Pedagogická fakulta
Faculty of Education

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Tvorba webových aplikací s využitím transpileru TypeScript 4.2

Creating web applications using the transpiler TypeScript 4.2

Bakalářská práce

Vypracoval: Tomáš Rothbauer

Vedoucí práce: PaedDr. Petr Pexa, Ph.D.

České Budějovice 2022

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Pedagogická fakulta

Akademický rok: 2020/2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Tomáš ROTHBAUER**
Osobní číslo:	**P19621**
Studijní program:	**B7507 Specializace v pedagogice**
Studijní obor:	**Fyzika se zaměřením na vzdělávání**
	Informační technologie se zaměřením na vzdělávání
Téma práce:	**Tvorba webových aplikací s využitím transpileru TypeScript 4.2**
Zadávající katedra:	**Katedra informatiky**

Zásady pro vypracování

Cílem bakalářské práce bude komplexní zpracování problematiky nové verze transpileru TypeScript 4.2, který má usnadnit práci s jazykem JavaScript, na kterém je založen. TypeScript je open-source programovací jazyk, vytvořený a spravovaný firmou Microsoft, jedná se vlastně o nadstavbu nad jazykem JavaScript, která jej rozšiřuje o statické typování a další atributy, které známe z objektově orientovaného programování. Samotný kód, psaný v TypeScriptu, je pak kompilován do JavaScriptu, při čemž kompilátor disponuje funkcí automatického doplňování a hlídá chyby před spuštěním. Teoretická část práce bude zaměřena na samotný TypeScript, jeho porovnání s JavaScriptem samotným, zhodnocení výhod, nevýhod a rozdílů, představení syntaxe a aktuálních funkcí s využitím JS frameworku React, který je mnoha vývojáři používán při tvorbě webových aplikací. Praktickou část pak bude tvořit sada demonstrativních webových aplikací, napsaných v jazyce TypeScript, které budou umístěny na vlastním speciálním webu a dokumentovány.

Rozsah pracovní zprávy:	**40**
Rozsah grafických prací:	**CD ROM**
Forma zpracování bakalářské práce:	**tištěná**

Seznam doporučené literatury:

1. TypeScript 4.2. TypeScript: Typed JavaScript at Any Scale [online]. 2021-3-31 [cit. 2021-04-02]. Dostupné z: https://www.typescriptlang.org/docs/handbook/release-notes/typescript-4-2.html
2. Getting Started. React: A JavaScript library for building user interfaces [online]. Facebook, ©2021 [cit. 2021-04-02]. Dostupné z: https://reactjs.org/docs/getting-started.html
3. Getting Started. Create React App: Set up a modern web app by running one command [online]. Facebook, ©2021 [cit. 2021-04-02]. Dostupné z: https://create-react-app.dev/docs/getting-started
4. DUBOIS, Sebastien a Alexis GEORGES. Learn TypeScript 3 by Building Web Applications: Gain a solid understanding of TypeScript, Angular, Vue, React, and NestJS. 2019-11-22. Birmingham: Packt Publishing, 2019. ISBN 1789615860.
5. JavaScript Tutorial. W3Schools: Online Web Tutorials [online]. ©1999-2021 [cit. 2021-04-02]. Dostupné z: https://www.w3schools.com/js/default.asp

Vedoucí bakalářské práce: **PaedDr. Petr Pexa, Ph.D.**
Katedra informatiky

Datum zadání bakalářské práce: **7. dubna 2021**
Termín odevzdání bakalářské práce: **30. dubna 2022**

L.S.

doc. RNDr. Helena Koldová, Ph.D.
děkanka

doc. PaedDr. Jiří Vaníček, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 7. dubna 2021

Prohlášení

Prohlašuji, že jsem autorem této kvalifikační práce a že jsem ji vypracoval(a) pouze s použitím pramenů a literatury uvedených v seznamu použitých zdrojů.

V Českých Budějovicích dne 30. června 2022.

Tomáš Rothbauer

Abstrakt/Anotace

Cílem bakalářské práce bude komplexní zpracování problematiky nové verze transpileru TypeScript 4.2, který má usnadnit práci s jazykem JavaScript, na kterém je založen. TypeScript je open-source programovací jazyk, vytvořený a spravovaný firmou Microsoft, jedná se vlastně o nadstavbu nad jazykem JavaScript, která jej rozšiřuje o statické typování a další atributy, které známe z objektového programování. Samotný kód, napsaný v TypeScriptu, je pak kompilován do JavaScriptu, při čemž kompilátor disponuje funkcí automatického doplňování a opravování chyby ještě před spuštěním samotného JavaScriptu. Teoretická část práce bude zaměřena na samotný TypeScript, jeho porovnání s JavaScriptem samotným, zhodnocení výhod, nevýhod a rozdílů, představení syntaxe a aktuálních funkcí s využitím JS frameworku React, který je mnoha vývojáři používán při tvorbě webových aplikací. Praktickou část pak bude tvořit sada demonstrativních webových aplikací, napsaných v jazyce TypeScript, které budou umístěny na vlastním webu a dokumentovány.

Klíčová slova

TypeScript, JavaScript, React

Abstract

The goal of the bachelor thesis will be a comprehensive elaboration of the new version of the TypeScript 4.2 transpiler, which is intended to facilitate the work with the JavaScript language on which it is based. TypeScript is an open-source programming language, created and maintained by Microsoft, it is actually asuperstructure over JavaScript, which extends it with static typing and other attributes we know from object-oriented programming. The code itself, written in TypeScript, is then compiled into JavaScript, with the compiler having the ability to auto-complete and correct errors before running JavaScript code. The theoretical part of the thesis will focus on TypeScript itself, comparing it with JavaScript itself, evaluating the advantages, disadvantages and differences, introducing the syntax and current features using the React JS framework, which is used by many developers to create web applications. The practical part will then consist of a series of demonstrative web applications, written in TypeScript, which will be hosted on their own website and documented.

Keywords

TypeScript, JavaScript, React

Poděkování

Tímto bych rád poděkoval panu PaedDr. Petru Pexovi, Ph.D., za jeho cenné rady a odborný a vstřícný přístup při vedení mé bakalářské práce.

Obsah

1 Úvod

1.1 Východiska práce

TypeScript je společností Microsoft propagován jako nadstavba JavaScriptu, umožňující snadnou kontrolu kódu ještě před jeho spuštěním a nalezení potenciálních chyb. JavaScript, který kompilací TypeScriptu vytvoříme, je plně funkční v prohlížečích, Node.JS nebo kdekoliv jinde, kde JavaScript funguje. TypeScript má zároveň integrovanou podporu většiny editorů a umožňuje tak snadný přechod z jednoho editoru na jiný. Aktuálně je hojně využíván v kombinaci s frameworkem Reakt jako kompilátor při tvorbě webových aplikací, pro zkvalitnění výsledného kódu díky jejich snadnému a rychlému propojení. Kvůli tomu, že se však jedná pouze o nadstavbu, je pro práci s transpilerem nutná předchozí znalost JavaScriptu. V současnosti je na českém internetu bohužel minimum zdrojů, které by využití TypeScriptu v kombinaci s Reactem pro tvorbu webových aplikací dokumentovaly.

1.2 Cíle práce

Cílem bakalářské práce bude komplexní zpracování problematiky transpileru TypeScript 4.2, který má usnadnit práci s jazykem JavaScript, na kterém je založen. TypeScript je open-source programovací jazyk, vytvořený a spravovaný firmou Microsoft, jedná se o nadstavbu nad jazykem JavaScript, která jej rozšiřuje o statické typování a další atributy, které známe z objektového programování. Samotný kód, napsaný v TypeScriptu, je pak kompilován do JavaScriptu, při čemž kompilátor disponuje funkcí automatického doplňování a opravování chyby před spuštěním samotného JavaScriptu. Tato funkce má usnadnit práci při psaní kódu tím, že nás TypeScript předem upozorní na chyby a nedovolí do proměnné jednoho typu zapsat hodnotu s rozdílným typem. Nemusíme tak trávit tolik času důkladným prověřováním našeho kódu, abychom odhalili všechny potenciální chyby, což je užitečné zejména u rozsáhlých programů o tisících řádcích kódu. V teoretická části se dále zaměřím na zhodnocení výhod a nevýhod a představení syntaxe pro JS framework React, který je mnoha vývojáři používán při tvorbě webových aplikací právě v kombinaci s TypeScriptem ve formě TSX kvůli jeho snadné instalaci.

Praktickou část pak bude tvořit sada demonstrativních webových aplikací, sloužících k představení TypeScriptu v TSX, které budou umístěny na vlastním webu a dokumentovány.

1.3 Metody práce

V úvodu práce se zaměřím na popsání jazyka TypeScript a jeho odlišností oproti JavaScriptu, popíši jeho účel, fungování a jak ho využít při práci. Následně se zaměřím na framework React a popis jazyka JSX. Využití TypeScriptu následně demonstruji na sedě demonstrativních aplikací napsaných v TSX, který je kombinací JSX a TypeScriptu.

2 TypeScript

2.1 Původ

JavaScript, vynalezený v roce 1995, byl navržen jako malý skriptovací jazyk pro jednoduché webové stránky v prohlížečích. Až v roce 1999 byl JavaScript schopen podporovat druhy dynamických webových stránek, které vidíme dnes, a používání JavaScriptu tímto způsobem nebylo až do roku 2005 běžnou praxí. Aby posloužil svému původnímu použití, byl JavaScript navržen tak, aby byl velmi flexibilní a snadno použitelný pro malé aplikace. Díky těmto funkcím je JavaScript skvělým prvním jazykem, který se lze naučit, ale také není ideální pro vytváření aplikací ve větším měřítku se stovkami nebo dokonce tisíci souborů. Přísnější programovací jazyky informují vývojáře, když změní jednu oblast kódu způsobem, který naruší ostatní oblasti. JavaScript ne, což často vede k neočekávanému chování za běhu. K vyřešení těchto nedostatků vyvinul Microsoft TypeScript a zveřejnil jej v roce 2012, aby spojil flexibilitu JavaScriptu s výhodami přísnějšího jazyka. [1]

2.2 Statická kontrola typu

Při programování se nikdy nevyhneme chybám, a to ať už ve formě špatné syntaxe, nebo přepsání se. Vždy po napsání krátkého kusu kódu, bychom mohli náš program uložit a zkusit spustit, ale ani tak nemusíme chybu odhalit. To platí zejména u velmi složitých programů s tisíci řádky kódu. Místo toho tedy využíváme jeden z nástrojů, které TypeScript poskytuje, a to statickou kontrolu typu, která ještě před spuštěním programu popíše, jak se bude každá proměnná chovat. TypeScript informací z této kontroly využívá, aby nás informoval, co a jakým způsobem bude v našem programu dělat problémy. [2]

2.3 Typy

2.3.1 Primitivní

JavaScript má tři velmi běžně používaná „primitivní“ datové typy: string, number a boolean. Každý z nich má odpovídající typ v TypeScriptu.

1. `string` typ sloužící k reprezentaci řetězce znaků jako například `"Ahoj, světe!"`
2. `number` JavaScript nerozlišuje mezi celým a desetinným číslem. Místo `int` a `float` tak máme pro všechny číselné hodnoty typ number.
3. `boolean` slouží pro uchování pravdivostních hodnot `true` a `false` [2]

2.3.2 Any

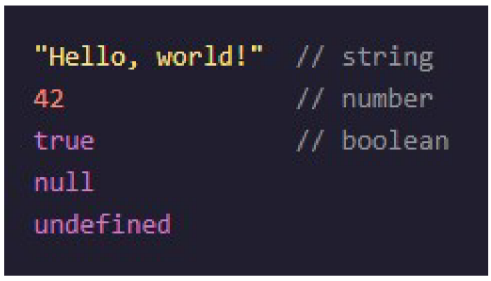
Když je proměnná deklarována bez přiřazení počáteční hodnoty, TypeScript ji považuje za typ any. [3] Obvykle se tomu však snažíme vyhnout, protože žádný není typově zkontrolován. [2] Proměnnou tohoto typu lze přeřazovat bez generování jakékoli chyby v TypeScriptu. [3] Typ any je užitečný, když nechceme vypisovat dlouhý typ, jen abychom přesvědčili TypeScript, že konkrétní řádek kódu je v pořádku. [2]

2.3.3 Null

Ve výchozím nastavení jsou podtypy všech ostatních typů `null` a `undefined`. To znamená, že k něčemu jako je číslo můžeme přiřadit hodnotu `null` a `undefined` a stejně jako `void` nejsou samy o sobě extrémně užitečné. [4] TypeScript zachází s `null` a `undefined` odlišně, aby odpovídal sémantice JavaScriptu. `string | null` je jiný typ než `string | undefined` a `string | undefined | null`. [5]

2.3.4 Typová anotace

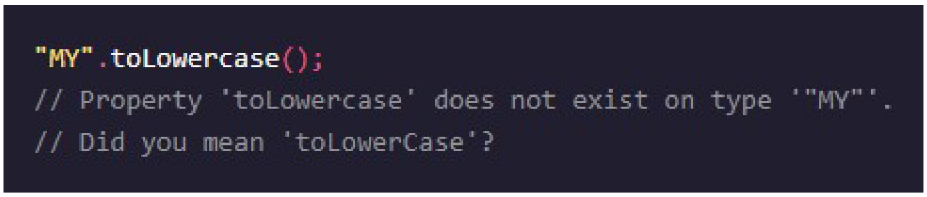
TypeScript používá typové anotace k explicitní specifikaci typů pro identifikátory, jako jsou proměnné, funkce, objekty atd. TypeScript používá syntaxi `: type` za identifikátorem jako anotaci typu, kde `type` může být jakýkoli platný typ. Jakmile je identifikátor označen typem, lze jej použít pouze jako tento typ. Pokud je identifikátor použit jako jiný typ, kompilátor ohlásí chybu. [6] Ve většině případů to však není potřeba. Kdykoli je to možné, TypeScript se snaží automaticky odvodit typy v našem kódu. [2]



Obrázek 1: Typy [3]

2.3.5 Tvar objektu

Protože TypeScript ví, jakého typu jsou naše objekty, ví také, jaké tvary naše objekty drží. Tvar objektu mimo jiné popisuje, jaké vlastnosti a metody obsahuje nebo jaké neobsahuje. [7] TypeScript zaprotokoluje chybu, pokud se kód pokusí získat přístup k členům objektu, o kterém je známo, že neexistuje. Může dokonce navrhnout možné opravy. [3] Mají-li dva předměty stejný tvar, považují se za stejného typu. Pokud má objekt nebo třída všechny požadované vlastnosti, TypeScript řekne, že se shodují, bez ohledu na podrobnosti implementace. [8]



Obrázek 2: Tvar objektu [7]

2.4 Funkce

Funkce jsou stavebními kameny čitelného, udržovatelného a opakovaně použitelného kódu. Jedná se o sadu příkazů k provedení konkrétního úkolu. Funkce organizují program do logických bloků kódu. Jakmile jsou definovány, mohou být volány k přístupovému kódu. Díky tomu je kód znovu použitelný. Funkce navíc usnadňují čtení a údržbu kódu programu. Deklarace funkce říká kompilátoru o názvu, návratovém typu a parametrech. Definice poskytuje skutečné tělo funkce. [9]

```
1 function logGreeting(name: string){
2   console.log(`Hello, ${name}!`)
3 }
```

Příklad 1: Funkce

[10]

2.4.1 Typová anotace parametru

Když deklarujeme funkci, můžeme přidat typové anotace za každý parametr, abychom deklarovali, jaké typy parametrů funkce přijímá. Anotace typu následuje po názvu parametru. [2]

2.4.2 Anotace návratového typu

Je možné také přidat anotaci návratového typu, která se přidává za seznam parametrů. Podobně jako u anotací typu proměnné obvykle nepotřebujeme anotaci návratového typu, protože TypeScript odvodí návratový typ funkce na základě návratové hodnoty. [2]

2.4.3 Návratový typ Void

Typ `void` představuje návratový typ funkce, když nevrací žádná data. Void je tedy skutečně užitečný pouze pro typy návratů funkcí. Je možné ho odvodit, ale můžeme jej explicitně definovat na funkcích, pokud budeme chtít. [11]

2.4.4 Nepovinné parametry

V JavaScriptu můžeme volat funkci bez předávání jakýchkoli argumentů, i když funkce určuje parametry. Proto JaveScript ve výchozím nastavení podporuje volitelné parametry. V TypeScriptu kompilátor zkontroluje každé volání funkce a vydá chybu v následujících případech:

- Počet argumentů se liší od počtu parametrů zadaných ve funkci.
- Nebo typy argumentů nejsou kompatibilní s typy parametrů funkcí.

Protože kompilátor důkladně kontroluje předávané argumenty, musíme anotovat volitelné parametry, abychom kompilátoru dali pokyn, aby nevydal chybu, když argumenty vynecháme. Chceme-li, aby byl parametr funkce volitelný, použijeme `?` za názvem parametru. [12]

2.4.5 Defaultní parametry

TypeScript poskytuje možnost přidat k parametrům výchozí hodnoty. Pokud tedy uživatel nezadá hodnotu argumentu, TypeScript inicializuje parametr s výchozí hodnotou. Výchozí parametry mají stejné chování jako volitelné parametry. Pokud není předána hodnota pro výchozí parametr ve volání funkce, musí výchozí parametr odpovídat požadovaným parametrům v podpisu funkce. Výchozí parametry lze tedy při volání funkce vynechat. [13]

```
function greet(name = 'Anonymous') {
  console.log(`Hello, ${name}!`);
}
```

Příklad 2: Defaultní parametry

[14]

2.4.6 Komentáře funkcí

TypeScript rozpozná syntaxi komentářů z JavaScriptu, ale je běžné vidět i třetí styl komentáře. Poznámky k dokumentaci jsou užitečné zejména pro dokumentaci funkcí. Komentář k dokumentaci funkce umístíme do kódu přímo nad deklaraci funkce. Ke zvýraznění určitých aspektů můžeme v komentáři použít speciální značky. Můžeme použít `@param` k popisu každého z parametrů a `@returns` k popisu toho, co funkce vrací. Mnoho textových editorů užitečně zobrazí komentáře k dokumentaci, například při najetí myší na název funkce.

```
/**
   * Returns the sum of two numbers.
   * @param x - The first input number
   * @param y - The second input number
   * @returns The sum of `x` and `y`
   */
```

```
7  function getSum(x: number, y: number): number {
8    return x + y;
9  }
```

Příklad 3: Komentáře funkcí

[15]

2.5 Pole

Pole je speciální typ datového typu, který může ukládat více hodnot různých datových typů postupně pomocí speciální syntaxe. TypeScript podporuje pole, podobně jako JavaScript. Existují dva způsoby, jak deklarovat pole:

1. Pomocí hranatých závorek. Tato metoda je podobná tomu, jak byste deklarovali pole v JavaScriptu.
   `let fruits: string[] = ['Apple', 'Orange', 'Banana'];`
2. Pomocí obecného typu pole, `Array<elementType>`.
   `let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];`

Obě metody produkují stejný výstup. [16]

2.5.1 Anotace typu pole

Chceme-li určit typ pole, jako je `[1, 2, 3]`, můžeme použít syntaxi pro číslo `number[]`; tato syntaxe funguje pro jakýkoli typ (např. `string[]` je pole řetězců). Můžete také vidět zápis jako `Array<number>`, což znamená totéž. [2]

2.5.2 Vícerozměrná pole

Prvek pole může svou hodnotou odkazovat na jiné pole. Taková pole se nazývají vícerozměrná pole. TypeScript podporuje koncept vícerozměrných polí. Nejjednodušší formou je pole dvourozměrné. [17] Typ pro vícerozměrné pole lze označit přidáním zvláštního `[]` pro každý další rozměr pole. [18]

2.5.3 Tuples

Tuple je dalším typem typu pole, který přesně ví, kolik prvků obsahuje a přesně jaké typy obsahuje na konkrétních pozicích. [19] Spoléhání se na pořadí může

znesnadnit čtení, údržbu a použití kódu. Z toho důvodu je dobré používat n-tice s daty, která spolu souvisí sekvenčním způsobem. Tímto způsobem je přístup k prvkům v pořadí součástí předvídatelného a očekávaného vzoru. Na druhou stranu přidružená data, která jsou volně svázaná, nejsou přínosná. Potřebovali bychom vysledovat kód nebo přečíst dokumentaci, abychom zjistili, jak jsou data mapována. Toto není ideální scénář a použití rozhraní by bylo mnohem lepší. [20] Pokud jde o JavaScript, n-tice fungují stejně jako pole. Oba mají vlastnosti `.length`. K prvkům obou můžeme přistupovat (nebo je měnit) pomocí `[index]`. Ale navzdory jejich podobnosti nemají n-tice a pole kompatibilní typy v rámci TypeScriptu. Konkrétně nemůžeme n-ticové proměnné přiřadit pole, i když jsou prvky správného typu. [21]

```
let numbersTuple: [number, number, number] = [1,2,3,4]; // Type
Error! numbersTuple should only have three elements.
let mixedTuple: [number, string, boolean] = ['hi', 3, true] // Type
Error! The first elements should be a number, the second a string,
and the third a boolean.
```

Obrázek 3: Tuples [21]

2.5.4 Odvození

Protože n-tice mají pevnou délku, nemohli bychom do n-tice přidat další prvky. Pole je oproti tomu pole méně omezující typ. [22] Na n-tici TypeScriptu lze zavolat metodu JavaScriptu `.concat()`, která vytváří nový typ pole namísto n-tice. [18] To nám umožňuje pole rozšířit. [22] Když je proměnná pole deklarována bez explicitní anotace typu, TypeScript automaticky odvodí takovou instanci proměnné jako pole namísto n-tice. [18] Když chceme n-tice, musíme použít anotace explicitního typu. [22]

2.5.5 Zbytek

Syntaxe zbývajícího parametru umožňuje funkci přijmout neurčitý počet argumentů jako pole, což poskytuje způsob, jak reprezentovat variadické funkce v JavaScriptu. Poslední parametr definice funkce může mít předponu "...", což způ-

sobí, že všechny zbývající (uživatelem dodané) parametry budou umístěny do standardního pole JavaScriptu. Pouze poslední parametr v definici funkce může být zbývajícím parametrem. [23] Zbývající parametry lze použít ve funkcích, šipkových funkcích nebo třídách. [24] Typ zbytku je typ pole. [25] Parametru rest uvnitř funkce je pomocí TypeScriptu implicitně přiřazen typ pole `any[]`. Explicitní zadání anotací parametru rest funkce upozorní TypeScript, aby zkontroloval typovou nekonzistenci mezi parametrem rest a argumenty volání funkce. [18]

```
1 function smush(firstString, ...otherStrings: string[]){
2   /*rest of function*/
3 }
```

Příklad 4: Rest

[26]

2.5.6 Syntaxe rozprostření

Syntaxi rozprostření lze použít s n-ticí jako argument pro volání funkce, jejíž typy parametrů se shodují s typy prvků n-tice. [18] Syntaxi rozprostření lze použít, když všechny prvky z objektu nebo pole musí být zahrnuty do seznamu nějakého druhu. Když funkci vyvoláme, předáme jí všechny hodnoty v poli pomocí syntaxe rozprostření a názvu pole. Pokud by pole obsahovalo více prvků, než funkce argumentů, pak by byly předány všechny prvky, ale byly by použity pouze první prvky z pole, počtně odpovídající počtu argumentů, pokud bychom do funkce nepřidali další argumenty. [27]

2.6 Komplexní typy

Předdefinované typy lze kombinovat do vlastních a složitějších typů. Složité typy jsou použitelné stejným způsobem jako jednodušší typy. Mohou být použity jako typové anotace během deklarace proměnných, jako typové anotace pro funkce a dokonce můžeme dělat typové odvození s komplexními typy. [28]

2.6.1 Výčet

V TypeScriptu jsou výčty nebo výčtové typy datové struktury konstantní délky, které obsahují sadu konstantních hodnot. Každá z těchto konstantních hodnot je známá jako člen výčtu. Výčty jsou užitečné při nastavování vlastností nebo hodnot, které mohou představovat pouze určitý počet možných hodnot. [29]

Číselné výčty jsou číselné výčty, tj. ukládají řetězcové hodnoty jako čísla. Výčtům jsou při ukládání vždy přiřazeny číselné hodnoty. První hodnota vždy nabývá číselnou hodnotu 0, zatímco ostatní hodnoty ve výčtu se zvyšují o 1. Máme také možnost inicializovat první číselnou hodnotu sami. Řetězcové výčty jsou podobné číselným výčtům s tím rozdílem, že hodnoty výčtu jsou inicializovány řetězcovými hodnotami, nikoli číselnými hodnotami. Výhodou použití řetězcových výčtů je, že nabízejí lepší čitelnost. Pokud bychom ladili program, je snazší číst řetězcové hodnoty než číselné hodnoty. [30]

```
1 enum Direction {
2   North,
3   South,
4   East,
5   West
6 }
```

Příklad 5: Výčet

[31]

2.6.2 Objektové typy

Objektové typy jsou v TypeScriptu extrémně užitečné, protože nám umožňují velmi jemnou kontrolu nad typy proměnných v našich programech. Jsou to také nejběžnější vlastní typy, takže jim musíme porozumět, pokud chceme číst programy jiných lidí. [32] Objektový literál JavaScriptu se skládá z párů vlastnost-hodnota. Chceme-li typově anotovat literál objektu, použijeme typ objektu TypeScriptu a určíme, jaké vlastnosti musí být poskytnuty, a jejich doprovodné typy hodnot. [18] Objektové typy mohou být anonymní, nebo mohou být pojmenovány pomocí rozhraní nebo aliasu typu.

```
let aPerson: {name: string, age: number};
```

Příklad 6: Objektové typy

[19]

2.6.3 Aliasy typů

Namísto opětovného deklarování stejného komplexního typu objektu všude, kde se používá, TypeScript poskytuje jednoduchý způsob opětovného použití tohoto typu objektu. Vytvořením aliasu s klíčovým slovem type mu můžeme přiřadit datový typ. Pro vytvoření aliasu využíváme syntaxi: `type <název aliasu> = <typ>` . Můžeme vytvořit více aliasů typu, které definují stejný datový typ, a použít aliasy jako přiřazení k proměnným. [18] Vymýšlet alternativní názvy pro řetězec nemusí být příliš užitečné, ale toto lze provést s jakýmkoli typem. Aliasy typu jsou skutečně zejména užitečné pro odkazování na komplikované typy, které je třeba opakovat, jako typy objektů a n-tic. [33] Téměř všechny funkce `interface` jsou dostupné v `type`, hlavní rozdíl je v tom, že `type` nelze znovu otevřít a přidat nové vlastnosti oproti `interface`, které je vždy rozšiřitelné.

```
type MyString = string;
let myVar: MyString = 'Hi';
```

Příklad 7: Aliasy typů

[5]

2.6.4 Generické aliasy typů

Generika jsou základním rysem staticky psaných jazyků a umožňují vývojářům předávat typy jako parametry jinému typu, funkci nebo jiné struktuře. Když vývojář udělá ze své komponenty generickou komponentu, dá této komponentě schopnost přijímat a vynucovat typování, které se předává při použití komponenty, což zlepšuje flexibilitu kódu, umožňuje komponenty opakovaně použít a odstraňuje duplicitu. TypeScript plně podporuje generika jako způsob, jak zavést typovou bezpečnost do komponent, které přijímají argumenty a vrací hodnoty, jejichž typ bude neurčitý, dokud nebudou později použity v našem kódu. [34]

2.6.5 Typy funkcí

Výhodou na JavaScriptu je, že funkce lze přiřadit proměnným. Výhodou TypeScriptu je, že můžeme přesně ovládat druhy funkcí, které lze přiřadit proměnné. Děláme to pomocí typů funkcí, které určují typy argumentů a návratový typ funkce. Deklarování aliasu typu funkce provádíme pomocí syntaxe:

`type <názevAliasu> = (<parametr>: <typ>) => <návratový typ> .`

```
let myFunc: StringsToNumberFunction;
myFunc = function(firstName: string, lastName: string) {
  return firstName.length + lastName.length;
};

myFunc = function(whatever: string, blah: string) {
  return whatever.length - blah.length;
};
// Neither of these assignments results in a type error.
```

Příklad 8: Typy funkcí

[35]

2.6.6 Obecné funkce

V TypeScriptu se obecné typy používají, když chceme popsat shodu mezi dvěma hodnotami. Toho dosáhneme deklarováním parametru typu v podpisu funkce. [36] Obecně platí, že zápis generických funkcí pomocí function `názevFunkce<T>` nám umožňuje použít `T` v anotaci typu jako zástupný symbol typu. Později, když je funkce vyvolána, je `T` nahrazeno zadaným typem. [37]

2.7 Union

TypeScript nám umožňuje psát proměnné s různými úrovněmi typové specifity. Pokud chceme vynutit, že proměnná je řetězec, můžeme ji zadat jako řetězec. Tento typ je velmi specifický, protože TypeScript umožní proměnné mít pouze řetězcovou hodnotu. Na druhém konci spektra specifičnosti bychom mohli zadat proměnnou jako `any`. Tento typ je velmi nespecifický. Tyto dvě úrovně typové

specifičnosti fungují pro mnoho částí našich programů. Problém s typem `any` je, že jakákoli hodnota nemusí s naším programem fungovat. Abychom tento problém vyřešili, TypeScript nám umožňuje být flexibilní s tím, jak specifické jsou naše typy kombinací různých typů. Když kombinujeme typy, říká se tomu unie. [38]

2.7.1 Definování sjednoceného typu

Občas narazíme na knihovnu, která očekává, že parametr bude buď `number` nebo `string`. Sjednocení popisuje hodnotu, která může být jedním z několika typů. K oddělení jednotlivých typů používáme svislou čáru (`|`), takže `number | string | boolean` je typ hodnoty, která může být číslo, řetězec nebo logická hodnota. [39]

```
1 let ID: string | number;
2
3 ID = 1; // number
4 ID = '001'; // or string
```

Příklad 9: Typy funkcí

[35]

2.7.2 Zúžení

Může nastat situace, kdy chceme, aby funkce vykonala určitou činnost, pokud je parametr typu `string`, a pokud je typu `number`, vykonala činnost odlišnou. K tomu slouží ochrana typu `typeof <parametr> === '<typ>'`. TypeScript sleduje možné cesty provádění programu, aby analyzoval nejkonkrétnější typ hodnoty na dané pozici. [40]

2.7.3 Odvození návratového typu funkce

TypeScript odvozuje návratový typ funkce, takže pokud funkce vrací více než jeden typ dat, TypeScript odvodí návratový typ jako spojení všech možných návratových typů. Pokud chcete přiřadit návratovou hodnotu funkce proměnné, zadejte proměnnou jako spojení očekávaných typů návratů. [41]

2.7.4 Pole a sjednocení

Chceme-li zadat pole s typem sjednocení, použijeme závorky k zabalení typu sjednocení, např. `const array:(string | number)[] = ['a', 1, 'b', 2];`. Pole může obsahovat pouze prvky, které jsou členy typu sjednocení. Pokus o přidání prvku s nekompatibilním typem způsobí chybu. [42]

2.7.5 Společné páry klíčů a hodnot

Pokud máme hodnotu typu sjednocení, můžeme přistupovat pouze k členům, které jsou společné pro všechny typy ve sjednocení. [39]

```
union-types/exercises/common-key-value-pairs}
const batteryStatus: boolean | number = false;

batteryStatus.toString(); // No TypeScript error
batteryStatus.toFixed(2); // TypeScript error
```

Příklad 10: Společné páry klíčů a hodnot

[43]

2.7.6 Sjednocení s doslovnými typy

Sjednocení doslovného typu je užitečné, když chceme v programu vytvořit odlišné stavy. Můžeme použít literálové typy s TypeScript sjednocením. Kdekoli je funkce volána, budou jí předány pouze povolené hodnoty.

```
type Color = 'green' | 'yellow' | 'red';

function changeLight(color: Color) {
  // ...
}
```

Příklad 11: Sjednocení s doslovnými typy

[44]

2.8 Rozdíly mezi JavaScriptem a TypeScriptem

	JavaScript	TypeScript
1	Nepodporuje silně typované nebo statické typování.	Podporuje funkci silně typovaného nebo statického typování.
2	Společnost Netscape ji vyvinula v roce 1995.	Anders Hejlsberg ji vyvinul v roce 2012.
3	Zdrojový soubor jazyka JavaScript má příponu ".js".	Zdrojový soubor jazyka TypeScript je v příponě ".ts".
4	Spouští se přímo v prohlížeči.	Není přímo spuštěn v prohlížeči.
5	Je to pouze skriptovací jazyk.	Podporuje koncepci objektově orientovaného programování, jako jsou třídy, rozhraní, dědičnost, generika atd.
6	Nepodporuje volitelné parametry.	Podporuje volitelné parametry.
7	Je to interpretovaný jazyk, proto zvýrazňuje chyby za běhu.	Zkompiluje kód a zvýrazní chyby během vývoje.
8	JavaScript nepodporuje moduly.	TypeScript poskytuje podporu modulů.
9	Řetězce a čísla jsou v `this` objektem.	Řetězce a čísla jsou v `this` rozhraním.
10	JavaScript nepodporuje generika.	TypeScript podporuje generika.

Tabulka 1: Porovnání rozdílů mezi JavaScriptem a TypeScriptem

[45]

2.8.1 Proč zvolit TypeScript

Kompilační doba kontroly typu

Ve Vanilla JavaScriptu se ověření typu provádí za běhu. To však zvyšuje režii běhu, které se lze vyhnout provedením ověření při kompilaci.

Obrovské projekty nebo více vývojářů

TypeScript bezproblémově funguje ve velkých projektech nebo při spolupráci mnoha vývojářů.

Snadná práce s novými knihovnami nebo frameworky

Pokud pracujeme s Reactem a neznáme jeho rozhraní API, můžeme si pořídit IntelliSense, které nám pomůže identifikovat nová rozhraní a orientovat se v nich. Obě však nabízejí definice typů.

[46]

2.8.2 Proč zvolit JavaScript

Malé projekty

Pro malé projekty s menším počtem kódů může být TypeScript příliš náročný.

Podpora frameworků

Pokud TypeScript nepodporuje zvolený framework - EmberJS, je možné, že nebudeme moci využít jeho funkce.

Nástroje pro sestavení

Aby bylo možné vygenerovat konečný JavaScript, který má být spuštěn, vyžaduje TypeScript krok sestavení. Vývoj aplikací v jazyce JavaScript bez použití jakýchkoli nástrojů pro sestavení je však stále neobvyklejší.

Postup testování

Pokud talentovaní vývojáři v jazyce JavaScript již používají vývoj řízený testy, nemusí být výhody přechodu na jazyk TypeScript dostatečné, aby ospravedlnily náklady.

[46]

2.9 Výhody a nevýhody

Výhody

- TypeScript vždy upozorňuje na chyby kompilace v době vývoje (před kompilací). Z tohoto důvodu je výskyt chyb za běhu méně pravděpodobný, zatímco JavaScript je interpretovaný jazyk.
- TypeScript podporuje statické/silné typování. To znamená, že typovou správnost lze kontrolovat již při kompilaci. Tato funkce není v jazyce JavaScript k dispozici.

- TypeScript není nic jiného než JavaScript a některé další funkce, tj. funkce ES6. V cílovém prohlížeči nemusí být podporován, ale kompilátor jazyka TypeScript dokáže soubory .ts zkompilovat také do ES3, ES4 a ES5. [47]

Nevýhody

- Obecně platí, že kompilace kódu v jazyce TypeScript trvá dlouho.

[47]

2.9.1 Shrnutí

TypeScript je moderní vývojový JavaScript, zatímco JavaScript je skriptovací jazyk, který pomáhá vytvářet interaktivní webové stránky. TypeScript používá k popisu používaných dat pojmy jako typy a rozhraní, zatímco JavaScript žádný takový pojem nemá. Dá se říci, že pokud zkušený vývojář pracuje na relativně malých projektech , pak je JavaScript ideální. Pokud však máme znalosti a zkušenosti vývojového týmu, pak je Typescript nejvhodnější volbou. [48]

3 React

3.1 O Reactu

React.js je JavaScriptová knihovna vyvinutá společností Facebook. Mezi nejvýznamější výhody patří:

- Rychlost. Aplikace vytvořené v Reactu zvládnou složité aktualizace a přesto se působí rychle a responsivně.
- Je modulární. Místo psaní velkých hustých souborů kódu, můžete psát mnoho menších opakovaně použitelných souborů. Modularita Reactu může být zároveň řešením problémů s údržbou JavaScriptu.
- Je škálovatelný. React funguje nejlépe ve velkých programech, které zobrazují mnoho měnících se dat.
- Je flexibilní. React můžeme použít pro zajímavé projekty, které nemají nic společného s tvorbou webové aplikace. [49]

3.2 JSX

JSX je syntaktické rozšíření JavaScriptu. Doporučuje se použít jej s Reactem k popisu toho, jak by mělo vypadat uživatelské rozhraní. JSX může připomínat jazyk šablon, ale přichází s plnou silou JavaScriptu. React podporuje skutečnost, že logika vykreslování je neodmyslitelně spojena s další logikou uživatelského rozhraní: jak se zachází s událostmi, jak se stav mění v průběhu času a jak jsou data připravena k zobrazení. React nevyžaduje použití JSX, ale většina lidí to považuje za užitečné jako vizuální pomůcku při práci s uživatelským rozhraním v kódu JavaScript. Umožňuje to také Reactu zobrazovat užitečnější chybové a varovné zprávy. [50]

```
1 const h1 = <h1>Hello world</h1>;
```

Příklad 12: JSX syntaxe

[51]

3.2.1 Prvek JSX

Prvky jsou to, z čeho jsou „vyrobeny“ komponenty. [52] Syntaxe Reactu je rozdělena na komponenty. Pro čitelnost můžeme JSX rozdělit na více řádků. I když to není nutné, při provádění tohoto postupu se také doporučuje zabalit jej do závorek, abychom se vyhnuli nástrahám automatického vkládání středníků. [50]

3.2.2 Atributy

Prvky JSX mohou mít atributy, stejně jako prvky HTML. Atribut JSX je zapsán pomocí syntaxe podobné HTML: `nazev-atributu="hodnota-atributu"`. Jeden prvek JSX může mít mnoho atributů, stejně jako v HTML. [53] Při vkládání JavaScript výrazu do atributu neuvádíme uvozovky kolem složených závorek. Měli bychom použít buď uvozovky (pro hodnoty řetězce) nebo složené závorky (pro výrazy), ale ne obojí ve stejném atributu. [50]

3.2.3 Vnoření prvků

V Reactu můžeme komponenty vkládat do sebe. To pomáhá při vytváření složitějších uživatelských rozhraní. Komponenty, které jsou vnořeny do nadřazených komponent, se nazývají podřízené komponenty. Klíčová slova Import a Export usnadňují vnořování komponent.

- Export – Toto klíčové slovo se používá k exportu konkrétního modulu nebo souboru a jeho použití v jiném modulu.
- Import – Toto klíčové slovo se používá k importu konkrétního modulu nebo souboru a jeho použití ve stávajícím modulu. [54]

```
1 const theExample = (
2    <a href="https://www.example.com">
3      <h1>Click me!</h1>
4    </a>
5  );
```

Příklad 13: Vnoření prvků

[55]

3.2.4 Vykreslování

Jedná se o popis toho, co si přejeme zobrazit na obrazovce. [52]

```
1 ReactDOM.render(<h1>Hello world</h1>,
2 document.getElementById('app'));
```

Příklad 14: Vykreslování

[56]

3.2.5 ReactDOM.render()

Knihovna JavaScript `react-dom`, někdy nazývaná `ReactDOM`, vykresluje prvky JSX do DOM tím, že vezme výraz JSX, vytvoří odpovídající strom uzlů DOM a přidá tento strom do DOM. První argument je výraz JSX, který se má zkompilovat a vykreslit. Druhý argument je prvek HTML, ke kterému jej chceme připojit. [57]

3.2.6 Vkládání výrazů

Pokud chceme využívat JavaScript v JSX, vložíme ho do složených závorek. Do složených závorek můžeme vložit jakýkoli platný JavaScriptový výraz. [50]

3.2.7 Virtuální DOM

React DOM porovnává prvek a jeho potomky s předchozím a aplikuje pouze aktualizace DOM nezbytné k uvedení DOM do požadovaného stavu. I když při každém tiku vytvoříme prvek popisující celý strom uživatelského rozhraní, pomocí React DOM se aktualizuje pouze textový uzel, jehož obsah se změnil. [52]

3.2.8 Gramatika

Protože JSX je blíže JavaScriptu než HTML, React DOM používá konvenci pojmenovávání vlastností camelCase namísto názvů atributů HTML. [50] Gramatika v JSX je většinou stejná jako v HTML, ale existují jemné rozdíly, na které je třeba dávat pozor. Pravděpodobně nejčastější z nich zahrnuje slovo `class`. V HTML je běžné používat `class` jako název atributu. V JSX nemůžeme použít slovo `class`!

Místo toho musíme použít `className`. Je to proto, že JSX se překládá do JavaScriptu a `class` je v JavaScriptu vyhrazené slovo. [58]

3.2.9 Sebe-uzavíratelné tagy

Většina prvků HTML používá dvě značky: úvodní značku (`<div>`) a závěrečnou značku (`</div>`). Některé prvky HTML, jako je `<img>` a `<input>`, však používají pouze jednu značku. Tag, který patří k elementu single-tag, není úvodní ani koncový tag, ale samouzavírací. Když píšeme samouzavírací značku v HTML, je volitelné vložit lomítko bezprostředně před poslední lomenou závorku, ale v JSX lomítko zahrnout musíme. Pokud v JSX napíšeme samouzavírací značku a zapomeneme lomítko, vyvoláme chybu. [59] Zároveň každá komponenta Reactu může být samouzavírací: `<div />`. `<div></div>` je také ekvivalent. [60]

3.2.10 JavaScript, proměnné a atributy

Jakýkoli text mezi značkami JSX bude čten jako textový obsah, nikoli jako JavaScript. Aby bylo možné text číst jako JavaScript, musí být kód vložen mezi složené závorky. [57] Když vložíme JavaScript do JSX, tento JavaScript je součástí stejného prostředí jako zbytek JavaScriptu v našem souboru. To znamená, že můžeme přistupovat k proměnným uvnitř výrazu JSX, i když byly tyto proměnné deklarovány venku. [61] Při psaní JSX je běžné nastavit atributy pomocí vložených proměnných JavaScriptu. [57]

3.2.11 Posluchači událostí

V JSX jsou posluchače událostí specifikovány jako atributy prvků. Název atributu posluchače události by měl být zapsán v camelCase, například `onClick` pro událost `onclick` a `onMouseOver` pro událost `onmouseover`. Hodnota atributu posluchače události by měla být funkce. Funkce posluchače událostí mohou být deklarovány inline nebo jako proměnné a mohou volitelně převzít jeden argument představující událost.

```
1 <img onClick={myFunc} />
```

Příklad 15: Posluchači událostí

[57]

3.2.12 If

`if-else` příkazy nefungují uvnitř JSX. Je to proto, že JSX je pouze pro volání funkcí a konstrukci objektů. [62]

Existují tři způsoby, jak vyjádřit podmínky pro použití s prvky JSX:

1. ternární ve složených závorkách v JSX
2. příkaz if mimo prvek JSX nebo
3. `&&` operátor. [57]

3.2.13 Key

„Klíč“ je speciální řetězcový atribut, který musíte zahrnout při vytváření seznamů prvků v Reactu. Klíče se používají v Reactu k identifikaci, které položky v seznamu byly změněny, aktualizovány nebo odstraněny. Jinými slovy, můžeme říci, že klíče se používají k poskytnutí identity prvkům v seznamech. Jako klíč se doporučuje použít řetězec, který jednoznačně identifikuje položky v seznamu. [63] `keys` nedělají nic, co vidíme! React je používá interně ke sledování seznamů. Pokud nepoužíváme klíče, když se to má, React může náhodně zakódovat naše položky seznamu do nesprávného pořadí.

```
1 <ul>
2   <li key="li-01">Example1</li>
3   <li key="li-02">Example2</li>
4   <li key="li-03">Example3</li>
5 </ul>
```

Příklad 16: Klíče

[64]

3.3 React.createElement

React kód můžeme psát bez použití JSX! Většina React programátorů JSX používá, ale je možné psát kód i bez něj. [65] Funkci `React.createElement()` používá React k vytvoření virtuálních prvků DOM z JSX. Když je JSX zkompilován, je nahrazen voláním `React.createElement()`.

```
const h1 = React.createElement(
  "h1",
  null,
  "Hello world"
);
```

Příklad 17: Výraz bez JSX

[57]

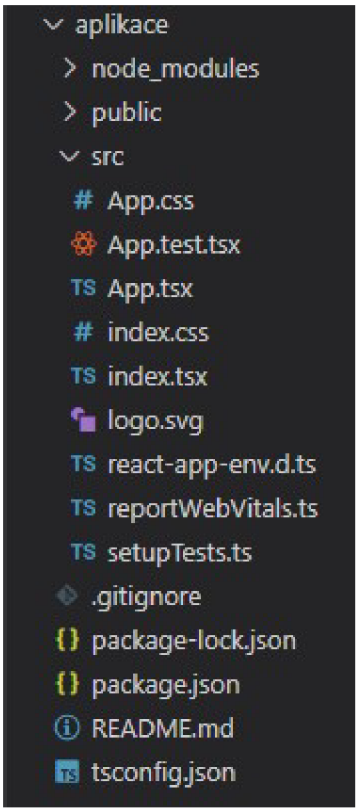
4 Praktická část

4.1 TypeScript-React instalace

Výhodou je, že jakmile nainstalujeme `Node.JS` a editor (v našem případě **Visual Studio Code**), instalace samotného TypeScript-Reactu je provedena jediným příkazem do konzole.

`npx create-react-app nazev-aplikace -template typescript` Editor následně zajistí vygenerování všech potřebných souborů, včetně jejich obsahu a knihoven. [66]

V určitých verzích Reactu tento příkaz však nefunguje a je nutné do příkazu specifikovat instalovanou verzi `create-react-app@5.0.1`.



Obrázek 4: Soubory vytvořené při instalaci

Nejpodstatnější částí pro tvorbu naší aplikace je složka **src**. Zde ponecháme **App.css**, **App.tsx**, **index.css**, **index.tsx** a **reportWebVitals.ts**. Zbytek souborů nebudeme potřebovat, a tak je odstraníme.

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
```

```
3  import App from './App';
4  import reportWebVitals from './reportWebVitals';
5
6  ReactDOM.render(
7    <React.StrictMode>
8      <App />
9    </React.StrictMode>,
10   document.getElementById('root')
11 );
12 reportWebVitals();
```

Příklad 18: index.tsx

Před psaním vlastního kódu odstraníme ze souboru **index.tsx** přebytečné importy již neexistujících souborů. Zbytek souboru ponecháme beze změny, neboť zajišťuje vykreslení obsahu, který vrací soubor App.tsx, ve kterém se bude nacházet hlavní část kódu pro naší aplikaci.

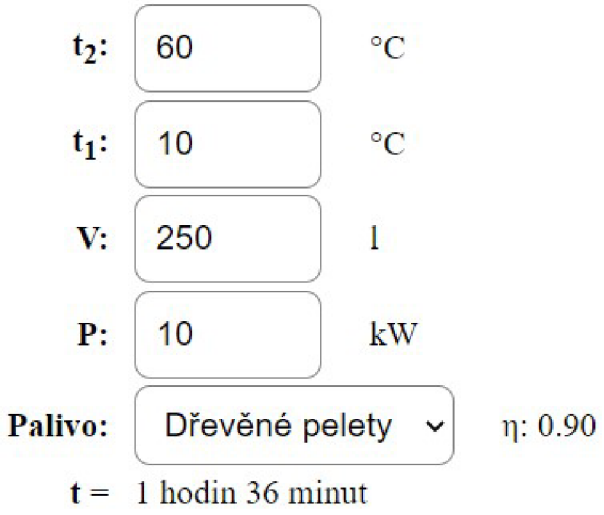
Ve složce **public** bychom našli soubor **index.html**, který je taktéž generovaný při instalaci. Jedná se však pouze o prázdnou stránku, obsahující pár meta tagů, neboť vše budeme tvořit uvnitř `App.tsx`

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import App from './App';
4  import reportWebVitals from './reportWebVitals';
5
6  ReactDOM.render(
7    <React.StrictMode>
8      <App />
9    </React.StrictMode>,
10   document.getElementById('root')
11 );
12 reportWebVitals();
```

Příklad 19: index.tsx

4.2 Bojler

Aplikace, vypočítávající dobu ohřevu vody v bojleru, v závislosti na zadaných vstupních parametrech.



Obrázek 5: Vstupy a výsledek pro aplikaci Bojler

4.2.1 Import

```
1 import React, {ChangeEvent, useState} from 'react';
```

Příklad 20: Import pro bojler

Pro aplikaci využijeme knihovny `React`, `ChangeEvent` a `useState`. S výjimkou `ChangeEvent` budeme tyto tři knihovny využívat ve všech našich programech.

4.2.2 useState

```
1 const App = () => {
2 // input
3 const[vystupniTeplota,setVystupniTeplota]=
4  useState<number>(55);
5 const[vstupniTeplota,setVstupniTeplota]=useState<number>(10);
6 const[objemVody,setObjemVody]=useState<number>(200);
7 const[prikon,setPrikon]=useState<number>(15);
```

```
8 const[ucinnost,setUcinnost]=useState<number>(0.98);
9 const[errorStatement,setErrorStatement]=useState<string>("");
```

Příklad 21: useState pro bojler

Pomocí `useState` deklarujeme „stavovou proměnnou“. Normálně proměnné „zmizí“, když funkce skončí, ale stavové proměnné jsou zachovány pomocí Reactu. Jediným argumentem Hooku `useState()` je počáteční stav. Na rozdíl od tříd, stav nemusí být objektem. Můžeme si ponechat číslo nebo řetězec, pokud je to vše, co potřebujeme. [67]

TypeScript dokáže v mnoha případech chytře odvodit typ pro useState, což je skvělé. Když TypeScript nedokáže odvodit typ, můžeme jej předat jako obecný parametr. [68]

4.2.3 handleChange

```
1  // handleChange pro input
2  const handleChange = (event:ChangeEvent<HTMLInputElement>):
3  void => {
4
5    errorMessage(event.target.name, event.target.value)
6    switch (event.target.name) {
7      case 'vystupniTeplota':
8        const valueVystupniTeplota = event.target.value;
9        setVystupniTeplota(Number(valueVystupniTeplota));
10       break;
11
12     case 'vstupniTeplota':
13       const valueVstupniTeplota = event.target.value;
14       setVstupniTeplota(Number(valueVstupniTeplota));
15       break;
16    // dalsi moznosti
17   };
18 };
19 // selectChange pro vyber paliva (ucinnosti)
20 const selectChange = (event:
```

```
21  React.ChangeEvent<HTMLSelectElement>) => {
22    const valueUcinnost = event.target.value;
23    setUcinnost(Number(valueUcinnost));
24  }
```

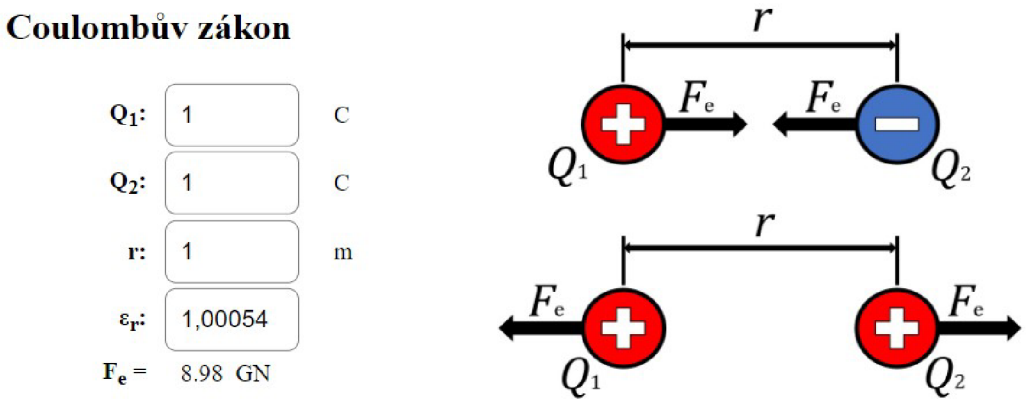
Příklad 22: handleChange a selectChange pro bojler

`ChangeEvent` využijeme k aktualizování vykreslovaného obsahu při každé změně jednoho ze vstupů. Rozhodování, který `input` upravujeme, provedeme pomocí `switch`, na základě hodnoty atributu `name`.

Chceme-li zadat událost `onChange` prvku v Reactu, nastavíme jeho typ na `React.ChangeEvent<HTMLInputElement>`. Typ `ChangeEvent` má vlastnost `target`, která odkazuje na prvek. Hodnota prvku je přístupná na `event.target.value`. [69]

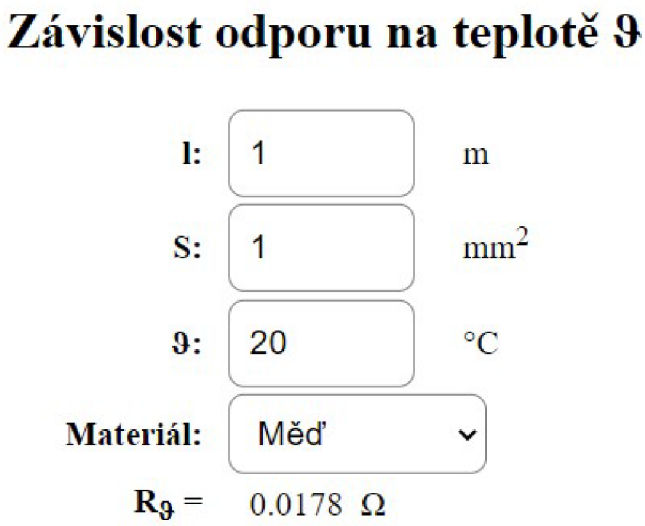
4.3 Fyzikální příklady

Cílem aplikace je pomocí vstupních parametrů spočítat základní fyzikální příklady pro střední školy z oboru elektroniky, a to Coulombův zákon a závislost elektrického odporu vodiče na teplotě, jeho délce a průřezu.



Obrázek 6: Výpočet prvního příkladu (Coulombův zákon)

Q_1 a Q_2 jsou velikosti jednotlivých nábojů, které na sebe vzájemně silově působí, r udává vzdálenost mezi bodovými náboji (zanedbáváme všechny vlastnosti až na hodnoty náboje). ϵ_r udává relativní permitivitu (míra odporu prostředí oproti vakuu). Výsledná síla vzájemného působení F_e je pak udávána v jednotkách Newton.



Obrázek 7: Výpočet druhého příkladu (Závislost odporu na teplotě)

Základní vzoreček pro závislost odporu na teplotě předpokládá referenční teplotu a odpor materiálu při této teplotě. V našem případě jako referenční teplotu volíme 20 °C a odpor pro tuto teplotu vypočítáváme z hodnoty měrného odporu (podle materiálu při teplotě 20 °C), délky vodiče l a jeho průřezu S. ϑ udává teplotu, při které chceme znát výsledný odpor R_ϑ

4.3.1 Import

```
1 import React, {ChangeEvent, useState} from 'react';
```

Příklad 23: Import pro příklady

Jako import využijeme tři základní funkce pro tento typ aplikací, a to `React`, `ChangeEvent` a `useState` z knihovny `react`.

4.3.2 useState

```
1  // Coulombuv zakon
2  const[Q1,setQ1]=useState<number>(1);
3  const[Q2,setQ2]=useState<number>(1);
4  const[r,setR]=useState<number>(1);
5  const[permitivitaProstredi,setPermitivitaProstredi]=
6  useState<number>(1.00054);
7
8  // Zavislost odporu na teplote
9  const[l,setL]=useState<number>(1);
10 const[S,setS]=useState<number>(1);
11 const[t,setT]=useState<number>(20);
12 const[rezistivita,setRezistivita]=useState<number>(0.0178);
13 const[R20,setR20]=useState<number>(0.0178);
14 const[tepSoucElOdporu,setTepSoucElOdporu]=
15 useState<number>(4.0);
```

Příklad 24: useState pro příklady

V našem případě pro `useState` píšeme typy převážně kvůli přehlednosti pro nás a pro snadnější čtení ostatním. TypeScript (zde TSX - kombinace JSX a TypeScriptu) je totiž schopen sám z defaultní hodnoty odvodit, jakého bude proměnná typu. [2]

Zároveň díky tomu, že většina editorů podporujících TypeScript dokáže při najetí myší na proměnnou zobrazit její typ, nemusíme při psaní kódu přemýšlet, jakou hodnotu daná proměnná nese.

4.3.3 handleChange

```
1  const handleChangeCoulomb = (event:
2  ChangeEvent<HTMLInputElement>): void => {
3
4    switch (event.target.name) {
5      case 'Q1':
6      const valueQ1 = Number(event.target.value);
```

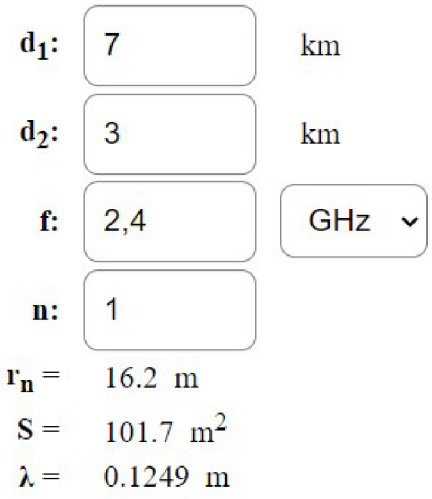
```
7       setQ1(valueQ1);
8       break;
9     // dalsi moznosti
10  }
11
12  const selectChangeTeplotniZavislost = (event:
13  React.ChangeEvent<HTMLSelectElement>) => {
14
15    switch (event.target.value) {
16      case 'Med':
17        setRezistivita(0.0178);
18        setR20((0.0178*l)/S);
19        setTepSoucElOdporu(4.0);
20       break;
21     // dalsi moznosti
22  }
```

Příklad 25: handleChange a selectChange pro příklady

Nevýhodou, se kterou musíme při práci s `useState` počítat je, že jakmile spustíme funkci `handleChange`, pracujeme s hodnotami `useState`, které byly při spuštění funkce, a to i přes to, že bychom hodnotu uvnitř funkce nechali změnit. Pokud bychom měli výpočet závislý na vstupu a chtěli, aby se s každou změnou v `input` znovu provedl a vykreslil výsledek, dostávali bychom požadované hodnoty vždy o krok později. Variantou, jak tento problém vyřešit, je využívat všude návratovou hodnotu prvku `event.target.value`, nebo tuto hodnotu zapsat do přehledněji pojmenované proměnné `const hodnotaTeploty = event.target.value`.

4.4 Výpočet Fresnelovy zóny

Tato aplikace slouží k výpočtu plochy Fresnelovy zóny v daném místě z důvodu určení povoleného zastínění, které je závislé na použité přenosové technologii. Zároveň udává poloměr Fresnelovy zóny a délku procházející vlny. d_1 a d_2 určuje vzdálenost, ve které se nachází vypočítávaná plocha S o poloměru r_n. f udává frekvenci v GHz nebo MHz o délce vlny λ. Vrstvu zóny udává n (většinou 1).



Obrázek 8: Vstupy a výsledek pro aplikaci výpočet Fresnelovy zóny

4.4.1 Import

```
1 import React, { useState, ChangeEvent } from 'react';
```

Příklad 26: Import pro výpočet Fresnelovy zóny

Protože `useState` a `ChangeEvent` jsou pojmenované exporty z knihovny texttreact, uvádíme je při importu ve složených závorkách. Název importu by měl být stejný jako název exportu. Oproti tomu `React` je `default export` a může nést libovolné pojmenování. [70]

4.4.2 useState

```
1   const[f,setF]=useState<number>(10.5);
2   const[fN,setFN]=useState<number>(1);
3   const[d1,setD1]=useState<number>(3);
4   const[d2,setD2]=useState<number>(2);
5   const[n,setN]=useState<number>(1);
6   const[rN,setRN]=useState<number>(5.8);
```

Příklad 27: useState pro výpočet Fresnelovy zóny

4.4.3 handleChange

```
const handleChange = (event: ChangeEvent<HTMLInputElement>):
 void => {

  switch (event.target.name) {
   case 'f':
     setF(Number(event.target.value));
     setRN(17.3*Math.sqrt((n*d1*d2)/
      ((d1+d2)*Number(event.target.value)*fN)));
     break;
     // dalsi moznosti
}

// selectChange pro vyber jednotek (GHz/MHz)
const selectChange = (event:
 React.ChangeEvent<HTMLSelectElement>) => {
   setFN(Number(event.target.value));
   setRN(17.3*Math.sqrt((n*d1*d2)/
    ((d1+d2)*f*Number(event.target.value))));
}
```

Příklad 28: handleChange a selectChange pro výpočet Fresnelovy zóny

4.5 Převodník barev

Převodník barev na RGB umožňuje výběr vstupní hodnoty ručním zadáním jako input, nebo pomocí color pickeru. Po přepnutí umožní převodník převádět z RGBA do HSL a osmimístného Hex kódu.

Obrázek 9: Převod z HEX na RGB

Obrázek 10: Převod z RGBA na HEX a HSL

4.5.1 Import

```
1 import React, {useState, ChangeEvent} from 'react';
2 import { SketchPicker, ColorResult } from 'react-color';
```

Příklad 29: Import pro převodník barev

Mimo našich standardních 3 funkcí využijeme nově `SketchPicker` a `ColorResult` z knihovny `react-color`. `SketchPicker` nám umožní vložit do aplikace `picker` a `ColorResult` nám umožňuje získávat výsledky jako `string` ve formátu `hex`. Tuto knihovnu je nutné nejprve dodatečně instalovat pomocí příkazu `npm i -save-dev @types/react-color`.

4.5.2 useState

```
1  const[disabled, setDisabled] = useState(true);
2  const[color, setColor] = useState<string>("#ffffff");
3  const[red, setRed] = useState<number>(255);
4  const[green, setGreen] = useState<number>(255);
5  const[blue, setBlue] = useState<number>(255);
6  const[alfa, setAlfa] = useState<number>(1);
7  const[H, setH] = useState<string>("0");
8  const[S, setS] = useState<string>("0 %");
9  const[L, setL] = useState<string>("100 %");
```

Příklad 30: useState pro převodník barev

Tlačítko na změnu převodu není odkaz na jinou stránku, ale pouze přepínač proměnné `disabled` mezi hodnotami `true` a `false`. Podle hodnoty této proměnné následně určujeme, který obsah chceme vykreslovat a který ne. Proměnné `red`, `green`, `blue` a `alfa` udávají jednotlivé složky barev, které se po převodu ukládají jako Hex kód do proměnné `color`, se kterou pracujeme při nastavení vzorového pozadí. Složky `H` (Hue), `S` (Saturation) a `L` (Light) jsou pak vypočteny ze zastoupení jednotlivých RGB složek.

4.5.3 handleChange

```
1  // handleChange pro rucni zadani hodnot
2    const handleChange = (event: ChangeEvent<HTMLInputElement>):
3     void => {
4     const value = event.target.value;
5        switch (event.target.name) {
6           case 'hex':
7              setColor(`#${value}`);
8                 setRed(parseInt(value.slice(0, 2), 16));
9                 setGreen(parseInt(value.slice(2, 4), 16));
10                setBlue(parseInt(value.slice(4, 6), 16));
11                setAlfa(1);
12                break;
```

```
13          case 'r':
14                setRed(Number(value));
15                toHex(Number(value), green, blue, alfa);
16                break;
17          // dalsi moznosti
```

Příklad 31: handleChane pro ruční zadávání hodnot

Funkce je spouštěna při každé změně libovolného ze vstupů. Jejím cílem je pomocí `switch` zjistit, která hodnota byla změněna a podle toho provést přepočet.

Picker a v našem případě i celá stránka přijímá hodnotu barvy jako `HEX`, ale `input` vrací `string` i v případě, že je specifikovaný jako `type="number"`. Z toho důvodu, musíme dbát na správný typ hodnot, abychom mohli provádět matematické operace nutné pro přepočet a vyhnuli se errorům.

Převod na číslo pomocí `Number(event.target.value)` musíme provádět i u všech ostatních aplikací, kde s číselným vstupem chceme provádět výpočty. Zde má TypeScript svou největší výhodu, kdy nám nedovolí zapsat do proměnné hodnotu ve špatném typu, čímž předcházíme errorům. [6]

```
1  // Vyber barvy pres picker a prevod do RGB
2  const handleChangePicker = (color: ColorResult) => {
3    const valueColor = color.hex;
4    setColor(valueColor);
5    setRed(parseInt(valueColor.slice(1, 3), 16));
6    setGreen(parseInt(valueColor.slice(3, 5), 16));
7    setBlue(parseInt(valueColor.slice(5, 7), 16));
8    setAlfa(1);
9  };
```

Příklad 32: handleChange pro výběr barvy v pickeru

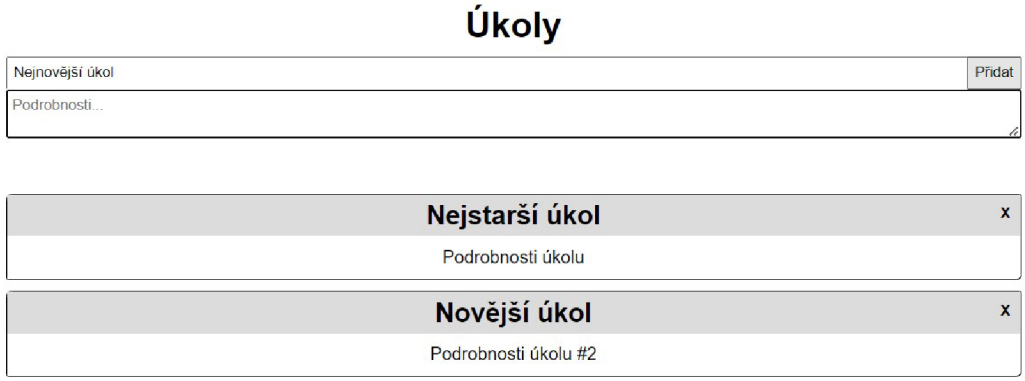
```
1 const toHex = (red:number, green:number, blue:number,
2   alfa:number) => {
```

Příklad 33: Parametry funkce pro výpočet HEX a HSL

TypeScript v určitých editorech dovede pomoci s nalezením správného typu v dokumentaci. Pokud by se nám nepodařilo najít vhodný `type` a TypeScript nám neustále vracel chybové hlášení, můžeme využít `any`. [2]

4.6 Úkolníček

Aplikace slouží jako příklad databáze využívající cookies jako způsob ukládání.



Obrázek 11: Úkolníček se seznamem úkolů

4.6.1 Import

```
1 import React, {FC, useState } from 'react';
2 import TodoTask from './Components/TodoTask';
3 import {interfaceTask} from './interface'
```

Příklad 34: Import pro úkolníček

Oproti našim ostatním aplikacím nevyužíváme `ChangeEvent`, ale `FC`.

`FC` (`FunctionComponent`) poskytuje implicitní definici dětí. To znamená, že definování komponenty pomocí `React.FC` způsobí, že implicitně převezme potomky typu `ReactNode`. To může způsobit zmatek, pokud naše součástka není určena pro děti! [71]

4.6.2 useState

```
let ces = getcookie();
const[task,setTask]=useState<string>("");
const[description,setDescription]=useState<string>("");
const[todoList,setTodoList]=useState<interfaceTask[]>(ces);
```

Příklad 35: useState pro úkolníček

Jako první krok při načtení stránky provedeme stažení cookies, které mohou posloužit jako defaultní hodnota pro `todoList`. Nejprve musíme pomocí funkce zjistit, zda cookies obsahují uložená data. Pokud zjistíme, že ano, stále jsou v našem případě uložena jako `string` ve formátu `<title>=<description>` a oddělena `;`. Abychom dostali požadované hodnoty, rozložíme cookies do pole po jednotlivých záznamech. Tyto řetězce budeme postupně rozebírat na název a popisek úkolu a ukládat je jako objekty do pole, které poslouží jako defaultní hodnota pro `todoList`.

4.6.3 interfaceTask

```
export interface interfaceTask {
    taskName: string;
    description: string;
}
```

Příklad 36: interface.ts

Vlastní `interface` můžeme definovat přímo v souboru, nebo samostatně. Druhá možnost je přehlednější, zejména v případě, kdy interface využíváme napříč několika soubory. V našem případě využíváme druhou možnost a export uvádíme jako jmenný. Z toho důvodu musíme `import` pojmenovat stejně a uzavřít ho do složených závorek. [70]

4.6.4 Vykreslení úkolů

```
import React from "react";
import { interfaceTask } from "../interface";

interface Props {
    task: interfaceTask;
    completeTask(taskNameDelete: string): void;
}
const TodoTask = ({ task, completeTask }: Props) => {
```

Příklad 37: TodoTask.tsx (jednotlivé úkoly)

Vykreslování jednotlivých úkolů je spravováno v samostatném souboru.

```
<div className="todoList">
 {todoList.map((task: interfaceTask, key: number) => {
  return <TodoTask key={key} task={task}
   completeTask={completeTask} />;
 })}
</div>
```

Příklad 38: todoList (oblast se všemi úkoly)

List všech úkolů pak dostaneme jako mapu přes všechny hodnoty pole.

4.6.5 Nový úkol

```
const addTask = (): void => {
  const newTask = {taskName:task, description:description};
  setTodoList([...todoList, newTask]);
  setTask("");
  setDescription("");
  // ulozeni do cookies
  document.cookie = task + "=" + description;
  }
```

Příklad 39: Přidání nového úkolu

Přidání nového úkolu řešíme tak, že ke zbytku (výčtu všech úkolů) přidáme aktuální obsah našich vstupů. Samozřejmě nesmíme při jakémkoliv přidání nebo odebrání úkolu zapomenout na úpravu cookies, abychom je udržovali stále aktuální.

4.7 Vytvoření webu

Pro hosting jednotlivých aplikací jsem zvolil web `vercel.com`, který umožňuje přímo z editoru nahrát aplikaci jako projekt. Nejprve se zaregistrujeme na webových stránkách. Následně ve složce s aplikacemi provedeme globální instalaci přes příkaz `npm i -g vercel` nebo `yarn global add vercel`. Po instalaci se přemístíme do složky s aplikací, kterou si přejeme nahrát na webový server a zadáme příkaz `vercel login`, kde zvolíme metodu přihlášení (v našem případě email). Po přihlášení zadáme příkaz `vercel`. Následně odpovíme na sérii otázek, což v našem případě znamená, že si přejeme vytvořit nový projekt, který není navázaný na žádný z předchozích projektů. Zvolíme si jméno a protože již jsme ve složce s programem, můžeme cestu ponechat defaultní. Následně zamítneme přepsání přednastavené volby pro spuštění. Tímto vytvoříme webový hosting pro aplikaci, který se zároveň sám postará o veškerou kompilaci TS a TSX souborů. V našem případě jsou všechny aplikace vedené jako jednotlivé projekty pod stejným účtem, na které přistupujeme z webového rozcestníku. [72]

Obrázek 12: Rozcestník na webové aplikace

5 Závěr

Bakalářská práce se zabývá syntaxí jazyka TypeScript a jeho propojením s jazykem JSX v rámci Reactu. Zaměřuje se na výhody, které při práci s ním poskytuje, stejně jako potenciální problémy a představuje rozdíly oproti JavaScriptu jak v syntaxi, které jsou viditelné na první pohled, tak přímo uvnitř kompilátoru.

Teoretická část se zabývá představením jednotlivých typů, se kterými pracujeme i v samotném JavaScriptu, zmiňuje problémy, které JavaScript má, a nastiňuje důležitost kontroly, jakého typu naše proměnné jsou. Dále popisuje využití typů pro pole a funkce, a to od úplných základů po komplexní operace.

Dále se teoretická část zabývá Reactem, jakožto jedním z nejpoužívanějších nástrojů pro vývoj webových aplikací, popisuje jeho výhody a představuje práci s JSX.

Praktická část je zaměřena na sadu demonstrativních aplikací, které mají nastínit, jak vypadá využití TypeScriptu v TSX a poskytnout podklady pro zhodnocení kladů, které Microsoft jako tvůrce TypeScriptu propaguje.

TypeScript je užitečný nástroj a jeho aktivní používání je patrné ze spousty zahraničních webů, které se TypeScriptem zabývají. Jeho hlavní výhoda spočívá v prevenci, kdy jsme jako programátor upozorněni na většinu potenciálních chyb již při kompilaci. S využitím správného editoru můžeme mít hlídání chyb přímo v reálném čase, a odhalit tak potenciální chybu již ve chvíli, kdy ji vytvoříme. To je užitečné zejména u velikých aplikací, které obsahují tisíce řádků kódu a je zde vysoké riziko chyby, nebo pouhého přepsání se.

Pokud však pracujeme na menších aplikacích, které se obejdou bez polí a objektů, výhody často ani nemusíme zaznamenat. Zejména pokud využíváme editor, který disponuje podobným typem kontroly.

V kombinaci s Reactem pak nastává největší problém a to typování specifických eventů, bez kterých se v dnešní době u webových aplikací neobejdeme. Pokud pracujeme s knihovnou, která má kvalitní dokumentaci ke každé funkci, chytré editory jako Visual Studio Code nám pomohou se správným specifikováním typu. V opačném případě i správně napsaná část kódu bude při kompilaci vracet chybové hlášení, protože postrádá správné definování typu. Způsobem, jak tento problém obejít, je definovat typ jako `any`, ale přicházíme tím o ochranu.

Obzvláště, pokud nemáme zkušenosti s JavaScriptem, může pro nás být TypeScript značně chaotický, neboť se jedná o rozšíření syntaxe JavaScriptu, se kterým souvisí přidání sady pravidel, která musíme dodržovat. Většina dostupných materiálů k TypeScriptu zároveň počítá s předchozí znalostí JavaScriptu. TypeScript je tak užitečným nástrojem, ale jeho výhody nejvíce pocítí zejména zkušenější programátoři velkých projektů.

Seznam použité literatury a zdrojů

[1] From JavaScript to TypeScript. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-25]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/introduction-to-typescript/exercises/from-javascript-to-typescript*

[2] TypeScript handbook. *TypeScript [online]. Redmond (Washington): Microsoft, c2012-2022, 24. 6. 2022 [cit. 2022-06-25]. Dostupné z: https://www.typescriptlang.org/assets/typescript-handbook.pdf*

[3] Types. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-25]. Dostupné z: https://www.codecademy.com/learn/learn-typescript/modules/learn-typescript-types/cheatsheet*

[4] Basic Types. *TypeScript [online]. Redmond (Washington): Microsoft, c2012-2022 [cit. 2022-06-25]. Dostupné z: https://www.typescriptlang.org/docs/handbook/basic-types.html*

[5] Advanced Types. *TypeScript [online]. Redmond (Washington): Microsoft, c2012-2022 [cit. 2022-06-25]. Dostupné z: https://www.typescriptlang.org/docs/handbook/advanced-types.html*

[6] Understanding Type Annotations in TypeScript. *TypeScript Tutorial [online]. TypeScript Tutorial, c2022 [cit. 2022-06-25]. Dostupné z: https://www.typescripttutorial.net/typescript-tutorial/typescript-type-annotations*

[7] Type Shapes. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/introduction-to-typescript/exercises/type-shapes*

[8] TypeScript for JavaScript Programmers. *TypeScript [online]. Redmond (Washington): Microsoft, c2012-2022 [cit. 2022-06-26]. Dostupné z: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html*

[9] TypeScript - Functions. *Tutorials Point [online]. India: Tutorials Point, c2022 [cit. 2022-06-26]. Dostupné z: https://www.tutorialspoint.com/typescript/typescript_functions.htm*

[10] Void Return Type. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-functions/exercises/void-return-type*

[11] Understanding and using the void type. *Learn TypeScript [online]. Rippon, c2022 [cit. 2022-06-26]. Dostupné z: https://learntypescript.dev/03/l3-void*

[12] TypeScript Optional Parameters. *TypeScript Tutorial [online]. TypeScript Tutorial, c2022 [cit. 2022-06-26]. Dostupné z: https://www.typescripttutorial.net/typescript-tutorial/typescript-optional-parameters*

[13] TypeScript - Functions. *TutorialsTeacher [online]. TutorialsTeacher, c2022 [cit. 2022-06-26]. Dostupné z: https://www.tutorialsteacher.com/typescript/typescript-function*

[14] Default Parameters. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-functions/exercises/default-parameters*

[15] Documenting Functions. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-functions/exercises/documenting-functions*

[16] TypeScript - Arrays. *TutorialsTeacher [online]. TutorialsTeacher, c2022 [cit. 2022-06-26]. Dostupné z: https://www.tutorialsteacher.com/typescript/typescript-array*

[17] TypeScript - Multidimensional Arrays. *Tutorials Point [online]. India: Tutorials Point, c2022 [cit. 2022-06-26]. Dostupné z: https://www.tutorialspoint.com/typescript/typescript_multi_dimensional_arrays.htm*

[18] Complex Types. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/learn/learn-typescript/modules/learn-typescript-complex-types/cheatsheet*

[19] Tuple Types. *TypeScript [online]. Redmond (Washington): Microsoft, c2012-2022 [cit. 2022-06-15]. Dostupné z: https://www.typescriptlang.org/docs/handbook/2/objects.html#tuple-types*

[20] TypeScript 3.0: Exploring Tuples and the Unknown Type. *Auth0 [online]. Bellevue (Washington): Arias, 2018, 2018-8-21 [cit. 2022-06-26]. Dostupné z: https://auth0.com/blog/typescript-3-exploring-tuples-the-unknown-type/#TypeScript-TupleWare*

[21] Tuples. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-arrays/exercises/tuples*

[22] Array Type Inference. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-arrays/exercises/array-type-inference*

[23] Rest parameters. *Mozilla [online]. Mountain View (California): Mozilla Corporation, c1998–2022, 2022-18-6 [cit. 2022-06-26]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters*

[24] TypeScript - Rest Parameters. *TutorialsTeacher [online]. TutorialsTeacher, c2022 [cit. 2022-06-26]. Dostupné z: https://www.tutorialsteacher.com/typescript/rest-parameters*

[25] Rest Parameters in TypeScript. *GeeksforGeeks [online]. India: GeeksforGeeks, 2022, 2022-2-14 [cit. 2022-06-26]. Dostupné z: https://www.geeksforgeeks.org/rest-parameters-in-typescript*

[26] Rest Parameters. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-arrays/exercises/rest-parameters*

[27] Spread syntax (...). *Mozilla [online]. Mountain View (California): Mozilla Corporation, c1998–2022, 2022-5-19 [cit. 2022-06-26]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax*

[28] Introduction. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-custom-types/exercises/introduction*

[29] How To Use Enums in TypeScript. *DigitalOcean [online]. New York: Cardoso, c2022, 2021-6-14 [cit. 2022-06-26]. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-use-enums-in-typescript*

[30] TypeScript Data Type - Enum. *TutorialsTeacher [online]. TutorialsTeacher, c2022 [cit. 2022-06-26]. Dostupné z: https://www.tutorialsteacher.com/typescript/typescript-enum*

[31] Enums. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-custom-types/exercises/enums*

[32] Object Types. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-custom-types/exercises/object-types*

[33] Type Aliases. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-custom-types/exercises/type-aliases*

[34] How To Use Generics in TypeScript. *DigitalOcean [online]. New York: Cardoso, c2022, 2021-11-6 [cit. 2022-06-26]. Dostupné z:*

https://www.digitalocean.com/community/tutorials/how-to-use-generics-in-typescript

[35] Function Types. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-custom-types/exercises/function-types*

[36] Functions: Generic Functions. *TypeScript [online]. Redmond (Washington): Microsoft, c2012-2022 [cit. 2022-06-15]. Dostupné z: https://www.typescriptlang.org/docs/handbook/2/functions.html#generic-functions*

[37] Generic Functions. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/typescript-custom-types/exercises/function-generics*

[38] Union Types: Introduction. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/union-types/exercises/introduction*

[39] Unions and Intersection Types. *TypeScript [online]. Redmond (Washington): Microsoft, c2012-2022 [cit. 2022-06-15]. Dostupné z: https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html*

[40] Narrowing. *Open Source: Microsoft [online]. Redmond (Washington): Microsoft, c2022 [cit. 2022-06-26]. Dostupné z: https://microsoft.github.io/TypeScript-New-Handbook/chapters/narrowing*

[41] Union Types *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/learn/learn-typescript/modules/learn-typescript-union-types/cheatsheet*

[42] Type an Array with a Union type in TypeScript. *Bobbyhadz blog [online]. Hadzhiev, 2022, 2022-3-2 [cit. 2022-06-26]. Dostupné z: https://bobbyhadz.com/blog/typescript-array-with-union-type*

[43] Common Key Value Pairs. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/union-types/exercises/common-key-value-pairs*

[44] Unions with Literal Types. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/learn-typescript/lessons/union-types/exercises/unions-with-literal-types*

[45] TypeScript Vs. JavaScript. *JavaTpoint [online]. Noida (India): JavaTpoint, c2011-2021 [cit. 2022-06-28]. Dostupné z: https://www.javatpoint.com/javascript-vs-typescript*

[46] When to Choose: Difference Between JavaScript and TypeScript. Radixweb [online]. *Ahmedabad (India): Raval, 2022, 2022-6-9 [cit. 2022-06-28]. Dostupné z: https://radixweb.com/blog/typescript-vs-javascript*

[47] Difference between TypeScript and JavaScript. *GeeksforGeeks [online]. India: GeeksforGeeks, 2022, 2022-5-19 [cit. 2022-06-28]. Dostupné z: https://www.geeksforgeeks.org/difference-between-typescript-and-javascript*

[48] Summary: What is the Difference Between TypeScript and JavaScript?. *Guru99 [online]. Wilmington (Delaware): Hartman, 2022, 2022-5-14 [cit. 2022-06-28]. Dostupné z: https://www.guru99.com/typescript-vs-javascript.html*

[49] Why React?. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-intro/exercises/why-react*

[50] Introducing JSX. *React: A JavaScript library for building user interfaces [online]. Menlo Park (California): Meta Platforms, c2022 [cit. 2022-06-26]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html*

[51] Hello World. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-intro/exercises/hello-world*

[52] Rendering Elements. *React: A JavaScript library for building user interfaces [online]. Menlo Park (California): Meta Platforms, c2022 [cit. 2022-06-26]. Dostupné z: https://reactjs.org/docs/rendering-elements.html*

[53] Attributes In JSX. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-intro/exercises/attributes-in-jsx*

[54] ReactJS Components: Type, Nesting, and Lifecycle. *Simplilearn [online]. San Francisco (California): Simplilearn Solutions, c2009-2022, 2021-11-11 [cit. 2022-06-26]. Dostupné z: https://www.simplilearn.com/tutorials/reactjs-tutorial/reactjs-components*

[55] Nested JSX. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-intro/exercises/nested-jsx*

[56] Rendering JSX. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-intro/exercises/render-jsx*

[57] JSX. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/learn/react-101/modules/react-101-jsx-u/cheatsheet*

[58] Class vs className. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-advanced/exercises/jsx-classname-class*

[59] Self-Closing Tags. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-advanced/exercises/self-closing-tags*

[60] Self-Closing Tag. *React: A JavaScript library for building user interfaces [online]. Menlo Park (California): Facebook, c2013–2015 [cit. 2022-06-26]. Dostupné z: https://react-cn.github.io/react/tips/self-closing-tag.html*

[61] Variables in JSX. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-advanced/exercises/jsx-variables*

[62] If-Else in JSX. *React: A JavaScript library for building user interfaces [online]. Menlo Park (California): Facebook, c2013–2015 [cit. 2022-06-26]. Dostupné z: https://react-cn.github.io/react/tips/if-else-in-JSX.html*

[63] ReactJS | Keys. *GeeksforGeeks [online]. India: GeeksforGeeks, 2022, 2022-5-11 [cit. 2022-06-26]. Dostupné z: https://www.geeksforgeeks.org/reactjs-keys*

[64] Keys. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-advanced/exercises/jsx-keys*

[65] React.createElement. *Codecademy [online]. New York: Codecademy, c2022 [cit. 2022-06-26]. Dostupné z: https://www.codecademy.com/courses/react-101/lessons/react-jsx-advanced/exercises/react-createelement*

[66] Adding TypeScript. *Create React App: Set up a modern web app by running one command [online]. Menlo Park (California): Facebook, c2022 [cit. 2022-06-26]. Dostupné z: https://create-react-app.dev/docs/adding-typescript*

[67] Using the State Hook. *React [online]. Menlo Park (California): Meta Platforms, c2022 [cit. 2022-06-26]. Dostupné z: https://reactjs.org/docs/hooks-state.html*

[68] Typed useState with TypeScript. *Carl Rippon [online]. Rippon, 2019, 2019-1-8 [cit. 2022-06-26]. Dostupné z: https://www.carlrippon.com/typed-usestate-with-typescript*

[69] Type the onChange event of an element in React (TypeScript). *Bobbyhadz blog [online]. Hadzhiev, 2022, 2022-4-17 [cit. 2022-06-26]. Dostupné z: https://bobbyhadz.com/blog/typescript-react-onchange-event-type*

[70] Export Default in React. *DelftStack [online]. Delft (Netherlands): DelftStack, 2022, 2022-4-1 [cit. 2022-06-26]. Dostupné z: https://www.delftstack.com/howto/react/export-default-in-react*

[71] Why You Should Probably Think Twice About Using React.FC. *Atomic Object [online]. Grand Rapids (Michigan): Bohn, 2022, 2022-1-4 [cit. 2022-06-26]. Dostupné z: https://spin.atomicobject.com/2022/01/04/think-twice-react-fc*

[72] Vercel: Deploying using the Vercel CLI. *LogRocket [online]. Boston (Massachusetts): Singh, 2022, 2022-2-2 [cit. 2022-06-28]. Dostupné z: https://blog.logrocket.com/8-ways-deploy-react-app-free*

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

A Příloha

CD/DVD obsahuje složku pro každou aplikaci, ve které se nachází všechny TS a TSX soubory, které obsahují mnou psaný kód a jejich JS a JSX varianty vzniklé kompilací. Příloha zároveň obsahuje text práce ve formátu PDF.

B Příloha

Web: `http://tomrot.narrasa.eu`