
**Střední průmyslová škola a Vyšší odborná škola, Písek,**
**Karla Čapka 402**
**397 11 Písek**

**Školní rok: 2020/2021**

## Obor vzdělání: 18-20-M/01 Informační technologie

# Maturitní práce

## Reaktivní HTML elementy

Téma číslo: 32

Jméno žáka: Ludvík Prokopec

Třída: **B4.I**

Vedoucí práce: Mgr. Milan Průdek

## Anotace

HTML kód webové stránky prohlížeč převádí do objektového modelu jazyka Javascript, který jeho manipulací zpětně provádí úpravy stránky. Tyto úpravy se často kaskádovitě šíří komplexním modelem, což snižuje výkon stránky. To je částečně kompenzováno množstvím optimalizací, díky nimž není většina objektů *reaktivních*. To znamená, že změny HTML stránky se nepropíší přímo do Javascriptových proměnných, které je nutné aktualizovat explicitně. Tyto nevýhody eliminuje Kiq.js vytvořením reaktivního proxy modelu.



**Klíčová slova:** Javascript, deklarativní programování, reaktivita, virtuální DOM



## Annotation

The browser converts the HTML code of the web page into a Javascript object model, which manipulates the page to reverse the page. These adjustments often cascade across the complex model, reducing page performance. This is partially offset by a number of optimizations, which make most objects inactive. This means that changes to an HTML page are not directly reflected in Javascript variables, which must be updated explicitly. Kiq.js eliminates these disadvantages by creating a reactive proxy model.



**Keywords:** Javascript, declarative programming, reactivity, virtual DOM

## Poděkování

Chtěl bych poděkovat vedoucímu dlouhodobé maturitní práce panu učiteli Mgr. Milanovi Průdkovi za skvělé vedení, za pomoc, ochotu a cenné rady při zpracování maturitní práce. Děkuji panu Jáchymovi Janouškovi za pomoc, rady, informace a ochotu při zpracování práce a také panu Mgr. Janu Turoňovi za rady, pomoc, testování a ochotu při tvorbě maturitní práce.

# Obsah

# 1 Úvod

## 1.1 Problematika

V dlouhodobé maturitní práci se zabývám problematikou zjednodušení tvorby webových stránek pomocí knihovny, která nahradí imperativní design manipulace s elementy v Javascriptu deklarativním.

## 1.2 Řešení problému

Řešením problému je vytvoření knihovny, která zajišťuje automatickou reakci DOM elementů na data. Díky tomu změna dat zároveň manipuluje s elementy. Vykreslování díky optimalizacím může být i rychlejší než vykreslování bez těchto optimalizací. Díky knihovně je možné vytvořit komponenty, které jsou takovými znovu použitelnými moduly v aplikaci podobně jako funkce.

## 1.3 Inspirace

Největší inspirací mi byly ostatní knihovny řešící tuto problematiku. Hlavně pak React.js od Facebooku a dále video od Jason Yu, který napsal a vysvětlil základní informace o virtuálním DOM ve 45 minutách.

## 1.4 Průběh řešení práce

Cíle bylo dosaženo pomocí vyhledávání informací na internetu, testování chyb mé knihovny a testování ostatních knihoven pro zjištění jejich nedostatků. V neposlední řadě také díky konzultacím s panem Jáchymem Janouškem a Mgr. Janem Turoněm.

## 1.5 Výsledek práce

Výsledkem práce je funkční, velice rychlá a malá Javascript knihovna zajišťující reaktivitu DOM elementů. Využívá virtuální DOM pro zjišťování změn s následným aplikováním změn do DOM elementů. Knihovna je také nahraná na Githubu, na NPM a má CDN odkaz, díky kterému je jednoduché knihovnu použít.

# 2 Terminologie

**DOM (Document Object Model)**

DOM je v javascriptový objekt. Každý objekt může mít vlastnosti a metody. U DOM objektu je tomu stejně. Vlastnosti popisují DOM objekt a metody slouží pro manipulaci. V dokumentaci Javascriptu není DOM popsán, protože DOM dokumentaci spravuje jiná organizace a ta je pro všechny programovací jazyky stejná.

Díky stromové struktuře dokumentu dovedeme rozlišit nadřazené, podřazené nebo rovnocenné elementy.[1]

**CDN (Content Delivery Network)**

Síť pro doručování obsahu je síť počítačů vzájemně propojených skrze internet, která zvyšuje dostupnost dat uživatelům.[2]

**Reaktivita**

Reaktivní programování je model programování orientovaný kolem datových toků a šíření změn. To znamená, že je možné vyjádřit statické nebo dynamické datové toky v programovacích jazycích jednoduše a že základní provedení modelu bude automaticky kopírovat změny.[3]

Názornou ukázkou reaktivity je tabulka v programu MS Excel. Pokud v tabulce změníme nějakou hodnotu, hodnoty na ní závislé se automaticky aktualizují.

**Modul**

Modulární programování je technika návrhu softwaru, kdy každý modul je samostatně funkční, to jest nezávislý, a zaměnitelný.

Jednotlivé moduly jsou odděleny samostatnou zodpovědností a zlepšují udržovatelnost softwaru přímým vyjádřením logických hranic mezi komponenty. Při vytváření většího množství softwarových projektů umožňující moduly vytvořené v jednom projektu použití i v jiných projektech.[4]

**Reflow**

Reflow je akce, při které prohlížeč musí zpracovat a nakreslit nebo překreslit část nebo celou webovou stránku. Jedná se především o přepočítávání pozic elementů a jejich rozložení na stránce.[5]

**Repaint**

Repaint je akce, při které prohlížeč vyvolá překreslení. Provádí změny ve vzhledu elementů a neovlivní rozložení na stránce.[6]

**JSX**

JSX neboli Javascript XML je rozšíření syntaxe jazyka Javascript. JSX má podobnou syntaxi jako HTML. Poskytuje způsob, jak strukturovat vykreslování komponent pomocí syntaxe HTML.[7]

**Transpiler**

Transpiler je překladač. Překládá zdrojový kód z jednoho programovacího jazyka do jiného.

Transpiler pracuje s jazyky na přibližně stejné úrovni abstrakce, zatímco tradiční kompilátor kompiluje jazyk na vysoké úrovni abstrakce do jazyka na nízké úrovni abstrakce.[8]

Nejpoužívanější transpilery pro Javascript jsou například Babel.js nebo TypeScript.

**Deklarativní programování**

Deklarativní programování je založeno na myšlence programování pomocí definic, tedy co se má udělat, a ne jak se to má udělat.[9]

**Imperativní programování**

Imperativní programování popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit.[10]

**Anti-pattern**

Návrhový anti-vzor představuje obecný postup při řešení opakujících se problémů při návrhu počítačových programů, který je všeobecně vnímán jako nesprávný nebo neefektivní.[11]

**Batch algoritmus**

Prohlížeč se již pokouší minimalizovat opakované spuštění *reflow*, takže pokud provedete více úprav DOM v jedné po sobě jdoucí části Javascriptu, prohlížeč počká na dokončení této části Javascript kódu a poté provede jedno přepočítávání pozic a jedno překreslení.[12]

Tímto algoritmem zvýšíme rychlost všech aktualizací DOM, které jsou při hledání změn ve virtuálním stromě zaznamenány a je vytvořena jedna funkce, která spouští aktualizace pro celý strom DOM elementů.

**BackTracing**

Algoritmus zpětného vyhledávání je založen na hledání hrubou silou procházením stromové struktury a zpětného vyhledávání rodičovských uzlů. Nejdřív jsou procházeny potomci a poté je vyhledávání vráceno zpět na rodičovský uzel.[13]

**DFS algoritmus**

Prohledávání do hloubky je grafový algoritmus pro procházení grafů metodou backtrackingu. Pracuje tak, že vždy expanduje prvního následníka každého vrcholu, pokud jej ještě nenavštívil. Pokud narazí na vrchol, z nějž už nelze dále pokračovat (nemá žádné následníky nebo byli všichni navštíveni), vrací se zpět backtrackingem.[14]

### RequestAnimationFrame

RequestAnimationFrame je funkce, která nám zajistí, že se všechny DOM operace budou vykreslovat najednou, s vyšším výkonem a nižší spotřebou baterie.

Pokud v prohlížeči překliknete na jinou stránku, nebo prohlížeč minimalizujete, vykreslování se zastaví, aby se šetřil výkon. Bude se pokračovat, jakmile bude stránka opět viditelná.[15]

Tato funkce je primárně určena pro použití při animacích. V případě Kiq.js je použita při prvním vykreslení DOM elementů, protože se tato funkce spouští těsně před tím, než prohlížeč začne přepočítávat pozice a styly elementů. Nedochází tak k nadbytečnému přepočítávání těchto údajů, které musí prohlížeč zpracovat. Tímto je v knihovně zajištěno rychlejší načtení stránky, protože tento kód není kódem, který při startu stránky začne blokovat hlavní (a jediné) vlákno procesoru, se kterým Javascript pracuje.

### Array.map

Tato metoda projde pole, u kterého je spuštěna a pro každý prvek zavolá funkci, která je parametrem metody *map*. Tato funkce může vrátit nějaký výsledek, který je potom zapsán do výsledného pole.

### Literal template string

Typ řetězce umožňující snadné vložení výrazů a vytváření víceřádkových řetězců.[16]

Označují se zpětnými uvozovkami.

### Tagged templates

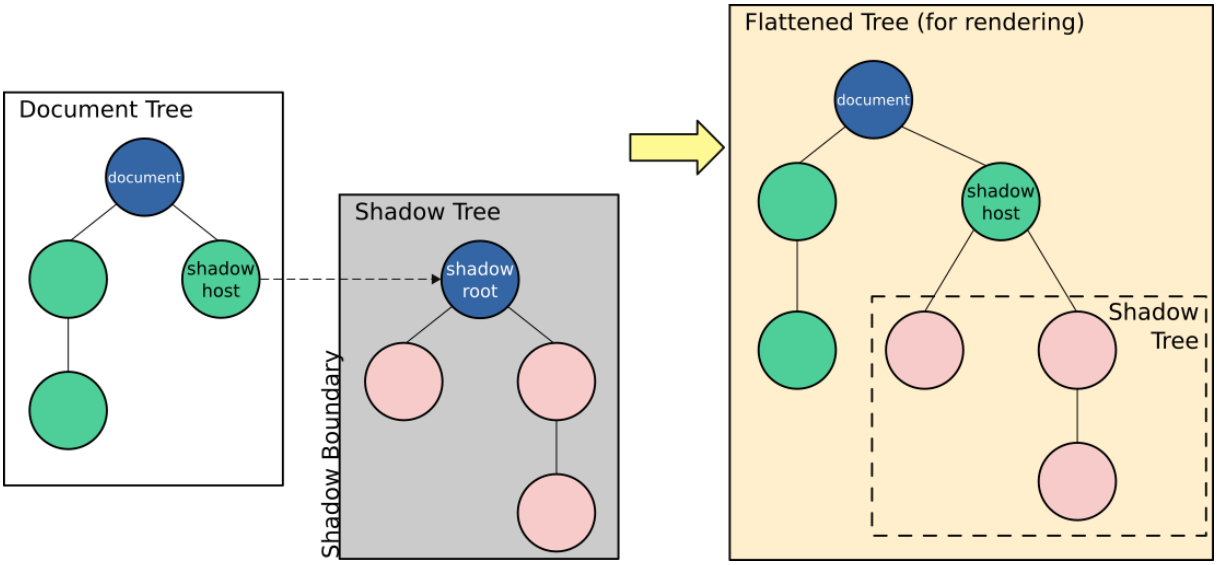
Tagy umožňují analyzovat (parsovat) literály šablon pomocí funkce. První argument funkce tagu obsahuje pole řetězcových hodnot. Zbývající argumenty se vztahují k výrazům.[17]

### Webové komponenty

Jsou sadou různých technologií, které umožňují vytvářet opakovaně použitelné vlastní elementy s funkcemi oddělenými od zbytku kódu.[18]

### Shadow DOM

Důležitým aspektem webových komponent je zapouzdření. Jedná se o schopnost udržet strukturu označení, styl a chování, skryté a oddělené od ostatního kódu na stránce, aby se jednotlivé části nesetkaly. Klíčovou součástí je rozhraní Shadow DOM API, které poskytuje způsob připojení skrytého odděleného DOM k elementu.[19]




*Obrázek 1 - Struktura Shadow DOM*

- Shadow host
  - DOM uzel, ke kterému je připojen Shadow DOM
- Shadow root
  - kořenový element Shadow DOM
- Shadow tree
  - strom DOM elementů uvnitř Shadow DOM
- Shadow boundary
  - místo, kde končí Shadow DOM a navazuje na něj klasický DOM

Podobně fungují i komponenty v Kiq.js, ale jsou pouze logickou definicí elementu a jeho dat, nikoliv zapouzdřením DOM elementů nebo CSS stylů, jako je tomu u Shadow DOM a web komponent.

### NPM

NPM je správce balíčků pro Javascript, výchozí správce balíčků pro prostředí Node.js.[20]

**Github**

GitHub je webová služba podporující vývoj softwaru za pomoci verzovacího nástroje Git.[21]

# 3 Virtuální DOM

Jedná se o obyčejný Javascript objekt, který slouží jako top-level klasického DOM objektu. Je to jakási zjednodušená kopie DOM objektu. Informace obsažené ve virtuálním DOM jsou popsány níže. Tyto informace jsou ve virtuálním DOM dostačující, protože většina informací v DOM objektu je redundantní (dá se jich dosáhnout více cestami).

Virtuální DOM slouží pro porovnávání dvou virtuálních stromů. Díky tomu můžeme manipulovat s virtuálním DOM jako s polem nebo řetězcem, kdy tyto změny nezpůsobují překreslení stránky a jsou tedy mnohem rychlejší než změny DOM elementů. Tyto změny jsou samozřejmě také později překresleny, ale za pomocí různých optimalizací.

## 3.1 Zjednodušená struktura virtuálního DOM

- **Type**
  - element
    - jméno tagu
  - komponenta
    - třída komponenty
- **Props**
  - element
    - vlastnosti DOM objektu
  - komponenta
    - Data komponenty, které přicházejí zvenčí.
- **Children**
  - element
    - Pole, ve kterém jsou všechny potomci daného virtuálního elementu.
  - komponenta
    - Pole, ve kterém jsou všechny potomci, ty jsou ovšem zapsány do dat komponenty *props,* odkud se dají získat pomocí `props.children`.

```
const virtualNode = {
    type: "div",
    props: {

        className: "container",
        id: "container1"

    },
    children: [
        {
            type: "h1",
            props: null,
            children: [
                "Hello, world!"
            ]
        }
    ]
};
```

Toto je jednoduchá struktura virtuálního elementu, který odpovídá následujícímu DOM elementu.

```
<div class="container" id="container1">
    <h1>Hello, world!</h1>
</div>
```

Při psaní kódu se nikdy nepoužije tato nepřehledná syntaxe zapsání virtuálního elementu, ale syntaxe pomocí speciální funkce knihovny:

```
const virtualNode = Kiq.createElement("div", {
        className: "container",
        id: "container1"
    },
    Kiq.createElement("h1", null, "Hello, world!")
);
```

Vytvoření virtuálního DOM pomocí knihovny Kiq.js bez použití jiných nástrojů.

```
/** @jsx Kiq.createElement */

const virtualNode = (
    <div className="container" id="container1">
        <h1>Hello, world!</h1>
    </div>
);
```

Vytvoření virtuálního DOM pomocí knihovny Kiq.js s použitím Babel.js transpileru.

```
const html = htm.bind(Kiq.createElement);

const virtualNode = (
    html`<div className="container" id="container1">
        <h1>Hello, world!</h1>
    </div>`
);
```

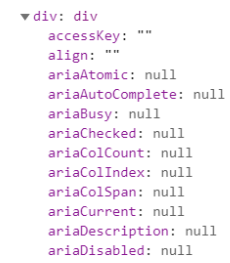
Vytvoření virtuálního DOM pomocí knihovny Kiq.js s použitím knihovny Htm.js.

## 3.2 Atributy virtuálního DOM

Atributy virtuálního DOM nejsou stejné jako atributy DOM elementu. Atributy virtuálního DOM reprezentují výsledné vlastnosti DOM elementu, který je později z virtuálního DOM vytvořený. Atributy nelze specifikovat přímo ve virtuálním DOM, dají se ovšem nastavit pomocí DOM API v Javascriptu. Můžeme tedy nastavit například třídu elementu pomocí *className* nebo nastavit zdroj obrázku pomocí *src* nebo styl elementu pomocí *style*. Nelze však nastavit přímo vlastní atribut, který by měl například jméno „attr“. Pro vytvoření atributu „attr“ u DOM elementu je potřeba přidat referenci k virtuálnímu DOM, která je popsána v kapitole Reference, a nastavit atribut pomocí `Node.setAttribute`.

### 3.2.1 Vlastnosti DOM elementů

Objekt DOM má své vlastnosti jako každý jiný objekt. Lze je nastavit pomocí klíčů a hodnot. Například nastavení `Node.className` vlastnosti je stejné jako nastavení atributu *class*.




*Obrázek 2 - Vlastnosti DOM elementu*

Většina vlastností DOM jsou po jejich nastavení odraženy v atributech výsledného elementu. Většina atributů odražena ve vlastnostech není, a to je důvod, proč knihovna používá nastavování vlastností DOM objektů namísto nastavování atributů. Bylo by možné nastavovat obojí, ale to by knihovnu zpomalilo, z tohoto důvodu je nastavování samotných atributů vynecháno a musí se nastavit manuálně.

## 3.3 Eventy virtuálního DOM

Eventy jsou zapisovány stejně jako atributy, ale ve výsledném elementu jsou potom přidány pomocí funkce `Node.addEventListener`, tzn. že nejsou u elementu odraženy v atributech ani ve vlastnostech výsledného DOM elementu. Každý event je přidán, odebrán či změněn automaticky podle knihovny.

```
/** @jsx Kiq.createElement */

const virtualNode = (
    <button onclick={ (e) => console.log('clicked') }>Click</button>
);
```

Na každé kliknutí tlačítka se do konzole vypíše „clicked“.

Každý event musí začínat na „on“ a následuje jméno příslušné události, který chceme volat. Eventy mohou být i vlastní (custom), které samotný Javascript neobsahuje, například „onswipe“.

## 3.4 Klíče

Klíče jsou u virtuálního elementu speciální atributy. Jsou použity pro jednoznačnou identifikaci virtuálního elementu, přičemž každý klíč mezi sourozenci musí být unikátní a trvalý po celou dobu chodu aplikace, podobně jako ID v databázi. Tento atribut není promítnut v DOM elementu.

Přiřazení klíče k virtuálnímu elementu je pomocí atributu **_key**.

Knihovna na základě klíče určuje aktualizace DOM elementů.

Klíče jsou většinou používány ve vytváření seznamů pomocí pole, kdy není jednoznačné, jak se má DOM element aktualizovat. Například přidáním prvku na začátek pole by byl přiřazen starý virtuální element a nový přiřazen špatně a docházelo by k redundantnímu překreslení či změně. Použitím klíčů můžeme tedy zamezit redundantním akcím, a tím i zvýšit výkon celé aplikace.

Jako klíč by se neměl používat index při procházení prvků v poli, protože indexy se při manipulaci s polem a následném novém projetí oproti starým indexům předešlého pole mění a nejsou tedy trvalými klíči.

Klíče nemusí být použity v konstantních listech, které se nemění.

Klíč by měl být pouze primitivního datového typu (řetězec, číslo). Klíče jsou vždy knihovnou převedeny do řetězce, to znamená, že klíč s číslem „1“ a řetězcem „1“ jsou identické.

```
/** @jsx Kiq.createElement */

const arr = [1, 2, 3, 4];

const virtualNodes = arr.map(item => <div>{ item }</div>);
```

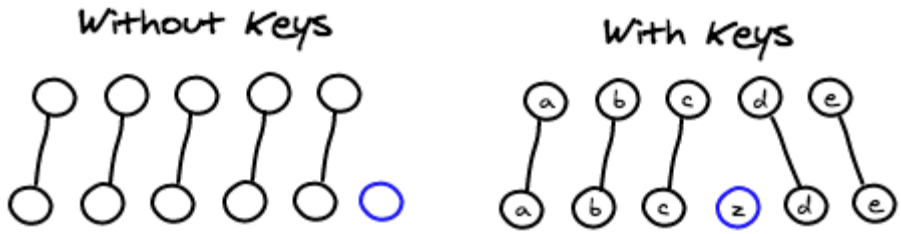
List bez použití _key.

```
/** @jsx Kiq.createElement */

const arr = [1, 2, 3, 4];

const virtualNodes = arr.map(item => <div _key={ item }>{ item }</div>);
```

List s použitím _key.




*Obrázek 3 - Porovnání párování elementů bez a s klíči[22]*

## 3.5 Reference

Reference jsou podobně jako klíče také speciální atributy, které očekávají funkci. Ta se spustí, jakmile je daný virtuální DOM vykreslený (sestavený DOM element z virtuálního DOM).

Získání reference přiřazené k virtuálnímu DOM je pomocí atributu **_ref**.

Reference slouží ke speciálním manipulacím s DOM elementem, které s virtuálním DOM nejsou možné, například jako metody `Node.click()` nebo `Node.focus()`.

Reference také mohou sloužit ke komunikaci Kiq.js s jinou knihovnou.

Reference by se ale neměli nadužívat. Manipulace, které provádí virtuálním DOM, by neměly být provedeny nativním způsobem.

Reference mohou také sloužit pro nastavování atributů, které nejdou nastavit pomocí vlastností DOM elementů.

Pokud reference změní DOM element „nelegálně“ (viz kap. Problém ruční manipulace), který odpovídá virtuálnímu DOM, kód se zhroutí. V tomto momentu se nemusí aplikace vrátit do funkčního stavu. Reference tedy nesmí ovlivňovat DOM elementy, se kterými manipuluje knihovna.

```
/** @jsx Kiq.createElement */

const virtualNode = (
    <div _ref={ el => console.log(el) }>Hello, world</div>
);
```

Tento příklad po vykreslení virtuálního DOM napíše do konzole element `div`.

## 3.6 Komponenty ve virtuálním DOM

Každá komponenta je vytvořena stejně jako virtuální element, ale místo jména tagu je třída komponenty a místo atributů jsou props (viz kap. Data komponenty). Při použití komponenty s potomky, jsou potomci uloženy ve speciálním objektu `props.children`. Jméno každé komponenty při použití JSX musí začínat velkým písmenem, aby transpiler rozlišil virtuální elementy a komponenty.

Protože komponenty jsou třídy, musí v aplikaci docházet k vytváření instancí těchto tříd. Vytváření instancí zajišťuje samotná knihovna, a to v procesu renderování virtuálního stromu. Instance komponent by nikdy neměla být vytvořena uživatelem knihovny.

```
/** @jsx Kiq.createElement */

const virtualNode = (
    <MyComponent date={ new Date() }/>
);
```

Ukázka aplikování komponenty bez použití `props.children`.

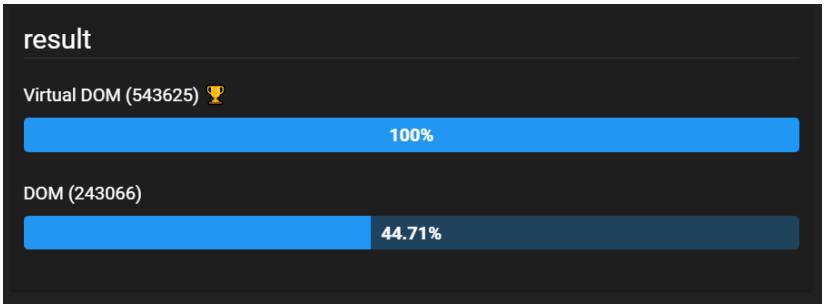
```
/** @jsx Kiq.createElement */

const virtualNode = (
    <MyComponent>{ new Date() }</MyComponent>
);
```

Ukázka aplikování komponenty s použitím `props.children`.

## 3.7 Rychlost virtuálního DOM

Používáním virtuálního DOM dvojnásobně zrychlujeme celou aplikaci, protože virtuální stromy se vytvářejí mnohem rychleji než DOM objekty.




*Obrázek 4 - Ukázka rychlosti virtuálního DOM na benchmarku*

Je to proto, že manipulace s DOM elementy a jejich tvorba je mnohem složitější operace než například změna řetězce. Změny v DOM způsobují reflow a repaint, změny virtuálního DOM nikoliv.

Manipulace s DOM elementy a jejich tvorba samozřejmě probíhá i v knihovně, ale je optimalizovaná a nikdy není redundantní.

# 4 Podmíněné renderování

Podmíněné renderování je typ vytvoření šablony s podmínkou.

V knihovně je podmíněné renderování velice jednoduché. Pro vytvoření podmínky se dá použít jednoduše *if, else, switch*, *ternary operator* nebo pomocí logického *AND* operátoru. Podmíněné renderování je stejné jako jakákoliv podmínka v Javascriptu.

```
/** @jsx Kiq.createElement */

let virtualNode;
let something = true;

if(something) {

    virtualNode = <div>true</div>;

} else {

    virtualNode = <div>false</div>;

}
```

Ukázka podmíněného vykreslení pomocí *if-else.*

```
/** @jsx Kiq.createElement */

let something = true;
let virtualNode;

switch(something) {

    case true:

        virtualNode = <div>true</div>;

    break;

    default:

        virtualNode = <div>something else</div>;

    break;

}
```

Ukázka podmíněného vykreslení pomocí *switch.*

```
/** @jsx Kiq.createElement */

let something = true;

const virtualNode = (something === true ?
    <div>true</div> :
    <div>something else</div>
);
```

Ukázka podmíněného vykreslení pomocí *ternary* operátoru.

```
/** @jsx Kiq.createElement */

const toShow = undefined;

const virtualNode = (
    <div>
        { toShow && <div>{ toShow }</div> }
    </div>
);
```

Ukázka podmíněného vykreslení pomocí logického and.

Každá podmínka vytvořená v komponentě je samozřejmě reaktivní. Při změně dat se může změnit i podmínka, tím může dojít k překreslení stránky, pokud se virtuální DOM změní.

# 5 Renderování seznamu

Renderování seznamů je v knihovně také velice jednoduché. Lze ho dosáhnout několika způsoby, stačí mít pole s virtuálními elementy. Nejsnazší způsob vytvoření seznamu je pomocí metody `Array.map`, která je vysvětlena ve stejnojmenné kapitole.

```
const arr = [1, 2, 3, 4];

const double = arr.map(item => item * 2);
//[2, 4, 6, 8]
```

Stejně jako je ukázáno na příkladu může fungovat i renderování seznamu v knihovně.

```
/** @jsx Kiq.createElement */

const arr = [1, 2, 3, 4];

const double = arr.map(item => <div _key={ item }>{ item }</div>);
//[<div>1</div>, <div>2</div>, <div>3</div>, <div>4</div>]
```

Toto je jedna z mnoha možností, je nejjednodušší na implementaci a je nejpřehlednější, protože ji lze použít přímo ve vytváření virtuálního stromu.

```
/** @jsx Kiq.createElement */

const arr = [1, 2, 3, 4];

const virtualNode = (
    <ul>
        { arr.map(item => <li _key={ item }>{ item }</li>) }
    </ul>
);
```

Celé toto pole je opět reaktivní. Pokud přidáme prvek do pole, odrazí se to i na DOM elementech na stránce, přičemž s ostatními DOM elementy se žádná akce nestane.

Při používání listu by měly být použity klíče, které již byly popsány.

# 6 Komponenty

Celá knihovna je založena na vytváření komponent a rozdělování kódu na menší části (moduly), které se dají znovu použít.

Pomocí komponent je mnohem jednodušší hledání a opravování chyb. Komponenty zajišťují správu jednotlivých virtuálních elementů, dat a rozdělení virtuálních elementů na menší části. Jakákoliv komponenta může být rodičovskou komponentou jakékoliv jiné. Komponenta je třída, která dědí ze speciální třídy z knihovny `Kiq.Component`.

## 6.1 Data komponenty

Každá komponenta může manipulovat s daty a každá komponenta může mít data dvojího typu:

- **Props**
  - Objekt, který do komponenty vstupuje zvenčí, slouží podobně jako parametry u funkce.
  - Tento objekt by měl být určen pouze pro čtení dat a může sloužit například pro přenos dat z rodičovské komponenty do komponenty, která je potomkem.
  - Na změnu tohoto objektu komponenta nijak nereaguje, není taková možnost. Docházelo by k narušení definice toho, k čemu *props* slouží a jednalo by se o tzv. *anti-pattern*.
- **State**
  - Objekt, který je soukromý a interní v každé komponentě.
  - Na změnu tohoto objektu elementy reagují překreslením podle změněných dat. Změněná data by měla být kopií dat předešlých, kvůli možnosti použití různých metod komponenty.
  - *State* mohou získávat data z *props*.
  - Ovládat *state* mohou i komponenty sloužící jako potomci, a to tak, že *state* vložíme do *props* komponenty uvnitř.

## 6.2 Virtuální element komponenty

Každá komponenta musí mít vytvořený virtuální element, který komponenta reprezentuje. Virtuální DOM se zapisuje do speciální metody Element.

```
/** @jsx Kiq.createElement */

class MyFirstComponent extends Kiq.Component {

    state = {
        count: 0
    };

    Element(props, state) {

        return <div>{ state.count }</div>;

    }
}
```

Ukázka jednoduché komponenty.

## 6.3 Reaktivita komponenty

Reaktivitu spouští speciální metoda komponenty `Component.setState`, ve které se dá nastavit více dat najednou, a která spustí proces zjišťování změn mezi starým (předešlým) virtuálním stromem a mezi novým stromem (vytvořeným po nastavení nových *state*).

`Component.setState` může být spuštěn pouze pokud je komponenta vykreslena. Kód, který se spouští přímo v těchto momentech je jednoduché vytvořit pomocí životních cyklů (*lifecycles*) komponenty, které jsou zmíněny níže.

```
/** @jsx Kiq.createElement */

class Counter extends Kiq.Component {

    state = {
        count: 0
    };

    Element(props, state) {

        return (
            <button
                onclick={ (e) => this.setState({ count: state.count + 1 }) }>
                { state.count }
            </button>
        );

    }

}
```

Jednoduchá komponenta vytvoří tlačítko. Když se na něj klikne, změní číslo jak v datech, tak i text uvnitř tlačítka. Přičemž není vytvořený žádný další *node*, na rozdíl například od nastavování `Node.innerHTML` tlačítka, kdy jsou vytvořeny na každé kliknutí dva *nody*.

Virtuální element zareaguje na jakoukoliv změnu *state*, uvnitř *state* objektu mohou být data jakéhokoliv typu.

Pokud provádíme více změn dat najednou, měla by se `Component.setState` funkce spouštět pouze jednou. To znamená nastavit všechna data společně. Pak se spustí proces hledání změn pouze jednou a dojde pouze k jednomu překreslení komponenty.

```
/** @jsx Kiq.createElement */

class List extends Kiq.Component {

    state = {
        arr: ["milk", "banana", "butter"]
    };

    onComponentRender() {

        setTimeout(() => {

            this.setState({
                arr: [...this.state.arr, "chocolate", "strawberry"]
            });

        }, 1000);

    }

    Element(props, state) {

        return (
            <ul>
                { state.arr.map(item => <li _key={ item }>{ item }</li>) }
            </ul>
        );

    }

}
```

Na tomto příkladu se po 1 sekundě přidají do listu dva prvky s hodnotami „chocolate“ a „strawberry“.

## 6.4 Životní cykly komponenty

Každá komponenta má své životní cykly. Životní cykly jsou metody komponenty, které jsou spuštěny v určitou dobu. V každém tomto životním cyklu se může provádět jakákoliv operace, kromě speciálních životních cyklů, které mají pravidla více definovaná.

### 6.4.1 Obyčejné životní cykly

Obyčejné životní cykly slouží pro spuštění kódu ve chvíli, kdy se komponenta nachází v daném stavu.

- **onComponentRender**
  - Tento životní cyklus je spuštěn potom, co je z komponenty vytvořen DOM element.
- **onComponentMount**
  - Tento životní cyklus je spuštěn potom, co je komponenta vykreslená na stránce.
- **onComponentUpdate**
  - Tento životní cyklus je spuštěn potom, co je komponenta aktualizovaná. Při aktualizaci rodičovské komponenty se spustí automaticky aktualizace i všech komponent, které jsou potomky té aktualizované komponenty.
- **onComponentCancelUpdate**
  - Tento životní cyklus je spuštěn potom, co komponenta zruší svou aktualizaci pomocí `Component.shouldComponentUpdate`.

### 6.4.2 Budoucí životní cykly

Tyto životní cykly fungují podobně jako obyčejné životní cykly, ale spouští se před tím, než nastane daná situace.

- **onComponentWillMount**
  - Tento cyklus je spuštěn před tím, než se komponenta vykreslí na stránku.
- **onComponentWillUnMount**
  - Tento cyklus je spuštěn před tím, než je komponenta odebrána z virtuálního stromu při aktualizaci.
  - Tento cyklus slouží hlavně pro vypnutí asynchronních operací, které způsobují aktualizaci komponenty. Ta již není vykreslená, je zničená a tím by docházelo k nechtěnému úniku paměti.
- **onComponentWillUpdate**
  - Tento cyklus je spuštěn před tím, než se komponenta začne aktualizovat.

### 6.4.3 Řídící životní cykly

Řídící životní cykly jsou speciální životní cykly komponenty, které rozhodují o různých událostech, například jestli se má komponenta aktualizovat.

- **shouldComponentUpdate**
  - Tento cyklus se dotazuje, jestli se má komponenta aktualizovat.
  - Slouží především jako optimalizační životní cyklus, díky kterému nemusíme aktualizovat všechny komponenty, které jsou potomky aktualizované komponenty, a tím zvýšit rychlost aplikace.
  - Pro zamezení aktualizace musíme z tohoto životního cyklu vrátit hodnotu *false*.
  - Pro aktualizaci komponenty musíme vrátit hodnotu *true*.
  - Výchozí nastavení všech komponent je, že tento životní cyklus vrací hodnotu *true*, tzn., že se komponenta vždy aktualizuje.
- **getSnapshotBeforeUpdate**
  - Tento cyklus slouží pro porovnání starých a nových dat. Na základě toho může provést další operace.
  - Cyklus je spuštěn před přiřazením nových *state* komponenty a před spuštěním `Component.onComponentWillUpdate`**.**
  - Pokud chceme data porovnávat, musíme v tomto cyklu vrátit data a následně využít „snapshot“ parametr v životním cyklu `Component.onComponentWillUpdate` nebo `Component.onComponentUpdate`.
- **componentWillGetProps**
  - Tento životní cyklus je spuštěn ještě před spuštěním `Component.onComponentWillUpdate`.
  - Slouží pro předávání dat z objektu *props*, které mají být propsány do objektu *state* tím, že je v tomto životním cyklu vrátíme pomocí *return*.

```
/** @jsx Kiq.createElement */

class Counter extends Kiq.Component {

    state = {
        count: 0
    };

    getSnapshotBeforeUpdate() {

        return {
            oldCount: this.state.count
        };

    }

    onComponentUpdate(snapShot) {

        console.log(snapShot.oldCount, this.state.count);

    }

    Element(props, state) {

        return (
            <button
                onclick={ (e) => this.setState({ count: state.count + 1 }) }>
                { state.count }
            </button>
        );

    }

}
```

Ukázka použití `Component.getSnapshotBeforeUpdate` životního cyklu.

Tento příklad po každé aktualizaci vypíše starou hodnotu *count* a aktualizovanou hodnotu.

# 7 Manipulace s DOM elementy

Aktualizaci DOM elementů zajišťuje sama knihovna za pomocí hledání rozdílů ve dvou virtuálních stromech (novým a starým) vygenerovaných v komponentě.

Díky tomuto faktu je možné aktualizovat pouze části DOM elementů, které opravdu potřebují aktualizovat nebo překreslit. Části DOM elementů, které při aktualizaci nepotřebují provést žádnou operaci, zůstanou nedotčeny.

## 7.1 Problém ruční manipulace

Tento problém spočívá v nativní manipulaci s DOM elementy. Pokud dojde k jakékoliv manipulaci či změně elementu, který má na starost knihovna, tak by jej měla aktualizovat pouze knihovna pomocí vstupních dat a virtuálního DOM.

Pokud v aplikaci dojde k tomuto problému, můžou nastat nečekané chyby. Aplikace se díky neoprávněným změnám nemusí vždy vrátit zpět do funkčního stavu.

## 7.2 Povolené ruční manipulace

Povolené ruční manipulace s DOM elementy jsou takové, o jaké se nestará sama knihovna. Například nastavování atributů, přidávání elementů mimo elementy, se kterými manipuluje knihovna nebo používání některých metod DOM elementů.

## 7.3 Chirurgická manipulace

Díky hledání změn ve virtuálních stromech, jsou DOM elementy aktualizovány s minimálními dopady na výkon a překreslování stránky. Všechny změny jsou nejdříve zjištěny, a pak jsou dosazeny jen změny, které jsou potřebné k překreslení. Nezměněné stavy elementů se neaktualizují a nepřekreslí. Tímto je zajištěn maximální výkon, spolehlivost DOM elementů a plynulost stránky při větších aktualizacích.

## 7.4 Optimalizace aktualizací DOM elementů

Knihovna zabalí všechny změny DOM elementů zjištěné z porovnávání virtuálních stromů do jedné funkce. Tím je zajištěno, že prohlížeč tyto změny vykoná mnohem rychleji. Jedná se o Batch algoritmus, který je popsán v kap. Terminologie.

# 8 Renderování virtuálního DOM

Renderování virtuálního DOM je operace, při které se z virtuálního DOM vytvoří DOM element. Pokud použijeme komponentu kdekoliv ve virtuálním stromě, každá z těchto komponent spustí životní cyklus `Component.onComponentRender`.

Každé renderování by mělo být v aplikaci zavoláno pouze jednou pro každou kořenovou komponentu.

Za běhu aplikace se o dodatečné renderování přidaných virtuálních elementů stará knihovna.

# 9 Vykreslení virtuálního DOM na stránku

Tento proces je spuštěn hned poté, co je z virtuálního DOM sestavený DOM element. Pokud použijeme komponentu kdekoliv ve virtuálním stromě, každá z těchto komponent spustí životní cyklus `Component.onComponentWillMount` před tím, než je komponenta vykreslena. Následně spustí životní cyklus `Component.onComponentMount`.

## 9.1 Optimalizace vykreslování

Při prvotním vykreslování aplikace je použita optimalizace `window.requestAnimationFrame`, která je popsána v kap. Terminologie.

# 10 Diff algoritmus

Algoritmus zjišťuje změny ve dvou virtuálních stromech, starém a novém. Tento algoritmus vytváří funkce (patche) neboli „opravy“ DOM elementů pro jejich překreslení. Tyto změny elementů ovšem nejsou spuštěny hned, ale čekají na všechny ostatní změny a spustí se všechny najednou viz Batch algoritmus popsán v kap. Terminologie.

## 10.1 Rozdělení diff algoritmu

Funkci je třeba rozdělit na porovnávání řetězců (text node), virtuálních elementů (DOM elementů) a komponent tak, aby nebylo porovnávání virtuálních stromů nadbytečné a bylo, co možná nejrychlejší.

## 10.2 Children reordering

Tento algoritmus slouží pro změnu struktury potomků v rodičovském elementu. Využívá klíče popsané v předešlém textu. Funkce vytvoří pole všech změn v přesunu elementů a spustí je společně s ostatními změnami.

# 11 Kiq.Component vs WebComponent

## 11.1 Web komponenty

Jedná se pouze o zapouzdření DOM elementů, kde není možné přidat datový systém. Data se přenáší pouze pomocí atributů, všechna data jsou ve formě řetězců. Web komponenty dále mohou využívat Shadow DOM popsán v kap. Terminologie.

## 11.2 Kiq.Component

Jsou zapouzdřením šablon dat a elementu, který je ovšem zapsán jako virtuální DOM. Jedná se spíše o logickou definici elementu a dat s elementem spojených. Komponenty jsou logickým rozdělením aplikace, nikoliv zapouzdřením DOM elementů jako tomu je u web komponent.

# 12 Závěr

## 12.1 Nevyužité sledování dat

Zamýšlel jsem, že nejlepší způsob implementace reaktivity bude změna dat, stejně jako v klasickém Javascriptu, bez použití speciální metody `Component.setState`.

Tohoto sledování změny dat bych dosáhl pomocí *Proxy* objektu, který dovoluje vytvořit vlastní akce pro manipulaci s objekty. Tento způsob sledování dat se v průběhu práce ukázal neefektivní. Pokud by se provedlo více aktualizací najednou, spustilo by se hledání změn vícekrát (redundantně) a k překreslení by také docházelo vícekrát (podle každé změny). Tudíž by rychlost knihovny klesla a nedalo by se využít **Batch** algoritmu, který je popsán v kap. Terminologie.

Všechna ostatní zadání jsou splněna.

## 12.2 Vlastní závěr

Knihovna nabízí oproti zadání mnohem více možností. Tím jsem překonal zadané cíle maturitní práce.

Díky deklarativnímu designu DOM manipulací je velice snadné knihovnu používat. Je jednoduché v ní pracovat a vytvářet webové aplikace.

Knihovna je velice rychlá a malá.

Inspiroval jsem se knihovnou **React.js**, která jako první použila a vyvinula technologii virtuálního DOM. Knihovna **Kiq.js** je mnohem rychlejší a mnohem menší než knihovna React.js.

Knihovna je jednoduše a volně stažitelná na Githubu. Dále je stažitelná na NPM, pomocí příkazu `npm i kiq`, pokud je na počítači nainstalovaný node.js. Také se dá připojit ke kódu pomocí CDN odkazu, který hostuje JSdelivr.

S výsledkem práce jsem maximálně spokojen.

Vytváření práce mě obohatilo o praktickou zkušenost v programování v jazyce Javascript.

Knihovna je prakticky využitelná pro ostatní webové programátory. Knihovna má dohromady přes 1000 stažení za měsíc.

# 13 Seznam obrázků

# 14 Přílohy

**příloha I:** Licenční smlouva
**příloha II:** Zadání dlouhodobé maturitní práce
**příloha III:** Poslední verze (0.0.6) knihovny Kiq.js
**příloha IV:** Odkazy pro stáhnutí poslední verze (0.0.6) Kiq.js
**příloha V:** Video od Jason Yu na serveru YouTube
**příloha VI:** Oficiální stránky knihovny React.js

# 15 Literatura


[1] DOM: Document Object Model [online]. https://www.tvorba-webu.c, c2003 - 2008 [cit. 2021-03-22]. Dostupné z: https://www.tvorba-webu.cz/dom/

[2] Content delivery network [online]. https://cs.wikipedia.org, 2021 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Content_delivery_network

[3] Reaktivní programování [online]. https://cs.wikipedia.org, 2021 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Reaktivn%C3%AD_programov%C3%A1n%C3%AD

[4] Modulární programování [online]. https://cs.wikipedia.org, 2020 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Modul%C3%A1rn%C3%AD_programov%C3%A1n%C3%AD

[5] Reflow [online]. https://developer.mozilla.org, 2020 [cit. 2021-03-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Reflow

[6] Repaint [online]. https://dev.to, 2020 [cit. 2021-03-22]. Dostupné z: https://dev.to/gopal1996/understanding-reflow-and-repaint-in-the-browser-1jbg

[7] React (webový framework) [online]. https://cs.wikipedia.org, 2021 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/React_(webov%C3%BD_framework)#JSX

[8] Transpiler [online]. https://cs.wikipedia.org, 2021 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Transpiler

[9] Deklarativní programování [online]. https://cs.wikipedia.org, 2020 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Deklarativn%C3%AD_programov%C3%A1n%C3%AD

[10] Imperativní programování [online]. https://cs.wikipedia.org, 2021 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Imperativn%C3%AD_programov%C3%A1n%C3%AD

[11] Návrhový antivzor [online]. https://cs.wikipedia.org, 2020 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/N%C3%A1vrhov%C3%BD_antivzor

[12] Executing multiple DOM updates with JavaScript efficiently [online]. https://stackoverflow.com, 2016 [cit. 2021-03-22]. Dostupné z: https://stackoverflow.com/questions/37039667/executing-multiple-dom-updates-with-Javascript-efficiently

[13] Backtracking [online]. https://cs.wikipedia.org, 2019 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Backtracking

[14] Prohledávání do hloubky [online]. https://cs.wikipedia.org, 2019 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Prohled%C3%A1v%C3%A1n%C3%AD_do_hloubky

[15] RequestAnimationFrame [online]. https://www.itnetwork.cz/, c2021 [cit. 2021-03-22]. Dostupné z: https://www.itnetwork.cz/Javascript/zaklady/tutorial-Javascript-requestanimationframe-za-lepsi-vykreslovani

[16] Template literals [online]. https://developer.mozilla.org, 2021 [cit. 2021-03-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Javascript/Reference/Template_literals

[17] Tagged Templates [online]. https://developer.mozilla.org, 2021 [cit. 2021-03-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Javascript/Reference/Template_literals#tagged_templates

[18] Web Components [online]. https://developer.mozilla.org, 2021 [cit. 2021-03-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Web_Components

[19] Shadow DOM [online]. https://developer.mozilla.org, 2021 [cit. 2021-03-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM

[20] Npm [online]. https://cs.wikipedia.org, 2020 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/Npm

[21] GitHub [online]. https://cs.wikipedia.org, 2020 [cit. 2021-03-22]. Dostupné z: https://cs.wikipedia.org/wiki/GitHub

[22] [online]. Dostupné také z: https://imgs.developpaper.com/imgs/3168123057-5c7699f9e0ef9_articlex.png