Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Analýza procesu vývoje komponenty Rich Text Editoru

Diplomová práce

Autor: Bc. Tomáš Archalous

Studijní obor: Informační management

Vedoucí práce: Ing. Karel Mls, Ph.D.

Hradec Králové Duben 2020

Prohlášení:

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

V Hradci Králové dne 19.8.2020

Bc. Tomáš Archalous

Poděkování

Chtěl bych poděkovat vedoucímu mé práce Ing. Karlovi Mlsovi, Ph.D. za odborný dohled, cenné rady a motivaci k napsání této práce.

Dále bych chtěl poděkovat mým bývalým a současným kolegům ve firmě QUADIENT s.r.o. Konkrétně Zdeňku Obstovi za skvělou práci teamleadera, Jiřímu Mikešovi za pečlivou práci quality assurance a Michalovi Bašemu s Martinem Hakem za spolupráci při vývoji Rich Text Editoru a psaní této práce.

Anotace

Tato Diplomová práce popisuje postup vývoje webové komponenty pro pokročilou editaci textu – Rich Text Editoru. Jsou zde popsány požadavky funkcionální vlastnosti, jeho technické parametry, použité technologie a proces vývoje. V práci jsou rovněž vyjmenovány problémy, se kterými by se vývojář obdobné komponenty reálně setkal a musel řešit. U těchto problému je popsána podrobná analýza, jakožto i firemní procesy, které tyto rozhodovací procesy doprovázely.

Cílem této práce je poskytnout programátorům, architektům, manažerům a user experience designerům ucelený souhrn potřebných informací k vývoji Rich text Editoru.

Klíčová slova

Rich Text Editor, Javascript, Bobril, Webová komponenta, Scrum, Agile

Annotation

This diploma thesis describes the process of developing a web component for advanced text editing - Rich Text Editor. It describes the requirements of functional properties, its technical parameters, technologies used and the development process. The work also lists the problems that a developer of a similar component would encounter and must solve. A detailed analysis of these problems is described, as well as the business processes that accompanied these decision-making processes.

The aim of this work is to provide programmers, architects, managers and user experience designers with a comprehensive summary of the information needed to develop a Rich Text Editor.

Title

Analysis of the development process of the Rich Text Editor component

Keywords

Rich Text Editor, Javascript, Bobril, Web Component, Scrum, Agile

Obsah

1 Úvod

V druhé dekádě dvacátého prvního století jsme byli svědky masivního rozšíření zařízení pro prohlížení webových stránek. Zároveň se připojení k internetu stalo natolik dostupné a spolehlivé, že je dnes považováno za samozřejmost. Díky tomu se také změnil způsob konzumace a vytváření digitálního obsahu. Uživatelé se naučili přijímat streamovaný obsah a ztratili zájem o lokálně uložená data. Stejný osud postihl i poskytování služeb. Velké a těžkopádně desktopové aplikace, které jsou mnohdy náročné na instalaci a údržbu, nahradily vždy webové aplikace. Tyto aplikace nabízí uživatelům okamžitý přístup k poskytovaným službám, jednotné uživatelské prostředí webového prohlížeče a další výhody spojené s možností online podpory.

Díky stále vyšší optimalizaci běhu Javascriptového kódu v moderních prohlížečích je možné nabízet přímo na webu aplikace, které dříve byly pouze desktopové. Tento přerod šel ruku v ruce s vývojem nových technologií, které umožňovaly jak plynulý chod webové aplikace pro uživatele, tak možnost snadného a rychlého vývoje pro vývojáře. Začalo vznikat nepřeberné množství Javascriptových frameworků a podpůrných vývojářských nástrojů.

Tato práce popisuje vývoj jedné takovéto webové aplikace. Konkrétně její komponenty Rich Text Editor. Je zde popsán životní cyklus vývoje od prvotního plánování až po testovací fázi. Můj osobní přínos při vývoji komponenty spočíval zejména v implementaci modulů pro stylování a vkládání referencovaných stylů. V oblasti grafiky jsem implementoval veškeré uživatelské rozhraní mimo textovou oblast. Ostatní vývojáři měli na starost zejména selekci textu, zadávání znaků a kopírování a vkládání textu. Kolektivně jsme se s týmem podíleli na plánovací fázi, kde jsme vybírali technologie, které je použít při vývoji, způsoby ukládání dat a metody vykreslování datového modelu do webového prohlížeče.

2 Požadavek na vytvoření Rich Text Editoru

V roce 2016 započal ve firmě GMC s.r.o. (později QUADIENT s.r.o.) vývoj produktu s interním označením RMAD – Rapid Mobile Apps Development. Tento software si kladl za cíl poskytnout korporátním zákazníkům komplexní nástroj pro komunikaci se svými zákazníky. Jeho klíčovou funkcionalitou bylo designování personifikované elektronické pošty, fyzické pošty a rovněž, což bylo považováno za klíčovou funkci oproti konkurenci, designování vlastních mobilních aplikací pro operační systémy Android a iOS. Pro mnohé tyto služby již GMC dlouhé roky poskytoval své produkty. Cílem RMADu bylo za prvé sjednocení těchto produktů, a hlavně jejich přesun do webového prostředí, jelikož distribuce služeb prostřednictvím desktopových aplikací bylo vyhodnoceno jako uživatelsky nepřívětivé. Z technického pohledu tady bylo nutné vytvořit webový editor, kde si uživatel může pomocí Drag and dropu sestavovat podobu emailů, fyzické pošty a mobilních aplikací. Jednalo se tedy a pokročilejší WYSIWYG (What you see is what you get) editor.

Uživatel si tyto objekty sestavoval pomocí skládání jednotlivých komponent. Tyto komponenty byly dvojího typu. Mohly být buď uživatelsky vytvořené, například layout, header, stylizované textové pole, nebo předdefinované. Předdefinované komponenty se vyznačovaly vyšší složitostí, které by běžný uživatel nebyl schopen v editoru dosáhnout. Byly to konkrétně upload area, základní datové tabulky nebo již zmiňovaný Rich Text Editor.

Účelem Rich Text Editoru však nemělo být pouze poskytnout koncovému uživateli možnost vkládat do hotové aplikace text, ale hlavně měl sloužit především designerovi této aplikace pro vytváření jednotlivých komponent. Pro uvedení příkladu je možné si představit situaci, kdy designer aplikace chce vytvořit nadpis v zelené barvě s kurzívou. Do prázdného projektu si tedy vloží komponentu Header, do ní vloží základní komponentu Textové pole, u které se po dvojkliku zobrazí textový editor s možností editace. Zde může designer provést jednoduché textové úpravy podobně jako v programu Microsoft Word.

3 Příprava na vývoj

3.1 Vlastní versus cizí řešení

Před tím, než začal samotný vývoj komponenty, byl investován přibližně měsíc analýze a architektuře. Jedním z prvních rozhodnutí bylo, zda vytvořit kompletně vlastní editor nebo použít již existující řešení. Rozhodovalo se mezi těmito třemi základními scénáři budoucího vývoje

3.1.1 Použití existujícího řešení v QUADIENTu

První variantou bylo znovupoužití již vyvinutého Rich Text Editoru, který se používal na jiných produktech. Výhodou by bylo navázání na vývoj ve firemním frameworku Bobril, se kterým jsou již vývojáři seznámeni a snazší integrace do nového produktu. Tímto však výčet pozitiv končí. V editoru chyběly pokročilé možnosti editování, které byly vyžadovány, a struktura kódu neumožňovala jednoduchou implementaci těchto funkcí. Znovupoužití starého editoru bylo zavrženo. Velice užitečné však byly zkušenosti vývojářů, kteří na této komponentě v minulosti pracovali a mohli nás upozornit na možné chyby, kterým jsme se později mohli vyvarovat.

3.1.2 Použití editoru třetí strany

Druhou variantou bylo využití existujícího editoru vytvořeného třetí stranou. Reálně se zvažovaly následující produkty:

- **CKEditor.** CKEditor je jako jediný ze zvažovaných WYSIWYG placený. Je momentálně jedním z nejstarších textových editorů na webu. Jeho první verze byla vydána roku 2003 a od té doby byl mnohokrát zcela přepsán. Nabízí široké možnosti pluginů a dalších uživatelských rozšíření. Toho však není plně využito, neboť uživatelská základna je velmi malá. Jeho přednostmi je zaměření na moderní prohlížeče a pravidelné updatování. Snaha udržet editor plně funkční s moderními prohlížeči je však vykoupena nižší kompatibilitou se starými prohlížeči. To je velký problém zejména v korporátním prostředí, kde aplikace běží mnohdy na již nepodporovaných a neaktualizovaných systémech. CKEditor je možné doporučit pro menší projekty bez potřeby komplexní funkcionality pro běžné uživatele. (1)

- **Draft.js**. Velké nadšení komunity vývojářů vyvolalo vydání editoru Draft.js společností Facebook. Ačkoliv je Draft.js publikovaný jako open-source, je jeho vývoj řízen primárně požadavky Facebooku. Z tohoto důvodu ho však lze označit jako nejspolehlivější a nejlépe fungující WISIVIG editor, který byl zvažován. Největším problémem je však jazyk, ve kterém je Draft.js napsán – Flow. Flow se v javascriptové vývojářské komunitě neuchytil a byl téměř zcela nahrazen Typescriptem od Microsoftu. Nutno podotknout, že Draft.js je ze zde zmíněných editorů jediným, který nabízí zdrojový kód s plnou typovostí. (2)
- **Slate**. Velice kvalitním a funkčním editorem je Slate. Slate je ve své podstatě přepsaný a reimplementovaný Draft.js s přidanou podporou pluginů třetích stran. Problémem je však pomalý vývoj. Slate je vyvíjen jedním vývojářem a jakékoliv opravy, či přidávání nových funkcionalit trvá velice dlouho. Přes to je Slate velice kvalitní software, který však zcela postrádá typovost, testy a je zde mnoho architektonických problémů, u kterých může trvat roky, než budou opraveny. (3)
- **Prosemirror**. Prosemirror je editor, který svými funkcemi a pozorností věnované detailům vyčnívá nad všemi ostatními. Ostatní editory dost často kopírují jeho funkce a UX design. Zejména oblasti podpory mobilních zařízení. Největší problém je zde však s čitelností kódu. I pro vývojáře je velice těžké vytvořit něco více než základní ukázky editoru. I když je tedy Prosemirror z hlediska použitelnosti na vysoké úrovni, jakákoliv komplexnější implementace je pro vývojáře velice obtížná. (1)
- **Quill**. V dnešní době jeden z nejpopulárnějších editorů. Opět je vhodný pro menší projekty bez potřeby komplexnější funkcionality, jelikož i jeho samotná funkčnost je velmi omezená. Jeho nespornou výhodou je implementace API, díky kterému nemusí ke zjištění stavu editoru číst podobu aktuálního HTML elementu. Například ke zjištění stylu textu na páté až šesté pozici, stačí provolat metodu **getFormat(5,2)**. (4)

3.1.3 Vytvoření vlastního editoru

Třetí varianta, která byla nakonec vybrána jako nejvýhodnější z dlouhodobého hlediska, byla vytvoření kompletně vlastního řešení za použití interních firemních nástrojů a frameworků. Důvody byly následující:

- **Rozšiřitelnost o další funkcionalitu.** Prvním hlavním důvodem, proč bylo rozhodnuto vyvinout vlastní komponentu, byla potřeba rozšíření o dodatečnou funkcionalitu. Frameworky třetích stran tuto cestu sice nezavírají, ale jelikož nepodporují pluginy, bylo by nutné buď kód při každém vydání nové verze aktualizovat, nebo update ignorovat. Rovněž je zde stejný problém jako při použití již hotového firemního řešení. Konkrétně nepřipravenost architektury kódu pro další funkcionalitu.
- **Kompatibilita frameworků.** Díky použití stejného frameworku, jako na zbytek aplikace, není nutné, aby vývojář neustále přepínal svůj kontext při přechodu z vnější aplikace do komponenty. Rovněž také odpadá potřeba případného druhého kompilátoru a běhového prostředí pro vývoj a zbuildění aplikace.
- **Datová struktura.** Produkt RMAD byl vytvořen pro použití specifické jednotné datové struktury pro všechny aplikace. Pokud bychom použili řešení třetí strany, bylo by nutné vytvořit rozhraní, které by zachycovalo změny v nativní datové struktuře a převedlo jej do formátu podporovaného RMADem. Ačkoliv by toto bylo technicky možné, znamenalo by to značnou výpočetní zátěž na straně klienta, která i tak byla na vysoké úrovni a bylo nutné ji optimalizovat na minimum.
- **Znalost kódu.** Díky vývoji celé komponenty uvnitř jednoho týmu jde jednoduše a rychle přenášet informace mezi jednotlivými členy týmu. V případě opravy či navázání na existující kód vytvořený třetí stranou by docházelo k značnému časovému prodlení a zanášení možných chyb.
- **Nízká kvalita kódu.** Při bližším testování a analýze jednotlivých existujících řešení se nejevila uspokojivá ani kvalita kódu. Mnohdy se v editorech vyskytovaly chyby během zadávání, grafické artefakty a ve všech chyběly unit testy, které by zaručily funkčnost i po případných změnách v kódu.

3.2 Použití Javascriptového frameworku

Vzhledem k plánované komplexnosti chystaného editoru bylo nezbytné využit některý z Javascriptových frameworků. Tyto frameworky nabývají na popularitě a z důvodu vysoké poptávky roste i jejich nabídka. Díky použití frameworku je možné zrychlit vývoj softwaru. V moderních frameworcích jsou již defaultně implementovány základní funkce, které uživatel potřebuje k vývoji. Kromě zjednodušení a zrychlení psaní kódu nabízí zvýšení efektivnosti díky jednotnosti, což umožňuje jednoduchou spolupráci více vývojářů na jednom projektu. Různé frameworky nabízí různou míru komplexity a abstrakce kódu, což dělá každý framework vhodný na jiný druh aplikace. Zde jsou vyjmenovány frameworky, které byly zvažovány k vývoji Rich Text Editoru. (5)

3.2.1 React

React je aktuálně jedním z nejpoužívanějších frameworků pro vývoj single-page aplikací. Jedná se o knihovnu vytvořenou a spravovanou společností Facebook. React, stejně jako většina moderních Javascriptových frameworků, je založen na principu virtuálního DOMu (Document Object Model), který umožňuje jednoduše provádět změny ve skutečném DOMu webové stránky bez nutnosti překreslování. React se opírá a početnou uživatelskou komunitu, která vytváří velké množství doplňků třetích stran. React v roce 2019 implementoval použití funkcionálních komponent (hooků), které umožňují psaní komponent s menším množstvím kódů a přehlednější strukturou. Problémem Reactu oproti ostatním frameworkům je jeho velikost a pomalost, která s rostoucím počtem funkcí stále narůstá. (6)

3.2.2 Angular

Kromě Facebooku má svůj vlastní open-source Javascript framework také Google Angular. Stejně jako React je hlavní zaměření Angularu na vývoj single page aplikací. Hlavními výhodami Angularu je nativní podpora Typescriptu, která byla dlouhou dobu hlavním argumentem oproti Reactu. Angular je rovněž od počátku přizpůsoben pro vývoj mobilních aplikací za účelem optimalizace jejich výkonu a snadnější implementace responsibility. Za zmínku stojí rovněž podpora dependency injection a Angular Language Service, který nabízí vývojářům pokročilé možnosti automatického dokončování kódu, zobrazování chybových hlášek v reálném čase, nápovědu a navigaci napříč komponentami. (7)

3.2.3 Svelte

Minimalistický framework, jehož autorem je Rich Harris. Jeho předností je, že silně pracuje se zkompilovaným kódem. V praxi to znamená, že vývojář píše kód, který vzdáleně připomíná Javascript a kompilátor ho následně přeloží do čistého Javascriptu. Toto umožňuje velmi jednoduchou implementaci reaktivnosti. Reálně tedy každá proměnná, která je vytvořena a vykreslena v prohlížeči je reaktivní, a prohlížeč ji překreslí kdykoliv je v kódu změněna její hodnota. Výhodou je velikost zkompilovaného kódu, který je velice kompaktní. Na druhou stranu je zde problém s dalšími moduly, které jsou vyžadovány pro běh zkompilovaného kódu a finální aplikace tímto ztrácí na své kompaktnosti. Na rozdíl od ostatních javascriptových frameworků Svelte nepoužívá virtuální DOMy, což by ulehčovalo práci s defaultními dekorativními funkcemi v elementu contentEditable. (8)

3.2.4 ivi

Knihovna ivi dosahuje velmi dobrých hodnot v benchmarcích a zároveň je velmi minimalistický v oblasti množství potřebného kódu a velikosti aplikace. Nezískal si však mezi vývojáři příliš velkou popularitu z důvodu přísného funkcionálního přístupu a absence TSX (TSX je obdobou JSX pro Typescript umožňující vkládání HTML tagů do Javascriptového kódu). Tímto frameworkem, byl z velké části inspirován Bobril, zejména v oblasti použití virtuálního domu. (9)

3.2.5 Blazor

Blazor je open-sourcový framework vyvíjený Microsoftem, který se rovněž snaží uchytit na poli Javascriptových frameworků. První verze Blazeru pracovala s konceptem, kdy veškerý kód běží na backendu a frontend se synchronizoval za pomocí SignalR, což je knihovna určená pro práci s websockety, a synchronizuje změny provedené na backendu s frontendem. Tento přístup byl v rozporu s aktuálním vývojem moderních webových aplikací, kdy se pomocí single-page aplikací snažíme přenést výpočetní výkon z backendu na frontend.

Druhá aktuálně vyvíjená verze využívá C#, který je zkompilován do WebAssembly, které již běží přímo v prohlížeči a s backendem komunikuje pomocí klasického API. Problém je zde ten, že výsledná aplikace nedosahuje rychlostí ani velikostí Javascriptovému kódu. Důvodem je to, že i přes rychlejší vnitřní výpočty musí

WebAssembly stále komunikovat s uživatelem přes virtuální HTML DOMy stejně jako Javascript, který pro tuto práci byl od prvopočátku primárně vyvíjen. Uplatnění však může najít v enterprise sféře, kde je více C# vývojářů, kteří odmítají vytvářet single-page aplikace v Javascriptu. (10)

3.2.6 Stencil

Stencil je framework, který se snaží primárně stavět na web komponentách. Tento přístup je vhodný u malých komponent, které s rozhraním sdílí malé množství parametrů. Nicméně pokud bychom chtěli tímto způsobem napsat celou stránku, tak bychom dříve nebo později potřebovali knihovnu typu Bobflux, která by nám zajišťovala reaktivitu v prohlížeči. (9)

3.2.7 Bobril

Hlavním a nakonec také vítězným kandidátem pro javascriptové framework Rich Text Editoru je Bobril. Bobril je rovněž open source framework vyvíjený Borisem Letochou. V současné době se využívá téměř ve všech webových aplikacích společnosti QUADIENT, kde postupně nahrazuje Angular od Googlu. Bobril je označovaný jako Reactlike framework. Ačkoliv byl Bobril napsán zcela od základu bez použití jiných řešení, jeho struktura a použití jsou velice podobné Reactu. Tato skutečnost umožňuje snadné přeorientování nových vývojářů právě z Reactu na Bobril. Stejně jako u Reactu se zde využívá virtuálního domu, který umožňuje rychlé překreslování stránek. Bobril byl od počátku vyvíjen pro použití v Typescriptu, což umožňuje jeho nasazení v komplexních systémech. (11)

Použití interního frameworku naráží na problém s neexistující komunitou. Tento fakt je však vyvážen tím, že lidé, kteří se přímo podíleli na vývoji, jsou vždy k dispozici vývojářům. Díky tomu je v případě nalezení chyby, či potřeby přidání funkcionality, možné dodat potřebné změny v řádu několika dní.

Bobril je ze své podstaty frameworkem zaměřeným na používání komponent. Jak již bylo řečeno, neexistuje komunita, která by tyto frameworky vyvíjela. Proto byla QUADIENTem vytvořena interní galerie komponent, kam zaměstnanci mohou přispívat a pomocí Yarnu implementovat již hotové komponenty. Zde je ukázka jednoduché komponenty časovače za použití TSX syntaxe.

```
import * as b from "bobril";

class Timer extends b.Component {
  private _time: number = 0;
  private _intervalId: number = 0;

  init(): void {
    this._intervalId = window.setInterval(() => this.tick(), 1000);
  }

  render(): b.IBobrilChildren {
    return (
      <>
        <p>Time: {this._time} [s]</p>
    <button onClick={() => this.reset()}>RESET</button>
    </>
  );
  }

  destroy(): void {
    window.clearInterval(this._intervalId);
  }

  private tick(): void {
    this._time++;
    b.invalidate(this);
  }

  private reset(): boolean {
    this._time = 0;
    b.invalidate(this);
    return true;
  }
}

b.init(() => <Timer />);
```

Zdrojový kód 1 : Komponenta časovače (11)

3.3 Použití Content Editable

Jedno z nejdůležitějších rozhodnutí, které bylo potřeba udělat před započetím vývoje, bylo, zda chceme jako jádro Rich Text Editoru použít nativní HTML element Content Editable, nebo vytvořit kompletně vlastní řešení.

Contenteditable je HTML atribut, který určuje, zde je obsah daného elementu editovatelný. V Praxi to znamená, že při použití parametru contenteditable na element obsahující text, vykreslí prohlížeč uživateli WYSIVIG editor. Ten nabízí základní funkce pro editaci textu jako je psaní, mazání a formátování textu pomocí klávesových zkratek. Klávesovými zkratkami lze naformátovat text na kurzívu, tučné písmo či podtržení. Do contenteditable je možné také vložit obsah schránky, který je následně

přetransformován na HTML objekty, vložen do zdrojového kódu stránky a vykreslen uživateli. (12)

```
<div contenteditable="true">Text
  <div><i>itallic</i></div>
  <div><b>bold</b></div>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</div>
```

Zdrojový kód 2: Contenteditable s naformátovaným textem. Zdroj: Vlastní

V tomto bodě se může zdát, že má vývojář již hotovou práci a není potřeba implementace žádné dodatečné funkcionality. A v některých případech tomu tak skutečně může být. Pokud je vyžadována pouze základní funkcionalita pro zadávání textu s jednoduchým formátováním, stačí po vytvoření elementu s contentEditable již pouze napsat skript, který po uložení přečte obsah elementu a dál ho zpracuje. Tento proces je však neslučitelný s filozofií Javascriptových frameworků využívající virtuální DOMy. Pokud by byly povoleny tyto defaultní akce vyvolané elementem s contenteditable, vznikl by konflikt mezi reálnou podobou DOMu a jejím virtuálním otiskem v paměti frameworku – v tomto případě Bobrilu. Bobril by rovněž nevěděl, že byly provedeny změny v DOMu a nemohl by na ně reagovat. Z tohoto důvodu je nutné veškeré defaultní akce v contentEditable vypnout a implementovat je ručně odchytáváním akcí uživatele. Může se zdát, že tímto jsme však zahodili všechny funkce contentEditable a nahradili je vlastní implementací. Nicméně je třeba si uvědomit, že contentEditable se také stará o zobrazování pozice karetu. Karet je označení pro indikátor, který uživateli říká, kde se momentálně v textu nachází a do jaké části se budou vkládat nově zadané znaky. Tuto funkci by v případě vlastního řešení bylo rovněž nutné kompletně reimplementovat. (13)

Po zvážení všech pro a proti použití contentEditable oproti vytvoření vlastního řešení bylo rozhodnuto pro contenteditable. Předpokládalo se, že již hotová implementace karetu a dalších funkcí, které jsou specifické pro jednotlivé prohlížeče

ušetří práci a čas na vývoji. Pokud se nyní po vytvoření Rich Text Editoru ohlédneme zpět, shoduje se tým na tom, že se jednalo o špatné rozhodnutí.

Při implementaci uživatelských vstupů na základě pozice karetu a při selekci se začaly objevovat další a další problémy, kdy byl problém synchronizovat skutečný stav DOMu a jeho uloženou podobu v Bobrilu. Tyto problémy umocňovala rozdílná implementace contentEditable napříč jednotlivými prohlížeči a jejich vykreslovacími jádry. Tyto problémy budou přesněji popsány v dalších kapitolách.

Na základě těchto zkušeností bychom budoucím vývojářům, při použití Javascriptového frameworku s virtuálním DOMem, doporučili zcela vynechat contentEditable a zaměřit se na vytvoření vlastního řešení.

3.4 Datová Struktura

Po rozhodnutí o použití elementu contentEditable přišel na řadu návrh podoby datové struktury Rich Text Editoru. Díky DSL modelingu je možné využít stejnou datovou strukturu jak pro uchování dat v přímo v editoru na straně klienta, tak je ve stejné struktuře mít uložená na backendu. Na backendu jsou tato data rovněž zpracovávána jazykem C#. Z tohoto důvodu, kdy na definování datové struktury závisela práce ostatních vývojářů napříč týmy, bylo nutné do její podoby investovat větší množství času a předejít tak budoucím přepisům. Tyto přepisy by se finančně prodražily nejen z důvodu opětovné implementace na více místech, ale i kvůli nutnosti vytvořit updatovací skripty, které by aktualizovaly již existující data. Psaní updatovacích skriptů by však bylo potřeba řešit pouze v případě, že je ostrá verze editoru již nasazena u zákazníků v produkci.

3.4.1 Ukládání dat do stringového řetězce

První, poněkud divokou myšlenkou na uložení dat, byl návrh ukládat aktuální podobu obsahu elementu contentEditable v čistém stringu. Respektive převést vnitřek HTML elementu včetně všech jeho potomků na stringový řetězec a ten poté odeslat na backend. Při přijmutí dat s backendu by byl text naparsován klientem na jednotlivé HTML domy se všemi parametry. Při tomto řešení by odpadla nutnost designování struktury dokumentu. Rovněž by se také snížila velikost dat, které by bylo potřeba

odesílat a přijímat ze serveru, neboť uložení textu v čistém HTML by nemuselo obsahovat složité jazykové definice a odkazy pro DSL modeling.

Hlavním problémem tohoto přístupu je však nejednotnost v zobrazování výsledného HTML kódu napříč prohlížeči. Tato problematika je blíže přiblížena v páté kapitole. Dále by tento způsob ukládání mohl v budoucnu zkomplikovat vkládání speciálních objektů, jako jsou například proměnné. Jelikož by tyto objekty nebylo možné nadefinovat pomocí klasických HTML elementů, bylo by nutné je nahradit například DIV elementy označenými speciálními parametry, které by vykreslovací komponentě řekly, jak s nimi zacházet. Tento přístup by byl s rostoucím počtem objektů a značek stále méně přehledným a dlouhodobě neudržitelný. Dále by se tím zcela zavřely dveře pro ukládání změn za pomocí změnových vektorů. Respektive změnový vektor by se rovnal celému obsahu Rich Text Editoru a kompletně by tím ztratil na významu.

3.4.2 Použití stromové struktury

Jako vhodná alternativa k ukládání dat do stringového řetězce byla vybrána možnost použití stromové struktury. Vzhledem k použití jazyku Javascript, byl logicky použit formát JSON.

Základní uložitelná instance Rich Text Editoru je tedy tvořena JSON, která je typu ImodelJson (viz. Zdrojový kód 3). Tento interface je poskytován DSL infrastrukturou a má následující parametry.

- Uid – Unikátní identifikátor, který musí být specifický pro každou jednu instanci ImodelJson. Nejde tedy pouze o instance Rich Text Editoru, ale o všechny modely použité v aplikaci.
- Name – Parametr „name“ není pro fungování modelu povinný. Jde spíše a pomocnou proměnnou pro vývojáře při vytváření a pojmenovávání unit testů.
- Uses – Nevýhodou jazyku JSON je, že nativně neumožňuje použití referencí na jiné, jako například XML (14). Z tohoto důvodu jsou v DSL modelu použity takzvané „uses“. Každý tento „use“ je v tomto objektu definován klíčem a k němu přiřazeným unikátním názvem definice daného jazyka. Klíče těchto identifikátorů jsou pak použity v jednotlivých nodech rootu.

- Roots – jedná se o pole základních nodů, ve kterých jsou uložena data každé instance modelu. Každý z těchto nodů musí mít unikátní identifikátor typu number. Ten je v objektu uložen pod znakem dolaru. Druhý parametr, který je uložen pod znakem podtržítka, značí typ daného nodu. Zde jsou využity klíče z „uses“, které referencují jednotlivé definice jazyků. Tyto klíče jsou složeny do jednoho stringu spolu s identifikátorem typu v daném modelu a jeho názvem. Název je opět nepovinná položka, jelikož daný typ je již přesně identifikován unikátním číslem. Po těchto dvou základních parametrech následuje již samotné vypsání jednotlivých parametrů nodu a k nim přiřazených hodnot. (viz. Zdrojový kód 4)

```
export const exampleData: IModelJson = {
  uid: "1234-1234-1234-1234",
  name: "Test Model",
  uses: {
    a: "richEditorViewDef0.1",
    b: "richEditorModelDef0.1",
    d: "expressionDef0.1"
  },
  roots: [
    {
      $: 2,
      _: "b:1:Document",
      "b:2:blocks": [
        {
          $: 3,
          _: "b:3:Block",
          "b:5:fragments": [
            {
              $: 4,
              _: "b:21:TextFragment",
              "b:22:value": "Text Fragment content 1"
            },
          ]
        }
      ]
    }
  ]
};
```

Zdrojový kód 3: Ukázka základní struktury dokumentu Rich Text Editoru (15)

```
{
  $: 14,
  _: "c:11:CssPropertyCustom",
  "c:9:key": "textAlign",
  "c:10:important": false,
  "c:12:value": {
    $: 15,
    _: "d:99:StringValue",
    "d:100:value": "left"
  }
```

Zdrojový kód 4: Ukázka nodu s více použitými parametry (15)

3.4.3 Definice jazyka

V Rich Text Editoru byly použity čtyři základní definice jazyka. Konkrétně richEditorViewDef, expressionDef, richEditorModelDef a stylesEditorDef. Tyto modely slouží pro uložení dat od aktuálního uživatelského stavu editoru až po definici jednotlivých textových proměnných.

3.4.3.1 Definice zobrazení editoru

Pro uložení aktuálního stavu editoru slouží richEditorViewDef (Rich Editor View Definition Language). Z hlediska datové struktury obsahuje také informace o všech fragmentech v textu a externích CSS stylech. Ty jsou však definovány v samostatných modelech richEditorModelDef a stylesEditorDef. Nejdůležitější definicí tohoto modelu je uchování dat o pozici karetu, který je definován datovým typem EditorPosition. Jak již bylo zmíněno dříve, při vypnutí funkcí contentEditablu a použití virtuálního domu nemůže vývojář spoléhat na propsání dat do dokumentu prohlížečem. Proto musí být odchyceny editorem a vloženy do modelu. Jelikož se o samotné vykreslování karetu však již stará prohlížeč, je potřeba dbát na to, že prohlížeč předpokládá pozici karetu na stejném místě jako je uložena v modelu. V případě že by pozice neodpovídaly, je zdrojem pravdy vždy datový model. V takovém případě by při zadávání znaků uživatel viděl nové znaky na jiném místě, než je samotný karet. Pozice karetu je v modelu určena identifikátorem fragmentu a offsetem, který značí, na kolikátém znaku daného fragmentu se nachází. Offset je indexem následujícího charakteru ve fragmentu. Pokud tedy umístíme ve slově „Škola“ karet mezi znaky „Š“ a „k“, získáme offset s hodnotou 1.

Záznam o pozici karetu rovněž obsahuje speciální nepovinný příznak „keepAtEndOfLine“. Pokud umístíme kurzor doprostřed paragrafu a nakonec řádku,

nemůžeme z modelu určit, zda se karet nachází na konci řádku, nebo na začátku následujícího řádku, jelikož fragment i offset jsou identické. Tato nejednoznačnost by pak znemožňovala vertikální pohyb pomocí kurzorových kláves a selekci textu. Proto se při posunu kurzoru na konec, či začátek řádku vytvoří příznak „keepAtEndOfLine“, který tuto informaci zahrne do modelu pro budoucí editaci.

Dalším důležitým parametrem uloženým v richEditorViewDef je informace o selekci. U selekce se musí řešit obdobné problémy jako u karetu. Tedy datová jednoznačnost mezi prohlížečem a datovým modelem. To vyplývá už z její datové definice. Selekce se skládá ze dvou parametrů – Anchor a Focus. Tyto dva parametry jsou, stejně jako samotný karet, datového typu EditorPosition. V případě selekce je ovšem nutné rozlišovat mezi pozicemi Anchor a Focus. Anchor je startovací pozice selekce a Focus je konečná. Pokud tedy uživatel kurzorem klikne na začátek věty a provede selekci na konec věty, bude na začátku věty Anchor a na jejím konci Focus. Pro samotné vykreslení selekce není tato informace potřebná. Je však nezbytná, pokud chceme provést dodatečnou úpravu selekce klávesovou zkratkou Shift + kurzorová šipka. Při této akci zůstává Anchor na stejné pozici a Focus se posouvá po jednom znaku. Pokud bychom selekci provedli obráceně, tedy začali selekci na konci věty a skončili na začátku, chceme stejnou zkratkou pohybovat Focusem na začátku věty.

3.4.3.2 Definice zobrazených objektů

RichEditorModelDef (Rich Editor Model Definition) obsahuje definice pro veškeré objekty, se kterými může Rich Text Editor pracovat. Základní jednotkou je zde datový typ Document. Document je jedním z parametrů typu View, který je popsán v kapitole 3.4.2.1 o richEditorViewDef. Document samotný nemá žádné další přídavné parametry. Teoreticky by mohl obsahovat informace o globálních nastaveních dokumentu, jako třeba barva pozadí, stránkování, zarážka a podobně. Tyto funkce však v současné verzi Rich Text Editoru nebyly vyžadovány. Proto obsahuje pouze kolekci objektů typu Block.

Block je defacto reprezentací jednoho odstavce dokumentu. Kromě jednotlivých fragmentů textu a dalších objektů obsahuje také ParagraphStyle. Jediným požadavkem pro stylování paragrafů bylo v době vývoje nastavování zarovnání. Zde je možné nastavit jedno ze sedmi typů zarovnání, které se používají v jazyku CSS pro stylování bloků. Konkrétně left, center, right, justifyLeft, justifyRight, justifyCenter a justifyBlock. Kromě

zarovnání se do budoucna počítalo rovněž s požadavkem na odrážkové a číselné seznamy, které by byly implementovány obdobným způsobem.

Základní jednotkou zobrazených dat v Rich Text Editoru je Fragment. Fragment sám o sobě nemůže existovat. Jedná se pouze o abstraktní třídu s volitelnými parametry urlLink, textStyle a fontFamily. Od této třídy dědí čtyři základní typy fragmentů. Prvním z nich je TextFragment, který zajišťuje ukládání textu. Dále SoftEnterFragment, který se vyvolá stisknutím klávesové zkratky SHIFT + ENTER, VariableFragment zobrazující textové proměnné a ImageFragment pro vykreslení obrázků.

3.4.3.3 Definice stylů

Pro datovou strukturu stylu je možné použít dva rozdílné přístupy. První možností je ukládání celých CSS definicí, které se poté aplikují na samotné HTML domy. Respektive fragmenty a paragrafy. Toto řešení je nejméně datově náročné a ve výsledku i jednoduché na implementaci. Z těchto důvodů může být ukládání čistých CSS definic použito u menších projektů bez rozsáhlejší funkcionality.

Pro Rich Text Editor by však tento přístup způsobil několik problému. Prvním problémem by bylo použití Generického editoru, který je detailněji popsán v kapitole o požadované funkcionalitě. Pokud by uživatel chtěl editovat svůj vytvořený styl pomocí generického editoru, namísto uživatelsky přívětivějšího Style Editoru, mohl by nastavit styl, který Rich Text Editor neumí vykreslit, popřípadě s ním neumí dále pracovat. Jelikož Generický editor nemůže pracovat s validačními pravidly, které jsou komplexnější než je kontrola typovosti, bylo by nutné validovat použité styly při načítání samotného Rich Text Editoru a jeho použitých stylů. Druhým důvodem, proč tento způsob ukládání pro Rich Text Editor není vhodný, je skutečnost, že je již použit DSL modeling, který umožňuje vývojáři přímou editaci stylů. Například pro uložení stylu „Bold“ by bylo nutné vytvořit mezivrstvu, která by na vybraný fragment uložila požadovaný CSS styl. Při použití DSL modelingu však stačí nastavit na stylu vybraného fragmentu „Bold“ na TRUE a provést transakci.

Z těchto důvodů byla vytvořena definice jazyka stylesEditorDef, která umožňuje nastavit pouze styly a jednotky na textové fragmenty a bloky, které Rich Text Editor podporuje. Hlavní Objektem jazyka stylů je TextStyleSet. TextStyleSet je datový typ, který může uživatel přiřadit každému dokumentu. V tomto objektu pak mohou být

definovány textové styly, paragrafové styly a fonty, které uživatel může aplikovat na text a paragrafy.

StylesEditorDef dále obsahuje definici TextStyle, která je již nejnižší jednotkou definice textového stylu. Je na něm možné nastavit booleanovské hodnoty pro styly bold, underline, italic, strikethough a fontSize definovaný typem number. fontSizeUnit lze nastavit typem fontSizeUnit na „px“ nebo „pt“. Barva je definována typem colorValue, který je definován v samostatné definici jazyka expressionDef. V typu paragraphStyle bylo podle dosavadních požadavků implementován pouze alignment.

3.4.3.4 Definice výrazů

Expression Definition Language je nejnižší úrovní modelovacího jazyka použitého v DX Builderu. Definuje základní datové typy, které využívají jeho předci. Konkrétně se jedná o datové typy ValueExpression, ConstantValue, AbstractStringValue, StringValue, ColorValue, DateTimeValue, BoolValue, EnumValue, ImageValue, NumberValue, VariableExpression, StructExpression, DataArrayExpression, StructField, BinaryOperator, naryOperator, BinaryExpression, UnaryExpression, TernaryExpression, NamedExpression, ReferenceExpression, GroupExpression, NaryExpression a Rule.

3.5 Použité technologie

3.5.1 Javascript

Javascript je programovací, nebo také skriptovací jazyk, který je široce využíván všemi rozšířenými webovými prohlížeči. (16) Javascript využívá implementace skriptovacího jazyku ECMA skript vyvíjený neziskovou společností ECMA International a komunitou ECMASkript community. Javascript je nyní využíván pro klientské neboli frontendové aplikace. V současné době je kromě frontendu využívána jeho obdoba Node.js rovněž na backendových aplikacích. ECMASkript je aktivně vyvíjen od roku 1996 po současnost. Výrazným skokem pro ECMASkript byla šestá edice ECMASkript 2015 ES6, která představila funkce známé z jiných programovacích jazyků, jako je deklarace tříd, použití modulů, arrow funkce, konstanty, promisy a pokročilé datové typy jako například binární data nebo nové kolekce. (17) V současné době se Javascript stabilně drží mezi pěti nejoblíbenějšími programovacími jazyky. (18)

3.5.2 Typescript

Hlavním problémem Javascriptu, který brání jeho použití ve velkých projektech, je absence striktní typovosti. Některé moderní frameworky řeší tento problém možností nadefinovat datové typy samostatnými objekty, u kterých následně kompilátor zkontroluje, zda posílané typy opravdu odpovídají navržené struktuře. Tento přístup je však již ze své podstaty vázán na daný framework. Z tohoto důvodu si vývojová prostředí s těmito typy neporadí a je potřeba doplňků třetích stran, které upozorní vývojáře na špatné použití typu ještě před kompilací kódu. (19)

S programovacím jazykem, který by řešil problém zastaralé syntaxe Javascriptu, se snažil přijít například projekt CoffeeScript nebo Dart od Googlu. V roce 2012 vydal Microsoft první verzi programovacího jazyka Typescript. Cílem Typescriptu bylo vytvořit programovací jazyk, který by umožnil vývojářům stejnou práci s typovostí třídami a dalšími funkcemi, které znali z aktuálně populárních programovacích jazyků jako C# nebo Java. Typescript sám o sobě není spustitelný kód. Pro jeho spuštění je nejprve nutná transkompilace do Javascriptu, kterou může následně spustit libovolný prohlížeč podporující Javascript. Díky široké vývojářské komunitě a pravidelné podpoře od Microsoftu, se Typescript nyní řadí mezi deset nejpopulárnějších programovacích jazyků. (20)

3.5.3 DSL Engineering

Spíše než technologií, je DSL Engineering metodikou, díky které byl vytvořen Modeling, na němž stojí celá datová struktura frontendové a backendové části Rich Text Editoru.

Termín programovací jazyk je obecně používán pro generické jazyky, jakými jsou například C# nebo Java. V terminologii DSL (Domain Specific Language) se programovacím jazykem nazývá právě nově vytvořený jazyk. Hlavním důvodem pro vytváření vlastních jazyků je abstrakce generického jazyka pro jednu danou konkrétní oblast. Díky této abstrakci je možné vytvořit jazyk, který umožní vývojářům rychlejší orientaci ve zdrojovém kódu. Rovněž psaní nového kódu se může značně urychlit využitím generických metod vytvořených na míru dané oblasti. Použití vlastního jazyka také umožňuje vývojářovi flexibilní přidávání a úpravy funkcionality jazyka. V neposlední řadě je možné DSL využít pro doménové experty, kteří mají menší

zkušenosti s generickými programovacími jazyky. Díky jazyku, který je specifický pro jejich zájmovou oblast, může expert rychle začít vytvářet vlastní kód, popřípadě se může usnadnit komunikace mezi vývojářem a samotným doménovým expertem.

Použití vlastního DSL má i své nevýhody. Jako největší problém je samotné vytvoření funkčního jazyka. Zatímco při použití generického jazyka může vývojář vytvořit „Hello world“ aplikaci během několika minut, za po použití vlastního doménového jazyka mu může podobný výstup zabrat i několik měsíců práce. Z tohoto důvodu je potřeba před vytvořením vlastního doménového jazyka zvážit finanční a časové náklady v poměru s ušetřeným budoucím časem. Ačkoliv byla zmíněn snadný onboarding pro nové nezkušené vývojáře, je nutné si uvědomit, že tento onboarding čeká i zkušené vývojáře, kteří s daným jazykem doposud nepřišli do styku. To může opět prodloužit vývoj a znesnadnit hledání nových vývojářů. (21)

```
export declare class MNode {
    readonly _concept: ConceptDef;
    readonly _id?: number;
    readonly _model?: IModel;
    readonly _propDef?: PropertyDef;
    readonly _parent?: MNode | ParentOfRootNode;
    readonly _previous?: MNode;
    readonly _next?: MNode;
    readonly _isInChildren: boolean;
    readonly _location: NodeLocation;
    readonly _isFree: boolean;
    readonly _isDeleted: boolean;
    readonly _isModelRoot: boolean;
    readonly _isRoot: boolean;
    getRoot(): MNode;
    getPropByDef(p: PropertyDef): NodePropertyValue;
    setPropByDef(p: PropertyDef, value: NodePropertyValue): void;
    getRawPropByDef(p: PropertyDef): NodePropertyValue;
    forEachDescendant(includeThis: boolean, callback: (child: MNode) => boolean | void):
boolean;
    insertAfter(insertionPoint: MNode): void;
    insertBefore(insertionPoint: MNode): void;
    replaceWith(node: MNode): void;
    detach(): this;
    clone(): this;
    checkReactive(rulesGroupId?: string): ICheckingMessage[] | undefined;
    analyze<Rest extends any[], A>(analysisType: INodeAnalysisType<Rest, A>,
...params: Rest): A | undefined;
}
```

Zdrojový kód 5 Ukázka základního nodu MNode (22)

3.5.4 Backend

Ačkoliv je Rich Text Editor frontendová komponenta, je nutné zmínit z důvodu úzké provázanosti i backendovou část. Backend je napsán kompletně v jazyce C# a frameworku .NET. Díky modelingu nebylo nutné vytvářet téměř žádné nové queries a requesty. Pro nové requesty byl použit nástroj Swagger, který automaticky generuje API funkce na frontendu. Po implementaci Swaggeru tedy odpadá ruční přepisování koncových nodů z backendu na frontend. Tyto přepisy jsou nejen časově náročné, ale zároveň hrozí zanesení chyb. Swagger je rovněž používán i ke generování API dokumentace, kterou mohou využívat samotní zákazníci, či jejich klienti. (23)

3.6 Požadovaná funkcionalita

V následující kapitole je popsána požadovaná funkcionalita Rich Text Editoru, která měla být součástí první verze produktu DX Builder. Tato funkcionalita je označována jako Minimum Viable Product (MVP). Účelem MVP je ukázat potencionálním zákazníkům základní funkčnost připravovaného produktu. V tomto případě DX Builderu. (24) Ačkoliv se u MVP nepředpokládá reálná funkčnost, ale spíše ukázka možností produktu, byla v případě Rich Text Editoru vyžadována jeho stoprocentní funkčnost. Bylo tak rozhodnuto z toho důvodu, že editor byl často součástí prezentace pro zákazníky. Pro tento účel byly stanoveny základní funkce editoru, které musí být implementovány a otestovány již v době vývoje.

3.6.1 Bold, Underline, Italic, Strikethrough

Základní funkcí textových editorů jsou stylovací příkazy Bold, Underline, Italic a Strikethrough. Stejně jako v běžných textových editorech i zde má uživatel na výběr dva způsoby, jak text nastylovat. První variantou je použití ikon na toolbaru editoru. Druhou možností je vyvolání akce stisknutím příslušných klávesových zkratek. Tyto zkratky jsou používány napříč produkty na různých operačních systémech. (25)

3.6.2 Velikost písma, font písma, barva písma

Velikost písma není na rozdíl od běžných textových editorů udávána v pixelech, nýbrž v jednotkách procent, kdy defaultní hodnota je sto procent. Uživatel si následně po výběru textu může nastavit velikost pomocí komponenty unit spinner krokovanou po

deseti procentech. Font písma je nastavitelný pomocí comboboxu, který uživateli po rozkliknutí vylistuje seznam aplikovatelných fontů. Seznam těchto fontů je uložen v definici stylů každého dokumentu. Pro výběr barev je použita již hotová komponenta Color Input. Po vybrání barvy v této komponentě se aplikuje daná barva na slovo s umístěným kurzorem nebo na vyselektovanou část textu.

Obrázek 1 Komponenta Color Input (15)

3.6.3 Obrázky

Vkládání a práce s obrázky byla plánovaná funkcionalita, která nestihla být v dosavadní verzi implementována. Nahrávání obrázků mělo probíhat skrze systémové assety DX Builderu. Po vložení obrázku do textového editoru měl mít uživatel k dispozici standartní možnosti základních úprav obrázků, jako je změna velikosti, změna poměru stran, překlopení či otočení. Z datového hlediska by měl být obrázek reprezentován datovým typem ImageFragment, který by dědil, stejně jako textový fragment od předka Fragment. To by umožňovalo jeho umístění mezi objekty parametru Fragments v jednotlivých blocích dokumentu.

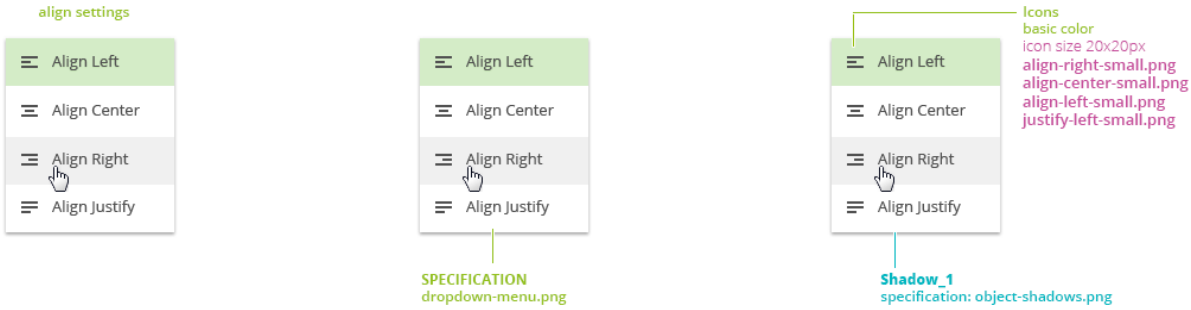
3.6.4 Proměnné

Proměnné byly stejně jako obrázky plánovanou a neuskutečněnou funkcionalitou. Přidaná hodnota pro uživatele při použití proměnných měla spočívat ve vložení speciálního fragmentu, který lze měnit napříč dokumentem, či více dokumenty. Například pokud by uživatel vytvářel textové labely pro nově vznikající aplikaci, u které doposud nebyl přesně stanovený název, popřípadě by existovala možnost jejího

budoucího přejmenování, mohl by namísto textu vložit proměnnou s dočasným názvem aplikace. U této proměnné by poté mohl skrze kontextové menu, nebo přes systémové assety DX builderu změnit její hodnotu napříč všemi jejími použitími. Datově by se jednalo, podobně jako u obrázku, o objekt VariableFragment, který by extendoval předka typu Fragment.

3.6.5 Stylování odstavců

Základním požadavkem pro stylování odstavců v Rich Text Editoru byla možnost nastavení jednoho ze sedmi blokových stylů. Konkrétně left, center, right, justifyLeft, justifyRight, justifyCenter a justifyBlock. Do budoucích verzí se plánovalo přidání možnosti nastylovat paragraf odrážkami s nastavitelnými symboly odrážek, či číslovanými paragrafy. Tato funkcionalita by umožňovala i vnořené odsázené bloky. Datově by to znamenalo přidání možnosti vložit do datového typu Block vedle Fragmentů i další vnořené Blocky.



Obrázek 2 Dropdown pro stylování odstavců (15)

3.6.6 Style editor

Style editor je nástroj, který není přímou součástí Rich Text Editoru, ale kromě generického editoru je jedinou možností vytvoření předkonfigurovaných CSS stylů. Samotný Style Editor umožňuje uživateli vytvoření vlastních textových a paragrafových stylů. V těchto stylech je možné nastavit pouze hodnoty, které Rich Text Editor podporuje. Tedy styly, které byly vyjmenovány v předchozích kapitolách. Hlavním důvodem pro použití Style editoru oproti sofistikovanějšímu CSS editoru, byla snaha umožnit vytváření stylů i uživatelům, kteří nemají žádné zkušenosti se psaním kaskádových stylů.

3.6.7 Předkonfigurované CSS styly

Aplikace předkonfigurovaných CSS stylů vychází z funkce style editoru. Po vytvoření textového stylu se uživateli zobrazí v hlavním toolbaru Rich Text Editoru seznam aplikovatelných stylů. Smyslem předkonfigurovaných stylů, je obdobně jako u proměnných, vytvoření stylu napříč dokumentem, či více editory a umožnit tak hromadnou editaci použitého stylu. Tato funkcionalita je vhodná například při rebrandingu aplikace. Pokud uživatel vytvoří části textu psané aplikační barvou, je možné ji při změně firemních barev jednoduše změnit ve Style Editoru a aplikovat na všechny doposud vytvořené texty. Další možností využití je vytváření šablon pro mobilní aplikace. Pokud by zákazník vytvářel aplikace se stejným základem pro různé zákazníky, může za pomocí referencovaných stylů a proměnných jednoduše vytvořit personalizovanou aplikaci na míru zákazníka z již vytvořené šablony.

Práce s těmito styly je obdobná jako v aplikaci MS Word. Po selekci textu, nebo po umístění karetu doprostřed slova, vybere uživatel jeden z předkonfigurovaných stylů. Po kliknutí na styl se aplikuje vybraný styl na všechny vybrané znaky, popřípadě slovo. Pokud byl na vybraných fragmentech již aplikovaný jiný styl, ať už předkonfigurovaný, či ručně vytvořený, budou všechny předešlé styly vymazány a nahrazeny vybraným stylem. Pro uživatele může nastat matoucí situace, pokud bude chtít aplikovat ruční stylovaní na již přiřazený referencovaný fragment. Například pokud si text QUADIENT nastaví na referencovaný styl „Company color“, který má pouze nastavenou specifickou barvu a následně nastyluje pomocí klávesové zkratky CTRL + B celou větu tučně. V takovém případě se referencovaný styl zruší jeho vlastnosti se aplikují na daný textový fragment. Následně je ke stylu fragmentu přidáno i tučné zvýraznění. Uživatel tak může předpokládat, že pokud změní barvu v referencovaném stylu „Company Color“, bude změněna i v upraveném textu, což již nemůže nastat, jelikož fragment ztratil informace o svém původním stylu.

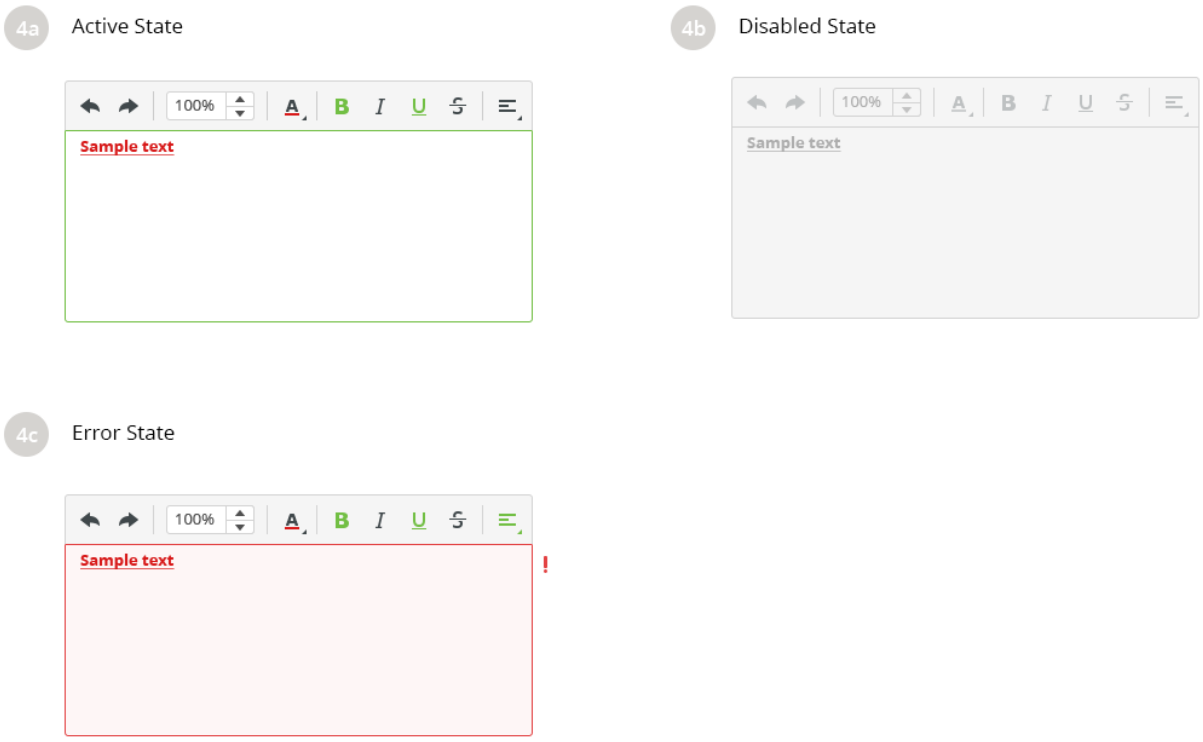
3.6.8 Generický editor

Generický editor je funkce určená pro pokročilé uživatele a vývojáře DX Builderu. Není možné ho otevřít pomocí grafického rozhraní aplikace. Namísto toho je možné ho vyvolat klávesovou zkratkou CTRL + ALT + SHIFT + E. Po stisknutí této zkratky se uživateli zobrazí editor aktuálního datového modelu. Generický editor respektuje datové typy modelingu. Z tohoto důvodu je možné nastavit jednotlivým parametrům pouze

hodnoty odpovídající jejich datovým typům. Hlavní motivací pro vytvoření Generického editoru je možnost změny atributů objektů, které nejsou možné pomocí grafického prostředí. Ačkoliv Generický editor před uložením kontroluje konzistenci dat, je stále možné uvést datový model do stavu, se kterým Rich Text Editor neumí pracovat. Proto je nutné, aby byl využíván pouze zkušenými uživateli.

3.6.9 Stavy Editoru

Rich Text Editor může nabývat tří různých stavů. Konkrétně Active, Disabled a Error Sate. V Active stavu je editor plně funkční. Uživatel s ním může plně interagovat a nejsou detekovány žádné chyby. V Disabled statu jsou všechny uživatelské vstupy vypnuty, včetně zadávání textu. Do tohoto stavu se uživatel může dostat, pokud je Rich Text Editor otevřený v modálním okně a uživatel využívá jiné části DX Builderu, popřípadě jinou instanci Rich Text Editoru. Třetím možným stavem je Error State. Do tohoto stavu by se za běžných okolností uživatel neměl dostat. Teoretickou možností, jak tento stav vyvolat, je vložení nebo nastavení nevalidních dat do modelu pomocí Generického editoru. Všechny tyto stavy jsou exkluzivní. Tudíž není možné, aby byl editor zároveň v disabled statu a error statu.



Obrázek 3 Ukázka stavů Rich Text Editoru (15)

3.6.10 Konkurenční editování

Konkurenční editování, známé také jako online nebo živá kolaborace, je možnost upravovat dokument společně s jinými uživateli a zároveň vidět v reálném čase jejich úpravy. Tato funkcionalita je použita například v Google Documents od společnosti Google nebo Office 365 od společnosti Microsoft.

Možnost pracovat na projektu v DX Builderu zároveň s ostatními uživateli, mělo být jedním z hlavních marketingových taháků. Konkurenční editování se mělo najít využití při vzdáleném školení nových pracovníků s DX Builderem a zároveň pro snazší kolaboraci při práci více osob na stejném projektu. Z těchto důvodů vznikl požadavek na konkurenční editování i v samotném Rich Text Editoru.

3.7 User Experience a Grafika

Velká část plánování projektu DX Builder byla věnována uživatelskému prožitku neboli user experience. Cílem tohoto oboru je nabídnout uživateli při používání aplikace co nejpříjemnější zážitek. Do této definice je však zahrnuta i samotná použitelnost

celého produktu. Pravidlem pro user experience designera by mělo být naplnění všech sedmi následujících bodů: (26)

- **Užitečnost.** Produkt musí přinášet uživateli přidanou hodnotu z vložené práce.
- **Použitelnost.** Uživatel by měl být schopen používat produkt, který se chová podle pravidel, podle nichž byl vytvořen a k účelu k němuž byl určen.
- **Žádanost.** Ačkoliv jsou použitelnost a žádanost hlavní kritérii pro user experience, rozhodně nejsou dostatečné. Zejména v silně konkurenčním prostředí, kdy je snadné vytvořit produkt s obdobnou funkcionalitou, je nutné vytvořit produkt, který je uživatelem přímo žádaný nejen na racionální, ale také na emoční úrovni.
- **Nalezitelnost.** Uživatelské prostředí musí být uživateli schopno uživatele rychle a intuitivně navést k hledanému objektu, popřípadě nabídnout nápovědu, která mu v tom pomůže.
- **Přístupnost.** Aplikace musí být přístupné pro všechny druhy uživatelů. V tomto ohledu je potřeba brát ohled na individuální způsoby používání produktu. Nejedná se však pouze a využívanou funkcionalitu, ale například i způsob konzumace obsahu. V případě webových stránek je potřeba zajistit responsibilitu na různých velikostech monitorů, popřípadě přizpůsobit uživatelské prostředí mobilním zařízením a dotykovým obrazovkám obecně. V neposlední řadě je také nutné zajistit dostatečnou přístupnost osobám se zrakovým, sluchovým, či pohybovým omezením. K tomuto účelu lze využít režimy vysokého kontrastu, navigace hlasem a mnohé další.
- **Kredibilita.** Kredibilita, nebo také důvěryhodnost, je klíčovým prvkem u aplikací, kterým uživatel svěřuje své citlivé údaje, popřípadě peníze. Kredibilita produktu není určena pouze user experience designem, ale také celkovým dojmem, kterým na venek působí autorská společnost a provozovatel.
- **Hodnota.** Finální produkt musí být použitelný a funkční do takové míry, aby uživateli přinášel žádaný užitek. Kromě finančního zisku se může

jednat rovněž o zábavu, či vyšší cíl v případě, že je produkt využíván neziskovými organizacemi.

3.7.1 Unifikovaný design

Napříč všemi produkty společnosti Quadient, je nyní používán jednotný grafický styl nazvaný „Green Stripe“. Tento grafický styl byl použitý i v projektu DX Builder a jeho základní prvky byly implementovány i do designu komponenty Rich Text Editor.

V oblasti vývoje softwaru můžeme nalézt tři hlavní důvody proč vyvíjet a aktivně aplikovat unifikovaný design: (27) (28)

- **Jednotné ovládání.** Vytvořit jednotné ovládací vzory je nutné zejména pokud uživatel využívá široké aplikační portfolio jedné společnosti. Uživateli tak stačí, aby si osvojil ovládání jedné části aplikace, a díky tomu je mu umožněn snazší přechod do jiných aplikačních částí, popřípadě do jiného produktu. V produktech společnosti Quadient to mohou být například postranní vysouvací okna či rozbalovací položky, které se vyskytují napříč všemi aplikacemi. U Rich Text Editoru je například znovupoužita komponenta Color Picker, se kterou se uživatel již mohl setkat v jiných částech DX Builderu.
- **Jednotný vzhled.** Hlavním důvodem, proč je nutné napříč aplikacemi udržovat jednotný vzhled, je zvýšení žádanosti aplikace. Díky tomu, že má uživatel k dispozici napříč aplikací, či aplikačním portfoliem stejné ovládací prvky, barvy a vizuál, zvyšuje se jeho spokojenost při používání aplikace. I v případě, že navštíví novou aplikační část, bude moci nalézt známe prvky, které u něj utlumí strach z neznámého a lépe si tak osvojí novou funkcionalitu. Příkladem může být Material Design od společnosti Google (29). Material Design se nejprve objevil poprvé v roce 2014 na platformě Android jako nástupce Holo Designu. Následně se rozšířil i do webových služeb jako Gmail, Youtube, Google Maps a dalších. Díky tomuto kroku může nyní Google nabídnout konzistentní uživatelské prostředí napříč všemi svými produkty a usnadnit adopci nových produktů pro stávající uživatele. Nyní je Material Design uvolněný jako open source a volně přístupný komunitě. (30)

- **Znovupoužitelnost.** Poslední výhodou unifikovaného designu je možnost znovupoužití komponent. Díky znovupoužívání komponent, je možné ušetřit až 80 % nákladů na tvorbu frontendu. Jelikož je použita komponenta, která je již implementována a široce používána, může si být vývojář být jistý její funkčnosti a prověřeností. Další výhodou je i snadné testování a přidání případných úprav. Pokud je nutné například změnit podobu komponenty Button, stačí úpravu provést pouze na jednom místě v kódu. Po aktualizace balíčku se změny projeví na všech místech, kde je komponenta použita. (31)

3.7.2 Persóny

V oblasti User Experience se často vyskytují takzvané „Persóny“. Jedná se o fiktivní osoby, které mají reprezentovat typy zákazníků, kteří budou s reálným produktem pracovat. U menších produktů, které nevyžadují například administrátorské role a u většiny funkcí se neliší způsob používání napříč uživatelskou demografií, může stačit jedna pečlivě vytvořená persóna. U větších projektů je vhodné vytvořit persón více. Tyto persóny tak mohou reprezentovat například různé pracovní pozice uživatelů, způsob, jakým aplikaci využívají a případně i jaký hardware používají ke své práci.

Při vytváření persón je nutné dbát na jejich realističnost a opírat se o skutečná data. Podle provedeného průzkumu, (32) pouze 5 % společností provedlo reálný výzkum, aby vytvořili podobu persón pro své projekty. Zbylých 95 % pouze odhadovalo podobu konečných zákazníků na základě interních domněnek.

Hlavním účelem persón je poskytnout vývojářům reálnou představu o podobě koncového zákazníka, jeho schopnostech, potřebách a zkušenostech. Kromě těchto informací mohou persóny usnadnit komunikaci napříč týmy. Namísto termínů jako „Document Administrator“, může být využito jméno persóny, díky čemuž následně vývojáři mohou získat k persónám bližší vztah. (33)

Pro projekt DX Builder byly vytvořeny dvě persóny. Konkrétně se jedná o Rebeccu a Kena.

- **Rebecca Flowers.** Rebece je 35 let, žije v Belgickém Ghentu a má pětileté zkušenosti s prací v oboru. Pracuje jako Document

Administrator v pojišťovací společnosti. Její práce s Rich Text Editorem v aplikaci DX Builder spočívá v editaci textu již vytvořených mobilních aplikací, popřípadě aplikovaní nových referencovaných textových stylů. Rebecca má středně vysoké zkušenosti s prací s informačními technologiemi, střední zkušenosti s Microsoft Office a základní zkušenosti s produktem DX Builder. Rebecca ke své práci využívá operační systém Windows na Full HD (1920x1080p) monitoru o velikosti minimálně 22 palců.

- **Ken Jansen.** Kenovi je 35 let, žije v Belgickém Ghentu a má tříleté zkušenosti s prací v oboru. Pracuje jako aplikační manager v pojišťovací společnosti. Jeho prací je vytváření nových aplikací a aplikačních v produktu DX Builder. Jeho úkol v Rich Text Editoru je, kromě zadávání defaultních textů do nových aplikací, vytváření nových textových referencovaných stylů pomocí CSS Editoru. Ken je středně pokročilý uživatel DX Builderu, má střední zkušenosti s Microsoft Office a má pokročilé zkušenosti s prací s informačními technologiemi. Ken ke své práci využívá operační systém Windows dvou na Full HD (1920x1080p) monitorech o velikosti minimálně 22 palců. Ken má střední zkušenosti s prací v operačních systému Linux.

Obrázek 4 Persóny Rebecca a Ken (15)

3.7.3 Konvence chování napříč textovými editory

Při vytváření Rich Text Editoru se objevilo několik případů, kdy nebylo z pohledu User Experience určit jednoznačné chování, které uživatel očekává. V takových případech bylo rozhodnuto o převzetí chování z textových editorů Word od Microsoftu a Docs od společnosti Google. Avšak ani v těchto editorech není vždy chování jednotné a není možné jednoznačně určit, jaké chování uživatel po provedené akci očekává.

V následujícím seznamu jsou rozebrány základní případy zadávání, výběru a stylování textu, na které je nutné se při vývoji vlastního textového editoru zaměřit. Toto chování je vhodné brát v potaz již při prvotní fázi návrhu editoru, jelikož snaha o pozdější implementaci by mohla mít za následek nutnost změny datové struktury a celkového konceptu práce s textem.

- **Aplikace stylu, pokud je uprostřed slova karet.** V případě, kdy uživatel umístí doprostřed slova karet a aplikuje textový styl, například psaní tučným textem klávesou zkratkou CTRL + B, může uživatel očekávat dva výstupy. V případě Google Docs a obecně ContentEditablu, se okamžitě neprojeví žádná změna a slovo zůstane nenastylované. Pokud uživatel začne poté zadávat další znaky, budou vkládány již nastylované tučným stylem. Google Docs uživatele upozorní o zapnutém stylu „aktivací“ ikonky Bold v hlavním toolbaru. Jakákoliv změna pozice karetu toto nastavení resetuje a při zadání dalšího znaku uživatel opět uvidí nenastylovaný text. Pokud uživatel stejnou akci provede v aplikaci Microsoft Word, dojde k nastylování celého slova. Za povšimnutí stojí, částečné ignorování speciálních znaku ve Wordu. Pokud je na konci slova například tečka, bude i ta po aplikaci stylu nastylována. Pokud je však slovo, nebo lépe textový fragment tvořen znaky jako jsou tečky, čárky, pomlčky, lomítka nebo podtržítka a uživatel chce nastylovat slovo umístěním karetu mezi tyto znaky a následnou aplikací stylu, zachová se Word stejně, jako mezi běžnými mezerami. Tedy aktivuje se psaní vybraným stylem, které opět resetuje při posunu karetu. V Rich Text Editoru bylo adoptováno řešení z aplikace Microsoft Word. Tedy stylování celého slova po aplikaci stylu. Speciální znaky uprostřed slov se ignorují. Jako separátor mezi slovy je použita pouze mezera, či nový odstavec.

- **Zapamatování si předchozí pozice karetu.** V tomto případě se jedná o nenápadnou funkci, která je jednotná u všech textových editorů a contentEditable ji obstarává sám. Nicméně pokud používáme vykreslování přes virtuální DOMy, je nutné ji implementovat zvlášť. Představme si následující situaci. V odstavci má každý řádek přibližně sto znaků a poslední řádek má padesát znaků. Následující odstavec má na prvním řádku pouze dvacet znaků. Pokud umístíme karet na konec padesátiznakového řádku, můžeme se kurzorovými šipkami pohybovat nahoru a dolů. Pokud se však takto dostaneme na konec dvacetiznakového řádku, očekává uživatel, že se při pohybu nahoru opět vrátí na padesátý znak. Je tedy nutné uložit index sloupce, na kterém byl karet naposled umístěn myší, nebo se do něj dostal horizontálním pohybem karetu. Poté je nutné udržovat karet stéle na tom samém indexu, pokud to počet znaků na řádku dovolí.
- **Postupná selekce slov.** Pokud chceme vybrat část textu pomocí selekce za použití myši namísto kurzorových kláves, dostaneme opět jiné chování ve Wordu a Google Docs. Při výběru textu ve Wordu probíhá výběr po celých slovech. Rovněž se krátce výběru několika prvních znaků označí i celé první slovo, na kterém je umístěna kotva karetu. Označena je i mezera na konci tohoto slova. Výjimku tvoří případy, kdy je na konci slova některý ze speciálních znaků, které byly zmínění v části o aplikaci stylu do slov s karetem. Za zmínku rovněž stojí možnost toto chování „vypnout“. Uživatel toho docílí tak, že pokud vybírá text zleva doprava, vrátí se kurzorem s aktivní selekcí zpět před kotvu selekce a následně zpět na původní pozici. Tímto Word vypne selekci celých slov a uživatel může provádět přesnější selekci na úrovni jednotlivých znaků. Oproti tomu Google Docs žádnou tuto funkcionalitu nenabízí. V reálu tedy pouze kopíruje chování selekce v komponentě ContentEditable, která při selekci vybírá pouze jednotlivé znaky bez ohledu na mezery v textu.
- **Smazání nastylovaného textu pomocí selekce.** Dalším nekonzistentním chováním v oblasti stylování je odmazání nastylované části textu pomocí selekce a následné opětovné zadávání znaků. Pokud například uprostřed slova vyselektujeme jeden znak, kterému nastavíme styl Bold, stiskneme

DELETE a začneme psát, dostaneme různý výsledek v Google Docs a Wordu. Ve Wordu jsou stisknutím klávesy DELETE zapomenuty předchozí styly, které byly na hranici s karety. Naopak v Google Docs si editor stále pamatuje styl, na kterém byla umístěna kotva karetu a při zadání nového znaku je text opět nastylován smazaným stylem. Tato informace o smazaném stylu však není uložena datově mezi textovými fragmenty a při posunu karetu je zcela zapomenuta.

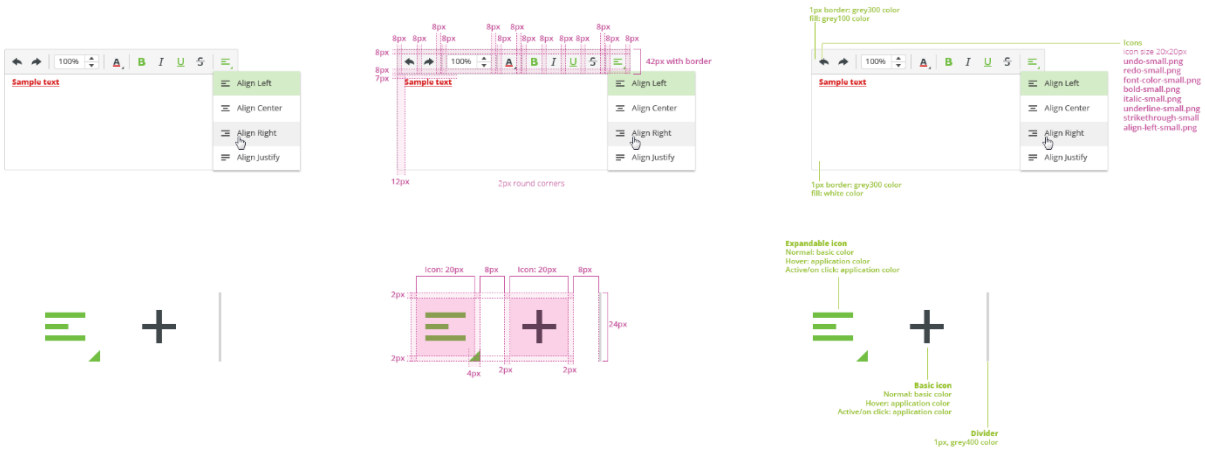
- **Selekce mezery na konci slova.** Funkce, která již byla zmíněna v části o posupném selektování slov. Pokud uživatel za pomocí dvojitého kliknutí vyselektuje slovo v textu, vybere se i mezera na jeho konci. Tato funkce je užitečná, pokud chce uživatel slovo smazat. Eliminuje se tak možnost dvojitých mezer mezi slovy. Problém však nastane, pokud uživatel slovo nechce smazat, ale nahradit jiným. V takovém v případě dvojitým kliknutím vybere slovo a začne ihned psát slovo nové. Za těchto okolností se opět liší chování Google Docs a Wordu. Google Docs po stisknutí prvního nového znaku vymaže vyselektované slovo i s mezerou a nový znak je umístěn přímo na prvním znaku následujícího slova bez dělící mezery. Oproti tomu Word pozná, že slovo bylo vybráno dvojitým kliknutím a při zadání dalšího znaku automaticky vloží před následující slovo mezeru. Tato funkce se opět automaticky vypne, pokud uživatel provede selekci s přehozením pozice karetů, jak bylo popsáno v odstavci o postupné selekci slov.
- **Přepínání stylů uvnitř selekce.** Další případ, kdy uživatel může dostat jiný výsledek, než očekává je aplikace stylu pomocí selekce, pokud je žádaný styl na textu již částečně aplikován. Uživatel může mít například větu, kdy je první půlka textu tučně zvýrazněná a druhá půlka je bez zvýraznění. Poté uživatel vybere celou větu pomocí selekce a pomocí zkratky CTRL + B nebo pomocí toolbaru aplikuje styl Bold. Jestliže je celý vybraný text již nastylovaný, styl se z výběru odebere. Pokud je text nenastylovaný, styl se naopak přidá. V našem případě, kdy styl není aplikován na celou větu, se chování opět liší v Google Docs a v Microsoft Word. Google Docs v tomto případě opět adoptuje chování komponenty contentEditable. Algoritmus projde všechny znaky v selekci, a pokud najde

jediný, který neobsahuje aplikovaný styl, provede přidání stylu na všechny nenastylované znaky. Ve Wordu se toto chování řídí podle prvního znaku selekce. Pokud je tedy první část věty nastylovaná a druhá část bez stylu, Word nastaví na veškerý text selekce opačný styl, než je na prvním karetu. Při implementaci této funkce je nutné si uvědomit, že Word opravdu bere znak na **prvním** karetu. Nikoliv na anchoru. Pokud tedy uživatel provede selekci zprava doleva, je nutné zjistit, který s karetů je první.

3.7.4 Grafický design

Podle Martina T. Peciny je účelem grafického designu co nejvíce vyhovět účelu daného produktu a usnadnit s ním práci. Kromě funkčního aspektu, který je spíše výhradou spíše user experience, má graficky design zejména estetickou funkci, která uživatelovi subjektivně zpříjemňuje práci s aplikací. (34)

Ve společnosti Quadient se o návrh grafického designu stará grafický designer, jehož úloha je popsána v kapitole o složení týmu.



Obrázek 5 Ukázka části grafické specifikace Rich Text Editoru (15)

4 Proces vývoje

V následující kapitole je popsán samotný proces vývoje Rich Text Editoru z pohledu managementu. Dále je zde rozebrán způsob, jakým byla aplikace testována a zajišťována kvalita.

4.1 Složení týmu

Samotný vývojový tým se skládal ze šesti osob. Konkrétně ze tří vývojářů, dvou QA (quality assurance) a jednoho team leadera. Další lidé, kteří se na vývoji přímo podíleli, ale nebyli součástí týmu, byli grafičtí designéři, user experience designéři a product owner. Každý z těchto skupin osob měla následující úkoly.

4.1.1 Vývojář

Vývojář (DEV) se stará o implementaci na funkcionality na nejnižší úrovni kódu. Jeho úkolem je převzít user story, která popisuje požadovanou přidanou hodnotu pro koncového uživatele a implementovat ji do aplikace. Jeho povinností je rovněž pokrytí veškeré logiky unitovými testy, oprava rozbitých unit testů a případně oprava rozbitých funkcionálních testů. Po dokončení funkcionality předává vývojář hotovou user story svému QA.

4.1.2 Quality assurance

Quality assurance, QA, tester nebo také „voňavka“ je osoba, která zajišťuje funkčnost a kvalitu hotové user story v podobě, ve které byla popsána product ownerem, či team leaderem. Po manuálním otestování user story je sepsán testovací scénář, ve kterém jsou projity všechny klíčové a kritické součásti této funkcionality. Pro tyto scénáře jsou následně vytvořeny funkční testy, které mohou být buď grafické, nebo API testy. Pokud je user story z pohledu quality assurance bez problému, je označena jako implemented a předána product ownerovi k finálnímu schválení, popřípadě user experience designerovi k UX checku.

4.1.3 Team leader

Team leader je zodpovědný za přerozdělování úkolů daného requierementu. Práce team leadera je spíše administrativního rázu a na samotném vývoji se podílí

minimálně, což je zároveň jeho úkol. Tj. naučit tým pracovat samostatně s žádnými nebo minimálními zásahy.

4.1.4 User experience designer

Jednou z nejdůležitějších pozic při vývoji nového softwaru, ač mnohdy značně opomíjená, je user experience designer (UX). Jeho úkolem je na základě požadavků product ownera na novou funkcionalitu vymyslet uživatelsky přívětivý způsob, jak daného cíle dosáhnout. Jednoduše řečeno, práce UX je navrhnout software tak, aby byl uživatel schopný jej používat i bez návodu. Jeho úkolem však není pouze designování jednotlivých dílčích funkcionalit, ale zároveň mít přehled o celkovém fungování aplikace a snažit se sjednotit fungování všech jeho částí. Pokud společnost vyvíjí více produktů, je vhodné, UX designer navrhnul jednotné chování napříč celým produktovým portfoliem firmy.

4.1.5 Graphic designer

Práce grafického designéra je často zaměňována s prací user experience designéra. Ačkoliv je jejich práce velice úzce spjata, jejich úkoly jsou jasně rozděleny. Zatímco UX designér se zaměřuje na způsob ovládání, grafický designér má za úkol vytvořit grafikou podobu produktu. To zahrnuje například ikony, paddingy, styl písma, barvy a další.

Po vytvoření user story product ownerem a UX designerem přijde na řadu grafik, který vytvoří „pixel-perfect“ specifikaci zadané funkcionality. V této specifikaci musí být vypsány přesné hodnoty barev, paddingů, marginů, názvy ikon a názvy použitých komponent, pokud jsou k dispozici. Rovněž vytváří veškeré ikony a obrázky.

4.1.6 Product owner

Product owner je zřejmě nejdůležitější osoba v životním cyklu celého requirementu. Jeho úkolem je přebírání vize produktu od vyššího vedení a jejich transformace na jednotlivé user story za pomoci team leadera. Úzce spolupracuje s UX, kterému tlumočí tyto požadavky a snaží se najít nejvhodnější způsob implementace z hlediska koncového uživatele. Product owner také vytváří celé requirementy, které později přiděluje po konzultaci s team leaderem jednotlivým týmům. Dalším jeho úkolem je zajistit, aby všechny týmy dodaly zadanou funkcionalitu do konce

čtyřměsíčního cyklu releasovacího období a tím doručit zákazníkovi včas novou verzi systému.

4.2 Testování a Quality Assurance

Komponenta Rich Text Editor je z pohledu stylování textu a práce s proměnnými, obrázky a odstavci aplikací, ve které se vyskytuje velké množství možných kombinací funkcí a objektů v datové struktuře. Z tohoto důvodu je komponenta náchylná k výskytu chyb vycházející z její komplexity. Jelikož jsou standardy společnosti Quadient zaměřeny zejména na kvalitu dodávaného softwaru, byly i u komponenty Rich Text Editor aplikovány všechny postupy testování a zajištění kvality jako u ostatních produktů.

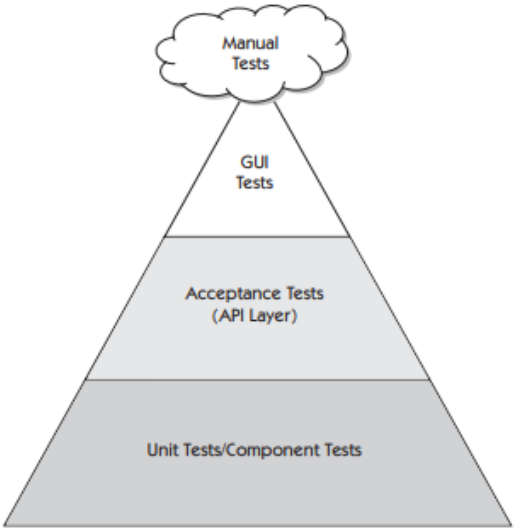
V současné době se již opouští od zažitého modelu, kdy tester pouze „proklikává“ aplikaci a hledá chyby. Prací testera je všeobecné zajištění kvality softwaru po všech stránkách. Poslední trendy rovněž směřují k tomu, aby quality assurance spíše předcházeli vzniku chyb nastavením přesně daných postupů při vývoji, než aby je pouze hledali. (35)

Podle Rona Pattona hovoříme o softwarové chybě, pokud je splněna alespoň jedna z následujících podmínek: (36)

- Software nedělá něco, co by podle specifikace produktu dělat měl.
- Software dělá něco, co by podle údajů specifikace produktu dělat neměl.
- Software dělá něco, o čem se produktová specifikace nezmiňuje.
- Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
- Software je obtížné srozumitelný, těžko se s ním pracuje, je pomalý, nebo – podle názoru testera softwaru – jej koncový uživatel nebude považovat za správný.

V oblasti automatických testů se využívá termín Testovací Pyramida. Testovací Pyramida zobrazuje tři, respektive čtyři základní druhy testů a jejich poměrové zastoupení při vývoji aplikace. Jedná se o unit testy, které by měly být nejpočetnější a pokrývat veškerý testovatelný kód, poté integrační testy zaměřené na práci s API, a nakonec GUI testy. Na vrcholku testovací pyramidy bývají někdy zobrazeny Manuální Testy. Ty již ze své podstaty nespadají mezi automatické testy, nicméně v některých

případech, kdy není možné klíčovou funkcionalitu pokrýt automatickými testy, jsou stále nepostradatelné.



Obrázek 6 Testovací pyramida (35)

4.2.1 Unit testy

Unit testy jsou nejzákladnější jednotkou automatických testů. Vznikly na samém počátku komplexního programování a dnes jsou základním stavebním kamenem moderních programovacích jazyků a platforem. Jejich největšími výhodami je jejich rychlost a přesnost. Přesností je myšleno zaměření na jednu konkrétní funkci a abstrakce ostatní navázané funkcionality. (37)

Cílem unit testu je na základě vstupních parametrů funkce a vstupního stavu navázaných tříd, určit správnost výstupu, či konečný stav aplikace. Důležitou myšlenkou unit testů je testování pouze kódu obsaženého v testované metodě. Pokud tedy funkce využívá logiku například z metod jiné třídy, je nutné tuto třídu namockovat a dodat funkcionalitu z této třídy bez ohledu na její skutečnou implementaci. Výhodou tohoto přístupu je možnost rychlého nalezení problému v kódu, kdy začne vykazovat chybu pouze jeden test, který testuje chybnou metodu. V opačném případě by začaly hlásit chyby unit testy všech metod, které využívají logiku dané metody. (38)

Následující zdrojový kód je ukázkou unit testu pro komponentu Rich Text Editor za pomocí testovacího frameworku Jasmine. Jednotlivé testy mohou být tříděny a seskupování pomocí funkce „describe“, která rovněž umožňuje provádění dodatečných

funkcí před, nebo po každém unit testu v daném describu. V ukázkovém testu máme na nejvyšší úrovní describe s názvem Styling, jelikož testujeme stejnojmennou třídu. Déle chceme testovat skupinu funkcí, které se starají o stylování fragmentů s textovým stylem. Součástí tohoto describu, je funkce beforeEach (39), která před každým spuštěním vnořeného testu inicializuje nové objekty „fragment“ a „textStyle“. Poslední describe sdružuje všechny testy, které testují funkci metody removeCssPropertyFromFragment. V tomto případě je vybrán test, který testuje odebrání stylu „strike“ z fragmentu, který na sobě má již aplikované styly „strike“ a „underline“. První část testu nastaví tyto dva požadované styly na hodnotu TRUE a přiradí styl k textovému fragmentu. V druhé části dojde k provolání metody removeCssPropertyFromFragment s požadovanými parametry. Na konci testu je styl fragmentu převeden na textový CSS styl a testuje se, že na fragmentu zůstala pouze textDecoration typu underline. Tímto se otestovalo, že kromě odebrání požadovaného stylu, zůstane na fragmentu i druhotný styl underline.

```
describe("Styling", () => {
  describe("removeCssPropertyFromFragment", () => {
    let fragment: Fragment;
    let textStyle: TextStyle;

    beforeEach(() => {
      fragment = remInst.TextFragment.create();
      textStyle = createTestTextStyle();
    });

    describe("Remove property from textStyle", () => {
      it("remove strike from fragment with textStyle with strike and underline", () => {
        //Given
        textStyle.underline = true;
        textStyle.strikeThrough = true;
        fragment.textStyle = textStyle;
        //When

        styling.removeCssPropertyFromFragment(fragment, createStrikeProperty());

        //Then
        let fragmentStyle = fragment.getStyle();
        expect(fragmentStyle["textDecoration"]).toBe("underline");
      });
    });
  });
});
```

Zdrojový kód 6 Ukázka unit testu (15)

4.2.2 Integrační testy

Integrační testy na rozdíl od unit testů testují větší část aplikační logiky napříč vícero funkcemi, či třídami. Jsou stále o jednu vrstvu níže než grafické testy, tudíž se nemusí vypořádávat s nestabilitou uživatelského rozhraní. Jejich funkcí je otestovat API a spojení dané aplikace. Hlavním cílem, na rozdíl od unit testů, není ukázat vývojáři kde se nachází chyba, ale spíše na chybu upozornit. (37)

Ačkoliv integrační testy našly své široké využití napříč aplikací DX Builder, nebyly nijak využity v komponentě Rich Text Editor. Jak již bylo dříve zmíněno, Rich Text Editor využívá ke komunikaci s backendem doménový jazyk Modeling. Backendová implementace modelingu již obsahuje unitové a integrační testy. Další integrační testy zaměřené na využití modelingu Rich Text Editorem, by znamenalo redundantní testování již otestované funkcionality.

4.2.3 GUI testy

Nedílnou součástí testování frontendové komponenty jsou GUI testy. Tento druh automatických testů simuluje reálné chování komponenty tak, jak by ji viděl uživatel ve webovém prohlížeči. K tomuto účelu je využíván například framework Selenium nebo Pupeteer od společnosti Google. (40)

Hlavním problémem GUI testů je jejich údržba. Ačkoliv dokáží velice spolehlivě odhalit chybu v aplikaci, jsou náchylné na hlášení false positive. Tedy na takové případy, kdy aplikace funguje podle očekávání, ale test přes to vyhodnotí chování jako chybné. Malá změna v aplikační logice, či pouze v uživatelském rozhraní může rozbít několik zdánlivě nesouvisejících testů. Z tohoto důvodu se GUI testy nacházejí na vrcholu testovací pyramidy. Jejich počet by měl být redukován na nezbytné minimum a měly by testovat pouze základní funkcionalitu aplikace, kterou nelze otestovat integračními a unit testy. (35)

Ačkoliv je logika Rich Text Editoru silně pokryta unit testy, některé funkce je třeba stále pokrýt GUI testy. Jde zejména o funkcionalitu, která je navázána na HTML element contentEditable. Její chování se může lišit napříč prohlížeči a může se chovat nekonzistentně i v závislosti na čase z důvodu vydání nové verze vykreslovacího jádra prohlížeče.

4.2.4 Manuální Testy

Nejmenší podíl na všech testech Rich Text Editoru měly manuální testy. Ruční testování mělo za cíl projít pouze nejzákladnější funkcionalitu editoru, jako je psaní, selekce a aplikace stylů. Při Manuálních testech je nejdůležitější vizuální kontrola komponenty testerem a odhalení potencionálních chyb, které automatické testy neodhalily.

4.3 Řízení práce

V projektech velikosti DX Builderu je nutné práci efektivně definovat, rozdělit a následně kontrolovat. V následujících odstavcích jsou popsány základní manažerské frameworky, nástroje a termíny, které byly při vývoji DX Builderu a Rich Text Editoru využívány.

4.3.1 Scrum

Scrum je metodika pro řízení vývoje, dodávání a údržbu komplexních softwarových produktů, či jiných produktů. Tento framework je používán k vývoji aplikací od devadesátých let. Nabízí účinné manažerské postupy, které umožňují průběžné vylepšování produktu a pracovního prostředí. Scrum se skládá z několika základních jednotek, jako jsou týmy, role, události a artefakty. Tyto jednotky mají každá svůj účel a jsou nezbytné k dosažení úspěchu Scrumu. (41)

Při vývoji aplikace Rich Text Editor, byly aplikovány veškeré události, jak je Scrum definuje. Vývoj byl rozdělen na dvoutýdenní sprinty. Před začátkem každého sprintu probíhal grooming, kde byla vývojáři odhadnuta náročnost plánovaných user storek. Každé user story byl poté po vzájemné domluvě přiřazen story point na v hodnotách 1, 2, 3, 5 nebo 8. Následoval planning, kde byly user story přiřazeny vývojářům a k nim jeden quality assurance. Každý den sprintu se ráno konal Daily Scrum (Standup), kde každý člen týmu řekl, na čem aktuálně pracuje, v jakém je stádiu, popřípadě co ho blokuje v práci. Na konci dvoutýdenního sprintu následoval Sprint Review, kde každý tým prezentoval svoji dokončenou práci ostatním vývojářským týmům. Následující týden se uskutečňovala Sprintová Retrospktiva, kde, kde členové týmu přednášely své pozitivní poznatku k uplynulému sprintu, popřípadě náměty na zlepšení.

Ve společnosti QUADIENT jsou Sprinty obaleny do releasovacích cyklů, které probíhají jednou za tři měsíce. Před koncem releasu nastává QA období, kdy se do aktuální verze již nepřidává nová funkcionalita a vývojářská kapacita je soustředěna na opravování a hledání případných chyb.

4.3.2 Youtrack

Pro trackování veškeré aktivity spojené s User Stories Requirementy a Bugy byl využíván software od společnosti Jetbrains – Youtrack. Youtrack nabízí také možnost vytváření Dashboardů a Reportů, které umožňují managementu lépe analyzovat proces vývoje a najít možné chyby v řízení práce. (42) Pro samotného vývojáře slouží Youtrack zejména k získání informací o podobě user story, na které pracuje, evidenci Bugů a vykazování času.

4.3.3 User Story

Základní jednotkou práce ve Scrumu je user story. User story popisuje funkcionalitu, která po implementaci přinese reálnou hodnotu uživateli. User story jsou rozděleny na tři části.

- Psaný, nebo grafický popis user story, který dává vývojářům potřebné informace o její finální podobě.
- Diskuze o detailech user story a způsobech její implementace, ať už z pohledu funkčního, či technického.
- Postup testování, díky kterému quality assurance určí, zda byla user story správně implementována.

Popis user story by se měl držet následujícího formátu:

Jako <název role>, za účelem <čeho chce dosáhnout>, potřebuji <způsob jakým kterým toho chce dosáhnout> (43)

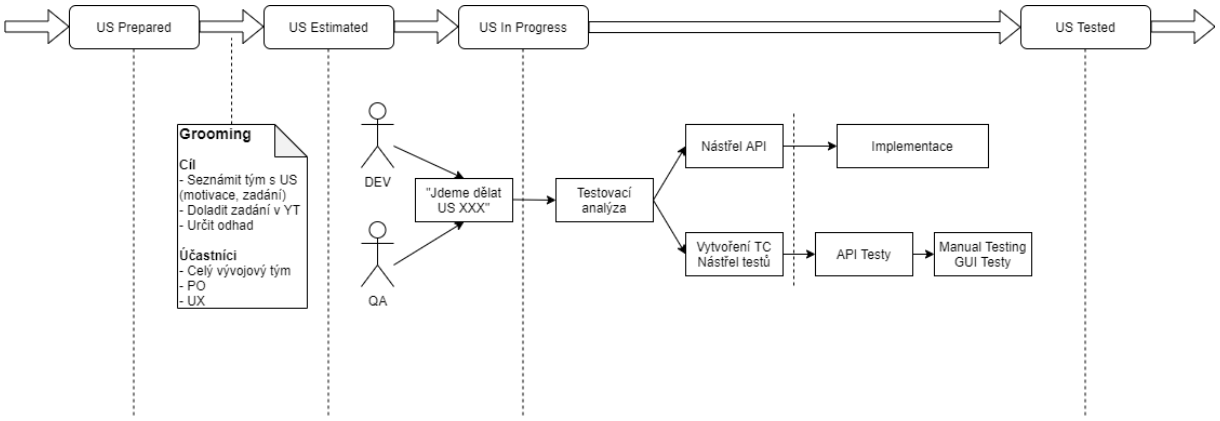
Příkladem popisu user story tedy může být: „As a Document Administrator, in order to change text size, I need unit spinner that will apply font size to selected text“. Diskuze by měla obsahovat informace od User Experience, které například popisuje chování, pokud chceme zvětšit velikost slova o 10% a zároveň je slovo již nastylováno vícero velikostmi. Je rovněž vhodné, aby vývojář během implementace dopisoval

poznámky pro quality assurance o tom, které části funkcionality je potřeba otestovat a kde by se mohla potencionálně nacházet chyba.

V produktu DX Builder a dalších aplikacích společnosti Quadient, je aplikován následující životní cyklus user story:

- **Sepsání user story.** Po definici requierementu následuje rozpad na jednotlivé user story. Velikost user story by neměla přesahovat dílku jednoho sprintu a zároveň by neměla být příliš malá, z důvodu fixní administrativní zátěže na každou user story. Ideální délka je tedy tři až pět dní čisté práce. Po rozpadu requirementu, sepíše product owner, popřípadě Team Leader podrobný popis user story, doplní případnou grafickou dokumentaci a vloží do backlogu.
- **Grooming.** Cílem groomingu je seznámení týmu s danou User Story a motivací k jejímu vytvoření. Je konzultována náročnost na implementaci a testování a na jejím základě je odhadnuta náročnost user story. Groomingu by se měl zúčastnit celý vývojový tým včetně quality assurance. V případě vyšší složitosti User Story je přizván user experience designer, popřípadě product owner.
- **Planning.** Před začátkem každého nového sprintu probíhá planning, kde je User Story přidělena vývojářovi a jednomu quality assurance, který ji bude po implementaci testovat.
- **Vývoj.** Samotný proces vývoje závisí již pouze na vývojáři a quality assurance. Vývojář by měl poskytnout potřebné informace o plánové podobě API, ideálně ihned ze začátku poskytnout ve vývojové branchi nefunkční API, které poskytnou quality assurance základ pro vytvoření automatických testů. Po implementaci User Story proběhne kompletní testování funkcionality a napsání unitových, API, popřípadě grafických testů.
- **Uzavření User Story.** Pokud quality assurance shledá implementaci odpovídající popisu User Story, následuje UX check a PO check, kdy user experience designer a product owner shlédnou finální podobu user story a popřípadě ji vrátí k dalším úpravám. Pokud se tak nestane, je vývojová branch přimergována do aktuální master branche. quality assurance

naposledy zkontroluje, že je User Story funkční i na master branchi, a nastaví stav v Youtracku na „US tested“. Tím je životní cyklus user story ukončen.



Obrázek 7 Životní cyklus User Story (15)

5 Problémy a jejich řešení

Během vývoje Rich Text Editoru bylo objeveno několik problémů, které vyžadovaly změnu architektury komponenty, popřípadě další neplánovaný vývojářský čas pro jejich řešení. V následující kapitole jsou popsány ty nejzávažnější problémy a možnosti jejich řešení.

5.1 Konkurenční editování

Jak již bylo zmíněno v kapitole o požadované funkcionalitě, pro Rich Text Editor bylo vyžadována možnost konkurenčního editování, respektive návrh architektury takovým způsobem, aby bylo možné jej implementovat v budoucnu bez nutnosti přepisu infrastruktury. Díky použití Modelingu se tato funkce částečně sama implementovala. Pokud uživatel změní text v Rich Text Editoru, instantně se odešle informace o změně na backend. Tato komunikace však funguje oboustranně. Pokud backend zaznamená změnu modelu, odešle notifikaci o změně včetně změněných dat na ostatní aktivní instance DX Builderu. Díky tomuto bylo možné prakticky ihned po implementaci vkládání textu navázaného na Modeling začít používat konkurenční editování. Problém nastává, pokud oba uživatelé chtějí editovat text ve stejném fragmentu. Pokud editují rozdílné fragmenty, Modeling odesílá na backend pouze identifikátory upravovaných fragmentů a nové textové hodnoty. Stejným principem pak backend posílá informace zpět ostatním instancím. Pokud by dva uživatelé upravili jeden fragment, byl by jeden požadavek přepsán druhým a byla by aplikována změna uživatele, jehož požadavek by přišel později.

Tento problém by byl vyřešen, pokud by Modeling podporoval změnové vektory. V takovém případě by se při zadání znaku do textového fragmentu odeslal pouze nově napsaný znak a jeho pozice s identifikátorem fragmentu. Obdobným způsobem probíhá zapisování dat do vnitřního modelu Rich Text Editoru. Pokud by bylo nutné konkurenční editování implementovat bez podpory vektorových změn v Modelingu, bylo by nutné ručně napsat requesty na backend které by přijímaly změnové vektory a překládaly je na změny celých objektů. Tímto by bylo možné vytvořit frontu, která by principem LIFO postupně odbavovala všechny příchozí změny. Tím by se však prakticky zahodily všechny výhody, které Modeling nabízí a byla by vytvořena pouze přechodná funkcionalita, která by později byla podruhé implementována přímo do Modelingu.

5.2 Ukládání Historie

Jedním z požadavků pro Rich Text Editor byla podpora historie. Tedy možnost, kdy se uživatel po stisknutí klávesy CTRL + Z vrátí do předchozího stavu editoru. Tuto funkci opět defaultně obstarává samotný ContentEditable, nicméně kvůli použití virtuálního domu, je nutné ukládání a aplikování historie implementovat ručně. Díky použitému Modelingu, je možné k tomuto problému přistupovat dvěma způsoby.

Prvním a implementačně nejsnadnějším způsobem, je prosté revertnutí poslední provolané transakce do Modelingu. Každá akce, která mění datový model Rich Text Editoru je obalena funkcí **undoableAction**. Tato funkce se kromě lokálního uložení dat do modelingu stará o odeslání dat na backend. Všechny tyto akce jsou logovány a je tedy možné se vrátit do libovolného předchozího stavu aplikace. Problém však nastává, pokud bychom chtěli uživateli umožnit revertování historie po celých slovech, či celých větách. Jeden krok zpět by tak mohl vyžadovat zpětné načtení až pro desítky transakcí, což by mělo vážný dopad na performance komponenty.

Druhým přístupem je ukládání psaného textu do průběžného zásobníku. Text v tomto zásobníku by se necommitoval po každém napsaném znaku, ale například po každém slovu, větě či určité době nečinnosti. Tento přístup se však vylučuje s požadavkem na konkurenční editování. Druhý uživatel by v takovém případě neviděl psaný text prvního uživatele v reálném čase, například pouze po dopsání každé věty, či slova.

V poslední verzi Rich Text Editoru nebyla požadována funkce vracení historie po celých slovech. Byla tedy zvolena první varianta, kdy se uživatel vrací zpět po každém zadaném znaku, popřípadě po každé provedené stylovací akci. Možným řešením tohoto problému by do budoucna byla možnost tagování commitnutých transakcí v modelingu. Editor by si držel informace o provedených akcích a identifikátorech jejích transakcí. V případě kroku zpět by pak modelingu předal identifikátor transakce, na kterou se chce vrátit a ten by se následně již automaticky vrátil do požadovaného stavu libovolný počet transakcí zpět.

5.3 Kopírování textu a vkládání textu

Při vlastní implementací funkce Copy-Paste je při použití virtuálních domů opět nutné použít vlastní řešení. Moderní webové prohlížeče však mají bezpečnostní opatření, které tuto funkcionalitu značně omezují. Relativně bezproblémovou akcí je kopírování textu z editoru. Pokud chceme kopírovat nastylovaný text z HTML elementu ContentEditable, vloží prohlížeč do schránky přesnou podobu textu v podobě HTML DOMů. To umožňuje nastylovaný text opět vložit do druhého ContentEditable, či jiného editoru, který si převede HTML obsah na datový formát, který umí zobrazit. Toto zkopírování prostého HTML není možné v Rich Text Editoru, neboť text je tvořen mnohými pomocnými DIVy a dalšími objekty, které umožňují práci s textem. Z tohoto důvodu je nutné zachytit, zda uživatel stiskl zkratku CTRL + C a nahradit vlastní akcí. V této vlastní akci si Rich Text Editor převede aktuálně vybrané fragmenty na čisté HTML, a to následně pomocí funkce **execCommand** vloží do schránky. S takovýmto obsahem schránky pak mohou libovolně pracovat ostatní textové editory. (44)

Komplikovanější částí je vkládání textu. Pokud by měl Javascript plný přístup ke schránce uživatele, mohl by se každý web dostat k jejímu obsahu. Mohl by tak potencionálně získat zkopírovaná hesla, popřípadě další citlivé informace. Z tohoto důvodu je tato funkce zablokovaná a je možné ji zpřístupnit pouze přidáním povolení clipboardRead do prohlížeče. Jediná možnost, jak tedy vložit obsah schránky na webovou stránku, je stisknutím zkratky CTRL + C během focusu na textový input, popřípadě na contentEditable. Přímé vkládání textu však není možné použít, jelikož v případě Rich Text Editoru je nutné text nejprve vložit do vnitřního datového modelu. Pro vkládání textu byl tedy použit následující workaround. Pokud uživatel uvnitř editoru stiskne zkratku CTRL + V, nebude odchycena jako v případě kopírování, ale editor nechá prohlížeč vložit obsah schránku přímo do textového pole. Ihned poté editor vložený text rozpozná, uloží do vlastní paměti a odstraní z místa původního vložení. Poté převede vložený HTML formát na podobu jednotlivých textových fragmentů a vloží ho zpět na místo karetu.

Neřešitelným problémem se také ukázalo vkládání textu přes kontextové menu. Napříč celým DX Builderem dostane uživatel po pravém kliknutí myši kontextové menu s funkcemi pro danou aplikaci. Toto chování muselo být z důvodu konzistence zachováno i v Rich Text Editoru. Po vyvolání kontextového menu na vybraném textu měl

mít tedy uživatel možnost vložit a kopírovat text. Kopírování textu je možné jednoduše pomocí metody **execCommand.** Nicméně jelikož je možné vkládání textu vyvolat pouze přes zkratku CTRL + V, byla v contextovém menu položka „Paste“ disablovaná a doplněna tooltipem, který vyzývá uživatele k použití zkratky CTRL + V.

5.4 Změna a smazání použitého referencovaného stylu

Problém, který si vyžádal poměrně velké množství vývojářského času, je změna a mazání aplikovaných referencovaných stylů. Ačkoliv to odporuje filozofii používání referencovaných objektů, uživatel může chtít ponechat původní podobu již aplikovaného stylu. Z tohoto důvodu bylo u referencovaných stylů zavedeno verzování. V případě změny již aplikovaného stylu, je uživateli zobrazen seznam dokumentů a textů, kde je tento styl již použit a nabídnuto, zda chce v daném textu styl aktualizovat, či ponechat původní.

Obdobnou situací je případ, kdy se uživatel rozhodne styl, který je již aplikovaný zcela smazat. Pro takové případy byla zavedena dvoustupňová úroveň mazání objektů. První úrovní je takzvaný Soft Delete. Pokud se uživatel rozhodne smazat referencovaný styl, je opět upozorněn na jeho výskyty, jako v případě editace. Po jeho smazání je styl stále uložen v systému a je zobrazen na všech použitých textech. Není však možné ho již znovu použít. Druhou úrovní smazání je Hard Delete. Po něm již nezůstává žádný záznam v databázi modelingu a styl zmizí ze všech textových fragmentů, na kterých byl aplikován.

5.5 Stejná barva textu a pozadí

Poněkud atypickým problémem se kterým se ostatní textové editory nepotýkají, je stejná barva textu a pozadí editoru. U běžných textových editorů, jako je Microsoft Word, či Google Docs, si uživatel může zvolit jak barvu písma, tak barvu textu. Uživatel tedy vidí finální podobu dokumentu, který vytváří. V tomto ohledu je účel Rich Text Editoru odlišný. Principem Rich Text Editoru je vytvořit text, který je následně zasazen do jiné komponenty pomocí Component Editoru. Tato komponenta má však vlastní barvu pozadí, či obrázek. Může se tedy stát, že pokud si uživatel chce nadesignovat aplikaci ve tmavém schématu, bude si vytvářet v Rich Text Editoru bílé texty, které nejsou vidět na bílém pozadí editoru.

Pro tento případ byl dodatečně do Rich Text Editoru přidán přepínač barvy pozadí textové oblasti. Největší výzvou pro tuto funkci nebyla její samotná implementace, ale řešení z pohledu User Experience, neboť funkce, která přepíná barvu pozadí, evokuje v uživateli pocit, že vybrané pozadí bude i ve finálním textu. Z tohoto důvodu nebyl přidán klasický Color Picker, ale pouze možnost přepnout mezi dvěma barvami. Bílou a tmavě šedou.

5.6 Selekce textu

Zdaleka nejnáročnější problém v celém vývoji Rich Text Editoru představovala selekce textu. Jak již bylo mnohokrát zmiňováno, Rich Text Editor využívá HTML element contentEditable pro práci s textem. Z důvodu použití virtuálních DOMů a nutnosti synchronizace s podobou aktuálního datového modelu s reálným obsahem contentEditablu, bylo nutné jeho mnohé funkce vypnout a nahradit vlastní implementací. Toto se týkalo i implementace selekce textu. V případě, že uživatel provedl selekci textu, uvidí v editoru klasické modré pozadí vybraného textu. Tato informace musí být uložena i v datovém modelu komponenty, aby bylo umožněno kopírování a mazání vybraného textu.

K implementaci selekce lze přistupovat dvěma způsoby. Prvním z nich úplná deaktivace selekce textu prohlížečem. V takovém případě by veškeré vykreslování selekce převzal Javascript. Bylo by potřeba brát ohled na obarvování vybraného textu a případné kolize barvy selekce s barvou pozadí. Rovněž každý prohlížeč používá jiné barvy a vizualizaci selekce. Pokud bychom chtěli zachovat chování, na které je uživatel zvyklý, bylo by nutné implementovat individuálně každý grafický styl selekce pro každý prohlížeč zvlášť. Obzvláště náročné by poté bylo zprovoznění selekce v mobilních prohlížečích uzpůsobených pro dotykové ovládání.

Druhou cestou pro práci se selekcí při použití virtuálních DOMů je využití zobrazení selekce, kterou nabízí vykreslovací jádro prohlížeče. V takovém případě se nabízí, aby prohlížeč na základě pohybů myší vytvořil a vykreslil selekci, kterou komponenta převezme a převede do svého datového modelu. Bohužel se ukázalo, že prohlížeče nemají konzistentní chování při provolávání eventů, jako jsou **selectstart**, **selectend**, **selectionchange** a dalších, které jsou využívány při změně selekce. Z tohoto důvodu bylo od těchto eventů kompletně opuštěno a bylo rozhodnuto použít opačný

postup pro práci s nativní selekcí textu prohlížeče. Editor tedy nezískává informace o již provedené selekci z prohlížeče, ale na základě uživatelských vstupů zjistí, kde má být selekce umístěna a pomocí metody **addRange** vytvoří selekci na místě, kde ji uživatel očekává. (45) S tímto přístupem však přichází poslední problém, kterým je detekce pozice kurzoru nad textem. Zjistit, nad kterým elementem se aktuálně nachází kurzor lze poměrně jednoduše za pomocí Javascriptových eventů **mousenter** a **mouseover**. Problém nastává, pokud má text na jednom řádku rozdílnou velikost. V takovém případě text s největším fontem sahá až k elementu na hořejším řádku, ale vzniká mezera nad textem s nižším fontem. K této mezeře se opět různé prohlížeče chovají odlišně. Někdy je při umístění kurzoru na toto prázdné místo označen jako aktuální element text pod kurzorem, což je očekávané chování. V jiných případech byl však jako aktuální hoverovaný element označen celý contentEditable, což znemožňovalo určit, kde se aktuálně uživatel nachází. V takovém případě bylo nutné přistoupit na určení pozice kurzoru pomocí binárního dělení, kdy byl editor rozdělen na jednotlivé sektory a následně kontroloval, na kterých částech se kurzor aktuálně nachází.

6 Doporučení pro vývoj vlastního Rich Text Editoru

Od zadání práce našemu týmu na Rich Text Editoru až po první commit uplynul více než měsíc. Tolik času zabrala přípravná fáze, kdy se hledaly nejlepší možná řešení architektury editoru. I přes to se po téměř ročním vývoji objevily pochyby o správnosti návrhu a procesu vývoje.

Největší pochybnosti ohledně správnosti plánovací fáze byly v případě použití HTML elementu **contentEditable**. Použití tohoto elementu může vývojáři velice usnadnit práci. Drtivou většinu akcí, jako je základní stylování, zadávání znaků, kopírování textu a selekce zvládne obstarat zcela sám. Problémy se začnou objevovat, pokud chceme tyto změny zároveň propagovat do vlastního modelu. Téměř všechny funkce contentEditable musely být nakonec vypnuty a nahrazeny vlastním řešením. Další problémy nastaly při řešení kompatibility s různými prohlížeči, kdy musela být z velké části nahrazena i samotná selekce textu. Pokud by byl tedy opět vytvářen Rich Text Editor se stejnými požadavky, pravděpodobně by bylo přistoupeno k implementaci zcela vlastního řešení. Tímto by odpadla fáze testování na více prohlížečích a oprava chyb, která na vývoji zabrala značnou část času a peněz. Jelikož byla většina funkci základního contentEditable již zakázána a nahrazena, bylo by prakticky nutné pouze vytvořit vlastní selekci a zobrazení karetu. Ačkoliv se o zpětném odstranění contentEditable během vývoje neuvažovalo, objevil se návrh na vlastní implementaci selekce. I když byla selekce v poslední verzi Rich Text Editoru zcela funkční, objevily se obavy o pracnosti implementace dalších prvků, jako jsou obrázky, emoji a proměnné, kvůli kterým by se opět muselo opravovat fungování selekce contentEditablu. Pokud tedy chce vývojář vytvořit editor se základní funkcionalitou, **je vhodné použít element contentEditable pro malé a středně velké projekty**. Pro projekty, které budou obsahovat komplexní funkcionalitu, obrázky a vlastní datový model, **je z dlouhodobého hlediska výhodnější vytvoření vlastního řešení namísto contentEditable.**

Další doporučení, týkající se datové struktury, editoru, je **nepoužívat pomocné flagy a jiné značky**. K používání této praktiky často svádí vidina rychlejší implementace funkcionality, či oprava chyby. Může to být například aplikace Boldu na konci slova, které se nijak neprojeví v textu, ale je potřeba ho uchovat v paměti, pokud uživatel začne zadávat znaky. Tyto informace nesmí být uloženy v datovém modelu dokumentu, ale pouze v dočasné paměti editoru.

Zřejmě nejdůležitějším bodem, který se nesmí zanedbat při vývoji Rich Text Editoru je testovací infrastruktura. Jelikož se jedná o frontendovou komponentu, je pochopitelné, že nebude dodržena podoba testovací pyramidy a nebudou se zde nacházet téměř žádné API testy. O to více je však **nutné se zaměřit na unitové a GUI testy**. Zejména rozsáhlá struktura unitových testů, do kterých bylo značně investováno, umožnila hladší pozdější vývoj. Obzvláště pak v momentech, kdy se nevyhnutelně muselo přistoupit ke změně datového modelu, nebo obecného chování editoru. Tyto testy však zajišťují pouze správnou podobu vnitřního datového modelu. Pokud se při vývoji použije contentEditable, jako v našem případě, je rovněž nutné mít k dispozici velké množství GUI testů, které budou kontrolovat vizuální podobu a stavbu HTML DOMů v prohlížeči.

7 Závěr

Na konci roku 2017 vznikl requirement na vytvoření textového editoru pro projekt DX Builder, který umožní uživatelům základní editaci a stylování textu komponent. V druhé polovině roku 2018 byla prvním potencionálním zákazníkům prezentována první Alpha verze DX Builderu s již funkční komponentou Rich Text Editor. V té době zvládal editor všechny základní funkce potřebné pro editaci a stylování textu. Jmenovitě zadávání a mazání znaků, aplikace stylů bold, strikethrough, italic a underline, úpravu velikosti písma, změna barvy písma, aplikace vlastních fontů, stylování odstavců, kopírování, vkládání a selekce textu. Přidanou hodnotu přinášela práce s referencovanými styly ve spolupráci se Style Editorem.

Z důvodu komplexity a návaznosti na technologie DX Builderu, bylo rozhodnuto při vývoji nepoužívat již hotová řešení, ať už interní, či třetích stran, ale vytvořit od základu vlastní editor. Jako Javascriptový framework byl použit Bobril, který kromě rychlosti měl tu výhodu, že s ním již byli vývojáři seznámeni a umožňoval snadnou integraci do projektu. K uchovávání dat se využila stromová struktura za pomoci interního doménového jazyka Modeling.

Vzhledem k velkému množství stavů, kterých může Rich Text Editor nabývat, bylo nezbytné pokrýt celou komponentu testovací infrastrukturou. K tomuto účelu posloužil framework Selenium, který zajišťoval grafické funkční testy a Jasmine, s jehož pomocí byly vytváření unitové testy. V týmu, který na projektu pracoval, se pracovalo šest lidí. Jeden Team Leader, dva quality assurance a tři programátoři včetně mě samého. Mimo tým se dále na vývoji podíleli grafici a User Experience Designéři. Celý vývoj byl řízen pomocí metodiky SCRUM.

Po dokončení základní funkcionality Rich Text Editoru, pokračoval autorský tým na dalších requirementech DX Builderu. Začátkem roku 2019 byl však projekt DX Builder ukončen a s ním i vývoj Rich Text Editoru. Důvodem k zastavení vývoje byl rozdílný směr, kterým se vyvíjel trh v oblasti Business to Customer a digitalizace dokumentů, než se očekávalo při prvotním plánování v letech 2015 až 2016. Do současné doby, se neobjevil projekt, který by již vytvořený textový editor požadoval a byl schopen implementovat. Vzhledem k návaznosti datové struktury editoru na Modeling, není ani jeho znuvupoužitelnost bez nutnosti rozsáhlého přepisu možná. Jeho

opětovné využití by vyžadovalo, aby daný projekt rovněž využíval Modeling, což v současnosti žádný prodkut ve společnosti QUADIENT nesplňuje. Ačkoliv byl celý projekt DX Builder ukončen, je zde stále možnost, opětovného využití technologií, které byly pro jeho potřebu vyvinuty. Pokud to nebude samotný Rich Text Editor, najde s nejvyšší pravděpodobností uplatnění nově vytvořený modelovací jazyk Modeling i s veškerou dodatečnou funkcionalitou, která u něj byla pro splnění požadavků Rich Text Editoru vyvinuta. Nejcennějším výstupem celého projektu jsou však zkušenosti. Ačkoliv byly všechny požadavky na vývoj splněny na výbornou, byl přes to projekt ukončen. Na celém DX Builderu pracovalo za celou dobu jeho trvání celkem dvanáct týmů, kteří za bezmála tři roky vývoje získali neocenitelné zkušenosti a znalosti, které mohou využít v dalších projektech.

Projekt DX Builder spolu s Rich Text Editorem jsou v současné době zakonzervovány a v případě obnovení projektu připraveny na opětovné zprovoznění.

8 Přílohy

8.1 Prohlášení o použití interních dokumentů společnosti QUADIENT s.r.o

Společnost QUADIENT s.r.o. souhlasí s publikováním diplomové práce **Analýza procesu vývoje komponenty Rich Text Editoru** včetně interních grafických specifikací a částí zdrojového kódu. Konkrétně:

- Část grafické specifikace komponenty Color Input (Obrázek 1)
- Části grafické specifikace komponenty Rich Text Editor (Obrázek 2, Obrázek 3, Obrázek 4)
- Zdrojové kódy komponenty Rich Text Editor (Zdrojový kód 3, Zdrojový kód 4, Zdrojový kód 5, Zdrojový kód 6)

V Hradci Králové dne 6.8.2020

Bc. Jan Růža
Vice President Research and Development

Příloha 1 Prohlášení o použití interních dokumentů

9 Bibliografie

1. **Steigerwald, Daniel.** ReactiveConf 2019 - Daniel Steigerwald: Making a New ContentEditable Editor. *Youtube.* [Online] 7. 11 2019. [Citace: 5. 3 2020.] https://www.youtube.com/watch?v=9zfEGSVlqgE.

2. **Shabrikov, Mikhail.** Draft.js — rich text editor framework for React from Facebook. *Medium.* [Online] 11. 7 2018. [Citace: 8. 8 2020.] https://medium.com/@mshabrikov/draft-js-rich-text-editor-framework-for-react-from-facebook-f236d02576f0.

3. Introduction - Slate. *Slate.* [Online] [Citace: 8. 8 2020.] https://docs.slatejs.org/.

4. **Quill.** Quill Rich Text Editor. *Why Quill.* [Online] [Citace: 8. 8 2020.] https://quilljs.com/guides/why-quill/.

5. **Andras, Samuel.** JavaScript Frameworks, why and when to use them. *hello.js.* [Online] 14. 02 2017. [Citace: 4. 1 2020.] https://blog.hellojs.org/javascript-frameworks-why-and-when-to-use-them-43af33d0608d.

6. **Garcia, Tim.** React Hooks! - An Introduction. *Rithm School.* [Online] 10. 22 2019. [Citace: 5. 3 2020.] https://www.rithmschool.com/blog/react-hooks-intro.

7. Angular Introduction: What It Is, and Why You Should Use It. *SitePoint.* [Online] 22. 3 2018. [Citace: 6. 5 2020.] https://www.sitepoint.com/angular-introduction/.

8. **Jorgé.** React vs. Svelte, the JavaScript build-time framework. *React, etc. Tech Stack.* [Online] 30. 11 2016. [Citace: 6. 5 2020.] https://react-etc.net/entry/react-vs-svelte-the-javascript-build-time-framework.

9. **Letocha, Boris.** Bobril - otázky a odpovědi. *Microsoft Stream.* [Online] 1. 4 2020. [Citace: 2. 3 2020.] https://web.microsoftstream.com/video/de248162-7824-41fb-8dec-50f7361d6896.

10. JavaScript-free frontends with Blazor. *LogRocket.* [Online] 15. 6 2020. [Citace: 1. 7 2020.] https://blog.logrocket.com/js-free-frontends-blazor/.

11. **Sychra, Tomáš.** Bobril. *Bobril.* [Online] 2019. [Citace: 3. 3 2020.] https://bobril.com/.

12. **Mozilla.** Making content editable. *MDN Web Docs - Mozilla.* [Online] Mozilla, 24. 2 2020. [Citace: 6. 5 2020.] https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Editable_content.

13. **Pilgrim, Mark.** The Road to HTML 5: contentEditable. *The WHATWG Blog.* [Online] 6. 3 2009. [Citace: 3. 2 2020.] https://blog.whatwg.org/the-road-to-html-5-contenteditable.

14. **Walsh, Norman.** A Technical Introduction to XML. *XML.com.* [Online] 3. 8 1998. [Citace: 19. 5 2020.] https://www.xml.com/pub/a/98/10/guide0.html.

15. **QUADIENT s.r.o.** QUADIENT. 2019.

16. **W3Techs.** Usage statistics of JavaScript as client-side programming language on websites. *W3Techs.* [Online] [Citace: 19. 6 2020.] https://w3techs.com/technologies/details/cp-javascript/.

17. **ECMA International.** ECMAScript® 2021 Language Specification. *TC39.* [Online] ECMA International, 9. 10 2019. [Citace: 6. 19 2020.] https://tc39.es/ecma262/.

18. **Carbonnelle, Pierre.** PYPL PopularitY of Programming Language. *PYPL.* [Online] 2020. [Citace: 19. 6 2020.] http://pypl.github.io/PYPL.html.

19. **Nnamdi, Chidume.** Understanding React PropTypes - Type-Checking in React. *Bits and Pieces.* [Online] 5. 2 2019. [Citace: 20. 6 2020.] https://blog.bitsrc.io/understanding-react-proptypes-type-checking-in-react-9648a62ce12e.

20. **Bright, Peter.** Microsoft TypeScript: the JavaScript we need, or a solution looking for a problem? *ARS Technica.* [Online] 10. 3 2012. [Citace: 20. 6 2020.] https://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/.

21. **Voelter, Markus.** *DSL Engineering: Designing.* místo neznámé : Createspace Independent Publishing Platform, 2013. 978-1481218580.

22. **Privitzer, Pavol.** 2017.

23. **SmartBear.** API Design. *Swagger.io.* [Online] SmartBear Software, 08. 05 2020. [Citace: 24. 6 2020.] https://swagger.io/solutions/api-design/.

24. **Ambler, Scott.** Project Management.com. *Defining MVP, MMF, MMP, and MMR.* [Online] 27. 12 2017. [Citace: 6. 24 2020.] https://www.projectmanagement.com/blog-post/61937/Defining-MVP--MMF--MMP--and-MMR.

25. **Microsoft.** Microsoft Support. *Keyboard shortcuts in Word.* [Online] Microsoft , 6. 23 2020. https://support.microsoft.com/en-us/office/keyboard-shortcuts-in-word-95ef89dd-7142-4b50-afb2-f762f663ceb2?ui=en-us&rs=en-us&ad=us.

26. **Morville, Peter.** User Experience Design. *Semantic Studios.* [Online] 21. 6 2004. [Citace: 7. 7 2020.] http://semanticstudios.com/user_experience_design/.

27. **White, Alex W.** Unified Design: Forcing elements to relate makes a design powerful. *The Alexander W. White Consultancy.* [Online] 16. 8 2011. [Citace: 7. 7 2020.] https://alexanderwwhite.wordpress.com/designer/type-design/%E2%80%9Cputting-it-together-achieving-a-unified-design%E2%80%9D/.

28. **Krug, Steve.** *Don't make me think!* místo neznámé : Pearson Education (US), 2013. 0321965515.

29. **Spradlin, Liam.** Android Police. *Exclusive: Quantum Paper And Google's Upcoming Effort To Make Consistent UI Simple.* [Online] 11. 6 2014. [Citace: 7. 7 2020.] https://www.androidpolice.com/2014/06/11/exclusive-quantum-paper-and-googles-upcoming-effort-to-make-consistent-ui-simple/.

30. **Google.** Material Design. *Material Design.* [Online] Google Inc. [Citace: 7. 7 2020.] https://material.io/design/introduction#getting-around.

31. **Medium.** Top 3 Reasons to Implement Reusable Components. *Medium.* [Online] 7. 3 2018. [Citace: 7. 7 2020.] https://medium.com/@Imaginovation/top-3-reasons-to-implement-reusable-components-a91a7aaddd38.

32. **Spool, Jared.** Crappy Personas vs. Robust Personas. *UIE.* [Online] 14. 11 2007. https://archive.uie.com/brainsparks/2007/11/14/crappy-personas-vs-robust-personas/.

33. **Snížek, Martin.** První setkání Czech SIGCHI. *Martin Snížek píše o webu.* [Online] 26. 3 2006. [Citace: 8. 7 2020.] https://www.snizekweb.cz/c/czech-sigchi-1.

34. **Pecina, Martin.** PROČ JE GRAFICKÝ DESIGN UŽITEČNÝ. *UNIE GRAFICKÉHO DESIGNU.* [Online] 1. 9 2012. [Citace: 14. 7 2020.] https://unie-grafickeho-designu.cz/proc-je-graficky-design-uzitecny/#.XwznFygzaUk.

35. **Crispin, Lisa.** *Agile Testing.* 978-0321534460.

36. **Patton, Ron.** *Testování softwaru.* místo neznámé : Computer Press (CP Books), 2002. 80-7226-636-5.

37. **Rasmussen, Jonathan.** *Way of the Web Tester.* místo neznámé : The Pragmatic Programmers, 2016. 9781680501834.

38. **Jung, June.** How to test software, part I: mocking, stubbing, and contract testing. *Circleci Blog.* [Online] 4. 4 2019. [Citace: 16. 7 2020.] https://circleci.com/blog/cd-for-ionic-apps/.

39. introduction.js. *Jasmine.* [Online] Jasmine. [Citace: 16. 07 2020.] https://jasmine.github.io/2.0/introduction.

40. **Martorell, Sebastian.** Automate your UI testing. *endeec.* [Online] 15. 1 2018. [Citace: 16. 7 2020.] http://www.endeev.com/blog/automate-your-ui-testing/.

41. **Ken Schwaber, Jeff Sutherland.** The Scrum Guide. *Scrum Alliance.* [Online] 2017. [Citace: 21. 07 2020.] https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf.

42. **Jetbrains.** YouTrack. *JetBrains.* [Online] 2020. [Citace: 21. 7 2020.] https://www.jetbrains.com/youtrack/.

43. **Cohn, Mike.** *User stories applied: for agile software development.* Boston : Addison-Wesley, 2004. 03-212-0568-5.

44. **Mozilla.** Interact with the clipboard. *MDN web docs.* [Online] 25. 6 2019. [Citace: 26. 7 2020.] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Interact_with_the_clipboard.

45. —. Selection. *MDN Web Docs.* [Online] 17. 1 2020. [Citace: 4. 8 2020.] https://developer.mozilla.org/en-US/docs/Web/API/Selection.

10 Seznam obrázků

11 Seznam zdrojových kódů

12 Seznam příloh