Masarykova univerzita
Fakulta informatiky

Implementace Virtual DOM pro klientskou vrstvu frameworku DotVVM

Diplomová práca

Bc. Martin Bojnanský

Brno, jar 2018

Masarykova univerzita
Fakulta informatiky

Implementace Virtual DOM pro klientskou vrstvu frameworku DotVVM

Diplomová práca

Bc. Martin Bojnanský

Brno, jar 2018

Na tomto mieste sa v tlačenej práci nachádza oficiálne podpísané zadanie práce a vyhlásenie autora školského diela.

Vyhlásenie

Vyhlasujem, že táto diplomová práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Bc. Martin Bojnanský

Vedúci práce: RNDr. David Gešvindr

Zhrnutie

Diplomová práca sa zaoberá analýzou realizovateľnosti a možného prínosu aktualizácie klientskej vrstvy webového frameworku DotVVM. V práci je diskutovaný spôsob prenesenia a automatizácie jeho fundamentálnych princípov do rozhrania knižnice React, pričom dôraz je kladený na zachovanie kompatibility s aktuálnou verziou frameworku a zvýšenie rýchlosti dynamických zmien stránky. Súčasťou práce je implementácia prototypu jadra frameworku a jeho základných komponentov, ktoré slúžia k overeniu a identifikácii možného prínosu navrhovaného spôsobu aktualizácie.

Kľúčové slová

DotVVM, React, Knockout.js, JavaScript, C#, ASP.NET, MVVM

Obsah

Úvod **1**

1 DotVVM Framework **2**
1.1 *Dôvody vzniku* 2
1.2 *Základné princípy* 4
1.2.1 Návrhový vzor MVVM 4
1.3 *Načítanie stránky* 6
1.4 *Akcie a udalosti* 7
1.4.1 Command 7
1.4.2 StaticCommand 8
1.5 *Použitie DotVVM* 9
1.5.1 View.dothtml 9
1.5.2 ViewModel.cs 10
1.5.3 Výsledná aplikácia 11

2 JavaScript vo webových aplikáciách **13**
2.1 *ECMAScript* 13
2.2 *Single-Page Applications (SPA)* 15
2.3 *Document Object Model (DOM)* 16
2.4 *Knockout.js* 16
2.4.1 Použitie Knockout.js 17
2.4.2 Návrhový vzor Observer 20
2.5 *Virtual DOM* 21
2.6 *React* 23
2.6.1 Komponenty 23
2.6.2 JSX 25
2.6.3 Parametre komponentov 26
2.6.4 Stav komponentov 27
2.6.5 Životný cyklus 28
2.6.6 Virtual DOM 30
2.7 *Porovnanie knižníc Knockout.js a React* 32
2.7.1 Rýchlostné porovnanie 34

3 Návrh Virtual DOM pre klientsku vrstvu frameworku DotVVM **36**

3.1 *Transformácia DOTHTML* 36
3.2 *Stav aplikácie* 38
3.2.1 Implementácia stavového kontajneru 39
3.2.2 Integrácia stavového kontajneru 41
3.3 *Dátový kontext* 42
3.3.1 Dátový kontext v React komponentoch 43
3.4 *Komponenty* 44
3.4.1 Transformácia komponentov 45
3.5 *Zobrazenie kolekcií dát* 47
3.5.1 Transformácia zobrazenia kolekcií 47
3.6 *Viazanie dát* 49
3.6.1 Transformácia výrazov 50
3.7 *Spôsob kompilácie* 54
3.7.1 Renderovanie stránok na strane serveru 56
3.8 *Single Page Application (SPA)* 56

4 Overenie správnosti navrhovaného riešenia 57
4.1 *Implementácia prototypu* 57
4.2 *Testovanie prototypu* 58
4.2.1 Veľkosť stránky a závislostí prototypu 58
4.2.2 Rýchlostné porovnanie prototypu 59

Záver 62

Bibliografia 63

Zoznam tabuliek

4.1 Porovnanie veľkostí obsahu stránky a závislostí prototypu *59*
4.2 Porovnanie výkonnosti aplikačnej stránky s 2 položkami zoznamu *60*
4.3 Porovnanie výkonnosti aplikačnej stránky s 1000 položkami zoznamu *60*

Zoznam obrázkov

1.1 Návrhový vzor MVVM *5*
1.2 Proces načítania stránky frameworku DotVVM *6*
1.3 Akcia frameworku DotVVM *7*
1.4 Statická akcia frameworku DotVVM *8*

2.1 ECMAScript Transpilation *14*
2.2 ECMAScript Polyfill *14*
2.3 Návrhový vzor Observer *20*
2.4 Proces renderovania stránky HTML *22*
2.5 Manipulácia s rozhraním DOM *22*
2.6 Manipulácia s rozhraním VDOM *23*
2.7 Vznik komponentu React *29*
2.8 Zmena stavu komponentu React *29*
2.9 Zmena vstupných parametrov komponentu React *30*
2.10 Zánik komponentu React *30*
2.11 Renderovanie zmien stavu komponentu React *31*
2.12 1. predpoklad heuristického algoritmu React *32*
2.13 Renderovanie 1000 položiek zoznamu v Knockout.js a React *34*

3.1 Navrhovaný spôsob načítania stránky *55*
3.2 Navrhovaný spôsob kompilácie stránky *55*
3.3 Navrhovaný spôsob presmerovania stránky SPA *57*

4.1 Náhľad testovacej aplikácie prototypu *58*

Úvod

Výber správnych technológií pre vývoj webových aplikácií častokrát zohráva dôležitú úlohu v životnom cykle projektu a ovplyvňuje veľké množstvo jeho faktorov. Príkladom môže byť efektivita vývojového tímu, kvalita a zrozumiteľnosť zdrojového kódu, výkonnosť výslednej aplikácie alebo náklady na vývoj a údržbu produktu. Z tohto dôvodu sú čoraz obľúbenejšie technológie zjednodušujúce bežne používané princípy, ktoré tak šetria potrebný čas, vynaložené úsilie a finančné prostriedky spoločností. Ich podpora a popularita sa však v posledných rokoch neustále mení a výber vhodných technológií pre dlhodobejšie projekty sa tak stáva neustále náročnejší.

Česká spoločnosť RIGANTI s.r.o., ktorá sa typicky zameriava na zákazkový vývoj komplexných informačných systémov, sa rozhodla riešiť úskalia existujúcich webových technológií platformy ASP.NET vytvorením vlastného open-source frameworku s názvom DotVVM. Ten integruje moderné princípy pre tvorbu dynamických formulárových aplikácií a oslobodzuje vývojárov od použitia programovacieho jazyka JavaScript. Pre zaistenie interaktívneho grafického rozhrania používa automatizovanú kompiláciu predpripravených komponentov a špeciálnych syntaktických pravidiel do rozhrania JavaScriptovej knižnice Knockout.js. Tá je však s postupom času stále menej používaná, a preto spoločnosť aktívne uvažuje nad jej výmenou za modernejšiu technológiu. Vzhľadom na to, že kód rozhrania knižnice Knockout.js je od používateľa frameworku skrytý, túto technológiu by teoreticky bolo možné nahradiť za inú a pritom zachovať jeho spätnú kompatibilitu.

V posledných rokoch si získali veľa pozornosti a priazne technickej komunity knižnice založené na princípe označovanom Virtual DOM. Ten pomáha zefektívniť proces dynamických zmien stránok, ale predstavuje diametrálne odlišný spôsob práce s rozhraním DOM. Z týchto dôvodov je predmetom diplomovej práce analýza realizovateľnosti a možného prínosu nahradenia klientskej vrstvy využívajúcej knižnicu Knockout.js za implementáciu princípu Virtual DOM. Možnosť nahradenia tejto vrstvy by pomohla zaistiť rýchlejší beh nových i súčasných aplikácií a zároveň prilákať nových používateľov.

1 DotVVM Framework

DotVVM je open-source webový framework založený na platforme ASP.NET a inšpirovaný nedostatkami existujúcich technológií ASP.NET WebForms a MVC. Framework je primárne určený pre vývoj komplexnejších informačných systémov a jednostránkových aplikácií pre potreby všetkých segmentov. Takýmito aplikáciami sú zvyčajne systémy typu LOB[1], kde sa vyskytuje veľké množstvo procesov, formulárových stránok a užívateľských rolí. Ich typickým objednávateľom sú spoločnosti vyžadujúce prispôsobené riešenia systémov ako napr. ERP[2], CRM[3] a podobne. DotVVM sa sústreďuje na zabezpečenie a zjednodušenie často používaných konceptov, kde zaraďujeme:

- Definíciu komplexných formulárov
- Komunikáciu medzi klientskou a serverovou časťou aplikácie
- Validáciu a formátovanie dát
- Zobrazovanie a filtrovanie dát
- Lokalizáciu stránok
- Definovanie šablón stránok a parametrických komponentov
- Testovateľnosť
- Modernizáciu súčasných aplikácií

1.1 Dôvody vzniku

Dôvody vzniku frameworku DotVVM boli podnietené viacerými faktormi. Nimi sú napr.:

- Absencia podpory technológie ASP.NET WebForms pre platformu .NET Core

1. Line-of-Business
2. Enterprise Resource Planing
3. Customer Relationships Management

- Potreba zlepšenia a modernizácie princípov používaných technológiou ASP.NET WebForms
- Zjednodušenie vývoja aplikácií typu LOB v technológiách odporúčaných spoločnosťou Microsoft

V minulosti boli programátori zameraný na používanie užšieho množstva programovacích jazykov a nástrojov, zatiaľ čo v dnešnej dobe zaznamenávame neustále stupňujúce sa nároky. V každom prípade, nie vždy je pre firmy ľahké vyhradiť si dostatok finančných prostriedkov na zaškolenie programátorov. Z tohto dôvodu naberajú na popularite práve nástroje, ktoré zjednocujú odporúčané princípy, eliminujú znalostné požiadavky a v neposlednom rade optimalizujú čas a finančné náklady [1].

Technológia ASP.NET WebForms, prvýkrát predstavená v roku 2002, už zdá sa nesplňuje nároky na moderné webové aplikácie a postupne stráca podporu [2]. Okrem toho, modernejšie technológie ASP.NET MVC a WebAPI sú navrhnuté príliš obecne, aby vyhovovali požiadavkám rôznych typov aplikácií. Z tohto dôvodu je tvorba aplikácií typu LOB na platforme .NET Core o niečo zložitejšia a ich nedostatky sa začali riešiť pridávaním klientskych frameworkov ako Angular, Knockout.js alebo React [3]. Pokiaľ by sme sa podrobnejšie pozreli na fungovanie technológie ASP.NET MVC tak zistíme, že princíp zmien pri interakcii používateľa so stránkou a jej funkciami je náročný na prenos dát. Framework po odoslaní požiadavky opätovne načíta celú stránku a jej kompletný obsah zo servera [3].

Jedným z riešení optimalizácie prenosu dát je využitie klientskych technológii ako Knockout.js. Komunikácia klientskej a serverovej časti prebieha výmenou serializovaného[4] dátového modelu a šetrí veľké množstvo prenášaných dát a obnovení stránky. Po spracovaní požiadavky serverovou časťou sa odošle nový dátový model na klienta a vykonajú sa zmeny len v miestach kde došlo k aktualizácii referenčných hodnôt.

Kód pracujúci s takýmto frameworkom je možné generovať, a tak využiť jednoduchší spôsob implementácie a integrácie s platformou ASP.NET. Práve tu sa DotVVM snaží pomôcť so zaistením generovania

4. Serializácia je proces transformácie objektov do vhodnej podoby štruktúrovaných dát.

často opakového kódu. Automaticky sprostredkováva komunikáciu rôznych častí aplikácií, znižuje množstvo prenášaných dát, znalostných požiadaviek a potrebného času na vývoj webových aplikácií.

1.2 Základné princípy

1.2.1 Návrhový vzor MVVM

DotVVM nasleduje návrhový vzor Model View ViewModel, v skratke známy ako MVVM. Ten bol predstavený softvérovým architektom spoločnosti Microsoft už v roku 2005 a vznikol ako špecializácia návrhového vzoru Presentation Model od Martina Fowlera [4].

Výhody MVVM Návrhový vzor bol vytvorený na základe potrieb zlepšenia testovateľnosti grafického rozhrania, čo bolo dosiahnuté jeho oddelením od aplikačnej logiky. Medzi hlavné výhody MVVM patrí [4]:

- **Zlepšenie testovateľnosti grafického rozhrania**

  Pre testovanie funkcionality postačuje inicializovať inštanciu ViewModelu a nie je potrebné fyzicky vytvárať komponenty grafického rozhrania.

- **Zníženie závislosti grafického rozhrania a aplikačnej logiky**

  Výmena komponentu v grafickom rozhraní prakticky nemá vplyv na funkcionalitu ViewModelu a nevyžaduje si jeho zmeny.

- **Zvýšenie prenositeľnosti aplikačnej logiky**

  Tento princíp však nie je vždy možný a v praxi je len veľmi málo používaný.

Nevýhody MVVM Samozrejme, používanie tohto návrhového vzoru môže prinášať aj isté nevýhody [5]:

- **Platformy bežne nepodporujú použitie MVVM**
- **Vo väčšine programovacích jazykov si jeho implementácia vyžaduje väčšie množstvo zdrojového kódu**

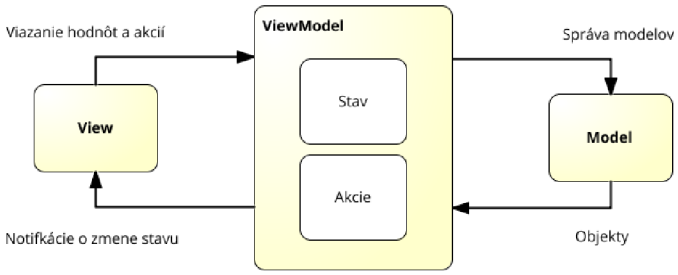
Napr.: C# aplikácie typicky musia naviac implementovať rozhranie INotifyPropertyChanged a knižnica Knockout.js vyžaduje používanie špeciálneho typu objektov.

- **Výkonnostné problémy pri nadmernom množstve prvkov užívateľského rozhrania**

  Tie sa často-krát objavujú pri rádovo tisícoch záznamov v tabuľke stránky.

Napriek tomu, framework DotVVM je navrhnutý tak, aby plne podporoval návrhový vzor MVVM a nevyžadoval si implementáciu nadbytočného zdrojového kódu. Tím zlepšuje architektonické rozloženie aplikácie a umožňuje jednoduchšiu testovateľnosť.

Štruktúra MVVM Aplikácia využívajúca návrhový vzor MVVM pozostáva z troch hlavných vrstiev:



Obr. 1.1: Návrhový vzor MVVM

- **Model**

  Modelmi sú entity aplikácie reprezentujúce akékoľvek dátové štruktúry alebo služby potrebné pre tvorbu aplikačnej logiky. Tieto entity sú typicky súčasťou triedy ViewModel a používajú sa na reprezentáciu stavu stránky alebo služieb biznis vrstvy pre získavanie dát a reakcií na akcie používateľa.

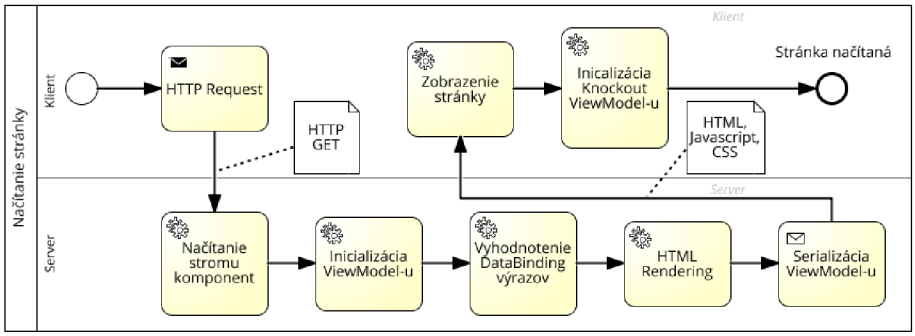
- **ViewModel**

  ViewModel je trieda nesúca aktuálny stav grafického rozhrania a obsahuje vlastnosti a akcie. Počet inštancií by spravidla mal odpovedať počtu jednotlivých grafických rozhraní označovaných View. ViewModel je zodpovedný aj za odosielanie notifikácií o zmenách svojho stavu triede View, ale nie za samotnú aktualizáciu jeho hodnôt [4]. Typicky by sa tu mala nachádzať len aplikačná logika týkajúca sa prezentačnej vrstvy aplikácie (napr. presmerovanie na inú stránku) a volania funkcií biznis vrstvy.

- **View**

  Predstavuje grafické rozhranie časti aplikácie a svoje hodnoty a akcie viaže na prislúchajúci ViewModel. Na základe notifikácií o zmenách stavu ViewModelu aktualizuje viazané hodnoty a tak dynamicky mení svoj stav. K tomu slúži princíp označovaný data-binding a v aplikácii DotVVM sa používa pre obojsmerný tok dát medzi vrstvami View a ViewModel. Zmeny hodnôt vo vrstve ViewModelu sa tak automaticky reflektujú vo vrstve View a naopak.

1.3 Načítanie stránky



Obr. 1.2: Proces načítania stránky frameworku DotVVM

Pri prvom zobrazení stránky sa po prijatí požiadavky na server inicializuje odpovedajúci strom komponentov reprezentovaný súborom

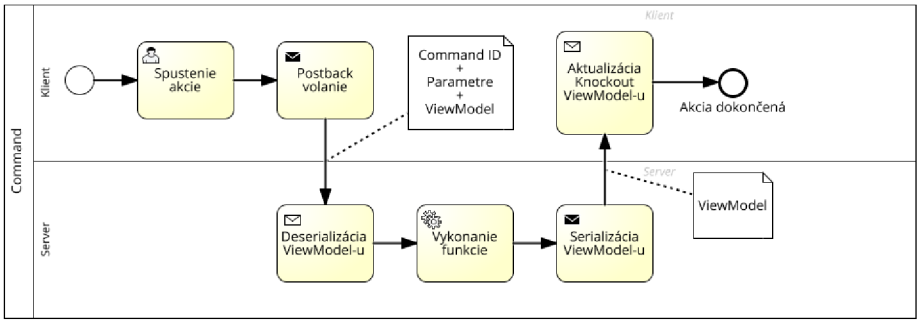
typu DOTHTML. Okrem načítania stromu komponentov sa inicializuje odpovedajúci ViewModel pre vyhodnotenie data-binding výrazov a následne prebieha preklad stránky do jazyka HTML. Jednotlivé referenčné hodnoty sú kompilované do syntaxi frameworku Knockout.js a prepájajú ich s odpovedajúcimi hodnotami v serializovanom ViewModeli.

1.4 Akcie a udalosti

Pri interakcii s webovou stránkou je neefektívne neustále opakovať proces načítania stránky, a preto DotVVM rozlišuje dva typy akcií. Tie sa snažia šetriť nielen množstvo prenášaných dát, ale zároveň zvyšovať rýchlosť celého procesu.

1.4.1 Command

Jedným z týchto typov akcií je Command. Command po svojom zavolaní odošle na server identifikačnú známku, parametre akcie a ViewModel s aktuálnym stavom aplikácie. Táto funkcia je následne identifikovaná serverom a po deserializácii ViewModelu sa vykoná. Server v poslednom kroku serializuje a odosiela nový ViewModel späť na klientske zariadenie. Po jeho úspešnom prijatí ho nastaví ako nový Knockout.js ViewModel. Nastavením nového ViewModelu sa prejavia požadované zmeny v grafickom rozhraní aplikácie.



Obr. 1.3: Akcia frameworku DotVVM

1.4.2 StaticCommand

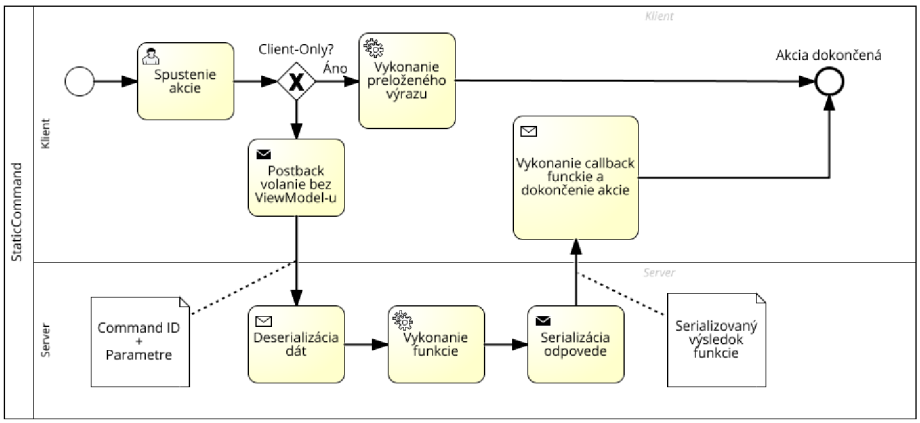
Pri používaní jednoduchších výrazov ako súčet viacerých hodnôt alebo formátovanie textu nie je potrebné využívať serverovú časť pre ich spracovanie. Z tohto dôvodu DotVVM disponuje tzv. statickými akciami. Statické akcie sú vykonávané buď čisto klientskym zariadením alebo pri volaní serverovej časti neprenášajú ViewModel aplikácie. Tieto akcie sú označované ako StaticCommand a rozlišujú dva prípady:

- **Funkciu je možné plnohodnotne a automaticky preložiť do jazyka JavaScript**

  Ich volanie žiadnym spôsobom nekontaktuje server a je vykonané lokálne na klientskom zariadení.

- **Funkciu nie je možné automaticky preložiť a je označená špeciálnym atribútom AllowStaticCommand**

  Na server sa odosiela zjednodušená požiadavka. Požiadavka nezahŕňa ViewModel s aktuálnym stavom aplikácie a po jej vykonaní je ako odpoveď vrátená len výsledná hodnota funkcie. Táto hodnota je ďalej spracovaná vygenerovaným JavaScriptovým kódom.



Obr. 1.4: Statická akcia frameworku DotVVM

1.5 Použitie DotVVM

Prácu s frameworkom DotVVM a základný princíp si pre lepšie pochopenie môžeme ilustrovať na zdrojovom kóde jednoduchej aplikácie pre zaznamenávanie úloh. Stránka aplikácie je rozdelená do dvoch súborov odpovedajúcich vrstvám návrhového vzoru MVVM.

1.5.1 View.dothtml

Prvým zo súborov je View.dothtml, ktorý reprezentuje grafické rozhranie stránky a okrem bežnej syntaxi jazyka HTML podporuje špeciálnu syntax pre:

- Data-binding výrazy
- Direktívy na identifikáciu príslušných modelov a šablón
- Preddefinované parametrické komponenty
- Validáciu dát

V žlto označenej časti na riadku 1 si môžeme všimnúť direktívu odkazujúcu kompilátor na ViewModel prislúchajúci tejto stránke. Na základe tohto prepojenia budú pri spracovávaní požiadavky načítania stránky vyhodnotené a transformované data-binding výrazy. Prepojenie je rovnako dôležité pre zaistenie funkcie IntelliSense.

V modro označenej časti na riadku 15 sa náchadza prepojenie akcie AddTask(). To je realizované použitím výrazu typu command. Ďalej kolekcia hodnôt Tasks je previazaná s ViewModelom aplikácie prostredníctvom výrazu typu value a nachádza sa v zeleno označenej časti na riadku 18. Obe prepojenia sú realizované ako parametrické hodnoty základných komponentov frameworku.

```
1  @viewModel DotVVM.Sample.TodoList.ViewModels.DefaultViewModel,
       DotVVM.Sample.TodoList
2  <!DOCTYPE html>
3  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
4    <head>
5      <meta charset="utf-8" />
6      <title>DotVVM.React</title>
7    </head>
```

```
8    <body>
9      <h1>{{value: Title}}</h1>
10     <p>
11       <dot:TextBox
12         Text="{value: NewTaskText}" />
13       <dot:Button
14         Text="Add Task"
15         Click="{command: AddTask()}" />
16     </p>
17     <dot:Repeater
18       DataSource="{value: Tasks}"
19       WrapperTagName="ul">
20       <ItemTemplate>
21         <li>
22           <dot:CheckBox
23             Checked="{value: IsCompleted}"
24             Text="{value: Text}" />
25           <dot:Button
26             Text="Delete"
27             Click="{command: _root.DeleteTask(_this)}" />
28         </li>
29       </ItemTemplate>
30     </dot:Repeater>
31   </body>
32 </html>
```

1.5.2 ViewModel.cs

Druhým zo súborov je ViewModel.cs, ktorý reprezentuje stav stránky a nájdeme v ňom deklarácie hodnôt a akcií referencovaných v súbore DOTHTML. Jednou z deklarácii hodnôt je kolekcia úloh Tasks (zeleno označená časť na riadku 11) a v grafickom rozhraní bola použitá ako zdroj hodnôt pre vykreslenie zoznamu úloh. Na riadku 13 je modrou farbou označená deklarácia akcie pridania novej úlohy. Tá bude spustená po kliknutí na tlačidlo definované v grafickom rozhraní.

```
1 using System.Collections.Generic;
2 using DotVVM.Framework.ViewModel;
3
4 namespace DotVVM.Sample.TodoList.ViewModels
5 {
6   public class DefaultViewModel : DotvvmViewModelBase
7   {
```

```
8       public string Title { get; set; } = "Task List";
9       public string NewTaskText { get; set; }
10
11      public List<TaskDTO> Tasks { get; set; } = new List<TaskDTO>();
12
13      public void AddTask()
14      {
15        Tasks.Add(new TaskDTO() { Text = NewTaskText });
16      }
17
18      public void DeleteTask(TaskDTO task)
19      {
20        Tasks.Remove(task);
21      }
22    }
23  }
```

1.5.3 Výsledná aplikácia

Po spracovaní požadovanej stránky je výstupom aplikácie webová stránka v jazyku HTML a JavaScript. Použité komponenty sú transformované na bežné HTML komponenty a ich vlastnosti sú preložené do syntaxi knižnice Knockout.js. Do stránky bol nsaviac ako skrytý element pripojený aj serializovaný ViewModel reprezentujúci aktuálny stav aplikácie. Vďaka automatickému prekladu a vygenerovaniu klientskej časti aplikácie, vývojár neprichádza do kontaktu s JavaScriptovým kódom ani knižnicou Knockout.js a k naprogramovaniu dynamického grafického rozhrania mu stačí využiť znalosť jazykov C#, HTML a CSS.

```
1  <!DOCTYPE html>
2  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
3    <head>
4      <meta charset="utf-8" />
5      <title>DotVVM.React</title>
6    </head>
7    <body>
8
9      <h1><!-- ko text: Title --><!-- /ko --></h1>
10     <p>
11       <input type="text" data-bind="value: NewTaskText"
           />
```

```
12        <input type="button" value="Add Task"
              onclick=dotvvm.postBack("root",this,[],
              "j6grtEuUIqcJIntx","",null,["validate-root"]);
              event.stopPropagation();return false; />
13      </p>
14      <ul data-bind="foreach: { "data": Tasks }" >
15          <li>
16            ...
17          </li>
18        </ul>
19      <script src="/dotvvmResource/knockout/knockout" type
          ="text/javascript"></script>
20      ...
21      <input type="hiddenid="__dot_viewmodel_root"value="{
22        "viewModel": {
23        "NewTaskText": null,
24        "Tasks": [
25        {
26          "IsCompleted": false,
27          "Text": "Go shopping"
28        },
29        {
30          "IsCompleted": false,
31          "Text": "Walk the dog"
32          }
33        ],
34        ...
35      }"/>
36      <script>
37      window.dotvvm.domUtils.onDocumentReady(function () {
38        window.dotvvm.init("root", "en-US");
39      });
40      </script>
41    </body>
42  </html>
```

2 JavaScript vo webových aplikáciách

JavaScript (pôvodne nazývaný Mocha a LiveScript), je skriptovací jazyk vyvinutý v roku 1995 spoločnosťou Netscape [6]. Od jeho vzniku prešiel veľkým množstvom zmien a v dnešnej dobe je podľa počtu aktívnych open-source projektov považovaný za najpopulárnejší programovací jazyk na svete [6]. JavaScript umožňuje dynamicky pracovať s obsahom stránky a tak tvoriť interaktívne grafické rozhrania bez nutnosti jej obnovenia. Jazyk sa natoľko rozšíril, že dnes je používaný aj na tvorbu serverových alebo mobilných aplikácií. Známymi serverovými riešeniami sú napr. MongoDB a Node.js.

Pre jednoduchosť jazyka Java sa spoločnosť Netscape rozhodla pre podobnú syntax a tak v kombinácii s podobným názvom došlo k častému zamieňaniu a nesprávnemu pochopeniu [6]. JavaScript je objektovo-orientovaný a dynamický programovací jazyk, ktorého kód býva umiestnený priamo v stránke ako HTML element alebo môže byť načítaný zo samostatného súboru. Z bezpečnostných dôvodov nie je povolené pristupovať k systémovým súborom a pre ukladanie dát slúžia špeciálne úložiská pre dočasné súbory.

2.1 ECMAScript

JavaScript ako programovací jazyk je závislý na webových prehliadačoch, ktoré zvyčajne disponujú rozdielnou podporou funkcionality a štandardou. Napriek solídnym pôvodným konceptom sa v snahe prispôsobiť jazyk odlišnostiam webových prehliadačov objavilo viacero nevhodných princípov [6]. Z tohto dôvodu vznikla špecifikácia skriptovacích jazykov nazývaná ECMAScript (ES). Jeho aktuálne najnovšou verziou je ES8 a najvyššou plne podporovanou verziou webovými prehliadačmi je ES5[1]. Pokiaľ chceme využívať najnovšie rozšírenia jazyka a zároveň zachovať plnú podporu pre webové prehliadače, sme nútený použiť dodatočné procesy zaisťujúce ich kompatibilitu.

1. Dostupné z: http://kangax.github.io/compat-table/es5/

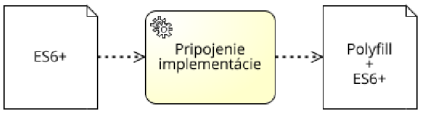
- **Transpilation**

  Anglický názov transpilation stojí za spojením transformácie a kompilácie kódu. Kód novšieho štandardu je pomocou určitých nástrojov kompilovaný do ekvivalentnej syntaxi niektorého zo starších štandardou [7]. Napr., využitím nástroja Babel je možné kód novšieho štandardu ES8 previesť na odpovedajúcu syntax štandardu ES5.

Obr. 2.1: ECMAScript Transpilation

- **Polyfills**

  Množstvo rozšírení si nevyžaduje transformáciu kódu, ale ich implementácia v staršom štandarde chýba. Pollyfill je princíp definovania chýbajúcej funkcionality a následného rozšírenia kódu [7]. Napr., v časoch kedy staršie webové prehliadače nedisponovali rozhraním pre prácu s dátovým formátom JSON sa využívala knižnica JSON2.js, ktorá toto rozhranie do prehliadačov pridala.



Obr. 2.2: ECMAScript Polyfill

2.2 Single-Page Applications (SPA)

Termín Single-Page Applications je používaný pre označenie jednostránkových webových aplikácií, ktoré počas interakcie nevyžadujú presmerovanie alebo obnovenie stránky [8]. Komunikácia so serverom ako aj zmeny grafického rozhrania sú realizované výhradne prostredníctvom JavaScriptového kódu. Obmedzením obnovení stránky sa znižuje potrebný čas na jej aktualizáciu [8] a grafické rozhranie je prekreslené plynulejšie a bez straty kontextu.

Výhody SPA

- Jednostránkové aplikácie sú rýchlejšie a plynulejšie, pretože väčšina zdrojov je načítaná pri prvom spustení a komunikácia so serverom prebieha len v podobe prenosu dát [8].
- Vývoj aplikácie je menej závislý na serverovej časti [8]. Tá ďalej nezaisťuje vykresľovanie stránok, ale sprostredkováva komunikáciu a výmenu dátových modelov. Takýto aplikačný model umožňuje využívať serverovú časť webovej aplikácie pre ďalšie platformy. Serverová časť aplikácie tak môže zabezpečovať výhradne úlohu REST API[2] a slúžiť ako spoločné komunikačné rozhranie pre webovú a mobilnú aplikáciu zároveň.
- Aplikácie sú testovateľné priamo v prehliadači.
- Načítané aplikácie je možné používať aj bez prístupu na internet.

Nevýhody SPA

- Aplikácie vykreslené na klientskom zariadení nemusia byť plne optimalizované pre internetové vyhľadávače [8].
- Prvé načítanie vyžaduje prenos väčšieho množstva dát [8].
- Stránka nemôže byť plnohodnotne zobrazená bez povoleného skriptovacieho jazyka vo webovom prehliadači.

2. REpresentational State Transfer

2.3 Document Object Model (DOM)

JavaScript je zväčša využívaný pre tvorbu dynamických webových aplikácií a je nesmierne dôležité aby dokázal manipulovať s jednotlivými komponentmi webových stránok. Webová stránka bežne pozostáva z jedného alebo viacerých dokumentov typu HTML. Aby skriptovacie jazyky dokázali manipulovať s komponentmi webovej stránky, HTML disponuje rozhraním označovaným Document Object Model (DOM) [6]. DOM je abstraktnou reprezentáciou HTML dokumentu a slúži ako model pre štrukturálne zmeny stránok prostredníctvom skriptovacích jazykov [6].

Webová stránka sa po načítaní stáva objektom, ku ktorému je možné pristupovať cez rezervovanú vlastnosť document alebo window.document. DOM Document Object je rodičovským elementom HTML dokumentu a vlastní všetky prvky aktuálnej webovej stránky. Nad modelom dokumentu alebo ktorýmkoľvek jeho potomkom je možné:

- Pohybovať sa stromom elementov.
- Vyhľadávať elementy na základe ich vlastností (typ, identifikátor, atď.).
- Meniť a pridávať obsah elementov a ich atribútov.
- Registrovať udalosti.

2.4 Knockout.js

Implementovať komplexné a dynamické grafické rozhranie u rozsiahlejších aplikácií môže byť náročné a zdĺhavé. Dynamické rozhranie sa často používa práve u formulárových stránok, kedy napr. po vyplnení niektorého z polí je potrebné iné polia zobraziť alebo naopak skryť. Okrem toho, v množstve prípadov kód vzniká vo viacčlenných tímoch a s pribúdajúcou funkcionalitou narastá aj množstvo kódu pre manipuláciu s HTML dokumentom. Voľnosť programovacieho jazyka dovoľuje implementovať kód odlišnými spôsobmi, a to môže mať časom nepriaznivý účinok na jeho údržbu. Preto je užitočné využívať

generický kód, ktorý zjednodušuje a zjednocuje procesy vývojového tímu.

Jednou z takýchto knižníc je Knockout.js. Knockout.js je jednoduchá a zároveň veľmi praktická JavaScriptová knižnica pre tvorbu dynamických klientskych aplikácií. Knižnica nasleduje návrhový vzor MVVM a slúži ako jednoúčelový nástroj [9]. Jeho hlavným cieľom je prepojiť ViewModel s grafickým rozhraním aplikácie.

Rovnako ako u DotVVM aplikácie, MVVM pomáha oddeliť aplikačnú logiku od grafického rozhrania. To priaznivo pôsobí na jeho testovateľnosť a JavaScriptový kód viac neobsahuje aplikačnú logiku pre manipuláciu s rozhraním DOM. Manipulácia so stránkou je špeciálnou syntaxou integrovaná priamo na komponenty HTML.

2.4.1 Použitie Knockout.js

Pre použitie knižnice je nutné:

1. **Pripojiť Knockout.js skript**

```
1 <script
2   type="text/javascript"
3   src="knockout-3.2.0.js">
4 </script>
```

2. **Vytvoriť ViewModel aplikácie**

   Vo ViewModeli aplikácie definujeme hodnoty, kolekcie hodnôt alebo akcie volané z grafického rozhrania.

```
1 var viewModel = {
2   title: "This is Knockout.js",
3   isChecked: ko.observable(true)
4 };
```

   - **Hodnota je statická**

     Hodnota sa počas interakcie so stránkou nemení, a teda nie je potrebné sledovať jej stav. Inicializujeme bežným spôsobom, ako JavaScriptový objekt:

```
1 title: "The Title"
```

- **Hodnota je dynamická**

  Hodnota sa počas interakcie so stránkou mení a elementy musia byť informované o zmenách. Hodnotu inicializujeme prostredníctvom funkcie ko.observable(), ktorá preberá nepovinný parameter s počiatočnou hodnotou:

```
ko.observable()
```

  alebo

```
ko.observable("John Doe")
```

3. **Naviazať hodnotu na vlastnosť elementu**

   Viazanie hodnôt je integrované do HTML prostredníctvom atribútu data-bind. Knockout.js pracuje s viacerými typmi viazania hodnôt, ktorých zoznam a podrobnejší popis je uvedený v oficiálnej dokumentácii[3].

```
<span data-bind="text: title"></span>
<input
  type="checkbox"
  data-bind="checked: isEnabled"/>
<ul data-bind="foreach: items"></ul>
<div data-bind="css:{ enabled: isEnabled }">
```

4. **Aplikovať Knockout ViewModel**

   Na aplikovanie ViewModelu slúži funkcia ko.applyBindings(). Inicializáciu ViewModelu je možné vykonať v momente kedy je pripravený DOM alebo ju pripojiť na koniec dokumentu HTML. Prvým povinným parametrom funkcie je objekt definujúci ViewModel aplikácie:

```
ko.applyBindings(viewModel);
```

   Pre aplikovanie ViewModelu len na určitý rozsah štruktúry HTML je možné využiť druhý nepovinný parameter. V tomto prípade nastavíme jeho hodnotu na selektor identifikujúci požadovaný element:

```
ko.applyBindings(viewModel, document.getElementById
    ("root"));
```

3. Dostupné z: http://knockoutjs.com/documentation/introduction.html

Nasledujúci zdrojový kód demonštruje ukážkovú aplikáciu pre zaznamenávanie úloh. Hodnota newTaskText je naviazaná na hodnotový parameter textového poľa stránky. Podobným spôsobom je definovaná udalosť click na tlačidlo „Add“, ktorá pridá novú hodnotu do kolekcie úloh tasks. Nižšie použitý typ viazania hodnoty foreach opakuje vykresľovanie šablóny pre každý prvok kolekcie tasks.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Knockout.js</title>
5      <script
6        type="text/javascript"
7        src="Resources/Scripts/knockout-3.4.2.js">
8      </script>
9      </head>
10     <body>
11       <p>
12         <input
13           type="text"
14           data-bind="value: newTaskText" />
15         <button
16           data-bind="click: add">
17           Add
18         </button>
19       </p>
20     <ul data-bind="foreach: tasks">
21       <li data-bind="text: text"></li>
22     </ul>
23     <script type="text/javascript">
24       var viewModel = {
25         newTaskText: ko.observable(),
26         tasks: ko.observableArray(),
27         add: function() {
28           this.tasks.push({
29             isDone: false,
30             text: this.newTaskText()
31           });
32           this.newTaskText("");
33         }
34       };
35       ko.applyBindings(viewModel);
36     </script>
37   </body>
38 </html>
```

2.4.2 Návrhový vzor Observer

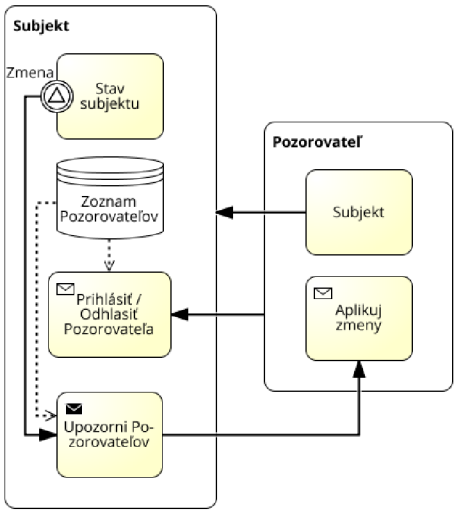
Na zaistenie dynamických zmien grafického rozhrania Knockout.js implementuje návrhový vzor Observer. Observer sa skladá z dvoch tried:

- **Subjekt (Subject)**

  Objekty tejto triedy môžu kedykoľvek meniť svoj stav a o svojej zmene informujú objekty typu Pozorovateľ [10].

- **Pozorovateľ (Observer)**

  Objekty triedy Pozorovateľ udávajú, že chcú byť informované o zmenách stavu určitého subjektu a reagujú na jeho zmeny [10].



Obr. 2.3: Návrhový vzor Observer

Princíp návrhového vzoru Observer je nasledovný:

1. Pozorovateľ informuje subjekt so žiadosťou o sledovanie zmeny stavu.

2. Subjekt si uloží pozorovateľa medzi aktívnych pozorovateľov.
3. Subjekt pri zmene hodnoty informuje všetkých Pozorovateľov o zmene stavu.
4. Aktívni pozorovatelia zareagujú na zmenu stavu subjektu.
5. Body 3-4 sa opakujú pokiaľ sa neodhlásia všetci pozorovatelia subjektu.

2.5 Virtual DOM

Vo webových aplikáciách sa čoraz viac využíva princíp jednostránkových aplikácií, ktoré neustále manipulujú s rozhraním DOM. Stromová štruktúra rozhrania DOM je síce nenáročná na prechádzanie, ale jej zmeny sú pre potreby jednostránkových aplikácií pomalé a závisle na konkrétnom type webového prehliadača [11]. To súvisí s procesom renderovania stránky, kedy prebieha [12]:

1. **Konštrukcia stromu DOM z analýzy HTML dokumentu**

   Načítaním a analýzou HTML dokumentu sú elementy postupne konvertované na DOM uzly a vzniká strom obsahu.

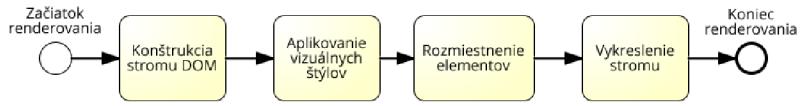
2. **Aplikovanie štýlov z analýzy CSS pravidiel**

   K stromu obsahu sú postupne pripojené CSS pravidlá a vzniká strom pre renderovanie. Tento strom obsahuje vizuálne vlastnosti ako napr. veľkosť a farba pozadia.

3. **Rozmiestnenie elementov**

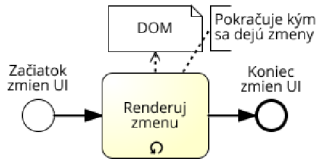
   Strom pre renderovanie je rozšírený o presné súradnice, kde na obrazovke majú byť umiestnené.

4. **Vykreslenie stránky na obrazovku webového prehliadača**



Obr. 2.4: Proces renderovania stránky HTML

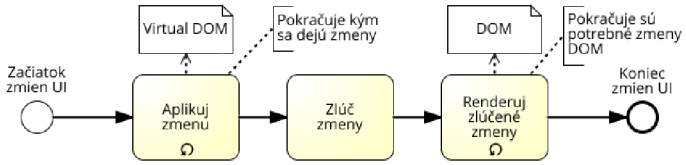
Problémom je, že pri súvislých zmenách grafického rozhrania často aktualizujeme viacero uzlov a proces renderovania sa opakuje pre každý z nich [13]. Predstavte si stránku webovej aplikácie s tabuľkou o 200 záznamoch. Akcia zmazania jedného z nich vyústi v spustenie SQL príkazu na serverovej časti a následne na klienta vráti aktuálnu sadu výsledkov. Avšak každá zmena objektu ViewModelu prinúti Knockout.js pristupovať k rozhraniu DOM a manipulovať s ním. Rovnako tak, pokiaľ by sme celú kolekciu dát zmazali a nahradili novou, Knockout.js bude generovať celú tabuľku odznova. To nie je zrovna najrýchlejšie, pretože generovanie vyžaduje množstvo operácií s rozhraním DOM a každá z nich predstavuje istú výkonnostnú réžiu.



Obr. 2.5: Manipulácia s rozhraním DOM

Z tohto dôvodu je výhodnejšie manipulovať s abstraktnou reprezentáciou DOM a zmeny zlúčiť do čo najmenšieho počtu renderovaní [13]. Súvislé úpravy sa postupne vykonávajú nad JavaScriptovými objektmi, ktoré sú kópiou natívnych objektov uzlov rozhrania DOM a manipulácia s nimi je o mnoho rýchlejšia, pretože nespúšťame proces renderovania stránky viac-krát. Ďalej sa na základe porovnania pôvodného stavu VDOM oproti novému rozhodne akým spôsobom zlúčiť zmeny uzlov čo najefektívnejšie. Až zlúčené zmeny vyplývajúce

z porovnania abstraktných stromov vyvolajú renderovanie nad rozhraním DOM. Tento princíp sa nazýva Virtual DOM (VDOM) a bol spopularizovaný najmä knižnicou React a frameworkom Vue [13].



Obr. 2.6: Manipulácia s rozhraním VDOM

2.6 React

Problémom optimalizácie výkonu jednostránkových webových aplikácií sa zaoberá aj spoločnosť Facebook Inc. Ich známa sociálna sieť Facebook je komplexnou webovou aplikáciou založenou na množstve dynamických dát a interakcií používateľov. To v roku 2013 spoločnosť priviedlo k predstaveniu nekonvenčnej JavaScriptovej knižnice nazývanej React [14]. Pre rozpor so základnými princípmi tvorby webových stránok bola knižnica spočiatku prijatá s istou dávkou skepticizmu [14], avšak podľa štatistík portálu Stackoverflow je dnes najobľúbenejšou knižnicou za rok 2017[4]. React neponúka komplexné riešenie pre tvorbu dynamických grafických rozhraní, ale sústreďuje sa výhradne na optimalizáciu ich výkonu a architektúry. Bežne používané koncepty ako smerovanie alebo lokalizácia teda nie sú jej súčasťou a sú ponechané na vývojárov a knižnice tretích strán.

2.6.1 Komponenty

Grafické rozhranie aplikácie React sa člení na menšie logické celky nazývané komponenty. Tie sú základným stavebným prvkom aplikácie [15] a s ohľadom na vstupné parametre a svoj stav definujú výstupný

4. Dostupné z: https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted

blok kódu HTML. Zapuzdrenie častí aplikačného kódu do komponentov umožňuje efektívnejšiu štrukturalizáciu aplikácie. Komponenty tak môžu byť súčasťou komplexnejšieho stromu komponentov alebo zdieľané medzi viacerými obrazovkami a aplikáciami.

Príkladom môže byť šablóna položky zoznamu úloh, ktorá na základe svojich vstupných parametrov zobrazí obsah položky v preddefinovanej štruktúre HTML. Tento komponent je možné definovať dvoma rovnocennými spôsobmi:

1. **Komponent ako definícia funkcie**

   Zrejme najjednoduchším spôsobom je zápis komponentu ako funkcie so vstupným parametrom pre vlastnosti a návratovou hodnotou pre požadované zobrazenie.

```
1  function TodoItem(props) {
2    return (
3      <label>
4        <input
5          type="checkbox"
6          checked={this.props.isDone} />
7        <span>{this.props.text}</span>
8      </label>
9    );
10 }
```

2. **Komponent ako definícia triedy**

   Komponenty sa bežne definujú aj ako triedy dediace od abstraktu React.Component. Jedinou podmienkou triedy je implementácia funkcie render() s návratovou hodnotou požadovaného zobrazenia.

```
1  class TodoItem extends React.Component {
2    render() {
3      return (
4        <label>
5          <input
6            type="checkbox"
7            checked={this.props.isDone}/>
8          <span>{this.props.text}</span>
9       </label>
10     );
11   }
12 }
```

Takto definovaný komponent následne môžeme vykresliť pomocou funkcie ReactDOM.render(), kde prvý parameter špecifikuje požadovaný komponent a druhý udáva DOM element v ktorom sa má zobraziť:

```
1 ReactDOM.render(
2   TodoItem,
3   document.getElementById("root")
4 );
```

2.6.2 JSX

Aby nemusel byť každý element zobrazenia definovaný volaním funkcie React.createElement(), knižnica pridáva rozšírenie jazyka JavaScript pod názvom JSX. Účelom rozšírenia je poskytnúť kratší zápis komplexného stromu elementov a ich parametrov [14]. JSX podporuje zápis komponentov vo formáte podobnom HTML a XML. Tie sú neskôr transformované na volania funkcií React.createElement().

S rozšírením JSX môžeme namiesto volania funkcie React.createElement():

```
1 React.createElement(
2   TodoItem,
3   {isDone: false, text: "Walk the dog"}
4 );
```

využiť rovnocenný zápis:

```
1 <TodoItem
2   isDone={false}
3   text="Walk the dog"
4 />
```

Oba typy zápisu elementu sú totožné a je ich možné ľubovoľne kombinovať. V jazyku JSX nájdeme aj syntax so svorkovými zátvorkami. Tá slúži pre vyhodnotenie výrazov, namiesto ktorých bude zobrazená ich návratová hodnota.

```
1 <span>
2   Task {this.props.isDone ? "is" : "isn't"} completed.
3 </span>
```

Syntax JSX je možné transformovať dvoma spôsobmi:

1. **Počas behu aplikácie**

   Transformácia počas behu aplikácie je síce jednoduchšia, ale má negatívny efekt na výsledný výkon aplikácie. Okrem toho syntax JSX nemôže byť minifikovaná a riadne testovaná v prehliadači [15].

2. **Pri zostrojení projektu**

   Efektívnejším spôsobom je proces transformácie pri zostrojí projektu [15]. Kód bude nahradený volaním funkcií React.createElement, čo umožní minifikáciu a testovateľnosť v prehliadači.

2.6.3 Parametre komponentov

Na prispôsobenie vlastností komponentov sa využívajú vstupné parametre označované skratkou props. Tie slúžia ako jednosmerný kanál pre tok dát z nadradených (rodičovských) komponentov. Parametre sú spracované pri renderovaní a nesmú byť žiadnym spôsobom modifikované samy sebou [15]. Zmeny parametrov sa teda prejavia jedine v prípade, že boli modifikované nadradeným komponentom a ten následne vyvolal funkciu renderovania.

Parametre je možné nastavovať dvoma spôsobmi:

1. **Ako kolekciu hodnôt funkcie React.createElement()**

```
1 React.createElement(
2   TodoItem, {isDone: false, text: "Walk the dog"}
3 );
```

2. **Ako atribúty elementu JSX**

```
1 <TodoItem
2   isDone={false}
3   text="Walk the dog"
4 />
```

K hodnotám parametrov sa v komponentoch pristupuje cez rezervovaný objekt props:

```
1 <span>
2   {this.props.text}
3 </span>
```

2.6.4 Stav komponentov

Zatiaľ čo parametre slúžia pre tok dát z nadradených komponentov, stav reprezentuje ich vlastné dynamické hodnoty. Aktualizáciou stavových hodnôt sa automaticky vyvolá jeho renderovanie, na základe ktorého sa prejavia požadované zmeny zobrazenia. Stav sa inicializuje vo vnútri konštruktora komponentu ako rezervovaný objekt state. Viazanie jeho hodnôt je založené na jednosmernom princípe a ich aktualizáciu je možné vykonať výhradne prostredníctvom funkcie setState. Zmena stavu sa následne dotýka všetkých podradených komponentov, ale žiadnym spôsobom nekomunikuje s nadradenými. V prípade potreby komunikácie s nadradeným komponentom sa využíva princíp prevzatia hodnôt a akcií prostredníctvom parametrov.

Príkladom stavového komponentu je napr. počítadlo, ktoré po kliknutí tlačidla pripočíta jednotkovú hodnotu a aktualizuje stav na obrazovke. V zeleno označenej časti na riadku 5 môžeme vidieť inicializáciu jednohodnotového stavu komponentu, ktorý je následne použitý vo funkcii render() na riadku 15. V modro označenej časti na riadku 19 je previazaná udalosť stlačenia tlačidla na funkciu onIncrement z riadku 8. Tá prostredníctvom funkcie setState aktualizuje stav komponentu a vyvolá funkciu render().

```
1  class Counter extends React.Component {
2    constructor(props) {
3      super(props);
4
5      this.state = { count: 0 };
6    }
7
8    onIncrement() {
9     this.setState({ count: this.state.count + 1 });
10   }
11
12   render() {
13     return (
14       <div>
15         <span>Count: {this.state.count}</span>
16         <input
17           type="button"
18           value="+"
19           onClick={this.onIncrement} />
```

```
20          </div>
21        );
22      }
23    }
```

2.6.5 Životný cyklus

Jednotlivé štádia a zmeny, ktoré postihujú existenciu komponentov sú popísané ich životným cyklom. V rámci neho rozlišujeme viacero fáz pre vznik, aktualizáciu a zánik. Tieto fázy sa skladajú z udalostí, na ktoré môže komponent reagovať, a tak prispôsobiť svoje chovanie.

Vznik komponentu

- **getDefaultProps()**

  Nie vždy je potrebné meniť vstupné parametre komponentov, preto môžu byť preddefinované tak, aby ich nebolo nutné pri každom použití komponentu nastavovať. Definovanie pôvodných hodnôt parametrov je možné práve vo funkcii getDefaultProps().

- **getDefaultState()**

  Obdobne existuje aj funkcia getDefaultState(), ktorá slúži na prednastavenie pôvodného stavu komponentu. Takto definovaný stav je užitočný najmä pri dedení komponentu.

- **componentWillMount()**

  Udalosť componentWillMount() nastáva tesne pred tým ako je komponent renderovaný a zavedený do stromu DOM.

- **render()**

  Slúži pre vykreslenie požadovaného zobrazenia. Funkcia render by v žiadnom prípade nemala manipulovať so stavom a parametrami komponentu [16].

- **componentDidMount()**

  Po renderovaní je komponent pripravený na definovanie manipulácie s rozhraním DOM alebo volanie dátových služieb [16].

Obr. 2.7: Vznik komponentu React

Aktualizácia stavu komponentu Stav komponentu môže byť po jeho zavedení kedykoľvek aktualizovaný. Proces aktualizácie stavu pozostáva z:

- **shouldComponentUpdate(nextProps, nextState)**

  Pred renderovaním komponentu je vždy vykonaná funkcia shouldComponentUpdate() [16]. Tu je možné upraviť logiku potreby renderovania, a to prípadne vynútiť alebo zakázať.

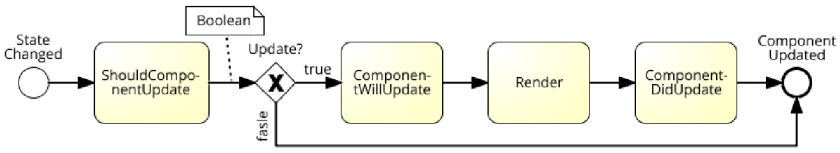
- **componentWillUpdate(nextProps, nextState)**

  Pokiaľ má nastať renderovanie tak proces aktualizácie stavu pokračuje funkciou componentWillUpdate(). Tá slúži na pripravenie aktualizácie a nemala by manipulovať so stavom komponenty [16].

- **render()** viď 2.6.5

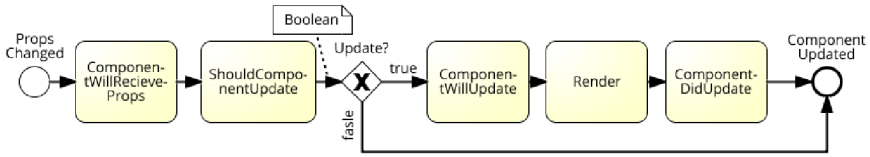
- **componentDidUpdate(prevProps, prevState, snapshot)**

  Podobne ako componentDidMount() slúži na operácie s rozhraním DOM [16].



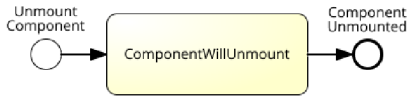
Obr. 2.8: Zmena stavu komponentu React

Aktualizácia parametrov komponentu Proces aktualizácie parametrov komponentu je takmer identický s procesom aktualizácie stavu [16]. Na začiatku procesu sa naviac nachádza metóda componentWillRecieveProps(nextProps), ktorá umožňuje aktualizáciu stavu komponentu tak, aby nebolo nutné vykonávať jeho renderovanie viac krát.



Obr. 2.9: Zmena vstupných parametrov komponentu React

Zánik komponentu Poslednou fázou životného cyklu komponentu je proces zániku. Ten sa skladá z jedinej funkcie componentWillUnmount(), ktorá je vykonaná tesne pred odstránením komponentu z DOM [16]. Metódu je možné využiť na operácie čistenia, deštrukciu časovačov a pod.



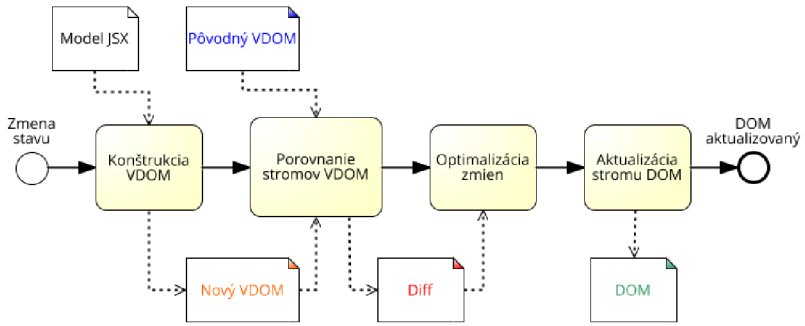
Obr. 2.10: Zánik komponentu React

2.6.6 Virtual DOM

Jedným z hlavných cieľov knižnice React je zrýchlenie aktualizácie grafického rozhrania [15]. To je dosiahnuté minimalizáciou náročných operácií s rozhraním DOM, ktoré sú potrebné pre jeho zmeny. Pre tieto účely je React implementáciou princípu VDOM diskutovaného v kapitole 2.5. Na základe zmien stavu jednotlivých komponentov knižnica špeciálnym algoritmom vyhodnocuje minimálny počet operácií s

rozhraním DOM. Programátor teda nekomunikuje s rozhraním DOM, ale deklaruje požadované zmeny a knižnica React rozhodne akým najefektívnejším spôsobom je možné tieto zmeny aplikovať.

Pri prvotnom zobrazení je z modelu JSX skonštruovaný abstraktný strom VDOM, ktorý je následne vykreslený do rozhrania DOM. V prípade aktualizácie stavu niektorého z komponentov sa z nového stavu vytvorí nový strom VDOM a porovná sa s pôvodným. Na základe zistených zmien sa ďalej optimalizujú potrebné operácie a požadované zmeny sa vykreslia do rozhrania DOM [13].



Obr. 2.11: Renderovanie zmien stavu komponentu React

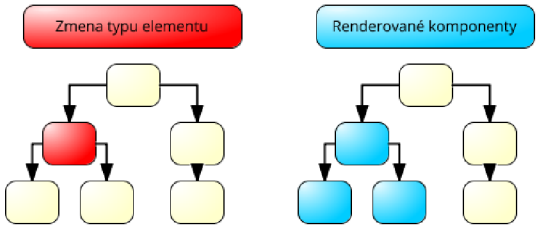
Zlúčenie zmien je však netriviálny problém o zložitosti $O(n^3)$, a preto React využíva heuristický algoritmus o zložitosti len $O(n)$ [13]. Ich algoritmus je založený na dvoch predpokladoch:

1. **Odlišné typy elementov produkujú odlišný strom elementov**

   Takéto elementy nie sú vôbec porovnávané a ich obsah je kompletne nahradený. Príkladom môže byť zmena elementu typu na .

2. **Elementy označené kľúčom sú pri opakovanom renderovaní považované za stabilné**

   Unikátne a stabilné kľúče pomáhajú porovnávaciemu algoritmu identifikovať položku, ktorá sa zmenila, bola pridaná alebo naopak odstránená. To je užitočné najmä pri renderovaní kolekcií,



Obr. 2.12: 1. predpoklad heuristického algoritmu React

kedy nemusí byť znovu vykreslený celý zoznam elementov, ale len jeho vybraná časť [13].

Spoločnosť Facebook aktuálne pracuje na reimplementácii jadra knižnice, známym pod názvom React Fiber. Tá je prepisom porovnávacieho algoritmu VDOM a namiesto rekurzívneho prechádzania stromu využíva novú dátovú štruktúru. Nová dátová štruktúra VDOM by mala zaistiť rýchlejšie porovnávanie. Okrem toho by mal React Fiber obsahovať aj nové funkcie ako je inkrementálne renderovanie alebo možnosti jeho plánovania [13].

2.7 Porovnanie knižníc Knockout.js a React

Pochopiteľne, odlišné princípy knižníc Knockout.js a React majú isté výhody a nevýhody. Táto podkapitola uvádza môj vlastný názor na ich pozitíva a negatíva, ktoré som nadobudol počas práce s oboma technológiami.

- Knockout.js je mierne staršia knižnica, ktorá priamo manipuluje s rozhraním DOM. To pre nespočetné renderovanie komplexnejšej stránky nie je efektívne a princíp Virtual DOM použitý u React sa zdá byť výrazne rýchlejší. Každopádne, cena za vyšší výkon operácii s DOM je použitie väčšieho množstva operačnej pamäte.
- Kým knižnica Knockout.js je veľmi jednoduchá na naučenie a jej použitie je možné v niekoľkých krokoch, React si vyžaduje hlbšie poznatky o technológií a nastavení vývojového prostredia.

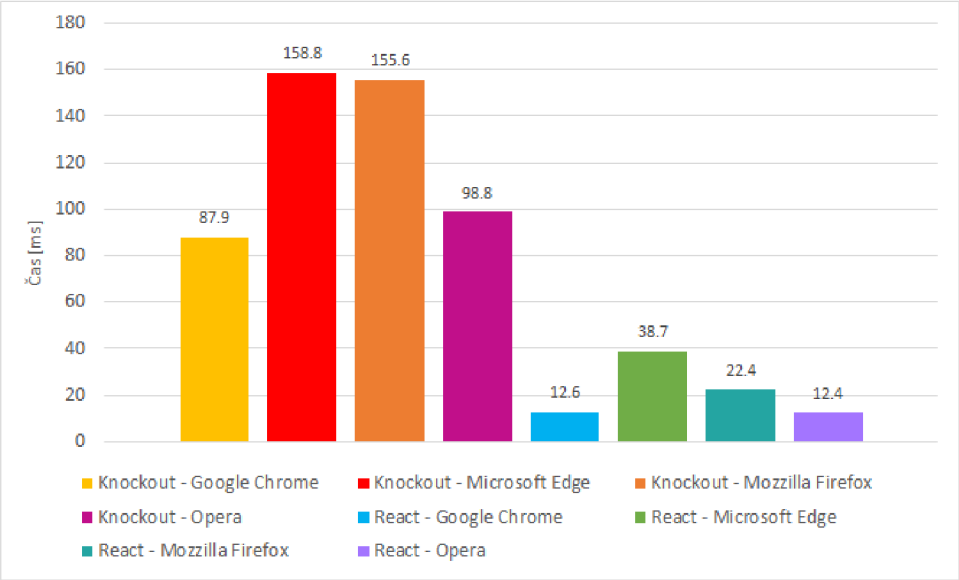
- Knockout.js je možné jednoducho implementovať aj do existujúcej stránky alebo šablóny, pretože definícia grafického rozhrania a dátový model sú logicky oddelené. Na viazanie dát využíva vlastný atribút elementov, ktorý je naviac validným atribútom od verzie HTML5. Naopak, knižnica React je založená na mierne nekonvenčnom prístupe, kedy rozhranie stránky je implementované výhradne v jazyku JavaScript a úzko späté s aplikačnou logikou.
- React je komponentovo-orientovaný, čo zaisťuje vyššiu znovupoužiteľnosť a prenositeľnosť aplikačných častí. Kód tak nemusí byť v prípade viac-násobného použitia kopírovaný ako pri knižnici Knockout.js. Okrem toho, komponenty sú zapuzdrené a tak ich aplikačná logika je zdieľaná pre všetky použitia a celkom nezávislá na ďalších komponentoch.
- Knockout.js je technológiou kompatibilnou aj so staršími prehliadačmi, zatiaľ čo React využíva novšie štandardy a jeho kompatibilita je podstatne obmedzenejšia.
- React využíva rozšírenie jazyka JavaScript, ktoré je nutné transformovať pri kompilácii aplikácie alebo počas jej behu.
- Veľkosť knižnice Knockout.js je výrazne menšia a nevyžaduje si tak veľké množstvo kódu ako obdobná aplikácia React.
- React podporuje len jednosmerné viazanie dát a pre aktualizáciu stavu jednotlivých komponentov je potrebné definovať udalosti. To je u niektorých typov aplikácií miernou nevýhodou.
- Pre webové prehliadače existujú rozšírenia pre testovanie aplikácií React, čo je oproti aplikáciám Knockout.js výhodou.
- Vývoj Knockout.js sa postupne spomaľuje a jej komunita slabne. Naopak komunita aj vývoj knižnice React postupne graduje.
- React ponúka možnosť renderovania na strane serveru, ktoré je vhodné pre účely optimalizácie stránky pre vyhľadávače.

S prihliadnutím na vyššie uvedené výhody a nevýhody, hodnotím knižnicu Knockout.js za oveľa prívetivejšiu a jednoduchšiu z pohľadu

vývojára webovej aplikácie. Avšak, pokles jej popularity, spomalenie vývoja a optimalizačné problémy hrajú viac v prospech knižnice React. Z perspektívy používateľa je rýchlosť aplikácií React nespochybniteľnou výhodou.

2.7.1 Rýchlostné porovnanie

Keďže jednou z najatraktívnejších výhod knižnice React je pravdepodobne rýchlosť aktualizácie grafického rozhrania aplikácie, rozhodol som sa toto tvrdenie overiť jednoduchým testom. K tomu slúžia dve pomerne jednoduché aplikácie, ktoré zaznamenajú čas pridania 1-tisíc položiek do zoznamu na obrazovke. Tento test je síce triviálny, avšak efektívnym spôsobom demonštruje častý prípad zobrazenia veľkej kolekcie dát, ktorý nájdeme vo viacerých komplexnejších systémoch. Všetky merania boli uskutočnené za rovnakých podmienok a to vo viacerých webových prehliadačoch. Uvedené hodnoty sú priemernými hodnotami z 10-tich meraní, pred ktorými bola stránka riadne obnovená.



Obr. 2.13: Renderovanie 1000 položiek zoznamu v Knockout.js a React

Z nameraných hodnôt je zrejmé, že React dosahuje výrazne lepších výsledkov. Zatiaľ čo Knockout.js zvládol vykreslenie položiek

v priemere za 125.275ms, u React to je iba 21.525ms. To je takmer 6 krát rýchlejšie a značí, že knižnica React založená na princípe VDOM je naozaj výkonnejšia. Vďaka meraniu si je možné všimnúť aj ďalší zaujímavý výsledok a to lepší výkon prehliadačov Google Chrome a Opera, ktoré na renderovanie využívajú jadro známe pod názvom webkit.

3 Návrh Virtual DOM pre klientsku vrstvu frameworku DotVVM

Táto kapitola sa zaoberá návrhom nového riešenia klientskej časti webového frameworku DotVVM, ktoré nahradí aktuálny spôsob viazania dát v grafickom rozhraní aplikácie za optimalizovanejšie riešenie využívajúce princíp Virtual DOM. Súčasná klientska vrstva frameworku zaisťuje obojsmerné viazanie dát prostredníctvom knižnice Knockout.js, ktorej implementácia návrhového vzoru Observer môže byť v prípade komplexnejších stránok neefektívna. Možným riešením tohto problému je využitie knižnice React, ktorá je implementáciou spomínaného princípu Virtual DOM a vykazuje lepšiu výkonnosť (viď. 2.6). Z kapitoly 2.7 však vyplýva, že tieto knižnice sú výrazne odlišné a ich nahradenie si vyžaduje rozsiahlejšie zmeny klientskej časti frameworku. Pre účely zistenia realizovateľnosti takejto zmeny sa diplomová práca ďalej venuje návrhu stavebných konceptov, medzi ktoré patrí:

- Reprezentácia stránok a komponentov v prostredí Virtual DOM
- Zaistenie obojsmerného viazania dát a vstupných parametrov
- Spôsob kompilácie aplikácie DotVVM
- Princíp viac-stránkových a jednostránkových aplikácii

Pozn. Dôležitým faktorom týchto zmien je zachovanie kompatibility s aktuálnym riešením.

3.1 Transformácia DOTHTML

Základnou stavebnou myšlienkou návrhu je transformácia stránok DOTHTML na dva súbory. Prvý z nich bude obsahovať základnú štruktúru HTML dokumentu, ktorá sa skladá z elementov html, head a body. Telo stránky sa ďalej extrahuje do separátneho komponentu rozhrania React a podľa uváženia môže/nemusí byť nahradený elementom pre jeho vykreslenie. Vykreslenie komponentov priamo do

tela stránky však zobrazí v konzolovej časti aplikácie varovanie o neodporúčanom princípe, pretože element body je často ovplyvňovaný knižnicami tretích strán. Na druhej strane, pridanie elementu by znamenalo zmenu chovania oproti predchádzajúcej verzii frameworku DotVVM a vyžadovalo by miernu úpravu stávajúcich aplikácií pri aktualizácii verzie frameworku. Osobne by som viac odporúčal princíp vykreslenia komponentu do dedikovaného elementu, a tak zabránil generovaniu varovných hlášok a možnému opakovaniu zahrnutých skriptov.

Pre ilustráciu slúži nasledujúci príklad transformácie jednoduchej DOTHTML stránky s jediným elementom vo vnútri tela dokumentu.

```
1  @viewModel DotVVM.Sample.TodoList.ViewModels.
       PageViewModel, DotVVM.Sample.TodoList
2
3  <!DOCTYPE html>
4  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
5    <head>
6      <meta charset="utf-8" />
7      <title></title>
8    </head>
9    <body>
10     <h1>Hello, world!</h1>
11   </body>
12 </html>
```

Hlavička dokumentu sa pri transformácii rozšíri o importovanie závislostí, medzí ktoré patrí knižnica React a skripty pre zaistenie funkcionality frameworku DotVVM. Telo dokumentu sa následne nahradí elementom označeným root a rozšíri o skript definujúci základný komponent stránky. Tento skript je pridaný na koniec stránky tak, aby bol vykonaný až v momente keď je stránka s referenciami načítaná a vykreslená do rozhrania DOM.

```
1  <!DOCTYPE html>
2  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
3    <head>
4      <meta charset="utf-8"/>
5      <title></title>
6      <script src="res/react.development.js"></script>
7      <script src="res/react-dom@16/umd/react-dom.
           development.js"></script>
8      <script src="res/dotvvm-react.min.js"></script>
9    </head>
```

```
10    <body>
11      <div id="root"></div>
12      <script src="Page.dothtml.js"></script>
13    </body>
14  </html>
```

Telo dokumentu je transformované na komponent rozhrania React a jeho obdobnú reprezentáciu v rozšírení JSX nájdeme ako súčasť funkcie render(). Obsah tela dokumentu je naviac obalený do neviditeľného elementu React.Fragment, ktorý umožňuje špecifikovať viacero elementov ako návratovú hodnotu funkcie render().

```
1  class DotView extends DotComponent {
2    render() {
3    return(
4      <React.Fragment>
5        <h1>
6          Hello, world!
7        </h1>
8      </React.Fragment>
9    );
10   }
11 }
12
13 ReactDOM.render(
14   <DotView/>,
15   document.getElementById("root")
16 );
```

3.2 Stav aplikácie

Z kapitoly 2.6 vyplýva, že každý komponent knižnice React môže zvonku prijímať vstupné parametre alebo pracovať so svojim vnútorným stavom. Na základe vyvolania zmeny stavu sa následne komponent prekreslí a aktualizuje všetky podradené komponenty, ktorých vstupné parametre závisia na novej stavovej hodnote. Naproti tomu, aplikácia DotVVM využíva návrhový vzor MVVM, kde stav zobrazenia je súčasťou vrstvy ViewModel a vo väčšine prípadov je spoločný pre viacero komponentov. Problémom je, že komponenty knižnice React sú zapuzdrené a podporujú výhradne jednosmernú komunikáciu smerom od nadradeného komponentu k podradenému. Ak by sme

teda chceli z podradeného komponentu aktualizovať stav nadradeného, sme nútený zvrchu predať parameter definujúci aktualizačnú funkciu. To by sa pri komplexnejšom strome komponentov stránky mohlo stať veľmi nepraktické a zložité na automatizovanie, pretože všetky aktualizačné funkcie aj stav zobrazenia by sa museli postupne predávať od najvrchnejšej úrovne až po komponent kde sú použité.

Riešenie tohto problému je inšpirované knižnicou Redux, ktorá extrahuje stav celej aplikácie do jediného zapuzdreného objektu, mimo koreňový komponent. Ten je následne vstupným parametrom všetkých komponentov a tak umožňuje manipulovať s aplikačným stavom z ktorejkoľvek časti stránky. Tento objekt sa označuje aj ako predvídateľný stavový kontajner, pretože jeho stav nie je možné meniť priamo, ale výhradne prostredníctvom preddefinovaných akcií. To má výhodu najmä u komplexnejších aplikácií, keďže akákoľvek manipulácia s ich stavom môže byť pre účely testovania zaznamenávaná a kedykoľvek replikovaná alebo vrátená.

3.2.1 Implementácia stavového kontajneru

Predvídateľný stavový kontajner je inicializovaný dvoma vstupnými parametrami a ako rozhranie objektu exportuje štyri základné funkcie. Prvým vstupným parametrom je počiatočný stav pod názvom initialState. Ten sa pri vzniku objektu uloží do privátnej premennej objektu tak, aby nemohol byť zvonku aktualizovaný. Na definovanie možných akcií a ich spôsobu aktualizácie stavu slúži druhý vstupný parameter pomenovaný reducer. Ten preberá redukčnú funkciu, ktorá na základe typu a parametrov akcie upraví kópiu aktuálneho stavu.

Rozhranie kontajneru ďalej exportuje funkcie pre čítanie, aktualizáciu a pozorovanie stavu. Čítanie stavu je dostupné prostredníctvom primitívnej funkcie getState(), ktorá jednoducho vracia hodnotu privátnej premennej. Pre aktualizáciu stavu slúži funkcia dispatch(action). Tá najskôr cez redukčnú funkciu upraví stav aplikácie a postupne zavolá aktualizáciu všetkých pozorovateľov. Tie sú registrované treťou exportovanou funkciou pod názvom subscribe(listener) a využívajú sa najmä na volanie funkcie renderovania komponentu. Pokiaľ by sme pre každú zmenu hodnoty niektorého z elementov stránky konštruovali celý abstraktný strom DOM, tak by sme s narastajúcou komplexnosťou stránky pozorovali výrazny pokles rýchlosti týchto zmien. Z tohto

dôvodu sú jednotlivé pozorovacie udalosti naviazané na vynútenie aktualizácie komponentov v ktorých sú použité. Okrem toho si môžeme všimnúť, že v prípade akcie typu SET sa aktualizuje len komponent, ktorý zmenu vyvolal. Pokiaľ sa však jedná o dokončenie zmien (napr. dokončenie písania do textového poľa), tak akcia typu APPLY spustí obnovenie aj všetkých ďalších komponentov, ktoré s touto hodnotou pracujú.

```
1  const dotStore = (initialState, reducer) => {
2    let state = initialState;
3    let listeners = {};
4
5    const getState = () => state;
6
7    const dispatch = (action) => {
8      state = reducer(state, action);
9
10     switch (action.type) {
11       case "SET":
12         action.forceUpdate();
13         break;
14       case "APPLY":
15         listeners[action.path].forEach(listener =>
               listener());
16         break;
17       case "POSTBACK":
18         listeners["rootViewModel"].forEach(listener =>
               listener());
19     }
20   };
21
22   const subscribe = (listener) => {
23     listeners[listener.path].push(listener.forceUpdate);
24   };
25
26   const unsubscribe = (listener) => {
27     ...
28   };
29
30   return { getState, dispatch, subscribe, unsubscribe };
31 }
```

3.2.2 Integrácia stavového kontajneru

Použitie generického stavového kontajneru umožní v generovanej aplikácii jednoduchým spôsobom zdieľať stav naprieč všetkými komponentmi a zároveň oddelí aplikačnú logiku od definície grafického rozhrania. Kontajner tak vytvorí jednotný a stabilný bod určený k predávaniu stavu medzi serverovou a klientskou časťou, čo súčasne odpovedá doterajšiemu riešeniu a návrhovému vzoru MVVM. Princíp integrácie stavového kontajneru s komponentmi knižnice React môže byť nasledovný:

1. Pred inicializáciou stavového kontajneru sa definuje redukčná funkcia, ktorá na základe typu a parametrov akcie rozhodne akým spôsobom sa má zmeniť stav aplikácie. Redukčná funkcia bude obsahovať zopár generických akcií zabezpečujúcich funkcionalitu frameworku a zároveň bude súčasťou jeho závislostí, tak aby nemusela byť definovaná pri každom jednom zobrazení. Konkrétna implementácia však úzko súvisí s viazaním dát a komunikáciou serverovej a klientskej časti aplikácie, a preto bude podrobnejšie diskutovaná v kapitole 3.6.

```
1 const dotReducer = (state, action) => {
2   switch(action.type) {
3     ...
4     default:
5       return state;
6   }
7 }
```

2. Pri načítaní stránky sa inicializuje stavový kontajner s počiatočnou hodnotou a deklaráciou redukčnej funkcie. Stav aplikácie môže niesť reprezentáciu aktuálnych ViewModelov DotVVM aplikácie, ale aj hodnoty jej globálnych nastavení a podobne.

```
1 const store = dotStore(
2   {
3     rootViewModel: {
4       Title: "Task List",
5       NewTaskText: "Go",
6       Tasks: [
7         { Text: "Go shopping", IsCompleted: false },
8         { Text: "Walk the dog", IsCompleted: false }
```

```
9         ]
10      }
11    },
12    dotReducer
13  );
```

3. V tomto kroku sa deklaruje funkcia vykreslenia koreňového komponentu do existujúceho elementu DOM.

```
1  const render = () => {
2    ReactDOM.render(
3      <DotView/>,
4      document.getElementById("root")
5    );
6  };
```

4. V predposlednej fáze sa prihlasuje volanie funkcie vykreslenia koreňového komponentu na zmenu hodnoty ViewModelu. To neskôr pri aktualizácii stavu aplikácie zabezpečí renderovanie komponentov a prekreslenie stránky. Avšak treba si uvedomiť, že v skutočnosti sa zostrojuje len abstraktná reprezentácia DOM, z ktorej knižnica React následne optimalizuje potrebné zmeny vykonané nad rozhraním DOM.

```
1  store.subscribe(
2    { path: "rootViewModel", forceUpdate: render }
3  );
```

5. Posledným krokom inicializácie je volanie funkcie vykreslenia koreňového komponentu pri načítaní stránky.

```
1  render();
```

3.3 Dátový kontext

Všetky elementy aplikácie DotVVM disponujú parametrickou vlastnosťou DataContext, ktorá im umožňuje špecifikovať oblasť dát, nad ktorou sú vyhodnocované výrazy pre viazanie stavových hodnôt alebo akcií. Hodnota dátového kontextu sa v štruktúre elementov automaticky dedí, a preto sa jeho nastavenie u niektorého z prvkov stránky aplikuje na všetkých jeho potomkov. Podobným princípom u knižnice

React je deklarácia vnútorného stavu komponentu, ktorý bohužiaľ nemôže byť zdieľaný s ostatnými časťami stránky ako tomu je v prípade návrhového vzoru MVVM. Využitie dátového kontextu naviac neobmedzuje aplikáciu na jedinú oblasť dát, ale u jednotlivých väzieb poskytuje možnosť pristupovať k hodnotám inej oblasti.

Pre ilustráciu je v nasledujúcej ukážke nastavený dátový kontext elementu na hodnotu NewContact. To zabezpečí, že všetky vnútorné väzby elementu budú vyhodnotené nad hodnotou jeho dátového kontextu. Pre pohyb hierarchiou kontextov sa pri viazaní dát využívajú aj rezervované názvy _this, _root alebo _parent:

- **_root** - Odkazuje na koreňový ViewModel aktuálnej stránky.
- **_parent** - Odkazuje na kontext nadradeného prvku.
- **_this** - Odkazuje na kontext aktuálneho elementu, čo je užitočné najmä v prípade predávania kontextu ako argumentu metódy.

```
1  @viewModel ContactsViewModel, ContactsApp
2  <div DataContext="{value: NewContact}">
3    <dot:TextBox Text="{value: Name}"/>
4    <dot:TextBox Text="{value: _this.PhoneNumber}"/>
5    <dot:Button Click="{command: _root.SaveContact(_this)}"
       />
6  </div>
```

```
1  public class ContactsViewModel {
2    public Contact NewContact { get; set; }
3
4    public void SaveContact(Contact contact) {
5      ...
6    }
7  }
8
9  public class Contact {
10   public string Name { get; set; }
11   public string PhoneNumber { get; set; }
12 }
```

3.3.1 Dátový kontext v React komponentoch

Jedným z možných riešení reprezentácie dátového kontextu, by bolo nahradenie všetkých väzieb pri kompilácii, za cesty rozšírené o dá-

tový kontext. Takéto riešenie by však nebolo schopné pracovať s indexmi kolekcií, pretože tie sú vykreslené na základe šablón až pri behu aplikácie. Okrem toho, v prípade opakovania dlhých selektorov by sa zbytočne zväčšoval výsledný zdrojový kód. Z tohto dôvodu navrhujem ponechať cesty väzieb dát a komponenty rozšíriť o vstupný parameter dátového kontextu. Pri vyhodnotení jednotlivých výrazov sa na klientovi prevedie spojenie dátového kontextu a väzby tak, aby cesta odpovedala konkrétnej hodnote v stavovom kontajneri aplikácie. Vďaka tomu, že spojenie cesty bude prebiehať na klientskom zariadení, vykreslenie kolekcií bude možné jednoduchým spôsobom rozšíriť o index špecifikujúci zdroj dát.

```
1  <DotComponent
2    Tag="div"
3    DataContext={{
4      _this: "rootViewModel.NewContact",
5      _root: "rootViewModel",
6      _parent: null
7    }}>
8    <DotTextBox
9      DataContext={{
10       _this: "rootViewModel.NewContact",
11       _root: "rootViewModel",
12       _parent: rootViewModel.NewContact
13     }}>
14     Text={this.bindable("Name")}/>
15   ...
16 </DotComponent>
```

3.4 Komponenty

Jednou z kľúčových funkcionalít frameworku DotVVM je implementácia vlastných komponentov. Tie sú bežne zdieľané pre viacero obrazoviek alebo aplikácií a pomáhajú štrukturalizovať aplikačný kód efektívnejšie. Ich základnou myšlienkou je zdieľanie znovu použiteľných časti grafického rozhrania tak, aby bol kód aplikácie prehľadnejší a dlhodobo udržiavateľný.

Podstatným rozdielom oproti rozhraniu React je, že komponenty DotVVM nešpecifikujú svoj vnútorný stav, ale sú závislé na vstupnom dátovom kontexte. Tým je zvyčajne ViewModel stránky alebo niektorá

z jeho hodnôt. Okrem toho, komponenty DotVVM naviac podporujú obojsmerný tok dát parametrov a prijímajú buď statické hodnoty alebo väzby na dáta, s ktorými môžu manipulovať.

3.4.1 Transformácia komponentov

Komponentový princíp oboch technológií zaiste zjednodušuje integráciu diskutovanej funkcionality a môže priniesť viaceré optimalizačné výhody. Pokiaľ neuvažujeme žiadané renderovanie serverovou časťou, jednou z výhod je, že viac-násobné použitie komponentu už nemusí byť vždy pred prenosom na klienta nahradené za výsledný kód HTML. Jeho reprezentáciu postačí odoslať na klienta len jediný krát a v prípade potreby bude využitá na renderovanie prostredníctvom knižnice React. Pochopiteľne, ďalšou z výhod je odstránenie šablón kódu z HTML alebo lepšia testovateľnosť.

Ako už bolo diskutované v kapitole 2.6, komponenty sú základným stavebným prvkom React aplikácie. Z tohto dôvodu navrhujem vytvoriť bázovú implementáciu komponentu, ktorá bude predstavovať rozhranie pre komunikáciu so stavovým kontajnerom, volanie akcií a ďalších bežne používaných konceptov. Predstavme si jednoduchý komponent pre položku zoznamu úloh, ktorý bude transformovaný na definíciu triedy jazyka JSX:

```
1  @viewModel DotVVM.Sample.Models.Task, Dotvvm.Sample
2  @baseType DotVVM.Sample.Controls.Task, Dotvvm.Sample
3  <li>
4    <label>
5      <input
6        type="checkbox"
7        checked="{controlProperty: Checked}" />
8      <span>
9        {{controlProperty: Text}}
10     </span>
11   </label>
12   <input
13     type="button"
14     Events.Click="{command: DeleteTask(_this)}" />
15 </li>
```

Odpovedá:

```
1  class Task extends DotComponent {
2    render() {
3      return (
4        <li>
5          <label>
6            <input
7              type="checkbox"
8              {...this.setAttributeBindable("checked", this.props.Checked)}
9            />
10           <span>
11           {this.evalBindable(this.props.Text)}
12           </span>
13         </label>
14         <input
15           type="button"
16           onClick={this.postback("j6grtEuUIqcJIntx")} />
17       </li>
18     );
19   }
20 }
```

Všimnime si, že v žlto-označenej časti sa nachádza transformácia väzby parametra na atribút elementu. Ten je preložený na funkciu typu setAttributeBindable(attribute, bindable), ktorá sa postará o samotné nastavenie atribútu a zároveň zabezpečí jeho obojsmernú väzbu na parameter komponentu. Pre zmenu, v zeleno-označenej časti nájdeme transformáciu hodnoty určenej výhradne na čítanie. Tá je sprostredkovaná ďalšou funkciou rozhrania pod názvom evalBindable(bindable). V neposlednom rade, v modro-označenej časti sa ďalej nachádza transformácia akcie, ktorá zabezpečí komunikáciu so serverom pomocou metódy postback(id).

Ďalším dôležitým faktom, ktorý treba zohľadniť pri transformácii DOTHTML je, že všetky jeho elementy môžu definovať nielen atribúty HTML, ale aj vlastnosti komponentov DotVVM. Sem patrí napr. dátový kontext, dynamické CSS triedy alebo udalosť kliknutia. Z tohto dôvodu, všetky bežné HTML elementy používajúce niektorú z vlastnosti frameworku budú taktiež transformované na nový komponent. V nasledujúcom príklade sú žltou farbou označené elementy DOTHTML, ktoré budú transformované na nový komponent.

```
1  <div DataContext="value: AppInfo»
2    <h1>
3      <span onClick={command: _root.AppTitleClick()}>
4        {{value: Title}}
5      </span>
6      <span Visible="value: IsDescriptionVisible»
7        {{value: Description}}
8      </span>
9    </h1>
10 </div>
```

3.5 Zobrazenie kolekcií dát

Pre zobrazenie kolekcie dát vo webovom rozhraní sa bežne používajú iteratívne princípy, ktoré pre každú z jej položiek postupne opakujú vykreslenie preddefinovanej šablóny. U frameworku DotVVM nájdeme viaceré takéto princípy integrované do zabudovaných komponentov. Sem patrí napr. komponent Repeater, ktorý pre jednotlivé položky kolekcie dát vlastnosti DataSource opakovane vykreslí hodnotu vlastnosti ItemTemplate.

```
1 <dot:Repeater
2   DataSource="{value: Tasks}"
3   WrapperTagName="ul">
4   <ItemTemplate>
5     <li>{{value:_this}}</li>
6   </ItemTemplate>
7 </dot:Repeater>
```

3.5.1 Transformácia zobrazenia kolekcií

Pre účely transformácie takéhoto komponentu navrhujem extrahovanie šablóny do separátnej triedy rozhrania React. Vzhľadom na to, že šablóny nie sú označované názvom a z dôvodu zapuzdrenia dátového kontextu nemôžu byť ani súčasťou parametra alebo obsahu inej triedy, je potrebné extrahovaný komponent označiť unikátnym menom a umiestniť ho mimo deklaráciu nadradeného komponentu. Odkaz na novo-vzniknutú triedu sa následne poskytne ako príslušný parameter ItemTemplate.

```
1  class DotTemplate-a3a3e686473 extends DotComponent {
2    render() {
3      return (
4        <React.Fragment>
5        <li>
6          <DotLiteral
7            DataContext={this.dataContext}
8            Text={this.bindable("_this")}
9            RenderSpanElement={false}/>
10       </li>
11       </React.Fragment>
12     );
13   }
14 }
15 class TasksPage extends DotComponent {
16   render() {
17     return(
18       <DotRepeater
19         DataContext={this.dataContext}
20         DataSource={this.bindable("Tasks")}
21         WrapperTagName="ul"
22         ItemTemplate={DotTemplate-a3a3e686473}
23       />
24     );
25   }
26 }
```

Z kapitoly 2.6.6 vyplýva, že pre efektivitu aktualizácie kolekcií DOM je vyžadované špecifikovanie unikátneho identifikátoru koreňového elementu šablóny. Na základe toho je jednoznačné, ktoré elementy boli aktualizované, a tak nie je nutné aktualizovať ich kompletný zoznam ale len vybranú časť. Podobný princíp nebol u aktuálnej verzie frameworku DotVVM potrebný, a preto je nutné vymyslieť spôsob akým kľúčom označovať koreňové elementy šablón. Medzi navrhované spôsoby patrí:

1. **Rozšírenie všetkých serializovaných objektov kolekcií o identifikačný parameter**

```
1 rootViewModel: {
2   Tasks: [
3     { _dotId: "76b5a618", Text: "Go shopping",
        IsCompleted: false },
```

```
4       { _dotId: "0d80c298", Text: "Walk the dog",
          IsCompleted: false }
5     ]
6   }
```

2. **Pridanie povinného parametru pre špecifikovanie kľúča**

   Takéto riešenie by pomohlo znížiť prenos nadbytočných dát, avšak znamenalo by nekompatibilitu s aktuálnou verziou frameworku.

```
1 <dot:Repeater
2   DataSource="{value: Tasks}"
3   WrapperTagName="ul">
4   <ItemTemplate ItemKeyBinding="{value: Id}">
5     <li>{{value:_Text}}</li>
6   </ItemTemplate>
7 </dot:Repeater>
```

3. **Kombinácia vyššie uvedených spôsobov**

   Pre zachovanie kompatibility by mohlo byť riešením pridať nepovinný parameter ItemKeyBinding, ktorý by mohol slúžiť ako možné nahradenie a optimalizácia riešenia číslo 1.

3.6 Viazanie dát

Viazanie dát (alebo data-binding) je kľúčovým mechanizmom pre synchronizáciu grafického rozhrania a stavových hodnôt aplikácie. Presnejšie, v DotVVM sa tento mechanizmus využíva na previazanie hodnôt a akcií vrstvy ViewModel so stránkou a jej komponentami. Na rozdiel od klasického imperatívneho spôsobu, kedy sa na zabezpečenie dynamických zmien manipuluje priamo z rozhraním DOM, u viazania dát sa využíva deklaratívnosť. Grafické rozhranie tak síce definuje čo sa má vykonať, ale nie akým spôsobom. Základnou motiváciou takéhoto princípu je oddelenie aplikačnej logiky a grafického rozhrania, čo však znamená, že viazanie dát musí byť podporované a plne sprostredkované samotnou technológiou. V aktuálnej verzii frameworku DotVVM sa synchronizácia rozhrania s dátovým modelom zaisťuje transformáciou väzieb na syntax knižnice Knockout.js, ktorej princíp je ale výrazne odlišný a nepodporuje obojsmerný tok dát.

3.6.1 Transformácia výrazov

Základným princípom pre zabezpečenie komunikácie rozhrania a dátového modelu by mohlo byť nahradenie jednotlivých typov väzieb za preddefinované funkcie pre prácu so stavovým kontajnerom. Navyše, tento spôsob by mohol pomôcť výrazne zjednodušiť preklad na strane kompilátoru, pretože každú z väzieb je možné priamo nahradiť funkciou a nemusia byť zoskupované do spoločného atribútu príslušného elementu. Okrem toho, cesty jednotlivých hodnôt výrazov môžu byť spracované až na klientskom zariadení, lebo všetky komponenty obsahujú parameter s aktuálnym dátovým kontextom.

Medzi pomocné funkcie transformácie a vyhodnotenia výrazov väzieb sa tak zaradia:

- **bindable(expression)**

  V prípade použitia niektorého z typu väzieb dát sa jednotlivé typy výrazy nahradia za pomocnú funkciu bindable(expression). Tá pri spracovaní vytvorí špeciálny typ objektu reprezentujúci väzbu v rozhraní React. Alternatívou by mohlo byť nahradenie výrazu priamo za serializovanú hodnotu objektu. Pomocná funkcia by však mohla slúžiť aj pre účely rozšírenia frameworku o možnosť implementácie komponentov v rozhraní React.

```
1 bindable(expression) {
2   ...
3   return new DotBindable(expression);
4 }
```

  Takto vytvorený objekt sa ďalej použije pre priame spracovanie alebo nastavenie parametra DotVVM komponentu. Pokiaľ sa jedná o výraz typu value, jeho hodnota sa transformuje na volanie funkcie bindable(expression), ale pre výrazy typu controlProperty sa nahradí čisto za volanie parametru komponentu, ktorý už objekt väzby obsahuje.

```
1 <dot:Literal Text="{value: Title}" />
2 <dot:Literal Text="{controlProperty: Title}" />
```

Odpovedá:

```
1 <DotLiteral Text={this.bindable("Title")} />
2 <DotLiteral Text={this.props.Title} />
```

- **evalBindable(bindable)**

  Pre vyhodnotenie objektu väzby sa využije druhá pomocná funkcia evalBindable(bindable). Keďže hodnotou parametra komponentu môže byť aj ľubovoľný typ a nie len výraz väzby dát, je dôležité aby sa funkcia rozhodla akým spôsobom zobrazí hodnotu. Okrem toho, pri vytvorení väzby sa komponent prihlasuje na pozorovanie stavu jej hodnoty, aby bol schopný v prípade jej zmeny iným komponentom zareagovať.

```
1  evalBindable(bindable) {
2    if(bindable instanceof DotBindable) {
3      if (!this.alreadySubscribed) {
4        this.store.subscribe({ path: this.buildPath(
             bindable.path), forceUpdate: () => this.
             forceUpdate() });
5      }
6
7      return this.get(bindable.path);
8    }
9    else {
10     return bindable;
11   }
12 }
```

  Metóda tak nahradí všetky výrazy väzieb v dvojitých zátvorkách a bude použitá aj v ďalšej pomocnej funkcii pre nastavenie väzby atribútu HTML elementu. Funkcia naviac prijíma vstupný parameter, ktorý závisí od typu použitej väzby. V prípade väzby typu value sa ako parameter priamo dosadí hodnota objektu. U väzby typu controlProperty bude vstupným parametrom odkaz na príslušný parameter komponentu.

```
1 {{value: Title}}
2 {this.evalBindable(this.bindable("Title"))}
```

Odpovedá:

```
1 {{controlProperty: Title}}
2 {this.evalBindable(this.props.Title)}
```

- **setAttributeBindable(attribute, bindable)**

  Pre nastavenie hodnoty väzby na požadovaný HTML atribút je dedikovaná funkcia setAttributeBindable(attribute, bindable). Tá jednoduchým spôsobom spracuje výraz prostredníctvom evalBindable(bindable) a jej výsledok vráti ako hodnotu atribútu. Okrem toho, metóda vychádza z predpokladu, že obojsmerná väzba dát môže byť využitá výhradne nad HTML atribútom. Pokiaľ sa teda jedná o väzbu nad obojsmerným atribútom, metóda rozšíri element o ovládač udalosti onChange, ktorý vyvolá aktualizačnú akciu nad stavovým kontajnerom. Spustením udalosti typu APPLY sa zmeny hodnoty prejavia aj v ďalších komponentoch závislých na aktualizovanej hodnote.

```
1  setAttributeBindable(attribute, bindable) {
2    let attributes = {};
3    attributes[attr] = this.evalBindable(bindable);
4
5    if (["value"].includes(attribute)) {
6      attributes["onChange"] = (e) => {
7        this.set(bindable.path, e.target[attribute]);
8      };
9      attributes["onBlur"] = () => {
10       this.store.dispatch({ type: "APPLY", path:
             this.buildPath(bindable.path) });
11     };
12   }
13
14   return attributes;
15 }
```

Väzby dát sa tak na základe svojho typu transformujú podobným spôsobom ako v predchádzajúcich prípadoch.

```
1 <input
2   type="text"
3   value="{value: Title}"
4 />
5 <input
6   type="text"
7   {...this.setAttributeBindable("value", this.
        bindable("Title"))}
8 >
```

Odpovedá:

```
1 <input
2   type="text"
3   value="{controlProperty: Title}"
4 />
5 <input
6   type="text"
7   {...this.setAttributeBindable("value", this.props.
        Title)}
8 >
```

- **postback(id)**

  Podobne ako v aktuálnej verzii, akcie sú spracovávané serverovou časťou aplikácie. Z tohto dôvodu je ich väzba nahradená za komunikačnú funkciu postback(id). Tá z bezpečnostných dôvodov neuvádza pravé meno metódy, ale odosiela identifikátor akcie. Po spracovaní akcie je klientskou aplikáciou prijatý nový stav a vyvolaná akcia jeho aktualizácie nad stavovým kontajnerom. V tomto prípade nevieme k akým zmenám hodnôt došlo, a preto sa konštruuje kompletný abstraktný strom DOM odznovu.

```
1 postback(id) {
2   return async() => {
3     let newState = await evaluatePostbackAsync(id,
          this.store.getState(), this.dataContext);
4     this.store.dispatch({
5       type: "POSTBACK",
6       state: newState
7     });
8   }
9 }
```

Väzby akcií sa tak jednoduchým spôsobom transformujú na volanie funkcie postback(id).

```
1 <dot:Button
2   Click="{command: AddTask()}" />
3 <DotButton
4   Click={this.postback("oVJJJoVlgziTaEbG")} />
```

Odpovedá:

```
1 <dot:Button
2   Click="{controlCommand: AddTask()}" />
3 <DotButton
4   Click={this.postback("oh+w3KnXNkAZp")} />
```

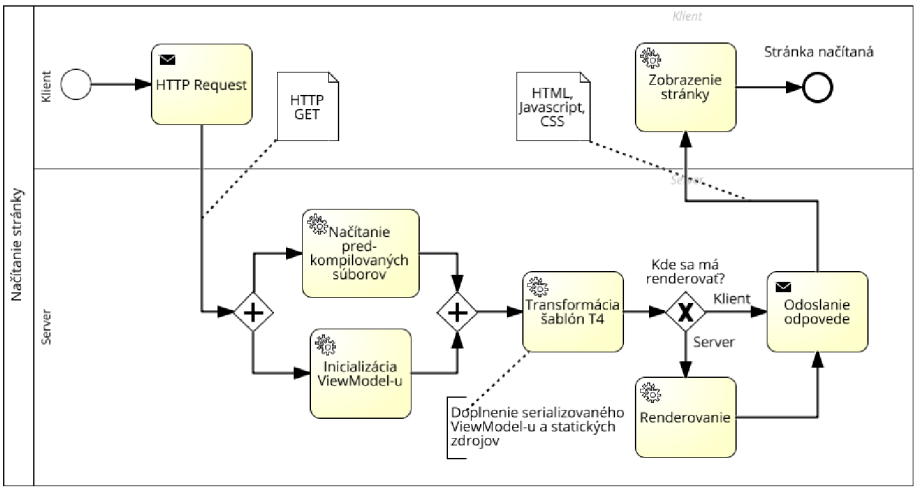
3.7 Spôsob kompilácie

Pre zvýšenie efektivity frameworku navrhujem využitie princípu kompilácie v čase zostrojenia projektu, v angličtine označovanú aj ako AOT - Ahead-of-Time. Týmto spôsobom by mohlo dôjsť k zrýchleniu spracovania požiadaviek, pretože aktuálna verzia frameworku vykonáva preklad stránok a komponentov až za behu aplikácie (JIT – Just-in-Time). Z tohto dôvodu, odporúčam pri zostrojení projektu skompilovať všetky stránky a komponenty do súborov typu HTML a JavaScript ako bolo diskutované v kapitole 3.1. Pri spracovaní požiadavky sa tak dohľadajú potrebné súbory, ktoré sa následne upravia a odošlú na klienta. Úpravou súborov sa rozumie napr. doplnenie serializovaného aplikačného stavu alebo nahradenie väzieb statických zdrojov. Takúto funkcionalitu je možné zabezpečiť nahradením dynamických častí za šablóny kódu jazyka C# a tie vykonať až pri spracovaní požiadavky. Pre tieto účely navrhujem využiť šablóny typu T4[1], ktoré špeciálnou syntaxou napovedajú JIT kompilátoru aké časti kódu majú byť vykonané a nahradené ich návratovou hodnotou.

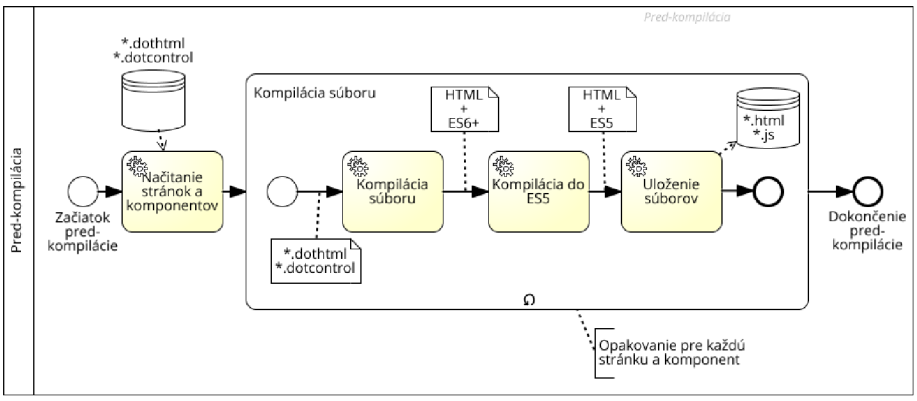
Samotná kompilácia súborov aplikačných stránok a komponentov tak prebehne postupným prechádzaním stromu DOTHTML a transformáciou ich elementov a komponentov na odpovedajúci kód rozšírenia JSX a štandardu ECMAScript 6+. Pre zaistenie kompatibility s webovými prehliadačmi sa dodatočne aplikuje transformácia

1. Dostupné z: https://msdn.microsoft.com/en-us/library/bb126445.aspx

a kompiláciu kódu do štandardu ECMAScript 5. K tomu je možné využiť konfigurovateľný kompilátor Babel[2].



Obr. 3.1: Navrhovaný spôsob načítania stránky



Obr. 3.2: Navrhovaný spôsob kompilácie stránky

2. Dostupné z: https://babeljs.io/

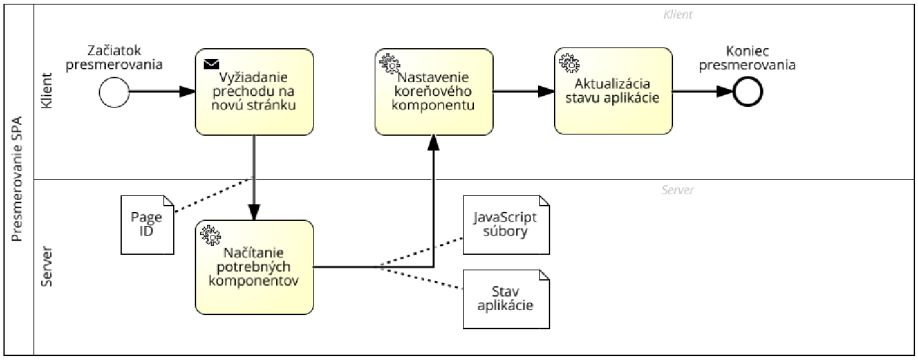
3.7.1 Renderovanie stránok na strane serveru

Za bežných okolností prebieha renderovanie aplikácií React na klientskom zariadení. V praxi ale nájdeme viacero prípadov, kedy je potrebné aplikáciu vykresliť už na serveri. Jedným z týchto prípadov je optimalizácia aplikácií pre vyhľadávacie nástroje, pretože tie nie sú schopné indexovať dynamicky vykresľovaný obsah stránky. Možnosťou zabezpečenia renderovania na strane serveru je použitie funkcií rozhrania ReactDOMServer, ktoré spracuje stránku ešte pred jej odoslaním na klienta. Okrem toho, rozhranie disponuje kontrolou pre zamedzenie druhého renderovania ihneď po načítaní stránky vo webovom prehliadači, takže sa netreba obávať že sa stránka vykreslí zbytočne dva krát.

Problém nastáva v prípade vnorenia komponentu s renderovaním na strane serveru do komponentu s renderovaním na strane klienta. Na rozdiel od aktuálneho princípu frameworku, kedy môže byť takmer ktorýkoľvek element stránky vykreslený na strane serveru, zapuzdrenie komponentov React to bohužiaľ neumožňuje. Bez vykreslenia rodičovského komponentu sa tak aj napriek renderovaniu požadovaný obsah nezobrazí. Z tohto dôvodu si myslím, že jediný stabilný a možný spôsob je serverové renderovanie kompletného obsahu stránky.

3.8 Single Page Application (SPA)

Vo frameworku DotVVM nájdeme aj podporu princípu jednostránkových aplikácií. Tie počas svojho behu nevyžadujú presmerovanie alebo obnovenie stránky, a tak môžu byť ich zmeny vykreslené plynulejšie a bez straty kontextu. Pre integráciu tohto konceptu s generovanou aplikáciou rozhrania React navrhujem prenos komponentov stránok na vyžiadanie. Po prijatí obsahu novej stránky a serializovaného stavu aplikácie sa nahradí koreňový komponent pre udalosť renderovania a aktualizuje sa hodnota stavového kontajnera. To následne vyvolá prekreslenie stránky aj bez nutnosti presmerovania alebo obnovenia.



Obr. 3.3: Navrhovaný spôsob presmerovania stránky SPA

4 Overenie správnosti navrhovaného riešenia

4.1 Implementácia prototypu

Pre overenie správnosti navrhovaného riešenia integrácie princípu VirtualDOM, bol ako súčasť diplomovej práce implementovaný základný prototyp jadra frameworku. Jeho úlohou je zabezpečenie procesu pred-kompilácie stránok DOTHTML, na ktorých bude možné overiť nielen samotnú realizovateľnosť ale aj efektivitu diskutovaného konceptu. Riešenie implementuje fundamentálne princípy a komponenty frameworku DotVVM v rozhraní React a následne ich kompiluje do spoločného súboru štandardu ES5. Okrem toho, poskytuje transformáciu všetkých DOTHTML súborov projektu na dvojice predkompilovaných dokumentov typu HTML a JavaScript. Ich preklad je založený na postupnom prechádzaní stromu DOTHTML, k čomu prototyp využíva princíp návrhového vzoru Visitor a služieb frameworku DotVVM.

Vytvorený prototyp kompilátoru obsahuje:

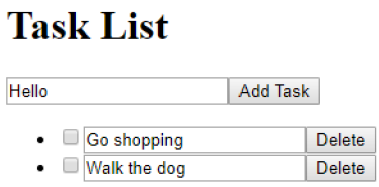
- Implementáciu stavového kontajneru a jeho akcií
- Implementáciu komponentov Literal, Button, CheckBox, TextBox a Repeater

- Automatizovaný preklad DOTHTML súborov do rozhrania React
- Transformáciu výsledného kódu do štandardu ES5

Pozn. Pre účely prototypu sa v aplikácii využíva falošná implementácia pôvodného stavu aplikácie a jej akcií, ktoré by sa v budúcnosti napojili na obsluhu požiadaviek frameworku DotVVM.

4.2 Testovanie prototypu

K testovaniu a porovnaniu prototypu bola vytvorená aplikácia pre zaznamenávanie úloh, ich editáciu alebo odstránenie. Aplikácia k tomu využíva 1 stránku, 5 základných komponentov, 1 šablónu kolekcie dát, 5 dátových väzieb a 2 väzby akcií. Výsledná aplikácia je zobrazená na nasledujúcom obrázku 4.1.



Obr. 4.1: Náhľad testovacej aplikácie prototypu

4.2.1 Veľkosť stránky a závislostí prototypu

K porovnaniu veľkosti stránky a ich závislostí bolo použité vývojárske rozhranie webového prehliadača Google Chrome. Z tabuľky 4.1 vyplýva, že minifikovaná stránka testovanej aplikácia je v rozhraní React približne o 7.6% väčšia a jej závislosti predstavujú veľkostný nárast až o 44.7%. Miernu optimalizáciu veľkosti stránky je možné docieliť efektívnejšou minifikáciou názvov parametrov a funkcií, avšak vzhľadom na množstvo implementovanej funkcionality môžeme v budúcnosti očakávať ďalší nárast závislostí frameworku. Podľa tvrdení z kapitoly 2.7 je tento nárast očakávaný a mal by byť daňou za zvýšenie výkonu.

Tabuľka 4.1: Porovnanie veľkostí obsahu stránky a závislostí prototypu

	Knockout.js [kB]	React [kB]
Obsah stránky	2.6	2.8
Závislosti frameworku	114	165
Spolu	116.6	168.7

4.2.2 Rýchlostné porovnanie prototypu

Pre zistenie výkonnosti bolo na testovanej aplikácií nameraných viacero udalostí súvisiacich s načítaním stránky a dynamickými zmenami jej hodnôt. Všetky merania boli samostatne zaznamenávané vo vývojárskom prostredí webového prehliadača Google Chrome a môžu byť mierne skreslené zaťažením procesora, operačnej pamäte a podobne. Pre zistenie vlastností týchto konceptov u stránok s odlišnou komplexnosťou, boli merania opakované pre stav aplikácie s dvomi položkami kolekcie a s tisíc položkami kolekcie. Rýchlosť procesov je v tabuľkách 4.2 a 4.3 vyjadrená sumou časových hodnôt načítania, renderovania, vykresľovania a iných bližšie nešpecifikovaných udalostí testovanej stránky.

Tabuľka 4.2: Porovnanie výkonnosti aplikačnej stránky s 2 položkami zoznamu

	Knockout.js [ms]	React [ms]	Rozdiel [%]
Zobrazenie stránky	288.3	121.7	**57.8**
Pridanie položky na koniec zoznamu	79.9	65.1	**18.5**
Odstránenie položky z konca zoznamu	71.3	66.1	**7.3**
Pridanie položky do stredu zoznamu	85.9	71.5	**16.7**
Odstránenie položky zo stredu zoznamu	53.9	43.6	**21**
Aktualizácia elementu s obojsmernou väzbou dát	69.6	25.4	**63.5**

Tabuľka 4.3: Porovnanie výkonnosti aplikačnej stránky s 1000 položkami zoznamu

	Knockout.js [ms]	React [ms]	Rozdiel [%]
Zobrazenie stránky	1824.6	730.8	**60**
Pridanie položky na koniec zoznamu	1949.2	245.1	**87.4**
Odstránenie položky z konca zoznamu	1434.9	83.5	**94.2**
Pridanie položky do stredu zoznamu	1260.8	276.8	**78**
Odstránenie položky zo stredu zoznamu	1271.5	663.7	**67.8**
Aktualizácia elementu s obojsmernou väzbou dát	188.2	99.9	**47**

V oboch prípadoch načítania stránok bolo zaznamenané takmer 60% celkové zrýchlenie procesu, čo by aj napriek väčšiemu objemu prenášaných dát predstavovalo výrazné zlepšenie výkonnosti oproti aktuálnemu riešeniu. Ďalšou silnou stránkou prototypu, vyplývajúcou z nameraných hodnôt je priamo-úmerný nárast rýchlosti zmien kolekcií dát. Tu sa v závislosti od typu operácie pohybuje nárast rýchlosti od 7% do 21% pre zoznam s 2 položkami a 68% až 94% pre zoznam s 1000 položkami. U jednoduchšej stránky bolo pri interakcii s komponentami zabezpečujúcimi obojsmernú väzbu dát namerané zrýchlenie procesu o 63.5% a 47% u komplexnejšej stránky.

Záver

Cieľom diplomovej práce bola analýza realizovateľnosti a možného prínosu nahradenia transformácie klientskej vrstvy frameworku DotVVM za modernejšiu technológiu založenú na princípe Virtual DOM. Práca v úvode oboznamuje čitateľa s diskutovaným frameworkom DotVVM a pokračuje objasnením významu a použitia programovacieho jazyka JavaScript pre tvorbu dynamických grafických rozhraní. Tu sú analyzované dva odlišné koncepty pre prácu s rozhraním DOM a zhrnuté základné poznatky o knižniciach Knockout.js a React. Porovnaním oboch knižníc boli identifikované výhody a nevýhody týchto riešení, z ktorých vyplýva, že použitie návrhového vzoru Observer je síce jednoduchšie, ale princíp Virtual DOM ponúka rýchlejšie zmeny komplexnejších webových stránok.

Praktická časť práce analyzuje fundamentálne princípy frameworku a navrhuje spôsob ich prenesenia a automatizácie do rozhrania knižnice React. Analýza bola využitá pre implementáciu prototypu jadra frameworku a zistenie realizovateľnosti a možného prínosu aktualizácie. Z meraní uskutočnených na testovanej aplikácií, vygenerovanej prototypom jadra frameworku, bolo zaznamenané výrazne zvýšenie výkonu u všetkých pozorovaných dynamických zmien aplikácie. Pre komplexnejšiu stránku s 1000 položkami kolekcie bolo zistené až 60% zrýchlenie procesu načítania stránky, 47% percentné zrýchlenie procesu obojsmerného toku dát a 78 - 94% zrýchlenie u rôznych procesov manipulujúcich s takto veľkou kolekciou dát.

Z práce vyplýva, že aktualizácia klientskej vrstvy frameworku je realizovateľná a mohla by znamenať signifikantný nárast výkonu. Väčšinu fundamentálnych princípov je naviac možné implementovať so zachovaním spätnej kompatibility, a tak modernizovať aj súčasné aplikácie vo frameworku DotVVM. Tu je však nutné zohľadniť zvýšené nároky na objem prenášaných dát, zaťaženia operačnej pamäte klientskeho zariadenia a rozsiahlu reimplementáciu aktuálnej verzie frameworku.

Bibliografia

1. STANGARONE, Joe. 7 principles of modern web application development [online] [cit. 2016-08-02]. Dostupné z: https://www.mrc-productivity.com/blog/2016/08/7-principles-of-modern-web-application-development/.
2. STORMPATH. Has ASP.NET Core Killed Web Forms? [online] [cit. 2016-10-24]. Dostupné z: https://stormpath.com/blog/dotnet-core-killed-web-forms.
3. HERCEG, Tomáš. *DotVVM: "Javascript" Apps With No Javascript* [Channel9.msdn.com. [online]. [cit. 2018-04-16]] [Dostupné z: https://channel9.msdn.com/Series/NET-DeveloperDays-2015-on-demand/dotVVM-Javascript-Apps-With-No-Javascript-Tomas-Herceg].
4. GAROFALO, Raffaele. *Building Enterprise Applications with Windows® Presentation Foundation and the Model View ViewModel Pattern*. 1005 Gravenstein Highway North Sebastopol, California 95472: O'Reilly Media, Inc., 2011. ISBN 978-0-735-65092-3.
5. CLEEREN, Gill. *Practical MVVM* [Pluralsight.com. [online]. [cit. 2018-04-18]] [Dostupné z: https://app.pluralsight.com/player?course=practical-mvvm].
6. ANTANI, Ved. *Mastering JavaScript: Explore and master modern JavaScript techniques in order to build large-scale web applications*. 35 Livery Street Birmingham B3 2PB, UK: Packt Publishing Ltd., 2016. ISBN 978-1-78528-134-1.
7. SIMPSON, Kyle. *You Don't Know JS: ES6 & Beyond*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2015. ISBN 978-1-491-90424-4.
8. NEOTERIC. Single-page application vs. multiple-page application [online] [cit. 2016-12-02]. Dostupné z: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.
9. MUNRO, Jamie. *Knockout.js: Building Dynamic Client-Side Web Applications*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2014. ISBN 978-1-491-91431-1.

10. BISHOP, Judith. *C# 3.0 Design Patterns*. 1005 Gravenstein Highway North Sebastopol, California 95472: O'Reilly Media, Inc., 2008. ISBN 978-0-596-52773-0.

11. KRAJKA, Bartosz. The difference between Virtual DOM and DOM [online] [cit. 2015-10-12]. Dostupné z: `http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/`.

12. GARSIEL T., Irish P. How Browsers Work: Behind the scenes of modern web browsers [online] [cit. 2011-08-05]. Dostupné z: `https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#The_rendering_engine`.

13. ACADEMY, Fullstack. *Tech Talk: What is the Virtual DOM?* [Youtube.com. [online]. [cit. 2018-04-18]] [Dostupné z: `https://www.youtube.com/watch?v=d7pyEDqBDeE`] [cit. 2017-09-28].

14. BANKS A., Porcello E. *Learning React: Functional Web Development with React and Redux*. 1005 Gravenstein Highway North Sebastopol, California 95472: O'Reilly Media, Inc., 2017. ISBN 978-0-596-52773-0.

15. MCLENNAN, Liam. *React Fundamentals* [Pluralsight.com. [online]. [cit. 2018-04-22]] [Dostupné z: `https://app.pluralsight.com/library/courses/react-fundamentals/table-of-contents`] [cit. 2017-09-28].

16. A., Sharif. Understanding the React Component Lifecycle [online] [cit. 2015-03-21]. Dostupné z: `http://busypeoples.github.io/post/react-component-lifecycle/`.