SLEZSKÁ UNIVERZITA V OPAVĚ
Filozoficko-přírodovědecká fakulta v Opavě

BAKALÁŘSKÁ PRÁCE

Opava 2024 Lukáš Sukeník

SLEZSKÁ UNIVERZITA V OPAVĚ
Filozoficko-přírodovědecká fakulta v Opavě

Lukáš Sukeník

Studijní program: Moderní informatika
Specializace: Informační a komunikační technologie

Porovnání SPA frontend frameworků

Comparison of SPA frontend frameworks

Bakalářská práce

Opava 2024

Vedoucí bakalářské práce:
doc. RNDr. Lucie Ciencialová, Ph.D.

SLEZSKÁ UNIVERZITA V OPAVĚ
Filozoficko-přírodovědecká fakulta v Opavě

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Akademický rok: 2023/2024

Zadávající ústav: Ústav informatiky

Student: Lukáš Sukeník

UČO: 60639

Program: Moderní informatika

Specializace: Informační a komunikační technologie

Téma práce: Porovnání SPA frontend frameworků

Téma práce anglicky: Comparison of SPA frontend frameworks

Zadání: Autor se seznámí s problematikou tvorby webových aplikací s využitím JavaScriptových frameworků. V teoretické části bude analyzovat možnosti vývoje frontendu webových aplikací postavených na JavaScriptu, vybere tři frameworky, kterým se bude dále věnovat. V praktické části autor navrhne aplikaci a její funkce. S využitím zvolených frameworků vytvoří tři demonstrační aplikace, uvede postupy jejich vývoje a závěrečné porovnání.

Literatura:

BANKS, Alex a PORCELLO, Eve. Learning React. Online. 2nd Edition. O'Reilly Media, 2020. ISBN 9781492051725. Dostupné z: https://www.oreilly.com/library/view/learning-react-2nd/9781492051718/. [cit. 2023-10-05].

MACRAE, Callum. Vue.js: Up and Running. Online. O'Reilly Media, 2018. ISBN 9781491997246. Dostupné z: https://www.oreilly.com/library/view/vuejs-up-and/9781491997239/. [cit. 2023-10-05].

SEGALA, Alessandro. Svelte 3 Up and Running. Online. Packt Publishing, 2020. ISBN 9781839213625. Dostupné z: https://www.oreilly.com/library/view/svelte-3-up/9781839213625/. [cit. 2023-10-05].

FLANAGAN, David. JavaScript: The Definitive Guide. Online. 6th Edition. O'Reilly Media, 2011. ISBN 9780596805524. Dostupné z: https://www.oreilly.com/library/view/javascript-the-definitive/9781449393854/. [cit. 2023-10-05].

MEYER, Eric a WEYL, Estelle. CSS: The Definitive Guide. Online. 4th Edition. O'Reilly Media, 2017. ISBN 9781449393199. Dostupné z: https://www.oreilly.com/library/view/css-the-definitive/9781449325053/. [cit. 2023-10-05].

Vedoucí práce: doc. RNDr. Lucie Ciencialová, Ph.D.

Datum zadání práce: 5. 10. 2023

Souhlasím se zadáním (podpis, datum):

.....

doc. RNDr. Luděk Cienciala, Ph.D.
vedoucí ústavu

Abstrakt

Text abstraktu v češtině. Rozsah by měl být 50 až 100 slov. Abstrakt není cíl práce, zde stručně popište, co čtenář má na následujících stránkách očekávat. Typické formulace: „V práci se zabýváme...“, „Tato bakalářská práce pojednává o...“, „součástí je“, „je provedena analýza“, „praktickou částí práce je aplikace xxx“ ... Prostě napište stručný souhrn či charakteristiku obsahu práce.

Klíčová slova

Napište 5–8 klíčových slov v českém jazyce (v jednotném čísle, první pád atd.), měla by vystihovat téma práce. Slova oddělujte čárkou. Snažte se vystihnout nejdůležitější pojmy vystihující práci.

Abstract

Anglická verze abstraktu by měla odpovídat české verzi, třebaže nemusí být úplně doslova. Když nutně potřebujete automatický překlad, použijte raději
https://www.deepl.com/cs/translator,
je lepší než Google Translator. Není nutno překládat doslova.

Keywords

Anglická obdoba českého seznamu klíčových slov.

Čestné prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškerou literaturu a další zdroje, z nichž jsem při zpracování čerpal, v práci řádně cituji a jsou uvedeny v seznamu použité literatury.

V Opavě dne 14. března 2024

.....

Lukáš Sukeník

Poděkování

Rád bych poděkoval za odborné vedení, rady a cenné poznatky k danému tématu vedoucímu práce doc. RNDr. Lucii Ciencialové, Ph.D. Také bych rád poděkoval mé rodině a přátelům za podporu a pomoc během mého studia.

Obsah

Úvod

V Úvodu především rozvedeme cíl práce (ten najdeme v zadání tématu práce), můžeme poněkud méně formálně o tématu povykládat (ale nepřehánějte to, žádných 10 stran úvodu prosím). Můžeme psát o své motivaci, tedy proč jsme si téma zvolili, proč je považujeme za zajímavé či důležité. Můžeme také jemně uvést čtenáře do problematiky.

Je zvykem zde psát o tom, z jakých částí se práce skládá. Není nutné jít po kapitolách, můžeme například napsat, že práce má teoretickou a praktickou část, přičemž v teoretické části je čtenář nejdřív seznámen s problematikou xxx, jsou zde vysvětleny základní pojmy a v několika kapitolách nejběžnější metody používané pro yyy. V praktické části je popsána aplikace zzz sloužící k qqqq, najdeme zde manuál k jejímu používání s postupem instalace a zprovoznění a také popis její vnitřní struktury a okomentované ukázky kódu. NEBO: Praktickou částí je srovnání metod sloužících k rrrrr, přičemž po analýze metod daného typu se zdůvodněním výběru a metodikou pro jejich porovnání jsou v jednotlivých kapitolách popsány vybrané metody a v poslední kapitole najdeme jejich vzájemné srovnání. NEBO: V první kapitole rozebíráme typické požadavky na sociální sítě/informační systémy/webové aplikace/... pro účel stanovený v zadání, v následující kapitole nastiňujeme momentální stav co se týče existujících produktů pro daný účel a hodnotíme, do jaké míry splňují stanovené požadavky. Další kapitoly obsahují návrh vlastní xxxxx s popisem jak samotné xxxx, jejího zprovoznění, rozhraní, atd., tak i postup vytvoření (byl použit programovací jazyk zzzz), a opět zhodnocení míry splnění stanovených požadavků.

Tato část úvodu má čtenáře připravit na vlastní obsah práce, tak si dejte záležet, ať čtenáře neotrávíte předem :-)

V Úvodu dále můžeme připsat informaci o tom, že obrázky bez uvedeného zdroje byly vytvořeny v nástroji xxxxx, případně také že u čtenáře předpokládáme alespoň základní znalosti v oblasti ... (programování, počítačových sítí, informačních systémů, sociálních sítí, tvorby webových stránek, atd. podle tématu), abyste nemuseli vysvětlovat ty nejzákladnější pojmy.

1 Webové aplikace

Text první kapitoly.

- popsání FE/BE,
- framework, SPA,
- asi spíše nečíslované podčásti nebo rozdělit do více kapitol...

2 Analýza frameworků

V této části práce se zabýváme analýzou frameworků Angular, React, Svelte a Vue. Nejdříve analyzujeme každý framework samostatně. Analyzujeme klíčové koncepty frameworků, především stavební bloky, správu stavů, předávání vlastností. Dále rozebíráme životní cykly, state management, routování a ekosystém jednolivých technologií. Kapitolu zakončuje porovnání analyzovaných frameworků.

Výběr porovnávaných technologiií byl proveden na základě analýzy výsledků z Developer Survey 2023 od Stack Overflow. Rozhodující byla sekce *Web frameworks and technologies*, ve které byly hodnoceny webové technologie dle jejich žádanosti na trhu a obdivem mezi vývojářskou komunitou. Tato strategie zajistila, že vybrané frameworky odpovídají současným trendům a potřebám webového vývoje, ale také vycházejí z uznání a podpory komunity.[37, 38]

2.1 Angular

Angular byl původně interní JavaScriptový framework společnosti Google. Dle oficiální dokumentace [4] je Angular kompletní platforma, určená k vývoji webových aplikací. Framework je postaven na principu komponent, jež tvoří základní stavební jednotku aplikace. Součástí frameworku jsou knihovny, které jsou velmi dobře integrovatelné a usnadňují práci s různými částmi aplikace. Dále také nástroje, které vývojářům usnadňují vývoj, testování či aktualizaci kódu.[4, 7]

Obrázek 1: Angular logo [3]

První verze Angularu byla vydána v roce 2012 pod názvem AngularJS. V roce 2016, po kolaboraci se společností Microsoft, Google vydal novou verzi, o které mluvíme jako o Angular 2. Verze 2 byla kompletně přepsána, framework byl přepsán z JavaScriptu do jazyku TypeScript. Součástí frameworku je i knihovna RxJS pro práci s asynchronními událostmi. Knihovnu reaktivních rozšíření, RxJS, mohou vyu-

žívat také programátoři při vývoji Angular aplikací. Angular se samozřejmě neustále vyvíjí a v současné době můžeme pracovat s verzí 17.[3, 7]

Díky robusnosti a velikosti frameworku je Angular vhodnější pro větší aplikace, které vyžadují mnoho funkcí a komplexních struktur. V poslední době framework spíše ztrácí na popularitě, stále jej však používá mnoho společností včetně Google.[7]

2.1.1 Komponenty

Hlavní bloky kódu v Angularu tvoří komponenty. Komponenta je OOP třída definovaná v rámci TypeScript souboru ve formátu nazev-tridy.component.ts. Komponenta obsahuje nastavení v dekorátoru @Component, v němž definujeme selektor a ostatní části (soubory) komponenty. Dále v metadatech může být nastavení použitého API, šablona nebo kaskádové styly. Mezi další části komponenty patří soubor s šablonou, kaskádovými styly či testovací scénáře. Tyto soubory jsou nepovinné, obvykle jsou však žádoucí a vedou k lepší organizaci kódu.

Původní komponenty byly tvořeny pomocí modulů (NgModules), které obsahovaly deklarace komponent, direktiv a služeb. Od verze 14 je možné využít standalone API, které umožňuje vytvářet komponenty bez nutnosti tvoření a registrace modulů v jiných souborech, a díky tomu je kód kratší. V dekorátoru @Component pak je třeba importovat všechny potřebné závislosti – direktivy, služby či jiné komponenty.[3, 7]

V šabloně vykreslíme hodnoty pomocí dvojitých závorek, které obsahují název proměnné. K podmíněnému zobrazení komponent/elementu slouží bloky @if, @else if, @else. Pro iteraci přes pole hodnot slouží blok @for.[3]

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-component',
  standalone: true,
  template: `
    <div>{{ content }}</div>
  `,
  styles: ['div { background-color: red; }']
})
export class MyComponent {
  content = 'nějaký-obsah';
}
```

2.1.2 Správa stavů

K vytvoření stavů v Angularu využijeme vlastnosti třídy (class fields). Vlastnosti mohou být veřejné (public), protected (chráněné) nebo soukromé (private). V případě, že viditelnost nespecifikujeme, je vlastnost veřejná. Soukromé vlastnosti jsou viditelné pouze v rámci třídy, chráněné v rámci třídy a šablony. Veřejné vlastnosti jsou viditelné odkudkoli.[3, 7]

Pro aktualizaci stavů využijeme přiřazení nové hodnoty do vlastnosti, k níž přistoupíme pomocí klíčového slova this.[3]

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-component',
  standalone: true,
  template: `
    <button (click)="increment()">
      Klikli jste na tlačítko {{ count }}x.
    </button>
  `,
})
export class MyComponent {
  protected count = 0;

  protected increment(): void {
    this.count++;
  }
}
```

Počínaje verzí 16 můžeme používat také nestabilní verzi signálů (Angular Signals), přičemž se jedná o systém, který mnohem efektivněji sleduje využití stavů v rámci aplikace. Signály pak umožňují efektivnější a optimalizavanější aktualizace DOM.

```
import { Component, signal } from '@angular/core';

@Component({
  selector: 'my-component',
  standalone: true,
  template: `
    <button (click)="increment()">
      Klikli jste na tlačítko {{ count() }}x.
    </button>
  `,
})
export class MyComponent {
  protected count = signal(0);

  protected increment(): void {
```

```
    this.count.update((value) => value + 1);
  }
}
```

Od verze 17 jsou signály stabilní součástí Angularu, na druhou stranu předávání a modely signálů zatím stabilní nejsou. Proto v této práci budeme využívat klasické hodnoty stavů.[3]

2.1.3 Předávání vlastností

Předávat vlastnosti či jiné hodnoty je možné pomocí vstupního dekorátoru @Input a výstupního dekorátoru @Output. V rámci šablony danou hodnotu předáme do vnořené komponenty skrze název vstupu v hranatých závorkách. Ve vnořené komponentě využijeme dekorátor @Input, sloužící k získání hodnot z rodičovské komponenty. K předaným hodnotám není možné přistupovat v rámci konstruktoru.

K předání hodnoty z vnořené komponenty nejprve předáme vnořené komponentě název výstupu v kulatých závorkách a obslužnou metodu, která se vykoná po předání vlastnosti. Dále definujeme @Output ve vnořené komponentě, na němž zavoláme metodu emit(), která přes argument metody umožňuje předání (vypublikování) hodnot z vnořené do rodičovské komponenty.[3, 7]

```
import { CommonModule } from '@angular/common';
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'parent-component',
  standalone: true,
  imports: [ChildComponent],
  template: `
    <child-component
      [color]="someProps.color"
      (colorClicked)="handleColorClicked($event)"
    />
    <p>{{ colorClickedText }}</p>
  `,
})
export class ParentComponent {
  protected someProps = { color: 'cervena' };
  protected colorClickedText = 'Na název barvy jste zatím neklikli.';

  protected handleColorClicked(clickCount: number): void {
    this.colorClickedText = `Na název barvy
      v ChildComponent jste klikli: ${clickCount}x`;
  }
}
```

```
@Component({
  selector: 'child-component',
  standalone: true,
  imports: [CommonModule],
  template: `<p [ngClass]="color" (click)="handleClick()">{{ color }}</p>`,
})
export class ChildComponent {
  private clickCount = 0;

  @Input() color = 'Žádná barva nebyla specifikována.';
  @Output() colorClicked = new EventEmitter<number>();

  protected handleClick(): void {
    this.clickCount++;
    this.colorClicked.emit(this.clickCount);
  }
}
```

2.1.4 Služby, direktivy, roury

Angular disponuje pestrou škálou možností, jak sdílet bloky kódu či logiku mezi různými částmi aplikace. Služby, direktivy, nebo také roury tvoříme pomocí JavaScriptových tříd. Služby (services) umožňují znovupoužití určité části kódu např. v rámci více komponent. Služby obvykle používáme ke komunikaci s HTTP endpointy, sdílíme v nich stavy více komponent, a také je využíváme k transformacím.[3, 7]

```
import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class DataService {
  private data = 'Initial data';

  getCurrentData(): string {
    return this.data;
  }

  setNewData(newData: string): void {
    this.data = newData;
  }
}
```

Direktivy (directives), ať už zabudované či vlastní, slouží k obohacení HTML elementů o různé funkce (dynamické atributy, CSS třídy) nebo manipulaci s DOM elementy. Roury (pipes) umožňují trasformaci hodnot v šabloně.[3, 4]

```
import { Component, Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'czechDate', standalone: true })
export class CzechDateFormatPipe implements PipeTransform {
  transform(value: Date): string {
```

```
    return value.toLocaleDateString('cs-CZ');
  }
}

@Component({
  selector: 'my-component',
  standalone: true,
  imports: [CzechDateFormatPipe],
  template: `<p>{{ today | czechDate}}</p>`,
})
export class MyComponent {
  protected today = ;
}
```

2.1.5 Životní cyklus

Životním cyklem komponenty rozumíme sekvenci kroků, které se vykonají mezi vytvořením a zničením komponenty. Životní cyklus může být rozdělen do čtyř částí: vytvoření, aktualizace, vykreslení a zničení.

První metoda, která se spouští při vytvoření komponenty, je konstruktor. Dále můžeme využít metody ngOnInit, ngOnChanges, ngDoCheck, ngAfterViewInit, ngAfterContentInit, ngAfterViewChecked, ngAfterContentChecked, jež se spouští v různých fázích aktualizace (detekce). Nedávno byly přidány také metody po vykreslení komponenty – afterNextRender a afterRender. V neposlední řadě je možné využít metodu ngOnDestroy, která se spouští při zničení komponenty.[3, 7]

2.1.6 State management

K základní práci se stavy slouží vlastnosti třídy, které inicializujeme JavaScriptovou hodnotou. Do budoucna můžeme počítat s lepší podporou signálů, které aktualizují DOM efektivněji a rychleji. Pokročilejší způsob sdílení stavů v rámci aplikace spočívá ve využití služeb, v nichž uložíme stavy a následně je sdílíme mezi komponentami.[3]

Pro reaktivní či asynchronní operace, nebo obecně složitější funkcionality, využijeme balíček RxJS, který je již součástí Angularu. RxJS poskytuje datový typ Observable, který reprezentuje data, jež se mohou měnit v čase.[4, 35]

V případě, že potřebujeme sdílet stavy globálně (mezi různými částmi aplikace), můžeme využít knihovnu NgRx, která je inspirována knihovnou Redux.[12, 27]

2.1.7 Routování

Angular poskytuje vestavěný systém routování – konkrétně balíček @angular/router, který na straně klienta umožňuje přepínat mezi různými částmi aplikace. Klasické webové stránky při změně URL pokaždé žádají o nové dokumenty. Routování na straně klienta může provést aktualizaci stránky bez dalších duplikátních dotazů. Při vyžádání dané cesty pak vykreslíme požadovaný obsah a požádáme pouze o data potřebné pro vykreslení. Získáme tak rychlejší uživatelskou zkušenost, jelikož prohlížeč nevyžaduje nové dokumenty a nemusí vyhodnocovat kaskádové styly či JavaScript.

Abychom zaregistrovali cesty aplikace pomocí knihovny @angular/router, poskytneme samotný router pomocí funkce provideRouter do aplikačního nastavení. Routeru následně předáme pole cest aplikace. Jednotlivé obsahy stránek, které se mají zobrazit, vykreslíme pomocí elementu router-outlet.[3, 7]

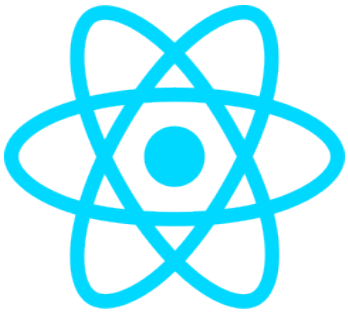
2.1.8 Ekosystém

Angular je komplexní framework, v němž jsou obsaženy základní balíčky všeho druhu potřebné pro vývoj webových aplikací. Framework se stále vyvíjí a jeho ekosystém se neustále rozrůstá, a to i přes velikost projektu. Angular nepostrádá ani v množství balíčků třetích stran, které vývojáři využívají pro usnadnění práce s různými částmi aplikace. Nechybí ani balíček @angular/cli, jenž usnadňuje vývoj aplikace, testování, aktualizaci kódu a jeho nasazení.[3, 7]

2.2 React

Pod pojmem React rozumíme open-source JavaScript framework, který vyvinula a dále vyvíjí společnost Meta (dříve Facebook). Podle [8] jde spíše o knihovnu funkcí, než-li o komplexní nástroj pro tvorbu webových aplikací (framework). Tato technologie se používá pro vývoj interaktivních uživatelských rozhraní a webových aplikací.[18]

První kořeny Reactu sahají až do roku 2010, kdy tehdejší společnost Facebook přidala novou technologii XHP do PHP. Jde o možnost znovu použít určitý blok kódu, stejného principu posléze využívá i React. Následně Jordan Walke vytvořil



Obrázek 2: React logo [31]

FaxJS, jenž byl prvním prototypem Reactu. O rok později byl přejmenován na React a začal jej využívat Facebook. V roce 2013 byl na konferenci JS ConfUS představen široké veřejnosti a stal se open-source.

Od roku 2014 vývojáři představují nespočet vylepšení samotné knihovny, stejně jako spoustu rozšíření pro zlepšení vývojových procesů. Kolem roku 2015 postupně React nabývá na popularitě i celkové stabilitě. Následně je představen také React Native, což je framework pro vývoj nativních aplikací. V dnešní době je React využíván společnostmi každého rozsahu po celém světě. Z těch největších jde například o Metu, Uber, Twitter a Airbnb.[8, 17]

2.2.1 Komponenty

Hlavním stavebním kamenem Reactu jsou komponenty, jež představují nezávislé, vnořitelné a opakovaně použitelné bloky kódu. Komponentu v Reactu tvoří JavaScript funkce a HTML šablona. Validně seskládané komponenty poté tvoří webovou aplikaci. V Reactu se můžeme setkat s funkčními a třídními komponentami. Vytváření třídních komponent oficiální dokumentace nedoporučuje.

Pro komunikaci mezi komponentami se používá předávání vlastností (props), přes které je možné předávat hodnoty jakýchkoli datových typů. Výstup komponent tvoří elementy ve formě JSX. Tyto elementy obsahují informace o vzhledu a funkcionalitě dané komponenty.[8, 31]

```
import React from 'react';

function ParentComponent() {
  const someProps = {color: 'cervena'};

  return (
```

```
    <div>
      <ChildComponent color={'cervena'} />
      <ChildComponent color={someProps.color} />
      <ChildComponent {...someProps} />
    </div>
  );
}

function ChildComponent({color}) {
  return (
    <div className={color}></div>
  );
}
```

2.2.2 JSX

Název JSX kombinuje zkratku jazyka JavaScript – JS a počáteční písmeno ze zkratky XML. Konkrétně jde o syntaktické rozšíření, které vývojářům umožňuje tvořit React elementy pomocí hypertextového značkovacího jazyku přímo v JavaScriptu. V rámci JSX pak je možné dynamicky vykreslovat obsah na základě logiky definované pomocí JavaScriptových hodnot. Při kompilaci se JSX překládá do JavaScriptu pomocí nástroje Babel.[8, 31]

```
import React from 'react';
import ChildComponent from './ChildComponent';

function MyComponent() {
  const loaded = true;

  return (
    <div>
      {loaded ?
        <ChildComponent color={'cervena'} width={100} heigth={100} />
        : 'Načítání ...'}
    </div>
  );
}
```

2.2.3 Správa stavů

Stav lze definovat jako lokální vnitřní vlastnost či proměnnou dané komponenty, jež představuje základní mechanismus pro uchovávání a aktualizaci dat. Pro aktualizaci komponenty je tedy nutné stav změnit. React pak na tuto skutečnost zareaguje a vyvolá tzv. re-render neboli překreslení komponenty s novými daty.

Za účelem ukládání stavu se využívá hook (funkce) useState. Ten poskytuje stavovou proměnnou, přes kterou se dostaneme k aktuálnímu stavu. Dále useState

poskytuje state setter funkci, díky které můžeme stav aktualizovat. Jediný argument useState definuje počátační hodnotu daného stavu.[24, 31]

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

function App() {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      Klikli jste na tlačítko {count}x.
    </button>
  );
}
```

2.2.4 Hooky

Specifickou funkcionalitou pro React jsou tzv. hooky, které byly do Reactu přidány až ve verzi 16.8.0.[32] Hook je definován jako funkce, která obohacuje komponenty pomocí předdefinovaných funkcionalit. Jedním z nejpoužívanejších hooků je useState. Vývojáři mohou používat již zabudované hooky, nebo si vytvářet své vlastní s pomocí předdefinovaných hooků. Mezi zabudované hooky patří např. useEffect, useMemo, useCallback, useRef, useContext.[31]

```
import { useEffect } from 'react';

useEffect(() => {
  // obvykle kód určený pro nastavení (setup)

  return () => {
    // kód pro úklid prostředků
  };
}, [
  // seznam závislostí, na jejichž změnu má efekt reagovat
]);
```

2.2.5 Životní cyklus

Životní cyklus komponenty je sekvence událostí, jež nastanou mezi vytvořením a zničením komponenty. Ve třídních komponentách existovaly speciální metody, tzv. lifecycle metody, starající se o provedení určité části kódu při daném okamžiku v životě komponenty. Nyní React disponuje pár hooky, které umožňují provádět side-effects podobně jako lifecycle metody.

O momentu, kdy je komponenta přidána na obrazovku, mluvíme jako o namontování (mount) komponenty. Při změně stavu či obdržení nových parame-

trů hovoříme o aktualizaci (update) komponenty. A v neposlední řadě okamžik, kdy je komponenta odstraněna z obrazovky, nazýváme odmontování (unmount) komponenty.[25, 31]

2.2.6 State management

Základní práce se stavy spočívá v lokálních stavech komponent a následným předáváním stavu do potomků či rodičů. V případě, že potřebujeme sdílet stav mezi komponentami, měli bychom zvážit odlišné řešení. React sám o sobě disponuje pouze základním řešením, kterému říká Context API. Context umožňuje sdílet data celému podstromu dané komponenty. To se může hodit například při vytváření barevných módů aplikace, sdílení informace o přihlášeném uživateli, anebo routování.[31]

Správa stavů v komplexních aplikacích se stává výzvou. Problémy začínají při potřebě sdílení identických dat mezi větším množstvím konzumentů. Existuje však mnoho knihoven třetích stran, které vývojáři využívají pro usnadnění manipulace se stavy. Společné cíle state management knihoven spočívají v ukládání a získávání globálního stavu, jednodušší správě stavů a rozšiřitelnosti aplikace. Mezi tyto knihovny patří kupříkladu Redux, MobX, Recoil nebo Jotai.[2, 14]

2.2.7 Routování

React nemá žádný nativní standard pro routování. Podle [8] je React Router jedním z nejvíce populárních řešení pro React. Knihovna React Router umožňuje nastavení jednotlivých cest aplikace. Zajišťuje tedy routování na straně klienta.

Instanci routeru vytvoříme například pomocí funkce createBrowserRouter, která přijímá pole definovaných cest aplikace. Další možností je vytvoření cest pomocí funkce createRoutesFromElements. Router následně předáme do komponenty RouterProvider. K vykreslení požadované komponenty, která je spojena s danou cestou, slouží komponenta Outlet.[8, 34]

2.2.8 Ekosystém

Tato knihovna sama o sobě není úplně komplexním nástrojem. I přesto se stále vyvíjí a její ekosystém se neustále rozrůstá. Na druhou stranu pak existuje mnoho

různých nástrojů třetích stran, funkcí, API, které mohou vývojáři při vývoji použít. Knihovny jsou velmi diverzifikované. Začíná to knihovnami zameřenými na stylování, či předpřipravenými komponentami pro uživatelské rozhraní. Samozřejmě najdeme i balíčky pro tvorbu tabulek, formulářů, grafů nebo grafikcých animací. Dále můžeme využít mnohých knihoven, které řeší správu stavů, routování, dotazování na API. Nechybí ani dokumentační knihovny, vývojářské rozšíření pro prohlížeče, striktní typování, překlady, testovací balíčky. V neposlední řadě pro React existují nadstavby ve formě frameworků, které pak poskytují lepší základy pro produkční aplikace.[5, 16, 31]

2.3 Svelte

Svelte je relativně novým open-source JavaScript frameworkem, za jejímž stvořením stojí vývojář Richard Harris. Framework kompiluje komponenty přímo do čistého nativního a vysoce optimalizovaného JavaScriptu bez potřeby runtime. To vše ještě před tím, než uživatel navštíví webovou aplikaci v prohlížeči. Tato metoda poskytuje výhodu hlavně co se týče rychlosti oproti klasickým deklarativním frameworkům jako jsou např. React, Vue nebo Angular. Stejně jako tyto frameworky je Svelte určen k vývoji rychlého a kompaktního uživatelského rozhraní pro webové aplikace.

Obrázek 3: Svelte logo [39]

První verze byla představena ke konci roku 2016. Verze 3, jež byla vydána v dubnu 2019, přinesla vylepšení týkající se zjednodušení tvorby komponent. Mimo jiné tato verze hlavně představila vylepšení ve smyslu reaktivity. Po této verzi framework nabral na popularitě díky jeho jednoduchosti. Verze 4 pak v roce 2023

představila pouze minimální změny, jež spočívají v údržbě a přípravách pro verzi nastávající.

Přestože Svelte nedisponuje rozsáhlým ekosystémem jako jiné JavaScriptové frameworky, získal si přízeň mnoha velkých společností. Mezi ně patří například firmy jako The New York Times, Avast, Rakuten a Razorpay.[15, 39, 43]

2.3.1 Komponenty

Podobně jako v jiných frameworcích, komponenty jsou základní stavební bloky Svelte. Komponentu tvoří HTML, CSS a JavaScript, kde vše patří do jednoho souboru s příponou .svelte. Všechny tři části komponenty jsou nepovinné. Logika komponenty musí být zapsána mezi párové script tagy. Následuje jedna nebo více značek pro definovaní šablony komponenty. V neposlední řadě kaskádové styly se zapisují mezi style tagy.

V rámci šablony Svelte umožňuje využívat logické bloky pro podmíněné vykreslování nebe také iterace přes pole hodnot (list). Zabudovaná je i podpora manipulace s asynchronním JavaScriptem - promises.[39]

```
<script>
  let content = 'nějaký-obsah';
</script>

<div>{content}</div>

<style>
  div {
    background-color: red;
  }
</style>
```

2.3.2 Reaktivita

Srdcem Svelte jsou reaktivní stavy komponenty, které jednoduše definujeme jako proměnné v JavaScriptu. Jejich hodnotu aktualizuje JavaScript funkce pomocí přidělování nových hodnot. Kupříkladu stav o datovém typu pole tudíž nelze aktualizovat pouze pomocí metody push či splice. Je nutné využít jiné intuitivní řešení pomocí přidělení nové hodnoty. O všechno ostatní se pak postará sám Svelte v pozadí. Svelte aktualizuje DOM při každé změně stavu komponenty.

Mezi specifické funkce Svelte patří reaktivní deklarace, které se starají o aktualizaci stavů na základě stavů jiných. Další zabudovanou funkcí jsou tzv. reactive statements, jež umožní definovat akce, které se mají vykonat reaktivně – jako reakce na nějaký výrok.[10, 39]

```
<script>
  let count = 0;

  function increment() {
    count++;
  }
</script>

<button on:click={increment}>
  Klikli jste na tlačítko {count}x.
</button>
```

2.3.3 Předávání vlastností

Pro komunikaci mezi komponentami slouží mechanismus předávání vlastností. V rodičovské komponentě je nutné komponentě říci, jakou hodnotu chceme předat a do jaké proměnné ji chceme uložit v child komponentě. Pak v child komponentě vytvoříme stejnojmennou vlastnost s klíčovým slovem export.

Pokud chceme předávat vlastnosti parent komponentě, je třeba vytvořit vlastnost již na parent komponentě. Následně ji předat child komponentě a v rodičovské komponentě přidat před předání vlastnosti do komponenty klíčové slovo bind.[39]

```
// Parent.svelte
<script>
  import ChildComponent from './Child.svelte';

  const someProps = {color: 'cervena'};
</script>

<ChildComponent color='cervena' />
<ChildComponent color={someProps.color} />
<ChildComponent {...someProps} />

// Child.svelte
<script>
  export let color;
</script>

<div class={color}></div>
```

2.3.4 Eventy

Svelte má velice jednoduché API pro práci s DOM eventy. Stačí použít direktivu on na HTML elementu, která vyžaduje název eventu a callback funkci.

```
<script>
  let count = 0;
</script>

<button on:click={() => count++}>
  Klikli jste na tlačítko {count}x.
</button>
```

Vývojáři také přišli s možností odesílání a přijímání eventů pro komponenty. V child komponentě je třeba mít nějaký DOM event handler, na který chceme reagovat v parent komponentě. Poté je nutné využít zabudovanou metodu createEventDispatcher, které předáme potřebné parametry – náš libovolný název pro event komponenty a hodnotu. V rodičovské komponentě pak reagujeme na event pomocí klíčového slova on a našeho libovolného názvu pro event. Naši hodnotu poté získáme v callback funkci.[10, 39]

```
// Parent.svelte
<script>
  import Child from './Child.svelte';

  function handleMessage(event) {
    alert(event.detail.text);
  }
</script>

<Child on:message={handleMessage} />

// Child.svelte
<script>
  import { createEventDispatcher } from 'svelte';

  const dispatch = createEventDispatcher();

  function sayHello() {
    dispatch('message', {
      text: 'Hello world!'
    });
  }
</script>

<button on:click={sayHello}>Klikněte pro "Hello world!"</button>
```

Zdroj zdrojového kódu: [39]

2.3.5 Životní cyklus

Komponenty ve Svelte disponují životním cyklem, který začíná v momentě vytvoření komponenty a končí jejím zničením. Funkce onMount tvoří callback, který je zavolán po přidání komponenty do DOMu. Pokud chceme vykonat určité akce při zničení komponenty, můžeme toho dosáhnout dvěma způsoby. Prvním způsobem je vracení callback funkce v rámci onMount funkce. Druhou možnost představuje využití funkce onDestroy, která v argumentu přijímá callback funkci.

Pro práci převážně s imperativními akcemi slouží zabudované funkce beforeUpdate a afterUpdate. V případě beforeUpdate funkce jde o callback, který se volá před aktualizací komponenty, tj. před prvním voláním onMount nebo po každé změně stavu. Oproti tomu, afterUpdate je callbackem, jenž Svelte vykoná po prvním zavolání onMount nebo po každé aktualizaci komponenty.[10, 39]

2.3.6 State management

Svelte poskytuje pestrou škálu API pro správu stavů aplikace v závislosti na rozsahu a složitosti ukládaných dat. Základním přístupem pro správu stavů je ukládání a manipulace se stavy v rámci stromu komponent. To zahranuje tvorbu reaktivních stavů a jejich distribuci ve stromě pomocí předávání vlastností, bindování či eventů.

Další možnost state managementu představuje využití Context API, které umožňuje jednorázové uložení jakékoli hodnoty. Nasledně je možné získat tuto hodnotu i v rámci neincidentních komponent. Ukládání a získávání contextu umožňují funkce setContext a getContext.

Pro sofistikovanější práci se stavy slouží tzv. stores. V podstatě se jedná o globání úložiště stavů, které umožňuje uchovávat a získávat data. Store je jednoduše objekt s metodou subscribe, která umožní konzumentu dostat aktualizovaná data. Jednodušší variantu pro získání aktuálních dat představuje použití znaku $ před názvem proměnné. Svelte nám poskytuje hned několik podob storu. To jsou jednak writable a readable stores, kde jediný rozdíl spočívá v možnosti aktualizace dat. Pro stavy, které jsou odvozeny z jiných stores, existuje tzv. derived store. V neposlední řadě nám Svelte povoluje vytvořit i vlastní store.

Již zabudované globální úložiště můžeme jednoduše vytvořit pomocí metod

writable, readable a derived. Writable požaduje jako argument počáteční hodnotu. Readable navíc jako druhý argument může přijímat funkci start, jež implementuje callbacky volající se při prvním a posledním subscribe.[10, 39, 42]

2.3.7 Routování

Svelte nemá přímou podporu routování v aplikacích. Oficiální dokumentace uvádí jako oficiální knihovnu pro routování SvelteKit. Ve skutečnosti se jedná o framework nad Svelte, který poskytuje i další možnosti rozšíření webové aplikace. Dokumentace však doporučuje i jiné knihovny pro routování na základě odlišných přístupů. Konkrétně knihovny page.js, svelte-routing, svelte-navigator, svelte-spa-router nebo routify.[39, 41]

Routování ve SvelteKit je implementováno pomocí file systému. Komponenta s názvem +page.svelte definuje stránku aplikace. Framework umožňuje pro opakující se uživatelská prostředí využít tzv. layouts. Jde o soubor, který aplikuje určité elementy (duplicitní kód) pro aktuální adresář komponent. Pro vykreslení obsahu na základě samotných komponent se využívá element slot. SvelteKit také umožnuje vytváření dynamických parametrů přímo v souborovém systému. Díky takovým cestám je možné tvořit např. individuální příspěvky na blogu. Pomocí +server.js můžeme definovat API routy (endpointy) aplikace. Chybové stránky vytváříme pomocí +error.svelte souborů.[39, 40]

2.3.8 Ekosystém

I přesto, že Svelte používá stále více vývojářů, framework nedisponuje příliš rozsáhlým ekosystémem. Hlavní rozšíření spočívá v použití rozšiřujícího frameworku SvelteKit a jazyka TypeScript. Vztah Svelte a SvelteKit můžeme definovat jako sourozenecký, kdy SvelteKit poskytuje adaptivní prostředí pro vývoj aplikace jakéhokoli rozsahu.

Dle [21] neexistuje mnoho specifických knihoven přímo pro Svelte. Na druhou stranu je možné využít rozsáhlého ekosystému JavaScriptu, jelikož Svelte poskytuje přímou kontrolu nad DOM. V porovnání se specifickými knihovnami tento přístup

však obvykle vyžaduje práci navíc. Problematické bývá využití knihoven používající API prohlížeče.[13, 21, 46]

2.4 Vue

Vue dostalo svůj název díky anglickému slovu view. Jedná se o deklarativní JavaScriptový open-source framework. Je určen efektivní tvorbě jak jednoduchých, tak i komplexních uživatelských rozhraní na webu. Framework je v současné době jedním z nejpopulárnějších frameworků pro tvorbu webových aplikací.[23, 47]

Obrázek 4: Vue logo [47]

Evan You, tvůrce Vue.js, se inspiroval určitými částmi frameworku AngularJS, který však měl velmi strmou křivku učení. Vue tedy mělo být lehké, přizpůsobivé a snadné k naučení. Bylo vytvořeno roku 2013, uvolněno do světa až o rok později. Od té doby byly vydány pouze 3 majoritní verze, avšak ty přinesly mnoho změn.[1, 48]

Vue nabízí svobodnou volbu při tvorbě komponent ve formě dvou hlavních API – Options a Composition API. Options API můžeme přirovnat k objektovému přístupu, kdežto Composition API využívá funkcionální přístup. Podle [47] Composition API přináší větší flexibilitu a umožňuje efektivnější návrhové vzory pro organizaci a znovupoužitelnost kódu. Z tohoto důvodu v analýze budeme využívat Composition API.

Framework klade důraz na progresivitu, což znamená, že roste s vývojářem a přizpůsobuje se jeho potřebám. Díky tomu si Vue oblíbily společnosti jako Xiaomi, Adobe, Gitlab, Trivago, BMW.[9, 47]

2.4.1 Single-File Components

Základní funkcí Vue jsou tzv. Single-File Components (SFC). Jedná se o hlavní stavební blok frameworku, který reprezenzuje část webové stránky. Komponenta se skládá z šablony, dat komponenty, funkcí a kaskádových stylů. Hlavní výhodu tohoto přístupu představuje znovupoužitelnost. JavaScriptové funkce musí být zapsány mezi párové značky script s atributem setup, šablona do template tagů a styly do style bloku.[23, 47]

```
<script setup>
  import { ref } from 'vue';

  const content = ref('nějaký-obsah');
</script>

<template>
  <div>{{ content }}</div>
</template>

<style scoped>
  div {
    background-color: red;
  }
</style>
```

2.4.2 Reaktivita

V komponentě můžeme uchovávat informace pomocí reaktivních stavů. Oficiální dokumentace doporučuje používat funkci ref, která vyžaduje počáteční hodnotu. K hodnotě stavu pak v rámci skriptu přistupujeme pomocí klíčového slova value. V šabloně nám stačí pouze název stavu. Modifikaci stavu lze provést pomocí přiřazení nové hodnoty. Při změně jakéhokoli stavu komponenty pak Vue automaticky aktualizuje DOM s novými daty.[47]

```
<script setup>
  import { ref } from 'vue';

  const count = ref(0);

  function increment() {
    count.value++;
  }
</script>

<template>
  <button v-on:click="increment">
    Klikli jste na tlačítko {{ count }}x.
```

```
  </button>
</template>
```

2.4.3 Předávání vlastností

Komponenty spolu komunikují pomocí předávání vlastností. V parent komponentě je třeba předat požadovanou hodnotu do proměnné child komponenty. V child komponentě pak definujeme props vlastnost, kterou vytvoříme pomocí funkce defineProps. DefineProps funkce vyžaduje objekt s názvem a datovým typem předávané vlastnosti.

Pro předání vlastnosti z child to parent komponenty se využívá tzv. emitování eventů. V potomku vytvoříme vlastnost emit, v níž nadeklarujeme pole emitovaných hodnot pomocí funkce defineEmits. Dále je třeba definovat jednotlivé emity. První argumentem je název emitu, další argumenty jsou již předávané hodnoty. Rodičovská komponenta musí naslouchat na emitované eventy. Toho lze docílit pomocí @response direktivy, která typicky v callback funkci přeukládá argumenty na lokalní stavy.[23, 47]

```
// Parent.vue
<script setup>
  import { ref } from 'vue';
  import ChildComponent from './Child.vue';

  const someProps = ref({color: 'cervena'});
</script>

<template>
<ChildComponent :color="'cervena'" />
  <ChildComponent :color="someProps.color" />
</template>

// Child.vue
<script setup>
  const props = defineProps({
    color: String
  });
</script>

<template>
  <div :class="color"></div>
</template>
```

2.4.4 Direktivy a eventy

Framework disponuje mnoha direktivami, které umožňují přidávat do šablony různé funkce. Logiku vykreslování umožňují direktivy v-if, v-else-if a v-else. Pro iteraci přes pole hodnot slouží v-for. Mezi další užitečné direktivy patří v-bind a v-model. Díky v-bind je možné přidat jakémukoli elementu dynamickou hodnotu atributu, direktivu můžeme zkrátit i pomocí dvojtečky. Direktiva v-model zase zajistí obousměrné propojení pro formulářové prvky.

```
<script setup>
  import { ref } from 'vue';

  const text = ref('');
</script>

<template>
  <input v-model="text" placeholder="Něco napište">
  <p v-if="text.length > 3">{{ text }}</p>
  <p v-else>Musíte zadat více než 3 znaky</p>
</template>
```

Vue také umožnuje naslouchat na DOM eventy pomocí direktivy v-on. Ta pak vyžaduje libovolný DOM event a callback funkci, jež se vykoná při daném DOM eventu. Můžeme také zvolit ekvivalentní zápis s @. Další možnost představuje využití modifkátorů eventů, které se postarají například o vypnutí výchozího chování prvku.[23, 47]

```
<script setup>
  import { ref } from 'vue';

  const count = ref(0);
</script>

<template>
  <button @click="() => count++">
    Klikli jste na tlačítko {{ count }}x.
  </button>
</template>
```

2.4.5 Životní cyklus

Každá instance Vue komponenty má daný životní cyklus. Ten můžeme rozdělit na 3 části – inicializační část, část, při které se mění data a část, kdy komponenta zaniká. Pro zajištění kontroly životního cyklu slouží tzv. hooks, které se vyznačují tím, že vždy přes svým názvem mají předponu on.

Při inicializaci komponenty můžeme využít hook akce beforeMount či mount. BeforeMount se volá ještě před tím, než se komponenta přidá na stránku. Mount až poté, kdy je vytvořen element komponenty – komponenta však ještě nemusý být v DOM. Před chystanou změnou dat v DOM se volá beforeUpdate, po vykonání změny je zavoláno updated. Před samotným zánikem komponenty Vue volá beforeUnmount. Po dokončení zničení se pak volá unmounted.[23, 47]

2.4.6 State management

Framework Vue v sobě nemá implementovaný žádný sofistikovaný state management. K základní práci se stavy aplikace poslouží reaktivní stavy jednotlivých komponent a jejich sdílení ve stromě komponent.

Pro komplexnější aplikace je třeba využít některou z knihoven třetích stran. Dokumentace doporučuje knihovnu Pinia, o kterou se starají vývojáři samotného Vue. Pinia je inspirována Vuex a využívá principu Composition API. Tato knihovna nabízí jednoduché API pro správu stavů v aplikaci. Hlavním prvkem jsou tzv. stores, do kterých ukládáme globální stavy aplikace. Store vytvoříme pomocí funkce defineStore, která požaduje identifikátor daného store a callback funkci. Ta by měla vracet samotný state definovaný pomocí ref() a akce (funkce, jež mění stavy) nad storem, případně computed state – jiný stav, který je typicky odvozen od původního stavu.[29, 47]

2.4.7 Routování

Samotný Vue framework neposkytuje zabudovanou podporu pro routování. Oficiální dokumentace doporučuje využití knihovny Vue Router, která umožňuje routování na straně klienta. Základní využití routeru spočívá ve vytvoření instance routeru. Tuto instanci inicializujeme polem požadovaných cest aplikace s konkrétními komponentami pro vykreslení. V rámci šablony pak můžeme použít router-link element, jež představuje odkaz na jinou cestu. Pro vykreslení obsahu po změně cesty slouží router-view. V podstatě se jedná o wrapper pro vykreslení komponenty, která je spojena s uživatelem vybranou cestou.

Vue Router disponuje také dynamickým routingem, který umožňuje definovat parametry cesty a vykreslit tedy stránku s dynamickými daty. Mezi další funkce patří např. zanořené routování a routování dle pojmenovaných cest aplikace.[47, 49]

2.4.8 Ekosystém

Vue framework sice nabízí solidní základ pro vývoj webových aplikací, sám o sobě ale není komplexním nástrojem pro vývoj aplikací většího rozsahu. Většina vývojářů využívá kromě frameworku i další knihovny třetích stran, které zvyšují produktivitu a zjednodušují vývoj požadovaných funkcí. V rámci ekosystému lze využít například knihovny pro práci s UI komponenty, routováním, state managementem. Nechybí ani knihovny pro typování, testování, statické generování či formátování kódu.

Při rostoucí komplexitě webových aplikací můžeme zvážit použití pokročilého frameworku Nuxt. Nuxt je postaven na základech Vue.js a poskytuje production-ready nástroje. Konkrétně řeší například pokročilou správu stavů, routing, hydrataci stránek či Server Side Rendering.[6, 47]

2.5 Porovnání analyzovaných frameworků

Analyzované technologie využívají podobné základní koncepty, což vývojáři ocení hlavně v momentě, kdy se chtějí naučit jiný framework. Každý z frameworků však disponuje odlišnou syntaxí, přístupy, možnostmi a API. Díky tomu každá technologie vyniká v jiných oblastech.

Framework Angular je velice robustním frameworkem s mnoha funkcionalitami, které jsou zabudovány uvnitř balíků frameworku. Díky tomu vývojáři disponují téměř všemi základními nástroji pro vývoj webových aplikací. Mezi další výhody můžeme zařadit také přechod z projektu na projekt jiný, jelikož ve většině případů projekty využívají stejné nástroje a konvence. Nevýhodou Angularu je jednoznačně složitější křivka učení, čemuž nepřispívá ani to, že v Angular projektech často využíváme knihovnu RxJS. Do nevýhod bychom také mohli zařadit velikost výsledné aplikace a delší syntax.

Oproti tomu React je populární díky své jednoduchosti a flexibilitě. Nesporno výhodu představuje to, že technologie se dá použít při vývoji jak webových, tak

i mobilních aplikací. Vývojáři mají možnost vybrat si z mnoha knihoven a nástrojů, které využijí k vývoji. Toto může být bráno i negativně, protože vývojáři musí mít přehled o balíčcích a nástrojích, které mohou použít. V důsledku pak může být vývoj aplikace složitější a vývojář rovněž nemusí využít vhodné postupy. Na druhou stranu pro React existuje mnoho návodů či turotiálů, které určitě pomohou při učení se tohoto frameworku.

Svelte je frameworkem nejmladším, zároveň jej však můžeme považovat za nejvíce inovativní. Technologie je vhodná pro začátečníky, protože má pouze minimum boilerplate kódu. Programátoři rozhodně ocení kompilaci zdrojových kódů do nativního JavaScriptu již při sestavení aplikace. Jako výhodu rovněž můžeme uvést pokročilejší optimalizace, které Svelte nabízí. Nevýhodou pak může být menší komunita kolem frameworku, což v důsledku může znamenat menší množství dostupných knihoven a nástrojů. Svelte však umožňuje využití JavaScriptových knihoven, které přímo ovlivňují DOM, což do jisté míry kompenzuje předchozí nevýhody.

V neposlední řadě Vue umožňuje vývojářům využívat jak objektový, tak i funkcionální přístup při tvorbě komponent. Framework, podobně jako Svelte, vyniká v oblasti optimalizace a výkonu. Rozsáhlá komunita Vue se aktivně podílí na vývoji nemalého množství knihoven a nástrojů. Kvůli tomu, že velká část komunity pochází z Číny, mohou být jak technické dokumentace, tak i online zdroje primárně v čínštině. To pak znesnadňuje hledání informací vývojářům v angličtině. Celkově ale můžeme říci, že Vue je frameworkem velice vyváženým. Díky inspirace z jiných frameworků nabízí mnoho propracovaných možností pro programování webových aplikací.

- co zjistím na první pohled, platforma,
- jako bych si četl reklamu ...,
- prvotní srovnání.

3 Testování frameworků

V rámci této kapitoly se zaměříme na srovnání tří vybraných frameworků. V první části navrhneme funkční a nefunkční požadavky demonstračních aplikací. Následně vytvoříme demonstrační aplikaci, která bude obsahovat stejné funkcionality ve všech třech vybraných frameworcích. V poslední části srovnáme ... .

3.1 Návrh aplikace

Začneme návrhem aplikace, kterou později implementujeme v rámci vybraných frameworků. Webová aplikace se bude skládat z několika stránek, které zobrazí navrhnuté funkcionality. Ty následně poslouží při porovnání přístupu a implementace jednolivých nástrojů.

3.1.1 Funkční požadavky

- Správa stavů, předávání vlastností – Counter komponenta:
  1. Uživateli bude zobrazen aktuální stav čítače.
  2. Uživateli bude umožněno zvýšit a snížit hodnotu čítače o 1.
  3. Uživatel bude mít možnost resetovat hodnotu čítače na 0.
- Interakce v uživatelském prostředí – Dropdown komponenta:
  1. Komponenta bude zobrazovat rozbalovací menu.
  2. Po kliknutí na menu se zobrazí seznam položek.
  3. Uživatel bude mít možnost vybrat jednu z položek, nebo zavřít menu.
  4. Aktuálně vybraná položka bude zobrazena v obsahu rozbalovacího menu.
- Reaktivita, asynchronní operace – Translator komponenta:
  1. Uživatel bude mít možnost zadat text, který chce přeložit.
  2. Uživatel bude mít možnost vybrat jazyk, do kterého chce text přeložit.
  3. Přeložený text získáme díky veřejnému API.
  4. Uživateli bude zobrazen přeložený text.
  5. V případě chyby při překladu bude uživateli zobrazena chybová hláška.
- Tvorba formulářů, validace – InvestForm komponenta:
  1. Komponenta bude zobrazovat formulář pro výpočet výnosu investice.

2. Uživatel bude mít možnost zadat vstupní hodnoty formuláře.
3. Formulář bude obsahovat validaci vstupních hodnot a nedovolí uživateli potvrdit formulář s jakoukoli nevalidní hodnotou formuláře.
4. Po potvrzení formuláře bude provedena kalkulace výsledků a ty budou zobrazeny uživateli.

- Modularita, použití knihoven – CountryGuesser komponenta:

1. Data o zemích získáme pomocí veřejného API.
2. V rámci hry bude náhodně vybrána země, kterou bude uživatel hádat.
3. Uživateli se v rámci hry budou postupně zobrazovat informace (nápovědy) o hádané zemi.
4. Uživatel bude mít možnost zadat či vybrat pouze existující název země, který chce tipovat.
5. Uživatel následně uvidí již zadané země a vzdálenost jím dané země od hádané země.
6. Uživateli bude při výhře a prohře zobrazeno modální okno s informacemi o výsledku hry.
7. V případě chyby při získávání dat o zemích bude uživateli zobrazena chybová hláška.

- Routování a layout aplikace:

1. Uživatel bude umožněna navigace mezi jednotlivými stránkami aplikace.
2. Uživatel bude mít možnost zapnout a vypnout tmavý režim aplikace.

3.1.2 Nefunkční požadavky

- Webová aplikace bude uživatelsky přívětivá.
- Webová aplikace bude responzivní a bude se správně zobrazovat na různých zařízeních.
- Implementace bude dodržovat principy čistého kódu, také principy KISS, DRY a SOLID.
- Aplikace bude získávat data z veřejných API.
- Při implementaci budeme dbát na použití moderních technologií a nástrojů.
- Budou využity open source knihovny, nástroje a ikony.

3.2 Implementace webových aplikací

V této kapitole popíšeme implementaci jednotlivých částí aplikace v rámci vybraných frameworků. Použijeme následující nástroje v níže uvedených verzích:

- Angular – verze 17.x.x
- React – verze 18.x.x
- Svelte – verze 4.x.x

Pro grafickou stránku aplikací využijeme CSS framework TailwindCSS [44] a ikony od tvůrců TailwindCSS – Heroicons [19]. Obě knihovny jsou open source a zdarma k použití pod licencí MIT License. K vytvoření nového projektu v jednotlivých frameworcích budeme potřebovat NodeJS a správce balíčků npm (případně jiného správce balíčků).

Tabulka 1: Instalace projektů v jednotlivých frameworcích.

Framework	**Příkazy pro instalaci**
Angular	npm init @angular@latest nazev-projektu cd nazev-projektu npm install -D tailwindcss postcss autoprefixer npx tailwindcss init -p manuální konfigurace TailwindCSS[1] npm run start
React	npm create vite@latest cd nazev-projektu npm install -D tailwindcss postcss autoprefixer npx tailwindcss init -p manuální konfigurace TailwindCSS[2] npm run dev
Svelte	npm create vite@latest cd nazev-projektu npx svelte-add@latest tailwindcss[3] npm install npm run dev

[1] https://tailwindcss.com/docs/guides/angular
[2] https://tailwindcss.com/docs/guides/vite#react
[3] https://github.com/svelte-add/tailwindcss

3.2.1 Angular

Správa stavů, předávání vlastností

Pro implementaci jednoduchého counteru nejprve vytvoříme counter komponentu. Můžeme začít se strukturou HTML značek pro hlavní komponentu.

```
// Soubor counter.component.html

<div class="bg-gray-200 p-6 rounded-md shadow-md">
  <p class="text-xl font-semibold mb-4">Current count: {{ count }}</p>

  <div class="flex gap-4">
    <counter-button
      [className]="'bg-blue-500 text-white hover:bg-blue-600'"
      (buttonClicked)="increment()"
    >
      Increment
    </counter-button>

    <!- další tlačítka... ->
  </div>
</div>
```

Jelikož potřebujeme opakovaně použít logiku jednotlivých tlačítek, vytvoříme komponentu counter-button. Ta může přijímat například CSS styly nebo přes output (*EventEmitter*) posílat informaci o kliknutí na tlačítko směrem nahoru ve stromě komponent.

```
// Soubor counter-button.component.ts

import {CommonModule} from '@angular/common';
import {Component, EventEmitter, Input, Output} from '@angular/core';

// Nastavení komponenty.
@Component({
  selector: 'counter-button',
  standalone: true,
  templateUrl: './counter-button.component.html',
  imports: [CommonModule],
})
export class CounterButtonComponent {
  // Vstupní vlastnost komponenty.
  @Input() public className = '';

  // Výstupní vlastnost komponenty.
  @Output() public buttonClicked = new EventEmitter<void>();
}
```

Funkci *emit() EventEmitteru* zavoláme na tlačítku v counter-buttonu právě tehdy, když uživatel klikne na tlačítko – použijeme listener ve formě *(click)*. K

propsání textu či jiných elementů nebo komponent mezi párovými tagy <counter-button> pak poslouží párový či nepárový element <ng-content />.

```
// Soubor counter-button.component.html

<button
  class="px-4 py-2 rounded-md focus:outline-none"
  [ngClass]="className"
  (click)="buttonClicked.emit()"
>
  <!- ng-content slouží k vykreslení obsahu, který vložíme
   mezi párové tagy (selectory) dané komponenty. ->
  <ng-content></ng-content>
</button>
```

Následně v counter komponentě importujeme třídu CounterButtonComponent a do všech elementů counter-button předáme jejich vstupy a výstupy. Námi defikovanovanému outputu *buttonClicked* předáme v šabloně metodu, která se vykoná po emitu (kliknutí na tlačítko ve vnořené komponentě) a metodu zavoláme pomocí kulatých závorek. V rámci counter komponenty pak definujeme stav jako vlastnost *count* na třídě. Vlastnost pak můžeme modifikovat skrze metody třídy, které voláme v outputu *buttonClicked*.

```
// Soubor counter.component.ts

import {CommonModule} from '@angular/common';
import {Component} from '@angular/core';
import {CounterButtonComponent} from './button/counter-button.component';

@Component({
  selector: 'counter',
  standalone: true,
  templateUrl: './counter.component.html',
  imports: [CommonModule, CounterButtonComponent],
})
export class CounterComponent {
  protected count = 0;

  protected increment(): void {
    this.count++;
  }

  protected decrement(): void {
    this.count-;
  }

  protected reset(): void {
    this.count = 0;
  }
}
```

- šablony + logika komponenty
- správa stavů (reaktivita)
- body k vypíchnutí: boilerplate frameworku

Interakce v uživatelském prostředí

Při vytváření jakékoli UI komponenty můžeme začít šablonou, nebo definovat funkční stránku. My začneme s tvorbou šablony. V případě vlastního dropdown samotným tlačítkem a seznamem možností. Otevření možností zajístíme tak, že na tlačítko přidáme click listener. Funkčnost pak zajistíme díky modifikaci stavu *isOpen*, který se provede při volání metody *toggleDropdown*. V rámci této metody je třeba zavolat i *event.stopPropagation()*. Předejdeme tak potenciální chybě ve formě tzv. event bubblingu – spuštění událostí na prvcích odlišných od cílového.

```
// Část souboru dropdown.component.html

<div class="rounded-md shadow-sm">
  <!- Pro poslouchání na události v DOMu můžeme
    použít syntaxi: (NÁZEV_UDÁLOSTI)="OBSLUŽNÁ_METODA". ->
  <button
    type="button"
    class="" <!- Statické styly... ->
    [ngClass]="buttonStyles + ' ' + sizeStyles"
    (click)="toggleDropdown($event)"
  >
    {{ selectedOption ? selectedOption.label : placeholder }}
    <!- Pro podmíněné vykreslovaní můžeme využít bloky @if, @else if, @else. ->
    @if (isOpen) {
      <arrow-up-icon />
    } @else {
      <arrow-down-icon />
    }
  </button>
</div>
```

Podmíněně pak můžeme vypsat list možností, které získáme v jednom z inputů. Pro vypsání všech možností použijeme blok *@for*. K vybraní konkrétní možnosti použijeme zase *(click)* a do obslužné metody pošleme aktuální prvek v poli – option. Metoda *handleOptionClick* pak zajistí uložení aktuálně vybrané možnosti, zavření dropdownu a vyemitování vybrané možnosti do rodičovské komponenty.

```
// Část souboru dropdown.component.html

@if (isOpen)
```

```
<div
  class="" <!- Statické styly... ->
  [ngClass]="divStyles"
>
  <div class="py-1" role="menu"> <!- WAI-ARIA atributy... ->
    <!- Pro vykreslení listu (pole hodnot) můžeme využít blok @for. ->
    @for (option of options; track option.value)
      <button
        class="block w-full text-left px-4 py-2 text-sm hover:text-gray-900"
        [ngClass]="optionStyles"
        role="menuitem"
        (click)="handleOptionClick(option)"
      >
         option.label
      </button>

  </div>
</div>
```

V případě, že máme dropdown otevřen a chceme jej po kliknutí mimo tentýž dropdown bezpečně zavřít, nehledě na počet vykreslených dropdown komponent na stránce, budeme postupovat následovně. Pro každou komponentu vytvoříme unikátní vlastnost ve formě ID. To pak dynamicky umístíme na kořenový element dropdownu.

```
// Část souboru dropdown.component.ts

protected dropdownId = `id-$crypto.randomUUID()`;

// Část souboru dropdown.component.html

<div class="relative inline-block text-left" [id]="dropdownId">
```

V komponentě pak budeme naslouchat na události v DOM pomocí dekorátoru *@HostListener*. Dekorátor přijímá DOM událost, na který má poslouchat – *document:pointerdown*, případně další argumenty nebo také formu vypublikované události. Pod dekorátorem pak definujeme obslužnou metodu, která se volá při emitu specifikované události. V rámci metody pak zajistíme uzavření aktuálně otevřeného dropdownu.

```
// Část souboru dropdown.component.ts

@HostListener('document:pointerdown', ['$event.target'])
onClickOutsideDropdown(target: HTMLElement): void {
  if (this.isOpen && !target.closest(`#${this.dropdownId}`)) {
    this.isOpen = false;
  }
}
```

Dropdown pak může mít různé inputy, které povedou k lepší znovupoužitelnosti. Hodnotu inputu (konkrétně např. *defaultValue*) v komponentě získáme v metodě životního cyklu *OnInit*. Styly ve formě JavaScriptových hodnot do šablony přidáme pomocí *ngClass*. Když těchto hodnot potřebujeme na elementu více, zřetězíme předávané hodnoty pomocí JavaScriptu. Další možnost spočívá ve sloučení požadovaných stylů na úrovni třídy.

- body k vypíchnutí: dynamické stylování, logika v template
- problémy: zavírání posledně otevřeného dropdownu před otevřením dalšího D.
- výhody frameworku: podle bodů nahoře..., tvorba typů ve Svelte

Reaktivita, asynchronní operace

Pro ukázku reaktivity a asynchronních operací můžeme vytvořit komponentu, která bude překládat zadaný text do vybraného jazyka. Začneme tedy vytvořením rodičovské komponenty, která při změně vlastností (zadaného textu uživatelem a výstupního jazyka) zavolá API, které vrátí přeložený text. V rámci této komponenty vytvoříme vnořené komponenty, které budou sloužit k zadání vstupního textu, výběru jazyka a zobrazení výsledku.

LanguageDropdownComponent umožní uživateli vybrat jazyk, do kterého chce přeložit text. Přes *EventEmitter* aktualizujeme výstupní jazyk v rodičovské komponentě. V rámci obslužné metody *handleLanguageChange* pak také aktualizujeme hodnotu vlastnosti *inputValuesChanges$*. Tato vlastnost je *Subject*, speciální typ observable, z knihovny RxJS. Později dovolí na základě změny hodnoty poslat dotaz na server ve správný moment. Podobným způsobem poté můžeme registrovat událost změny vstupního textu – naslouchat na změnu vstupního textu.

```
// Část souboru translator.component.ts

protected handleLanguageChange(outputLanguage: Option): void {
  this.outputLanguage = outputLanguage.value;
  // Synchronní aktualizace hodnoty observable (v tomto případě Subjectu).
  // Slouží pro následné operace při změně hodnoty observable.
  this.inputValuesChanges$.next(outputLanguage.value);
}
```

Zadání vstupního textu pak může řešit komponenta TranslationInputComponent, která obdobným způsobem aktualizuje hodnotu vstupního textu v rodičovské

komponentě. Aktuální hodnotu formulářového prvku nastavíme pomocí *[ngModel]*. Pro naslouchání na změnu hodnoty formulářového prvku zase využijeme *(ngModelChange)*.

```
// Část souboru translation-input.component.html

<textarea
  autosizeTextArea
  class="block w-full min-h-0 p-3 pr-12 pb-8 resize-none !outline-none"
  placeholder="Type to translate ..."
  [ngModel]="inputText"
  (ngModelChange)="handleInputChange($event)"
>
</textarea>
```

V případě, že potřebujeme aktualizovat výšku textového pole na základě jeho obsahu, můžeme využít vlastní direktivu AutosizeTextAreaDirective. V konstruktoru direktivy získáme element, na který přidáme tuto direktivu. Dále budeme potřebovat třídu *Renderer2*, která umožňuje manipulovat s DOM. V direktivě budeme naslouchat na změnu hodnoty textového pole pomocí dekorátoru *@HostListener* a události input. Následně v rámci obslužné metody zajistíme aktualizaci výšky.

Změny hodnoty vlastnosti *inputValuesChanges$* začneme odebírat pomocí *subscribe*. Abychom předešli dotazování serveru ihned po změně hodnoty vlastnosti *inputValuesChanges$*, použijeme operátor *debounceTime*. Ten povolí poslat dotaz na server až po uplynutí určité doby od poslední změny, kterou můžeme nastavit. Subscribe zavolá věřejnou metodu služby (*getTranslation*), která vrací přeložený text. Nakonec, aby dotazování serveru fungovalo, je třeba metodu *setupInputChangeSubscription* zavolat v konstruktoru nebo hooku *OnInit*.

```
// Část souboru translator.component.ts

private setupInputChangeSubscription(): void {
  // Naslouchá změnám vstupního textu a výstupního jazyka.
  // Operátor debounceTime zajistí, že se změna vstupního textu
    nebo výstupního jazyka vyhodnotí až po uplynutí 300 ms.
  // Dále operátor distinctUntilChanged zajišťuje,
    že se změna vyhodnotí pouze v případě, kdy je odlišná od předchozí hodnoty.
  // Operátor takeUntil() zajišťuje,
    že se subscription zruší při zničení komponenty.
  // Pokud se změní vstupní text nebo výstupní jazyk,
    v rámci methody subscribe se spustí překlad.
  this.inputValuesChanges$
    .pipe(debounceTime(300), distinctUntilChanged(), takeUntil(this.destroy$))
    .subscribe(() => this.triggerTranslation());
}
```

V rámci služby TranslationService použijeme třídu *HttpClient* z Angular modulů, která umožňuje odesílat HTTP požadavky na server. Službu *HttpClient* získáme v konstruktoru, kde ji pomocí klíčového slova *private* přiřadíme do vlastností třídy. Pokračujeme zavoláním metody *post* na HTTP klientovi s patřičným nastavením. Ze serveru pak v odpovědi dostaneme přeložený text. Pokud úspěšná odpověď ze serveru obsahuje složitější strukturu, ze které potřebujeme získat jen nějakou část, pak s konverzí odpovědi pomůže RxJS operátor *map()*. Metoda *getTranslation* vrací observable, v translator komponentě proto hodnoty odebíráme pomocí metody *subscribe*.

```
// Část souboru translation.service.ts

return this.httpClient
  .post<TranslationResponseData>(url, body, options)
  .pipe(map(data => this.convertToOutputText(data)));

// Část souboru translator.component.ts

// Slouží ke zrušení subscriptions při zničení komponenty.
private destroy$: Subject<void> = new Subject();
// Slouží k naslouchání na změny vstupního textu a výstupního jazyka.
private inputValuesChanges$ = new Subject<string>();

public ngOnDestroy(): void {
  // Slouží k manuálnímu unsubscribe všech observables při zničení komponenty.
  this.destroy$.next();
  this.destroy$.complete();
}

this.translationService
  .getTranslation(this.inputText, this.outputLanguage)
  .pipe(
    // Zajišťuje, že se subscription zruší při zničení komponenty.
    takeUntil(this.destroy$),
    // Zachytí chybu v observable.
    catchError(error => this.handleError(error)),
  )
  // V metodě subscribe dostaneme transformovanou odpověď
    (v rámci next callbacku) nebo chybu (v rámci error callbacku).
  // Po poslední úspěšné aktualizaci observable se volá callback funkce complete.
  .subscribe({
    next: response => (this.outputText = response),
    error: error => (this.error = error),
    complete: () => (this.loading = false),
  });
```

V momentě, kdy obdržíme odpověď ze serveru, zobrazíme přeložený text uživateli. K tomu poslouží TranslationOutputComponent, které na vstupu předáme

výstupní text spolu s dalšími vstupními vlastnostmi. V rámci šablony pak podmíněně vykreslíme přeložený text, chybu nebo načítání.

Při zarovnání vstupního a výstupního pole v UI si musíme dát pozor na to, že šířku je potřeba nastavit již v prvním potomku div elementu, na kterém nastavíme flexbox. Důvod spočívá v tom, že Angular v DOMu vytváří element pro každou komponentu.

```
// Část souboru translation.service.ts

<div class="flex text-xl">
  <translation-input
    <!- Vstupní a výstupní vlastnosti... ->
    class="relative w-1/2"
    <!- Šířka musí být nastavena zde. ->
  />

  <translation-output
    <!- Vstupní a výstupní vlastnosti... ->
    class="relative w-1/2"
    <!- Šířka musí být nastavena zde. ->
  />
</div>
```

- předávání vlastností nahoru a dolů
- fetchování dat
- body k vypíchnutí: velice odlišné reakce na změny, stylování komponent nebo elementů, update textarey (hodnoty), jiné řešení modularity (update stylů textarey)
- problémy:
- výhody frameworku: předávání vlastností má nej Svelte

Tvorba formulářů, validace

Angular je flexibilní z pohledu možností tvorby formulářů. My použijeme reaktivní formuláře, jelikož jsou flexibilnější a umožní nám jednodušší reakce na změny prvků. Vytvoříme komponentu zaměřenou na jednoduché investiční kalkulace. Bude obsahovat dvě vnořené komponenty: formulář pro zadání vstupních dat a komponentu výsledku kalkulace, která se zobrazí po potvrzení formuláře.

Začneme s tvorbou reaktivního formuláře. Typ *InvestForm* popisuje strukturu souvisejících formulářových prvků formuláře.

```
// Část souboru invest-form.component.ts

type InvestForm = FormGroup<{
  oneOffInvestment: FormControl<number | null>;
  investmentLength: FormControl<number | null>;
  averageSavingsInterest: FormControl<number | null>;
  averageSP500Interest: FormControl<number | null>;
}>;
```

Protože prvků budeme mít více, deklarujeme formulářovou skupinu jako vlastnost třídy, ve které následně definujeme samotné formulářové prvky. Vlastnost *investForm* pak umožní přístup k hodnotám formuláře a jeho validaci. Zde narazíme na problém s nenastavením počáteční hodnoty vlastnosti přímo nebo v konstruktoru. Můžeme ho vyřešit za pomoci vykřičníku – řekneme tak TypeScriptu, že obsah proměnné je nenulový. Další možností je vypnout pravidlo *strictPropertyInitialization* v souboru *tsconfig.json*.

```
// Část souboru invest-form.component.ts

protected investForm!: InvestForm;
```

Hodnotu vlastnosti *investForm* nastavíme pomocí metody *initializeInvestForm* v rámci *OnInit* hooku. Tento postup zvolíme, protože chceme nastavovat počáteční hodnoty formuláře na základě vstupní vlastnosti *defaultValues*. Důvodem je, že hodnoty vstupních vlastností jsou v komponentě dostupné nejdříve v rámci hooku *OnInit*.

Metoda *initializeInvestForm* vrátí instanci třídy *FormGroup*, kterou vytvoříme pomocí třídy *FormBuilder* ze základního balíčku *@angular/forms*. Argumentem pro metodu *group* pak je objekt, který popisuje strukturu formuláře. Vlastnosti objektu budou klíče formulářových prvků a jejich hodnoty pole, kde první prvek bude počáteční hodnota a druhý prvek pole validátorů.

```
// Část souboru invest-form.component.ts

private initializeInvestForm(): InvestForm {
  // Vytvoření formuláře s výchozími hodnotami
    (případně vlastnostmi) a validátory.
  // Jednotlivé prvky FormGroup bývají označovány jako FormControl.
  return this.fb.group({
    oneOffInvestment: [
      this.defaultValues.oneOffInvestment,
      [Validators.required, Validators.min(20), Validators.max(99_999_999)],
    ],
```

```
    // Další formulářové prvky...
  });
}
```

V šabloně následně propojíme formulářovou skupinu s formulářem. K tomu poslouží direktiva *[formGroup]* a její hodnotu nastavíme na vlastnost *investForm*. V rámci formuláře pak vytvoříme formulářové prvky, které propojíme direktivou *formControlName*. Hodnota pak musí odpovídat klíči prvku ve formulářové skupině. Pro zajištění efektivní obsluhy chyb formuláře můžeme využít getter metody, které vrátí konkrétní formulářový prvek.

```
// Část souboru invest-form.component.html

<form [formGroup]="investForm" (ngSubmit)="onSubmit()">
  <div class="md:flex md:gap-4">
    <div class="mb-4 md:w-1/2">
      <input-label id="oneOffInvestment">
        One-off investment (20-99.999.999€)
      </input-label>

      <!- Direktiva formControlName slouží k propojení inputu
        s odpovídajícím FormControl v FormGroup. ->
      <input
        id="oneOffInvestment"
        type="number"
        formControlName="oneOffInvestment"
        class="" <!- Statické styly... ->
      />

      @if (oneOffInvestmentControl.errors?.['required']) {
        <p class="text-red-500 text-xs italic mt-1">
          Please enter a valid amount of one-off investment (positive number).
        </p>
      }
      <!- Další chybové hlášky... ->
    </div>

    <!- Další formulářové prvky... ->
  </div>
</form>
```

Dále vytvoříme tlačítko s typem submit, přes které uživatel formulář potvrdí. Na form značku přidáme *(ngSubmit)*, který vyemituje událost při potvrzení formuláře. Obslužná metoda pak prostřednictvím výstupové vlastnosti publikuje aktuální hodnotu reaktivního formuláře do rodičovské komponenty.

V rámci rodičovské komponenty tedy vykreslíme samotný formulář a při jakémkoli potvrzení formuláře získáme aktuální hodnoty z formuláře díky outputu.

Hodnoty formuláře pak dostaneme v obslužné metodě *handleFormChanged*. Pomocí služby FutureValuesCalculatorService tyto hodnoty transformujeme do požadovaného formátu. Výsledek uložíme do vlastnosti *futureValues*.

Když jsou hodnoty vypočteny, vykreslíme je na stránce prostřednictvím komponent future-values-info a future-value-info. První z komponent slouží k rozložení výsledků do požadovaného formátu a vytvoření komponent pro jednotlivé výsledky. Komponenta future-value-info pak přijímá vstupní vlastnost, kterou v šabloně před vykreslením v DOM přetransformujeme díky rouře (*LocalizedNumberPipe*).

```
// Část souboru future-value-info.component.html

<p class="text-5xl font-bold"> futureValue | localizedNumber </p>
```

Stejného výsledku bychom mohli dosáhnout i přes metodu na třídě. Tento přístup Angular nedoporučuje, jelikož metody se v rámci šablony spouští opakovaně a mohou způsobit problémy s výkonem. Oproti tomu roura umožní lepší znovupoužitelnost a přehlednost.

```
// Soubor localized-number.pipe.ts

import Pipe, PipeTransform from '@angular/core';

// Roura, která převede číslo na formátovaný string s měnou (€).
@Pipe({name: 'localizedNumber', standalone: true})
export class LocalizedNumberPipe implements PipeTransform {
  public transform(value: number): string {
    return `${value.toLocaleString('de-DE')}€`;
  }
}
```

Modularita, použití knihoven

V této sekci vytvoříme webovou hru, kde cílem uživatele bude uhádnout název státu na základě poskytnutých nápovědí. Práci si ulehčíme pomocí externích knihoven a služeb. Ve hře se postupně bude odkrývat 8 nápovědí, které by měly pomoct s uhádnutím daného státu. Klíčovým prvkem je textové pole, přes které uživatel zadává názvy hádaných zemí a tlačítko pro potvrzení. Součástí je také seznam již zadaných hádaných zemí a modální okna sloužící k vyhodnocení hry.

Začneme s implementací rodičovské komponenty, jež bude získávat data o všech zemích světa z veřejného API. Další zodpovědností této komponenty bude

vykreslování odpovídajících stavů při získávání dat – stav načítání, úspěšné získání dat a chyba při získávání dat. Vytvoříme službu CountryService, díky které budeme moci získávat data o zemích. Konkrétně k tomu využijeme metodu *getAllCountries*, která vrátí observable pole všech zemí. Výsledek registrace služby a přímé zavolání metody *getAllCountries* uložíme do vlastnosti třídy.

```
// Soubor country-guesser-wrapper.component.ts

protected countries$: Observable<Countries>
  = inject(CountryService).getAllCountries();
```

V šabloně posléze potřebujeme odebírat hodnotu z observable. Práci v šabloně výrazně ulehčí knihovna *ngx-load-with* [20]. Tato knihovna poskytuje integrovanou podporu načítání a zpracování chyb. To programátorovi umožní využívat předdefinované šablony pro dané stavy bez nutnosti další implementace. Navíc se programátor nemusí starat o zrušení odběru observable.

```
// Část souboru country-guesser-wrapper.component.html

<ng-container
  *ngxLoadWith="countries$ as countries;
  loadingTemplate: loading; errorTemplate: error"
>
  <country-guesser [countries]="countries" />
</ng-container>

<!- #loading je reference na načítací šablonu. ->
<ng-template #loading>
  <!- Vlastní načítací šablona... ->
</ng-template>

<!- #error je reference na chybovou šablonu. ->
<!- let-error umožňuje přístup k chybě. ->
<ng-template #error let-error>
  <!- Vlastní chybová šablona... ->
</ng-template>
```

V rámci komponenty country-guesser budeme implementovat jednotlivé herní prvky, komponenta také bude vyhodnocovat průběh hry. Definujeme tedy vlastnosti třídy, které budou reprezentovat stav a průběh hry. V hooku *OnInit* získáme náhodou zemi (zemi pro uhádnutí). Dále zde zavoláme veřejnou metodu *usePolyfill* na službě CountryFlagPolyfillService, která zajistí zobrazení ikon vlajek v prohlížečích, které nepodporují zobrazení vlajek. Do komponenty také přidáme obslužné metody

handleEvaluateGuessAndUpdateState a *handleSetInitialState*, ve kterých implementujeme logiku hry. V šabloně následně vykreslíme UI komponenty hry a podmíněně modální okna při výhře či prohře.

V metodě *usePolyfill* služby CountryFlagPolyfillService zavoláme funkci *polyfillCountryFlagEmojis* z knihovny *country-flag-emoji-polyfill* [45]. Pokud prohlížeč uživatele nepodporuje zobrazení ikon vlajek, ale podporuje emojis a webové fonty, skrze funkci *polyfillCountryFlagEmojis* knihovna přidá webový font do HTML hlavičky. Font Twemoji Country Flags pak umožní zobrazení vlajek. Aby se programaticky přidaný font použil, nesmíme zapomenout nastavit font-family pravidlo v rámci CSS stylů.

```
// Část souboru styles.css

@layer base {
  html {
    font-family: 'Twemoji Country Flags', 'ALTERNATIVNÍ_FONTY...';
  }
}
```

HintBoxesComponent postupně vykreslí nápovědy. Při jakékoli změně vstupních vlastností vytvoříme pole nápověd pomocí vlastnosti randomCountry. V šabloně iterujeme přes pole nápověd a vykreslíme jednotlivé nápovědy. Vlastnost hintEnabled nastavíme pomocí indexu a vstupní vlastnosti hintsEnabledCount. Samotný hint-box pak dynamicky vykreslí název a SVG ikonu nápovědy, textovou nápovědu, případně obrázek vlajky státu.

Pokračujeme implementací komponenty country-guess-input, která uživateli umožní zadat svůj tip. Začneme šablonou, kde vytvoříme formulářový prvek pro zadání názvu země a potvrzovací tlačítko. Dále podmenu textového pole, které zobrazí nejpodobnější země na základě zadaného textu – filtrované země a chybové hlášky. Můžeme také rovnout přidat obslužné metody pro akce a události nad formulářem, které následně postupně doimplementujeme.

V souboru *country-guess-input.component.ts*, tedy v rámci třídy CountryGuessInputComponent, při změně vstupních vlastností (v hooku *OnChanges*) aktualizujeme vlastnost *countriesWithoutAlreadyGuessed* a *filteredCountries*. V případě první vlastnosti jde o pole všech zemí bez těch, které uživatel již hádal. Druhá vlastnost poté představuje pole počátečních 8 prvků vlastnosti *countriesWithoutAlrea-*

dyGuessed. Metoda *handleGuessButtonClick* zavolá obslužnou metodu rodičovské komponenty, která vyhodnotí tip a aktualizuje stav hry. Aktualizujeme také hodnoty aktuálního tipu, filtrovaných zemí a uzavřeme podmenu, k čemuž slouží metoda *handleChangeSelectedGuess* volaná i napřímo z šablony. Tělo metody *handleInputChange* převede uživatelům tip do správného formátu a pomocí převedené hodnoty aktualizuje aktuální tip spolu s filtrovanými zeměmi. Metoda *handleKeyDown* se postará o interakce s podmenu pomocí klávesnice. Skrze šipky nahoru a dolů povolíme uživateli vybrat hádanou zemi. Enter umožní změnu aktuálního tipu názvu země na právě tu, kterou uživatel označil v podmenu. Escape poslouží k uzavření podmenu.

Pomocná metoda *updateGuessAndFilteredCountries* pak modifikuje vlastnost *currentGuess*. Následně pomocí metody *getFilteredCountries* získá aktuálně filtrované země na základě uživatelova tipu. Dále nastaví vlastnost *isValidGuess*, která určuje, zda je uživatelův tip validní (taková země existuje). V neposlední řadě se metoda stará i o aktualizaci vlastnosti *selectedGuessIndex*, jež určuje, která země je vybraná v podmenu. K tomu slouží metoda *clampSelectedGuessIndex*, která index udrží v požadovaném rozmezí (0 až počet filtrovaných zemí). Metoda *getFilteredCountries* získává filtrované země na základě vlastnosti *currentGuess*. Pomocná metoda *changeSelectedGuessIndex* aktualizuje vlastnost *selectedGuessIndex* o hodnotu předanou v argumentu. K převodu tipu uživatele slouží pomocná metoda *convertToFormattedGuess*. Metoda zajistí, aby tip začínal velkým písmenem a zbytek řetezce byl složen z malých písmen.

Implementujeme komponentu guessed-countries-list, jenž zobrazí seznam již hádaných zemí. Mezi vstupními vlastnostmi bude pole všech zemí (*countries*), pole hádaných zemí uživatelem (*guessedCountries*) a také země, kterou uživatel musí uhodnout (*randomCountry*). Pomocí vstupních vlastností a služby EnrichGuessedCountriesService získáme pole hádaných zemí s jejich vlajkou a vzdáleností od *randomCountry* (*distanceFromRandomCountry*). Služba EnrichGuessedCountriesService ke každé hádané zemi přidá vlajku z pole všech zemí a vypočte *distanceFromRandomCountry*. Pro vypočtení vzdálenosti použijeme funkci *getDistanceBetweenTwoPoints* a vlastnost *latlng*, kterou získáme z API při získávání všech zemí. Funkci *getDistanceBetweenTwoPoints* importujeme z knihovny *calculate-distance-*

between-coordinates [26]. Hodnotu vlastnosti *enrichedGuessedCountries* aktualizujeme v rámci hooku *OnChanges*. Seznam hádaných zemí následně vykreslíme v šabloně.

Pokračujeme implementací modálních oken, které se zobrazí při výhře či prohře. Vlastnosti *isWinModalOpen* a *isLoseModalOpen*, určující, zda se mají okna zobrazit, bychom již měli aktualizovat v rámci metody *handleEvaluateGuessAndUpdateState* v CountryGuesserComponent. Oběma modálním oknům předáme vlastnost *randomCountry* a output *handleClose* v podobě obslužné události, která se vyvolá při zavření modálního okna. Výhernímu modálu také vlastnost *totalGuessesNeeded*, jíž využijeme v obsahu okna. Obě modální okna budou velice podobné, a proto je vhodné vytvořit komponentu base-modal, která bude sloužit jako šablona pro obě okna. BaseModalComponent bude přijímat titulek, obsah modálního okna a *handleClose* jako výstupní vlastnost. Šablona base-modal pak vykreslí základní strukturu modálního okna, s dynamicky nastaveným titulkem, obsahem a obslužnou metodou volanou při zavření modálního okna.

Routování a layout aplikace

Demonstrační aplikace bude složena z hlavičky, patičky a samotného obsahu, v nemž se vykreslí jednotlivé komponenty. Mezi jednotlivými stránkami se uživatel bude moct přepínat pomocí navigačního menu.

K routování mezi jednotlivými stránkami využijeme modul Router přímo od Angularu. Nejprve vytvoříme cesty (*routes*) pro jednotlivé stránky v souboru *app.routes.ts*. Proměnnou *routes* vytvoříme dle předpisu *Routes* a následně exportujeme. Cesty pak poskytneme routeru v rámci *app.config.ts*.

```
// Část souboru app.routes.ts

import {Routes} from '@angular/router';

export const routes: Routes = [
  {
    title: 'Home',
    path: '',
    component: LandingComponent,
    pathMatch: 'full',
  },
  {
    title: 'Counter',
```

```
    path: 'counter',
    component: CounterComponent,
  },
  // Další cesty...
  {path: '**', component: PageNotFoundComponent},
];

// Část souboru app.config.ts

export const appConfig: ApplicationConfig = {
  // V tomto nastavení poskytujeme služby a poskytovatele pro celou aplikaci.
  providers: [
    provideRouter(routes),
    // Další poskytované služby...
  ],
};
```

Pokračujeme vytvořením požadované struktury stránek v AppComponent. Šablona bude obsahovat hlavičku, patičku a obsah, který vykreslíme pomocí elementu *router-outlet*.

```
// Soubor app.component.html

<div class="min-h-screen flex flex-col">
  <app-header />

  <main class="flex-grow p-8">
    <!- Router-outlet vykresluje šablonu (komponentu) pro aktuální URL adresu. ->
    <router-outlet></router-outlet>
  </main>

  <app-footer />
</div>
```

V rámci komponenty hlavičky pak vytvoříme navigační menu, které bude obsahovat odkazy na jednotlivé stránky. Můžeme se inspirovat například architekturou a vzhledem navigačního menu Flowbite. Pokračujeme vypsáním všech cest aplikace, k čemuž využijeme direktivy *routerLink*, *routerLinkActive*, *routerLinkActiveOptions* a referenci *#link*. Do *routerLink* předáme patřičnou cestu a *routerLinkActive* umožní naslouchat na aktuální URL. Direktiva *routerLinkActiveOptions* pak přepíše výchozí nastavení *routerLinkActive*. Díky referenci *#link* získáme informaci o tom, zda je odkaz aktivní. To využijeme při podmíněném nastavení správných CSS tříd na aktivních a neaktivních odkazech.

```
// Část souboru header.component.html

@for (route of appRoutes; track route.title) {
```

```
  <li>
    <a
      [routerLink]="route.path"
      routerLinkActive
      [routerLinkActiveOptions]="routerLinkActiveOptions"
      #link="routerLinkActive"
      class="block py-2 pr-4 pl-3 lg:p-0"
      [ngClass]="{
        'STATICKÉ STYLY PRO AKTIVNÍ ODKAZ...': link.isActive,
        'STATICKÉ STYLY PRO NEAKTIVNÍ ODKAZ...': !link.isActive
      }"
      ariaCurrentWhenActive="page"
    >
      {{ route.title }}
    </a>
  </li>
}
```

Přepínání barevného režimu, otevírání a zavírání mobilní navigace implementujeme pomocí obslužným metod a vlastností třídy. Informaci o tom, zda má uživatel zapnutý tmavý režim ukládáme do LocalStorage v prohlížeči. Při kliknutí na tlačítko pro přepnutí režimu zavoláme metodu *toggleDarkMode*, která změní hodnotu vlastnosti a uloží ji do LocalStorage.

```
// Část souboru header.component.ts

protected toggleDarkMode(): void {
  this.isDarkMode = !this.isDarkMode;
  this.updateDarkMode();
}

private updateDarkMode(): void {
  if (this.isDarkMode) {
    document.documentElement.setAttribute('data-mode', 'dark');
    localStorage.setItem('data-mode', 'dark');
  } else {
    document.documentElement.removeAttribute('data-mode');
    localStorage.removeItem('data-mode');
  }
}
```

3.2.2 React

Správa stavů, předávání vlastností

Při implementaci jednoduchého čítače začneme tím, že vytvoříme Counter komponentu. Ta bude mít stav *count* a setter *setCount* pro tento stav.

```
// Část souboru Counter.tsx
```

```
function Counter() {
  const [count, setCount] = useState(0);
}

export default Counter;
```

Dále vytvoříme komponentu Button kvůli principu DRY a celkově znovupoužitelnosti kódu. Typ *ButtonProps* obsahuje vlastnosti, které můžeme tlačítku předat – *className*, *onClick* a *children*. Díky tomu, že typ rozšiřuje *ButtonHTMLAttributes<HTMLButtonElement>*, můžeme předat do komponenty i další běžné atributy HTML tlačítek (např. type, value, disabled).

```
// Část souboru Button.tsx

interface ButtonProps extends ButtonHTMLAttributes<HTMLButtonElement> {
  className: string;
  onClick: () => void;
  children: ReactNode;
}
```

V rámci argumentu Button komponenty použijeme ES6 destructuring assignment pro získání vlastností. Z objektu vlastností získáme *className* a *children*, ostatní vlastnosti ponecháme zabalené v proměnné props pomocí spread operátoru. Nyní můžeme vytvořit JSX pro samotné tlačítko. Vlastnost *className* přidáme do tříd tlačítka. Pomocí *children* můžeme do tlačítka vložit libovolný obsah, který bude mezi párovými značkami <Button>. Všechny ostatní vlastnosti pomocí spread operátoru předáme přímo tlačítku.

```
// Část souboru Button.tsx

function Button({className, children, ...props}: ButtonProps): JSX.Element {
  return (
    <button
      type="button"
      className={`px-4 py-2 rounded-md focus:outline-none ${className}`}
      {...props}
    >
      {/* children slouží k vykreslení obsahu,
        který vložíme mezi párové tagy dané komponenty. */}
      {children}
    </button>
  );
}

export default Button;
```

V Counter komponentě v rámci JSX vrátíme hodnotu stavu *count* a vykreslíme Button komponenty, jimž předáme potřebné vlastnosti. Pro aktualizaci stavu

využijeme vlastnost *onClick*, které předáme anonymní funkci (arrow function) a v ní zavoláme *setCount*.

```
// Část souboru Counter.tsx

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div className="bg-gray-200 p-6 rounded-md shadow-md">
      <p className="text-xl font-semibold mb-4">Current count: {count}</p>

      <div className="flex gap-4">
        <Button
          className="bg-blue-500 text-white hover:bg-blue-600"
          onClick={() => setCount(count + 1)}
        >
          Increment
        </Button>

        {/* Další komponenty Button... */}
      </div>
    </div>
  );
}
```

Interakce v uživatelském prostředí

Pro vytvoření jakékoliv UI komponenty můžeme začít tvořit jak JSX, definici komponenty, nebo znovupoužitelný hook. V tomto případě, při vývoji komponenty rozevíracího seznamu, začneme naprogramováním vlastního hooku, který se odděleně postará o veškerou logiku seznamu.

Hook *useDropdown* bude mít 2 parametry – obslužnou funkci ke změně vybrané v možnosti v rodičovské komponentě (*onChange*) a výchozí hodnotu vybrané možnosti (*defaultValue*). V rámci hooku nadefinujeme stavy *selectedOption*, *isOpen* a vygenerujeme unikátní identifikátor. Dále vytvoříme funkci *handleOptionClick*, která zajistí změnu vybrané možnosti, zavření seznamu a vypublikuje změnu hodnoty do rodičovské komponenty. Z hooku vracíme potřebné stavy a funkce ve formě objektu nebo pole – pole musíme označit jako *const*.

```
// Část souboru useDropdown.ts

function useDropdown(
  onChange: (selectedOption: Option | null) => void,
  defaultValue: Option | null,
) {
  const [selectedOption, setSelectedOption]
```

```
    = useState<Option | null>(defaultValue);
  const [isOpen, setIsOpen] = useState(false);

  // Toto ID je třeba nastavit na kořenový element dropdown komponenty.
  const dropdownId = `id-${crypto.randomUUID()}`;

  // Ubslužná funkce, která se stará o logiku
    po kliknutí na jednotlivé položky v dropdownu.
  const handleOptionClick = (option: Option) => {
    setSelectedOption(option);
    setIsOpen(false);
    onChange(option);
  };

  // Vrátíme všechny hodnoty, které chceme mít dostupné zvenčí.
  return [dropdownId, selectedOption,
    isOpen, setIsOpen, handleOptionClick] as const;
}
```

Pokračujeme tvorbou JSX komponenty Dropdown, kde vložíme tlačítko a seznam možností. Otevření možností zajistíme přidáním *onClick* (což je vlastně *MouseEventHandler*). V anonymní funkci pak změníme stav pomocí *isOpen* na opačnou hodnotu. Abychom předešli event bubblingu, v rámci anynomní obslužné funkce zavoláme *event.stopPropagation()*.

```
// Část souboru Dropdown.tsx

<div className="rounded-md shadow-sm">
  {/* Pro poslouchání na události v DOMu můžeme použít syntaxi:
    NÁZEV_UDÁLOSTI={OBSLUŽNÁ_METODA}. */}
  <button
    type="button"
    className={`STATICKÉ STYLY... ${sizeStyles} ${buttonStyles}`}
    onClick={event => {
      event.stopPropagation();
      setIsOpen(!isOpen);
    }}
  >
    {selectedOption ? selectedOption.label : placeholder}
    {isOpen ? arrowUpIcon : arrowDownIcon}
  </button>
</div>
```

Seznam možností zobrazíme podmíněně na základě stavu *isOpen*. Pro vykreslení možností seznamu (*options*) použijeme JavaScriptovou funkci *map* uvnitř JavaScriptové hodnoty v JSX. V Reactu je důležité vždy při použití funkce *map* nastavit unikátní klíč (*key*) pro každou položku v seznamu. Tento klíč slouží k identifikaci jednotlivých prvků a optimalizaci procesu renderování. Pro vybrání konkrétní možnosti použijeme *onClick*, kterému předáme anonymní funkci. V anonymní funkci

zavoláme funkci *handleOptionClick* hooku *useDropdown* s aktuální položkou ze seznamu.

```
// Část souboru Dropdown.tsx

{isOpen && (
  <div className={`STATICKÉ STYLY... ${divStyles}`}>
    <div
      className="py-1"
      role="menu"
      aria-orientation="vertical"
      aria-labelledby="options-menu"
    >
      {/* Pro vykreslení seznamu (listu) můžeme využít bloky { }
       a JavaScriptovou funkci .map(). */}
      {options.map(option => (
        <button
          key={option.value}
          className={`STATICKÉ STYLY... ${optionStyles}`}
          role="menuitem"
          onClick={() => handleOptionClick(option)}
        >
          {option.label}
        </button>
      ))}
    </div>
  </div>
)}
```

Abychom uzavřeli jakýkoli aktuálně otevřený rozbalovací seznam na stránce, po kliknutí mimo tento seznam, předáme kořenovému elementu dříve vytvořený unikátní identifikátor. Do *useDropdown* přidáme *useEffect* a díky němu budeme naslouchat na události pointerdown v DOM. Obslužná funkce pak zajistí zavření aktuálně otevřeného dropdownu.

```
// Část souboru useDropdown.ts

useEffect(() => {
  // Obslužná funkce handleClickOutsideDropdown zajistí zavření dropdownu,
    pokud uživatel klikne mimo něj.
  const handleClickOutsideDropdown = ({target}: PointerEvent) => {
    if (!(target as HTMLElement).closest(`#${dropdownId}`)) {
      setIsOpen(false);
    }
  };

  // Přidáme posluchač události na událost pointerdown a jeho obslužnou funkci.
  document.addEventListener('pointerdown', handleClickOutsideDropdown);

  // Funkce, která se zavolá při odpojení komponenty.
  return () => {
    // Odebereme posluchač události na událost
```

```
      pointerdown a jeho obslužnou funkci.
    document.removeEventListener('pointerdown', handleClickOutsideDropdown);
  };
  // Druhý parametr useEffectu je pole závislostí,
    které určuje, kdy se má useEffect spustit.
}, [dropdownId]);
```

Dropdown samozřejmě může mít i jiné vstupy, které povedou k lepší znovupoužitelnosti. Dynamické CSS třídy ve formě JavaScriptu na element přidáme pomocí šablonových literálů (template literals) a JavaScriptové hodnoty.

Reaktivita, asynchronní operace

Následující komponenta bude demonstrovat využití reaktivity a asynchronních operací. Vytvoříme komponentu, která přeloží zadaný text do cílového jazyka. Začneme vytvořením komponenty Translator. Komponenta při změně stavů (zadaného textu uživatelem a výstupního jazyka) zavolá API, které vrátí přeložený text. V rámci komponenty vytvoříme vnořené komponenty pro zadání vstupního textu, výběr jazyka a zobrazení výsledku.

Komponenta LanguageDropdown uživateli umožní vybrat jazyk, do kterého chce text přeložit. Díky vlastnosti *onChange* (callback funkci) aktualizujeme výstupní jazyk v rodičovské komponentě.

Pokračujeme implementací komponenty TranslationInput, která bude sloužit k zadání vstupního textu přes textové pole. Aktuální hodnotu formulářového prvku nastavíme pomocí atributu *value*. Po změně hodnoty textového pole, kterou získáme v události přes atribut *onChange*, aktualizujeme hodnotu vstupního textu v Translator komponentě.

```
// Část souboru TranslationInput.tsx

<textarea
  ref={textAreaRef}
  className="block w-full min-h-0 p-3 pr-12 pb-8 resize-none !outline-none"
  placeholder="Type to translate ..."
  value={inputText}
  onChange={handleInputChange}
/>
```

Abychom reaktivně aktualizovali výšku pole na základě obsahu, použijeme vlastní hook. Hook bude potřebovat referenci elementu, a tak vytvoříme ref, který přidáme na element textového pole.

```
// Část souboru TranslationInput.tsx

function TranslationInput({inputText, setInputText}: TranslationInputProps) {
  // Vytvoření reference pro textovou oblast.
  const textAreaRef = useRef<HTMLTextAreaElement>(null);

  // Použití vlastního hooku pro automatickou aktualizaci výšky
    textové oblasti na základě reference.
  useAutosizeTextArea(textAreaRef.current, inputText);

  return (
    {/* JSX... */}
  );
}
```

Hook *useAutosizeTextArea* bude přijímat referenci na element. Dále také hodnotu textového pole, aby po jakékoli změně této hodnoty přepočítala výška pole. V rámci hooku vytvoříme *useEffect*, který se znovu zavolá při každé změně *textAreaRef*, nebo hodnoty textu. Následně v rámci těla hooku aktualizujeme výšku textového pole.

```
// Část souboru useAutoresizeTextarea.ts

const useAutosizeTextArea = (
  textAreaRef: HTMLTextAreaElement | null, value: string
) => {
  useEffect(() => {
    if (textAreaRef) {
      // Abychom získali správnou výšku scrollHeight
        pro textovou oblast, musíme výšku resetovat.
      textAreaRef.style.height = '0px';

      // Výšku pak nastavíme přímo na nativní prvek.
      // Při pokusu o nastavení této hodnoty pomocí stavu
        nebo odkazu bude výsledkem nesprávná hodnota.
      textAreaRef.style.height = `${textAreaRef.scrollHeight + 36}px`;
    }
  }, [textAreaRef, value]);
};
```

V Translator komponentě potřebujeme ukládat vstupní hodnotu a výstupní jazyk z vnořených komponent. Dále při každé změně těchto hodnot zavoláme API, k čemuž využijeme *useEffect*. V rámci hooku definujeme asynchronní funkci *handleTranslation*, která pomocí fetch API odešle korektní HTTP POST požadavek na server. Pokud bychom definovali funkci mimo *useEffect*, museli bychom ji přidat do pole závislostí hooku. Při úspěšné odpovědi aktualizujeme stav s přeloženým textem, v opačném případě nastavíme chybový stav.

```
// Část souboru Translator.tsx

useEffect(() => {
  // Funkce pro zpracování přeložení textu.
  const handleTranslation = async () => {
    if (!inputText.length) return;

    // Zrušení předchozího asynchronního požadavku.
    abortControllerRef.current?.abort();
    // Vytvoření nového kontroleru pro zrušení asynchronního požadavku.
    abortControllerRef.current = new AbortController();
    setLoading(true);

    const parsedInputText = inputText.replace(/
n/g, '');
    const url = `${import.meta.env.VITE_RAPID_API_BASE_URL}${outputLanguage}${
      import.meta.env.VITE_RAPID_API_QUERY_PARAMS
    }`;
    const options = {
      method: 'POST',
      headers: {
        'content-type': 'application/json',
        'X-RapidAPI-Key': import.meta.env.VITE_RAPID_API_KEY,
        'X-RapidAPI-Host': 'microsoft-translator-text.p.rapidapi.com',
      },
      body: `[{"Text":"${parsedInputText}"}]`,
      signal: abortControllerRef.current?.signal,
    };

    try {
      // Odeslání HTTP POST požadavku na server,
        který nám vrátí přeložený text v nějaké struktuře.
      const response = await fetch(url, options);

      if (!response.ok) {
        throw new Error(
          `Something went wrong: ${response.status} Error.
          Please reload the page.`,
        );
      }

      const result = await response.json();
      const translatedText = result[0].translations[0].text as string;
      setOutputText(translatedText);
      // eslint-disable-next-line @typescript-eslint/no-explicit-any
    } catch (error: any) {
      // Pokud je chyba typu AbortError, tak ji ignorujeme.
      if (error.name === 'AbortError') return;

      setError(error);
    } finally {
      setLoading(false);
    }
  };

  // Zrušení předchozího časovače.
  clearTimeout(delayTimerRef.current);
```

```
  // Zpoždění překladu o 300 ms.
  delayTimerRef.current = setTimeout(() => handleTranslation(), 300);

  // Zrušení časovače při zničení komponenty.
  return () => clearTimeout(delayTimerRef.current);
}, [inputText, outputLanguage]);
```

Aby dotazování fungovalo, vytvoříme referenci *delayTimerRef*. V rámci těla *useEffect* hooku nejprve zrušíme předchozí časovač. Funkci *handleTranslation* zavoláme v callbacku funkce *setTimeout*, která umožní předejít dotazování serveru ihned po změně nějaké vstupní hodnoty. Výsledek funkce *setTimeout* uložíme do *delayTimerRef.current*. Nesmíme také zapomenout na zrušení časovače při zničení komponenty.

V okamžiku, kdy obdržíme odpověď ze serveru, zobrazíme přeložený text uživateli pomocí komponenty TranslationOutput. Předáme jí samotný výstupní text a další vstupní vlastnosti, na základě kterých pak podmíněně vykreslíme přeložený text, chybu nebo načítání.

Tvorba formulářů, validace

React sám o sobě poskytuje jen základní API pro správu formulářů. Disponuje však mnoha knihovnami, které tuto funkcionalitu rozšiřují. Mezi takové knihovny patří např. Formik, Redux Form nebo React Hook Form. V této sekci se zaměříme na tvorbu formulářů pomocí React Hook Form [33]. Vytvoříme komponentu pro jednoduchou investiční kalkulaci. V rámci této komponenty naprogramujeme formulář pro zadání vstupních dat a komponentu výsledku kalkulace, která se zobrazí po potvrzení formuláře.

Začneme s reaktivním fomulářem, který bude přijímat počáteční hodnoty (*defaultValues*) a callback funkci *handleFormSubmit* pro předání hodnot formuláře do rodičovské komponenty. Strukturu formuláře popíšeme v typu *InvestFormData*. Pomocí hooku *useForm* z knihovny React Hook Form vytvoříme instanci formuláře, které předáme *defaultValues* a nastavíme reaktivní validaci. Následně z hooku dostameme funkce *register*, *handleSubmit* a *formState*, které poslouží ke správě formuláře.

```
// Část souboru types.ts

export interface InvestFormData {
```

```
  oneOffInvestment: number;
  investmentLength: number;
  averageSavingsInterest: number;
  averageSP500Interest: number;
}

// Část souboru InvestForm.tsx

const {
  register,
  handleSubmit,
  formState: {errors},
} = useForm<InvestFormData>({defaultValues, mode: 'onChange'});
```

Následně do JSX přidáme form s *onSubmit* atributem, kterému předáme funkci *handleSubmit* z React Hook Form. Do *handleSubmit* pak vložíme vstup *handleFormSubmit* v rámci nějž získáme aktuální hodnoty formuláře. Ve formuláři vytvoříme formulářové prvky, které propojíme s reaktivním formulářem pomocí funkce *register*. První argument představuje název formulářového prvku, druhý argument je validační objekt. V rámci range inputu potřebujeme HTML atributy min a max, díky kterým omezíme rozsah vstupních hodnot. Abychom mohli měli přístup k aktuální hodnotě range inputu, využijeme vlastnost *value* a *onChange*. Chyby formuláře získáme z *formState* a vykreslíme je pod formulářovými prvky. V neposlední řadě přidáme tlačítko s typem submit, které zajistí odeslání formuláře a zavolání callback funkce *handleFormSubmit*.

```
// Část souboru InvestForm.tsx

return (
  <form onSubmit=handleSubmit(handleFormSubmit)>
    <div className="md:flex md:gap-4">
      <div className="mb-4 md:w-1/2">
        <InputLabel id="oneOffInvestment">
          One-off investment (20-99.999.999€)
        </InputLabel>

        <input
          id="oneOffInvestment"
          type="number"
          // Vytvoření prvku ve formuláři, přidání validátorů
          // a jiného nastavení formulářového prvku.
          ...register('oneOffInvestment',
            required: true,
            valueAsNumber: true,
            min: 20,
            max: 99_999_999,
          )
          className="STATICKÉ STYLY..."
```

```
        />
        errors.oneOffInvestment?.type === 'required' && (
          <p className="text-red-500 text-xs italic mt-1">
            Please enter a valid amount of one-off investment (positive number).
          </p>
        )
        /* Další chybové hlášky... */
      </div>
    </div>

    /* Další formulářové prvky... */

    <button type="submit" className="STATICKÉ STYLY...">
      Calculate
    </button>
  </form>
);
```

V rodičovské komponentě získáme aktuální hodnoty formuláře díky obslužné funkci *handleFormSubmit*. Pomocí funkce *futureValuesCalculator* získáme hodnoty, které následně vykreslíme v komponentě FutureValuesInfo. Tato komponenta obsahuje dvě vnořené komponenty FutureValueInfo, pro zobrazení jednotlivých výsledků. Hodnotu v JSX transformujeme pomocí JavaScriptové funkce.

```
// Část souboru FutureValueInfo.tsx

const getLocalizedFutureValue = (value: number): string =>
  `$value.toLocaleString('de-DE')€`;

function FutureValueInfo({children, futureValue}: FutureValueInfoProps) {
  return (
    <div className="p-1 sm:w-1/2">
      <p className="text-xl font-semibold mb-2 text-gray-800">{children}</p>
      <p className="text-5xl font-bold">
        {getLocalizedFutureValue(futureValue)}
      </p>
    </div>
  );
}
```

Modularita, použití knihoven

V následující sekci vytvoříme webovou hru, ve které je cílem uživatele uhádnout název státu na základě poskytnutých nápovědí. Práci si ulehčíme pomocí externích knihoven a služeb. Postupně se bude odkrývat 8 nápovědí, které by měly pomoci s uhádnutím daného státu. Klíčovým prvkem je textové pole, přes které uživatel zadává názvy hádaných zemí a tlačítko pro potvrzení. Součástí také bude seznam zemí, které uživatel hádal a modální okna sloužící k vyhodnocení hry.

Začneme s implementací rodičovské komponenty, která získá země z veřejného API. Naprogramujeme hook *useAllCountries*, který bude vracet data (*countries*), chybu a stav načítání. V rámci *useEffect* zavoláme asynchronní funkci *fetchCountriesData*. Uvnitř funkce *fetchCountriesData* zavoláme funkci *getAllCountries*. Ve funkci *getAllCountries* využijeme knihovnu *axios* [51] a převzatou funkci *requestHandler* [11]. Balíček *axios* slouží jako HTTP klient pro tvorbu asynchronních dotazů, zatímco *requestHandler* umožňuje otypování příchozí odpovědi. Po ošetření chyb aktualizujeme patřičné stavy, které následně z hooku vrátíme.

```
// Část souboru useAllCountries.ts

useEffect(() => {
  const getSortedCountriesByName = (countries: Countries): Countries => {
    return countries.toSorted((a, b) =>
      a.name.common.localeCompare(b.name.common));
  };

  const fetchCountriesData = async () => {
    // Tělo asynchronní funkce fetchCountriesData.
  };

  fetchCountriesData();
}, []);

// Část souboru getAllCountries.ts

export const getAllCountries =
  requestHandler<CountriesRequestOptions, Countries>(params =>
    axios.request(getRequestConfig(params)),
  );
```

Opakování asynchronních dotazů při chybě zajistíme pomocí knihovny *axios-retry* [28]. Opakování nakonfigujeme v souboru *main.tsx*. Abychom nemuseli implementovat načítací a chybové stavy, anebo rušení či opakování asynchronních dotazů, můžeme použít knihovnu *react-query*.

```
// Část souboru main.tsx

import axiosRetry, exponentialDelay, isNetworkError, isRetryableError
from 'axios-retry';

axiosRetry(axios, {
  retries: 3,
  retryDelay: (...arg) => exponentialDelay(...arg, 500),
  retryCondition(error) {
    return isNetworkError(error) || isRetryableError(error);
  },
});
```

Následně v rámci rodičovské komponenty podmíněně vykreslíme dané komponenty. V případě chyby komponentu ErrorAlert. Pokud ze serveru úspěšně dostaneme země, tak vykreslíme komponentu CountryGuesser. Pokud nevykreslíme ani jednu z předchozích komponent, zobrazíme LoadingSkeleton.

```
// Část souboru CountryGuesserWrapper.tsx

function CountryGuesserWrapper() {
  const [countries, error] = useAllCountries();

  if (error) {
    return (
      <WrapperDiv><ErrorAlert message={error.message} /></WrapperDiv>
    );
  }

  if (countries.length > 0) {
    return <CountryGuesser countries={countries} />;
  }

  return (
    <WrapperDiv><LoadingSkeleton /></WrapperDiv>
  );
}
```

Komponenta CountryGuesser bude vyhodnocovat průběh hry a zobrazovat jednotlivé herní prvky. Začneme definicí stavů a inicializujeme náhodnou zemi (*randomCountry*), kterou bude uživatel hádat. Dále použijeme hook *useCountryFlagPolyfill*, který při namontování komponenty zajistí podporu zobrazení ikon vlajek v prohlížečích, které to přímo nepodporují. Prohlížeč pak však musí podporovat emojis a webové fonty. Pokračujeme implementací obslužných funkcí *handleEvaluateGuessAndUpdateState* a *handleSetInitialState*, které budou sloužit k aktualizaci stavu hry. V rámci JSX pak vykreslíme jednotlivé herní prvky a modální okna při výhře či prohře.

Hook *useCountryFlagPolyfill* zavolá funkci *polyfillCountryFlagEmojis*, která do HTML hlavičky přidá webový font Twemoji Country Flags. Aby se font využil, přidáme jej do CSS stylů.

```
// Část souboru index.css

@layer base {
  html {
    font-family: 'Twemoji Country Flags', 'ALTERNATIVNÍ_FONTY...';
  }
}
```

Úkolem komponenty HintBoxes bude postupné vykreslení nápověd. Na základě vstupu *randomCountry* vytvoříme pole nápověd. V JSX pak iterujeme přes pole nápověd a vykreslíme jednotlivé nápovědy. Jednotlivé komponenty HintBox pak dynamicky vykreslí název a SVG ikonu nápovědy, textovou nápovědu, případně obrázek vlajky státu.

Klíčová komponenta CountryGuessInput, kterou definujeme v rámci souboru *CountryGuessInput.tsx*, umožní uživateli zadat svůj tip. Začneme s JSX, kde vytvoříme formulářový prvek pro zadání názvu země, potvrzovací tlačítko a podmenu textového pole, které zobrazí nejpodobnější země na základě zadaného textu (filtrované země). Přidáme obslužné funkce pro akce a události nad formulářem, které následně doimplementujeme.

V komponentě, na základě vstupu *countries*, získáme pole všech zemí bez těch, které uživatel již hádal (*countriesWithoutAlreadyGuessed*). Poté definujeme a inicializujeme ostatní stavy. Při kliknutí na tlačíko se zavolá funkce *handleGuessButtonClick*, která volá obslužnou funkci *handleEvaluateGuessAndUpdateState* v rodičovské komponentě a také funkci *handleChangeSelectedGuess*. Funkce *handleChangeSelectedGuess* aktualizuje aktuální tip, filtrované země a uzavře podmenu. Funkce *handleInputChange* převede tip uživatele do daného formátu, poté aktualizuje aktuální tip a filtrované země. Ovládání formulářového prvku pomocí klávesnice umožní funkce *handleKeyDown*.

Pomocná funkce *updateGuessAndFilteredCountries* získá aktuálně filtrované země na základě uživatelova tipu. Následně aktualizuje stavy *currentGuess*, *isValidGuess* a *filteredCountries*. Funkce *clampSelectedGuessIndex* zajistí, aby index uživatelem vybrané země byl v požadovaném rozmezí (0 až počet filtrovaných zemí). Pro aktualizaci vlastnosti *selectedGuessIndex* slouží funkce *changeSelectedGuessIndex*, která index aktualizuje o hodnotu předanou v argumentu. V neposlední řadě funkce *convertToFormattedGuess* převede tip uživatele tak, aby začínal velkým písmenem a zbytek řetezce byl složen z malých písmen.

Ke zobrazení všech již hádaných zemí uživatelem vytvoříme komponentu GuessedCountriesList. Ze vstupních vlastností *countries*, *guessedCountries* a *randomCountry* získáme proměnnou *enrichedGuessedCountries*. Jde o uživatelem hádané země s vlajkou a vzdáleností od *randomCountry*. K převodu využijeme JavaScrip-

tové funkce z jiného souboru. K vypočtení vzdálenosti použijeme knihovnu *calculate-distance-between-coordinates* [26], která obsahuje funkci *getDistanceBetweenTwoPoints*. Jednotlivé prvky pole *enrichedGuessedCountries* pak vykreslíme v rámci JSX.

Nakonec vytvoříme modální okna, která se zobrazí při výhře či prohře. Stavy *isWinModalOpen* a *isLoseModalOpen* aktualizujeme v rámci funkce *handleEvaluateGuessAndUpdateState* v CountryGuesser. Na základě těchto stavů pak podmíněně vykreslíme daná modální okna. Oběma modálům předáme *randomCountry* a obslužnou funkci *handleClose*. Výhernímu modálu také počet potřebných pokusů. V jednotlivých komponentách (WinModal, LoseModal) vykreslíme komponentu BaseModal, která bude sloužit jako šablona pro obě okna. Do této komponenty vždy předáme titulek, obsah a obslužnou funkci *handleClose*. BaseModal následně v JSX vykreslí základní strukturu modálního okna, s dynamickými možnostmi pro titulek, obsah a obslužnou funkci *handleClose*.

Routování a layout aplikace

Layout aplikace bude rozdělen do tří částí: hlavičky, patičky a samotného obsahu, v nemž se vykreslí jednotlivé komponenty. Uživatel bude mít možnost přepínání mezi jednotlivými stránkami pomocí navigačního menu.

Pro routování využijeme knihovnu *react-router-dom* [34]. Začneme vytvořením souboru s cestami (*appRoutes*).

```
// Část souboru appRoutes.ts

interface AppRoute {
  name: string;
  path: string;
  component: ComponentType;
  index?: boolean;
}

export const appRoutes: ReadonlyArray<AppRoute> = [
  {
    name: 'Home',
    path: '/',
    component: Landing,
    index: true,
  },
  {
    name: 'Counter',
    path: '/counter',
```

```
    component: Counter,
  },
  // Další cesty...
];
```

Následně v kořeni aplikace vytvoříme *router* pomocí předem definovaných cest aplikace. *Router* vytvoříme díky dvěma pomocným funkcím k tomu určených: *createBrowserRouter* a *createRoutesFromElements*. Pokračujeme přiřazením proměnné *router* do kořenové komponenty aplikace, konkrétně do poskytovatele *RouterProvider*.

```
// Část souboru main.tsx

const router = createBrowserRouter(
  createRoutesFromElements(
    <Route path="/" element={<AppLayout />} errorElement={<ErrorPage />}>
      {appRoutes.map(route => (
        <Route
          key={route.name}
          index={route.index}
          path={route.path}
          Component={route.component}
          caseSensitive
        />
      ))}
    </Route>,
  ),
);

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <RouterProvider router={router} />
  </StrictMode>,
);
```

Hlavní komponenta AppLayout pak v JSX vykreslí hlavičku, patičku a dynamický obsah dle aktuální URL adresy, jenž vykreslí komponenta *Outlet*.

```
// Část souboru AppLayout.tsx

function AppLayout() {
  return (
    <div className="min-h-screen flex flex-col">
      <Header />

      <main className="flex-grow p-8">
        {/* Outlet vykresluje šablonu (komponentu) pro aktuální URL adresu. */}
        <Outlet />
      </main>

      <Footer />
    </div>
```

```
  );
}
```

V hlavičce aplikace se budou nacházet odkazy na jednotlivé stránky. My se inspirujeme architekturou a vzhledem navigačního menu Flowbite. V rámci komponenty Header vypíšeme všechny cesty aplikace pomocí komponenty *NavLink* z knihovny *react-router-dom*. NavLink umožňuje v rámci atributu *className* přistoupit k vlastnosti *isActive*, která indikuje, zda je cesta odkazu aktivní. Vlastnosti *isActive* tedy využijeme pro podmíněné přidání CSS stylů. Pro korektní nastavení aria-current atributu, v závislosti na aktuální URL, použijeme hook *useLocation*, který vrací aktuální URL.

```
// Část souboru Header.tsx

{appRoutes.map(route => (
  <li key={route.name}>
    <NavLink
      to={route.path}
      // react-router-dom poskytuje vlastnost "isActive"
        pro zvýraznění aktivního odkazu.
      className={({isActive}) =>
        'block py-2 pr-4' +
        `${
          isActive
            ? ' STATICKÉ STYLY PRO AKTIVNÍ ODKAZ...'
            : ' STATICKÉ STYLY PRO NEAKTIVNÍ ODKAZ...'
        }`
      }
      aria-current={route.path === currentPathName ? 'page' : undefined}
    >
      {route.name}
    </NavLink>
  </li>
))}
```

Mobilní navigaci a barevný režim implementujeme díky stavům *isMobileNavOpen* a *isDarkMode*. Informaci o tom, zda má uživatel zapnutý tmavý režim budeme ukládat do LocalStorage v prohlížeči. K tomu použijeme *useEffect* hook, který při změně stavu *isDarkMode* přidá patřičný data-mode a provede aktualizaci LocalStorage.

```
// Část souboru Header.tsx

useEffect(() => {
  if (isDarkMode) {
    document.documentElement.setAttribute('data-mode', 'dark');
    localStorage.setItem('data-mode', 'dark');
```

```
  } else {
    document.documentElement.removeAttribute('data-mode');
    localStorage.removeItem('data-mode');
  }
}, [isDarkMode]);
```

3.2.3 Svelte

Správa stavů, předávání vlastností

Prvním krokem k vytvoření jednoduchého čítače bude definice komponenty Counter s reaktivním stavem *count*.

```
// Část souboru Counter.svelte

<script lang="ts">
  let count = 0;
</script>
```

Dále vytvoříme komponentu Button z důvodu dodržování principu DRY a efektivnějšímu znovupoužití kódu v budoucnu. Komponenta bude přijímat vlastnosti *className* a *onClick*. *ClassName* rozšíří CSS třídy tlačítka a *onClick* bude obsahovat obslužnou funkci, která se zavolá při kliknutí na tlačítko. Nyní do šablony přidáme tlačítko a předáme mu vlastnosti *className* a *onClick*. Svelte umožňuje zachytit všechny nedefinované vlastnosti do proměnné *$$restProps*. Proměnnou *$$restProps* tedy pomocí spread operátoru předáme tlačítku a tím jej obohatíme o další vlastnosti. Obsah tlačítka, který definujeme mezi párovými značkami Button, vykreslíme pomocí komponenty slot.

```
// Soubor Button.svelte

<script lang="ts">
  export let className: string;
  export let onClick: () => void;
</script>

<!- Proměnná $$restProps obsahuje ostatní vlastnosti,
  které nejsou v komponentě přijímány pomocí klíčového slova "export". ->
<button
  type="button"
  class="px-4 py-2 rounded-md focus:outline-none {className}"
  on:click={onClick}
  {...$$restProps}
>
  <!- slot slouží k vykreslení obsahu,
    který vložíme mezi párové tagy dané komponenty. ->
  <slot />
</button>
```

V Counter komponentě pak v rámci šablony vykreslíme stav *count* a Button komponenty, kterým předáme příslušné vlastnosti. Pro aktualizaci stavu *count* použijeme obslužné funkce, v nichž přímo tento stav modifikujeme.

```
// Část souboru Counter.svelte

<script lang="ts">
  import Button from '../components/button/Button.svelte';

  let count = 0;

  const increment = () => (count += 1);
  const decrement = () => (count -= 1);
  const reset = () => (count = 0);
</script>

<div class="bg-gray-200 p-6 rounded-md shadow-md">
  <p class="text-xl font-semibold mb-4">Current count: {count}</p>

  <div class="flex gap-4">
    <Button
      className="bg-blue-500 text-white hover:bg-blue-600"
      onClick={increment}
    >
      Increment
    </Button>

    <!- Další komponenty Button... ->
  </div>
</div>
```

Interakce v uživatelském prostředí

V této sekci implementujeme rozbalovací seznam s možnostmi (dropdown). Tvorbu UI komponenty můžeme začít jak vytvořením HTML struktury, tak definicí funkční stránky komponenty.

My začneme tvorbou šablony, v níž vytvoříme tlačítko a seznam možností. Otevření seznamu možností zajistíme přidáním *on:click* události na tlačítko. V obslužné funkci pak změníme stav *isOpen*.

```
// Část souboru Dropdown.svelte

<div class="rounded-md shadow-sm">
  <!- Pro poslouchání na události v DOMu můžeme použít syntaxi:
    on:NÁZEV_UDÁLOSTI={OBSLUŽNÁ_METODA}. ->
  <button
    type="button"
    class={`STATICKÉ STYLY... ${sizeStyles} ${buttonStyles}`}
    on:click|stopPropagation={() => (isOpen = !isOpen)}
  >
```

```
    {selectedOption ? selectedOption.label : placeholder}
    <!- Pro podmíněné vykreslovaní můžeme využít bloky
      #if, :else if, :else a /if. ->
    {#if isOpen}
      <ArrowUpIcon />
    {:else}
      <ArrowDownIcon />
    {/if}
  </button>
</div>
```

Seznam možností zobrazíme podmíněně na základě *isOpen*. Pro vykreslení možností seznamu (dle vstupu *options*) použijeme blok *#each*. Pro vybrání konkrétní možnosti použijeme *on:click* událost, při které v anonymní funkci zavoláme funkci *handleOptionClick* s aktuální položkou ze seznamu.

```
// Část souboru Dropdown.svelte

{#if isOpen}
  <div class={‘STATICKÉ STYLY... ${divStyles}‘}>
    <div
      class="py-1" role="menu"
      aria-orientation="vertical" aria-labelledby="options-menu"
    >
      <!- Pro vykreslení listu (pole hodnot) můžeme využít blok #each. ->
      {#each options as option}
        <button
          class={‘STATICKÉ STYLY... ${optionStyles}‘}
          role="menuitem"
          on:click={() => handleOptionClick(option)}
        >
          {option.label}
        </button>
      {/each}
    </div>
  </div>
{/if}
```

Dropdown komponenta bude přijímat vlastnosti *options* a *onChange*, případně další vlastnosti pro znovupoužitelnost. Pro každou komponentu také vytvoříme jednoznačný identifikátor, který využijeme při uzavírání seznamu. Obslužná funkce *handleOptionClick* zajistí změnu vybrané možnosti, zavření seznamu a provede změnu hodnoty v rodičovské komponentě.

```
// Část souboru Dropdown.svelte

<script lang="ts">
  export let options: ReadonlyArray<Option>;
  export let onChange: (selectedOption: Option | null) => void;
  export let defaultValue: Option | null = null;
```

```
  let selectedOption: Option | null = defaultValue;
  let isOpen = false;

  // Toto ID je třeba nastavit na kořenový element dropdown komponenty.
  let dropdownId = `id-${crypto.randomUUID()}`;

  // Obslužná funkce, která se stará o logiku
    po kliknutí na jednotlivé položky v dropdownu.
  const handleOptionClick = (option: Option) => {
    selectedOption = option;
    isOpen = false;
    onChange(option);
  };
</script>
```

K uzavření jakéhokoli otevřeného seznamu, při kliknutí mimo tento seznam, vytvoříme akci (Svelte action) *clickOutsideDropdown*. V rámci akce *clickOutsideDropdown* budeme naslouchat na události pointerdown v DOM. Obslužná funkce pak zajistí spuštění callbacku v Dropdown komponentě.

```
// Soubor clickOutsideDropdown.ts

export const clickOutsideDropdown = (
  node: HTMLDivElement,
  callback: (event: PointerEvent) => void
) => {
  const handlePointerDown = (event: PointerEvent) => callback(event);

  document.addEventListener('pointerdown', handlePointerDown);

  return {
    destroy() {
      document.removeEventListener('pointerdown', handlePointerDown);
    },
  };
};
```

Na kořenový element komponenty přidáme dříve vytvořený unikátní identifikátor a akci pomocí direktivy *use*. Akci následně předáme obslužnou funkci *handleClickOutsideDropdown*, která zavře aktuálně otevřený dropdown.

```
// Část souboru Dropdown.svelte

<script lang="ts">
  // Ostatní stavy, vstupy, funkce v komponentě...

  // Obslužná funkce, která zavře dropdown, pokud uživatel klikne mimo něj.
  const handleClickOutsideDropdown = ({target}: PointerEvent) => {
    if (isOpen && !(target as HTMLElement).closest(`#${dropdownId}`)) {
      isOpen = false;
    }
```

```
  };
</script>

<div
  class="relative inline-block text-left"
  id={dropdownId}
  use:clickOutsideDropdown={handleClickOutsideDropdown}
>
  <!- Vnořené elementy... ->
</div>
```

Třídy CSS v JavaScriptové formě přidáme k elementu pomocí šablonových literálů a JavaScriptové hodnoty.

Reaktivita, asynchronní operace

V rámci této sekce se zaměříme na reaktivitu a asynchronní operace. Naprogramujeme komponentu, která přeloží zadaný text do cílového jazyka. Začneme vytvořením komponenty Translator. Komponenta reaktivně (při změně zadaného textu či výstupního jazyka) zavolá API, které vrátí přeložený text. V Translator komponentě využijeme vnořené komponenty, které budou sloužit k zadání vstupního textu, výběru jazyka a zobrazení výsledku.

Skrze komponentu LanguageDropdown umožníme uživateli vybrat jazyk, do kterého bude chtít text přeložit. Výstupní jazyk v rodičovské komponentě změníme přes vlastnost *onChange*.

Pokračujeme implementací komponenty TranslationInput, která umožní zadat vstupní text (*inputText*) přes textové pole. Aktuální hodnotu formulářového prvku nastavíme pomocí *bind:value*. V rámci rodičovské komponenty použijeme *bind*, díky čemuž pak reaktivně aktualizujeme *inputText* v Translator komponentě.

```
// Část souboru TranslationInput.svelte

<textarea
  bind:value={inputText}
  use:autoresizeTextArea
  class="block w-full min-h-0 p-3 pr-12 pb-8 resize-none !outline-none"
  placeholder="Type to translate ..."
/>

// Část souboru Translator.svelte

<script lang="ts">
  let inputText = '';
</script>

<TranslationInput bind:inputText />
```

K reaktivní změně výšky textového pole použijeme akci *autoresizeTextArea*. Akce přijme element, na kterém se má provést změna výšky. Elementu přidáme listener na událost input. V obslužné funkci následně modifikujeme výšku pole.

```
// Soubor autoresizeTextArea.ts

export const autoresizeTextArea = (element: HTMLTextAreaElement) => {
  element.addEventListener('input', () => resizeTextArea(element));

  return {
    destroy() {
      element.removeEventListener('input', () => resizeTextArea(element));
    },
  };
};

const resizeTextArea = (element: HTMLTextAreaElement) => {
  // Abychom získali správnou výšku scrollHeight
    pro textovou oblast, musíme výšku resetovat.
  element.style.height = '0px';
  // Výšku pak nastavíme přímo na nativní prvek.
  element.style.height = `${element.scrollHeight + 36}px`;
};
```

V rámci rodičovské komponenty zavoláme API v momentě, kdy dojde ke změně vstupního textu nebo výstupního jazyka. K tomu použijeme reaktivní prohlášení (reactive statement). V těle prohlášení zrušíme předchozí časovač a pomocí funkce *setTimeout* zavoláme funkci *handleTranslation*. Tímto způsobem předejdeme dotazování serveru ihned po změně nějaké vstupní hodnoty. Při zničení komponenty zrušíme časovač a asynchronní požadavky.

```
// Část souboru Translator.svelte

<script lang="ts">
  // Ostatní stavy, vstupy, funkce v komponentě...

  $: if (inputText.length && outputLanguage) {
    // Zrušení předchozího časovače.
    clearTimeout(delayTimer);

    // Zpoždění překladu o 300 ms.
    delayTimer = setTimeout(() => handleTranslation(), 300);
  }

  onDestroy(() => {
    // Zrušení asynchronního požadavku a časovače při zničení komponenty.
    clearTimeout(delayTimer);
    abortController?.abort();
  });
</script>
```

Účelem asynchronní funkce *handleTranslation* pak je odeslání korektního HTTP POST požadavku na server pomocí fetch API. Při úspěšné odpovědi aktualizujeme stav s přeloženým textem, v opačném případě nastavíme chybový stav.

Při obdržení odpovědí ze serveru vykreslíme přeložený text uživateli pomocí komponenty TranslationOutput. Komponentě předáme výstupní text spolu s dalšími vlastnostmi, na základě kterých v šabloně podmíněně vykreslíme přeložený text, chybu nebo načítání.

Tvorba formulářů, validace

Svelte, stejně jako React, nepodporuje pokročilou správu formulářů. Můžeme však využít knihovny třetích stran, jako např. svelte-forms-lib nebo Superforms, které nám umožní lépe spravovat a validovat formuláře. V rámci této sekce vytvoříme komponentu pro jednoduchou investiční kalkulaci s využitím knihovny *svelte-forms-lib* [50]. Komponenta InvestForm bude obsahovat formulář pro zadání vstupních hodnot a komponentu FutureValuesInfo pro zobrazení výsledků kalkulace.

Začneme implementací reaktivního formuláře, který bude přijímat počáteční hodnoty (*defaultValues*) a *investFormData* k předávání hodnot formuláře do rodičovské komponenty. Strukturu formuláře popíšeme v typu *InvestFormData*. Pomocí funkce *createForm* z knihovny *svelte-forms-lib* vytvoříme instanci formuláře, které předáme *defaultValues* do vlastnosti *initialValues*. V rámci nastavení formuláře také definujeme validační schéma pomocí knihovny *yup* [30] a *onSubmit* obslužnou funkci. Knihovnu yup volíme, jelikož je jedinou kompatibilní možností s knihovnou *svelte-forms-lib* ve verzi 2.0.1.

```
// Část souboru InvestForm.svelte

<script lang="ts">
  const validationSchema = object().shape({
    oneOffInvestment: number().min(20).max(99_999_999).required(),
    investmentLength: number().min(3).max(60).required(),
    averageSavingsInterest: number().min(0).max(10).required(),
    averageSP500Interest: number().required(),
  });
</script>
```

Aby nastavená výstupní data (*investFormData*) odpovídala typu *InvestFormData*, musíme transformovat hodnoty formuláře pomocí funkce *cast* na validačním

schématu. V opačném případě budou hodnoty typu string. Z *createForm* následně získáme obslužné funkce *handleChange* a *handleSubmit*, dále stavy formuláře *form*, *errors* a *isValid*. Ke stavům formuláře přistupujeme pomocí $, protože jde o stores observables.

```
// Část souboru InvestForm.svelte

<script lang="ts">
  const {form, errors, isValid, handleChange, handleSubmit} = createForm({
    initialValues: defaultValues,
    validationSchema,
    onSubmit: values => {
      // Převede hodnoty formuláře na typ InvestFormData.
      investFormData = validationSchema.cast(values);
    },
  });
</script>
```

Do šablony přidáme form s *on:submit* událostí, které předáme *handleSubmit*. Pokračujeme vytvořením formulářových prvků. Jednotlivé prvky propojíme s reaktivním formulářem pomocí *bind:value* a události *on:change*, do které přiřadíme funkci *handleChange*. Chyby formuláře získáme z *errors* a vykreslíme je pod formulářovými prvky. V neposlední řadě přidáme tlačítko s typem submit, které se postará o odeslání formuláře a zavolání obslužné funkce *onSubmit*.

```
// Část souboru InvestForm.svelte

<form on:submit={handleSubmit}>
  <div class="md:flex md:gap-4">
    <div class="mb-4 md:w-1/2">
      <InputLabel id="oneOffInvestment">
        One-off investment (20-99.999.999€)
      </InputLabel>

      <!- Propojení formulářového prvku se stavem formuláře
        pomocí bind:value={$form.NÁZEV_POLE}. ->
      <!- Propagace změn do stavu formuláře
        zajišťuje on:change={handleChange}. ->
      <input
        id="oneOffInvestment"
        type="number"
        on:change={handleChange}
        bind:value={$form.oneOffInvestment}
        class="STATICKÉ STYLY..."
      />

      {#if $errors.oneOffInvestment}
        <p class="text-red-500 text-xs italic mt-1">
          Please enter a valid amount of one-off investment (positive number).
        </p>
```

```
      {/if}
    </div>
  </div>

  <!- Další formulářové prvky... ->

  <button
    type="submit"
    disabled={!$isValid}
    class="STATICKÉ STYLY..."
  >
    Calculate
  </button>
</form>
```

Pokračujeme tím, že v rodičovské komponentě pomocí *bind* získáme aktuální hodnoty formuláře (*investFormData*). Po změně hodnot formuláře hodnoty transformujeme pomocí funkce *futureValuesCalculator*. Výsledek (*futureValues*) pak zobrazíme v komponentě FutureValuesInfo. Jednotlivé výsledky budou zobrazeny v rámci vnořených komponent FutureValueInfo. K modifikaci vstupní hodnoty v komponentě FutureValueInfo použijeme reactive statement.

```
// Soubor FutureValueInfo.svelte

<script lang="ts">
  export let futureValue: number;

  $: localizedFutureValue = `${futureValue.toLocaleString('de-DE')}€`;
</script>

<div class="p-1 sm:w-1/2">
  <p class="text-xl font-semibold mb-2 text-gray-800"><slot /></p>
  <p class="text-5xl font-bold">{localizedFutureValue}</p>
</div>
```

Modularita, použití knihoven

Nyní vytvoříme webovou hru, ve které bude úkolem uživatele uhádnout název státu na základě poskytnutých nápovědí. Práci si zlehčíme využitím externích knihoven. V rámci hry se postupně odkryje 8 nápovědí, které uživateli pomohou uhádnout název daného státu. Mezi klíčovými prvky bude textové pole pro zadání názvu země a potvrzovací tlačítko. Dále ve hře bude seznam zemí, které uživatel hádal a modální okna pro vyhodnocení hry.

Začneme s programováním rodičovské komponenty, jejíž úkolem bude získat země z veřejného API. K tomu využijeme balíčky *axios* [28] a *@tanstack/svelte-query* [22]. Knihovna *@tanstack/svelte-query* umožní snadnou správu asynchronních

operací. Její API poskytuje např. podporu načítacích i chybových stavů, také rušení či opakování dotazů a mnoho dalších funkcí. Nejdříve inicializujeme *svelte-query* klienta v rámci poskytovatele *QueryClientProvider* v App.svelte.

```
// Část souboru App.svelte

<script lang="ts">
  // Importy, konstanty, funkce...

  // Vytvoření instance QueryClient pro HTTP dotazy.
  const queryClient = new QueryClient();
</script>

<QueryClientProvider client={queryClient}>
  <!- Layout aplikace... ->
</QueryClientProvider>
```

Dále vytvoříme funkci *useAllCountries*, která vrátí výsledek HTTP dotazu (*CreateQueryResult*) pomocí funkce *createQuery*. Argumentem funkce *createQuery* bude objekt s názvem dotazu (*queryKey*) a funkce, která vykoná dotaz (*queryFn*).

```
// Část souboru queries.ts

export const useAllCountries = (): CreateQueryResult<Countries, Error> => {
  return createQuery<Countries>({
    queryKey: ['allCountriesQuery'], queryFn: getAllCountries
  });
};
```

Dotaz na server provede asynchronní funkce *getAllCountries*, v níž využijeme převzatou asynchronní funkci *requestHandler* [11] a knihovnu *axios* [51]. Po získání odpovědi ošetříme chyby a vrátíme výsledek.

```
// Část souboru getAllCountries.ts

export const getAllCountries = async () => {
  const fetchCountriesData = requestHandler<object, Countries>(() =>
    axios.request(getRequestConfig())
  );
  const response = await fetchCountriesData({});

  if (response.code === 'error') {
    throw new Error(
      `There was an error with getting the countries data
      (${response.error.message}). Please reload the page.`
    );
  }

  if (response.data.length === 0) {
    throw new Error('There are no countries to guess. Please try again later.');
  }
```

```
  return getSortedCountriesByName(response.data);
};
```

V rámci rodičovské komponenty dostaneme výsledek dotazu a uložíme jej do proměnné *countries*. Následně pomocí vlastností *isError* a *data* podmíněně vykreslíme jednotlivé komponenty. V případě chyby zobrazíme komponentu ErrorAlert. Když úspěšně získáme pole zemí, vykreslíme komponentu CountryGuesser. LoadingSkeleton zobrazíme, pokud se nezobrazí žádná z předchozích komponent.

```
// Soubor CountryGuesserWrapper.svelte

<script lang="ts">
  // Importy...

  const countries = useAllCountries();
</script>

{#if $countries.isError}
  <div
    class="container flex flex-col justify-center justify-items-center mx-auto"
  >
    <ErrorAlert message={$countries.error.message} />
  </div>
{:else if $countries.data}
  <CountryGuesser countries={$countries.data} />
{:else}
  <div
    class="container flex flex-col justify-center justify-items-center mx-auto"
  >
    <LoadingSkeleton />
  </div>
{/if}
```

Komponenta CountryGuesser zobrazí jednotlivé herní prvky a bude vyhodnocovat průběh hry. Začneme definicí stavů a náhodně vybereme náhodnou zemi (*randomCountry*), kterou uživatel bude hádat. V hooku *onMount* zavoláme funkci *polyfillCountryFlagEmojis*. Při namontování komponenty tak zajistíme zobrazení ikon vlajek v prohlížečích, které to přímo nepodporují. Prohlížeč uživatele však musí podporovat emojis a webové fonty. Funkce *polyfillCountryFlagEmojis* přidá do hlavičky stránky webový font Twemoji Country Flags. Aby se font použil, přidáme jej do CSS stylů.

```
// Část souboru app.css

@layer base {
  html {
```

```
    font-family: 'Twemoji Country Flags', 'ALTERNATIVNÍ_FONTY...';
  }
}
```

Pokračujeme implementací obslužných funkcí *handleEvaluateGuessAndUpdateState*, *handleSetInitialState*, které budou sloužit k aktualizaci stavu hry. V rámci šablony zobrazíme jednotlivé herní prvky a modální okna při výhře či prohře.

V rámci komponenty HintBoxes postupně zobrazíme nápovědy. Dle vstupu *randomCountry* budeme reaktivně vytvářet pole nápověd, jelikož *randomCountry* se může změnit. V šabloně posléze vykreslíme napovědy pomocí HintBox komponent. HintBox dynamicky vykreslí název a SVG ikonu nápovědy, textovou nápovědu, případně obrázek vlajky státu.

Komponenta *CountryGuessInput.svelte* umožní uživateli zadání názvu země (uživatelova tipu). Začneme šablonou, kde vytvoříme formulářový prvek pro zadání tipu a potvrzovací tlačítko. Dále také podmenu textového pole, které zobrazí nejpodobnější země na základě zadaného textu (filtrované země). Přidáme obslužné funkce pro akce a události nad formulářem, které následně doimplementujeme.

V rámci skriptové části, na základě vstupu *countries*, získáme pole všech zemí bez těch, které uživatel již hádal (*countriesWithoutAlreadyGuessed*). Dále definujeme a inicializujeme ostatní stavy komponenty. Po kliknutí na potvrzovací tlačítko zavoláme funkci *handleGuessButtonClick*. V tělě funkce zavoláme obslužnou funkci *evaluateGuessAndUpdateState*, pomocí níž vyhodnotíme stav hry v rodičovské komponentě. Následně také funkci *handleChangeSelectedGuess*, která aktualizuje aktuální tip, filtrované země a uzavře podmenu. Funkce *handleInputChange* převede tip uživatele do daného formátu, aktualizuje aktuální tip a filtrované země. Ovládání textového pole pomocí klávesnice umožní funkce *handleKeyDown*.

V pomocné funkci *updateGuessAndFilteredCountries* nejprve získáme filtrované země podle uživatelova tipu. Následně aktualizujeme stavy *currentGuess*, *isValidGuess* a *filteredCountries*. Funkce *clampSelectedGuessIndex* zajistí, aby index vybrané země byl v požadovaném rozmezí (0 až počet filtrovaných zemí). K modifikaci stavu *selectedGuessIndex* použijeme funkci *changeSelectedGuessIndex*, která index aktualizuje o hodnotu předanou v argumentu. Tip uživatele v rámci funkce *convertToFormattedGuess* převedeme tak, aby začínal velkým písmenem a zbytek řetezce

byl složen z malých písmen.

Pro zobrazení všech již hádaných zemí uživatelem vytvoříme komponentu GuessedCountriesList. Ze vstupních vlastností *countries*, *guessedCountries* a *randomCountry* získáme proměnnou *enrichedGuessedCountries*. Jde o uživatelem hádané země s vlajkou a vzdáleností od *randomCountry*. K převodu využijeme JavaScriptovou funkci z jiného souboru. Vzdálenost zemí vypočteme pomocí knihovny *calculate-distance-between-coordinates* [26], která exportuje funkci *getDistanceBetweenTwoPoints*. Proměnnou *enrichedGuessedCountries* následně vykreslíme v rámci šablony.

Nakonec vytvoříme modální okna, které vykreslíme při výhře nebo prohře. Stavy *isWinModalOpen* a *isLoseModalOpen* aktualizujeme v rámci funkce *handleEvaluateGuessAndUpdateState* v CountryGuesser. Na základě těchto stavů podmíněně zobrazíme daná modální okna. Oběma oknům předáme *randomCountry* a obslužnou funkci *handleClose*. Do výherního modálu také počet potřebných pokusů. V jednotlivých komponentách (WinModal, LoseModal) vykreslíme komponentu BaseModal, která bude sloužit jako šablona pro obě okna. Do této komponenty pak předáme titulek, obsah modálu a obslužnou metodu *handleClose*. V šabloně BaseModal vykreslíme základní strukturu modálního okna s dynamickými vlastnostmi.

Routování a layout aplikace

Aplikaci rozdělíme do tří částí: hlavičky, patičky a samotného obsahu, v němž vykreslíme jednotlivé komponenty. Uživatel se bude moci přepínat mezi jednotlivými stránkami přes navigační menu.

Pro routování v aplikaci využijeme knihovnu *svelte-spa-router* [36]. Nejprve vytvoříme seznam cest aplikace (*appRoutes*).

```
// Část souboru appRoutes.ts

interface AppRoute {
  name?: string;
  path: string;
  component: ComponentType;
}

export const appRoutes: ReadonlyArray<AppRoute> = [
  {
    name: 'Home',
    path: '/',
    component: Landing,
```

```
  },
  {
    name: 'Counter',
    path: '/counter',
    component: Counter,
  },
  // Další cesty...
  {
    path: '*',
    component: PageNotFound,
  },
];
```

Následně v hlavní komponentě transformujeme *appRoutes* do požadovaného formátu (typu *RouteDefinition*) a výsledek uložíme do proměnné *routes*. V šabloně zobrazíme hlavičku, patičku a *Router*, kterému předáme proměnnou *routes*. *Router* následně vykreslí šablonu na základě aktuální URL adresy.

```
// Část souboru App.svelte

<script lang="ts">
  // Importy...

  // Vytvoření cest pro svelte-spa-router.
  const routes: RouteDefinition = appRoutes.reduce(
    (routesMap, route) => routesMap.set(route.path, wrap({
      component: route.component
    })),
    new Map()
  );
</script>

<QueryClientProvider client={queryClient}>
  <div class="min-h-screen flex flex-col">
    <Header />

    <main class="flex-grow p-8">
      <!- Router vykresluje šablonu (komponentu) pro aktuální URL adresu. ->
      <Router {routes} />
    </main>

    <Footer />
  </div>
</QueryClientProvider>
```

Hlavička zobrazí odkazy na jednotlivé stránky. Architekturu a vzhled navigačního menu převezmeme např. od Flowbite. V rámci komponenty Header vypíšeme cesty aplikace pomocí HTML elementu a, na nějž přidáme atribut href. Dále přidáme také akce *link* a *active*, které poskytuje *svelte-spa-router*. Akci *active* předáme objekt, kde přes vlastnosti *className* a *inactiveClassName* nastavíme požadovanou

CSS třídu podle toho, zda je odkaz aktivní nebo neaktivní. K nastavení aria-current použijeme *location* objekt ze *svelte-spa-router*, díky kterému získáme aktuální URL.

```
// Část souboru Header.svelte

{#each routes as route}
  <li>
    <!- svelte-spa-router poskytuje akce "link" a také "active". ->
    <!- Akce active slouží k nastavení CSS na základě aktivního odkazu. ->
    <a
      href={route.path}
      class="block py-2 pr-4 pl-3 lg:p-0"
      use:link
      use:active={{
        className: 'STATICKÉ STYLY PRO AKTIVNÍ ODKAZ...',
        inactiveClassName: 'STATICKÉ STYLY PRO NEAKTIVNÍ ODKAZ...',
      }}
      aria-current={route.path === $location ? 'page' : null}
    >
      {route.name}
    </a>
  </li>
{/each}
```

Stavy *isMobileNavOpen* a *isDarkMode* umožní ovládat zobrazení mobilní navigace a barevného režimu. K uložení preference tmavého režimu využijeme LocalStorage v prohlížeči. Logiku pro přepnutí barevného režimu zavoláme v rámci hooku *beforeUpdate*. Tento hook se spustí po změně lokálního stavu, ale před aktualizací HTML.

```
// Část souboru Header.svelte

beforeUpdate(() => {
  if (isDarkMode) {
    document.documentElement.setAttribute('data-mode', 'dark');
    localStorage.setItem('data-mode', 'dark');
  } else {
    document.documentElement.removeAttribute('data-mode');
    localStorage.removeItem('data-mode');
  }
});
```

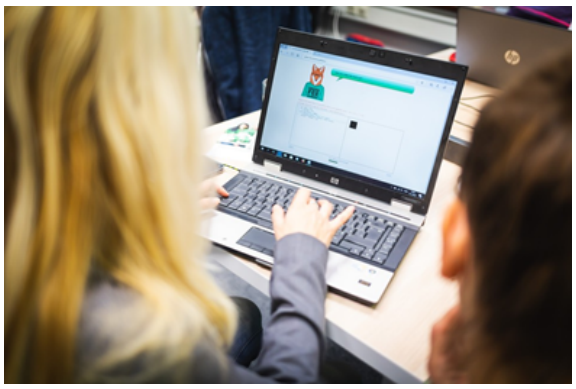
3.3 Srovnání implementace aplikací

4 Ukázková kapitola

4.1 Obrázky a tabulky

Obrázky a tabulky mají být uzavřeny v příslušném prostředí (pro obrázky je to „figure“, pro tabulky „table“, vždy s titulkem, viz níže). Toto prostředí zajistí správné zarovnání a umístění objektu. Není nutné, aby objekt byl přesně na místě, kde se o něm píše v textu, lze použít odkaz, třeba na obrázek 5 na straně 78.

Popisky sázíme pod obrázky a nad tabulky. Popisek nebastlete ručně, ale využijte prostředky LaTeXu – důvodem je, aby bylo možné automaticky vygenerovat seznam obrázků a seznam tabulek.



Obrázek 5: Ukázka vložení titulku s označením zdroje[52]

Všimněte si ve zdroji souboru (v tomto případě soubor zakladni-info.tex), jak je zajištěno, aby označení zdroje sice bylo u obrázku, ale neobjevilo se v seznamu obrázků – makro `\caption` má povinný parametr (to, co se objeví u obrázku) i nepovinný parametr (to, co se objeví v seznamu obrázků).

V tabulkách použijte raději řádkování trochu užší než 1,5, třeba 1.1 nebo 1.2.

Pokud jste přejali tabulku nebo to, co se v ní nachází, také je nutné uvést zdroj, podobně jako u obrázku.

Pokud obrázky a tabulky nepoužíváte (nebo máte jednu či dvě), vzadu odstraňte seznam obrázků/tabulek.

Tabulka 2: Ukázka tabulky

Číslo	**Jméno**	**Věk**	**Očkování**
1	Žeryk	4	ano
2	Andy	7	ano
3	Ťapka	2	ne

4.1.1 Vkládání ukázkového kódu

Pokud vkládáte ukázkový kód, můžete buď formátovat ručně, nebo použít zde definované prostředí `prog`, nebo použít vhodný balíček, můžu doporučit například algorithm2e – informace, manuál a samotný balíček najdete na [55].

Doporučení pro ruční formátování:

- V daném úseku nastavte řádkování na 1 nebo jen mírně větší:

  ```
  \radkovani[1]
  ```

- Použijte neproporcionální písmo a menší font:

  ```
  {\ttfamily\small
  ...
  }
  ```

- Vraťte řádkování na výchozí hodnotu:

  ```
  \radkovani
  ```

Ukázka využití prostředí `prog`:

```
if (pocet_bodu > 100)
print ("chyba při výpočtu bodů nebo zásah hackera");
ukonci_program();

if (pocet_bodu > 50)
udelit_zapocet(pocet_bodu);
else
informuj_nahradni_terminy();
```

Kolem tohoto prostředí vždy nechejte volný řádek.

4.2 Pojmenované odstavce

Finální úpravou je zajištění toho, aby na koncích řádků nebyly jednopísmenné předložky a spojky. V LaTeXu toho docílíme tak, že mezeru mezi jednopísmennou předložkou/spojkou a následujícím slovem nahradíme vlnkou: `~`. Koncem řádku by taktéž

nemělo být odděleno číslo od jednotky, tedy například 12 kg. Lze použít program `vlna` od Olšáka, informace a program na [62].

Poznámky pod čarou používejte jen v nejnutnějších případech – prosím nenuťte čtenáře každou chvíli šilhat na konec stránky :-).

Pojmenovaný odstavec. Takto můžete vyřešit situaci, kdy potřebujete sekci rozdělit, ale nechcete použít číslovaný nadpis (třeba proto, že tento text by byl jen krátký a nemá smysl navážet ho do obsahu).

Závěr

Úvod, Závěr a Seznam použité literatury jsou nečíslované kapitoly řazené do Obsahu.

Do Závěru píšeme souhrn poznatků zjištěných v práci, hodnotíme výsledek práce (někde by tu měla být i větička „Cíl práce byl splněn“, nějak vhodně rozvedená). Také tu můžeme psát o případných problémech, se kterými jsme se při psaní práce setkali, možnostech využití, námětech na pokračování do budoucna (tj. co by se dalo zlepšit, přidat, rozšířit, atd.).

Seznam použité literatury

[1] *All You Need to Know About VueJS*. Online. Flexiple. Dostupné z: https://flexiple.com/vue/deep-dive. [cit. 2023-11-07].

[2] ALSWEIRKI, Nouraldin. *State Management in React*. Online. Medium. 2023. Dostupné z: https://medium.com/@nouraldin.alsweirki/state-management-in-react-d086459e0bc5. [cit. 2023-10-17].

[3] *Angular*. Online. Dostupné z: https://angular.dev/. [cit. 2024-03-12].

[4] *Angular: Deliver web apps with confidence*. Online. Dostupné z: https://angular.io/. [cit. 2024-03-12].

[5] *Awesome React*. Online. Github. Dostupné z: https://github.com/enaqx/awesome-react. [cit. 2023-10-23].

[6] *Awesome Vue.js*. Online. Github. Dostupné z: https://github.com/vuejs/awesome-vue. [cit. 2023-11-08].

[7] BAMPAKOS, Aristeidis a DEELEMAN, Pablo. *Learning Angular: A no-nonsense guide to building web applications with Angular*. Online. 4. Packt Publishing, 2023. ISBN 9781803240602. Dostupné z: https://www.oreilly.com/library/view/learning-angular/9781803240602/. [cit. 2024-03-12].

[8] BANKS, Alex a PORCELLO, Eve. *Learning React*. Online. 2nd Edition. O'Reilly Media, 2020. ISBN 9781492051725. Dostupné z: https://www.oreilly.com/library/view/learning-react-2nd/9781492051718/. [cit. 2023-10-15].

[9] BREWSTER, Cordenne. *15 Examples of Global Websites Using Vue.js in 2023*. Online. Trio Developers - Stop searching. Start building. 2021. Dostupné z: https://www.trio.dev/blog/websites-using-vue. [cit. 2023-11-07].

[10] COPES, Flavio. *The Svelte Handbook – Learn Svelte for Beginners*. Online. FreeCodeCamp Programming Tutorials: Python, JavaScript, Git & More. 2019. Dostupné z: https://www.freecodecamp.org/news/the-svelte-handbook/. [cit. 2023-10-29].

[11] COSDEN, Darius. *Axios requestHandler function*. Online. Github. 2023. Dostupné z: https://github.com/cosdensolutions/code/blob/master/videos/long/request-handler-example/src/api/requestHandler.ts. [cit. 2024-03-10].

[12] DHOKAI, Chintan. *Angular State Management: A Comparison of the Different Options Available*. Online. Medium. 2023. Dostupné z: https://dev.to/chintanonweb/angular-state-management-a-comparison-of-the-different-options-available-100e. [cit. 2024-03-12].

[13] GAVINO, Jessica. *The Rise of Svelte: A New Era in Front-End Development*. Online. Hey Reliable. 2023. Dostupné z: https://heyreliable.com/the-rise-of-svelte-a-new-era-in-front-end-development/. [cit. 2023-11-02].

[14] GASANOV, Teimur. *React State Management for Enterprise Applications*. Online. Toptal. Dostupné z: https://www.toptal.com/react/react-state-management-tools-enterprise. [cit. 2023-10-17].

[15] *Getting started with Svelte*. Online. MOZILLA FOUNDATION. MDN Web Docs. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Svelte_getting_started. [cit. 2023-10-29].

[16] GOPINATH, Vishwas. *The React Ecosystem in 2023*. Online. Builder.io. 2023. Dostupné z: https://www.builder.io/blog/react-js-in-2023. [cit. 2023-10-23].

[17] HÁMORI, Ferenc. *The History of React.js on a Timeline*. Online. RisingStack. 2022. Dostupné z: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/. [cit. 2023-10-15].

[18] HERBERT, David. *What is React.js? (Uses, Examples, & More)*. Online. HUBSPOT. HubSpot Blog. 2022. Dostupné z: https://blog.hubspot.com/website/react-js. [cit. 2023-10-15].

[19] *Heroicons: Beautiful hand-crafted SVG icons, by the makers of Tailwind CSS*. Online. Dostupné z: https://heroicons.com/. [cit. 2024-03-13].

[20] JASPERS, Rens. *NgxLoadWith library*. Online. Github. 2023. Dostupné z: https://github.com/rensjaspers/ngx-load-with. [cit. 2024-03-10].

[21] KRAMER, Nimrod. *Svelte pros and cons, ecosystem overview and top resources*. Online. Daily.dev. 2020. Dostupné z: https://daily.dev/blog/building-with-svelte-all-you-need-to-know-before-you-start. [cit. 2023-11-02].

[22] LINSLEY, Tanner a COLLINS, Lachlan. *TanStack Query v5 library: Svelte Query*. Online. Github. 2023. Dostupné z: https://github.com/TanStack/query. [cit. 2024-03-10].

[23] MACRAE, Callum. *Vue.js: Up and Running*. Online. O'Reilly Media, 2018. ISBN 9781491997246. Dostupné z: https://www.oreilly.com/library/view/vuejs-up-and/9781491997239/. [cit. 2023-11-07].

[24] MÁCA, Jindřich. *Lekce 5 - Stavy v Reactu a hook useState()*. Online. Itnetwork.cz - Učíme národ IT. 2019. Dostupné z: https://www.itnetwork.cz/javascript/react/zaklady/stavy-v-reactu-a-hook-usestate. [cit. 2023-10-16].

[25] MARANAN, Menard. *The React lifecycle: methods and hooks explained*. Online. Retool. 2022. Dostupné z: https://retool.com/blog/the-react-lifecycle-methods-and-hooks-explained/. [cit. 2023-10-17].

[26] MARTENS, Tijs. *Calculate Distance between coordinates library*. Online. Github. 2022. Dostupné z: https://github.com/TijsM/distance-between-coordinates. [cit. 2024-03-10].

[27] *NgRx: Reactive State for Angular*. Online. Dostupné z: https://ngrx.io/. [cit. 2024-03-12].

[28] NORTE, Rubén. *Axios-retry library*. Online. Github. 2016. Dostupné z: https://github.com/softonic/axios-retry. [cit. 2024-03-10].

[29] *Pinia | The intuitive store for Vue.js*. Online. Dostupné z: https://pinia.vuejs.org/. [cit. 2023-11-07].

[30] QUENSE, Jason. *Yup library*. Online. Github. 2014. Dostupné z: https://github.com/jquense/yup. [cit. 2024-03-10].

[31] *React*. Online. 2023. Dostupné z: https://react.dev/. [cit. 2023-10-15].

[32] *React*. Online. Github. Dostupné z: https://github.com/facebook/react. [cit. 2023-10-15].

[33] *React Hook Form library*. Online. Github. 2019. Dostupné z: https://github.com/react-hook-form/react-hook-form. [cit. 2024-03-10].

[34] *React Router*. Online. Dostupné z: https://reactrouter.com/en/main. [cit. 2023-10-18].

[35] *RxJS: Reactive Extensions Library for JavaScript*. Online. Dostupné z: https://rxjs.dev/. [cit. 2024-03-12].

[36] SEGALA, Alessandro. *Svelte-spa-router library*. Online. Github. 2019. Dostupné z: https://github.com/ItalyPaleAle/svelte-spa-router. [cit. 2024-03-10].

[37] *Stack Overflow - Where Developers Learn, Share, & Build Careers*. Online. Dostupné z: https://stackoverflow.com/. [cit. 2023-11-14].

[38] *Stack Overflow Developer Survey 2023*. Online. 2023. Dostupné z: https://survey.stackoverflow.co/2023/. [cit. 2023-11-14].

[39] *Svelte • Cybernetically enhanced web apps*. Online. Dostupné z: https://svelte.dev/. [cit. 2023-10-29].

[40] *SvelteKit • Web development, streamlined*. Online. Dostupné z: https://kit.svelte.dev/. [cit. 2023-11-04].

[41] *Svelte For Beginners*. Online. Joy of Code. 2021. Dostupné z: https://joyofcode.xyz/svelte-for-beginners. [cit. 2023-10-31].

[42] *Svelte State Management Guide*. Online. Joy of Code. 2022. Dostupné z: https://joyofcode.xyz/svelte-state-management. [cit. 2023-10-31].

[43] *Svelte, Solid and Qwik: the rise of new front-end frameworks*. Online. DevInterface. 2023. Dostupné z: https://www.devinterface.com/en/blog/svelte-solid-and-qwik-the-rise-of-new-front-end-frameworks. [cit. 2023-10-29].

[44] *Tailwind CSS: Rapidly build modern websites without ever leaving your HTML*. Online. Dostupné z: https://tailwindcss.com/. [cit. 2024-03-13].

[45] TEESELINK, Egbert a HUISMAN, Daniëlle. *Country Flag Emoji Polyfill library*. Online. Github. 2022. Dostupné z: https://github.com/talkjs/country-flag-emoji-polyfill. [cit. 2024-03-10].

[46] *Using JavaScript Libraries In Svelte*. Online. Joy of Code. 2023. Dostupné z: https://joyofcode.xyz/using-javascript-libraries-in-svelte. [cit. 2023-11-02].

[47] *Vue.js - The Progressive JavaScript Framework*. Online. Dostupné z: https://vuejs.org/. [cit. 2023-11-07].

[48] *Vue.js history*. Online. Medium. Dostupné z: https://madushaprasad21.medium.com/vue-js-history-1a6b8567198f. [cit. 2023-11-07].

[49] *Vue Router | The official Router for Vue.js*. Online. Dostupné z: https://router.vuejs.org/. [cit. 2023-11-07].

[50] YEUNG, Tjin Au. *Svelte forms lib*. Online. Github. 2019. Dostupné z: https://github.com/tjinauyeung/svelte-forms-lib. [cit. 2024-03-10].

[51] ZABRISKIE, Matt. *Axios library*. Online. Github. 2014. Dostupné z: https://github.com/axios/axios. [cit. 2024-03-10].

[52] 11 Mýtů o programátorech. *Green Fox Academy* [online]. 2018 [cit. 2022-07-22]. Dostupné z: https://www.greenfoxacademy.cz/post/11-mytu-o-programatorech

[53] BARTOŠ, Aleš. *Autorské právo v otázkách a odpovědích*. Praha: Pierot, 2012. ISBN 978-80-7353-223-9.

[54] *Bibliografické citace*. Praha, 2011.

[55] FIORIO, Christophe. Algorithm2e.sty – package for algorithms. CTAN: Package algorithm2e [online]. 2017 [cit. 2022-09-15]. Dostupné na: https://www.ctan.org/pkg/algorithm2e

[56] Generátor citací [online]. *Citace PRO* [cit. 2022-07-22]. Dostupné z: https://www.citacepro.com

[57] KOČIČKA, Pavel a Filip BLAŽEK. *Praktická typografie*. Dotisk druhého vydání. Brno: Computer Press, 2007. DTP. ISBN 80-722-6385-4.

[58] LaTeX, Evolved. *Overleaf, online LaTeX editor* [online]. Overleaf.com [cit. 2022-09-15]. Dostupné z: https://www.overleaf.com/

[59] MEŠKO, Dušan, Dušan KATUŠČÁK, Ján FINDRA a kol. *Akademická příručka*. 2. upravené a dopl. vyd. Martin: Osveta, 2005. ISBN 80-8063-200-6.

[60] *Metodický pokyn děkana č. 1/2021 ke zpracování, zveřejňování a ukládání závěrečných prací na Filozoficko-přírodovědecké fakultě v Opavě* [online]. Opava: FPF SU v Opavě, 2021 [cit. 2022-09-15]. Dostupné na: https://www.slu.cz/fpf/cz/sostatnizaverecnezkousky

[61] Stanford Libraries: Find Dissertations and Theses [online]. *Stanford.edu* [cit. 2018-03-19] Dostupné z: http://library.stanford.edu/guides/find-dissertations-and-theses

[62] RUSSL. Tipy pro LaTeX: Program vlna pro Windows. WorkIT [online]. K-media [cit. 2022-09-15], 2014. Dostupné na: https://www.work-it.cz/tipy-pro-latex-program-vlna-pro-windows/

[63] SCHENK, Christian. Welcome to the MikTEX project page. MikTEX [online]. [cit. 2022-09-15], ©2022. Dostupné z: https://miktex.org/

[64] ŠANDEROVÁ, Jadwiga. *Jak číst a psát odborný text*. Praha: Slon, 2007. ISBN 978-80-86429-40-3.

Seznam obrázků

Seznam tabulek

Seznam zkratek

API	Application Programming Interface
DRY	Don't repeat yourself
KISS	Keep it simple, stupid!
OOP	Objektově orientované programování
SOLID	Návrhový princip SOLID

Přílohy

Do tohoto seznamu napište přílohy vložené přímo do této práce a také seznam elektronických příloh, které se vkládají přímo do archivu závěrečné práce v informačním systému zároveň se souborem závěrečné práce. Elektronickými přílohami mohou být například soubory zdrojového kódu aplikace či webových stránek, předpřipravený produkt (spustitelný soubor, kontejner apod.), vytvořená metodická příručka, tutoriál... (tento text odstraňte)

- Přílohy v souboru závěrečné práce:
  - Příloha A xxxx
  -
- Elektronické přílohy:
  - Příloha A xxxx
  -