Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Nástroje pro zefektivnění práce s CSS

Bakalářská práce

Autor: Jan Thér
Studijní obor: Aplikovaná informatika

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

Hradec Králové Duben 2019

Prohlášení:

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

V Hradci Králové dne 24.4.2019 Jan Thér

Poděkování:

Děkuji vedoucí bakalářské práce Mgr. Daniele Ponce, Ph.D. za metodické a svědomité vedení práce, a především za notnou dávku trpělivosti.

Anotace

Cílem bakalářské práce je popsání, srovnání a zhodnocení přístupů k zefektivnění vývoje frontendu webových stránek a aplikací ve spojitosti s kaskádovými styly, a to zejména za pomoci CSS procesorů a CSS metodik. V teoretické části je čtenář seznámen s konceptem vývoje webových stránek, CSS procesory, jejich funkcemi a rozdíly mezi preprocesory a postprocesory. Dále je pak seznámen s CSS metodikami, a to zejména z pohledu jejich možnosti navázání na zmíněné nástroje. Praktická část je zaměřena na porovnání procesorů PostCSS a Stylecow na základě testů při běžném vývoji webových stránek a následného návrhu možného způsobu propojení všech zkoumaných nástrojů a postupů s cílem nejvyššího zefektivnění přístupu k vývoji frontendu.

Annotation

Title: Tools for Increasing Efficiency of Work with CSS

The aim of the bachelor thesis is to describe, compare and evaluate approaches for more efficient development of front-end of web pages and applications in connection with cascading styles, especially with the help of CSS processors and CSS methodologies. In the theoretical part, the reader is acquainted with the concept of web development, CSS processors, their functions and differences between preprocessors and postproceossors. Then the reader is acquainted with CSS methodologies, especially from the perspective of their ability to be combined with mentioned tools. The practical part is focused on the comparison of PostCSS and Stylecow processors on the basis of tests in common website development and subsequent proposal of possible combination of all examined tools and procedures with the aim of the most effective approach to front-end development.

Obsah

1 Úvod

Internet se stal neodmyslitelnou součástí našeho každodenního života. Mezi jeho základní pilíře patří webové stránky, které jsou pro běžného konzumenta zdrojem téměř veškerého obsahu. S vlastní webovou prezentací může v současnosti přijít snad každý. K tomu většinou vede seznámení se základními webovými technologiemi a postupy. Mnohdy to u nich i končí a přístup k vývoji webových stránek není zdaleka tak efektivní, jak by mohl být.

Základním kamenem každého webu jsou technologie CSS a HTML. Zejména v souvislosti s prvním zmíněným v posledních letech vznikají nástroje a postupy, které umožňují přístup k vývoji webu výrazně zefektivnit a zjednodušit. Tato práce si dává za cíl tyto technologie čtenáři představit, popsat jejich použití, a především navrhnout a doporučit cesty, jak tyto nástroje co nejefektivněji zapojit do jeho současného pracovního procesu.

Pro dosažení tohoto cíle bude uživatel nejdříve seznámen se základními webovými technologiemi CSS ve spojení s HTML, jelikož jejich znalost je nezbytně nutná pro pochopení nástrojů schopnosti CSS rozvíjejících, které budou představeny v následujících částech. Těmito nástroji budou CSS procesory. V první řadě dojde k představení preprocesorů a jejich schopností, které přidávají CSS zcela novou funkcionalitu. Tyto schopnosti budou následně reflektovány v souvislosti s praktickou částí, ve které dojde ke srovnání postprocesorů PostCSS a Stylecow, které se preprocesorům v mnohém podobají, ale v mnohém se také liší. K jejich vybrání došlo právě na základě těchto podobností, jelikož v současnosti představují jakousi pomyslnou konkurenci preprocesorů. Kontrastem k těmto webovým technologiím pak budou vybrané metodiky, které přistupují k CSS zcela odlišným způsobem, než procesory a jsou tak jejich zajímavou alternativou.

V praktické části dojde ke srovnání vybraných technologií. Kritéria tohoto srovnání budou zvolena na základě jejich užitečnosti pro běžného uživatele a bude se jednat zejména o funkce, které dané nástroje nabízí. Všechny tyto nástroje budou otestovány v rámci běžného vývoje webové stránky. Na základě tohoto testu dojde k jejich obodování a následnému vyhodnocení, které by mělo jednoznačně určit ten

lepší z nich. Tato srovnání budou doplněna o ukázky kódu, které se mohou objevit napříč celou prací tam, kde to bude vhodné. Následně dojde ke shrnutí všech představených nástrojů a postupů a pokusu o návrh jejich nejefektivnější kombinace, která bude podpořena argumenty na základě informací nabitých touto prací.

2 Základní webové technologie

Dříve než dojde k představení nástrojů, které jsou centrem zájmu této práce, je nutné seznámit čtenáře se základními webovými technologiemi, pro které tyto nástroje představují podporu. Mezi tyto technologie patří především CSS a HTML, přičemž důraz bude kladen zejména na první ze jmenovaných technologií.

2.1 Seznámení s kaskádovými styly

CSS neboli Cascading Style Sheets, v češtině kaskádové styly, je stylovací jazyk, který vznikl za účelem zjednodušení formátování dokumentů popsaných značkovacími jazyky jako HTML, XHTML nebo například XML [1]. V následující práci bude však CSS popisováno především ve spojitosti s HTML.

Za formátování je považována například změna velikosti písma, jeho barva, barva pozadí, odsazování textů, pozice jednotlivých prvků HTML kódu a to na základě definovaných pravidel. Lze tedy říci, že CSS umožňuje převést obsah a strukturu stránky do pro uživatele stravitelnější, přehlednější a celkově líbivější podoby. Mimo jiné umožňuje tento stylovací jazyk přizpůsobovat design nejrůznějším typům zařízení, čímž podporuje trend dnešní doby – responzivní weby. Samotné formátování pak může být popsáno jako proces, kdy jsou jednotlivým elementům v přidělena určitá pravidla, podle kterých se mají tyto konkrétní elementy chovat – tedy jak se mají uživateli zobrazit.

K objasnění slova „kaskádové“ bude brán v potaz následující příklad – k jednomu HTML elementu může být přiděleno více pravidel, ale co se stane, když je jedním pravidlem natavena barva textu jako modrá a dalším pravidlem barva stejného textu jako červená? Při procesu kaskádování dochází k určení toho, které z pravidel má přednost. Když k tomuto určení dojde, je prvek vykreslen tou vlastností, pro kterou byla určena nejvyšší váha.

2.2 Vznik kaskádových stylů

Líbivý design a tedy i úprava vzhledu webových stránek dle vlastních představ, je dnes neodmyslitelnou součástí světa internetu, ovšem nebylo tomu tak od začátku.

Přibližně v roce 1990 jsou v CERNu položeny základy internetu tak, jak je známý dnes. U jeho zrodu stojí Tim Berners-Lee, který dal mimo jiné za vznik také HTML neboli HyperText Markup Language. Tento značkovací jazyk slouží k popisu struktury obsahu webové stránky [2].

Tim Berners-Lee od počátku zastával názor, že styl a obsah webu mají být odděleny. Pokusy o stylování webu se sice začaly objevovat už v této době, nicméně neexistoval jediný a standardní přístup, jak při stylování postupovat. Mnoho tvůrců zastávalo názor, že stylování by mělo zůstat výhradně na prohlížeči. Možnosti úpravy vzhledu většinou končily u změny barvy textu, či ztučnění písma. Když začal web získávat na popularitě, zejména díky prohlížeči NCSA Mosaic, tento přístup se ukázal jako nevhodný. Tou dobou se na scéně objevil Håkon Wium Lie, další ze zaměstnanců CERNu, který se rozhodl tento přístup změnit. V roce 1994 prezentoval svůj první návrh CSS. Nejenom, že CSS dávalo tvůrcům mnohem větší svobodu při designování stránek, ale také zachovalo původní myšlenku HTML, tedy oddělení vzhledu webu od jeho struktury. Netrvalo dlouho a CSS se stalo standardem webových stránek a jinak tomu není dodnes [3].

CSS ve verzi 1 bylo jako takové poprvé implementováno do prohlížeče Internet Explorer v roce 1996, implementace do prohlížečů Netscape následovala záhy. Verze 1 umožňovala například úpravy velikosti, barvy, zarovnání, či fontu textu, přidávání obrázků, změnu barev různých prvků, pozicování, vytváření ohraničení, zejména však klasifikování prvků do různých tříd, či dle unikátních identifikátorů. CSS v současnosti existuje ve verzi 3, ta přináší navíc například zaoblení rohů, stín u blokových prvků, či textu, transformace a mnoho dalšího [4].

2.3 CSS v současnosti

Od svého vzniku v roce 1994 se na poli kaskádových stylů udála celá řada změn. Jednotlivé verze tohoto jazyka k těm předchozím přidávaly další zejména složitější

a pokročilejší funkcionalitu. Nástroj, který ve verzi CSS Level 1, sloužil zejména k úpravě velikosti, odsazení a barvě textu, se přes Level 2, kdy bylo představeno například absolutní, relativní, či fixní pozicování, vyvinul v nástroj, který dokáže v řádu milisekund spočítat složité 3D animace, nejrůznější transformace a přechody prvků, či stínování textu, které byly představeny ve verzi Level 3 [4].

CSS Level 3 je v současnosti nejaktuálnější verzí tohoto jazyka a tak tomu je už dlouhá léta. Nelze konkrétně určit, jak je tomu dlouho. Během vyvíjení Levelu 3 totiž došlo k zásadním změnám – nová verze již neexistuje jako jeden celek všech vlastností, postupů a požadavků popsaných v jenom dokumentu, nýbrž jako výčet modulů, ke kterým jejich vývojáři přistupují jednotlivě, čímž bylo dosaženo snazší implementace úprav a změn v CSS všeobecně. Uživatel se tedy může setkat s celou řadou modulů, které existují v Levelu 3, ale data jejich poslední aktualizace se mohou lišit nejen v řádu měsíců, ale klidně i let [5].

Faktem vyplývajícím z tohoto rozdělení do jednotlivých modulů je to, že CSS4 jako takové nejspíš nebude nikdy existovat. A i když je možné setkat se s moduly, které jsou dostupné v Levelu 4, stejně tak je možné setkat se s moduly, které, ač aktuální, nesou označení Level 1. V následujících letech tedy bude nadále docházet k aktualizacím a vývoji v současnosti existujících modulů, ke vzniku nových modulů a nejspíš i k zániku těch, které nebudou pro svou dobu aktuální a vhodné [5].

2.4 Princip CSS

To, jakým způsobem je webová stránka uživateli prezentována vychází z CSS pravidel, jak tato pravidla fungují bude popsáno v následující části.

Dříve než bude možné pokročit v popisu fungování kaskádových stylů, je třeba představit pojem DOM. Document Object Model, je způsob, jakým je možné reprezentovat HTML soubor. Soubor je rozdělen do stromové struktury a jednotlivé části nabývají podoby objektů. Tento přístup umožňuje jednodušší manipulaci s HTML [1].

2.4.1 Selektory

Stěžejním principem kaskádových stylů je použití selektorů. Selektor může být považován za jakýsi ukazatel na místo ve struktuře webu. Toto místo může být jednoznačně určeno, může být určeno v závislosti na jiných elementech webové stránky, nebo může být určeno například svou pozicí v rámci jednoho řádku textu.

Tyto selektory mohou být rozděleny do několika základních kategorií:

1. Jednoduché selektory – tyto selektory umožňují upravovat jeden či více prvků a to na základě třídy nebo identifikátorů přiřazených jednotlivým HTML elementům, či tagů označujících tyto elementy [1].

2. Atributové selektory – tento typ selektorů umožňuje upravovat prvky na základě atributů, které jim jsou přiřazeny [1].

3. Pseudotřídy – selektory, které ovlivňují elementy na základě jejich současného stavu, či pozice ve struktuře webové stránky. Takovým ovlivněným elementem může být například odkaz, ke kterému se právě přiblížil kurzor myši, či formulářový prvek, do kterého je právě vkládán text. Odlišným příkladem může být například poslední potomek rodičovského prvku v DOM struktuře webové stránky, který je třeba odlišit od všech ostatních potomků [1].

4. Pseudo-elementy – na rozdíl od pseudotříd tyto selektory neslouží k ovlivňování HTML elementů, nýbrž jen jejich částí. Barevné odlišení prvního slova řádku, či vygenerování určitého výrazu následujícím právě na konci řádku, jsou jen dvěma z mnoha příkladů jejich použití [1].

5. Vícenásobné selektory – jednoduše řečeno se jedná o kombinaci výše popsaných selektorů, kdy je odkazováno na více elementů webové stránky. Tyto elementy jsou odděleny čárkou. Pravidla z tohoto

selektoru jsou následně aplikována na všechny elementy uvedené v selektoru [1].

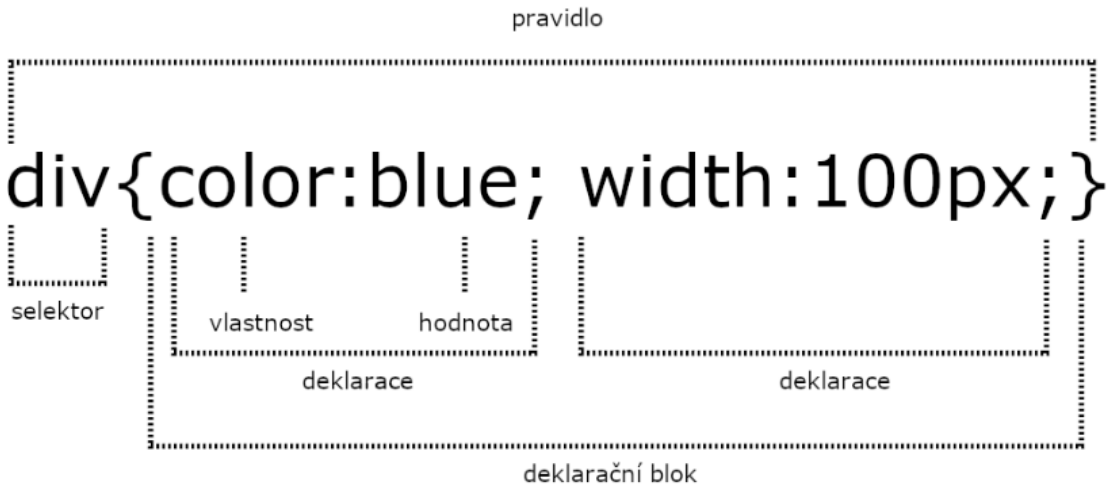
Většinu z výše popsaných selektorů je možné kombinovat, tím je možné dosáhnout nejen úspory, či čitelnosti kódu. Jejich kombinací je také možné řešit problémy, které by v jiném případě vyžadovaly zásah do kódu popisujícího strukturu webové stránky a to je postup, který bývá zřídkakdy ideální a vhodný.

2.4.2 Vlastnosti a jejich hodnoty

Krom selektorů jsou zcela nezbytnou součástí kaskádového pravidla také vlastnosti, které udávají, jakým způsobem má být daná webová stránka prohlížečem zobrazena.

Každá vlastnost musí mít svoji hodnotu. Hodnota udává, k jakému cíli by měl prohlížeč při vykreslování prvku dojít. Vlastnosti mohou nabývat celé řady hodnot. Mezi nejpoužívanější patří běžné číselné hodnoty, procenta, hodnoty určující barvu – například RGB, či dokonce funkce, které mimo jiné umožňují vytvářet animace [1].

Vlastnosti je nutné zapsat do deklaračních bloků, které následují za selektory a jsou značeny složenými závorkami. Jednotlivé vlastnosti jsou odděleny středníkem:



Obrázek 1: Popis CSS pravidla

2.4.3 Kaskádování

Kaskádování přichází na řadu zejména tehdy, kdy je na jeden prvek aplikováno více pravidel a na prohlížeči je zvolit jejich nejvhodnější kombinaci tak, aby došlo k co nejideálnějšímu vykreslení tohoto prvku. Každý prohlížeč má v základu sadu instrukcí, dle kterých by měl prvky vykreslovat. Jedná se o nejzákladnější stylistické zásahy, kaskádování se tak nelze vyhnout ani vhodným použitím selektorů [1].

Pravidla, dle kterých dochází k vybrání té nejvhodnější vlastnosti budou popsána v následující části.

2.4.3.1 **Důležitost a zdroj**

Důležitost, která se v CSS značí pomocí direktivy `!important`, je v podstatě jakýmsi esem při determinaci, jakým způsobem prvky vykreslit. Vlastnosti, u jejichž deklarace je použita tato direktiva – zapisuje se za hodnotu vlastnosti a středník, oddělující ji od ostatních pravidel, se zapisuje až po ní. Tato direktiva přebijí všechny ostatní vlastnosti, které by se mohly ucházet o aplikaci na konkrétní prvek.

Je dobré mít na paměti, že používání této direktivy není v praxi doporučováno a nutnost jejího použití většinou ukazuje na problémy v návrhu webové stránky. Uživatel by se měl tak jejímu použití za každou cenu vyvarovat [1].

Není však pravdou, že použití této direktivy vždy zajistí aplikování uživatelem zvolené vlastnosti. V tuto chvíli přichází na řadu zdroj, který udává, odkud nebo spíš kým jsou daná pravidla volána.

V CSS existuje několik základních způsobů, jak mohou být pravidla volána. Je vhodné zdůraznit, že se jedná o volání celého stylopisu, ve kterém jsou tato pravidla popsána. Zde následuje jejich pořadí, seřazené od nejnižší po nejvyšší váhu:

1. Pravidla pocházející od agenta – tato pravidla mají nejnižší váhu a jsou defaultními pravidly, na jejichž základě jsou HTML elementům přiřazovány jejich vlastnosti. Agent je pak samotný prohlížeč. To znamená, že chování webové stránky, pokud není řečeno jinak, se liší na základě toho, kterým prohlížečem je vykreslována [1].

2. Deklarace uživatelem – uživatel je v tomto případě ten, jemuž je webová stránka zobrazována, nejedná se tedy o autora stylopisu. Takový uživatel si může napsat své vlastní CSS, nalinkovat jej do prohlížeče a ten jej považuje za defaultní způsob, jak vykreslovat webové stránky [1].

3. Deklarace autorem – autorem je myšlen původní tvůrce webové stránky, tedy ten, který napsal CSS kód, připojil jej k HTML, tento web umístil na server a ostatní jej mohou pomocí svých prohlížečů navštívit [1].

4. Deklarace pomocí direktivy `!important` autorem – jedná se o obdobný případ, jako při běžné deklaraci autorem. Nicméně vlastnosti, u nichž je použito `!imporatant`, mají vyšší váhu [1].

5. Deklarace pomocí direktivy `!important` uživatelem – stejný případ, jako v případě popsaném výše. Nicméně tentokrát jsou vlastnosti vytvořeny koncovým uživatelem [1].

2.4.3.2 **Specificita**

Specificita je váha, kterou prohlížeč přiřazuje jednotlivým deklaracím na základě toho, jakým způsobem dojde k jejich zápisu. Tato váha vychází zejména z použitých selektorů, nicméně ji mohou ovlivnit i faktory jako použití direktivy `!imporatant`, či způsob spojení CSS s HTML. Na základě této váhy dochází k determinaci toho, která z deklarací bude při finálním vykreslení použita [1]. Dle konkrétních

specifikací [1] vytvořil autor práce pro účely snazšího pochopení následující tabulku:

Typ	Zápis	Váha
Element	p{}	1
Pseudotřída	:visited{}	10
Třída	.class{}	10
Identifikátor	#identificator{}	100
Inline	style=“”	1000
Important	!important	10000

Tabulka 1: Přehled vah jednotlivých zápisů

Pokud uživatel požaduje aplikování vlastnosti na určitý prvek, zanořený v jiném prvku, dochází při determinaci deklarace ke sčítání vah selektorů, odkazujících na určitý prvek. To znamená, že pokud má být změněn vzhled elementu `<p>` nacházejícím se uvnitř elementu s identifikátorem, celková váha vlastností uvnitř těchto elementů je rovna 101.

2.4.3.3 **Pořadí**

Pokud nastane situace, kdy je jednomu prvku přiřazováno více stejných vlastností se stejnou specificitou, přichází na řadu určení na základě pořadí.

Je nutné zdůraznit, že pravidla, která jsou určena pomocí takzvaného inline zápisu, jsou upřednostňována. Tato skutečnost byla již částečně nastíněna v tabulce pro specificitu, nicméně je dobré poznamenat, že umístění stylování přímo do elementu, je také jeho pořadím.

Dále už je postupováno tak, jak bývá při psaní kódu zvykem. To znamená, že vlastnost, na kterou narazí prohlížeč během procházení stylopisu jako poslední, je upřednostňována.

Popsaná pravidla se týkají jen jednotlivých vlastností, mezi kterými existuje podobnost. Pokud je například velikost textu jednoho prvku určena třídou, ale jeho barva je určena uvnitř identifikátoru, jsou na prvek aplikovány obě tyto hodnoty, jelikož jsou vlastnosti, které je charakterizují, zcela odlišné. K těmto pravidlům je přistupováno jen ve chvíli, kdy dochází k použití vlastností, které jsou ve vzájemném rozporu. Teprve v tu chvíli je třeba determinovat, která vlastnost má přednost.

2.4.4 Spojení s HTML

Než mohou být pravidla pro stylování webové stránky použita, musí být prohlížeči předána. Pro propojení CSS s HTML se používají tři základní způsoby.

1. Inline styly – jedná se o způsob zápisu, kdy jsou vlastnosti deklarovány přímo do konkrétního elementu. Při inline zápisu se používá atributu style, do nějž jsou následně vepsány požadované vlastnosti. Tyto vlastnosti mohou být následně aplikovány jen na tento element. Pokud uživatel vyžaduje změnu vlastností elementu popsaných inline zápisem, je nutné tak učinit u každého jednotlivého elementu. Je zřejmé, že takový postup je neefektivní. Navíc během tohoto zápisu dochází k mísení HTML s CSS, což může vést k nečitelnosti a je také v rozporu s oddělením vzhledu od struktury. Tento zápis se používá zejména v situacích, kdy chce uživatel rychle otestovat některou konkrétní vlastnost a nechce se probírat výčtem všech pravidel. Používání inline stylů každopádně není doporučováno [1].

2. Interní styly – obdobně jako u inline stylů, i zde dochází k zápisu CSS přímo do HTML kódu. Tyto dva zápisy se nicméně liší v tom, že CSS je od zbytku kódu odděleno umístěním do elementu `<style>`, který je umístěn v hlavičce HTML stránky. V důsledku toho je celkový kód přehlednější. Pokud se web uživatele skládá z několika odlišných stránek, naráží

interní styly na své limity a těmi je fakt, že pokud uživatel vyžaduje globální změnu vzhledu svého webu, musí tento interní kód změnit ve všech konkrétních souborech reprezentující jeho web [1].

3. Externí styly – jednoznačně nejpoužívanější způsob kaskádového stylování. Samotný kód reprezentující webovou stránku je v tomto případě zcela oddělen od HTML kódu. Pro správnou funkčnost externích stylů je třeba vytvořit soubor s koncovkou „.css“. Do tohoto souboru je následně vepsán CSS kód. Aby mohl prohlížeč řádně aplikovat vlastnosti z tohoto souboru, je třeba jej nalinkovat do požadovaného HTML kódu, reprezentující webovou stránku. Odkaz na tento soubor je umístěn v elementu `<link>`, který je opět umístěn v hlavičce HTML. Největší výhodou externích stylů je jejich globálnost, to jinými slovy znamená, že jakékoliv změny provedené v jednom souboru budou aplikovány na všechny HTML kódy, ke kterým byl daný soubor nalinkován. K jednomu kódu může být nalinkováno více externích stylů. Díky tomu je možné oddělit styly, například pro jednotlivé části webu, od sebe [1].

3 Vybrané podpůrné nástroje kódování

Každý, kdo už se někdy podílel na vývoji webových stránek, či aplikací, které se nesestávají jen z jednoduchého rozcestníku a pár obrázků, jistě narazil na to, že i nepříliš rozsáhlý projekt se po čase může měnit v chaos. Počty řádků stylopisu mohou časem dosahovat i desetitisíců, takový kód je často nepřehledný, a i jednoduché změny mohou stát autora mnoho úsilí. Kaskádové styly jsou jednoduchý jazyk. To, co se zpočátku zdá jako výrazné plus, se může záhy ukázat jako zásadní problém. Pro svou jednoduchost není CSS zcela ideálním nástrojem pro velké týmové projekty, které se mohou proměnit v obrovský kolos, jehož udržitelnost se snižuje tak, jak roste jeho velikost. Existuje spousta příruček, jak psát efektivní kód, spousta návodů popisujících ideálních způsoby, jak postupovat při vývoji webových stránek. Existují ale také nástroje, které tento proces umožňují výrazně zjednodušit a to bez nutnosti od základu změnit přístup k tvorbě webových stránek.

3.1 CSS preprocesory

Jedním z takových nástrojů jsou preprocesory. Preprocesory, jak už napovídá jejich název, jsou jakýmisi „předzpracovateli“. Jejich princip je v zásadě jednoduchý. Autor napíše kód v syntaxi jazyka některého z preprocesorů a ten je následně zkompilován do podoby nativního CSS.

Kaskádové styly jsou založeny na deklarativním přístupu. Tím je dosaženo právě jejich jednoduchosti. Preprocesory naopak inklinují k imperativnímu a objektovému přístupu ke kódování, kterým se velmi přibližují běžným programovacím jazykům a jejich výhod a schopností plně využívají. Preprocesory jsou často spojovány s takzvaným „DRY“ přístupem neboli „don’t repeat yourself“, do češtiny přeloženo jako „neopakuj se“ [6]. Časté opakování stejného kódu bývá obvykle největším problémem CSS. Nejenže kvůli tomu dochází k rapidnímu přibývání řádků, ale kód se vinou tohoto přístupu stává také velmi nepřehledným. V případě kaskádových stylů se bohužel často jedná o nejjednodušší a nebo jediné možné řešení.

Snížení opakování kódu ale nejsou jedinou devízou těchto nástrojů. Preprocesory přináší základní matematické operace, proměnné, mixiny, které se na první pohled podobají funkcím známých z programovacích jazyků. Ve skutečnosti se ale jedná o

jakési znovupoužitelné bloky vlastností. Jednou z nejužitečnějších vlastností preprocesorů je pak zahnízďování, které výrazně zjednodušuje a zpřehledňuje využívání předků a potomků, na jejichž existenci je HTML struktura prakticky založena. CSS preprocesory nabízí celou řadu dalších užitečných postupů a schopností. Všechny budou detailněji popsány na následujících řádcích.

3.2 Jednotlivé CSS preprocesory

Ač je každý z preprocesorů jazykem sám o sobě, jen málo se liší od původní CSS syntaxe, tyto změny jsou nicméně zřejmé už na první pohled. Mezi jednotlivými preprocesory pak neznalý uživatel hledá rozdíly jen těžko a ani rozdíly ve funkcionalitě napříč jednotlivými preprocesory nemusí být na první pohled zřejmé. Všechny dále představené preprocesory umožňují používat proměnné, mixiny, hnízdění, umí zpracovat matematické operace, či jednoduché rozhodování známé z běžných programovacích jazyků. Rozdíly mezi jednotlivými preprocesory nicméně existují, nejdříve ale budou krátce představeni jejich nejpopulárnější zástupci.

3.2.1 Sass

Jako první bude představen CSS preprocesor Sass neboli Syntactically Awesome Style Sheets. Neobjevuje se na první pozici čistě náhodou. Sass je právě TÍM preprocesorem. Jedná se o první preprocesor, který se dostal do širšího povědomí a stal se de facto standardem a šablonou pro to nejen, jak by měl správný preprocesor vypadat a jaké funkce nabízet, ale především pro to, co preprocesor vůbec je.

Sass byl poprvé představen už téměř před třinácti lety, tedy v roce 2006. Jeho původními tvůrci jsou Hampton Catlin a Natalie Weizenbaum, k nim se záhy přidal Chris Eppstein, který se po Catlinově odchodu v roce 2009 zařadil po bok Natalie Weizenbaum. Tito dva spolupracují na vývoji Sassu dodnes [7].

Inspirací pro Sass byl jazyk Haml, který přidával nové dynamické schopnosti k HTML. Tvůrci tohoto CSS preprocesoru si dali za cíl přinést obdobnou dynamiku do kaskádových stylů. Svého cíle původně dosáhli pomocí jazyka Ruby, ve kterém byl Sass původně napsán. Tento jazyk však musel používat každý, kdo se pokoušel za pomocí Sassu vyvíjet, což odradilo mnoho tvůrců od jeho používání. Z tohoto

důvodu byl později v jazyce C/C++ vytvořen port LibSass, usnadňující kompilaci tohoto jazyka. V současnosti je možné jej používat i ve spojení s Node, PHP či C [8].

Překvapivým zjištěním pro nové uživatele může být fakt, že Sass se vyskytuje ve dvou odlišných syntaxích. Syntaxe, na které byl Sass původně postaven, je nazvaná jednoduše Sass. Tento způsob zápisu zcela postrádá středníky a složené závorky a chování tak vychází zásadně ze správného odsazování a řádkování textu. Jelikož byla tato syntaxe pro mnohé překážkou, vzniká později syntaxe, která se více podobá původnímu zápisu CSS, tedy používání středníků a složených závorek, a od ostatní preprocesorů je tak téměř k nerozeznání. Nazývá se SCSS a stala se v podstatě standardem pro používání Sassu [7].

Nejen díky svému prvenství, ale také díky své velmi rozsáhle komunitě je Sass v současnosti nejspíš nejznámější a nejpoužívanější preprocesor na světě. A když může být na tomto preprocesoru založen například populární framework pro vývoj webu Bootstrap [7], nemůže být o jeho popularitě pochyb.

3.2.2 Less

Dalším zástupcem z řady preprocesorů je Less, také známý jako Leaner Style Sheets. Tento nástroj je ihned po Sassu pravděpodobně nejznámějším a nejpoužívanějším preprocesorem. Jeho počátky sahají až do roku 2009, kdy jej poprvé zveřejnil Alexis Sellier, pro internetovou komunitu spíše známý jako cloudhead. Obdobně jako Sass i Less byl založený na jazyce Ruby a obdobně jako u jazyka Sass i v případě Lessu záhy následoval port pro JavaScript a další jazyky [9].

Není překvapením, že celé tři roky po vzniku Sassu, se Less inspiroval právě zmíněným nástrojem. Tvůrce si dal za cíl zejména napravit nedostatky týkající se nepopulární syntaxe druhého nástroje. V době jeho vzniku totiž neexistuje syntaxe SCSS. Pro nováčky je tedy používání Lessu zcela intuitivní, jelikož je od běžné CSS syntaxe až na drobné detaily téměř k nerozeznání. V roce 2012 předává Sellaris žezlo svým nástupcům v podobě skupiny vývojářů, která se dodnes aktivně podílí na vývoji tohoto nástroje [9].

Za zajímavý rozdíl oproti jeho preprocesorovým kolegům může být v případě Lessu považován fakt, že je jej možné kompilovat přímo v prohlížeči. Kódy ostatních

preprocesorů je nutné po uložení nejdříve převést do klasické CSS syntaxe a tento zkompilovaný soubor následně klasickým způsobem nalinkovat do HTML souboru. V případě Lessu si prohlížeč snadno poradí s nezkompilovaným souborem v podobě syntaxe Less. Právě proto je mnohými uživateli upřednostňován. Tento způsob zpracování kódu však není doporučován, protože autor si nemůže být nikdy jistý, jak bude konkrétní prohlížeč na jeho Less kód reagovat.

3.2.3 Stylus

Posledním blíže představeným preprocesorem je Stylus. Obdobně jako předchozí popsané nástroje, i Stylus je založen na open source přístupu. Tento nástroj byl do světa vypuštěn v roce 2010 populárním vývojářem TJ Holowaychukem. V současné době ale dochází k vývoji tohoto nástroje spíše prostřednictvím komunity. Stylus byl napsán pomocí Node.js. Stejně jako Sass a Less jej lze díky tomu v současnosti snadno zapojit do uživatelova pracovního procesu. [8].

Obdobně jako v jazyce Sass i ve Stylusu je k dostání možnost vybrat ze dvou alternativ zápisu kódu. A obdobně jako v Sassu se jedná o syntaxi, která se velmi podobná nativnímu CSS, je nutné tedy používat běžné středníky a složené závorky. Druhá syntaxe je naopak založena na správném odsazování a odřádkování. Při použití této syntaxe je tedy možné zcela vypustit složené závorky, nebo také středníky, či dokonce dvojtečky, které udávají deklaraci určitých vlastností. Zajímavostí Stylusu je možnost tyto syntaxe kombinovat. Na rozdíl od Sassu mezi těmito dvěma syntaxemi neexistuje v rámci kompilace jazyka zásadní rozdíl. Tam, kde uživatel uzná za vhodné používat středníky, tam je může ale nemusí použít, stejně tak je tomu u závorek nebo dvojteček. V každém případě je ale třeba dbát na to, jakým způsobem je kód v souboru odsazován a odřádkován.

3.3 Zavedení preprocesorů do pracovního procesu

Před tím, než je možné začít ze schopností preprocesorů těžit, je nutné tyto nástroje zprovoznit. Jejich spuštění není nikterak složité, nicméně je dobré přiblížit, v čem se ono první spuštění u jednotlivých preprocesorů liší a na co si dávat pozor.

Pokud je řeč o instalaci a zavedení preprocesorů do pracovního procesu, jedná se zejména o otázku toho, jakým způsobem dojde ke kompilaci jednotlivých jazyků. Tedy překladu preprocesorového jazyka, do pro prohlížeče čitelného CSS.

V současnosti je prakticky nejpoužívanějším způsobem, jak preprocesory využívat, příkazová řádka. Zavítá-li uživatel na oficiální webové stránky některého z výše zmiňovaných preprocesorů, jako první se většinou setká právě s tímto přístupem. V souvislosti s řádkovým přístupem je nejčastěji zmiňován tzv. runtime Node.js., který zajišťuje správný běh a zpracování JavaScriptových kódů. Po nainstalování Node.js už jen stačí zadat do běžné příkazové řádky například příkaz `npm install -g less`, který slouží k instalaci kompilátoru pro jazyk Less. V takovém případě dochází k používání příkazové řádky i při samotné kompilaci. Při změně kódu stačí do příkazové řádky zadat v příslušné složce příkaz `lessc style.less style.css`, který, jak naznačuje zmíněný kód, nezkompilovaný soubor s koncovkou „.less" převede do běžného kódování kaskádových stylů [9]. Jelikož byla spousta preprocesorů napsaná pomocí JavaScriptu, případně existují jejich porty do toho jazyka, dá se tento přístup aplikovat na valnou většinu z nich a tedy i na již zmíněné jazyky Sass a Stylus. Detailní postupy společně s příkazy a příklady je možné najít na webových stránkách reprezentující jednotlivé preprocesory.

Dalším způsobem kompilace je použití běžného JavaScriptu. Tento postup je nicméně nedoporučovaný a z výše zmíněných preprocesorů v zásadě použitelný jen v případě Lessu. Pro tento způsob stačí v zásadě napsat libovolný kód pomocí lessovské syntaxe, tento soubor uložit s koncovkou „.less", následně jej pak už jen stačí nalinkovat do hlavičky HTML souboru obdobně, jako je tomu při linkování běžného externího CSS souboru. Následně už jen stačí přidat do hlavičky další odkaz, ten může buď odkazovat na patřičnou JavaScriptovou knihovnu na disku počítače, případně do hlavičky vepsat link odkazující na online knihovnu Lessu, která se o vše potřebné postará.

I práce s kódem pak probíhá podobně jako u běžného CSS. Při každé změně stačí soubor uložit a požadovanou webovou stránku znovu načíst. Je třeba mít na paměti, že tento přístup je oproti alternativám zdaleka nejpomalejší a nejnebezpečnější,

uživatel si totiž nikdy nemůže být kompilací zcela jistý, proto jsou uživatelé od používání této metody při běžném nasazení na síti odrazováni [9].

Pro uživatele, kteří upřednostňují jiné přístupy než příkazovou řádku, existuje alternativa v podobě grafického prostředí. Existuje celá řada takových nástrojů, některé z nich zdarma, za některé je třeba zaplatit. Takové nástroje stačí nainstalovat. V některých detailech se jejich implementace může lišit, ale ve většině případů stačí pomocí běžného průzkumníku souborů najít požadovaný kód napsaný pomocí preprocesoru, zvolit způsob kompilace, kompilovat a výsledný soubor, již s příponou „.css“ klasicky nalinkovat do hlavičky HTML souboru [9].

V neposlední řadě je možné využít schopností online kompilátorů. Tento přístup nevyžaduje žádné instalace ani učení se nových příkazů. Jednoduše stačí rozkliknout patřičnou webovou stránku a začít psát. K překladu do CSS pak většinou dochází automaticky a výsledek se objeví okamžitě, většinou po boku uživatelova preprocesorového kódu. Tento přístup samozřejmě není vyvinutý pro nasazení při běžném pracovním procesu. Jedná se spíše o způsob, jak si snadno a rychle otestovat a vyzkoušet jednotlivé syntaxe. Při běžném používání by pak bylo třeba vždy patřičný zkompilovaný kód nakopírovat do CSS souboru propojeného s HTML webové stránky. Existují nicméně i webové aplikace, které umožňují ukládání takového kódu na disk a práce by tak probíhala obdobně jako při používání běžného vývojového prostředí. Mezi takové weby patří například Sassmeister [10] pro kompilaci jazyku Sass, či Online Less Compiler [11], který umožňuje snadnou kompilaci jazyka Less.

Není třeba zmiňovat, že každý z těchto postupů má své pro a proti a je tedy na konečném uživateli, kterou ze zmíněných cest uzná za vhodnou. Za to je třeba zmínit, že existuje spousta dalších knihoven a pluginů, které umožňují tak či onak práci s jednotlivými preprocesory zefektivnit.

3.4 Funkce preprocesorů

Preprocesory nabízí různé funkce a schopnosti, které usnadňují a zefektivňují práci s CSS. Každý z preprocesorů se alespoň částečně v různých aspektech liší. Většina z nich, však nabízí ve svém jádře víceméně totožné funkce. Rozdíly mezi jejich

zpracováním od sebe mohou být na první pohled k nerozeznání, a někdy je syntaxe tím jediným, co dokáže konkrétní procesor prozradit. Jelikož smyslem této práce není porovnávat preprocesory, budou tyto funkce popsány jen okrajově, protože jejich poznání je zcela signifikantní pro pochopení práce s postprocesory, které budou představeny později.

3.4.1 Proměnné

Proměnné bývají prvním argumentem pro přechod k preprocesorům. Není to překvapivé. Snad každý, kdo se kdy setkal s praktickým užíváním CSS tuto schopnost nejednou postrádal. Proměnné v rámci preprocesorů fungují přesně tak, jak je od nich očekáváno, tedy obdobným způsobem, jako v běžných programovacích jazycích. Jednoduše je to místo v paměti s nějakou hodnotou, ke které je možné přistupovat.

I když bývají proměnné často tím nejhlasitějším argumentem, jak získat nové uživatele na svou stranu, pravda je taková, že již několik let je možné využívat proměnné i v rámci nativního CSS. Je sice skutečností, že jejich implementace do pracovního procesu je závislá spíše na straně prohlížečů, ale v roce 2019 se mezi populárnějšími prohlížeči snad nedá setkat se zástupcem, který by proměnné alespoň částečně nepodporoval [1].

K proměnným v rámci nativního CSS je ale přistupováno přece jen trochu jinak než jak je tomu u preprocesorů. Pokud dojde k použití proměnných v rámci preproceosorů, je přirozeně nezbytné nejdříve takovou proměnnou deklarovat. K této proměnné pak může preprocesor v určitých místech kódu přistupovat. Při kompilaci preprocesorového kódu jsou pak tato místa nahrazena konkrétní hodnotu, která je již pro prohlížeče v rámci CSS čitelná. Jistě, existuje zde oproti běžně známým postupům v CSS jistá dynamika, nicméně jakmile je takový kaskádový styl jednou načten prohlížečem, už jej nelze změnit.

Pokud se však uživatel rozhodne posunout používání nativního CSS o úroveň výše a využívat jeho schopností v plné šíři, seznámí se s prací s proměnnými, v CSS nazvanými jako „CSS custom propertis“, velmi krkolomně přeloženo jako „vlastními

vlastnostmi“, které toho nabízí daleko více, a to bez nutnosti jakýchkoliv instalací, či inovací.

3.4.1.1 CSS proměnné

Před použitím tohoto nástroje je nutné seznámit se s pseudotřídou `:root`. Tato pseudotřída slouží k takzvané globální deklaraci proměnných. K těmto proměnným je poté možno přistupovat v rámci celého kódu. Práce s touto pseudotřídou probíhá v rámci CSS jako v případě běžných selektorů. K samotné deklaraci proměnné pak dochází podobně jako při přiřazování hodnot vlastnostem v rámci deklaračního bloku. K samotnému deklarování je pak třeba použití dvou za sebou jdoucích pomlček, po kterých následuje uživatelem vhodně zvolený název proměnné. Je třeba míti na paměti, že tyto proměnné jsou citlivé na velikosti znaků . `--promenna a --Promenna` jsou pak tedy dvěma zcela odlišnými entitami a takovým způsobem k nim prohlížeč také přistupuje. Po názvu proměnné pak nastupuje dvojtečka, která slouží k přiřazení hodnoty proměnné a následně už zmiňovaná hodnota. To může být například barva, číslo, či rozměr [1].

K hodnotám proměnných je přistupováno pomocí direktivy `var()` a její používání probíhá obdobně, jako v případě volání funkce v rámci běžného programovacího jazyka – mezi závorky je umístěn název proměnné, jejíž hodnotu má autor v úmyslu použít. V rámci nativního CSS se tak jedná o jakousi pseudofunkci, jejíž jediným úkolem je vracet hodnotu atributu, který je zadaný jako parametr. Deklarace a přistupování k hodnotě proměnné by probíhalo následovně:

```
/* Deklarace */
:root{
  --barvaPozadi: blue;
}
/* Získávání hodnoty */
#blok{
  background-color: var(--barvaPozadi);
}
```

K lokální deklaraci, i když s pojmem lokální je to v rámci uživatelských vlastností trochu složitější, dochází obdobným způsobem. Jako deklarační blok se však

nevyužívá ten pro pseudotřídu `:root`, nýbrž ten, ve kterém je žádoucí danou proměnnou využívat.

Dosud se používání CSS proměnných od těch preprocesorových příliš neliší. Rozdíl je nicméně zásadní. Na rozdíl od těch preprocesorových proměnných, které jsou statické a jsou vlastně jen přepsáním hodnoty v souboru s kaskádovým stylem, mají uživatelské vlastnosti představu o tom, jak vypadá DOM struktura HTML souboru a je možné k nim přistupovat i pomocí JavaScriptu. [1].

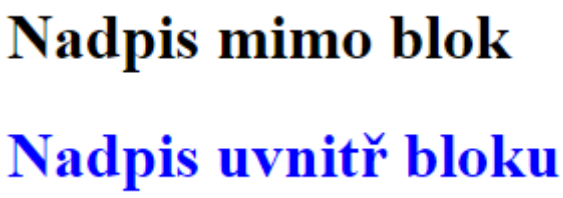
To, že mají uživatelské vlastnosti přehled o DOM struktuře dokumentu znamená, že podléhají kaskádování stejným způsobem, jako je tomu u ostatních prvků CSS. Jinými slovy prohlížeč pozná, zdali má funkce `var()` přístup k proměnné a to ne na základě toho, kde byla deklarována, nýbrž na základě toho, v jakém kontextu se vyskytuje v rámci HTML kódu. Dochází zde ke stejné dědičnosti potomků od předků, jako u běžných vlastností. V případě globálnosti, které je dosaženo již výše zmíněnou pseudotřídu `:root`, jde v podstatě také o běžnou dědičnosti. Tato pseudotřída je jen elementem, který je jakýmsi prapředkem všech ostatních elementů. Tento přístup je zásadně rozdílný od přístupu proměnných v preprocesorech, které mají rozsah buď čistě globální anebo jen v rámci jednoho deklaračního bloku. Jsou tedy zcela nezávislé na struktuře HTML. Snadným příkladem tohoto přístupu může být následující scénář:

Na uživatelově webové stránce se vyskytuje několik nadpisů nejvyšší úrovně. Dále existuje blokový element patřící do třídy „blok“. Pokud je v deklaračním bloku pravidla s třídním selektorem `.blok` deklarována proměnná, může k ní přistupovat jen to pravidlo, které se v HTML struktuře vyskytuje na místě potomka blokového prvku s třídou „blok“. Pro ilustraci následuje krátký kód a ukázka:

```
/* HTML */
<h1>Nadpis mimo blok</h1>
  <div class="blok">
    <h1> Nadpis uvnitř bloku</h1>
  </div>

/* CSS */
.blok{
  --barva: blue;
}
h1{
  color: var(--barva);
}
```

Výsledek:



Obrázek 2: Ukázka použití custom properties

Jak je z ukázky vidět, prvek nadpisu, který se nacházel mimo blokový element má černou barvu. To je důsledek toho, že tento nadpis nemůže přistupovat k proměnné `–barva`. Nenalezne-li prohlížeč hodnotu proměnné, jsou prvku přiděleny defaultní vlastnosti vycházející z pravidel konkrétního prohlížeče. Podobného výsledku je možné pomocí preprocesorových proměnných dosahovalo jen velmi obtížně.

CSS uživatelské vlastnosti jsou tedy velmi mocným, avšak často opomíjeným nástrojem. Je proto třeba brát jejich existenci v potaz před tím, než dojde k uvažování nad tím, zdali jsou preprocesory pro pracovní proces nezbytné.

3.4.1.2 **Preprocesorové proměnné**

Proměnné a rozdíly mezi jejich aplikací v rámci celé řady preprocesorů nejde snadno shrnout jen do jedné kapitoly. Cílem této části tak bude seznámit čtenáře se základními postupy při práci s proměnnými a to zejména v případě představených preprocesorů.

Jak už bylo nastíněno výše, proměnné v rámci preprocesorů se svým chováním v zásadě nijak neliší od proměnných vyskytujících se v běžných programovacích jazycích. Jejich schopnosti lze významně docenit právě tam, kde dochází k častým změnám a na více místech v kódu. Jelikož tato funkcionalita nebyla implementována do CSS od začátku, tvůrci preprocesorů si dali za cíl tento nedostatek odstranit a umožnit uživatelům těchto schopností využívat i v rámci kaskádových stylů.

Základním předpokladem pro používání proměnných je jejich deklarace. Každý z jazyků k deklaraci přistupuje po svém. Přístupy k ní navíc nemusí končit jen u jednotlivých jazyků, ale i každá z jednotlivých syntaxí (pokud jich je víc) konkrétního jazyka zastává v rámci této problematiky svůj postoj. Pro jednoduchost bude popsána deklarace v jazyce Less.

Deklarace proměnných v jazyce Less je velmi intuitivní. Probíhá obdobným způsobem jako v případě uživatelských vlastností s tím rozdílem, že pro globální rozsah není třeba využívat žádných pseudotříd. Proměnnou stačí jednoduše zapsat mimo deklarační blok pravidel a je globálně přístupná. Naopak, pokud je třída deklarovaná v rámci některého z deklaračních bloků, může k ní být přistupováno jen v rámci právě onoho konkrétního pravidla. Syntaxe v jazyce Less se pak vyznačuje používáním znaku „@“, který je třeba vložit před název proměnné. Opět je možné se zde setkat s dvojtečkou, za níž už pak následuje hodnota, které by měla proměnná nabývat. K takové proměnné je přistupováno obdobným způsobem. Opět je třeba využít znaku „@“, ale teď už na opačném konci přiřazení [9].

Kód pro ukázku:

```
/* Less */
@barva: blue;

h1{
  color: @barva;
}

/* Zkompilovaný kód */
h1 {
  color: #0000ff;
}
```

Obdobně pak k deklaraci a přistupování k proměnné dochází i v případě jazyků Sass a Stylus, syntaxe se však drobně liší.

Proměnné lze v případě většiny preprocesorů používat s celou řadou datových typů a to od prostých čísel až po jiné proměnné. Díky nim je také možné provádět celou řadu matematických operací. Většina preprocesorů také umožňuje používat proměnné na pozici samotných vlastností, či selektorů CSS pravidel [9].

3.4.2 Mixin

Pokud jsou proměnné způsob, jakým se preprocesory přibližují programátorskému přístupu, o takzvaných „mixinech“ to platí dvojnásob.

Mixiny mohou svým použitím do značné míry připomínat funkce tolik známé z imperativního programování. Tam, kde je znovu použitelnost jednotlivých hodnot či vlastností málo, nastupují právě mixiny. Ty umožňují znovu použít celé skupiny vlastností a to společně s jejich hodnotami. Použití mixinů může být přirovnáno k volání funkce uvnitř jiné funkce. Přirovnání k funkcím je navíc podpořeno faktem, že mixiny podporují zpracování parametrů, které do nich vstupují podobným způsobem.

Jejich použití se opět napříč preprocesory liší spíše v detailech, největší rozdíl se tak vyskytuje znovu zejména v syntaxi. V jazyce Less tak v podstatě mezi běžným pravidlem a mixinem nemusí být žádný patrný rozdíl. Stačí vytvořit běžné pravidlo a jeho selektor následně vložit do pravidla jiného. Obsah mixinu, který může být zároveň běžným pravidlem, se tak po kompilaci do CSS jednoduše objeví, jako výčet vlastností společně s jejich hodnotami tak, jak byly deklarovány v mixinu [12].

V jazyce Sass je pak za použití SCSS syntaxe obdobného cíle dosaženo pomocí direktivy `@mixin`, za níž následuje název mixinu, společně s deklaračním blokem a v něm uloženými vlastnostmi, společně s jejich hodnotami. K volání takového mixinu pak dochází použitím direktivy `@include` a po ní názvu konkrétního mixinu [7]. Sass tak na první pohled upřednostňuje jednoznačné odlišení mixinů od ostatních prvků. Tento fakt sice přidává nutnost pamatovat dalších syntaktických pravidel, nicméně činí kód daleko přehlednějším.

Zde následuje krátká ukázka zpracování mixinu zapsaného v jazyce Sass:

```
/* Sass */
@mixin mujBlok {
    background-color: black;
    color: white;
    width: 100px;
}

.blok1 {
    @include mujBlok;
}
.blok2 {
    @include mujBlok;
    font-size: 12px;
}

/* Zkompilované CSS */
.blok1 {
  background-color: black;
  color: white;
  width: 100px;
}

.blok2 {
  background-color: black;
  color: white;
  width: 100px;
  font-size: 12px;
}
```

Z ukázky je patrné, že mixiny jsou velmi efektivním nástrojem, co se znovu použitelnosti týče. Používání mixinů sice velmi zjednoduší stylování webové stránky a ušetří spoustu času. Jejich neuvážené používání však může vést k velmi rychlému nabobtnání kódu. Pravdou totiž je, že valná většina scénářů, při kterých se uživatelé přiklánějí k používání tohoto nástroje, může být řešena vhodnou kombinací pravidel. Takový postup ale velmi často vyžaduje nežádoucí zásah do HTML kódu. Ideální cestou je tedy najít rovnováhu mezi těmito dvěma přístupy.

Funkcí, kde mixinům nativní CSS zcela nestačí jsou parametrické mixiny. Jejich použití je podobné jako v případě běžných mixinů. Za název mixinu stačí ve většině případů přidat složené závorky (některé preprocesory takový přístup vyžadují defaultně) a do nich vložit požadované hodnoty. Tyto parametry by byly následně volány v rámci mixinu podobně jako proměnné. Obsah mixinu se tak může měnit na

základě toho, z kterého pravidla je právě volán. V kombinaci s běžnými operátory se pak parametrické mixiny stávají ještě mocnějším nástrojem.

Krátká ukázka zpracování parametrického mixinu, opět v jazyce Sass:

```
/* Sass */
@mixin rozmery($w: 500px, $h: 600px) {
    width: $w;
    height: $h;
}
.ctverec {
    @include rozmery(420px,420px);
}
.obdelnik {
    @include rozmery;}

/* Zkompilované CSS */
.ctverec {
  width: 420px;
  height: 420px;
}

.obdelnik {
  width: 500px;
  height: 600px;
}
```

Dle ukázky je patrné, že parametrické mixiny jsou opravdu silným nástrojem, který lze nativním CSS nahradit jen těžko. V ukázce je také možné vypozorovat přítomnost defaultních hodnot v bloku vytváření mixinu. Jejich použití funguje na základě přítomnosti či nepřítomnosti hodnot v kulatých závorkách při volání mixinu. Pokud mixin nenalezne počet hodnot odpovídající počtu parametrů, je za vynechanou hodnotu doplněna ta defaultní.

Mixiny nabízí další řadu méně či více významných schopností. Pro základní představu o tom, co je vlastně tato funkce zač, však tento krátký popis plně postačí.

3.4.3 Extend

Extend neboli rozšíření může být na první pohled považován za nástroj velmi podobný právě již zmiňovaným mixinům. Také umožňuje efektivně sdílet vlastnosti i s jejich hodnotami napříč kódem. K tomuto cíli se však vydává naprosto opačným způsobem nežli výše zmiňovaný nástroj a to navíc za pomocí přístupu DRY. Na rozdíl od mixinů totiž v případě extend nedochází ke znovu používání kódu, ale k vytváření

vícenásobných selektorů, to jinými slovy znamená, že v kódu nepřibývají řádky takovou rychlostí, ale uživatel se může při používání rozšíření i ve svém vlastním kódu velmi snadno ztratit. V následujících řádcích bude přiblíženo, jakým způsobem extend operuje.

Tento přístup je v zásadě založený na vlastnosti CSS, kdy při použití vícenásobných selektorů, ve kterých jsou jednotlivé selektory odděleny čárkou, dochází k aplikování všech vlastností, které jsou uvnitř deklarovacího bloku pravidla, na všechny HTML prvky, které podléhají této selekci. Extend tento přístup umožňuje výrazně zjednodušit. Podobně jako proměnné či mixiny, i tato funkce se napříč preprocesory liší zejména v syntaxi. Ve většině populárnějších preprocesorů se však v rámci jejich syntaxe objevuje výraz `extend`, který se může svým dalším zpracováním mírně lišit. V jazycích Sass a Stylus se objevuje v následující podobě: `@extend`, kdy za touto direktivou následují selektory pravidel, která by měla být rozšířena. Tento výraz musí být samozřejmě umístěn uvnitř bloku, jehož výčet vlastností uživatel zamýšlí rozšířit [7]. Less pak k obdobnému výsledku dochází pomocí direktivy `&:extend( )` [8]. V takovém případě je třeba následně do kulatých závorek umístit název pravidla, které se má o své vlastnosti podělit s pravidlem volajícím. Rozšiřovanými pravidly samozřejmě nemusí být jen třídy, ale veskrze všechny typy pravidel s jejich selektory, které už byly zmíněny.

Následující ukázka v jazyce Sass snad pomůže osvětlit některé základní principy tohoto nástroje:

```
/* Sass */
.cervenyText {
  color: red;
}
.modrePozadi {
  @extend .cervenyText;
  background-color:blue;
  font-size:15px;
}

/Zkompilované CSS */
.cervenyText, .modrePozadi {
  color: red;
}

.modrePozadi {
  background-color: blue;
  font-size: 15px;
}
```

Na základě ukázky je pak základní princip relativně snadno pochopitelný. Selektor pro prvek, který má nabýt nových vlastností, je přidán k již existujícímu selektoru pravidla, které požadované vlastnosti obsahuje. Vlastnosti, které jsou pro dané pravidlo speciální, pak zůstávají v původním pravidle.

Při používání této funkce je pak třeba dávat si dobrý pozor na to, jaké selektory se objevují v rámci volání. Uživatel by si měl nejdříve dobře promyslet, kde všude v kódu se mohou daná pravidla vyskytovat a zejména obezřetně je pak nutné přistupovat k rozšiřování pravidel, která jsou již zaštítěna vícenásobným selektorem. V takovém případě může často dojít i k rozšíření pravidel, která zdaleka neodpovídají záměru uživatele [13].

3.4.4 Hnízdění

Hnízdění, v angličtině také známe jako nesting je schopností preprocesorů umožňující přehledné, snadno rozšiřitelné a zejména velmi intuitivní zanořování. Jinými slovy tato vlastnost zjednodušuje vytváření vztahů mezi potomky a jejich rodiči.

Svým přístupem se zanořování velmi přibližuje HTML. Zanořené pravidlo je vloženo do deklaračního bloku rodičovského pravidla. Vložené pravidlo se tak stává jeho potomkem a odkazuje tak na HTML strukturu dokumentu. U předchozích funkcích byla touto dobou zmíněna zejména syntaxe a její rozdíly mezi jednotlivými preprocesory. V tomto případě je syntaxe napříč preprocesory v zásadě stejná – jedno standardní pravidlo je vloženo do druhého pomocí běžného zápisu CSS.

Pokud si dá autor za cíl například obarvit na červeno jen odstavce, které se nacházejí uvnitř tříd s názvem „blok“, jednoduše vytvoří běžným způsobem pravidlo se selektorem `.blok`. Tomu mohou být přiřazeny konkrétní vlastnosti. Následně je třeba do tohoto deklaračního bloku umístit další pravidlo, a to pravidlo se selektorem `p`. Do něj poté stačí vložit požadované vlastnosti.

Pro lepší pochopení následuje krátká ukázka výše zmíněného scénáře:

Nejdříve zjednodušená struktura HTML:

```
<div class="blok">
<p>Červený odstavec</p>
</div>
<div>
<p>Odstavec, který by byl rád červený<p>
</div>
```

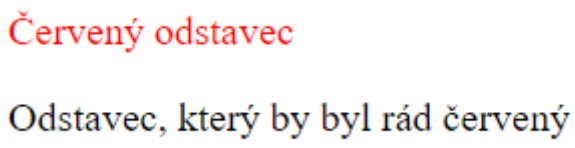
Zpracování zahnízdění pomocí Lessu:

```
/* Less */
p{
font-size:20px;
}

.blok {
  p{
  color:red
  }
}

/* Zkompilované CSS */
p {
  font-size: 20px;
}
.blok p {
  color: #ff0000;
}
```

A výsledek:



Obrázek 3: Výsledek zahnízdění

Z ukázky je patrné, že došlo k obarvení jen jednoho z odstavců, je to zapříčiněno tím, že jen jeden je potomkem bloku a splňuje tak kritéria pro obarvení.

Většina preprocesorů také nabízí takzvaný odkaz na rodiče. Ve své podstatě se nejedná o zanoření jako takové, nicméně je to funkce založená na obdobné syntaxi. Jedná se o zpřesňování vazeb mezi pravidly. Tato funkce je zejména užitečná v kombinaci s pseudotřídami, kdy je nutné zavést například pravidlo, které značí, že odstavec, nad kterým se vznáší uživatelův kurzor, bude mít jinou barvu. Tohoto cíle je dosaženo pomocí znaku „&“. Ten je vložen před název selektoru, který se nachází na místě potomka. Obdobně jako při běžném zanořování, i zde je možné tuto syntaxi aplikovat napříč většinou populárnějších preprocesorů.

Ukázka zpracování odkazu na rodiče:

```
/* Sass */
p {
color:blue;
  &:hover{
    color:red
  }
}

/* Zkompilované CSS */
p {
  color: blue;
}
p:hover {
  color: #ff0000;
}
```

Obdobný přístup je samozřejmě možné zvolit i pro jiné selektory nežli pseudotřídy.

Stejně jako u předchozích funkcích, i u hnízdění je nutné přistupovat k používání obezřetně. Jelikož tato syntaxe umožňuje velmi snadný zápis vnořených pravidel, snadno také může dojít k jejich překombinování a takový přístup vede k požadované funkčnosti jen zřídkakdy a výsledný kód se stává velmi nepřehledným.

3.4.5 Další funkce

Preprocesory nabízí celou řadu funkcí, ty nejpoužívanější z nich již byly popsány na řádcích výše. Preprocesory však nabízejí spoustu dalších užitečných vlastností a nástrojů, které není vhodné přehlédnout, ty budou ale představeny už jen ve zkratce.

První z takových funkcí je **import**. Jedná se o velmi užitečný nástroj, který umožňuje předávat prohlížeči větší množství CSS kódu a to za cenu zachování přehlednosti. Tato vlastnost se objevuje i v nativním CSS, pracuje však na jiném principu. Pokud prohlížeč při čtení CSS narazí na direktivu `@import`, za kterou následuje cesta k požadovanému souboru, musí vyslat požadavek, kterým takový soubor načte, aby s ním následně mohl pracovat [1].

Preprocesory pracují jinak. Stejně jako v nativním CSS, i u preprocesorů je většinou možné setkat se s direktivou `@import`, stejně tak za ní následuje cesta k souboru. Zpracování však probíhá jiným způsobem. Když během kompilace překladač narazí

na `@import`, načte jeho obsah a uloží jej do finálního CSS souboru, společně s obsahem preprocesorového souboru, ve kterém se požadavek na import objevil. V takovém případě již prohlížeč nemusí vysílat několik požadavků, ale jen jeden, a to pro soubor CSS, který vznikl kompilací souborů v jazyce daného preprocesoru.

Další zajímavou funkcí, se kterou se tentokrát není možné v nativních kaskádových stylech setkat jsou **řídící struktury** – tedy například větvení na základě podmínek a cykly. Tyto struktury nabízí obdobnou funkcionalitu, se kterou je možné setkat se v běžných imperativních programovacích jazycích. Jejich síla se projevuje zejména ve spojení s mixiny. Je tedy možné nastavit hodnotu určité vlastnosti, jen pokud jsou splněny některé z podmínek. Případně je možné takovou vlastnost vůbec nepoužívat. Obdobně jako v programovacích jazycích fungují i cykly, které dokážou například vygenerovat několik pravidel, jejichž selektory se liší jen v použitém čísle – například hromadné vytvoření pravidel pro nadpisy různých úrovní.

Existuje celá řada dalších zajímavých funkcí, které preprocesory nabízejí, často je zmiňována například **práce s barvami**, kde je možné jednoduše některé prvky zesvětlit bez nutnosti vyhledat konkrétní barvu v paletě barev, ale jen za použití například funkce `lighten`. V kódech napsaných v některém z těchto jazyků je také možné narazit na jednoduché **matematické funkce**, tedy například sčítání různých velikostí, či barev. Matematické funkce se ale v poslední době začínají prosazovat i v rámci nativního CSS, kdy je možné dojít obdobných výsledků za pomoci funkce `calc()` [1].

Tímto výčet nástrojů, funkcí, vlastností a schopností preprocesorů samozřejmě zdaleka nekončí. Všechny další možné funkce, společně s jejich detailními popisy a příklady, je možné nalézt na oficiálních stránkách konkrétního preprocesoru.

3.5 CSS postprocesory

Jak už vyplývá z názvu, tam kde preprocesory vstupovali před CSS a docházelo k jakémusi předzpracování a funkční CSS kód byl jejich produktem, tam postprocesory následují po již funkčním a pro prohlížeče čitelném kódu. Alespoň tak by tomu mělo být v teorii.

Pokud je řeč o postprocesorech, je třeba rozlišovat mezi jednotlivými nástroji, které plní jednu konkrétní funkcionalitu – například minifikace kódu, a komplexními řešeními, které využívají právě kombinace těchto jednotlivých nástrojů a velmi se tak přibližují preprocesorům.

Rozdílem mezi preprocesory a postprocesory je ten, že postprocesory nejsou na rozdíl od preprocesorů novým jazykem. Pro práci s nimi není nutné učit se nové syntaxi. Jejich základní myšlenkou je využívaní funkčního CSS v nativní podobě, ke kterému je přidaná určitá funkcionalita navíc [14].

Jedním z příkladů takových funkcí může být automatické prefixování, které má za úkol vyhledat nutné prefixy vlastnostem, které je vyžadují, doplnit je do uživatelova nativního CSS kódu a zajistit tím kompatibilitu napříč prohlížeči.

Použití postprocesorů bylo zpočátku zaměřeno zejména na optimalizaci a automatizaci. Postupně získávaly postprocesory na popularitě a tím pádem začaly přibývat i nové funkce. Takové funkce, které ale nejsou zcela konzistentní s původní myšlenkou postprocessingu – funkce jako proměnné, mixiny, či hnízdění, se kterými je možné setkat se u preprocesorů a už z principu jejich fungování je nelze označovat za funkce, které následují až po standardním a pro prohlížeče již čitelném kódu. Sami tvůrci těchto nástrojů proto pomalu začínají upouštět od názvu „postprocesory“ a začínají sami své nástroje nazývat jednoduše „procesory“ – kombinují to nejlepší z obou světů [14].

Jeden zásadní rozdíl zde však lze přece jen nalézt. Pokud pracovní proces uživatele vyžaduje proměnné, musí v případě preprocesorů začít používat celý tento nástroj se všemi jeho klady i zápory, naučit se novou syntaxi a změnit celý svůj pracovní přístup. V případě postprocesoru mu na druhou stranu stačí stáhnout jediný plugin, který mu umožní používat proměnné a zbytek kódu zůstává nezměněn. Postprocesory jsou totiž založeny na pluginech, které je třeba stáhnout před použitím jejich jednotlivých funkcí [14].

Postprocesory mnohdy také nabízejí možnost, jak vytvářet své vlastní funkce. Pokud má tedy autor jakýsi vlastní pracovní postup, který mu neumožňuje zautomatizovat žádný ze zmíněných procesorů, může si napsat ve formě pluginu funkcionalitu

vlastní. V tom také spočívá jedna z dalších výhod postprocesorů. Objeví-li se nová funkcionalita, není třeba čekat na velký update celého nástroje. Takovou funkcionalitu stačí jednoduše stáhnout a přidat mezi ostatní pluginy.

Mezi nejčastěji skloňované zástupce patří zejména nástroj PostCSS, který se inspiroval prapůvodním postprocesorem zvaným Rework, ten již ale v současnosti není vyvíjen a je považován spíše za mrtvý [16]. Další takový nástroj, který se svojí funkcionalitou přibližuje preprocesorům je například Stylcow.

Komplexní postprocesory tak v dnešní době poskytují velmi obdobnou funkcionalitu jako preprocesory, přičemž přidávají i něco navíc. Cenou za to je ale nutnost obdobně jako v případě preprocesorů tyto nástroje implementovat do svého pracovního procesu, což nemusí být vždy jednoduché a to obzvlášť pro začátečníky, nicméně pokud uživatel překoná tyto počáteční potíže, postprocesory mu jeho snahu oplatí.

4 Metodické postupy kódování a jejich navázání na nástroje

Dalším způsobem, jak dosáhnout zefektivnění a zpřehlednění práce s CSS jsou takzvané metodiky. Tyto metodiky jsou založeny zejména na tom, jakým způsobem uživatel uvažuje nad CSS kódem. Není tak nutné stahovat, či instalovat nové nástroje. Některé metodiky lze považovat za jakousi sbírku tipů a triků. Některými z jejich doporučení se pak může uživatel již nevědomky řídit. Jiné mohou být založené na striktních postupech, které vyžadují od základu změnit přístup ke kódování. Existuje celá řada metodik a ty nejpopulárnější z nich budou popsány v následující kapitole.

4.1 OOCSS

OOCSS neboli Object-Oriented CSS je pravděpodobně první pro veřejnost známou metodikou. Svou tvůrkyní Nicolle Sullivan byla představena již v roce 2008 a dodnes patří mezi nejpoužívanější metodiky [16]. Z názvu jasně vyplývá inspirace objektovým přístupem v programování. Jak ale objektového přístupu dosáhnout v CSS? Dle Sullivan je objekt „*Jakýkoliv opakující se vizuální vzor, který může být abstrahován do nezávislé části HTML, CSS nebo JavaScriptu. Takový objekt může být nadále znovupoužit napříč webovou stránkou*“ [17].

OOCSS se tak snaží dosáhnout zejména znovu použitelnosti a rozdělení závislostí známých z objektového programování.

Tato metodika má v zásadě dva základní principy. Těmito principy jsou oddělení struktury od vzhledu a oddělení obsahu od kontejneru [17].

První z principů je v zásadě jasný. Pro vzhled a strukturu by měly existovat dvě separátní třídy. Za strukturu může být v tomto případě považována zejména velikost objektů, tedy jejich šířka a výška, nebo jejich odsazení vůči ostatním objektům. Vzhledem jsou pak myšleny vlastnosti jako barva objektu, font textu v objektu, stínování a tak dále. Dalším pravidlem, které přichází s tímto principem je užívání téměř výhradně třídních selektorů místo spoléhání na selektory elementů. Dále je vhodné vyvarovat se identifikátorům. Už z principu objektově orientovaného postupování není vhodně využívat selektory, které by měly ovlivňovat jen jeden

jediný prvek. Vhodným postupem při využívání tohoto principu je vytváření pravidel s názvy selektorů takovými, aby bylo snadné rozpoznat, která pravidla k sobě patří [17].

Druhý princip, tedy oddělení obsahu od kontejneru, se snaží autora kódu odvrátit od používání zanořených pravidel. To v zásadě znamená, že by uživatel neměl používat pravidla, která vychází z umístění prvku v rámci struktury HTML kódu. Takové prvky mívají často vysokou specifičnost a lze je jen velmi těžko znovupoužít [17].

Je možné setkat se i s jinými variantami těchto principů. Jejich seznam se víceméně odvíjí od zaběhlých pracovních postupů jednotlivých uživatelů a tedy jejich pohledu na tvorbu kódu. V zásadě však všechny tyto principy vychází z těchto původních dvou, tak jak byly navrženy Sullivan.

Tyto principy je možné demonstrovat na objektu typu tlačítko. Tlačítko je prvek, který se na webové stránce objevuje zřídkakdy ojediněle, většinou má nějaký základní design, který je aplikován na všechna tlačítka, kdy často dochází jen ke změně barvy napříč jednotlivými tlačítky. Objekt byl tedy identifikován a teď už jen stačí postupovat dle výše popsaných principů.

Krátká ukázka kódu pro vytvoření tlačítka bez použití OOCSS:

Nejdříve HTML kód:

```
<button class="btn-dulezite">Click me</button>
```

CSS kód:

```
.btn-dulezite{
  color:red;
  width:100px;
  height:50px;
  background-color:blue;
}
```

Na takovém kódu není v zásadě nic špatně, nicméně nedodržuje základní principy OOCSS, v tomto případě nedošlo k oddělení vzhledu od struktury. Správný kód by vypadal následovně:

HTML:

```
<button class="btn btn-dulezite">Click me</button>
```

CSS:

```
.btn{
  width:100px;
  height:50px;
}

.btn-dulezite{
  color:red;
  background-color:blue;
}
```

V HTML kódu přibyla v případě metodiky elementu button jedna třída. Došlo tak k rozdělení struktury od vzhledu. Při vytváření pravidel pro tlačítka odlišného vzhledu už tak nebude nutné znovu opisovat vlastnosti obsažené v pravidle `.btn`. Díky tomuto přístupu se stává CSS nejen přehlednější, ale u větších projektů mnohem kratší. Zde také přichází na řadu možná úskalí spojená s tímto přístupem.

Je třeba mít na paměti, že ne každý projekt může být popsán pomocí objektů. Ne na každý web je možné tyto principy aplikovat. Pro menší projekty pak může být použití této metodiky trochu moc. Pokud se na uživatelově webu objevuje tlačítko jen jednou, je nejspíš zbytečné držet se těchto principů, protože k žádnému znovupoužití stejně nedojde a kód by tak opravdu mohl být zbytečně dlouhý.

4.2 BEM

Název BEM je zkratkou pro Block, Element a Modifier a tato tři slova v podstatě shrnují základní myšlenku fungování této metodiky. Dá se o ní uvažovat jako o jakési nadstavbě metodiky OOCSS, která však více lpí na zavedené syntaxi. Vždyť „block“ v této metodice není v podstatě nic jiného nežli objekt, který byl představen výše.

Metodika BEM byla poprvé oficiálně představena ruskou společností Yandex v roce 2009. V následujících sekcích je tato metodika podrobněji představena (zpracováno podle [18]).

Jako ostatní metodiky, i tato vznikla za účelem zjednodušení organizace kódu, jeho zpřehlednění a zejména jeho znovupoužitelnosti. Tohoto cíle je dosaženo rozdělením kódu do bloků. Součástí těchto bloků jsou takzvané elementy, ty si je možné představit například jako záložku v menu, tlačítko ve skupině tlačítek, či buňku tabulky. Modifikátory jsou pak prvky, které ovlivňují stav, chování, či vzhled bloků a jejich elementů. Na rozdíl od OOCSS je pak každý z těchto prvků pojmenován pomocí speciální syntaxe, která je pro tuto metodiku stěžejní.

Jmenná konvence by se měla řídit těmito zásadami:

1. Názvy pravidel by měly být malými písmeny
2. Pokud je název pravidla složitější, měla by být jednotlivá slova oddělena pomlčkou
3. Název bloku je jmenným prostorem pro jeho elementy a s ním související modifikátory
4. Jméno elementu je od jména bloku odděleno dvěma podtržítky
5. Jméno modifikátoru je od jména bloku nebo elementu odděleno jedním podtržítkem
6. Hodnota modifikátoru je od názvu modifikátoru oddělena jedním podtržítkem
7. Pokud je modifikátor booleanovský (není důležitá jeho konkrétní hodnota, jen to, zdali je či není přítomen) není jeho hodnota uvedena v jeho jméně.

Blok – jak již bylo zmíněno výše, blok je možné připodobnit k objektům, tak jak byly popsaná v OOCSS. Jedná se o jakési prvky webové stránky, u nichž je očekávána znovu použitelnost. Blok by neměl být závislý na tom, kde se na dané stránce nachází. Opět se zde tedy objevují principy OOCSS – důraz na co nejmenší specifičnost, tedy nepoužívání zanořování pravidel do sebe. Stejně tak není vhodné používat jako selektory elementy HTML prvků, ale pro každé pravidlo vytvořit selektor třídní. Bloky se v BEM nepojmenovávají nijak zvláštním způsobem.

Element – element je prvek, který by se nikdy neměl vyskytovat sám, mimo svůj blok. Příkladem elementu může být například záložka rozcestníku. Element je pak

vhodné pojmenovat následujícím způsobem: `.rozcestnik__zalozka{}`. Při pojmenovávání je tedy nutné použít název bloku, jako součást názvu elementu. Díky tomuto postupu lze jednoznačně určit, s jakým blokem tento element souvisí.

Modifikátor – modifikátory se dají rozdělit na dva typy, modifikátory bloků a modifikátory elementů. V zásadě mezi nimi není žádný větší rozdíl. Modifikátory bloků by měly upravit vzhled celého bloku, zatímco modifikátory elementů upravují vzhled jednotlivých elementů. Základní syntaxe zůstává v obou případech stejná, jen je třeba dávat pozor na to, který prvek bude na místě prefixu.

Modifikátor bloku: `.rozcestnik_viditelny{}`.

Modifikátor elementu: `.rozcestnik__zalozka_navstivena{}`.

Použití BEM metodiky se dá pak snadno demonstrovat na příkladu s rozcestníkem ve kterému bude zvýrazněna záložka reprezentující část webu, ve které se uživatel právě nachází.

HTML kód:

```
<ul class="rozcestnik" >
  <li class="rozcestnik__zalozka">
    <a href="">První záložka</a>
  </li>
  <li class="rozcestnik__zalozka rozcestnik__zalozka_navstivena">
    <a href="">Druhá záložka</a>
  </li>
</ul>
```

CSS:

```
.rozcestnik{
  background-color:red;
}
.rozcestnik__zalozka{
  color:red;
}
.rozcestnik__zalozka_navstivena{
  color:green;
}
```

Pomocí této metodiky je na rozdíl od OOCSS snadné jednoznačně nalézt souvislosti mezi různými pravidly. Názvy pravidel však již na první pohled vyžadují více textu. Některá složitější pravidla tak nemusí působit příliš esteticky. Stejně jako tomu bylo

u předchozí metodiky i tato má své plusy a mínusy a jelikož BEM je jakousi nástavbou OOCSS jsou pak misky vah rozloženy podobným způsobem. U menších projektů může být používání BEM zbytečné a může vést k nabobtnání kódu. Také je třeba při návrhu stránky brát jeho používání v potaz, protože ne všude je možné tuto metodiku použít. Pokud je k němu však přistupováno obezřetně, může být velmi užitečným sluhou, který nejen ušetří spoustu psaní, či kopírování, ale učiní kód také mnohem přehlednějším.

4.3 SMACSS

Další blíže popsanou metodikou bude metodika SMACSS neboli Scalable and Modular Architecture for CSS. Tato metodika byla v roce 2011 představena Jonatanem Snookem a jejím základním kamenem je rozdělení částí projektu do pěti kategorií a na základě tohoto rozdělení s těmito částmi odlišně pracovat [19]. Na rozdíl od dvou předchozích metodik se v tomto případě opravdu jedná spíše o tipy, jak nejvíce zefektivnit a nejlépe zorganizovat svůj pracovní proces, nežli o seznam pravidel, které je třeba dodržovat pro správné zachování postupů.

Pěti základními kategoriemi, které se v této metodice objevují jsou pravidla pro základ, layout, moduly, stav a téma. V praktickém použití se tyto kategorie mohou částečně překrývat, v jistých projektech se některé z kategorií také nemusí vůbec vyskytovat. Při používání této metodiky je tak třeba mít na paměti, že se jednotlivé postupy mohou přizpůsobovat konkrétnímu projektu.

Základ – Tyto pravidla udávají defaultní hodnoty vlastností, které se mohou napříč uživatelovou webovou prezentací objevit. Může se jedna například o font, velikost či barvu textu, základní odsazování či vzhled pozadí. Mezi selektory pro deklaraci těchto pravidel se objevují zejména třídy, elementy či pseudotřídy. Naopak selektory obsahující identifikátory by se mezi základními pravidly vyskytovat neměly, protože už z jejich principu nastává předpoklad jejich unikátního výskytu napříč webovou prezentací. Pro tato pravidla není doporučena žádná konkrétní syntaxe [19].

Layout – V této kategorii by se měla objevit zejména pravidla popisující základní části webové stránky. Za tyto části může být považován například header, footer či

postranní panely stránky. Pravidla v kategorii layout by měla být deklarována naopak od základu zejména pomocí identifikátorových selektorů, v opačném případě je také možné používat třídy, zde je ovšem doporučeno použití prefixu „l-„ před názvem každé třídy. Tak může být jednoznačně identifikováno, že se jedná o pravidla layoutu [19].

Modul – Modul může být připodobněn k tomu, co bylo v OOCSS představeno jako objekt a v BEM jako element. Mělo by se tedy jednat o jakousi část webové stránky, která se v ideálním případě vyskytuje na více místech. Moduly by se měly vyskytovat zejména uvnitř layoutů, případně uvnitř jiných modulů. Stejně jako v případě předchozích metodik je zásadně doporučováno vyvarovat se použití selektorů pro elementy a identifikátory a držet se zejména použití selektorů třídních. SMACCS v tomto případě nepředkládá žádnou konkrétní syntaxi. Nicméně doporučuje přijít s nějakým vlastním postupem, jak jednoznačně rozeznat pravidla pro moduly od ostatních [19].

Stav – Další kategorie, která může být přirovnávána k postupům objevujícím se v předchozích metodikách. Zejména pak především s modifikátory vyskytujícími se v metodice BEM. Stav určuje vzhled konkrétního výskytu jak modulu, tak layoutu. Je tak vhodné tato pravidla použít všude tam, kde se konkrétní část webové stránky odlišuje od normy. Opět je tedy možné zmínit tlačítko, kdy tlačítko „STOP“ může mít jinou barvu než všechny ostatní. Případně záložka může být právě navštěvována a tak odlišena od záložek ostatních. Použití stavových pravidel může také naznačovat provázanost s JavaScriptem. Na rozdíl od předchozích metodik není v této kategorii striktně vyloučenou použití direktivy `!important`, opět by k ní však mělo být přistupováno obezřetně. Pro stavová pravidla je pak opět vhodné používat zejména třídních selektorů. Snook potom doporučuje zavést syntaxi skládající se z prefixu „is-“, který je následován názvem stavu. Případně, pokud je stav více specifický, je vhodné použít při vytváření selektoru i název modulu, či layoutu, jehož stav je upravován. V případě navštívené záložky se tak jedná o `.zalozka-is-visited` [19].

Téma – Tato kategorie je jakousi obdobou kategorie stav, nicméně nejde o aplikaci vzhledu na konkrétní moduly a layouty jako spíš na hromadnou změnu vzhledu jejich výskytů. Příkladem může být například tmavé téma – kdy je vzhled většiny

prvků na webové stránce změněn do tmavších odstínů. Tato témata je pak vhodné rozdělit do samostatných CSS souborů, tak aby byla zachována přehlednost a snadná upravitelnost [19].

Stejně jako v předchozích metodikách i zde je možné setkat se s doporučeními ohledně hnízdění. Nicméně tam, kde bylo v předchozích případech nedoporučováno, nebo od jeho používání striktně odrazováno, v tomto případě se jedná spíše o doporučení ohledně používání s mírou. Pokud dochází k hlubšímu hnízdění, je pak vhodné místo složitého selektoru raději vytvořit novou třídu. Jak hluboké je pak už hlubší hnízdění, to je na každém konkrétním autorovi [19].

Jak už bylo řečeno, metodika SMACSS je spíše souborem tipů a triků, které by měly uživatele navést správným směrem. Stěží je proto možné přijít s konkrétními nedostatky této metodiky. Naopak, zejména pro uživatele, kteří ještě neobjevili „svůj systém“ může být vhodnou příručkou, jak zavést v chaosu pořádek a to bez nutnosti od základu měnit jakým způsobem uživatel ke svému kódu a webu celkově přistupuje.

4.4 Navázání metodik na nástroje

Na první pohled se může dle výše uvedených informací zdát, že CSS procesory a metodiky jsou přístupy, které není možné skloubit dohromady. I když se oba přístupy snaží dosáhnout stejného cíle, každý k němu směřuje z jiného konce. Některé přístupy CSS procesorů jdou dokonce proti všem zásadám zmíněných metodik. Kupříkladu hnízdění, či rozšíření, jsou přístupy, které ve výsledném CSS dosahují stavů, od kterých metodiky zásadně odrazují. Pravdou však zůstává, že vhodným spojením metodik a procesorů lze dosáhnout vysoké efektivity a to za velmi malou cenu.

Především během používání metodik jako OOCSS a BEM se může uživatel setkat s velmi častým opakováním názvů svých pravidel. Také je třeba myslet na to, že autor je se svým kódem spokojený hned na poprvé jen zřídkakdy. To vede k častým změnám a v případě metodik tedy k neustálému opakování a přepisování názvů u mnoha pravidel. V tu chvíli však přichází na pomoc hnízdění, především **odkazování na rodiče**.

Pro jednoduchost je opět použit příklad s návštěvou záložky v rozcestníku. V tomto případě v jazyce Sass, ale samozřejmě je možné použít jakýkoliv jazyk, či nástroj, který tuto funkci podporuje. Nejefektivněji je možné sílu této kombinace demonstrovat na metodice BEM.

Ukázka vhodného použití odkazu na rodiče v kombinaci s BEM:

```
/* Sass */
.rozcestnik {
  background-color:red;
  &__zalozka{
    color:red;
      &_navstiven{
    color:green;
  }
  }}

/* Zkompilované CSS */
.rozcestnik{
  background-color:red;
}
.rozcestnik__zalozka{
  color:red;
}
.rozcestnik__zalozka_navstivena{
  color:green;
}
```

Tento přístup dokáže zejména v rozsáhlém projektu ušetřit mnoho úsilí. Pokud si uživatel rozmyslí název své třídy, či přidá novou, musí při psaní nativního CSS přepisovat kód na mnoha místech svého souboru. V tomto případě stačí přepsat jen předka na nejvyšší úrovni a změna se projeví u všech jeho potomků. Kód je navíc v takovém případě přehledně uspořádán a všechna spolu související pravidla se nachází v jednom jediném bloku. Obdobný postup lze samozřejmě použít i například při zápisu pravidel ve stavové kategorii v metodice SMACSS.

Další možností, jak zefektivnit používání metodiky BEM je s využitím pluginů pro PostCSS, jmenovitě **PostCSS BEM**. Tento plugin výrazně zjednodušuje vytváření BEM struktury a to za pomocí direktiv `@component` – značící blok, `@descendent` – značící element a `@modifier` – značící modifikátor. Tyto direktivy mohou být použity na místě odkazu na rodiče, v takovém případě uživatel nemusí doplňovat znaky, které jsou vyžadovány BEM syntaxí, PostCSS je za něj doplní samo.

Obdobného výsledku jako v předchozí ukázce je pak možné dosáhnout následujícím zápisem:

```
@component rozcestnik {
  background-color: red;
  @descendent zalozka {
    color: red;
    @modifier navstiven {
      color: green;
    }
  }
}
```

Smyslem metodik je mimo jiné učinit kód přehlednějším. Mnoho z nich tedy nabádá k řádnému oddělení jednotlivých sekcí logicky od sebe. V metodice BEM se může jednat o rozdělení na základě bloků, či modulů, u metodiky SMACCS je to rozdělení do kategorií. Ideálním způsobem, jak od sebe jednotlivé části oddělit, je pomocí jednotlivých CSS souborů. Takový přístup se pak může stát v případě rozsáhlejšího projektu problematický. Ke každému souboru je přistupováno zvlášť, to vede k poklesu výkonu a dlouhému načítání stránky. Zde opět přichází na řadu procesory a konkrétně jejich funkce **import**. Pravidla je v takovém případě možné na základě jejich příslušnosti rozdělit do jednotlivých souborů a tyto soubory následně naimportovat do jednoho hlavního, na který stačí v HTML kódu odkázat.

Existuje mnoho dalších způsobů, jak propojit metodiky s CSS procesory. Metodika BEM například nemusí končit jen u názvů selektorů, ale může být zavedena i nad proměnnými, čímž dochází ke značnému zpřehlednění. Je možné se setkat s mnohými tipy a přístupy, jak procesory a metodiky co nejlépe provázat a dosáhnout tak nejlepší efektivity. Nicméně je na každém jednotlivém uživateli, jakou cestou se vydá a jaký postup se stane tím ideálním právě pro něj.

5 Srovnání postprocesorů

To, zdali jsou postprocesory dalším krokem v evoluci vývoje webových prezentací, či jen slepou uličkou, ukáže až čas. Nicméně je pravdou, že dokáží odstranit spousty nedostatků současného vývoje webu, dokáží zefektivnit práci nejen se samotným CSS, ale dokáží být i efektivními pomocníky současným preprocesorům, dokáží ušetřit velkou spoustu času, a to jen za cenu investice několika málo minut do jejich zavedení do pracovního procesu.

Následující stránky si dávají za cíl nejen přiblížit základní funkcionalitu a práci s těmito nástroji, ale pokusí se také čtenáře přesvědčit o tom proč a jak jim dát šanci. Pokusí se odpovědět na to, nejen který z postprocesorů je ten lepší a zejména zdali je vhodné dát těmto nástrojům přednost před preprocesory. Tato práce tak může být považována za jakési navázání na bádání Bc. Adama Černého, který detailně srovnal preprocesory ve své práci „CSS preprocesory jako efektivní nástroj pro tvorbu frontendu“ [20].

Jelikož se jedná o nástroje relativně mladé, k mnoha z nich neexistuje konkrétní a přehledná dokumentace. Jelikož lze najít mezi preprocesory a postproceosry mnohé paralely, může být u některých nástrojů tohoto faktu využito právě při srovnávání. V některých případech autor upřednostnil srovnání na základě vlastností a schopností, které sám považuje jako běžný uživatel za nejdůležitější. Cílem je ovšem poskytnout zcela objektivní pohled na věc.

Existuje celá řada postprocesorů, které si dávají za cíl řešit konkrétní jednotlivé problémy, takovými nástroji jsou například **Pleeease**, který řeší například problémy s prefixy, či jinou kompatibilitu napříč prohlížeči [21]. Za další takový může být považován například nástroj **CSSNext**, jehož smyslem je umožnit využívání budoucí CSS syntaxe již v současnosti. Tyto nástroje ovšem nepřináší komplexní řešení v takovém rozsahu, aby o nich mohlo být uvažováno jako o možných náhradách preprocesorů.

Nicméně přece jen existují nástroje, které mohou být svým komplexním přístupem považovány za obstojné konkurenty preprocesorů. Takovými nástroji jsou **PostCSS** a **Stylecow** a právě k jejich srovnání dojde.

5.1 Představení postprocesorů

5.1.1 PostCSS

PostCSS je v současnosti a v dohledné době pravděpodobně nejpopulárnějším nástrojem na poli postprocesorů. Tento nástroj se stal de facto předobrazem toho, jak by měl postprocesor vypadat.

Samotné PostCSS je jen jakousi API, která slouží k analýze a modifikaci CSS. PostCSS samo o sobě jako takové však žádné modifikace nevykonává. K tomu jsou třeba pluginy, které mohou být považovány za jakýsi stavební kámen tohoto nástroje. Těchto pluginů může být celá řada. Od převádění syntaxe CSS do českého jazyka, přes proměnné a mixiny, tak jak jsou známe z preprocesorů, až po nástroje umožňující využívat syntaxi CSS tak, jak zatím existuje jen v experimentálních podmínkách a v současnosti není prohlížeči podporována. Tento nástroj vzniká již od svého počátku za pomocí jazyka JavaScript. Nejinak je tomu u jeho pluginů [22].

Za první vlaštovku PostCSS může být považován takzvaný Autoprefixer. Jedná se o postprocesor sloužící ke generování vendor-prefixů za účelem kompatibility CSS vlastností napříč prohlížeči. Tento postprocesor vznikl ještě před samotným uvedením PostCSS a to za pomoci nástroje zvaného Rework, který lze považovat za jakéhosi prapředka všech postprocesorů. Autoprefixer vytvořil Andrey Sitnik, který se později stal hlavním vývojářem PostCSS. S první stabilní verzí tohoto nástroje se mohli uživatelé setkat na podzim roku 2013 [15].

V souvislosti s PostCSS je možné narazit na pojem vysoká modulárnost. Ta se projevuje zejména tím, že PostCSS umožňuje snadné vytváření pluginů na straně jeho uživatelů, každý z těchto pluginů si poté žije svým vlastním životem. Pro přidání nové funkcionality do svého pracovního procesu pak není nutné čekat na aktualizaci celého nástroje, nýbrž zavítat na webové stránky s pluginem spojené a odtud jej stáhnout. V současnosti PostCSS nabízí přes 200 pluginů [22].

5.1.2 Zavedení PostCSS do pracovního procesu

5.1.2.1 Instantní spuštění

Stejně jako je tomu u preprocesorů i v případě PostCSS musí dojít před tím, než je prohlížeč schopen uživatelem napsaný kód zpracovat, k jeho kompilaci. A stejně jako je tomu u preprocesorů, PostCSS nabízí více způsobů, jak toho dosáhnout.

První a nejjednodušší řešení a to zejména pro uživatele, kteří si příliš nerozumí s příkazovou řádkou přichází v podobě aplikací s grafickým rozhraním. Jedním z nich je například aplikace Prepos, tu stačí jednoduše stáhnout, nainstalovat, spustit a následně zadat cestu k požadovanému souboru. Nevýhodou této aplikace je nemožnost instalace pluginů. Toto rozhraní nabízí jen omezený množství pluginů, které již nelze dále rozšiřovat.

Dalším způsobem, jak se vyhnout příkazové řádce je použitím internetové aplikace CodePen [23], ta na rozdíl od Preposu nabízí mnohem širší portfolio pluginů, které je navíc možné dále rozšiřovat. Tato aplikace mimo jiné podporuje celou řadu dalších CSS procesorů a jejich fungování dokáže převádět do podoby běžné webové stránky v reálném čase. Webový charakter tohoto řešení ovšem naznačuje spíše testovací přístup nežli cestu, jak efektivně vytvářet webové prezentace.

5.1.2.2 Příkazová řádka

Jelikož předchozí řešení umožňuje využívat jen značně omezené schopnosti PostCSS, dochází k jeho používání zejména pomocí příkazové řádky. Stejně jako je tomu v případě preprocesorů i zde je nutné spolupracovat s prostředím Node.js, se kterým byl čtenář již seznámen v souvislosti s preprocesory. Tento přístup v zásadě přináší nejjednodušší cestu, jak využívat schopnosti tohoto postprocesoru v plné míře.

Pokud se uživatel seznámil s používáním Node.js, stačí do běžné příkazové řádky zadat následující příkaz:

```
npm install -g postcss-cli
```

Pro co nejjednodušší představení postačí nainstalovat příkazovou verzi PostCSS globálně, ale samozřejmě je i možné nainstalovat lokálně do konkrétní složky.

Jak již bylo řečeno, jádro PostCSS samo o sobě příliš mnoho nedokáže, proto je nutné nainstalovat požadované pluginy. Jména těchto pluginů nejdou ve všech případech naproti uživatelovu očekávání a může se tak stát, že se při instalaci objeví nečekaný problém, proto je vhodné před instalací navštívit domovské stránky pluginů a následovat autorových pokynů.

Pokud chce uživatel dosáhnout například automatického prefixování pomocí vendor prefixů a následné minifikace, tedy odstranění všech zbytečných mezer a odsazení, které při strojovém čtení CSS nejsou potřeba a přidávají tak zbytečné kilobyty navíc, musí tyto pluginy nejdříve nainstalovat následujícími příkazy, v tomto případě opět globálně:

```
npm install -g autoprefixer
npm install -g cssnano
```

Proběhla-li instalace úspěšně, uživatel se může pustit do postprocesování. Ke správnému použití těchto pluginů už pak stačí jen zadat příkaz:

```
postcss zdroj/style.css -u cssnano -u autoprefixer -o cil/styles.css
```

Do běžné řeči je možné tento příkaz přeložit jako: „PostCSS, vezmi soubor style.css, zpracuj jej pomocí cssnano a autoprefixeru, výsledek ulož jako styles.css do složky cil.“

Za předpokladu, že existuje alespoň zdrojový CSS soubor a všechny pluginy jsou nainstalovány správným způsobem, v cílové složce „cil“ dojde k vytvoření, případně aktualizaci souboru „styles.css“. V opačném případě zahlásí příkazová řádka chybu s informací o tom, ve kterém z kroků došlo k selhání.

Nevýhodou tohoto přístupu je nutnost vypisování názvů jednotlivých pluginů za sebou. Takový přístup se stává při používání většího množství pluginů vysoce neefektivní. Naštěstí autoři tohoto nástroje mysleli i na tyto případy. Řešením je javascriptový soubor „postcss.config.js“. Tento soubor umožňuje schraňovat přehledně a na jednom místě všechny uživatelem užívané pluginy. Ty stačí opět

nainstalovat, v tomto případě však ideálně lokálně a umístit odkaz na ně do tohoto souboru.

Soubor „postcss.config.js“ s již přidanými referencimi na pluginy pro autoprefixování a minifikaci:

```
module.exports = {
  plugins: [
    require('autoprefixer'),
    require('cssnano')
  ]
}
```

V takovém případě už poté stačí do příkazové řádky zadat jen následující příkaz a PostCSS se o vše postará:

```
postcss zdroj/style.css -o cil/styles.css
```

Soubor „postcss.config.js“ navíc umožňuje přidávat pluginům nejrůznější nastavení, či přidané funkce, ke kterým by uživatel přistupoval z čisté příkazové řádky velmi složitě.

Dalším způsobem, jak používat PostCSS pomocí příkazové řádky, je pak s využitím takzvaných task runnerů. Task runnery jsou nástroje, které slouží k automatizaci práce. Pokud by chtěl uživatel propojit například preprocesor Sass s PostCSS, automaticky aktualizovat, či mazat údaje v databázi, musel by tyto kroky řešit jednotlivými příkazy. Task runnery tento problém řeší. Při jejich správném použití pak stačí zadat jediný příkaz a o všechno ostatní se postará task runner. Mezi ty nejznámější patří Gulp a Grunt [22].

5.1.3 Stylecow

Stylecow je o poznání menším nástrojem nežli PostCSS. Poprvé byl představen v polovině roku 2014 web designerem Oscarem Oterem. Obdobně jako PostCSS si dává za cíl řešit problémy, na které samotné preprocesory nestačí a stejně jako výše zmíněný postprocesor na to jde za pomocí stahovatelných pluginů. Byl napsaný v jazyce JavaScript a nejinak je tomu u jeho pluginů [24].

Pluginů pro Stylecow je výrazně méně, než jak je tomu u jeho staršího kolegy. Je jich jen několik desítek a všechny byly vytvořeny samotným Oterem [24]. Stylecow tak

zdaleka nenabízí takové množství možností, jako PostCSS. Stejně jako v případě předchozího zmíněného nástroje, je použití pluginů naprosto nezbytné pro jakoukoliv pokročilejší práci s CSS. Všechny základní pluginy jsou pak při defaultním nastavení nainstalovány společně s jádrem, což je oproti PostCSS výrazně snazší přístup. Jelikož poslední aktualizace jádra tohoto nástroje proběhla již více než před rokem a komunita okolo něj nebyla nikdy příliš rozsáhlá, nevypadá to s jeho budoucností příliš nadějně. Přes to se jedná o projekt, který stojí za pozornost a minimálně jako odrazový můstek do světa procesorů, je plně dostačující.

5.1.4 Zavedení Stylecow do pracovního procesu

Na rozdíl od PostCSS v případě Stylecow neexistují žádné podpůrné nástroje, které by umožňovaly jeho snadnou implementaci bez použití příkazové řádky. Na druhou stranu je v jeho případě práce s příkazovou řádkou o poznání snazší, než je tomu v případě jeho konkurenta. I zde je nutné využívat nástroje Node.js, pokud je nainstalován správným způsobem, stačí jen do příkazové řádky zadat příkaz:

```
npm install stylecow -g
```

Výše uvedeným příkazem došlo k instalaci Stylecow, k tomuto nástroji je tak možné přistupovat globálně. Samozřejmostí je pak existence lokální varianty instalace.

Dále je nutné zadat příkaz následující:

```
stylecow init
```

Tento přikaz slouží k vytvoření iniciačního souboru, ze kterého vychází veškeré další nastavení a hodnoty pro pluginy tohoto nástroje. Při jeho vytváření je uživatel dotázán na základní nastavení a je mu umožněno vybrat pluginy, k jejichž instalaci dojde. Umožňuje také zadat cestu k zdrojovému a cílovému CSS souboru. Po zvolení preferovaných variant dojde na disku k vytvoření souboru ve formátu JSON, defaultně se pak jedná o soubor „stylecow.json“. Tyto iniciační hodnoty není problém v budoucnu kdykoliv pozměnit.

Pokud je pro uživatele žádoucí využívat pluginu, k jehož stažení nedošlo při vytváření iniciačního souboru, je třeba v případě prefixeru zadat následující příkaz:

```
npm i stylecow-plugin-prefixes
```

Po jeho stažení je pak nutné upravit v iniciačním souboru sekci s pluginy, v tomto případě je nutné doplnit do vygenerovaného souboru výraz „plugins“:

```
//Iniciační soubor vygenerovaný nástrojem Stylecow
{
  "files": [
    {
      "input": "zdroj.css",
      "output": "cil.css"
    }
  ],
  "support": {
    "explorer": 11,
    "edge": 17,
    "firefox": 61,
    "chrome": 68,
    "safari": 11.1,
    "opera": false,
    "android": false,
    "ios": 11
  },
  "plugins": [
    "prefixes"
  ],
  "code": "normal",
  "map": "auto"
}
```

V tuto chvíli je vhodné blíže představit části výše zobrazeného kódu. V první sekci „files“ jsou zobrazeny soubory, se kterými Stylecow pracuje, v tomto případě tedy CSS soubory „zdroj.css“ a „cil.css“. V sekci „support“ je možné se setkat s výčtem podporovaných prohlížečů a jejich verzí. To je vhodné například při vytváření vendor-prefixů. Na základě tohoto seznamu se vytvoří prefixy pro vybrané prohlížeče. V sekci „plugins“ se pak nachází seznam všech uživatelem používaných pluginů. Vlastnost „code“ pak udává, jakým způsobem bude finální CSS kód vygenerován. Druhou možností, která zde není zobrazena, je možnost „minify“, ta vede k minifikaci kódu, tedy odstranění přebytečných mezer a odsazení. Vlastnost „map“ potom udává, jakým způsobem bude provedeno mapování k původnímu CSS. Tato funkce se projevuje zejména ve chvíli, kdy uživatel upravuje vzhled stránky přímo v prohlížeči. Zkompilovaný kód se liší od toho původního. Mapování tak slouží k tomu, aby byl prohlížeč schopen odkázat na správné místo v původním nezkompilovaném kódu. Volba „auto“ pak nechá na procesoru zvolit nejlepší variantu. Dalšími možnostmi jsou pak „file“, vygenerování mapy do externího

souboru, „embed“, vygenerování mapy do finálního CSS a „none“, které zamezí generování jakékoliv mapy.

Ke zpracování uživatelského kódu dochází zadáním příkazu `Stylecow` do příkazové řádky.

Stylecow obdobně jako PostCSS umožňuje napojení do task runnerů a to zejména Grunt a Gulp [24].

5.2 Srovnání

V této kapitole již dojde k samotnému srovnání dvou výše zmíněných postprocesorů. Schopnosti, k jejichž srovnání dojde, byly záměrně zvoleny na základě své podobnosti s preprocesory, možnosti propojení s metodikami a zároveň na základě jejich užitečnosti pro vývoj běžné webové stránky. Tam, kde nabízejí postprocesory více přístupů k řešení problematik, může dojít ke srovnání těchto nástrojů i v rámci jednoho postprocesoru.

Na úvod každé sekce bude detailně popsáno, jak se daný nástroj staví k vybrané problematice. Tam, kde je to vhodné, budou uvedeny ukázky kódu.

Na konci každé podkapitoly jednoho nástroje dojde ke zjištění stavu vývoje a popularity nástroje. A to zejména z pohledu data poslední aktualizace, oblíbenosti ve formě hvězdiček, změn čekajících na zavedení, které mohou být ukazatelem aktivního vývoje, či počtu stažení týdně. K těmto účelům poslouží správce balíčků Node Package Manager a repositář GitHub, které mohou být v současnosti považovány za centra technologického dění. Po představení přístupu obou nástrojů bude na konci každé kapitoly vyhodnocení srovnání.

Nástroje budou hodnoceny dle následujících kritérií:

Postprocesor vůbec nabízí nástroj, který řeší danou problematiku – **3 body.**

Pokud jeden z nástrojů přináší schopnosti, který druhý ne, získává **1 bod,** pokud tyto schopnosti přináší oba nástroje, získají oba po **1 bodu.** Pokud nabízí schopnost oba nástroje a dá se jednoznačně určit, který přístup je lepší, získá jeden nástroj **1 bod** a druhý **body 2.**

Popularita a stav vývoje – nástroj, který se zdá býti v lepším stádiu vývoje získá **1 bod**.

Před závěrečným vyhodnocením dojde ke srovnání celkové popularity nástrojů. Poté již bude následovat závěrečné vyhodnocení.

5.2.1 Prefixování

Prefix neboli vendor prefix je způsob, jak prohlížeči sdělit, jakým postupem zpracovávat CSS kód tak, aby byl kompatibilní a prováděl uživatelem požadovanou funkcionalitu napříč všemi prohlížeči. Tento přístup se začal zásadně prosazovat s příchodem CSS3, kdy byla představena a (a stále je představována) řada nových funkcí, jejichž podpora není na mnohých z prohlížečů dosud zcela stoprocentní [1].

Výčet prefixů pro ty nejpopulárnější z prohlížečů je podle [1] následující:

`-moz-` (Firefox)

`-o-` (Opera)

`-ms-` (Internet Explorer)

`-webkit-` (Safari, Chrome)

Praktickým příkladem nechť je vlastnost „transition“, ta umožňuje přechod mezi více stavy vlastností jednotlivých prvků webové stránky. Pokud uživatel vyžaduje například po přejetí myši po prvku div změnit jeho podobu:

```
div{
  width: 100px;
  height: 100px;
  transition: width 3s;}

div:hover{
  width:200px;}
```

Takový kód se zdá být na první pohled v pořádku, ovšem jeho zápis není korektní. Aby byl tento přechod proveden korektně napříč různými prohlížeči, v tomto případě i v prohlížečích Safari, je třeba kód upravit tak, že pod řádek

```
transition: width 3s;
```

Bude dopsán řádek:

```
-webkit- transition: width 3s;
```

Při vyšším počtu vlastností, které vyžadují prefixy, je toto řešení velmi únavné a časově náročné. Tvůrce kódu musí také neustále pamatovat na to, jaké vlastnosti prefixy vyžadují. Řešením tohoto problému jsou automatické prefixery.

V následujícím srovnání bude brán zřetel zejména na to, jak detailně přistupují jednotlivé nástroje k přidávání prefixů, jak náročné je pro uživatele vybrat ideální kombinaci prohlížečů, pro které je generovat, případně zdali nabízí nějaké nadstandardní řešení.

5.2.1.1 PostCSS

V PostCSS je problematika prefixů řešena pomocí nástroje zvaného Autoprefixer. Tento nástroj umožňuje automaticky do výsledného CSS kódu doplnit potřebné prefixy tak, aby byla zachována správná kompatibilita napříč prohlížeči a to na základě konkrétních požadavků uživatele.

5.2.1.1.1 Autoprefixer - řešení

Jelikož je hlavním cílem tohoto nástroje umožnit uživateli zcela zapomenout ne existenci prefixů, k využívání tohoto nástroje není třeba učit se nové syntaxe, či změně přístupu. Stačí jednoduše psát vlastnosti tak, jak je uživatel zvyklý a po uložení už dojde k vygenerování korektního kódu bez jakýchkoliv neobvyklých zásahů autora do CSS. Defaultně tento nástroj podporuje poslední dvě verze prohlížečů, které mají větší než půlprocentní zastoupení mezi prohlížeči.

Autoprefixer neslouží jen k práci s vlastnostmi, ale umožňuje také upravovat například pseudo-elementy, jako je `::placeholder`, u nichž není zaručena 100% kompatibilita napříč prohlížeči.

K získávání údajů o tom, které prohlížeče vyžadují prefixy k jakým vlastnostem, využívá databáze „Can I Use“ [25].

Zde krátká ukázka zpracování pomocí Autoprefixeru:

```
/* Před zpracováním */
::placeholder{
 color:blue;
}

/* Po zpracování */
::-webkit-input-placeholder{
color:blue;}

:-ms-input-placeholder{
color:blue;}

::-ms-input-placeholder{
 color:blue;}

::placeholder{
 color:blue;}
```

Užitečnou schopností tohoto pluginu však může být navíc možnost upřesňovat chování pomocí komentářů, kdy je možné například vepsáním komentáře `/* autoprefixer: off */` zakázat prefixování v celém bloku selektoru, zatímco u zbytku kódu dojde k jeho běžnému zpracování.

5.2.1.1.2 Autoprefixer - volby

Tento nástroj nabízí celou řadu možných nastavení, díky kterým může uživatel upravit jeho práci k obrazu svému. Zde následuje výčet těch nejzajímavějších u nich:

browsers - tato, bezesporu nejužitečnější volba, umožňuje uživateli upřesnit, pro které z prohlížečů prefixy generovat a to nejen jako jejich výčet, ale nabízí možnost generovat prefixy například podle popularity jednotlivých prohlížečů, či upravit verze, se kterými je třeba při zpracování kódu počítat. Tento výběr je velmi rozsáhlý a nabízí celou řadu možností, jak tato nastavení kombinovat.

remove - toto nastavení umožňuje odstraňovat zastaralé prefixy. Je tedy velmi užitečné pro případy, kdy uživatel vezme svůj starší kód s již doplněnými prefixy a rozhodne se jej aktualizovat. Toto nestavení je defaultně nastavené na true, pracuje tedy automaticky.

cascade - toto nastavení upravuje způsob, jakým se vygenerované řádky s prefixy zarovnávají.

Všechny tyto volby byly otestovány na základě dokumentace, která je k nalezení na webu Autoprefixeru [25].

5.2.1.1.3 Autoprefixer - stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [25] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

AUTOPREFIXER	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	28.3.2019
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	1
GITHUB HVĚZDIČKY	16 857
NPM STAŽENÍ TÝDNĚ	7 731 368

Tabulka 2: Popularita a stav vývoje nástroje Autoprefixer

5.2.1.2 **Stylecow**

5.2.1.2.1 Prefixes - řešení

V nástroji Stylecow se kompatibilita napříč prohlížeči řeší pomocí pluginu prefixes, stejně jako Autoprefixer využívá webu „Can I Use“ [24]. Kritéria pro tato pravidla jsou pak zadána při vytváření iniciačního souboru. Při nutnosti tato kritéria změnit je nutné tento soubor upravit dle uživatelových požadavků, stejně jako Autoprefixer dokáže vygenerovat předpony jak pro CSS vlastnosti, tak pro pseudo-elementy a k této funkcionalitě nevyžaduje žádnou další přidanou syntaxi.

Následuje krátká ukázka práce prefixeru.

Zpracování pomocí Prefixes:

```
/* Kód před zpracováním */
::placeholder{
  color:blue;}

/* Kód po zpracování */
::-ms-input-placeholder{
   color: blue;}

::-webkit-input-placeholder{
 color: blue;}

::placeholder{
 color: blue;}
```

Z ukázky je zřejmé, že tento nástroj vygeneroval o pravidlo méně nežli nástroj předchozí. Po nahlédnutí do databáze „Can I Use“ se tento výsledek ukazuje jako správný, nicméně je pravdou, že pravidla vygenerovaná předchozím nástrojem jsou ve skutečnosti korektním způsobem zpracování tohoto problému, proto je možné považovat tuto neúplnost za nedostatek.

Tím v podstatě výčet schopností prefixeru končí.

5.2.1.2.2 Prefixes - stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [24] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

PREFIXES	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	18.4.2017
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	0
NPM STAŽENÍ TÝDNĚ	11

Tabulka 3: Popularita a stav vývoje nástroje Prefixes

5.2.1.3 **Závěr**

Oba postrprocesory získávají po **3 bodech** za řešení prefixů.

Použití tohoto nástroje se v syntaxi nijak neliší a to ani od nativního CSS. Nicméně Stylecow umožňuje snazší nastavení kritérií a to již při vytváření iniciačního souboru a při vybírání základních hodnot tak není nutné manuálního zápisu do žádného souboru, zatímco Autoprefixer vyžaduje pro změnu základních kritérií manuální zápis. Stylecow tak získává **2 body** a Autoprefixer **1**.

Oba nástroje nabízí řešení pro pseudo-elementy, získávají tak po **1 bodu.**

Jak bylo demonstrováno na ukázce kódu, je ve své primární práci, tedy generování prefixů, Autoprefixer důkladnější a také nabízí mnohem rozsáhlejší možnosti, jak upravovat pravidla pro generování prefixů, proto získává **2 body** a nástroj Stylecow jen **1**.

Autoprefixer umožňuje upravovat jen určité části kódu – **1 bod**

Autoprefixer umožňuje automaticky mazat zastaralé prefixy – **1 bod**

V neposlední řadě pak Autoprefixer dosahuje mnohem vyšší popularity a stability na úrovni vývoje. Jedná se o nejpoužívanější PostCSS plugin a na jeho vývoji a popularitě se tento fakt odráží. Získává tak další **bod** k dobru

PREFIX	**POSTCSS**	**STYLECOW**
NABÍZÍ	3	3
SYNTAXE A SPUŠTĚNÍ	1	2
PSEUDO-ELEMENTY	1	1
DŮSLEDNOST PREFIXŮ	2	1
JEN ČÁST KÓDU	1	0
MAZÁNÍ ZASTARALÝCH PREFIXŮ	1	0
POPULARITA A VÝVOJ	1	0
CELKEM:	10	7

Tabulka 4: Vyhodnocení prefixovacích nástrojů

Vyšší počet bodů: PostCSS – Autoprefixer

5.2.2 Proměnné

Proměnné fungují mezi procesory obdobným způsobem, jako v preprocesorech, či programovacích jazycích. Během srovnání bude kladen důraz především na syntaxi, interpolaci, rozsah a datové typy proměnných. Dále je u některých nástrojů nutné oddělit schopnosti nástroje od těch, které přináší nativní CSS.

5.2.2.1 **PostCSS**

V PostCSS je možné pracovat s proměnnými pomocí několika různých pluginů, v následující části budou představeni dva zástupci PostCSS pluginů, jelikož každý z nich přistupuje k proměnným zcela odlišně.

5.2.2.1.1 Simple Variables - řešení

První, čeho si uživatel může všimnout, při seznamování s tímto nástrojem je syntaxe. Jelikož se tvůrci netají faktem, že jeho syntaxe, potažmo i funkcionalita vychází z používání proměnných v preprocesoru Sass [27], není překvapením, že samotná deklarace proměnné probíhá pomocí znaku „$“ a pomocí tohoto znaku probíhá také přistupování k jejich hodnotám. K přiřazování dochází pomocí dvojtečky.

Deklarace proměnné:

```
$sloupec: 200px;
```

Při interpolaci, tedy schopnosti používat proměnné jako názvy selektorů, či samotných vlastností jsou proměnné deklarovány obdobným způsobem, nicméně jejich volání je odlišné.

Takové proměnné se volají také pomocí „$“, nicméně musí být uzavřeny v kulatých závorkách.

Zpracování interpolace pomocí Simple Variables:

```
/* Deklarace proměnných */
$menu:.menu;
$barva: color;

/* Přistupování k hodnotám proměnných */
$(menu)_link{
background-$(barva): red;}

/* Výsledné CSS */
.menu_link{
background-color: red;}
```

Scoping neboli rozsah hledání proměnných zde funguje globálně. To znamená, že nehledě na to, zda došlo k deklaraci proměnné uvnitř, či vně pravidla, je hodnota proměnné dostupná odkudkoli. Pokud je proměnná deklarována až poté, co je k ní v kódu přistupováno, zachová se PostCSS stejně, jako kdyby neexistovala, zahlásí chybovou hlášku a ke kompilaci nedojde. Při deklaraci proměnných uvnitř zanořených pravidel, funguje scoping také obdobně. Pokud je proměnná deklarovaná v rodičovském bloku, všichni jeho potomci mohou tuto proměnnou používat. Je-li hodnota této proměnné v toku zanořeného kódu změněna, použije se první výskyt nacházející se nad jejím voláním.

Simple Variables ve svém základu podporuje následující datové typy:

- Čísla – 1.5, 10, 3px
- Textové řetězce – s uvozovkami i bez
- Barvy – blue, #b30000, rgba(255, 0, 0, 0.5)
- Seznam hodnot, který se dá oddělit čárkou - "Times New Roman", Times, serif;
- Odkazy na další proměnné: $color: $barva
- Složené funkce: $color: calc(10 + $hodnota)

5.2.2.1.2 Simple Variables - volby

Variables - Pokud je tato vlastnost nastavena, umožňuje pluginu načítat proměnné, které jsou definovány v externím souboru. Když je později tento plugin spuštěn, aplikuje seznam těchto proměnných na každý CSS soubor, který je mu vystaven.

onVariables - Vytvoří seznam všech proměnných v dokumentu, se kterým se dá následně pracovat

Silent - Umožňuje vygenerovat CSS soubor i přes to, že obsahuje chybné proměnné. V takovém případě jsou proměnné zobrazeny ve výsledném CSS stejným způsobem, jakým jsou zapsány v původním souboru.

Všechny tyto volby byly otestovány na základě dokumentace, která je k nalezení na webu Simple Variables [27].

5.2.2.1.3 Simple Variables - stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [27] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

SIMPLE VARIABLES	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	18.2.2019
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	309
NPM STAŽENÍ TÝDNĚ	175 879

Tabulka 5: Popularita a stav vývoje nástroje Simple Variables

5.2.2.1.4 Custom properties - řešení

Na rozdíl od předchozího pluginu se tento nesnaží o imitaci syntaxe některého z preprocesorů, ale ani nepřináší žádnou vlastní. Vychází ze syntaxe takzvaných Custom Properties neboli proměnných, známých z nativního CSS. Deklarace probíhá pomocí dvojité pomlčky, tuto deklaraci je navíc nutné provést uprostřed speciální pseudotřídy `:root`. K použití hodnoty proměnných následně dochází pomocí funkce `var()`.

Ukázka zpracování proměnných pomocí Custom properties:

```
/* Deklarace v pseudotříde :root */
:root {
    --color: blue;}

/* Přistupování k proměnné */
h1 {
  color: var(--color);}


/* Zpracované CSS */
:root {
    --color: blue;}

h1 {
  color: blue;
  color: var(--color);}
```

Jak je z ukázky patrné, tento plugin nejenže generuje obsah zavolané proměnné, ale společně s ním i samotné generování a následné volání této proměnné. Generování samotných proměnných je defaultně povoleno, nicméně je možné jej pomocí jedné z voleb tohoto pluginu vypnout.

K zajímavé situaci dochází při načtení tohoto vygenerovaného souboru prohlížečem, tedy alespoň jedním z moderních prohlížečů, který podporuje práci s proměnnými v nativním CSS.

Při kontrolování jednotlivých prvků je možné zjistit, že řádek s dosazenou proměnnou je přeskočen a aplikuje se až řádek pod ním, tedy `color: var(--color);`.

Pokud je ovšem obdobná situace aplikována například na prohlížeč Internet Explorer, je možné zjistit, že prohlížeč CSS zpracovává přesně opačným způsobem, to znamená, že pro obarvení elementu využívá řádku `color: blue;`, zatímco řádek pod ním je pro prohlížeč nesrozumitelný.

Tento plugin bohužel neumožňuje jakoukoliv interpolaci, jak už název napovídá, jedná se o „vlastní vlastnosti“, z toho tedy plyne, že dochází jen k úpravám a deklarací vlastností a není tedy možné pomocí tohoto nástroje volat vlastní selektory, či jejich atributy. Otázkou ale zůstává, zdali má taková schopnost v praxi vůbec nějaké smysluplné využití a zdali ji tedy uživatel potřebuje.

Další funkci, známou z nativního CSS, kterou tento plugin převzal, jsou výchozí hodnoty. Jedná se o jakousi záložní hodnotu, k jejímuž použití dojde ve chvíli, kdy se kompilátoru nepovedlo nalézt požadovanou proměnnou. K použití této funkce dojde jednoduše přidáním výchozí hodnoty za název požadované proměnné. Výchozí hodnotu je od názvu proměnné nutné oddělit čárkou.

Ukázka použití výchozí hodnoty:

```
/* Přistupování k proměnné, která neexistuje */
.trida {
  color: var(--promenna, red);
}

/* Zpracované CSS */
.trida {
  color: red;
  color: var(--promenna, red);
}
```

Kompilátor nenalezl proměnnou `–promenna`, na místo této nenalezené proměnné je tak dosazena hodnota „red“.

Ke scopingu pomocí Custom properties je třeba, kvůli specifičnosti tohoto nástroje, přistupovat s rezervou. Jelikož k překládání proměnných dochází jen v případě jejich deklarace v pseudotřídě `:root`, v podstatě existuje jen globální rozsah. Pokud pak není proměnná deklarována uvnitř této třídy a přes to dochází k použití funkce `var()` s jejím názvem, zachová se nástroj stejně, jako kdyby proměnná vůbec neexistovala, finální kód obsahuje volání funkce tak, jak jej zapsal uživatel - `var(--promenna);` Nicméně i přesto zde dochází k určitému vymezení rozsahu, tento fakt je ale zapříčiněn tím, že s touto syntaxí dokáže pracovat samotné CSS, proto je nutné nastavit hranici mezi těmito dvěma nástroji a je třeba říci, že samotné Custom properties nabízí jen globální rozsah.

Datové typy:

- Čísla – 1.5, 10, 3px
- Textové řetězce – s uvozovkami i bez
- Barvy – blue, #b30000, rgba(255, 0, 0, 0.5)
- Seznam hodnot, který se dá oddělit čárkou - "Times New Roman", Times, serif;
- Odkazy na další proměnné: --druhy: var(--prvni);

Nativní CSS navíc podporuje možnost předávat v rámci funkce `var()` i složené výrazy pomocí funkce `calc()`. Je tak možné například násobit proměnné, či sečíst zadané konstanty. I s tímto případem si tento plugin dovede poradit. Opět při standartním nastavení vytvoří nejdříve řádek ve kterém je na místo proměnné dosazena její hodnota. Tento řádek je následován dalším, který obsahuje celý složený výraz tak, jak jej zadal uživatel. Je k němu tedy přistupováno jako k textovému řetězci. Pokud uživatel nepoužívá plugin pro zpracování matematických funkcí, je zodpovědnost předána prohlížeči a ten přistupuje k funkci `calc()` standartním způsobem. Není-li tento složený výraz obalen funkcí `calc()` zahlásí kompilátor chybovou hlášku a ke kompilaci nedojde.

5.2.2.1.5 Custom properties - volby

Preserve - Pokud se tento atribut nastaví na hodnotu false, výsledný CSS soubor bude vygenerovaný bez pseudotřídy `:root`, deklarací a volání proměnných.

ImportFrom - Tento atribut umožňuje do zpracovávaného CSS souboru importovat externí proměnné a to nejen v podobě dalšího CSS, ale i v souborech typu JSON, JS, či v podobě objektu.

ExportTo - Pomocí tohoto atributu je možné naopak exportovat proměnné, které jsou zpracovávány v uživatelově kódu, do externího souboru a to opět pomocí CSS, JSON, JS či jako objektu.

Všechny tyto volby byly otestovány na základě dokumentace, která je k nalezení na webu Custom properties [28].

5.2.2.1.6 Custom properties - stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [28] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

CUSTOM PROPERTIES	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	30.3.2019
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	347
NPM STAŽENÍ TÝDNĚ	1 376 748

Tabulka 6: Popularita a stav vývoje nástroje Custom properties

5.2.2.2 **Stylecow**

V tomto postprocesoru je s proměnnými možné pracovat pomocí nástroje zvaného prostě a jednoduše Variables, tedy proměnné.

5.2.2.2.1 Variables - řešení

Stejně, jako PostCSS nástroj Custom Properties, i tento vychází ze zápisu proměnných tak, jak se začínají v poslední době prosazovat v nativním CSS. Stejně, jako v předchozím případě se zde objevuje pseudotřída `:root`, nicméně zápis do ní je nutný jen v případě, kdy uživatel vyžaduje globální rozsah. Každá jednotlivá proměnná v této pseudotřídě je definována pomocí dvojité pomlčky a stejně tak je k jejímu obsahu přistupováno pomocí funkce `var()` a následné uvedení požadované proměnné do kulatých závorek i se dvěma pomlčkami za slovem var.

Ukázka zpracování kódu pomocí Variables:

```
/* Deklarace proměnné */
:root {
 --color: blue;}

/* Přistupování k hodnotě proměnné */
h1 {
  color: var(--color);}

/* Zpracované CSS */
h1 {
  color: blue;
}
```

Z ukázky vyplývá, že výstupní vygenerovaný CSS kód je v případě tohoto nástroje mnohem kratší nežli v případě Custom Variables. Neobjevuje se zde pseudotřída `:root`, deklarování proměnných a ani jejich volání, stejně jako tomu je, pokud je v předchozím nástroji zakázána možnost Preserve.

To jinými slovy znamená, že chování proměnných v tomto případě probíhá tak, jak je tomu u většiny preprocesorů. Na druhou stranu v tomto případě není možné využívat standardní funkcionality, kterou nabízí samotné CSS proměnné.

Je-li proměnná v případě tohoto nástroje deklarována mimo `:root`, je možné si všimnout, že na rozdíl od předchozího nástroje, dochází k jejímu přeložení. Proměnná nicméně musí být deklarována uvnitř deklaračního bloku.

Jak bylo poznamenáno výše, Variables umožňují překlad i proměnných deklarovaných mimo pseudotřídu `:root`, proto je možné pozorovat u nich rozsah a to nezávisle na běžné CSS syntaxi proměnných. Proměnné, které jsou deklarovány v této pseudotřídě mají globální rozsah, zatímco proměnné deklarované v nekořenovém bloku vlastností mají rozsah pouze lokální. Pokud je navíc k lokální proměnné přistupováno z jiného bloku, nedojde k jejímu přeložení, ale ani k nahlášení chyby. Je-li globální proměnná přepsána lokálně, ve výsledném kódu se objeví nejbližší výše deklarovaná proměnná v tomto bloku. Pokud k deklaraci proměnné nedojde, nebo k ní dochází až po pokusu o přístupu k její hodnotě, nedojde k přeložení volání funkce `var()` a ani k zahlášení chyby.

Co se týče zanořených pravidel, tedy při zanořování pomocí nesting pluginu, který funguje stejně jako tomu je u většiny preprocesorů, je proměnná viditelná i ve všech potomcích tohoto pravidla a pokud je v průběhu změněna, dochází k použití nejbližší nadřazené deklarace v tomto bloku.

Stejně, jako předchozí plugin, ani tento neumožňuje jakoukoliv interpolaci, pokud se navíc uživatel o něco podobného pokouší, příkazová řádka nahlásí chybu odkazující uživatele na část kódu, ve které se vyskytla chyba, a ke kompilaci ani nedojde.

Tak jako je tomu u předchozího pluginu, i zde je možné využívat výchozích hodnot. Tato funkce zde funguje totožným způsobem jako v případě nativního CSS.

Datové typy:

- Čísla – 1.5, 10, 3px
- Textové řetězce – s uvozovkami i bez
- Barvy – blue, #b30000, rgba(255, 0, 0, 0.5)
- Seznam hodnot, který se dá oddělit čárkou - "Times New Roman", Times, serif;
- Odkazy na další proměnné: --druhy: var(--prvni);

I v tomto pluginu je možné nalézt částečnou podporu složených výrazů. Stejně jako předchozí nástroj i tento přistupuje ke složenému výrazu jako k textovému řetězci. Složený výraz nicméně nemusí být obalený do funkce `calc()`. Pokud se ve složeném výrazu objevuje proměnná, je na její místo dosazena její hodnota, k ostatním hodnotám je přistupováno jako k běžnému textovému řetězci. Překlad proměnné se však podobně jako při standartním použití tohoto nástroje nedá zvrátit a síla nativního CSS je tak ztracena.

5.2.2.2.2 Variables - stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [24] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

VARIABLES	**HODNOTY**
GITHUB POSLEDNÍ AKTULIZACE	20.8.2017
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	0
NPM STAŽENÍ TÝDNĚ	10

Tabulka 7: Popularita a stav vývoje nástroje Variables

5.2.2.3 **Závěr**

Jednoznačně nejjednodušší a nejintuitivnější syntaxi nabízí Simple Variables pro PostCSS a získává tak **2 body**, zbylé dva nástroje se ve svém přístupu neliší a náleží jím po **1 bodu.**

Interpolaci nabízí jen Simple Variables – **1 bod.**

Jelikož Stylecow umožňuje využívat i jiného, nežli globálního rozsahu, nezbývá než připsat **2 body** právě jemu. Zbylé nástroje si pak odnáší po **1 bodu**, protože přináší jen globální rozsah.

Výchozí hodnoty přináší jen Custom Properties a proměnné v pluginu pro Stylecow. Tyto dva nástroje tak získávají po **1 bodu**.

Všechny zmíněné nástroje nabízí obdobný přístup k datovým typům, proto získávají všechny po **1 bodu**.

V neposlední řadě jen Custom Properties umožňují využívat i schopností nativního CSS a tak získává tento nástroj jako jediný **bod** navíc.

I když nabízí Simple Variables a Custom Properties zajímavé volby nastavení, nedají se považovat za jednoznačný přínos k řešení proměnných, a proto za tuto funkcionalitu nezískávají body navíc.

Nejpopulárnějším nástrojem s nejstabilnějším vývojem jsou pak Custom Properties a tak získávají další **bod** k dobru.

PROMĚNNÉ	**POSTCSS – SIMPLE VARIABLES**	**POSTCSS – CUSTOM PROPERITES**	**STYLECOW**
NABÍZÍ	3	3	3
SYNTAXE	2	1	1
INTERPOLACE	1	0	0
ROZSAH	1	1	2
VÝCHOZÍ HODNOTY	0	1	1
DATOVÉ TYPY	1	1	1
NATIVNÍ CSS	0	1	0
POPULARITA	0	1	0
CELKEM:	8	9	8

Tabulka 8: Vyhodnocení nástrojů pro proměnné

Vyšší počet bodů – Custom Properties.

5.2.3 Extend

Postprocesory nabízí v rámci rozšíření obdobné možnosti jako je tomu v případě preprocesorů. Během následujícího srovnání bude brán zřetel zejména na syntaxi, možnosti používat tiché třídy, způsob, jakým je přistupováno k pořadí selektorů,

rozšiřování napříč media queries, přístupu k možným duplicitám, či řetězení extendů.

5.2.3.1 PostCSS

V nástroji PostCSS je možné setkat se s více nástroji, které umožňují práci s extendy, v následující části proto bude popsán ten nejpopulárnější z nich.

5.2.3.1.1 Postcss-extend - řešení

Obdobně, jako je tomu u spousty dalších nástrojů PostCSS, i zde je možné vypozorovat inspiraci, ať už co se syntaxe týče, tak nejrůznějšími postupy zpracování kódu, preprocesory. Lze si tedy všimnou, že k samotnému zavolání této funkce dochází pomocí direktivy `@extend`, po které ihned následuje selektor pravidla, jehož vlastnosti dědí pravidlo, ve kterém je tato funkce volána.

Na následujícím příkladu je prakticky ukázána výše popsaná funkcionalita:

```
/* vytvoření pravidla */
.trida{
color:red;}

/* pravidlo, které rozšiřuje předchozí pravidlo */
.rozsirujiciTrida {
  @extend .trida;
  border-radius: 5px;}

/* vygenerované CSS */
.trida, .rozsirujiciTrida{
color:red;
width:200px;}

.rozsirujiciTrida {
  border-radius: 5px;}
```

Z ukázky je možné si všimnout, že došlo ke zdědění vlastností a hodnot, které se vyskytují ve třídě `.trida`, zatímco třída `.rozsirujiciTrida` k těmto vlastnostem přidá i své vlastnosti.

Nejinak je tomu u takzvaných „placeholders“ neboli tichých tříd. Jedná se o třídy, ze kterých může být děděno, nicméně se neobjeví ve finálním CSS.

Stejně jako v jazyce Sass tak i zde se takové třídy deklarují pomocí znaku „%“. Na rozdíl od Sassu je ale možné obdobné funkcionality dosáhnout pomocí výrazu `@define-placeholder`, případně `@define-extend` nebo `@extend-define`. Na

rozdíl od selektorů definovaných pomocí „%“ není například možné tyto selektory definovat v blocích pro media queries, případně je není možné volat dříve, nežli jsou v kódu definovány.

Dále je vhodné poznamenat, že k rozšiřování nemusí docházet jen v případě tříd, ale i dalších selektorů, jako jsou identifikátory, či typové selektory.

Dalším zajímavým rozdílem je fakt, že na rozdíl od Sassu není možné rozšiřovat prvky, které jsou na jiné než rodičovské pozici.

Obrovskou výhodou oproti již zmíněnému Sassu je možnost využívat funkci extend napříč bloky například media queries. To znamená, že pokud jsou vlastnosti pro požadovaný selektor volány kupříkladu jen tehdy, kdy je zobrazován na obrazovce s šířkou pod 600px, může sdílet vlastnosti i se selektorem mimo tento blok. Tam, kde Sass zahlásí chybu, přichází postcss-extend s důmyslným řešením. Přirozeně není možné přidat název pravidla k rozšiřovanému pravidlu, jak je tomu u běžného rozšiřování. Tím by funkce media query pozbyla významu. PostCSS Jednoduše zkopíruje ty vlastnosti, které obsahuje rozšiřované pravidlo mimo tento blok. Pokud narazí na vlastnosti, které již obsahuje původní pravidlo, ale v rozšiřovaném pravidlu existuje ta samá vlastnost ale s jinou hodnotu, původní pravidlo si zachová původní hodnotu takové vlastnosti.

Nastíněnou schopnost je možné pozorovat na následujícím příkladu:

```
/* Pravidlo mimo media query */
.okno {
color: white;
background-color: black;}

/* Media query */
@media (width > 800px) {
  .okno:hover {
    color:blue;}

  .dvere {
    @extend .okno;
    color: red;
    font-size: 4em;}

/* Zpracované CSS */
.okno {
color: white;
background-color: black;}

@media (width > 800px) {
  .okno:hover, .dvere:hover {
    color:blue;}

  .dvere {
    color: red;
    font-size: 4em;
    background-color: black;}
}
```

Jak je možné si všimnout – v případě rozšiřování mezi více bloky nedojde k hromadné deklaraci pomocí více selektorů, jako tomu bylo v předchozím případě, ale k rozšíření vlastností v požadované třídě.

Pokud je naopak rozšiřováno pravidlo nacházející se uvnitř media query pravidlem vně takového bloku, je zachována původní funkcionalita extend. To znamená, že k selektoru rozšiřovaného pravidla je přidán selektor pravidla, který toto pravidlo rozšiřuje. Nově získané vlastnosti se tak projeví jen v případě, kdy je splněna podmínka media query.

Další funkcí, kterou se tento nástroj nechal inspirovat Sassem je pak takzvané řetězení extendů. To jinými slovy znamená, že je možné rozšiřovat třídu, která sama rozšiřuje jinou třídu. V takovém případě pak může nastat zacyklení extendů, kdy rozšiřovaná třída rozšiřuje třídu, kterou je sama rozšiřovaná. V takovém případě je uživatel v příkazové řádce upozorněn na tuto chybu. PostCSS se jej však pokusí i

přes to zpracovat a ve valné většině případů také úspěšně. V opačném případě je uživatel upozorněn na nemožnost splnění tohoto úkonu a ke kompilaci nedochází. PostCSS se navíc dokáže při tomto použití zbavit všech duplicit.

5.2.3.1.2 Postcss-extend - stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [29] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

POSTCSS-EXTEND	**HODNOTY**
GITHUB POSLEDNÍ AKTULIZACE	17.10.2017
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	3
GITHUB HVĚZDIČKY	95
NPM STAŽENÍ TÝDNĚ	85 784

Tabulka 9: Popularita a stav vývoje nástroje Postcss-extend

5.2.3.2 **Stylecow**

5.2.3.2.1 Extend - řešení

Stejně, jako je tomu u pluginu pro PostCSS, tak i v tomto případě je možné vypozorovat syntaxi, která se nápadně podobá té v případě Sassu, nicméně na webu tohoto projektu se lze dočíst, že syntaxe ve skutečnosti vychází z návrhu Taba Atkinse [24].

Není tedy překvapením, že podobně, jako je tomu v předchozím nástroji, dochází k volání této funkce pomocí direktivy `@extend`, za kterou následuje název selektoru, který by měl být rozšířen. Ovšem pokud se jedná o samotné názvy selektorů a jejich vytváření, přichází tento nástroj s vlastním řešením. Analogicky jako je tomu v nástroji postcss-extend, i zde se používá znaménka procenta „%“, nicméně,

jakýkoliv selektor, u kterého by mělo dojít k rozšíření, je nutné deklarovat pomocí tohoto znaménka. To znamená, že všechny selektory k rozšíření zde fungují jako tiché třídy. Pokud dojde k pokusu o rozšíření existujícího selektoru bez znaku „%“, kód zůstane ve své původní nezpracované podobě. Tato skutečnost je tedy v souladu s doporučením používání vlastnosti extend, takže zabraňuje mnohým chybám, nicméně odpírá uživateli možnost rozhodnout se, jelikož ji není možné ani deaktivovat.

Jak vyplývá z textu výše, tak na rozdíl od Postcss-extend není možné pomocí tohoto nástroje rozšiřovat jiné selektory nežli tiché třídy.

Oproti předchozímu nástroji se tento nástroj svým chováním přibližuje v případě rozšiřování selektorů, které se nachází i na jiné než na rodičovské pozici, spíše svým preprocesorovým kolegům. To znamená, že je možné rozšířit například tichou třídu `%radek`, která se ale nachází v třídě `.odstavec` zapsané tedy takto

```
.odstavec %radek {
  font-weight: bold;}
```

Následné volání by pak probíhalo podobně jako v případě předchozího pluginu.

Oproti předchozímu nástroji tento nástroj neumožňuje rozšiřovat selektory napříč bloky. Pokud je tedy tichá třída deklarována mimo blok media query a je volána zevnitř tohoto bloku, tváří se kód, že došlo k jeho zpracování, protože volání `@extend` je odstraněno, ale v jiných aspektech se změna neprojeví.

Pokud dojde k mnohonásobnému volání extendu, se kterým se může uživatel setkat například při řetězení extendů – tedy pokusu o rozšíření třídy, která sama rozšiřuje jinou třídu, nedokáže se tento nástroj na rozdíl od předchozího vypořádat s duplicitami.

Příklad tohoto zpracování:

```
/* Řetězení rozšíření */
%h1 {
  color:blue;}

%h2 {
  @extend %h1;
  color:red;}

%h5 {
    @extend %h2;
    @extend %h1;}

.h5 {
   @extend %h5;}

/* Zpracované CSS */
.h5,.h5 {
  color: blue;}

.h5 {
  color: red;}
```

Jak je z ukázky vidět, na prvním řádku zpracovaného CSS se při hromadné deklaraci z nepochopitelných důvodů objevuje dvakrát jedna a ta samá třída. Takový zápis by na finální chování samozřejmě neměl žádný vliv, nicméně se jedná o zarážející chybu.

Pokud pak dojde k rozšiřování třídy třídou, která již rozšiřuje tuto třídu – zjednodušeně k zacyklení, zahlásí Stylecow chybu ohledně přetečení paměti a ke kompilaci nedojde.

5.2.3.2.2 Extend - Stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [24] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

EXTEND	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	18.9.2015
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	0
NPM STAŽENÍ TÝDNĚ	5

Tabulka 10: Popularita a stav vývoje nástroje Extend

5.2.3.3 **Závěr**

Oba nástroje nabízejí své řešení extendů, získávají tak po **3 bodech.**

Syntaxe se liší a to zejména ve faktu, že nástroj Stylecow neumožňuje rozšiřovat jiné než tiché třídy, zatímco plugin pro PostCSS ano a navíc pro tiché třídy přináší i jiné syntaxe, PostCSS tak získává **2 body** za syntaxi a Stylecow jen **1**.

Oba nástroje umožňují využívat tichých tříd – každý získává **1 bod.**

Možnost rozšiřovat selektory i na jiné, než rodičovské pozici přináší jen Stylecow a tak získává **1 bod.**

Rozšiřování napříč media queries umožňuje jen PostCSS, získává tak **1 bod.**

Duplicity se mohou objevit jen v případě nástroje Stylecow, a proto získává PostCSS **1 bod** za jejich odstraňování.

Řetězení umožňují oba nástroje, nicméně jen jeden z nich si dokáže poradit i s jejich zacyklením a tím je nástroj pro PostCSS, získává tak **2 body** a Stylecow jen **1**.

Tím populárnějším nástrojem z této dvojice, který zároveň nabízí stabilnější vývoj je plugin pro PostCSS, a proto získává **1 bod** navíc.

EXTEND	**POSTCSS**	**STYLECOW**
NABÍZÍ	3	3
SYNTAXE	2	1
TICHÉ TŘÍDY	1	1
NERODIČOVSKÝ SELEKTOR	0	1
MEDIA QUERIES	1	0
ODSTRANĚNÍ DUPLICIT	1	0
ŘETĚZENÍ	2	1
POPULARITA	1	0
CELKEM:	11	7

Tabulka 11: Vyhodnocení nástrojů pro rozšíření

Vyšší počet bodů: PostCSS - postcss-extend

5.2.4 Nesting

Hnízdění, či anglicky nesting, je obdobně jako v preprocesorech způsob, jak zpřehlednit vytváření vnořených pravidel.

Během tohoto srovnání bude brána v potaz syntaxe, uspořádání výsledného kódu, schopnost odkazovat na rodiče, či možnost využít tento nástroj v rámci některé z metodik.

5.2.4.1 **PostCSS**

Stejně, jako je tomu u jiných pluginů, i v případě nestingu existuje v PostCSS celá řada různých nástrojů, které slouží k zavedení této funkcionality. V rámci zjednodušení srovnání bude popsán nejpoužívanější z těchto nástrojů.

5.2.4.1.1 Postcss-Nested - řešení

Na první pohled se tento nástroj syntaxí nijak neliší od preprocesorů. To znamená, že k zahnízdění stačí vložit do deklaračního bloku jednoho pravidla pravidlo jiné, jenž má zaujmout místo potomka.

Ukázka:

```
/* Zahnízdění */
.sloupec {
        height: 500px;

    .radek {
        width: 200px;
    }}

/* Zpracované CSS */
.sloupec {
       height: 500px;
}
.sloupec .radek {
        width: 200px;
}
```

Jak z ukázky vyplývá, používání této funkcionality je velmi intuitivní a jednoduché.

Další důležitou schopností, na kterou jsou zvyklí všichni uživatelé preprocesorů, je takzvaný odkaz na rodiče. Ten je užitečný zejména všude tam, kde je využíváno názvu rodičovského selektoru, který je následně doplněn o další upřesňující informaci – například pseudotřídu, či jako upřesňující vlastnosti určitého selektoru. Funguje tedy jako vhodný nástroj pro aplikaci některé z metodik CSS.

Pro odkaz na rodiče je zde použito stejně, jako v případě většiny nejpoužívanějších preprocesorů znaménka ampersand – „&“. Tento znak rozhoduje o tom, na jaké místo v selektoru bude umístěn rodičovský selektor.

Stejně jako u Sassu nedochází na odkazování pouze na prvního rodiče nacházejícího se nad potomkem, nýbrž na všechny jeho předchůdce.

Pro pochopení krátká ukázka:

```
/* Odkaz na rodiče */
.okno {
    .nadpis {
        width: 300px;
         &:hover {
            color: red;
        }
       .radek & {
             font-size: 12px;
        }
    }
    img {
        display: block;
    }
}

/* Zpracované CSS */
.okno .nadpis {
        width: 300px;}

.okno .nadpis:hover {
      color: red;}

.radek .okno .nadpis {
      font-size: 12px;}

.okno img {
     display: block;}
```

Jak vyplývá z ukázky, i když je například třída `.radek` na první pohled potomkem tříd `.okno` a `.nadpis`, použitím ampersand se stává jejich rodičem. Zatímco z pseudotřídy `:hover`, která sama o sobě nic neznamená, se díky ampersandu stane pseudotřída rozšiřující třídu `.nadpis`.

Pokud pak dojde k použití více „&“ za sebou, dojde k přidání dalších potomků, či předků. Za každý nový ampersand přibydou znovu všechny s ním související selektory. Nezbývá nežli přemýšlet nad významem této funkcionality.

Možnost odkazovat na konkrétní selektor a ne tedy na všechny, které předcházejí vybranému selektoru, tak jak je tomu u odkazu na rodiče, nabízí plugin zvaný „PostCSS Nested Ancestors“ [30].

Vyžaduje-li uživatel funkci vnořených vlastností, která se objevuje například v Sassu a využívá se u vlastností, které se nachází ve stejném jmenném prostoru, kterým je například `font`, pro vlastnosti jako `fonta-family`, či `font-size`, je nezbytně nutné nainstalovat plugin zvaný „PostCSS Nested Props“ [29].

5.2.4.1.2 PostCSS-Nested - volby

Tento nástroj také nabízí několik voleb pro usnadnění, či vylepšení pracovního procesu.

bubble - defaultním chováním tohoto nástroje při nalezení direktivy `@media` či `@support` je takzvané „vybublání“.

Vybublání je stav, kdy se tato direktiv dostane do nejvyšší úrovně CSS, i když se před tím mohla nacházet uvnitř selektoru a stane se de facto rodičem všech jejich do té doby rodičovských selektorů. Tato volba umožňuje nastavit vlastní takové direktivy.

unwrap - obdobně jako bubble, když unwrap narazí na jistou direktivu, dostane tuto direktivu na nejvyšší úroveň, zcela mimo všechny selektory. V tomto případě se ale z této direktivy stane samostatné pravidlo. I tato volba umožňuje vytvořit vlastní seznam direktiv.

preserveEmpty - pokud během zpracování kódu dojde k vytvoření pravidla, které neobsahuje žádné vlastnosti, je takové pravidlo smazáno, tato volba umožňuje taková pravidla zachovat.

Všechny tyto volby byly otestovány na základě dokumentace, která je k nalezení na webu PostCSS Nested [30].

5.2.4.1.3 PostCSS-Nested - stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [30] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

POSTCSS-NESTED	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	25.3.2019
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	540
NPM STAŽENÍ TÝDNĚ	323 934

Tabulka 12: Popularita a stav vývoje nástroje PostCSS-Nested

5.2.4.2 **Stylecow**

5.2.4.2.1 Nested-Rules - řešení

Stejně, jako v předchozím případě, i zde je syntaxe nerozpoznatelná od té, která se nachází mezi představenými preprocesory. Intuitivně tedy stačí vložit jedno pravidlo do bloku jiného pravidla a stává se jeho potomkem. Takový kód je potom zpracovaný téměř analogicky jako v předchozím případě. Nicméně pokud je výčet selektorů pravidla delší, jsou tyto selektory ve výsledném souboru rozděleny na řádky a zkompilovaný kód je tak přehlednější, jedná se o schopnost, kterou předchozí nástroj nenabízí.

Zpracování pravidla se složitějším selektorem může vypadat například následovně:

```
/* Zahnízdění */
.trida1,
.trida2,
.trida3 {
  width: 300px;
  .trida4 {
    font-size: 12px;
  }
}

/* Zpracované CSS */
.trida1,
.trida2,
.trida3 {
    width: 300px;
}
.trida1 .radek,
.trida2 .radek,
.trida3 .radek {
    font-size: 12px;
}
```

Odkazování na rodiče probíhá, stejně jako v předchozím případě, pomocí ampersandu, nicméně jeho chování se znatelně liší. Na rozdíl od předchozího případu neurčuje ampersand jen to, jestli se selektor stane předkem, nebo potomkem všech selektorů, ale pomocí jeho opakovaného použití je možné určit i pořadí vybraného selektoru v této struktuře. Nachází-li se kupříkladu selektor v nejnižší pozici struktury a následně je za něj přidán „&“, posouvá se tento selektor o jednu úroveň výš v této struktuře a jeho přímý předek se stává jeho potomkem. Analogicky lze pokračovat až k nejvyšší úrovni. Pokud počet ampersandů překročí počet úrovní, nedochází již k jejich zpracování a zůstávají tak součástí vygenerovaného kódu.

Při odkazování na rodiče se však objevuje zarážející chyba. Pokud dojde k odkazování na pravidlo výše, a to za pomocí selektoru, který nezapadá do běžné CSS selektorové syntaxe, tedy pokud se nejedná o třídu, identifikátor, běžný element, či pseudotřídu, Stylecow si nedokáže s takovým případem poradit. Nehledě na postavení ampersandu vůči takovému selektoru, název selektoru se vždy objevuje na první pozici výsledného pravidla. Takový přístup pak zcela odrazuje od

používání nejen metodik, ale jakýchkoliv prostých kombinací názvů, které patří mezi základní stavební kameny funkce odkazování na rodiče.

Zde následuje ukázka takového problému:

```
/* Odkazování na rodiče s nestandartním selektorem */
.trida {
  background: blue;
  &_radek {
    background: green;
  }
}

/* Očekávané zpracování */
.trida {
  background: blue;
}
.trida_radek {
  background: green;
}

/* Reálný výsledek zpracování */
.trida {
    background: blue;
}
_radek.trida {
    background: green;
}
```

Obdobně jako v předchozím případě, ani zde se nenachází odkaz na selektor, tuto funkcionalitu však není ani možné získat pomocí jiného pluginu.

Stejně tak není možné využívat tento nástroj k hnízdění vlastností, které se nacházejí ve stejném jmenném prostoru. Stylecow pro tuto funkcionalitu nenabízí ani jinou alternativu.

5.2.4.2.2 Nested-Rules – Stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [24] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

NESTED-RULES	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	9.10.2015
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	0
NPM STAŽENÍ TÝDNĚ	10

Tabulka 13: Popularita a stav vývoje nástroje Nested-Rules

5.2.4.3 **Závěr**

Protože oba nástroje nabízejí řešení funkce hnízdění, získávají oba po **3 bodech.**

Syntaxe obou nástrojů se nijak neliší od nativního CSS a získávají tak oba po **1 bodu**.

Kód vygenerovaný Stylecow je výrazně přehlednější než v případě PostCSS, a proto získává Stylecow **2 body** a PostCSS jen **1**.

Oba nástroje umožňují odkazovat na rodiče, Stylecow v tomto případě však přichází se zajímavým řešením určení posloupnosti selektorů, získává tak **2 body** a PostCSS jen **1.**

PostCSS umožňuje používat při odkazu na rodiče jiné než standardní názvy selektorů a umožňuje tak rozvíjet metodiky jako je BEM, či OOCSS, a proto získává **1 bod**.

PostCSS navíc sice přináší volby, které umožňují schopnost zahnízdění dále rozvíjet, nejsou ale buď přímo součástí tohoto nástroje, nebo je využití jejích praktického nasazení při nejmenším diskutabilní a nezískávají za ně žádný bodový zisk navíc.

V popularitě a vývoji pak jasně vítězí PostCSS a získává tak další **bod** k dobru.

HNÍZDĚNÍ	**POSTCSS**	**STYLECOW**
NABÍZÍ	3	3
SYNTAXE	1	1
PŘEHLEDNOST	1	2
ODKAZ NA RODIČE	1	2
NESTANDARDNÍ SELEKTOR	1	0
POPULARITA	1	0
CELKEM:	8	8

Tabulka 14: Vyhodnocení nástrojů pro hnízdění

Vyšší počet bodů: Nerozhodně.

Jelikož však PostCSS získává srovnávací bod za popularitu, kterou lze jen stěží považovat za směrodatnou při hodnocení nástroje, může být v této kategorii Stylecow považováno za pomyslného vítěze.

5.2.5 Mixin

Stejně, jako je tomu v případě dalších populárních funkcí preprocesorů i v případě mixinů je možné setkat se v portfoliu postprocesorů.

Jelikož Stylecow řešení mixinů zcela postrádá, dá se následující kapitola považovat spíše za popsání schopností přístupu PostCSS k řešení této problematiky.

5.2.5.1 **PostCSS**

PostCSS v mnohých případech nabízí několik přístupů ke každé ze svých funkcí, nejinak je tomu v případě mixinů. V následující části proto bude kladen důraz na představení a popsání nejpopulárnějšího z těchto nástrojů.

5.2.5.1.1 PostCSS Mixins – řešení

Zejména v případě tohoto nástroje je třeba nutné brát ohledy na pořadí referencí v iniciačním souboru. Například v kombinaci s pluginy pro hnízdění a proměnné, je třeba umístit referenci na tento nástroj před reference na zbylé dva nástroje. Bez dodržení tohoto přístupu není možné využívat jejich plného potenciálu a v krajních případech nemusí dojít ani ke správné kompilaci takového kódu.

V případě syntaxe je opět zřejmá inspirace preprocesorem Sass. I zde se při deklaraci využívá znaku „@“, tím však podobnost v zásadě končí. Při deklaraci se pak používá direktivy `@define-mixin` a za ní je uveden název mixinu. K volání dochází pomocí direktivy `@mixin`.

Krátká ukázka zpracování mixinu:

```
/* Definice mixinu */
@define-mixin mixin {
  background-color: blue;
  width: 100px;
}

/* Volání mixinu */
.blok {
  color: white;
  @mixin mixin;
}

/* Zpracované CSS */
.blok {
  color: white;
  background-color: blue;
  width: 100px;
}
```

Jak je z ukázky patrné, základní použití mixinů se krom drobných rozdílů v syntaxi od použití v preprocesorech příliš neliší.

Zde však schopnosti tohoto nástroje zdaleka nekončí. Dnes již samozřejmou vlastností mixinů jsou takzvané parametry a nejinak je tomu v případě tohoto nástroje. Parametr stačí umístit za název mixinu. Tento parametr je pak od názvu odlišen pomocí znaku „`$`“. K volání parametrického mixinu následně dochází obdobně jako v předchozím případě, na místo parametru je však dosazena požadovaná hodnota a to bez jakékoliv přidané syntaxe. Parametrické proměnné je pak navíc možné vhodně kombinovat například se zahnízděním, či dalšími pluginy. Parametrům je možné přiřadit i defaultní hodnoty. Jejich použití je vhodné zejména v případech, kdy uživatel používá jeden mixin na více místech kódu, přičemž mění hodnoty jen v některých výskytech. Defaultní hodnota se vyjádří obdobně jak je tomu při deklaraci proměnných. Za název parametru je umístěna požadovaná hodnota, které předchází dvojtečka.

Ukázka výše popsaných vlastností:

```
/* Definice parametrického mixinu */
@define-mixin blok $barva-nazev, $color: blue {
  .blok-is-$(barva-nazev) {
    color: $color;
  }
  .blok-is-$(barva-nazev):hover {
    color: white;
    background: $color;
  }
}

/* Volání mixinu */
@mixin blok modry {
}
@mixin blok cerveny, red {
}

/* Zpracované CSS */
.blok-is-modry {
    color: blue;
  }
.blok-is-modry:hover {
    color: white;
    background: blue;
  }
.blok-is-cerveny {
    color: red;
  }
.blok-is-cerveny:hover {
    color: white;
    background: red;
  }
```

Jak vyplývá z ukázky, parametry je možné využít nejen na pozici hodnoty vlastností, ale také v rámci interpolace názvů. Takový přístup je zejména užitečný při využívání některé z metodik.

I v případě mixinů je možné mluvit o vymezení rozsahu, tak jak je známé z proměnných. Co se nástroje PostCSS Mixins týče, v podstatě každý deklarovaný mixin, nehledě na jeho umístění, má globální rozsah. To znamená, že mixin, který je definovaný v rámci určitého bloku, může být přístupný v rámci ostatních bloků. Pokud je pak v souboru deklarováno na více místech kódu několik mixinů se stejným jménem a takový mixin je následně volán, dosadí se na jeho pozici hodnoty z první v kódu výše umístěné definice takového mixinu a to bez ohledu na to, v jaké úrovni zanoření se vyskytuje. Pokud definice volaného mixinu neexistuje, případně je zapsána až po jeho volání, dochází k vypsání chybové hlášky a kompilace není uskutečněna.

PostCSS navíc nabízí takzvanou `@mixin-content` funkci. Tato funkce umožňuje do mixinu vložit celý blok vlastností a následně jej umístit na libovolné místo, a to za pomocí pravidel pro zanořování.

5.2.5.1.2 PostCSS Mixins – Stav vývoje a popularita

Stav vývoje na základě repositáře GitHub [31] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

POSTCSS MIXINS	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	15.12.2018
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	312
NPM STAŽENÍ TÝDNĚ	134 270

Tabulka 15: Popularita a stav vývoje nástroje PostCSS Mixins

5.2.5.2 Stylecow

Tento postprocesor nenabízí žádné řešení mixinů.

5.2.5.3 Závěr

Jelikož řešení mixinů přináší jen PostCSS, odnáší si zisk **3 bodů.**

Protože druhý nástroj řešení mixinů zcela postrádá a neexistuje tak možnost srovnání, nebude brán v tomto případě zřetel na další kritéria hodnocení.

PREFIX	**POSTCSS**	**STYLECOW**
NABÍZÍ	3	0
CELKEM:	3	0

Tabulka 16: Vyhodnocení nástrojů pro mixiny

Vyšší bodový zisk: PostCSS – PostCSS Mixins

5.2.6 Popularita postprocesorů

Popularita nemůže být v žádném případě považována za směrodatnou hodnotu v rámci srovnávání kvalit a to zejména v případě dvou technologií. Nicméně se i tak jedná o zajímavý ukazatel a to především proto, že populárnějším nástrojům se ve většině případů dostává lepší péče, co se na poli vývoje týče. Na základě popularity pak může být také odhadnuto, jakými dalšími cestami se může daný nástroj vydávat.

Pro srovnání budou brány v potaz opět informace vycházející z verzovacího nástroje GitHub a správce balíčků Node Package Manager (NPM).

5.2.6.1 **PostCSS**

Stav vývoje na základě repositáře GitHub [22] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

POSTCSS	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	25.3.2019
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	1
GITHUB HVĚZDIČKY	20 514
NPM STAŽENÍ TÝDNĚ	18 962 515

Tabulka 17: Popularita a stav vývoje PostCSS

5.2.6.2 **Stylecow**

Stav vývoje na základě repositáře GitHub [24] a počet stažení týdně pomocí správce balíčků NPM [26] k datu: 31.3.2019.

STYLECOW	**HODNOTY**
GITHUB POSLEDNÍ AKTUALIZACE	20.6.2017
GITHUB ZMĚNY ČEKAJÍCÍ NA ZAVEDENÍ	0
GITHUB HVĚZDIČKY	132
NPM STAŽENÍ TÝDNĚ	7

Tabulka 18: Popularita a stav vývoje Stylecow

5.2.6.3 **Shrnutí**

Čísla hovoří jasně, PostCSS svou popularitou Stylecow dalece přesahuje. Od poloviny roku 2017 nedošlo v případě tohoto nástroje k aktualizaci a jen čas ukáže, zdali se to změní. Na domovských stránkách tohoto nástroje je dokonce možné se dočíst jistých plánů na spojení s PostCSS [24]. Budoucnost Stylecow se tak nezdá příliš příznivá. PostCSS se pak dostává větší podpory i ze strany vývojových prostředí, či internetových aplikací. Vítězem této kategorie se tak stává PostCSS a odnáší si **1 bod** k dobru.

Vyšší bodový zisk: PostCSS

5.3 Závěrečné shrnutí a vyhodnocení

Předtím, než dojde k samotnému vyhodnocení, je třeba uvést několik věcí na pravou míru. Nejdříve je třeba podotknout, že nejen PostCSS, ale i Stylecow nabízí mnohem větší množství pluginů, nežli bylo popsáno. Jen samotný jejich výčet by vydal na několik stránek. Výběr tedy padl na ty nejzásadnější z nich, a to zejména na základě jejich podobnosti s preprocesory a možnosti kombinace s metodikami. Samotné množství pluginů je možné považovat za jedno z kritérií hodnocení. Ve Stylecow jich je pár desítek, v PostCSS dokonce stovky a mnoho dalších jich pravidelně vzniká. V takovém případě si samotné PostCSS může odnést pomyslné body navíc. Je však nutné podotknout, že v reálném nasazení se uživatel nesetká nikdy s větším než malým množstvím z nich.

V rámci podobných srovnání pak bývá často zvykem srovnání výkonu, v případě postprocesorů by se pak jednalo o to, kolik času stráví každý z nich zpracováním uživatelova kódu. Vhodným přístupem by při obdobném výzkumu bylo zvolit nejpodobnější pluginy a naměřené hodnoty vzájemně porovnat. Jen zřídkakdy se však v kódu vyskytují situace, kdy dochází k aplikaci jen jednoho z pluginů. Pro získání reálných výsledků je třeba srovnat výkon kombinace některých z nich. Tyto výsledky jsou často neprůkazné a liší se kombinaci od kombinace. Zásadním pro zanedbání této kategorie se však stal fakt, že při měření dochází k rozdílům maximálně v rámci několika desítek milisekund. Tento rozdíl může být výrazný například v případě dotazování databáze, kdy může probíhat stovky dotazů za

vteřinu. S obdobnou situací se nedá za standartních podmínek v případě CSS setkat. Obzvlášť v době dnešních výkonných strojů je takové srovnání zcela zbytečné.

Nezbývá, než sečíst jednotlivé složky:

SLOŽKA	**POSTCSS**	**STYLECOW**
PREFIX	10	7
PROMĚNNÉ	9	8
EXTEND	11	7
HNÍZDĚNÍ	8	8
MIXIN	3	0
POPULARITA	1	0
CELKEM	42	30

Tabulka 19: Závěrečné sečtení bodů za jednotlivé nástroje

PostCSS získává celkem **42 bodů** a Stylecow pak **30**.

Srovnání postprocesorů tak dopadlo podle očekávání. Vyšší počet bodů si odnesl ten z nich, který je dnes ve světě známý jako TEN postprocesor. I když oba tyto nástroje měly ve svých počátcích velmi totožné charakteristiky a odrazové můstky, jak nasvědčuje nejen stav popularity, ale také počty jednotlivých pluginů a frekvence jejich aktualizací, ve světě technologií se uchytil jen jeden z nich. I přes to, že Stylecow prohrál ve většině kategorií srovnání, zůstává pravdou, že se stále jedná o zajímavý nástroj, který může mnoho nabídnout. Pro případné zájemce by se především svou jednoduchostí aplikace do pracovního procesu, mohl stát zajímavou alternativou.

PostCSS pak dopadlo podle představ. Z nástroje, který původně svému autorovi sloužil jen jako jednoduchý pomocník pro řešení prefixů, se stal matador, který si v ničem nezadá s nejpopulárnějšími preprocesory. Naopak, oproti preprocesorům

nabízí mnoho schopností navíc. V kombinaci s možností vytvářet vlastní pluginy, možností upravit si pracovní proces zcela dle svých představ, či si jednoduše zvolit z velkého množství nástrojů ten, který uživateli vyhovuje, se stává PostCSS zajímavou alternativou ke všem menším, či větším preprocesorům a je třeba s ním do budoucnosti začít počítat.

6 Doporučené řešení

V předchozích částech byla představena celá řada nástrojů, které podporují efektivitu a zjednodušují vývoj webových stránek. Obdobně došlo k představení metodik, které slouží jako návod k co nejefektivnějšímu přístupu psaní jak nativního CSS a HTML, tak používání těchto podpůrných nástrojů. Mnohé z těchto přístupů je možné vzájemně kombinovat, jiné svojí filozofií stojí na opačných koncích spektra a jejich vzájemné používání je vyloučeno. Otázkou tedy zůstává, jaká kombinace těchto nástrojů vede k co nejefektivnějšímu vývoji webových prezentací. A má cenu tyto nástroje vůbec používat v době, kdy samotné nativní CSS přináší vlastní řešení proměnných, matematických operací a ve stále více případech i řešení dalších nedostatků, jejichž odstranění si dali za cíl tvůrci nástrojů představených v této práci?

6.1 Vlastní nástroj

Jedním z nejzajímavějších způsobů, jak přijít se zcela vlastním řešením je bezesporu vytvoření nového pluginu. Zmíněné procesory přináší nástroje, které vytváření pluginů velmi usnadňují.

V principu se jejich přístup zásadně neliší. Oba procesory jsou založeny na vytváření takzvaného abstraktního syntaktického stromu. Ten může být připodobněn ke struktuře DOM známé z HTML. Celý obsah stylopisu je rozdělen do jednotlivých objektů a to na základě jejich příslušnosti, například na základě toho, zda se jedná o pravidlo, či deklaraci vlastnosti. S tím, o jaký typ se jedná, jsou také spojeny funkce a atributy, ke kterým je možné přistupovat. S těmito objekty je možné pracovat jak jinak než pomocí JavaScriptu [24].

Jak už bylo nastíněno, oba nástroje přistupují k této funkci obdobným způsobem, popularita PostCSS však také přináší výhodu v podobě vyššího počtu návodů a tutoriálů napříč webem, a proto bude následující část přibližovat zejména základy tvorby pluginů pomocí PostCSS.

6.1.1 Základní vlastnosti

PostCSS rozděluje obsah stylopisu do tříd mnoha způsoby. Ty, pro začínajícího vývojáře nejzajímavější, jsou následující:

Rules – Objekty, které vychází ze samotných CSS pravidel, tedy selektoru následovaného deklaračním blokem a pravidly uvnitř. K těmto interním entitám je poté možné přistupovat.

AtRules – Jedná se o speciální výrazy, které sdělují, jak se má CSS chovat, jsou odlišeny pomocí znaku „@“, jedná se tak například o @media, či uživatelem definované výrazy [1]. PostCSS umožňuje přistupovat k pravidlům, které těmto výrazům následují.

Declaration – Objekty získané z deklarací uvnitř pravidel. Mezi jejich atributy patří například právě vlastnost a hodnota uvedené v deklaraci.

Bezesporu nejužitečnější jsou však metody, které ve spojení s těmito třídami PostCSS nabízí. Jedná se především o metody typu Walk – WalkRules, WalkAtRules, WalkDecls, které umožňují procházet polem všech objektů na základě jejich příslušnosti a následně s nimi pracovat stejným způsobem, jako s běžnými javascriptovými entitami. Například filtrace na základě deklarované vlastnosti, či hodnoty. Každá třída pak přichází s celým spektrem vlastních užitečných metod.

6.1.2 Postup vývoje

Před tím, než je možné začít na pluginu pracovat, je nutné ve složce „node_modules“, ve které jsou umístěny soubory se všemi staženými pluginy, vytvořit novou složku s názvem vznikajícího pluginu. V ní stačí vytvořit javascriptový soubor s názvem „index“ a následně je už možné začít psát kód.

Nezbytně nutným základním stavebním kamenem, je následující kód [32], který je nutné zkopírovat na začátek každého PostCSS projektu:

```
//Kód nezbytně nutný pro funkční zpracování CSS
var postcss = require('postcss'); //Reference na samotné PostCSS

module.exports = postcss.plugin('postcss-muj-plugin, function (opts) {
    opts = opts || {};
    // Zpracování voleb uživatele
    return function (root, result) {
        // Zde probíhají veškeré transformace CSS
    };
});
```

Z kódu je patrné, že se svým přístupem vývoj PostCSS pluginu nijak neliší od běžného javascriptového kódování.

Následná práce na vývoji nového nástroje by pak probíhala podle stejných syntaktických a logických pravidel jako je tomu v případě jakéhokoliv jiného javascriptového souboru. Jen je nezbytné připomenout, že pro otestování vzniklého nástroje je nutné umístit referenci na tento nástroj do iniciačního souboru, tak jak bylo popsáno v sekci s představením PostCSS. V místě názvu je nutné uvést název složky obsahující uživatelův nástroj.

Autor práce otestoval vývoj vlastního pluginu na několika konkrétních situacích a je nutné podotknout, že pro uživatele, který má zkušenosti s jazykem JavaScript je opravdu snadné nový nástroj vytvořit. A i kód sestávající se z pár desítek řádků dokáže práci s CSS velmi zefektivnit. Při tvorbě vlastního nástroje se však objevuje jeden problém a tím je nápad. Napsat vlastní nástroj je snadné, nicméně přijít se zajímavým novým řešením CSS už tolik ne, obzvlášť když PostCSS nabízí stovky pluginů, které pravděpodobně už uživatelův problém řeší.

6.2 Efektivní přístup k vývoji webové prezentace

Před tím, než přijde řada na používání některého z popsaných procesorů, je nutné zamyslet se nad tím, na jakém projektu uživatel pracuje a co od něj očekává. Neexistuje žádná tabulka, podle které lze přesně určit, kdy už je webová prezentace dostatečně rozsáhlá na to, kdy výhody zmíněných procesorů, či metodik začnou převyšovat úsilí vynaložené pro jejich aplikaci. Je třeba mít na paměti, že ne každý

projekt je stavěný na jejich používání. Stejně tak ne každý projekt vyžaduje rozsáhlé a složité funkční mixiny, které slouží například ke změně barvy na základě velikosti písma v daném bloku. Hnízdění, jak je známé z mnohých procesorů se zdá na první pohled syntakticky ideální, kolikrát však webová stránka vyžaduje zahnízdění čtvrté úrovně, aby došlo ke správnému obarvení položky v menu? Pravdou tedy zůstává, že nejdůležitější části tvorby samotného projektu je jeho důkladné promyšlení. Před tím, než dojde na psaní samotného kódu, je tak ideální chopit se prostého papíru a tužky a projekt si nakreslit. Už zde mohou vznikat první objekty, či části, které je pak možné dělit na základě metodik OOCSS a SMACSS a na základě tohoto testu určit další kroky.

Pokud je projekt dostatečně rozsáhlý na to, aby bylo vůbec třeba uvažovat nad zavedením podpůrných nástrojů do jeho vývoje, může dojít k zavedení předpokladu, že dojde k využití právě některé z metodik, ideálně ke kombinaci všech v práci zmíněných. K jejich zavedení není třeba žádných nových nástrojů, jejich zdokonalení vyžaduje jen velmi malé množství času, které je vykoupeno mnohem vyšší efektivitou a přehledností výsledného kódu. Neexistuje tedy žádný důvod, proč se jim vyhýbat.

6.3 Vhodná kombinace nástrojů

CSS metodiky a procesory začaly vznikat přibližně ve stejné době a v zásadě řeší podobnou problematiku. Vždyť mixin je v podstatě jakýmsi objektem, tak jak je známý z OOCSS přístupu. Každý z těchto přístupů však míří k cíli jinými prostředky. Zde se ukazuje základní problém. Je nutné využívat procesorů, pokud v projektu dochází k použití metodik? Již zmíněný mixin je zcela v rozporu s filozofií OOCSS, kde dochází k dědění vlastností pomocí vhodných selektorů a správně uspořádaného HTML. V popředí je tedy pokus o zamezení vzniku duplicit. Mixiny naopak nejsou ničím než chytrým kopírováním části kódu z jednoho místa na jiné. Valnou většinu mixinů je tak možné nahradit pomocí vhodně použitých metodik. Není tak nutné instalovat žádný procesor a také dochází k zamezení duplicit, které vznikají právě používáním mixinů. Funkční mixiny sice přinášejí řešení, které nemá v nativním CSS svého přímého zástupce, ale má tento přístup skutečně své praktické

opodstatnění? Jelikož dynamičnost procesorů končí ve chvíli jejich kompilace a není tak možné ovlivnit hodnotu parametrů vstupujících do těchto mixinů za chodu stránky, musí být logicky možné jejich funkcionalitu nahradit nativním CSS. Dynamiku, které se tyto parametrické mixiny snaží dosáhnout pak nabízí nativní CSS proměnné, jejichž stav, jak bylo ukázáno, je navíc možné měnit právě za chodu stránky.

Obdobným případem je i hnízdění, které rozvíjí funkcionalitu vnořených pravidel. Jak však nasvědčuje kapitola s metodikami, mezi základní pilíře většiny z nich patří minimalizace zanořování pravidel. Odkazování na rodiče na druhou stranu práci s metodikami značně zpříjemňuje, v případě hnízdění se však jedná spíše o vedlejší funkcionalitu.

Nativní CSS proměnné v dnešní době podporuje již valná většina prohlížečů a to stejné je možné říct o matematických funkcích [1], jejich schopnosti pak dalece přesahují ty známé z procesorů. Mají tedy procesory co nabídnout?

Co se procesorů týče, stále je zde funkce import, kterou není možné v nativním CSS nahradit a navíc je ji možné ideálně zkombinovat se zmíněnými metodikami. To je jeden plus jek pro preprocesory tak postprocesory. Jelikož je v současnosti možné nahradit snad jakoukoliv funkcionalitu preprocesorů některým z postprocesorových nástrojů, kdy navíc postprocesory přináší obrovské množství funkcí navíc a to zejména v případě autoprefixerů, či možnosti vytvářet nové nástoje, vzniká předpoklad, že právě postprocesory nabízí mnohem vyšší efektivitu a to konkrétně PostCSS, jelikož se stalo vítězem srovnání.

Preprocesory jako takové nepřináší vlastní řešení vendor prefixů, s výjimkou Sassu, který tak činí za pomoci frameworku Compass [7]. Ten však slouží jako jakási rozsáhlá knihovna mixinů. Takový mixin je pak na základě použití nutné vložit na určené místo v kódu. Uživatel tedy není odstíněn od nutnosti pamatovat si, kdy je a kdy není nutné tyto prefixy použít, autoprefixery na druhou stranu vyžadují jen stažení a instalaci. Žádné přemýšlení a bádání navíc.

Pokud se uživatel bez preprocesorů, ať už ze zvyku, či díky jejich pohodlnosti, neobejde, vhodným doporučením se zdá býti kombinace některého z populární

preprocesorů na základě vlastní volby a metodik **OOCSS**, **BEM** a **SMACSS** a PostCSS pluginu **Autoprefixer**.

OOCSS + BEM + SMACSS + LESS/Sass/Stylus + PostCSS Autoprefixer

Tato kombinace umožňuje skloubit to nejlepší ze všech světů a v krajních případech uživateli umožňuje sáhnout i po funkcích známých z preprocesorů a to okamžitě.

Je-li uživatel otevřený změnám, či preprocesory zcela nepoznamenán, nezbývá než nabídnout postprocesorovou variantu. Jedinou nevýhodou PostCSS oproti běžným preprocesorům tak může být složitější zaváděcí proces a nutnost jednotlivé pluginy stáhnout a nainstalovat. Pokud však uživatel snese tyto počáteční investice, PostCSS se mu odvděčí svým výkonem a především rozsáhlými možnostmi, které vychází z potřeb každého jednotlivého uživatele.

Jak již bylo zmíněno výše, za obrovskou výhodu preprocesorů může být považován jejich přístup k importům. I k nim nabízí PostCSS svou alternativu a to v podobě pluginu „**postcss-import**“, ač nebyl ve srovnávací části zmíněn, vyznačuje se podobnou funkcionalitou jako jeho starší preprocesoroví sourozenci. Samozřejmostí je pak zapojení pluginu podporujícího metodiku BEM, tedy „**PostCSS BEM**“, který může být nahrazen pluginem „**Postcss-Nested“** zejména s využitím jeho schopnosti odkazu na rodiče**.** Dále je vhodné využít služeb „**Custom properties**“, který podporuje nativní CSS proměnné a ty navíc rozvíjí svou zpětnou kompatibilitou s prohlížeči, které tyto proměnné nepodporují. Samozřejmostí je pak využití „**Autoprefixeru**“ a to vše v kombinaci s metodikami „**OOCSS**“, „**BEM**“ a „**SMACCS**“. Poté už záleží na každém konkrétním vývojáři jakou kombinaci dalších pluginů nabízených tímto procesorem, uzná za vhodnou.

Na základě provedeného výzkumu pomocí praktických testů při běžném vývoji webových stránek se zda být nejefektivnější možná kombinace představených metodik a nástrojů tato následující:

OOCSS + BEM + SMACSS + PostCSS BEM + Postcss-Nested + Custom properties + postcss-import + Autoprefixer

Po tomto doporučení už nezbývá než jen přiznat, že neexistuje jeden ideální recept na to, jak postupovat při vytváření webové prezentace a každý se liší právě na

základě konkrétního zadání a velikosti projektu. Tento návrh by měl však spolehlivě najít své uplatnění u většiny standardních webových stránek a pokud ani to nestačí, vždy je možné vytvořit vlastní nástroj pomocí PostCSS.

7 Závěr

V rámci práce došlo k představení základních webových technologií a jejich provázání s podpůrnými nástroji a postupy, které si dávají za cíl zefektivnit přístup autora webové stránky k jejímu vývoji.

Důraz byl kladen především na srovnání technologií postprocesorů, které na rozdíl od preprocesorů umožňují vylepšit přístup k vývoji bez nutnosti od základu tento přístup změnit. Závěrem plynoucím z tohoto srovnání je převaha nástroje PostCSS, nicméně i jeho konkurent v podobě Stylecow je na základě prověřených schopností velmi užitečným nástrojem, který neprávem nedosahuje ani zdaleka takové popularity.

Zejména propojení s popsanými metodikami, tak jak bylo popsáno v šesté kapitole, činí tyto nástroje vysoce efektivní a na základě provedených testů, mnohem silnější, než je tomu v případě preprocesorů. Možnost vytvořit vlastní nástroj pomocí postprocesorů je navíc velmi užitečným doplňkem, a proto není vhodné tuto funkcionalitu přehlížet.

Otázkou zůstává, zdali tyto nástroje přináší dostatečné výhody na to, aby uživatel, který s těmito technologiemi ještě nepřišel do styku, vyměnil pohodlí nativního CSS, které již odstranilo mnoho nedostatků, kvůli kterým zmíněné nástroje vznikaly, za schopnosti, které se dají v mnoha případech nahradit vhodným přístupem k CSS, tak jak bylo demonstrováno zmíněnými metodikami. Vždyť „custom properties“ v mnohém předčí alternativy proměnných známých z procesorů. Je jistě jen otázkou času, kdy CSS představí vhodnější řešení prefixů. A další nové funkce, které nemusí nabízet žádný současný podpůrný nástroj, možná právě vznikají. Nebo se v blízké budoucnosti objeví zcela nová technologie, která nahradí CSS?

To ukáže až čas.

8 Seznam použité literatury

[1] MOZILLA AND INDIVIDUAL CONTRIBUTORS. Learn to style HTML using CSS [online]. 2019 [cit. 2019-01-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS

[2] RAGGET, Dave, Jenny LAM, Ian ALEXANDER a Michael KMIEC. *Raggett on HTML 4*. 2nd ed. Harlow: Addison-Wesley Longman, 1998. ISBN ISBN 0-201-17805-2.

[3] BOS, Bert. A brief history of CSS until 2016 [online]. 2016 [cit. 2019-01-12]. Dostupné z: https://www.w3.org/Style/CSS20/history.html

[4] V, Victor. A Brief History of CSS [online]. 2016 [cit. 2019-01-12]. Dostupné z: http://www.css-class.com/a-brief-history-of-css/

[5] PRIETO, Ricardo. CSS4: The new version of CSS that will never exist [online]. 2018 [cit. 2019-01-13]. Dostupné z: https://www.silocreativo.com/en/css4-the-new-version-of-css-that-will-never-exist/

[6] JAVOREK, Honza. CSS preprocesory: méně psaní, vyšší efektivita [online]. 2011 [cit. 2019-01-15]. Dostupné z: https://www.zdrojak.cz/clanky/css-preprocesory-mene-psani-vyssi-efektivita/

[7] HAMPTON, Catlin, Natalie WEIZENBAUM a Chris EPPSTEIN. Sass [online]. c2006-2019 [cit. 2019-01-20]. Dostupné z: https://sass-lang.com

[8] MONUS, Anna. Popular CSS Preprocessors With Examples: Sass, Less & Stylus [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://raygun.com/blog/css-preprocessors-examples/

[9] LESS TEAM. Less [online]. Nedatováno [cit. 2019-01-20]. Dostupné z: http://lesscss.org

[10] FOSTER, Jed a Dale SANDE. Sassmeister [online]. 2015 [cit. 2019-02-15]. Dostupné z: https://www.sassmeister.com/about

[11] LAGENDIJK, Mark. Online Less Compiler [online]. c2011-2019 [cit. 2019-02-15]. Dostupné z: http://winless.org/online-less-compiler

[12] GERCHEV, Ivaylo. A Comprehensive Introduction to Less: Mixins [online]. 2012 [cit. 2019-01-28]. Dostupné z: https://www.sitepoint.com/a-comprehensive-introduction-to-less-mixins/

[13] BROWN, Tiffany B. No seriously: Don't use @extend [online]. 2017 [cit. 2019-01-29]. Dostupné z: https://webinista.com/updates/dont-use-extend-sass/

[14] BAUMGARTNER, Stefan. Deconfusing Pre- and Post-processing [online]. 2015 [cit. 2019-02-05]. Dostupné z: https://medium.com/@ddprrt/deconfusing-pre-and-post-processing-d68e3bd078a3

[15] SITNIK, Andrey. PostCSS Second Birthday [online]. 2015 [cit. 2019-02-25]. Dostupné z: https://evilmartians.com/chronicles/postcss-second-birthday

[16] ANJIT, Preethi. Understanding CSS Writing Methodologies [online]. 2018 [cit. 2019-02-10]. Dostupné z: https://www.hongkiat.com/blog/css-writing-methodologies/

[17] SULLIVAN, Nicole. Object Oriented CSS [online]. 2010, aktualizováno 26 Dec 2013 [cit. 2019-02-10]. Dostupné z: https://github.com/stubbornella/oocss/wiki

[18] YANDEX. BEM Methodology [online]. Nedatováno [cit. 2019-02-10]. Dostupné z: https://en.bem.info/methodology/

[19] SNOOK, Jonathan. Scalable and Modular Architecture for CSS [online]. 2012 [cit. 2019-02-15]. Dostupné z: https://smacss.com/

[20] ČERNÝ, Václav. CSS preprocesory jako efektivní nástroj pro tvorbu frontendu. Hradec Králové, 2018. Bakalářská práce. Univerzita Hradec Králové. Vedoucí práce Doc. Ing. Filip Malý, Ph.D.

[21] OLIVEIRA, Vincent De. Pleeease [online]. c2014-2016 [cit. 2019-02-25]. Dostupné z: http://pleeease.io/

[22] SITNIK, Andrey. PostCSS [online]. 2013, aktualizováno Feb 14, 2019 [cit. 2019-02-25]. Dostupné z: https://github.com/postcss/postcss

[23] CODEPEN. CodePen [online]. c2019 [cit. 2019-02-25]. Dostupné z: https://codepen.io/

[24] OTERO, Oscar. Stylecow [online]. 2015 [cit. 2019-02-25]. Dostupné z: http://stylecow.github.io/

[25] SITNIK, Andrey. Autoprefixer [online]. 2013, aktualizováno Feb 28, 2019 [cit. 2018-05-11]. Dostupné z: https://github.com/postcss/autoprefixer

[26] NPM, INC. Node Package Manager [online]. 2009 [cit. 2019-04-20]. Dostupné z: https://www.npmjs.com/

[27] SITNIK, Andrey. PostCSS Simple Variables [online]. 2013, aktualizováno Feb 18, 2019 [cit. 2018-15-11]. Dostupné z: https://github.com/postcss/postcss-simple-vars

[28] THIROUIN, Maxim. PostCSS Custom Properties [online]. 2014 [cit. 2019-04-20]. Dostupné z: https://github.com/postcss/postcss-custom-properties

[29] CLARK, David. Postcss-extend [online]. 2015 [cit. 2019-04-20]. Dostupné z: https://github.com/travco/postcss-extend

[30] SITNIK, Andrey. PostCSS Nested [online]. 2013, aktualizováno Sep 09, 2018 [cit. 2018-15-11]. Dostupné z: https://github.com/postcss/postcss-nested

[31] SITNIK, Andrey. PostCSS Mixins [online]. 2015 [cit. 2019-04-20]. Dostupné z: https://github.com/postcss/postcss-mixins

[32] BRACEY, Kezz. PostCSS Deep Dive: Create Your Own Plugin [online]. 2015 [cit. 2019-03-30]. Dostupné z: https://webdesign.tutsplus.com/tutorials/postcss-deep-dive-create-your-own-plugin--cms-24605

9 Seznam použitých tabulek

10 Seznam použitých obrázků

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Thér Jan	Královec 61, Královec	I1600621

TÉMA ČESKY:

Nástroje pro zefektivnění práce s CSS

TÉMA ANGLICKY:

Tools for Increasing Efficiency of Work with CSS

VEDOUCÍ PRÁCE:

Mgr. Daniela Ponce, Ph.D. - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

Student se seznámí s nejčastěji používanými nástroji a postupy pro zefektivnění práce s CSS, posoudí jejich silné a slabé stránky ve využití pro vytváření několika typických webových aplikací a své zkušenosti zformuluje v podobě návrhů na funkční vylepšení.

Osnova:

1. Úvod
2. Vybrané základní webové technologie
3. Vybrané podpůrné nástroje kódování
4. Metodické postupy kódování a jejich navázání na nástroje
5. Porovnání nástrojů a vyhodnocení
6. Možná vylepšení
7. Závěr

SEZNAM DOPORUČENÉ LITERATURY:

Kennedy A., de León I. Dynamic CSS. In: Pro CSS for High Traffic Websites. Apress, Berkeley, CA (2011) ISBN 978-1-43023289-6.

JACKSON, Brian. PostCSS - Transforming Your CSS with JavaScript [online]. OCTOBER 18, 2018 [cit. 2018-10-29]. Dostupné z: https://www.keycdn.com/blog/postcss

BAUMGARTNER, Stefan. Deconfusing Pre- and Post-processing [online]. Aug 12, 2015 [cit. 2018-10-29]. Dostupné z: https://medium.com/@ddprrt/deconfusing-pre-and-post-processing-d68e3bd078a3

Podpis studenta: ..... **Datum:** 15.10.2018

Podpis vedoucího práce: ..... **Datum:** 15.10.2018