskádových stylů dle metodiky BEM sloužící k pojmenování CSS tříd.

Jde v podstatě o konvenci pojmenovávání tříd v CSS, která umožňuje zápis kódu lépe číst a orientovat se v něm. Kromě toho umožňuje pak snadnější práci s těmito objekty.

Dohromady pak modul, který je součástí této závěrečné práce, společně s aplikací **CSS-should** vytváří komplex, který umožňuje uživatelům kontrolovat správnost zápisu (linting) tříd CSS dle výše zmíněné metodiky BEM.

Podobné validátory kódu momentálně ve světě tvorby CSS nejsou příliš vyvíjeny. Pár jich ale existuje, nicméně pro mé potřeby nejsou vyhovující nebo jsou příliš komplexní pro efektivní využití.

V závěru práce pak srovnávám výhody a nevýhody oproti již existujícímu konkurenčnímu nástroji.

Tato skutečnost mi vnukla onu myšlenku vytvořit tento nástroj pro budoucí použití nejen pro studentské účely, ale právě pro profesní účely.

V praxi totiž výše zmiňovanou metodiku pro jmennou konvenci pojmenovávání CSS tříd hojně používám. Nejen já, ale i spousta mých kolegů ve firmě. Rozhodně by se tedy mně i jim tento nástroj pro vývoj hodil. Díky této komunitě vývojářů pak mohu v budoucnu odhalit nedostatky a případně doplnit nějakou další funkcionalitu.

2 Cíl a metodika

Práce je rozdělena na dvě části: teoretická a praktická.

V první teoretické části bude představena problematika HTML a kaskádových stylů (CSS), metodiky pro zápis CSS tříd BEM. Dále pak úvod zápisu v jazyce Javascript a jeho podmnožiny pro efektivnější práci s ním – TypeScript.

Tato část také obsahuje principy pro izolovaní testování funkcionality pomocí jednotkových testů (Unit testing). S tím souvisí i představení použitého testovacího Frameworku AVA. Definice pojmu Middleware. Nakonec pak vysvětlení lintingu a pojmu verzovacího systému GIT.

Druhá praktická část je zaměřena na analýzu řešení a implementaci samotné aplikace pomocí nástrojů uvedených v teoretické části této práce. Prochází jednotlivé části implementace a dokumentuje teoretická východiska souvisící s teoretickou částí práce.

2.1 Hlavní cíl

Hlavním cílem je analýza, návrh a implementace validačního nástroje pro zápis kaskádových stylů dle metodiky Block Element Modifier. Záměrem vytvoření této aplikace je pak vyplnění nedostatku validačního nástroje tohoto konkrétního typu v oblasti open source[1]. Práce má dále za cíl poskytnout vhled do použitých technologií.

2.2 Dílčí cíle

- charakteristika kaskádových stylů a možnost zápisu pro HTML
- využití jazyku Javascript na serveru (Node.js), především pak jeho modifikace TypeScriptu
- popis metodiky BEM
- charakteristika middleware

[1] Definice Open source na https://opensource.com/resources/what-open-source

2.3 Metodika

Záměrem této práce je vytvoření funkčního modulu pro již existující aplikaci. Jedná se tedy o návrh nové samostatné aplikace zosobňující svá specifika, která ovšem musí implementovat rozhraní middleware. Jednotlivé kroky řešení vychází z metodiky Unified Process[2].

Jednotlivé etapy lze rozdělit na kroky:

- Studium BEM metodiky
- Definice požadavků, které middleware musí splňovat pro integraci
- Navržení funkcionality
- Implementace a testovaní
- Vytvoření příkladu pro demonstraci funkčnosti
- Publikování middlewaru do NPM[3]

2.4 Volba tématu

Toto téma bylo zvoleno na základě absence podobného nástroje pro kontrolu zápisu CSS tříd dle BEM metodiky. Inspiroval mne můj bývalý kolega z práce Robin Pokorný, který mi tuto myšlenku nadhodil. Na našem projektu Jobs.cz se tato metodika hojně používá. Aplikace tudíž obohatí jak přímo projekt, na kterém pracuji, tak i jakéhokoliv uživatele, který si jej pod MIT[4] licencí opatří. Například přes npmjs.com.

---

[2] Popis Unified Process na: http://fei.mtrakal.cz/materialy_public/7.semestr/%5B2010-2011%5DINPSW_Simerda/prednasky/02_UPIntroduction.pdf
[3] O NPM na: https://docs.npmjs.com/getting-started/what-is-npm
[4] Definice MIT licence: http://www.webopedia.com/TERM/M/MIT_license.html

3 Teoretická východiska

3.1 Hypertext markup language – HTML

Je standardní značkovací jazyk pro tvorbu dokumentů, jehož prvky tvoří základní stavební kameny všech webových stránek a webových aplikací. (W3C)

První specifikace byla vytvořena v roce 1990 Timem Berners-Leem, který jazyk vymyslel. Vycházel z jazyka SGML (Longman, 1998) . Postupně vznikly další specifikace HTML 2, 3, 4 a 5.

HTML definuje strukturu dokumentu a umožňuje tak pomocí značek:

- vytvořit nadpisy, odstavce textů, tabulky, seznamy, obrázky
- pohybovat se díky hypertextovým odkazům po různých webových stránkách
- navrhovat formuláře pro konání transakcí se vzdálenými službami, vyhledávání informací, objednávaní zboží, vytvoření rezervace či zanechání vzkazu
- vkládat tabulky spread-sheets, filmové či zvukové soubory apod. (W3C)

3.2 Struktura HTML5

HTML dokument se skládá ze 3 částí:

- řádek obsahující HTML verzi (Doctype)
- hlavičku (Head) - obsahuje meta tagy, titulek a zdroje na kaskádové styly, skripty apod.
- tělo (Body) - obsahuje bloky, které jsou pak ve stránce viditelné (W3C)

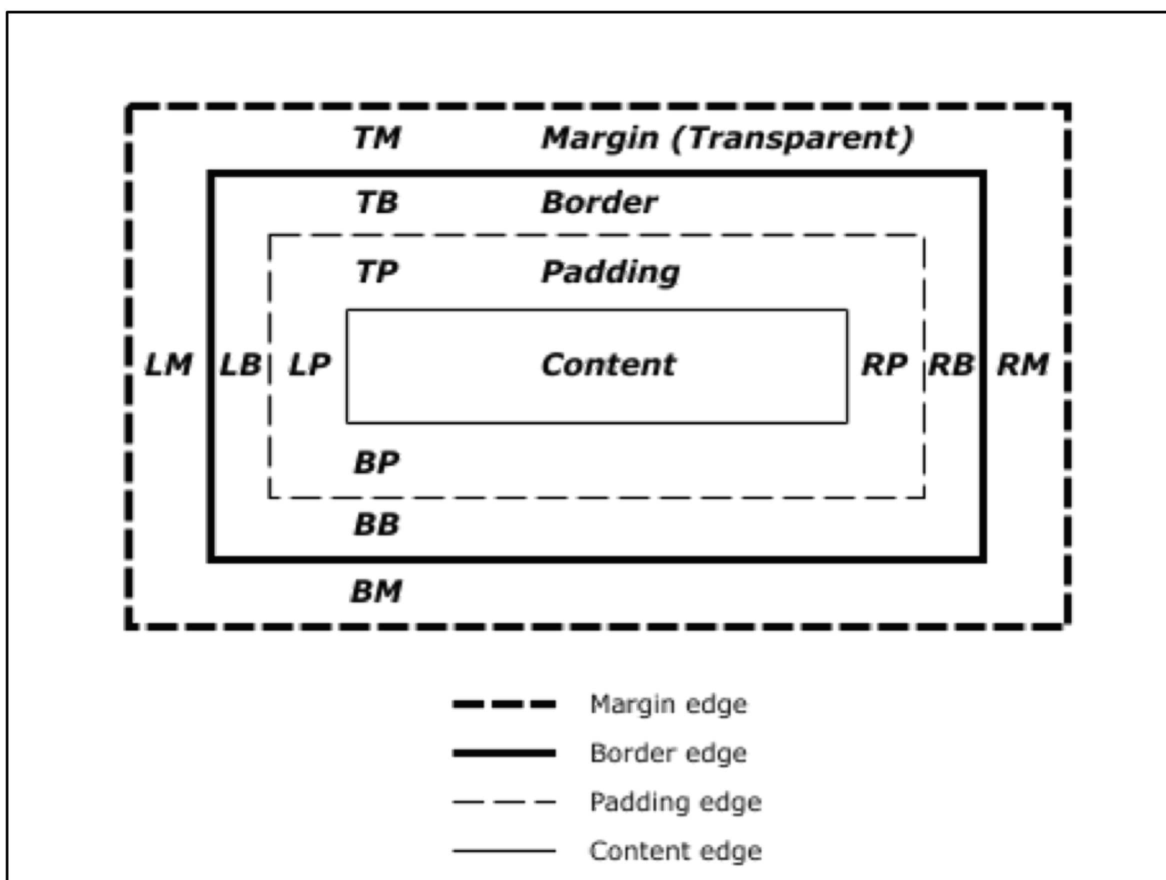
```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
<div>
<p>content</p>
</div>
</body>
</html>
```

Obrázek 1: Příklad struktura HTML

3.3 Box Model

Box model popisuje obdélníkové boxy, které se generují kolem jednotlivých elementů ve stromu dokumentu stanovených podle vizuálního formátovaného modelu. (W3C)

- vnitřní okraj (content) – definován výškou a šířkou elementu
- padding – vrstva mezi content a rámečkem (border) - pokud je padding nulový, tak je jeho velikost shodná s content boxem
- rámeček (border) - platí podobné pravidlo jako výše
- vnější okraj (margin) - transparentní vrstva



Obrázek 2: Box model

3.4 Kaskádové styly – CSS

Je deklarativní jazyk pro obohacení webových dokumentů o grafickou podobu prvků (písmo, barvy, odsazení, pozicování, apod.) (Bos)

První specifikace CSS vznikla ve W3C v roce 1996 (W3C). Pomocí tohoto jazyka je tedy možné definovat vizuální podobu webové stránky. Vlastnosti prvků lze deklarovat pro jednotlivé HTML elementy, třídy, identifikátory či data-atributy.

3.4.1 CSS Vlastnosti:

stylování:

- pozadí – background-color, background-image, ...
- text – color, text-align, text-shadow, ...
- písmo – font-family, font-style, font-size, ...
- odkazy – a:link, a:visited, a:hover, a:active
- seznamy – list-style-type, list-style-image, list-style-position, list-style-type

okraje:

- styl okrajů – border-style
- tloušťka okrajů – border-width
- barva okrajů – border-color
- lze nastavovat i jednotlivé okraje (left, top, right, bottom)
- vnější odsazení – margin
- vnitřní odsazení – padding

pozicování:

- float (left, right)
- flex-box
- grid

(T.)

3.4.2 Zápis CSS

Stylování HTML přímo uvnitř dokumentu pomocí je velmi nepřehledné, pro vývoj neudržitelné a neefektivní.

3.4.3 Zápis pomocí obecných HTML značek

```
<p style="color: red;">Red text</p>
```

Obrázek 3: CSS v HTML

Proto je doporučeno vytvářet samostatné soubory obsahující CSS, které se pak připojí v hlavičce HTML dokumentu. Deklarované vlastnosti se dají aplikovat obecně na HTML značky:

- body – tělo dokumentu
- p – odstavec
- h1, h2, h3, h4, h5, h6 – jednotlivé úrovně nadpisů
- ul – nečíslovaný seznam
- ol – číslovaný seznam

- li – prvek seznamu
- div – blokový oddíl
- span – řádkový oddíl
- table – tabulka
- tr – řádek tabulky
- td – buňka tabulky
- a – odkaz

```
body {
    background-color: red;
}

p {
    text-align: center;
    color: white;
}
```

Obrázek 4: CSS pro obecné HTML elementy

3.4.4 *Zápis pomocí unikátního identifikátoru*

Dále je možno použít místo obecných značek unikátní identifikátor ID v HTML. Tím, že je identifikátor unikátní a dle standardu by se měl vyskytovat v dokumentu pouze jednou, je toto použití velmi konkrétní a není znovu použitelné.

```
<p id="text">Text</p>
```

Obrázek 5: Unikátní identifikátor v HTML

Zápis vlastností pak vypadá následovně, důležitý je znak "#":

```
#text {
    text-align: center;
    color: white;
}
```

Obrázek 6: Zápis CSS pro ID

Nejčastějším řešením, jak cílit na elementy, které mají být stylovány, je použití CSS tříd. Třídy se mohou opakovat.

```
<p class="text">Text</p>
```

Obrázek 7: Třída pro CSS v HTML

V CSS syntaxi se pro třídy používá znak “.”.

```
.text {
    text-align: center;
    color: white;
}
```

Obrázek 8: Zápis tříd v CSS

Tato varianta zápisu je nutností pro možnost využití metodiky, které se tato práce věnuje.

3.5 Block Element Modifier – BEM

Jde o konvenci pojmenování pro psaní objektově orientovaných kaskádových stylů.

BEM byl vymyšlen v ruské firmě Yandex[5]. Hlavní myšlenka je v rozdělení tříd do těchto tří kategorií:

blok	**.block**
element	**.block__element**
modifikátor	**.block__modifier**

Potomek bloku je element, modifikátor je pak variantou konkrétního elementu. Jde tedy o logické rozdělení zanořených elementů (selektorů) uvnitř.

[5] Odkaz na web společnosti: https://www.yandex.com/

Tato metodika pojmenování se hojně používá ve světě vývoje, jelikož je tak kód na front-endu[6] přehlednější a srozumitelnější. CSS třídy jsou totiž rovněž viditelné uvnitř HTML kódu a tím se právě ona čitelnost projevuje. Tento způsob velmi usnadňuje týmovou práci na projektu.

Kromě této domény přidává tato metodika ještě benefit pro rozšiřování zápisu již existujících tříd. K tomu se hodí modifikátory, které nějakým způsobem upravují již definovanou komponentu. Například může modifikátor měnit barvu, písmo, pozici a další libovolné CSS vlastnosti. (Vsevolod Strukchinsky)

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
<ul class="nav nav--hidden" role="navigation">
<li class="nav__item">
<a href="/">Úvod</a>
</li>
<!-- … -->
</ul>
</body>
</html>
```

Obrázek 9: Ukázka BEM metodiky v HTML na neseřazeném seznamu

- **.nav** je blok (název komponenty)
- **.nav--hidden** je modifikátor (varianta komponenty)
- **.nav__item** je element (potomek komponenty)

```css
.nav {
    display: block;
}
.nav--hidden {
    display: none;
}
.nav__item {
    padding: 10px;
}
```

Obrázek 10: Ukázka CSS pro BEM metodiku k příkladu na obrázku 9

[6] Definice slova front-end: http://it-slovnik.cz/pojem/frontend

V CSS je pak vidět, že třída ***nav—hidden,*** nastavuje neviditelnost celé navigace. Tím, že se používá obecné pojmenování je pak hned z kódu jasné, k čemu je daná třída určena. Nesmí se ale pod jednu třídu logicky kombinovat více vlastností, aby pak vlastnosti třídy nedělaly více funkcionality, než je syntakticky záhodno.

Čili by například třída ***nav—hidden*** neměla obsahovat CSS vlastnosti pro velikost písma, či barvy atd. Tyto vlastnosti by pak měla definovat nějaká jiná třída, která by mohla být znovupoužitelná a aplikována na jiných elementech.

Ku příkladu, pokud chceme definovat nějakému elementu červenou barvu textu, tak vytvoříme další třídu a té tuto vlastnost nastavíme. Třídu ale pojmenujeme nějak obecně jako ***text—warning,*** aby se vystihlo chování třídy, která upozorňuje v textu na něco varujícího. Jde o další využití modifikátoru čili předpokládáme, že už existuje třída ***text***, která definuje vlastnosti textu (velikost, šířku písma, styl písma apod.). Pokud by ona třída **text** neexistovala, jde o špatné použití modifikátoru, jak je zmíněno výše. Modifikátor může být definován pouze pro již existující element.

BEM metodiku lze celkem jednoduše aplikovat a lehce se ji naučit, tudíž se z ní stává standard (Michálek)

Preprocesory[7] pro psaní CSS pak velmi dobře umí usnadnit BEM zápis. Jde o zanořování dalších tříd do sebe.

3.6 Javascript

Je skriptovacím programovacím jazykem, který běží jak na straně serveru, tak na straně klienta v prohlížeči. Na serveru je potřeba pro běh Framework Node.js.

V roce 1996 byl standardizován jako ECMAScript. Hojně se používá na statických webech pro větší dynamičnost (animace). Nicméně v poslední době se rozvíjí webové aplikace právě díky Javascriptu (Čápka). Nejnovější platná specifikace se jmenuje ECMAScript 2016 [8].

[7] O preprocesorech na: http://www.vzhurudolu.cz/blog/12-css-preprocesory-1

[8] Definice ECMAScript na: https://www.ecma-international.org/ecma-262/7.0/

Javascript je netypový jazyk, je objektově orientovaný, třídy fungují na bázi Prototypes[9].

V projektu byl pro implementaci použit právě serverový Javascript ECMASript 2015 známý také jako ES6.

```
class Person {
    constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }

    getFullName() {
        return this.name + ' ' + this.surname;
    }
}
```

Obrázek 11: ukázka zápisu třídy v ECMAScript 2015

3.7 TypeScript

Je nadmnožinou Javascriptu. Je to open source nástroj od společnosti Microsoft, který obohacuje samotný Javascript o funkce, které ho přibližují syntakticky k Javě[10] či dotNetu[11]. Přidává datové typy, interface a možnost psát dle paradigmatu OOP[12]. Nicméně stále je nutná znalost Javascriptu pro pochopení toho, jak interpret funguje.

Samotný kód, který je nutné psát do souborů s koncovkou ***"ts"*** jsou kompilovány do standardního Javascriptu. Během kompilace[13] lze přijít na chyby v kódu a ty případně odladit, což je nesmírná výhoda. Programátor tedy může chybu objevit dříve, než se samotný kód provede. Toto velmi šetří čas při vývoji. (Microsoft)

V projektu je použita verze 2.0.

---

[9] Prototypes popis: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype
[10] Popis jazyku Java: https://techterms.com/definition/java
[11] Popis jazyku ASP.Net: https://techterms.com/definition/aspnet
[12] Popis a definice OOP: https://techterms.com/definition/oop
[13] O kompilaci: http://it-slovnik.cz/pojem/kompilace

K tomu, aby kompilátor mohl pracovat s konkrétními typovými definicemi je nutné tyto typy doinstalovat. Kupříkladu pokud je v projektu použit balíček **css**[14], který slouží k parsování[15] CSS souborů je záhodno nainstalovat soubor definicí pro tento balíček. Postup je jednoduchý, stačí spustit příkaz pro stažení balíčku pomocí NPM z repozitáře DefinitelyTyped[16].

Díky těmto definicím umožňuje vývojové prostředí – IDE[17] našeptávat názvy metod apod., což velmi usnadňuje vývoj.

```
interface Person {
    firstName: string;
    surname: string;
}

class Student {
    private fullName: string;

    constructor(private person: Person) {
        this.fullName = person.firstName + ' ' + person.surname;
    }

    public getFullName() : string {
        return this.fullName;
    }
}

const person : Person = {firstName: 'John', surname: 'Doe'};
const student : Student = new Student(person);
console.log(student.getFullName());
```

Obrázek 12: ukázka zápisu třídy v TypeScriptu

3.8 *Jednotkové testování – Unit testing*

Je druh testování software. Testuje se zde funkcionalita na úrovni jednotek tedy metod (funkcí). Testy jsou izolované a rychlé. Lze tak otestovat velké množství různých

[14] Repozitář projektu: CSS https://github.com/reworkcss/css
[15] Vysvětlení pojmu parsování na: http://kam.mff.cuni.cz/~kuba/vyuka/programovani/xa-ver/parsovani.html
[16] Zdroj repozitáře na: http://definitelytyped.org/
[17] Popis IDE pojmu: http://searchsoftwarequality.techtarget.com/definition/integrated-de-velopment-environment

vstupů metody. Vytvoření unit testů k funkcionalitě lze ještě před naimplementováním samotného testované kódu.

Takové technice se pak říká test driven development (TDD) (Agiledata, 2013 str. 2016). Umožňuje programátorovi velmi efektivně navrhnout veřejné rozhraní. Tento projekt tak byl i částečně vytvořen.

Další technikou je Behavioral driven development (BDD)[18]. Zde nejde o pořadí psaní unit testů, nýbrž o pojmenování testovacích scénářů dle toho, jak se aplikace chová. Nikoliv podle toho, co přesně dělá. (Dan North & Associates)

Příklad:

testShouldResolveUrlIfResponseNotFound

Jednotkové testy jsou velmi užitečné při tvorbě funkcionality i při jejím dalším udržováním a rozšiřováním. Lze velmi jednoduše poznat, zdali úprava způsobila chybu či nestabilitu aplikace. Záhodno by bylo mít vše pokryto těmito testy, ale to vždy nejde. Nadstavbou Unit testů jsou pak integrační testy, ty testují, jak název napovídá, integraci jednotlivých služeb a modulů v aplikaci.

3.8.1 Data provider

Je datová struktura obsahující více prvků, které nesou data pro testování. Je to nejčastěji vícerozměrné pole nebo objekt implementující Iterator interface[19], aby bylo možné nad prvky iterovat. Výhodou použití data-provideru je, že lze ušetřit spoustu společného kódu a měnit pro jednu testovací metodu parametry a očekáváné výstupy. Je to rozhodně efektivnější a rychlejší než psát pro každý test-case zvláštní metodu. (PHPUnit)

3.9 AVA testovací Framework

Je jedno-vláknový open source testovací Framework napsaný v Node.js. Slouží k unit testování funkcionality psané v Javascriptu. Umožňuje paralelní běh v

[18] Vysvětlení pojmu BDD na: https://www.agilealliance.org/glossary/bdd/
[19] Definice Iteratoru: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators

nezávislých procesech a tím roste rychlost provádění testování v izolovaném prostředí. (Sorhus)

```
import test from 'ava';

test(t => {
    t.deepEqual([1, 2], [1,2]);
});
```

Obrázek 13: ukázka syntaxe kódu v AVA

Lze jej nainstalovat pomocí balíčkovacího manažeru NPM, Yarn[20] či manuálně z Githubu:

`https://github.com/avajs/ava`

3.10 Middleware

Tento pojem definuje určitou část kódu (knihovnu) třetí strany, kterou lze obohatit danou aplikaci. Je to alternativa plug-inu[21] na serverové části aplikace. Řeší samostatně několik problému a připravuje pro “mateřskou” aplikaci vstupy. (IT Slovník)

Ku příkladu tato závěrečná práce je middlewarem pro aplikaci **CSS-should**. Cílem tohoto middlewaru je připravit pravidla, se kterými pak hostitelská aplikace umí pracovat. Middleware musí implementovat veřejné rozhraní aplikace. Toto rozhraní slouží k umožnění komunikace se softwarem třetích stran.

CSS-should-plugin-bem tudíž musí implementovat metody:

- ***preprocess*** (vrací volání funkce[22])
- ***name*** (vrací datový typ[23] string)

[20] Definice Yarn: https://yarnpkg.com/lang/en/
[21] O významu slova plug-in: https://techterms.com/definition/plugin
[22] Definice funkce v Javascriptu: https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Functions
[23] Přehled o datových typech v Javascriptu: http://www.itnetwork.cz/javascript/zaklady/javascript-js-tutorial-datove-typy-retezce-cisla-funkce

3.11 Linting

Linting představuje proces, který pomáhá hledat a odstraňovat chyby z kódu programátorů. Jako první analyzuje celý kód a automaticky v něm nalezne a vyznačí chyby. Může jít například o chyby v syntaxi, překlepy apod. Velmi tím usnadňuje práci s kódem ve chvíli, kdy nefunguje. Zároveň tvoří ideální podmínky pro jeho opravu. (IT Slovník)

CSS-should aplikace dokáže do CLI[24] (konzole) vypsat nalezené chyby.

3.12 Verzovací systém GIT

Představuje verzovací systém, který umožňuje vývojářům efektivně pracovat společně na jednom či více projektů najednou. Uchovává diference mezi jednotlivými verzemi v čase a tím pádem umožňuje sestavit jakýkoliv stav kódu v minulosti. Popisky k jednotlivým změnám pak ještě popisují, proč vůbec ke změnám došlo. (GIT SCM)

Díky Gitu lze vytvářet vývojové větve, vytvářet ***tagy*** (verze), skládat větve dohromady (***merge*** a ***rebase***), vytvářet změny (***commit***), publikovat změny (***push***), stahovat změny (***fetch a pull***).

Veškerý kód se pak uchovává v repositáři. Slouží tak zároveň jako záloha.

Pro vývoj aplikace je její verzování důležité a k tomu slouží proces „tagování" (verzování). Tím je zabezpečeno, že si pak uživatel může stáhnout a nainstalovat konkrétní verzi aplikace, pokud zrovna nepotřebuje z nějakého důvodu tu nejnovější. Verzování aplikace se provádí jednoduchým příkazem:

```
git tag <verze>
```

A následně verzi zveřejnit přes příkaz:

```
git push –tags
```

[24] Vysvětlení pojmu https://techterms.com/definition/command_line_interface

Doporučení způsob pro přidání verzí je zmíněn v podkapitole 3.14. Tato metoda je použita i v případě tohoto projektu (Git).

3.13 GitHub

Tato webová služba umožňuje hostovat Git repositáře na svém cloudu. Díky přehlednému webovému grafickému rozhraní (GUI) lze přehledně sledovat obsah repozitáře, seznam verzí a změn.

Důležitou funkcí je možnost přidávat kód do projektu po schválení vlastníka repozitáře (***Pull request***[25]). (Github)

3.14 Publikování balíčku do NPM (Node Package Manager)

Aby mohl být balíček publikován do balíčkovacího systému NPM je záhodno mít správně definován soubor package.json[26]. Primárně se musí správně definovat aktuální verze balíčku dle sémantického verzování (SemVer)[27].

Jakmile je tento postup správný, stačí si vytvořit účet na NPM. Pak jako uživatel v konzoli CLI v repozitáři projektů k publikování balíčku postačí zadat příkaz:

```
npm publish
```

Samozřejmě, že v průběhu vývoje vznikají další verze projektu. Pro navýšení verze již existujícího balíčku je zapotřebí vykonat následující příkazy:

```
npm update <nová verze>
npm publish
```

Takto se nová verze dostane do cílového repozitáře uvnitř NPM (npmjs).

---

[25] O pull requestech na GitHubu: https://help.github.com/articles/about-pull-requests/

[26] K čemu se používá package.json soubor: https://docs.npmjs.com/getting-started/using-a-package.json

[27] Definice sémantického verzování: http://semver.org/

Aktuální verze mého balíčku je stejná jako na Githubu a nachází se na webu npmjs:

`https://www.npmjs.com/package/css-should-plugin-bem`

3.15 Návrh funkcionality

K tomu, aby aplikace CSS-should mohla s vytvářeným middlewarem fungovat je zapotřebí přidat do zparsovaného stromu CSS souboru experimentální CSS vlastnosti i s hodnotami.

Middleware obdrží strom CSS, který si projde, nalezne CSS třídy, které jsou psány dle BEM na základě modifikátorů a bloků. K těmto pak do objektu jejich deklarací připojí výše zmíněné vlastnosti jako další hodnoty deklarace.

Objekt deklarace pro komunikaci s CSS-should vypadá takto:

```
{
    type: 'declaration',
    property: 'x-should',
    value: `match '${param}'`,
}
```

Obrázek 14: Deklarace pro komunikace s CSS-should aplikací

Kde ***$(param)*** obsahuje vzor tříd, které daná CSS třída musí mít jako předka dle BEM.

Například CSS třída modifikátoru **tabs_item—link** musí mít jako předka některé z hodnot:

`.tabs * nebo .tabs__item`

Jelikož je těchto hodnot více, tak se v objektu skládají pod sebe:

```
{
    type: 'declaration',
    property: 'x-should',
    value: `match \'.tabs *\'`,
},
{
    type: 'declaration',
    property: 'x-should',
    value: `match \'.tabs__item\'`,
}
```

Obrázek 15: Ukázka skládání více deklarací

Tedy v CSS souboru musí být definovány CSS třídy **tabs** a **tabs__item**. Protože **.tabs__item** musí mít jako předka právě CSS třídu **tabs.**

Pokud v CSS stromu některé z tříd chybí, zápis této třídy je dle BEM metodiky nevalidní a program vyhodnotí zápis jako špatný.

4 Praktická část

4.1 Zvolené technologie

K návrhu projektu jsem se rozhodl pro UML[28], které je standardem. Při zvolení programovacího jazyka jsem dlouho neváhal. Jelikož je hostovaná aplikace **CSS-should**, do které je tento middleware vytvořen, napsána v Javascriptu, respektive Node.js [29], tak i tato práce je napsaná ve stejném softwarovém systému a programovacím jazyce.

K verzování jsem se rozhodl pro Git a samotný repozitář je hostovaný právě na Githubu a jeho obsah je možné vidět na URL:

`https://github.com/MartinKristof/css-should-plugin-bem`

4.2 Příprava prostředí

K vývoji je potřeba nainstalovat aktuální verzi Node.js a balíčkový manažer NPM (Node Package Manager).

Do kořenové složky přidat prázdné adresáře "***src***", "***test***".

Nelze zapomenout také na README.md soubor, kde uživatelé najdou informace o balíčku s návodem instalace a použití. Dále je zde uvedena historie verzí aplikace. Obsah souboru je zapsán pomocí značkovacího jazyku Markdown[30].

4.2.1 Instalace Node.js

Pro jednotlivé platformy se instaluje různým způsobem, ale z webové stránky projektu lze stáhnout aktuální verzi pro všechny známe platformy. V mém případě jsem použil balíček pro operační systém OS X od Apple.

[28] Popis UML: http://proteus.fav.zcu.cz/~mautner/Pt/UML_richta.pdf
[29] Definice Node.js: https://nodejs.org/en/about/
[30] O jazyce Markdown: https://daringfireball.net/projects/markdown/

4.2.2 Instalace NPM

Nejjednodušší cestou k instalaci NPM je skrze CLI. Stačí se tedy dostat v adresářové struktuře do adresáře s projektem a spustit příkaz:

```
npm install npm@latest -g
```

Tímto se NPM nainstaluje poslední verze globálně do systému.

4.3 Package.json

K tomu, aby mohl být middleware publikovaný v balíčkovacím manažeru NPM, je nutno definovat soubor package.json.

Dále se v něm deklaruje název balíčku, verze, typ licence, autoři, cesta k repozitáři (v tomto případě na GitHubu) a závislosti (***dependencies***, v našem případě i závislosti pouze pro vývoj – ***devDependencies***), které pak lze jednoduše nainstalovat pomocí příkazu v konzoli:

```
npm install
```

Příkaz stáhne přes internet z výše zmíněného zdroje konkrétní závislosti a nainstaluje je do nově vytvořené složky **node_modules.**

Závislosti se ale nemusí psát ručně do souboru, ale vkládají se tam automaticky, pokud při instalaci konkrétního balíčku použijeme přepínač ***—save***.

```
npm install název-balíčku —save
```

Ještě je dobré definovat cestu k souboru index.js (main), protože se nenachází v kořenovém adresář, ale v adresáři „***src***“.

Seznam použitých závislost:

- ava (vysvětleno výše v samostatné podkapitole)
- babel-cli (vysvětleno níže v samostatné podkapitole)
- babel-preset-latest (vysvětleno níže v samostatné podkapitole)

- chalk[31]
- css
- diff[32]
- dále pak definition types pro výše zmíněné závislosti

4.4 Kompilátor Babel

Jelikož jsem se rozhodl psát aplikaci v ECMAScript 2015, tak je nutné nainstalovat vývojovou závislost Babel[33] a její podmnožiny. Ta se stará o kompilaci nové syntaxe do podoby ECMAScript 5, které rozumí všechny webové prohlížeče a verze Node.js.

```
npm install –save babel-register
```

Samotná kompilace se spouští příkazem:

```
babel src -d lib -q
```

- **src** – zastupuje parametr zdrojové složky, která obsahuje skripty ke kompilaci
- **lib** – tento parametr definuje složku, kam se budou výsledné zkompilované soubory ukládat
- přepínač **d** – příznak k tomu, že se zkompilované soubory exportují do jiné složky (v tomto případě do složky **lib**)

4.5 Instalace TypeScriptu

Jak je zmíněno v teoretické části, rozhodl jsem se využít k zápisu kódu syntaxi TypeScriptu. I tento balíček je potřeba nainstalovat do projektu:

```
npm install –save typescript
```

Aby ale kompilátor Typescriptu věděl, jaké soubory má překládat a do jakého výstupu je má překládat slouží konfigurační soubor tsconfig.json.

[31] O Chalk: https://github.com/chalk/chalk
[32] Popis Diff závislosti: https://github.com/kpdecker/jsdiff
[33] O Babelu na: http://babeljs.io/

Kompilace je dostupná pod:

```
tsc -p ./
```

4.6 AVA

Rozhodl jsem se implementovat TDD metodikou, tudíž je instalace testovacího Frameworku AVA nezbytná:

```
npm install —save ava
```

V package.json přidat pak vzor jmen testovaných souborů, v tomto projektu končí suffixem[34] Test.js.

Běh kódu nad unit testy lze spustit příkazem:

```
ava —v
```

4.7 Vytvoření tasků

V provádění úloh výše zmiňovaných záleží na pořadí. Nejdříve je nutno přeložit kód z Typescriptu a až potom lze buď kompilovat kód pomocí Babelu do produkčního adresáře nebo lze spustit kontrolu unit testů.

Aby se jednak omezilo spouštění úloh ve špatném pořadí a zkrátil se zápis příkazů, je možno díky NPM v souboru package.json zapsat úlohy tzv. ***tasky*** v sekci "***scripts***".

- **`compile:`**

  `npm run tsc && npm run build`

  Zavolání kompilace Typescriptu a následně Babelu

- **`build`**

  `babel src -d lib -q`

  Kompilace Babel z ECMAScriptu2016 do ECMAScript 5

[34] Suffix definice: https://en.oxforddictionaries.com/spelling/prefixes-and-suffixes

- **`tsc:`**

  `tsc -p ./`

  Přeložení Typescript scriptů

- **`test:`**

  `ava -v`

  Spuštění jednotkových testů s příznakem verbose, čili vypisuje případné chyby detailněji do konzole

- **`lint:`**

  `npm run compile && babel-node ./example/example-lint.js`

  Spustí task ***build***, následně ***tsc*** a provede linting ukázkového CSS souboru, který neobsahuje chyby dle BEM metodiky, pro demonstraci funkčnosti

- **`lint-with-errors:`**

  `npm run compile && babel-node ./example/example-lint-with-errors.js`

  Obdobné jako předchozí jen s tím rozdílem, že je zdrojový CSS soubor s chybami dle BEM metodiky, tudíž lze pak vypozorovat chyby, které v souboru jsou

- **`declarations:`**

  `npm run compile && babel-node ./example/example-declarations.js`

  Tento task zajistí, že se k CSS vlastnostem ze vstupního CSS souboru přidají BEM definice, se kterými pak může pracovat aplikace **CSS-should**, výstup se pak uloží do nového souboru a vypíše do konzole (vstupní soubor neobsahuje chybné pojmenování tříd podle BEM konvence)

- **`declarations-with-errors:`**

  `npm run compile && babel-node ./example/example-declarations-with-errors.js`

  Stejná úloha, jen s tím rozdílem, že vstupní soubor obsahuje nesprávný zápis CSS tříd dle BEM metodiky

Na základě výše definovaných tasků (úloh) je umožněno pomocí jednoho příkazu spustit vše, co je pro běh aplikace potřebné a ve správném pořadí:

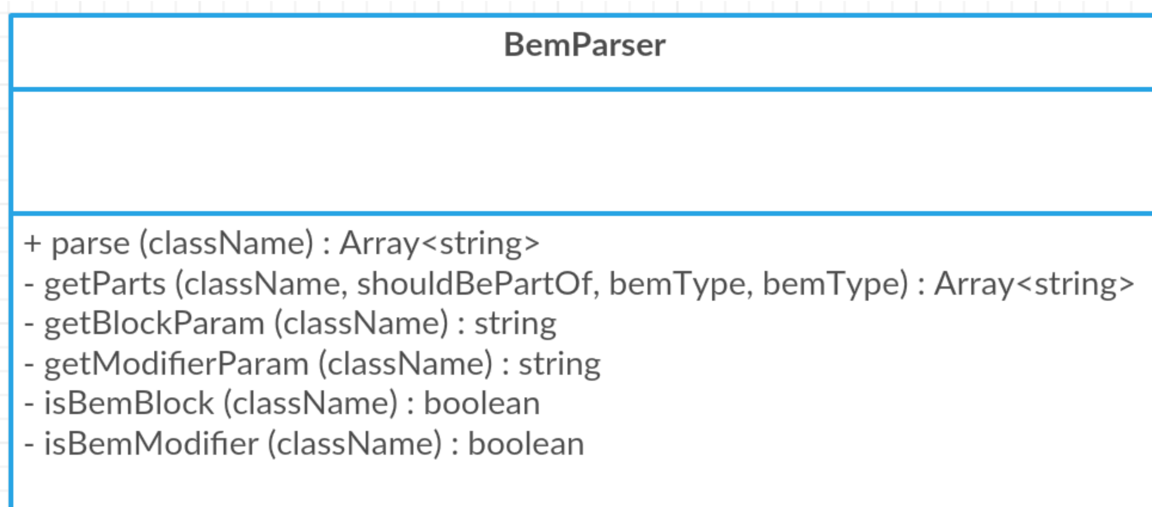
```
npm run compile
```

4.8 Návrh implementace

Protože middleware musí implementovat rozhraní, které je zmíněno výše, rozhodl jsem se logiku, která generuje pravidla, zapouzdřit do samotné třídy.

4.9 Rozdělní na třídy dle OOP

4.9.1 Třída BemParser a jednotkový test



Obrázek 16: UML diagram třídy BemParser

Tato třída dokáže, díky své veřejné metodě **parse**, určit, které selektory jsou CSS třídy zapsané dle BEM metodiky, na základě předaných selektorů. Jednotlivé detekce jsou implementovány pomocí regulárních výrazů[35]. Uvnitř třídy je logika zajišťující zjišťování toho, které CSS třídy musí být předky předaných CSS tříd, jak bylo popsáno v kapitole s navržením funkcionality.

Příklad vrácených hodnoty metody **parse,** která vrací pole řetězců**:**

[35] Vysvětlení regulárních výrazů v Javascriptu na URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp

- [.tabs *] pro `.tabs__link:hover`
- ['.tabs *', '.tabs__link'] pro `.tabs__link–active`
- [.item] pro `a.item--bold:active`
- ['.item *', '.item__icon'] pro `i.item__icon.item__icon–spinning`
- [] pro `.button`

Další příklady na obrázku č.18.

```
private static getBlockParam(className : string) : string {
    const blockPattern : RegExp = /\S+?(?=__)/g;

    return className.match(blockPattern) && '.'+
    className.match(blockPattern)[0] + ' *';
}

private static getModifierParam(className : string) : string {
    const modifierPattern : RegExp = /\S+?(?=--)/g;

    return className.match(modifierPattern) && '.' +
    className.match(modifierPattern)[0];
}

private static isBemBlock(className : string) : boolean {
    const blockPattern : RegExp = /\w+__\w+/g;

    return className.match(blockPattern) ? true : false;
}

private static isBemModifier(className : string) : boolean {
    const modifierPattern : RegExp = /\w+--\w+/g;

    return className.match(modifierPattern) ? true : false;
}
```

Obrázek 17: Metody pro detekci BEM zápisu

Implementace této třídy byla učebnicovým příkladem k použití TDD metodiky. Nejprve jsem si vytvořil testovací soubor **BemParserTest.ts** do složky **test.** Pomocí data provideru jsem definoval test case a edge case [36]. Následně jsem začal

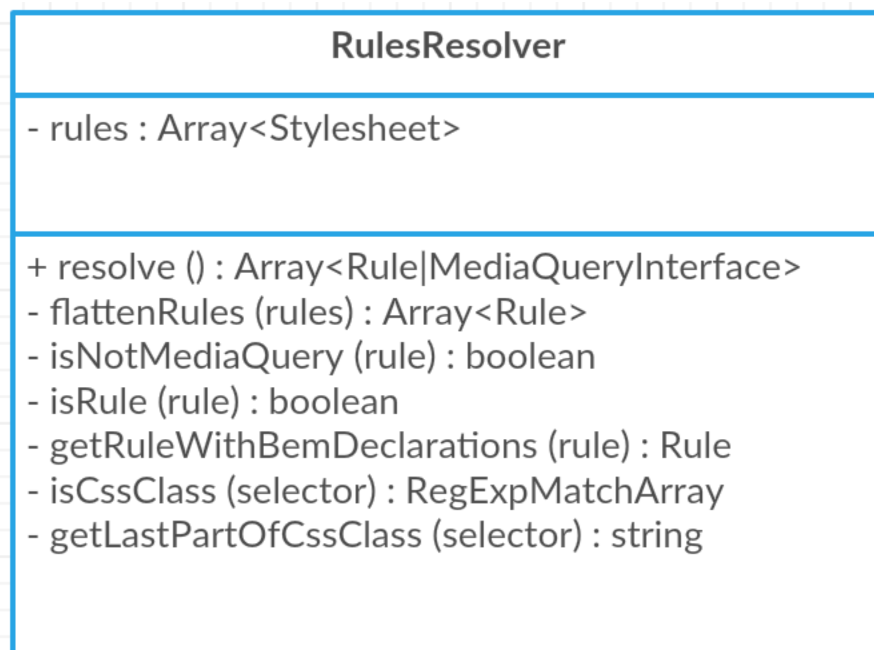
[36] Popis problematiky Test cases: https://www.teachengineering.org/activities/view/uno_doesitwork_lesson01_activity1

implementovat funkcionalitu tak, že začaly procházet všechny jednotkové testy. Postupně jsem tedy doplňoval kód, než prošly všechny testy.

```
    {
        description: 'class on anchor with pseudo class',
        className: '.tabs__link:checked',
        expectedSub: ['.tabs *'],
    },
    {
        description: 'list with modifier',
        className: 'ul.menu--primary',
        expectedSub: ['.menu'],
    },
    {
        description: 'modifier with entire element',
        className: 'i.item__icon.item__icon--spinning',
        expectedSub: ['.item *', '.item__icon'],
    },
    {
        description: 'modifier without parent element',
        className: '.item--bold',
        expectedSub: ['.item'],
    },
    {
        description: 'modifier on pseudo classes',
        className: 'a.item--bold:active',
        expectedSub: ['.item'],
    },
```

Obrázek 18: Ukázka testovacích dat pro jednotkový test BemParserTest

4.9.2 Třida RulesResolver a jednotkové testy



Obrázek 19: UML diagram třídy RulesResolver

Instance této třídy umožňují přidat k již existujícímu stromu pravidel vlastní pravidla čili ty, které definují správný zápis dle metodiky BEM. Nejprve jsou pravidla a selektory převedeny na plochou strukturu pomocí metody **flatten**.

Pro ilustraci jde o následující zápis a je převeden do nové podoby:

```
.tabs__link--xxx,
.tabs__link--itemize,
.headline {
    background-color: #1b6494;
}
```

Obrázek 20: Zápis jedné CSS vlastnosti pro více CSS tříd

```
.tabs__link–xxx {
    background-color: #1b6494;
}
.tabs__link–itemize {
    background-color: #1b6494;
}
.headline {
    background-color: #1b6494;
}
```

Obrázek 21: Nová plochá struktura zápisu jedné CSS vlastnosti pro více CSS tříd

Teprve k této struktuře jsou BEM pravidla připojena, jen pokud se jedná o CSS třídy. Pokud nejsou předané selektory CSS třídy, tak jsou ignorovány (ID apod.) a vůbec se nekontrolují, protože je to zbytečné. BEM metodika se totiž vztahuje pouze na zápis CSS třídy nikoliv identifikátorů.

V kódu se volá metoda **parse,** jež je definována v přechozí kapitole. Tím jsou předána do třídy hodnoty.

Ošetřena jsou i CSS pravidla uvnitř media–query[37]. Celou tuto funkcionalitu zapouzdřuje veřejná metoda **resolve.**

```
{
            type: 'rule',
            selectors: ['.tabs__item--active'],
            declarations: [
                {foo: 'bar'},
                {
                    type: 'declaration',
                    property: 'x-should',
                    value: `match \'.tabs *\'`
                },
                {
                    type: 'declaration',
                    property: 'x-should',
                    value: `match \'.tabs__item\'`
                }
            ],
            position: {boo: 'bazz'}
},
```

Obrázek 22: Ukázka vrácených pravidel do zparsovaného stromu CSS v Javascriptu

[37] Definice media-query na URL: https://www.w3.org/TR/css3-mediaqueries/

```
public resolve() : Array<RuleInterface|MediaQueryInterface> {
    let result : Array<RuleInterface|MediaQueryInterface> = [];

    this.rules.forEach((rule : any) : void => {
        if (RulesResolver.isNotMediaQuery(rule) &&
            RulesResolver.isRule(rule)
           ) {
            let rules = this.flattenRules([rule]);

            rules.map((rule) => {
                result.push(this.getRuleWithBemDeclarations(rule));

            });

            return;
        }

        rule.rules = this.flattenRules(rule.rules);
        rule.rules = rule.rules.map((rule) => {
            return this.getRuleWithBemDeclarations(rule);
        });

        result.push(rule);
    });

    return result;
}
```

Obrázek 23: Ukázka metody resolve ve třídě RulesResolver

Jednotkový test ověřuje, že se funkcionalita uplatňuje skutečně jen u CSS tříd, nikoliv u ID selektorů. Včetně kontroly, že vše funguje i u pravidel uvnitř bloků media–query..

4.9.3 Soubor index.ts a jednotkový test

Tento soubor je klíčový pro celý middleware. Obsahuje veřejné API[38] pro komunikaci s aplikací **CSS-should**. Uvnitř souboru se implementují metody **preprocess** a **name.** První zmiňovaná metoda pouze přijme pravidla a pomocí třídy **RulesResolver** přibalí k těmto pravidlům nová pravidla, které už obsahují BEM definice. Metoda **name** vrací pouze název balíčku.

[38] Definice pojmu API: https://techterms.com/definition/api

```
import RulesResolver from './RulesResolver';

export const preprocess : Function = (
    ctx : Stylesheet,
    next : Function
) => : Function {
    ctx.stylesheet.rules = (new RulesResolver(ctx.stylesheet.rules)
)
        .resolve();

    return next();
};

export const name : string = 'BEM';
```

Obrázek 24: Ukázka metod v souboru index.ts

Jednotkový test jen ověřuje, že se skutečně volá metoda **next.** S tím pak ještě ověřuje jméno middlewaru.

4.10 Demonstrace funkčnosti

Jelikož je aplikace **CSS-should** stále ve vývoji a nemohu ji k ukázce funkčnosti ještě použít, rozhodl jsem se vytvořit modul, který ji simuluje ve dvou oddělených krocích. Umístil jsem jej do nové složky s názvem ***example***.

První část se skládá z obohacování CSS pravidel o ta vlastní, které definují pravidla pro BEM, tak aby jim rozuměla aplikace **CSS-should**.

Druhá část pak tyto pravidla zkoumá a umí rozpoznat chybu v zápisu dle BEM metodiky.

Napsal jsem si tedy testovací CSS soubory jako zdrojové, se kterými pak celý program pracuje. Tyto soubory obsahují jak CSS třídy, tak ID či definice uvnitř media-query.

Tento kód už není nutné testovat jednotkovými testy a postačí pouze manuální test, jelikož jde jen o volání **CSS-should-plugin-bem** middlewaru, který je kompletně jednotkovými testy ověřen.

4.10.1 Ukázka linting procesu bez chyb ve vstupním souboru

Exemplární příklad pro kontrolu vstupního souboru a nalezení, v tomto případě nenalezení, chybějících tříd uvnitř souboru. Tyto CSS třídy způsobují, že některé zbylé třídy pak nejsou validně zapsány dle BEM metodiky.

Úloha se provede spuštění příkazu v CLI konzoli:

```
npm run lint
```

Výstup je vypsán na obrazovku do konzole následovně:

```
Linting file <cesta k souboru>

No errors detected!
```

Obrázek 25: Výstup výsledku do konzole

Jak jsem zmiňoval, vstupní CSS soubor je napsán validně čili obsahuje potřebné CSS třídy dle BEM metodiky.

```css
.tabs {
  border: 1px solid #1b6494;
}
.tabs__item--abc {
  background-color: red;
}
.tabs__item {
  display: block;
}
```

Obrázek 26: Úryvek validně definovaného vstupního CSS souboru

Na obrázku výše je vidět, že CSS třída **tabs_item—abc** ***(modifikátor)*** musí podle metodiky mít definovaného předka, a to právě třídu **tabs_item.** Ta se v souboru nachází ***(element)*** a její předek, který musí uvnitř souboru také být, je třída **tabs** ***(blok).*** Ona třída je viditelná hned na první řádku ukázky.

Všechny třídy jsou tedy validně zapsány a kontrola tudíž proběhla v pořádku a chyba nebyla nalezena.

4.10.2 Ukázka linting procesu s chybami ve vstupním souboru

Obdobný případ jako předchozí, nicméně vstupní soubor s kaskádovými styly obsahuje nevalidně zapsané třídy dle BEM metodiky. Některé třídy tam tudíž nejsou definovány.

Úloha se provede spuštění příkazu v CLI konzoli:

```
npm run lint-with-errors
```

Výstup je vypsán na obrazovku do konzole následovně:

```
Linting file <cesta k souboru>
There are some bad class names according to BEM detected!
You have to specify this classes:
.tabs__link, .tabs__header, .tabs__item
```

Obrázek 27: Výstup do konzole v případě lintingu nevalidního CSS souboru

Uživatel se tímto způsobem dozví, jaké třídy v souboru chybí a je potřeba je definovat, aby byl soubor validní.

```
.tabs {
  border: 1px solid #1b6494;
}

.tabs__item--abc {
  background-color: red;
}

.tabs__link--xxx,
.tabs__link--itemize,
.headline {
  background-color: #1b6494;
}
```

Obrázek 28: Ukázka nevalidního zápisu CSS tříd uvnitř vstupního souboru

Třída **tabs_link *(element)*** v souboru chybí, která je potřeba pro správné použití jejích ***modifikátorů*** (**.tabs_item—abc** a **.tabs_link—itemize**). Stejně tak je tomu i v případě ***modifikátoru*** **tabs_item—abc**, pro který není definována ***element*** třída **tabs_item.** CSS třída **tabs** sice v souboru je, ale není blokem pro žádnou ***element*** třídu, protože zmiňované ***element*** třídy nejsou v souboru nalezeny.

Tato a předchozí úloha jsou pouze nadstavbou a jsou vytvořeny pro poukázání, že linting funguje. Opravdová funkcionalita pro výpis chyb bude implementována v projektu CSS-should, kde bude určitě výstup detailnější.

4.10.3 Ukázka doplnění BEM deklarací bez chyb ve vstupním souboru

Pomocí tohoto příkladu lze demonstrovat, jak opravdu funguje výsledek této práce. Program načte ze vstupního CSS souboru selektory a jejich definice a obohatí je o BEM definice, jak je popsáno v kapitole o vývoji třídy RulesResolver.

Proces se zahájí příkazem:

```
npm run declarations
```

Výsledek zpracování je vypsán jak do konzole, tak i do nového výstupního souboru. Vytvořený soubor obsahuje stávající deklarace a selektory a nově i BEM definice.

Výstup vypsaný v CLI konzoli zobrazuje obsah současného souboru a barevně pak doplněné a odebrané řádky kódu. Zeleně nové řádky a červeně řádky vymazané. Šedou barvou jsou pak označeny řádky, které se nezměnily.

```
@media (max-width: 480px) {
  .tabs__link--xxx,
  .tabs__link--active,
  .tabs__header--active,
  .tabs__link--xxx {
    color: red;
    x-should: match '.tabs *';
    x-should: match '.tabs__link';
  }

  .tabs__link--active {
    color: red;
    x-should: match '.tabs *';
    x-should: match '.tabs__link';
  }

  .tabs__header--active {
    color: red;
    x-should: match '.tabs *';
    x-should: match '.tabs__header';
  }

  #id2 {
    color: red;
  }

  .tabs__item {
    color: blue;
    x-should: match '.tabs *';
  }
}
```

Obrázek 29: Ukázka výstupu do konzole po přidání BEM deklarací

Stejný obsah je uložen do výše zmiňovaného výstupního souboru, samozřejmě bez barev. Do konzole je ještě vypsána cesta k nově vytvořenému souboru:

```
css-should declarations will be written into file /css-should-plugin-bem/example/output/example-out.css
```

Tento výstup pak bude použit právě pro CSS-should aplikaci, která už s ním bude dále operovat a vypíše uživateli informaci o tom, kde přesně udělal chybu. V tomto případě mu bude sděleno, že je vše v pořádku.

4.10.4 Ukázka doplnění BEM deklarací s chybami ve vstupním souboru

Stejný průběh úlohy jako v přechozí podkapitole, ale vstupní soubor obsahuje nevalidně zapsané CSS třídy podle BEM.

Spuštění procesu se uskuteční zadáním příkazu do konzole:

```
npm run declarations-with-errors
```

Znovu je vypsán barevný výstup s nově přidanými řádky s BEM definicemi a odebranými řádky do konzole včetně cesty k novému souboru, do kterého je rovněž výstup uložen.

Aplikace CSS-should bude na základě tohoto výstup hlásit chyby uživateli s podrobným popisem, kde je problém a jaké třídy je potřeba dodefinovat do vstupního souboru.

Ukázku pro tuto úlohu není třeba poskytovat, protože se v zásadě ve výstupu velmi neliší.

5 Diskuse

Testování aplikace prokázalo, že nástroj funguje spolehlivě, nicméně je zde prostor pro zlepšení. Například po stránce konfigurace, kde jsou rezervy. Uživatel by si mohl definovat vlastní oddělovače bloků a modifikátorů, namísto pevně daných. Tento pevný způsob byl vybrán na základě četnosti používání v komunitě. Další prostor pro optimalizaci by mohlo být zvýšení rychlosti algoritmu, a to na základě použití perzistentních datových struktur (immutable.js)[39].

Dále připadá v úvahu integrovat tuto aplikaci do internetového prohlížeče v podobě plug–inu. Bohatě by stačila podpora v Chrome. Tudíž by nebylo třeba instalovat Node.js a zadávat cestu ke vstupním souborům. Uživateli by byl přímo v konzoli prohlížeče vypsán výsledek kontroly vstupního souboru, který by si stáhl rovnou z navštívené webové stránky. K tomu bych se mohl dostat v budoucnu.

V porovnání s podobnými nástroji, jako je například ***post-css-bem-linter***[40] , je můj modul poněkud odlehčený. Oba ale potřebují pro svoji funkčnost mateřskou aplikaci, kam se pak jako middlewary vloží. Ovšem hostující aplikace pro výše zmíněný middleware – ***PostCSS*** je kolos oproti **CSS-should**, v tom bych viděl výhodu mé varianty. Není potřeba instalovat velké množství souborů. Naopak komunita PostCSS je určitě větší, a to je naopak výhoda pro druhou stranu.

Ačkoliv je výsledkem této práce nástroj, který by měl zamezovat špatnému zápisu dle metodiky, nenaučí programátory BEM správně používat, respektive správně pojmenovávat jednotlivé bloky a modifikátory. Toto je záležitost zcela mimo, která se nedá jen tak naprogramovat. Ne nadarmo se říká, že jedno z největších umění na programování je právě pojmenovávání.

Také bohužel neumí nástroj odhalit špatně sémanticky pojmenované prvky, tím je myšleno to, že například programátor pojmenuje třídu odstavce jako tlačítko apod. Vše už je pak v rukách a hlavě vývojáře.

[39] Popis a podrobnosti o knihovně Immutable.js na: https://facebook.github.io/immutable-js/

[40] Popis balíčku na URL: https://github.com/postcss/postcss-bem-linter

Po dokončení aplikace **CSS-should** se exemplární příklady mohou vymazat, jelikož už nebudou pro demonstraci potřeba. Tato událost je v horizontu několika měsíců a pak se můj middleware stane plnohodnotným nástrojem pro vývojáře.

6 Závěr

Cílem této závěrečné práce bylo vytvořit nástroj pro kontrolu zápisu kaskádových stylů dle metodiky BEM. Tato část byla splněna v praktické části této práce.

Jejím výsledkem je pak vytvořená aplikace **CSS-should-plugin-bem**, která je dostupná pod MIT licencí na Githubu, NPMjs a je i součástí přílohy k závěrečné práci.

V sekci teoretická východiska byl splněn dílčí cíl, a to popsat použité technologie. Došlo tedy k osvětlení pojmů HTML, CSS, Javascript, TypeScript, linting, middleware a samotné definice Block Element Modifier metodiky pro pojmenovávání CSS tříd.

Dále jsou zmíněny i principy jednotkového testování, které byly použity pro kontrolu funkčnosti samotné aplikace i za pomocí použitého testovacího Frameworku AVA.

Definován je také postup publikování do výše zmíněných zdrojů, kde je aplikace dostupná veřejnosti.

Ukázka použití je dostupná a byla splněna rovněž v praktické části práce. Je tedy možno demonstrovat funkcionalitu závěrečné práce v praxi. S tím je popsáno, jak bude aplikace integrována s hostovanou aplikací **CSS-should**.

V diskuzi jsem pak zhodnotil funkčnost aplikace, porovnal jsem ji s konkurenčními nástroji a nastínil další budoucnost projektu, jako je například integrace s internetovým prohlížečem.

Použitá literatura

- **Čápka, David.** Javascript. *IT network.* [Online] [Citace: 13. 12 2016.] http://www.itnetwork.cz/javascript/zaklady/javascript-tutorial-uvod-do-javascriptu-nepochopeny-jazyk.
- **Agiledata. 2013.** Agiledata. *Agiledata.* [Online] 19. 4 2013. http://agiledata.org/essays/tdd.html.
- **Bos, Bert.** CSS. *W3C.* [Online] [Citace: 13. 12 2016.] https://www.w3.org/Style/CSS/ .
- **Dan North & Associates.** Introducing BDD. *Dan North & Associates.* [Online] [Citace: 13. 2 2017.] https://dannorth.net/introducing-bdd/.
- **GIT SCM.** About. *GIT SCM.* [Online] [Citace: 12. 2 2017.] https://git-scm.com/about.
- **Github.** About. *Github.* [Online] [Citace: 13. 2 2017.] https://guides.github.com/activities/hello-world/.
- **IT Slovník.** IT Slovník. *Middleware.* [Online] [Citace: 11. 2 2017.] http://it-slovnik.cz/pojem/middleware.
- **—.** Linting. *IT Slovník.* [Online] [Citace: 19. 2 2017.] http://it.slovnik.cz/pojem/linting/?utm_source=cp&utm_medium=link&utm_campaign=cp .
- **Longman, Addison Wesley. 1998.** A history of HTML. *W3C.* [Online] 1998. [Citace: 14. 12 2016.] https://www.w3.org/People/Raggett/book4/ch02.html .
- **Microsoft.** Introduction. *TypeScript.* [Online] [Citace: 12. 2 2017.] https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html.
- **Michálek, Martin.** BEM. *Vzhůru dolu.* [Online] [Citace: 13. 12 2016.] http://www.vzhurudolu.cz/prirucka/bem .
- **PHPUnit.** Writing Tests for PHPUnit. *PHPUnit.* [Online] [Citace: 6. 2 2017.] https://phpunit.de/manual/current/en/writing-tests-for-phpunit.html.
- **Sorhus, Sindre.** AVA. *Github.* [Online] [Citace: 10. 2 2017.] https://github.com/avajs/ava.
- **T., Pitner.** Informační systém Masarykovi Univerzity. *Informační systém Masarykovi Univerzity.* [Online] [Citace: 14. 12 2016.] is.muni.cz/el/1433/jaro2012/PB138/um/web/foil27.html.
- **Vsevolod Strukchinsky, Vladimir Starkov and contributors.** GetBEM. *GetBEM.* [Online] [Citace: 10. 2 2017.] http://getbem.com/.
- **W3C.** Box model. *W3C.* [Online] [Citace: 17. 12 2016.] https://www.w3.org/TR/CSS2/box.html .
- **—.** HTML & CSS. *W3C.* [Online] [Citace: 17. 12 2016.] https://www.w3.org/standards/webdesign/htmlcss .
- **—.** The global structure of an HTML document. *W3C.* [Online] [Citace: 17. 12 2016.] https://www.w3.org/TR/html401/struct/global.html .
- **—.** W3C HTML. *W3C.* [Online] [Citace: 14. 12 2016.] https://www.w3.org/html/ .

Seznam příloh

- Příloha 1 – CD se zdrojovými kódy