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.

**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 5. 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

# 3 Testování frameworků

- proč a co je obsahem kapitoly?

## 3.1 Analýza a návrh testových úloh

- co a proč porovnávám,
- v návrhu - jak, jaké testové úlohy?
- (dokumentace - možná nahoře, syntax, výkonnostní testy, velikosti bundlů, účel aplikace, rychlost, srozumitelnost, ...)

## 3.2 Demonstrační aplikace

V této kapitole srovnáme implementaci stejných funkcionalit ve třech vybraných frameworcích.

### 3.2.1 Angular

#### Instalace projektu

- Node.js + NPM
- npm init @angular@latest NAZEV_APLIKACE
- https://www.npmjs.com/package/@angular/create
- https://tailwindcss.com/docs/guides/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() našeho 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></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 musíme importovat třídu CounterButton-Component a do všech elementů counter-button předat 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());
}
```

Se službou TranslationService a veřejnou metodou pro vykonání dotazu nám pomůže třída HttpClient. Ta je dostupná přímo v základních modulech Angularu. Službu HttpClient získáme v konstruktoru, kde ji pomocí klíčového slova private přiřadíme do vlastností třídy. Následně na HTTP klientovi zavoláme metodu post vůči API, které vrátí přeložený text ze serveru. 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.

```
// 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. 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 podporu zobrazení ikon vlajek v prohlížečích, které to přímo nepodporují. 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.

Služba CountryFlagPolyfillService, jak již bylo popsáno výše, nám dopomůže s podporou zobrazení ikon vlajek v prohlížečích, které nemají podporu zobrazení vlajek. Pokud prohlížeč uživatele podporuje emojis a webové fonty, zavoláním funkce polyfillCountryFlagEmojis přes metodu usePolyfill knihovna přidá webový font Twemoji Country Flags do HTML hlavičky. 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 komponentou 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 třídě 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 countriesWithoutAlreadyGuessed. 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. 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.

### Layout aplikace, routování

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 pro jednotlivé stránky v souboru app.routes.ts. Cesty nakonfigurujeme podle předpisu Routes a pole cest exportujeme. Následně tyto cesty aplikace poskytneme routeru v rámci app.config.ts.

```
// Část souboru app.routes.ts

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, s čímž nám pomohou direktivy routerLink, routerLinkActive, routerLinkActiveOptions a reference #link. Do routerLink předáme cestu a routerLinkActive umožní naslouchání na aktuální cestu, kde se nacházíme. Direktiva routerLinkActiveOptions pak přepíše výchozí nastavení routerLinkActive. Reference #link nám umožní získat informaci o tom, zda je odkaz aktivní. To využijeme při podmíněném nastavení správných CSS tříd.

```
// 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_LINK': link.isActive,
        'STATICKÉ_STYLY_PRO_NEAKTIVNÍ_LINK': !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

#### Instalace projektu

#### 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.

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).

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.

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.

### Interakce v uživatelském prostředí

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

Hook useDropdown bude mít 2 parametry – výchozí hodnotu vybrané možnosti (defaultValue) a obslužnou funkci ke změně vybrané v možnosti v rodičovské komponentě (onChange). 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.

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().

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.

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.

Dropdown samozřejmě může mít i jiné vstupy, které povedou k lepší znovupoužitelnosti. Dynamické 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 funkce) 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. 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.

Hook useAutosizeTextArea bude příjímat 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.

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 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.

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. 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 výsledků 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.

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 aktu-

ální 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.

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.

### Modularita, použití knihoven

V následující sekci vytvoříme webovou hru, ve které cílem uživatele je 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 getAllCountries. Jde o převzatý requestHandler s využitím knihovny axios, jenž umožní otypování příchozí odpovědi. Po ošetření chyb aktualizujeme patřičné stavy, které následně z hooku vrátíme. Implementaci načítacích a chybových stavů nebo rušení asynchronních dotazů nám do značné míry může ulehčit knihovna react-query.

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.

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 useCountryFlag-

Polyfill, 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 metod 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.

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 pak uživateli umožní 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 metody pro akce a události nad formulářem, které následně doimplementujeme.

Ve funkční komponentě CountryGuessInputComponent na základě vstupu countries získáme pole všech zemí bez těch, které uživatel již hádal (countriesWithoutAlreadyGuessed). Poté definujeme a inicializujeme ostatní stavy komponenty. 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 JavaScriptové funkce z jiného souboru. K vypočtení vzdálenosti použijeme knihovnu calculate-distance-between-coordinates, 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 metodu handleClose. BaseModal následně v JSX vykreslí základní strukturu modálního okna, s dynamickými možnostmi pro titulek, obsah a obslužnou metodu handleClose.

### Layout aplikace, routování

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. Začneme vytvořením souboru s cestami (routes). 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 routeru do kořenové komponenty aplikace, konkrétně do poskytovatele routeru.

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

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. Ta nám umožní v atributu className získat vlastnost isActive, která indikuje, zda je odkaz aktivní. Pro korektní nastavení aria-current atributu použijeme hook useLocation, který nám umožní získat aktuální cestu.

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.

### 3.2.3 Svelte

#### Instalace projektu

#### Správa stavů, předávání vlastností

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

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.

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 modifikujeme count.

**Interakce v uživatelském prostředí**

**Reaktivita, asynchronní operace**

**Tvorba formulářů, validace**

**Modularita, použití knihoven**

**Layout aplikace, routování**

## 3.3 Testování aplikací a výsledky

- výsledky a průběh z 3.1

# Seznam obrázků

# Seznam tabulek

# 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
  -