Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Srovnání formulářů ve frameworcích pro vývoj webových aplikací

Madi Khabidenov

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Madi Khabidenov

Informatika

Název práce

Srovnání formulářů ve frameworcích pro vývoj webových aplikací

Název anglicky

Framework forms comparison for web development

Cíle práce

Hlavním cílem této práce je navrhnout, implementovat a srovnat vybrané formuláře ve frontendových frameworcích jako Angular a React pro vývoj webových aplikací.
Vedlejší cíle práce jsou:
– porovnat metody, nástroje a osvědčené postupy pro vytváření formulářů
– implementovat vybrané formuláře a připravit experiment
– provést hodnocení na základě výkonnosti formulářů

Metodika

Teoretická část práce bude představena popisem využitých technologii a popisem principů fungování vybránych formulářů.

Práce bude zaměřená na praktické použití formulářů pro vývoj webových aplikací.

Praktická část práce bude spočivat v implementaci webových aplikací s použitím formulářů a bude provedena na základě experimentu.

Bude vytvořena jedná šablona jako konfigurace pro všechny formuláři, podle které budou vyvíjet webové aplikace. Aplikace budou naimplementované pomocí různých metod (Reactive Forms, ControlValueAccessor v Angular a další).

Vyhodnocení je tvořeno srovnáním výkonu vyvinutých aplikací, poukázáním na nedostatky měření a následujícím shrnutím celého experimentu.

Z teoretických poznatků a experimentu budou formulovány závěry práce.

Doporučený rozsah práce

40

Klíčová slova

frontend development, forms, angular, react, reactive forms, template-driven forms, react hook form, formik, experiment

---

Doporučené zdroje informací

Angular – documentation [online]. Dostupné z: https://angular.io/

HOGAN, Brian P. HTML5 a CSS3: Výukový kurz webového vývojáře. Brno: Computer Press, 2011. ISBN 9788025135761.

MOHAN, Mehul. Advanced Web Development with React: SSR and PWA with Next.js using React with advanced concepts. ENG ed. New Delhi: BPB Publications, 2020. ISBN 978-9389423594.

Murray, N., Coury, F., & Lerner, A. (2018). Ng-book: The complete guide to angular. Createspace Independent Publishing Platform.

React – documentation [online]. Dostupné z: https://reactjs.org/

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

RNDr. Alexander Galba

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 4. 7. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 15. 03. 2024

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Srovnání formulářů ve frameworcích pro vývoj webových aplikací" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.03.2024 _____

Poděkování

Rád bych touto cestou poděkoval RNDr. Alexanderovi Galbovi, za vedení své diplomové práce a konzultace.

Srovnání formulářů ve frameworcích pro vývoj webových aplikací

Abstrakt

Tato bakalářská práce se zaměřuje na porovnání a implementaci formulářů v rámci moderních technologií pro vývoj frontendu. V teoretické části jsou detailně rozebrány principy a technologie, které jsou klíčové pro vytváření formulářů. Praktická část práce se věnuje implementaci těchto metod a jejich srovnání. Kritickou součástí této práce je provedení unit testů pro ověření funkčnosti a spolehlivosti jednotlivých implementací formulářů. Závěrečná část práce se zabývá analýzou a vyhodnocením výsledků získaných z testování. Na základě těchto informací jsou prezentovány závěry týkající se efektivity různých technologií a metodik použitých při vytváření formulářů pro webové aplikace.

Klíčová slova: Frontend vývoj, formuláře, Angular, React, Reactive forms, Template-driven forms, React hook form, formik, Experiment

Framework forms comparison for web development

Abstract

This bachelor's thesis focuses on the comparison and implementation of forms within modern frontend development technologies. The theoretical part thoroughly discusses the principles and technologies that are key to creating forms. The practical section of the work is dedicated to the implementation of these methods and their comparison. A critical part of this work is conducting unit tests to verify the functionality and reliability of each form implementation. The final part of the work deals with the analysis and evaluation of the results obtained from testing. Based on this information, conclusions are presented regarding the efficiency of various technologies and methodologies used in creating forms for web applications.

Keywords: Frontend development, forms, Angular, React, Reactive forms, Template-driven forms, React hook form, formik, Experiment

Obsah

1 Úvod

V dnešní době, jak se digitalizace stává nedílnou součástí našich životů, rozplétá se před námi složitost webových aplikací rychlostí, která nás může ohromit. Ten softwarový základ, na kterém tyto aplikace stojí, je klíčový pro jejich správnou funkčnost a budoucí rozvoj. Využívání konkrétních technologií se ukazuje jako stále důležitější pro rychlý vývoj a bezproblémový chod těchto aplikací. Formuláře, kterými uživatelé s platformami komunikují, jsou jedním z rozhodujících prvků.

Za úspěchem, který odpovídá potřebám uživatelů, stojí intuitivní design pro interakci, precizní ověřování dat a schopnost efektivně vytvářet formuláře. Tyto aspekty jsou zásadní pro uspokojení uživatelských požadavků. Formuláře samy o sobě hrají klíčovou roli, protože umožňují uživatelům odesílat data na server prostřednictvím webového prohlížeče. Bez formulářů by nebylo možné online hledání informací, provádění finančních transakcí, správa uživatelských účtů nebo využívání marketingových strategií specifických pro online prostředí. Jinými slovy, každý online proces nebo transakce dnes probíhá díky existenci formulářů na internetu.

Front-end vývojáři, kteří se specializují na návrh interakcí webových stránek, včetně tvorby formulářů, jsou nezastupitelnou součástí procesu vývoje aplikací. Díky tomu, že formuláře jsou často prvním bodem kontaktu mezi uživatelem a platformou, jejich práce je zásadní. Dobře navržené a implementované formuláře mohou výrazně zlepšit uživatelskou zkušenost a efektivitu interakcí na webových stránkách.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této práce je navrhnout, implementovat a srovnat vybrané formuláře ve front-end frameworcích jako Angular a React pro vývoj webových aplikací.

Vedlejší cíle práce jsou:

- porovnat metody, nástroje a osvědčené postupy pro vytváření formulářů
- implementovat vybrané formuláře a připravit experiment
- provést hodnocení na základě výkonnosti formulářů

2.2 Metodika

V teoretické časti budou popsané základní principy fungování formulářů.

První podkapitola bude věnována principům fungování formulářů. Druhá kapitola bude věnována popisem technologie, které budou použité obojími frameworky v rámci experimentu. V třetí části bude popis použitých frameworků a popis jednotlivých nástrojů pro vývoj formulářů. V poslední částí bude popsána metodika hodnocení experimentu.

Praktická část práce bude spočívat v implementaci webových aplikací s použitím formulářů. Bude vytvořena jedná šablona jako konfigurace pro všechny formuláři, podle které budou vyvíjet webové aplikace. Aplikace budou naimplementované pomocí různých metod: Reactive Forms, Template-driven Forms v Angular a knihoven Formik, React Hook Form v React.

Na konci praktické časti bude provedeno hodnocení webových aplikací podle určitých kritérii na základě experimentu.

3 Teoretická východiska

Teoretická část práce se zaměřuje na vysvětlení principů fungování formulářů a podrobně rozebírá různé typy jednotek formuláře. Dále jsou v práci popsány technologie využité během experimentálního provedení, včetně HTML, Angular, React, Template-driven forms, Reactive Forms, Formik, React Hook Form, JSON a Unit testování. Tato sekce poskytne čtenářům ucelený přehled o metodách a nástrojích použitých v rámci výzkumu.

3.1 Principy fungování formulářů

V rámci rozsáhlého a důkladného výzkumu, který je součástí bakalářské práce, byla zvolena klíčová strategie pro implementaci všech formulářů. Tato strategie spočívá v tom, že všechny formuláře byly napsány s využitím značkovacího jazyka HTML (Hypertext Markup Language), což představuje základní stavební kámen a jazyk pro strukturování a prezentaci webových stránek a webových aplikací.

HTML formuláře představují základní komponentu interaktivní komunikace mezi uživateli a webovými aplikacemi. Reprezentují sekce webového dokumentu, které se vyznačují nejen standardním textovým obsahem, ale rovněž specifickými HTML prvky, označovanými jako „ovládací prvky“ (controls) a jejich příslušnými „popisky“ (labels). Tyto ovládací prvky se mohou objevovat v různých formách a typech, závisle na specifických potřebách formuláře. Mezi často využívané ovládací prvky patří zaškrtávací políčka (checkboxy), přepínače (radio buttony), textová pole, seznamy pro výběr a další. Každý z těchto elementů plní specifickou funkci a umožňuje shromažďování určitých dat nebo vstupů od uživatelů. Proces vyplňování HTML formulářů je koncipován tak, aby byl pro uživatele intuitivní, a je obecně známý. Interakce uživatelů s formulářem spočívá v modifikaci ovládacích prvků, což může zahrnovat vkládání textu do textových polí, zaškrtávání políček, výběr možností ze seznamů a podobné činnosti. Skrze tyto akce uživatelé postupně zadávají požadované informace nebo realizují výběry v souladu s danou situací [1].

Klíčovým okamžikem v procesu vyplňování formuláře je okamžik odeslání dat agentovi, který je schopen tyto informace zpracovat. Agentem může být například webový

server, e-mailový server nebo jiný backendový systém. Odeslání dat z formuláře znamená, že uživatelé potvrzují své volby nebo vstupy a odesílají je k dalšímu zpracování nebo ukládání. Tím dochází k důležitému propojení mezi front-endem a backendem webových aplikací [2].

Zde je ukázka implementace formuláře v HTML kódu [3].

```
<form>
  <label for="fname">First name:</label><br>
  <input type="text" id="fname" name="fname" value="John">
  <label for="lname">Last name:</label><br>
  <input type="text" id="lname" name="lname" value="Doe">
  <input type="submit" value="Submit">
</form>
```

3.1.1 HTML Form Controls

Interakce uživatele s formulářem je zásadním aspektem uživatelského rozhraní v rámci webových aplikací. Realizace této interakce probíhá skrze různorodé pojmenované prvektypy „ovládací prvky“ (controls), které představují jádro každého formuláře. Každý z těchto ovládacích prvků je charakterizován specifickým typem a má určený účel. Zásadní roli při manipulaci s těmito prvky hraje jejich identifikátor, jenž určuje, jakého typu ovládací prvek je. Formulář obvykle zahrnuje minimálně jeden takový ovládací prvek, který definuje jeho obecnou strukturu. Každý ovládací prvek typicky nese počáteční hodnotu, specifikovanou atributem „value“, a skutečnou hodnotu, jež může být modifikována buď uživatelskou interakcí, nebo programově skrze skript [1].

Různé typy ovládacích prvků umožňují odlišné formy interakce s formulářem:

- **Tlačítka (Buttons)**: Tlačítka představují fundamentální element formulářů, umožňující uživatelům vykonávat specifické akce. Je nezbytné určit atribut „type“ k definování funkcionality tlačítka. Existují tři hlavní typy tlačítek: „button“, „submit“ a „reset“, přičemž typ „submit“ je ve většině prohlížečů považován za výchozí, s výjimkou Internet Exploreru, kde výchozím typem je „button“.
- **Zaškrtávací políčko (Checkbox)**: Zaškrtávací políčka umožňují uživateli zapnout či vypnout konkrétní volbu. Specifickým atributem pro checkboxy a rádiová tlačítka je „checked“, který má boolean hodnotu a indikuje, zda je políčko zaškrtnuto. Umožňuje uživatelům vybírat více možností současně.
- **Rádiová tlačítka (Radio)**: Rádiová tlačítka jsou analogická zaškrtávacím políčkům, avšak fungují na principu výlučné volby. Uživatel může vybrat jen jednu možnost z nabízené skupiny rádiových tlačítek. Výběrem jedné možnosti se ostatní tlačítka ve skupině automaticky zruší.
- **Vstupní a textová pole (Input/Textarea)**: Tato pole poskytují prostor pro textový vstup od uživatele. Zásadní rozdíl mezi nimi spočívá v tom, že vstupní pole (input) je určeno pro jednořádkový vstup, zatímco textová pole (textarea) umožňují zadávání textu na více řádků. Hodnotou těchto polí je vložený text [2].

3.1.2 TypeScript

TypeScript je programovací jazyk, který vznikl na základě JavaScriptu a přináší do světa vývoje softwarových aplikací novou úroveň robustnosti a spolehlivosti. TypeScript byl vytvořen společností Microsoft jako interní projekt v průběhu roku 2010 a byl ve vývoji až do jeho veřejného uvedení v roce 2012. TypeScript byl původně vytvořen jako reakce na požadavky jak interních vývojářů, tak externích klientů, kteří si přáli frontendový jazyk, který by byl bezpečnější k použití než existující JavaScript a zároveň stejně snadný na implementaci a zavedení. TypeScript byl odpovědí společnosti Microsoft na tyto

požadavky. Tímto způsobem TypeScript pomáhá odhalit chyby a problémy v kódu již při vývoji, což usnadňuje proces ladění a zvyšuje spolehlivost aplikací [5] [6].

Jedná se o primární aspekt TypeScriptu, který značně usnadňuje jeho adopci mezi vývojáři. Díky zachování kompatibility se syntaxí a funkcemi JavaScriptu může být přechod na TypeScript plynulý a nevyžaduje rozsáhlé přeškolování nebo zvládnutí zcela nového jazyka. Tato vlastnost také umožňuje, aby byly stávající projekty napsané v JavaScriptu postupně refaktorovány do TypeScriptu, což přináší výhody silnějšího typového systému a lepšího nástrojového supportu, aniž by bylo nutné provádět radikální změny ve způsobu vývoje nebo v architektuře aplikace [5].

Fundamentálním aspektem TypeScriptu je schopnost explicitní definice typů. To umožňuje vývojářům určit typy proměnných, očekávané typy parametrů funkcí a návratové typy funkcí. Kontrola typů se aplikuje staticky, tj. je realizována v průběhu kompilace kódu, nikoli během spuštění aplikace. Tento přístup vede k předčasnému odhalení chyb, což pozitivně přispívá k celkové kvalitě kódu a umožňuje vývojářům věnovat větší pozornost logice aplikace. [6].

TypeScript poskytuje také širokou škálu nástrojů a knihoven, které usnadňují vývoj, testování a správu kódu. Díky těmto nástrojům mohou vývojáři psát efektivní a udržitelný kód, což je klíčové pro dlouhodobou úspěšnost projektů [7].

3.2 Frameworky

3.2.1 Angular

Angular je komplexní platforma pro vývoj webových aplikací, postavená na základech programovacího jazyka TypeScript a značkovacího jazyka HTML. Tento framework se rychle stal jedním z klíčových hráčů ve světě moderního webového vývoje, a to díky svým mnoha výhodám a schopnostem.

Hlavním cílem Angularu je umožnit vývoj tzv. single-page aplikací (SPA). SPA jsou webové aplikace, které nabízejí plynulou a rychlou uživatelskou zkušenost tím, že načítají jednu stránku a následně dynamicky mění obsah této stránky bez nutnosti kompletního znovunačítání. To znamená, že uživatelé nemusí čekat na přechod mezi stránkami, což zvyšuje interaktivitu a komfort používání webových aplikací [8].

Výhody frameworku Angular jsou mnohostranné a zahrnují:

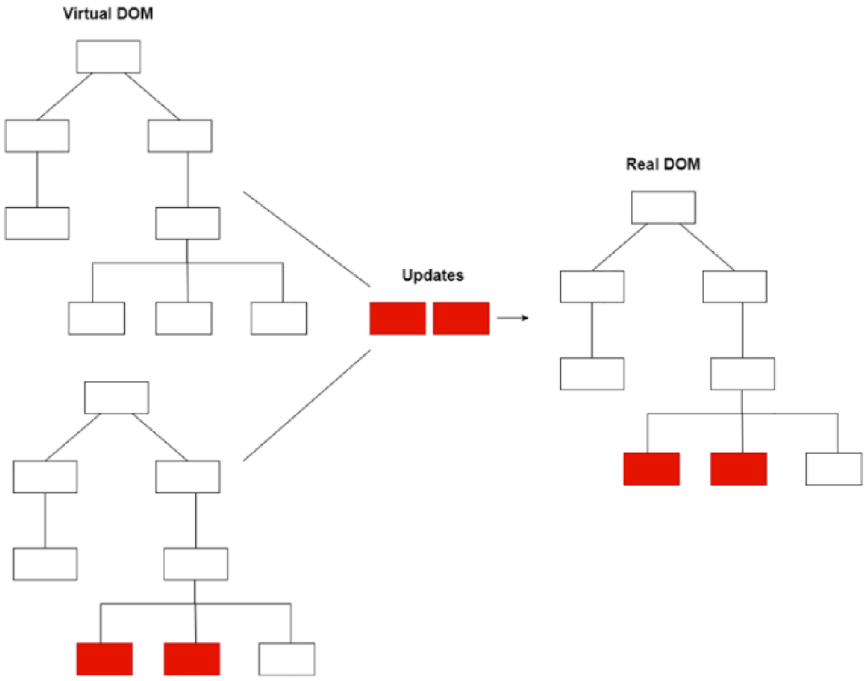
- **Komponentový Přístup**: Angular staví na komponentovém modelu, což umožňuje rozdělit aplikaci na menší a opakovatelné části. Každá komponenta má svou vlastní logiku, HTML šablonu a styly, což zjednodušuje vývoj a správu kódu. Komponenty mohou být znovupoužity napříč celou aplikací, což usnadňuje škálování projektu [9].
- **Rozsáhlá Knihovna**: Angular obsahuje rozsáhlou knihovnu funkcí a nástrojů, které usnadňují vývoj aplikací. Tato knihovna zahrnuje funkce pro routování (navigaci mezi různými stránkami), řízení formulářů, komunikaci s backendem pomocí HTTP služeb a mnoho dalších [9].
- **Vývojářské Nástroje**: Angular poskytuje širokou paletu nástrojů, které pomáhají vývojářům při tvorbě, testování a správě kódu. Mezi tyto nástroje patří Angular CLI (Command Line Interface) pro generování projektů a kódování, k dispozici jsou také rozšíření pro populární vývojová prostředí [9].
- **Škálovatelnost**: Díky komponentové architektuře a dobře definovaným vzorům je Angular vhodný pro tvorbu škálovatelných a rozsáhlých webových aplikací. To je zásadní pro projekty, které potřebují růst a vyhovovat stále narůstajícím potřebám uživatelů [9].

3.2.2 **React**

React byl vyvinut společností Facebook s cílem poskytnout jejich kódové bázi větší strukturu a umožnit její lepší škálovatelnost. React se pro Facebook ukázal být natolik účinným, že byl nakonec uvolněn jako open source. V současnosti představuje React jednu z nejpopulárnějších JavaScriptových knihoven pro vývoj frontendů. Umožňuje nám vytvářet malé, izolované a vysoce znovupoužitelné komponenty, které lze kombinovat dohromady za účelem vytvoření složitých frontendových rozhraní [10].

Existuje několik klíčových výhod Reactu, které přispívají k jeho širokému využití:

- **Jednoduchost vytváření dynamických aplikací**: React nabízí přístup k tvorbě uživatelských rozhraní, který je zároveň jednoduchý a výkonný. Vývojáři mohou pracovat s tzv. "komponentami," což jsou znovupoužitelné a izolované stavební bloky aplikace. Tato komponentová architektura zjednodušuje organizaci kódu a umožňuje snadnou údržbu a rozšiřitelnost aplikace [15].
- React využívá koncept nazvaný **Virtuální DOM**, díky kterému funguje trochu odlišně. Vytváří dvě kopie DOM v paměti. Při změně DOM dojde ke změně jedné kopie DOM. Následně se tato kopie porovná s druhou kopií za účelem určení, co bylo změněno. Tento proces se nazývá diffing (rozlišování rozdílů) [11]**.**



Obrázek 1 Virtuální DOM v Reactu. Zdroj: [11]

- Znovupoužitelnost komponent: Jako součást komponentového modelu umožňuje React vývojářům vytvářet znovupoužitelné komponenty. To znamená, že jednou napsaný a otestovaný komponent může být opakovaně použit v různých částech aplikace, což šetří čas a snižuje riziko chyb [11].

- V Reactu umožňují komponenty znovupoužitelnost a modularitu. Každá komponenta má svůj úkol a soustředí se na něj. Existují dva typy komponent v Reactu: třídní komponenty a funkční komponenty (může být použit termín funkční komponenta nebo funkcionalitní komponenta, oba výrazy označují totéž) [11].

React je takovým stavebním kamenem moderního front-end vývoje a nabízí širokou škálu nástrojů a knihoven pro tvorbu interaktivních a rychlých webových aplikací. Díky jeho jednoduchosti a výkonnosti je oblíbeným výběrem pro vývojáře a firmami, které se snaží nabídnout uživatelům optimální uživatelské zkušenosti [11].

3.3 Formuláře

Angular nabízí dvě hlavní metody pro ovládání uživatelských vstupů pomocí formulářů: Templat-driven (templátově řízené) a Reactive (reaktivní) formuláře. Obě tyto metody umožňují vývojářům zpracovávat uživatelské vstupy, validovat je, vytvářet modely formulářů a uchovávat data, která uživatelé do formulářů zadávají. Kromě toho umožňují sledovat změny v datech a provádět reakce na tyto změny [9] [12].

V kontextu Reactu, knihovny pro vývoj uživatelských rozhraní, se zacházení s těmito prvky liší. React poskytuje komponenty pro tvorbu formulářů s jednosměrným tokem dat, což umožňuje efektivní správu a dynamické interakce. Tento přístup zahrnuje jednosměrný tok dat, prevenci výchozích akcí formulářů, vytváření vlastních komponent formulářů a zpracování dat formulářů. React tedy nabízí rozšířené možnosti pro práci s formuláři ve srovnání s tradičním HTML, vyžadující porozumění specifikům knihovny pro efektivní využití. [15]

3.3.1 Angular Reactive forms

Reactive forms přináší do vývoje webových aplikací další dimenzi a možnosti, které mohou být pro některé projekty klíčové. Reactive forms jsou postaveny na konceptu reaktivního programování a umožňují vývojářům přímý přístup ke základnímu objektovému modelu formuláře. To znamená, že vývojáři definují formulářové prvky a jejich chování programově v TypeScriptu. Místo definování formuláře přímo v HTML, jak

to činíme u Template-driven forms, zde máme plnou kontrolu nad tím, jak formulář funguje.

Mezi přední přednosti Reactive formulářů patří zejména jejich škálovatelnost. Umožňují konstrukci komplexních formulářů obsahujících rozsáhlý počet prvků, přičemž pro tyto prvky lze definovat sofistikovanou logiku a validaci. Dále se vyznačují vysokou mírou znovupoužitelnosti. Vyvinuté komponenty formulářů lze efektivně využívat opakovaně v různých sekci aplikace, což přispívá k lepší správě kódu a zefektivnění procesu vývoje. Nezanedbatelnou výhodou je rovněž testovatelnost. Reactive formuláře nabízejí pohodlnou platformu pro provádění jak jednotkových, tak i integračních testů. Simulace uživatelských interakcí a ověřování správné funkcionality formuláře jsou prováděny s vysokou mírou efektivity [8].

Nicméně, pro jednoduché formuláře může být použití Reactive forms poněkud příliš masivní. V takových případech může být Template-driven přístup jednodušší a rychlejší [8].

Zde je ukázkový kód implementace Reactive Forms:

```typescript
import { Component } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-reactive-forms-sample',
  templateUrl: './reactive-forms-sample.component.html',
  styleUrls: ['./reactive-forms-sample.component.scss']
})
export class ReactiveFormsSampleComponent {
  LOGIN = 'login';
  EMAIL = 'email';

  formSample: FormGroup = new FormGroup( controls: {
    [this.LOGIN]: new FormControl( value: ''),
    [this.EMAIL]: new FormControl( value: ''),
  })

  submitResult() {
    alert(this.formSample.value.toString());
  }
}
```

Obrázek 2 Reactive forms implementace komponenty. Zdroj: vlastní zpracování

```html
<div class="form-container">
  <h2>Hello User</h2>
  <form class="my-form" [formGroup]="formSample" (ngSubmit)="submitResult()">
    <div>
      <label for="my-login">Login</label>
      <input type="text" id="my-login" [formControlName]="LOGIN">
    </div>
    <div>
      <label for="my-email">Email</label>
      <input type="text" id="my-email" [formControlName]="EMAIL">
    </div>
    <button class="submit-button"
            type="submit"
            [disabled]="!formSample.valid">Submit values</button>
  </form>

  <div class="output-container">
      <div class="output-result">Login output: {{formSample.controls[LOGIN].value}}</div>
      <div class="output-result">Email output: {{formSample.controls[EMAIL].value}}</div>
  </div>
</div>
```

Obrázek 3 Reactive forms implementace šablony. Zdroj: vlastní zpracování

3.3.2 Angular Template-driven forms

Template-driven formuláře nabízí zjednodušený a efektivní způsob tvorby formulářů, ideální pro situace, kdy je potřeba rychle vyvinout formuláře s nekomplexní strukturou a logikou. Tento přístup je založen na využití direktiv pro řízení a manipulaci s

formulářovým objektovým modelem přímo během tvorby. Vývojáři specifikují prvky formuláře přímo v HTML šabloně, aplikují různé Angular direktivy k navázání těchto prvků na Angular komponenty, čímž mohou efektivně a rychle vytvářet formuláře, jež jsou v souladu s návrhem jejich aplikace [12].

Template-driven forms se vyznačují vhodností pro tvorbu jednoduchých formulářů v aplikacích s minimálními požadavky na složitou logiku. Jsou-li potřeby omezeny na formuláře s základními vstupními poli a není vyžadována pokročilá validace či interaktivní funkce, Template-driven přístup umožňuje rychlou a efektivní implementaci [12].

Nicméně, je důležité poznamenat, že Template-driven forms mají své limity, zejména pokud jde o škálovatelnost a složitější logiku. Pro projekty, které vyžadují komplexní validaci, dynamické zobrazení a reakci na uživatelské vstupy, může být vhodnější použít Reactive forms. Reactive forms poskytují vývojářům více kontroly nad formulářem a jsou vhodnější pro složitější scénáře [12].

Zde je ukázkový kód implementace Template-driven forms:

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

class UserAccount {
  login: string;
  email: string

  constructor(login: string, email: string) {
    this.login = login;
    this.email = email;
  }

  static of(login: string, email: string) {
    return new UserAccount(login, email);
  }
}

@Component({
  selector: 'app-template-driven-forms-sample',
  templateUrl: './template-driven-forms-sample.component.html',
  styleUrls: ['./template-driven-forms-sample.component.scss']
})
export class TemplateDrivenFormsSampleComponent {
  model: UserAccount = UserAccount.of( login: 'previousUser',  email: 'previousEmail@mail.com');
  modelOutputHistory: UserAccount[] = [];

  submitResult() {
    this.modelOutputHistory.push({login: this.model.login, email: this.model.email});
  }
}
```

Obrázek 4 Template-driven forms implementace komponenty. Zdroj: vlastní zpracování

```html
<div class="form-container">
  <h2>Hello User</h2>
  <form class="my-form" (ngSubmit)="submitResult()">
    <div>
      <label for="my-login">Login</label>
      <input type="text" id="my-login" [(ngModel)]="model.login" name="login">
    </div>
    <div>
      <label for="my-email">Email</label>
      <input type="text" id="my-email" [(ngModel)]="model.email" name="email">
    </div>
    <button class="submit-button"
            type="submit">Submit values
    </button>
  </form>

  <div class="output-container">
    <ng-container *ngFor="let output of modelOutputHistory; let i = index">
      <h3>{{i + 1}}</h3>
      <div class="output-result">Login output: {{output.login}}</div>
      <div class="output-result">Email output: {{output.email}}</div>
    </ng-container>
  </div>
</div>
```

Obrázek 5 Template-driven forms implementace šablony. Zdroj: vlastní zpracování

3.3.3 React Formik

Formik představuje knihovnu pro React, jejímž cílem je zjednodušit správu formulářů v aplikacích vyvíjených v rámci tohoto rozšířeného JavaScriptového frameworku. Koncipována s důrazem na jednoduchost, výkonnost a škálovatelnost při manipulaci s formuláři, umožňuje Formik vývojářům efektivně vytvářet interaktivní a přívětivé uživatelské formuláře bez potřeby rozsáhlého a repetitivního kódování [14].

Níže jsou uvedeny základní aspekty a funkce, které Formik nabízí:

- **Jednoduchost použití:** Formik je navržen tak, aby byl snadno použitelný a intuitivní. Proces vytvoření formuláře pomocí Formiku je jednoduché a vyžaduje jen minimální množství kódu. Stačí importovat knihovnu, specifikovat strukturu formuláře a jeho komponenty, a Formik se postará o další procesy.
- **Správa stavu formuláře:** Formik poskytuje vývojářům nástroje pro efektivní správu stavu formuláře, včetně monitorování hodnot vstupních polí, ověřování jejich platnosti a zpracování případných chyb. Díky Formiku je možné snadno získat přístup k hodnotám polí, určit validitu formuláře a interagovat s ním.
- **Validace formuláře:** Nabízí jednoduchý, avšak efektivní mechanismus pro validaci formulářů. Je možné definovat validační schémata pro jednotlivá vstupní pole, přičemž Formik automaticky ověřuje jejich splnění a v případě chyby zobrazuje relevantní chybová hlášení.
- **Práce s událostmi formuláře:** Formik usnadňuje zachycení a zpracování událostí spojených s formuláři, jako je odeslání, resetování či jiné akce. Umožňuje definovat vlastní způsoby manipulace s daty formuláře v závislosti na těchto událostech.
- **Podpora pro pole s více hodnotami:** Formik umožňuje práci s vstupními poli, která obsahují více hodnot, což je klíčové pro komplexnější formulářové struktury.
- **Rozšiřitelnost a přizpůsobení:** Díky své flexibilitě a modulární stavbě lze Formik rozšířit a přizpůsobit vytvářením vlastních komponent a funkcí, které lze snadno integrovat.

Vzhledem k těmto charakteristikám je Formik vhodný pro širokou škálu aplikací, od jednoduchých kontaktních formulářů až po složité struktury s mnoha poli a validačními pravidly. Cílem této knihovny je poskytnout uživatelům nástroje pro efektivní a jednoduchou práci s formuláři v prostředí React [13].

```jsx
import React from 'react';
import { useFormik } from 'formik';

const SignupForm = () => {
  // Pass the useFormik() hook initial form values and a submit function that will
  // be called when the form is submitted
  const formik = useFormik({
    initialValues: {
      email: '',
    },
    onSubmit: values => {
      alert(JSON.stringify(values, null, 2));
    },
  });
  return (
    <form onSubmit={formik.handleSubmit}>
      <label htmlFor="email">Email Address</label>
      <input
        id="email"
        name="email"
        type="email"
        onChange={formik.handleChange}
        value={formik.values.email}
      />

      <button type="submit">Submit</button>
    </form>
  );
};
```

Obrázek 6 Formik implementace. Zdroj: [14]

3.3.4 **React Hook Form**

React Hook Form je knihovna pro React, navržená pro zjednodušení správy formulářů. Zaměřuje se na jednoduchost, efektivitu a flexibilitu, umožňuje rychlé vytváření formulářů s minimálním množstvím kódu. Klíčové funkce zahrnují snadnou správu stavu formuláře pomocí hooků, rychlou validaci, optimalizaci výkonu a podporu pro komplexní formulářové struktury. React Hook Form je kompatibilní s různými nástroji a umožňuje rozšíření pro vlastní validátory a komponenty, což z něj dělá vhodnou volbu pro širokou škálu aplikací [16].

Zde jsou klíčové aspekty a funkce, které React Hook Form nabízí:

Jednoduchost použití: Díky hooku **useForm** lze snadno zaregistrovat vstupní pole a ovládat odesílání formuláře s minimálním množstvím kódu. Tato funkcionalita podstatně snižuje složitost při tvorbě a správě formulářů.

Rychlá validace formuláře: Podporuje standardní HTML validace (jako je **required**, **min**, **max**, **minLength**, **maxLength**, **pattern**) a umožňuje definovat vlastní validační funkce. Tím umožňuje efektivní správu validačních pravidel a chybových hlášek.

Optimalizace výkonu: Má vynikající výkon díky minimalizaci re-renderování komponent. Toto chování je zásadní pro zlepšení uživatelského zážitku ve složitějších formulářích s velkým množstvím polí.

Podpora pro pole s více hodnotami: Umožňuje snadno pracovat s poli obsahujícími více hodnot, jako jsou zaškrtávací políčka nebo select boxy, což rozšiřuje možnosti použití knihovny na širokou škálu formulářů.

Rozšiřitelnost a přizpůsobení: React Hook Form je navržen tak, aby byl rozšiřitelný a snadno integrovatelný s existujícími UI komponentami a frameworky, což usnadňuje práci s komplexními formulářovými scénáři a uživatelskými rozhraními.

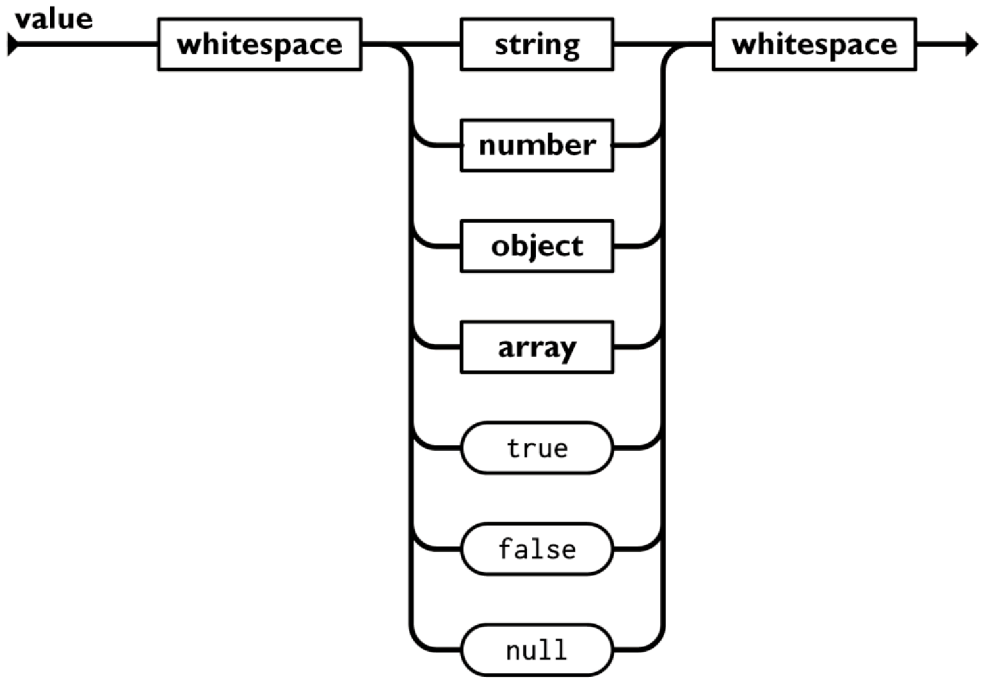
Integrace s jinými komponentami je jednoduchá a lze ji provést pomocí Controller komponenty, která umožňuje použití vstupních polí z jiných knihoven UI, jako je Ant Design [17].

```tsx
enum GenderEnum {
  female = "female",
  male = "male",
  other = "other"
}

interface IFormInput {
  firstName: String;
  gender: GenderEnum;
}

export default function App() {
  const { register, handleSubmit } = useForm<IFormInput>();
  const onSubmit = (data: IFormInput) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label>First Name</label>
      <input {...register("firstName")} />
      <label>Gender Selection</label>
      <select {...register("gender")}>
        <option value="female">female</option>
        <option value="male">male</option>
        <option value="other">other</option>
      </select>
      <input type="submit" />
    </form>
  );
}
```

Obrázek 7 React Hook Form implementace. Zdroj: [17]

3.4 JSON

JSON neboli JavaScript Object Notation, je lehký textově založený otevřený standard navržený pro výměnu dat čitelných člověkem. Konvence, které JSON využívá, jsou programátorům známé, včetně těch, kteří mají znalosti v C, C++, Java, Python, Perl atd. Formát byl specifikován Douglasem Crockfordem. Byl navržen pro výměnu dat čitelných člověkem. Přípona názvu souboru je „.json“. Internetový mediální typ JSON je application/json. JSON je snadno čitelný a zapisovatelný. JSON je nezávislý na programovacím jazyku [18].



Obrázek 8 JSON ukázka syntaxe. Zdroj: [19]

Původně vznikl jako součást JavaScriptu, což mu dodává blízkou vazbu na tento skriptovací jazyk. Nicméně JSON je nezávislý na programovacím jazyce a může být používán v různých prostředích a platformách. Díky tomu lze data uložená ve formátu JSON snadno přenášet mezi různými aplikacemi a systémy [18].

JSON formát je založen na kolekcích párů atribut–hodnota, což znamená, že data jsou strukturována do objektů a pole. Toto uspořádání umožňuje reprezentovat složité datové struktury, což je zvlášť užitečné při přenosu informací mezi klientem a serverem [20].

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
```

Obrázek 9 JSON ukázka kódu. Zdroj: [21]

V kontextu vývoje formulářů, jak je zmíněno v rámci experimentu, může být JSON využit jako šablona pro definici struktury a vlastností jednotlivých ovládacích prvků. Tímto způsobem lze snadno konfigurovat a upravovat formuláře bez nutnosti manuální úpravy kódu. Každý prvek formuláře může být reprezentován objektem s atributy, které určují jeho chování, vzhled a validaci.

3.5 Unit testování

Unit testování je součet akcí, které se odehrávají mezi vyvoláním veřejné metody v systému a jedním patrným výsledkem, který je testem tohoto systému zjištěn. Patrný konečný výsledek lze pozorovat bez zkoumání vnitřního stavu systému, pouze prostřednictvím jeho veřejných API a chování. Konečný výsledek může být některý z následujících:

Vyvolaná veřejná metoda vrátí hodnotu (funkci, která není prázdná).

Dojde k patrné změně stavu nebo chování systému před a po vyvolání, kterou lze určit bez zkoumání soukromého stavu. (Příklady: systém může přihlásit dříve neexistujícího uživatele, nebo se změní vlastnosti systému, pokud je systém stavovým automatem.)

V případě, kdy je provedeno volání do systému třetí strany, nad nímž testovací proces nevykonává kontrolu, a tento externí systém nevrací žádnou hodnotu, anebo veškeré potenciálně vrácené hodnoty nejsou během dalšího postupu zohledňovány. Tento scénář může nastat například při integraci s logovacím systémem poskytovaným třetí stranou, jenž

nebyl vyvinut v rámci vlastní organizace a ke zdrojovému kódu tohoto systému neexistuje přístup. [22].

V rámci experimentu v praktické časti budeme používat Jasmine a Jest. Jasmine pro Angular a Jest pro React respektive.

Jest je open-source testovací framework založený na JavaScriptu, který je primárně navržený pro práci s webovými aplikacemi založenými na React a React Native. Často nejsou unit testy při běhu na frontendu jakéhokoli softwaru příliš užitečné. Je to hlavně proto, že unit testy pro frontend vyžadují rozsáhlou a časově náročnou konfiguraci. S frameworkem Jest lze tuto složitost výrazně snížit [23].

```
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

Obrázek 10 Jest funkce pro testování. Zdroj: [26]

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Obrázek 11 Jest ukázka testovacího scénáře. Zdroj: [26]

Jasmine je framework pro behavior-driven development (BDD) JavaScriptových aplikací, který vývojářům umožňuje psát testy specifikující očekávané chování kódu. S Jasmine, vývojáři definují "specifikace" pomocí funkcí describe, která agrupuje jednotlivé testy, a it, která popisuje očekávané chování v konkrétním testovacím případě [29].

Framework poskytuje sadu "matcherů", což jsou funkce, jež umožňují vyjádřit očekávané výsledky testů, například zda hodnoty splňují určité podmínky. Jasmine také podporuje testování asynchronního kódu, což je důležité pro JavaScriptové aplikace, které často pracují s asynchronními operacemi jako jsou API volání.

Kromě toho je Jasmine nezávislý na jiných JavaScriptových knihovnách a lze ho použít v různých prostředích a technologiích, což z něj činí flexibilní nástroj pro vývojáře. Jasmine je populární díky své jednoduchosti a snadnému nasazení, což vývojářům umožňuje rychle začít psát testy bez potřeby složité konfigurace [24].

```
describe("A suite is just a function", function() {
  let a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Obrázek 12 Jasmine ukázka testovacího scénáře. Zdroj: [27]

4 Vlastní práce

Empirická část práce zpracovává témata, která souvisí s vývojem aplikací obsahující formuláře. Pro vývoj formulářů použijeme Reactive Forms, Template-driven forms v Angular a Formik, React Hook form knihovny v React. Pro testovácí podle stanovených kriterií využijeme Jasmine v Angular a Jest v React. Nejprve stanovíme kritérii, podle kterých budeme hodnotit výkonnost vyvinutých formulářů pomoci nástrojů pro jejich vytváření a nastavení konfiguraci pro formuláře, následně implementace a testování.

4.1 Stanovení kritérií

Stanovení kritérií pro srovnání výkonnosti formulářů ve webových aplikacích je klíčovým krokem při zajišťování efektivní a uživatelsky přívětivé interakce. V této kapitole se zaměříme na tři základní kritéria, která lze použít k hodnocení a porovnání výkonu formulářů ve webových aplikacích:

1. **Měření doby načtení formuláře**: Toto kritérium zahrnuje měření času potřebného od momentu požadavku uživatele na načtení formuláře do jeho plného zobrazení a připravenosti k interakci. Doba načtení formuláře může být ovlivněna různými faktory, včetně velikosti formuláře, počtu a typu vstupních polí, a efektivity kódu. Optimalizace pro rychlé načítání je zásadní, protože uživatelé očekávají rychlé reakce, a dlouhé načítání může vést k frustraci a odchodu z webu.

2. **Měření doby reakce na uživatelské vstupy**: Zde je cílem posoudit, jak rychle formulář reaguje na akce uživatele, jako je vstup do textového pole, výběr z rozbalovacího seznamu, nebo kliknutí na tlačítko. Rychlá odezva zvyšuje plynulost interakce a přispívá k lepší celkové uživatelské zkušenosti. Výkonnost při reakci na uživatelské vstupy může být optimalizována pomocí efektivního kódu a minimalizací zpracování na straně klienta.

3. **Měření doby reakce na dynamické přidávání a odebírání prvků formuláře**: Toto kritérium se zaměřuje na schopnost formuláře flexibilně reagovat na změny, jako je přidání nebo odebrání vstupních polí, bez negativního vlivu na výkon. To

je obzvláště důležité v dynamických formulářích, které se mění na základě uživatelských akcí. Přidávání a odebírání prvků by mělo být hladké a bez zbytečného zdržení, aby uživatelé neměli pocit zpomalení nebo přerušení práce s formulářem.

4.2 Datová struktura konfigurace

Důvodem pro využití konfigurace skrze rozhraní (interface) v TypeScriptu je poskytnout přesnou specifikaci struktur dat, které vaše aplikace používá. V TypeScriptu rozhraní definují tvar objektů, což zahrnuje jak typy, tak očekávané klíče. Vývojáři tak mohou sestavit striktně typované konfigurace, které umožňují chytrou nápovědu a kontrolu typů při kompilaci. To vede ke zvýšení efektivity práce a redukci chyb, protože TypeScript už ve fázi vývoje upozorní na nesrovnalosti v datech a jejich používání.

V kontextu konfigurace formulářů to znamená, že každá kontrola formuláře má pevně stanovený typ, očekávané vlastnosti a metody. To usnadňuje práci s formuláři, protože vývojáři mohou snadno rozšířit nebo modifikovat formuláře s jistotou, že všechny potřebné vlastnosti a validace jsou na svém místě a správně nastaveny. TypeScript tedy hraje klíčovou roli v zajištění, že komponenty formulářů a jejich interakce jsou dobře definované a správně typované

Začněme vytvořením výčtového typu (enum) pro různé typy polí, což umožní znovupoužitelnost kódu. V případě jakékoli změny bude nutné upravit hodnotu pouze v tomto souboru.

```
export enum QuestionType {
  TEXT = 'text',
  NUMBER = 'number',
  CHECKBOX = 'checkbox',
  EMAIL = 'email',
  PASSWORD = 'password',
  SELECT = 'select',
}
```

Obrázek 13 enum pro typy otázek. Zdroj: vlastní zpracování

Dál vytvoříme rozhraní, které bude popisovat strukturu objektu, který nám přichází ze strany serveru.

```
export interface FormConfig {
  fields: FormControlConfig[];
}
```

Obrázek 14 Rozhraní pro FormConfig. Zdroj: vlastní zpracování

FormConfig zahrnuje atribut „fields“, který může obsahovat více instancí FormControlConfig, popisujících rozhraní pro jednotlivá pole.

```
export interface FormControlConfig {
  type: QuestionType;
  name: string;
  label: string;
  placeholder?: string;
  validations?: Validations;
  options?: QuestionOptions;
  checked?: boolean;
}
```

Obrázek 15 Rozhraní pro FormControlConfig. Zdroj: vlastní zpracování

V tomto rozhraní představují „type“, „name“ a „label“ základní atributy. Atribut „type“ specifikuje typ pole, „name“ slouží jako jedinečný identifikátor daného pole a „label“ poskytuje popis pole. Tyto atributy jsou základní a vyžadované pro definici každého pole. Na rozdíl od „type“, „name“ a „label“, ostatní atributy nejsou povinné a mohou proto přijímat hodnotu „undefined“.

```
export interface QuestionOptions {
  value: string;
  display: string;
  defaultSelected?: boolean;
}
```

Obrázek 16 Rozhraní pro QuestionOptions. Zdroj: vlastní zpracování

QuestionOptions je rozhraní pro volitelný atribut „options“ ve FormControlConfig, specificky navržené pro typ pole „select“, kde uživatel vybírá z několika možností. Atribut „value“ funguje jako klíč, zatímco „display“ představuje text, který vidí uživatel. Atribut „defaultSelect“ je volitelná položka určená pro přednastavení hodnoty.

```
export interface Validations {
  required: boolean;
  isEmail?: boolean;
  inputRestrictions?: InputRestrictions;
}
```

Obrázek 17 Rozhraní pro Validations. Zdroj: vlastní zpracování

Validations je rozhraní určené pro validaci jednotlivých polí. Klíčovým atributem je „required“, který zajišťuje, že pole musí být vyplněno před jeho odesláním.

```
export interface InputRestrictions {
  maxLength?: number;
  minLength?: number;
  min?: number;
  max?: number;
}
```

Obrázek 18 Rozhraní pro InputRestrictions. Zdroj: vlastní zpracování

InputRestrictions je rozhraní pro dodatečné validace. „maxLength“ a „minLength“ omezuje vstup uživatelé na minimální a maximální počet charakterů. „min“ a „max“ – číselná omezení.

Podle definované struktury vytvoříme objekt, jejíhož data budou přijaty klientem a zobrazené dynamicky.

```
        {
  "fields": [
    {
      "type": "text",
      "name": "name",
      "label": "Jméno",
      "placeholder": "Zadejte své jméno",
      "validations": {
        "required": true,
        "inputRestrictions": {
          "minLength": 3
        }
      }
    },
    {
      "type": "number",
      "name": "age",
      "label": "Věk",
      "placeholder": "Zadejte svůj věk",
      "validations": {
        "required": true,
        "inputRestrictions": {
          "minLength": 18,
          "maxLength": 100
        }
      }
    },
    {
      "type": "email",
      "name": "email",
```

```
  "label": "Email",
  "placeholder": "Zadejte svůj email",
  "validations": {
    "required": true,
    "isEmail": true
  }
},
{
  "type": "password",
  "name": "password",
  "label": "Heslo",
  "placeholder": "Zadejte heslo",
  "validations": {
    "required": true,
    "inputRestrictions": {
      "minLength": 8
    }
  }
},
{
  "type": "select",
  "name": "favoriteColor",
  "label": "Oblíbená barva",
  "options": [
    {"value": "red", "display": "Červená"},
    {"value": "blue", "display": "Modrá"},
    {"value": "green", "display": "Zelená"}
  ],
  "validations": {
    "required": true
  }
},
```

```
    {
      "type": "checkbox",
      "name": "acceptTerms",
      "label": "Souhlasím s podmínkami",
      "validations": {
        "required": true
      }
    }
  ]
}
```

4.3 Implementace formulářů

4.3.1 Angular Reactive Forms

Ve vývoji aplikací pomocí Angular je jedním z klíčových prvků práce s formuláři způsob, jakým jsou data formulářů přenášena a spravována. Zde hraje důležitou roli FormTransportService, služba, která zajišťuje komunikaci mezi klientem a serverem. Pomocí injektovaného HttpClient je schopna získávat konfiguraci formuláře a data přes HTTP požadavky. Tento service může být v Angularu poskytnut "root", což zajišťuje jeho dostupnost napříč celou aplikací.

```typescript
@Injectable({
  providedIn: 'root'
})
export class FormTransportService {
  static FORM = 'form';
  static DATA = 'data';
  static PATCH = 'patch';
  private readonly localApi = 'http://localhost:4200';

  constructor(private readonly http: HttpClient) {
  }

  getFormData(): Observable<FormConfig> {
    const request = `${ this.localApi }/${ FormTransportService.FORM }/${ FormTransportService.DATA }`;
    return this.http.get<FormConfig>(request);
  }
}
```

Obrázek 19 FormTransportService implementace. Zdroj: vlastní zpracování

V komponentě Angular, která používá reaktivní formuláře, se veškerá inicializace a logika správy formulářů obvykle odehrává v různých metodách a životních cyklech komponenty.

```
constructor(
  private readonly formTransportService: FormTransportService,
  private readonly fb: FormBuilder
) {
}
```

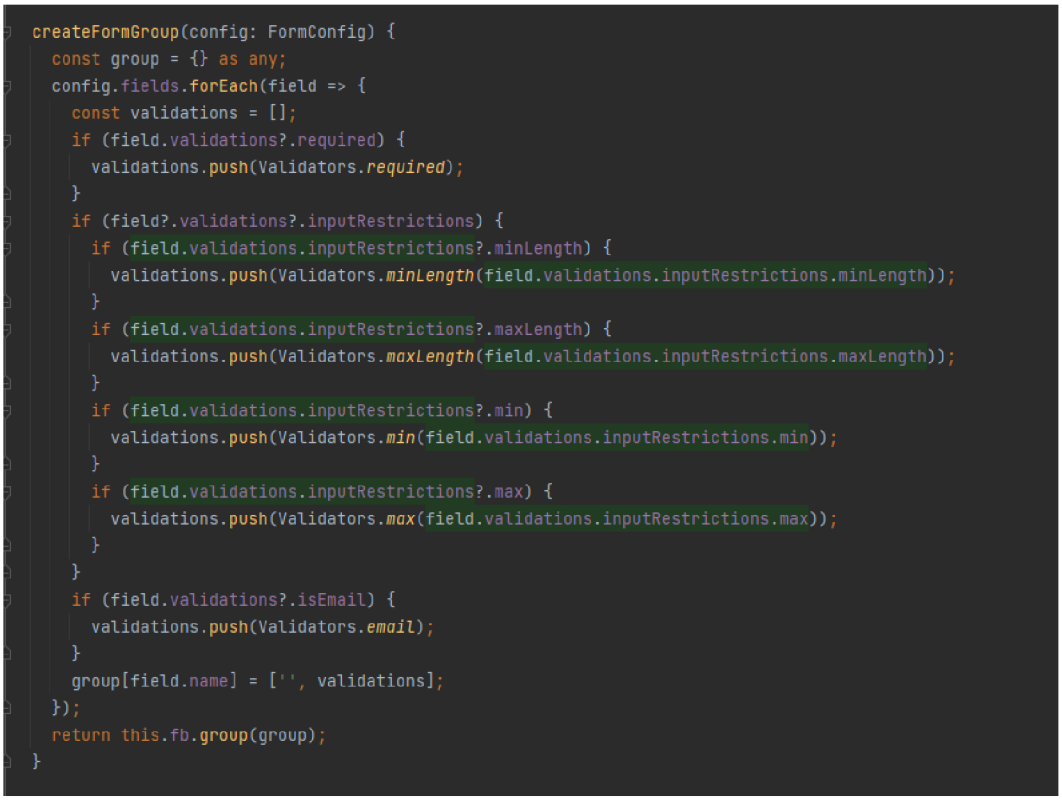
Obrázek 20 ReactiveFormsSampleComponent konstruktor. Zdroj: vlastní zpracování

Konstruktor třídy ReactiveFormsSampleComponent je určen pro injektování závislostí potřebných pro práci s formuláři a komunikaci s API. V tomto případě jsou injektovány dvě služby: FormTransportService pro získání konfigurace formuláře z backendu a FormBuilder pro programové vytváření instancí formulářů v Angularu. Tyto služby jsou uloženy do privátních vlastností třídy a jsou přístupné z ostatních metod třídy.

```
ngOnInit() {
  this.formTransportService.getFormData().subscribe( observerOrNext: (config: FormConfig) => {
    this.config = config;
    this.form = this.createFormGroup(config)
  })
}
```

Obrázek 21 ngOnInit načtení dat ze serveru. Zdroj: vlastní zpracování

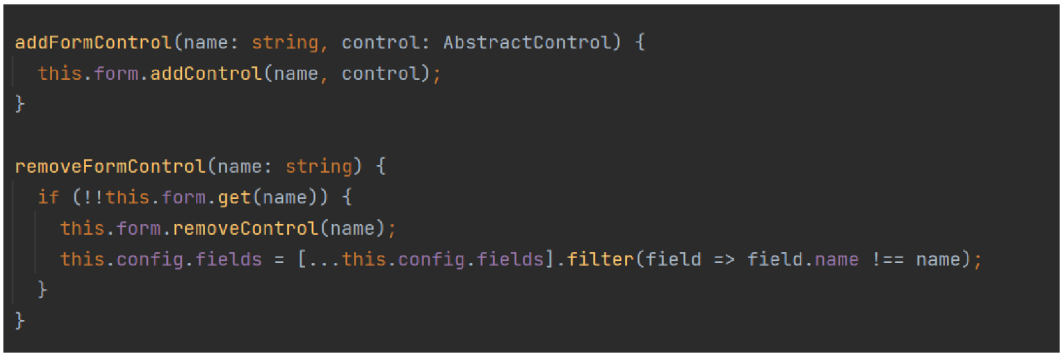
Metoda ngOnInit je životní cyklus hook volaný ihned po konstruktoru, který se používá k inicializaci komponenty. V tomto případě získává data konfigurace formuláře pomocí formTransportService a poté používá tyto data k vytvoření nové skupiny formulářů voláním createFormGroup. Tím se zajistí, že jakmile je komponenta připravena a data jsou dostupná, formulář je ihned vytvořen a připraven k použití.



```
createFormGroup(config: FormConfig) {
  const group = {} as any;
  config.fields.forEach(field => {
    const validations = [];
    if (field.validations?.required) {
      validations.push(Validators.required);
    }
    if (field?.validations?.inputRestrictions) {
      if (field.validations.inputRestrictions?.minLength) {
        validations.push(Validators.minLength(field.validations.inputRestrictions.minLength));
      }
      if (field.validations.inputRestrictions?.maxLength) {
        validations.push(Validators.maxLength(field.validations.inputRestrictions.maxLength));
      }
      if (field.validations.inputRestrictions?.min) {
        validations.push(Validators.min(field.validations.inputRestrictions.min));
      }
      if (field.validations.inputRestrictions?.max) {
        validations.push(Validators.max(field.validations.inputRestrictions.max));
      }
    }
    if (field.validations?.isEmail) {
      validations.push(Validators.email);
    }
    group[field.name] = ['', validations];
  });
  return this.fb.group(group);
}
```

Obrázek 22 ReactiveFormsSample implementace funkce pro validace. Zdroj: vlastní zpracování

createFormGroup je metoda, která přijímá konfiguraci formuláře jako argument a vytváří dynamicky FormGroup na základě této konfigurace. Pro každé pole ve formuláři nastavuje výchozí hodnoty a validátory podle definovaných pravidel ve FormConfig. Toto umožňuje flexibilní vytváření formulářů podle externě definované konfigurace.



```
addFormControl(name: string, control: AbstractControl) {
  this.form.addControl(name, control);
}

removeFormControl(name: string) {
  if (!!this.form.get(name)) {
    this.form.removeControl(name);
    this.config.fields = [...this.config.fields].filter(field => field.name !== name);
  }
}
```

Obrázek 23 ReactiveFormsSample přidaní a odebírání prvků formuláře. Zdroj: vlastní zpracování

Tyto metody umožňují dynamické přidávání a odebírání kontrol z instance FormGroup. addFormControl přidává novou kontrolu formuláře do skupiny a aktualizuje konfiguraci formuláře, zatímco removeFormControl odebírá existující kontrolu a

aktualizuje konfiguraci tak, aby odrážela tuto změnu. Tyto operace jsou užitečné pro formuláře, jejichž struktura se může za běhu aplikace měnit.

```
<ng-container *ngIf="form">
  <form class="my-form" [formGroup]="form" (ngSubmit)="submitResult()">
    <ng-container *ngIf="config?.fields">
      <ng-container *ngFor="let field of config?.fields">
        <div
          class="field-container"
          *ngIf="[QuestionType.TEXT, QuestionType.NUMBER,QuestionType.EMAIL,QuestionType.PASSWORD].includes(field.type)"
        >
          <label [for]="field.name">{{field.label}}</label>
          <input [ngClass]="{'invalid': isInvalidFormControl(field.name)}" [id]="field.name"
                 [type]="field.type"
                 [formControlName]="field.name"
                 [placeholder]="field.placeholder"
          >
        </div>

        <div class="field-container" *ngIf="field.type === 'select'">
          <label [for]="field.name">{{field.label}}</label>
          <select [ngClass]="{'invalid': isInvalidFormControl(field.name)}"
                  [id]="field.name" [formControlName]="field.name">
            <ng-container *ngFor="let option of field.options">
              <option [value]="option.value">{{option.display}}</option>
            </ng-container>
          </select>
        </div>

        <div class="field-container" *ngIf="field.type === 'checkbox'">
          <label [for]="field.name">{{field.label}}</label>
          <input [ngClass]="{'invalid': isInvalidFormControl(field.name)}" [id]="field.name"
                 [type]="field.type"
                 [formControlName]="field.name"
                 [placeholder]="field.placeholder"
          >
        </div>
      </ng-container>
    </ng-container>
    <ng-container...>
```

Obrázek 24 ReactiveFormSample implementace šablony. Zdroj: vlastní zpracování

V poskytnuté šabloně je [formGroup] použita pro přiřazení instance FormGroup, vytvořené v komponentě, k elementu <form> ve šabloně. To umožňuje všem vloženým FormControl elementům (např. <input>, <select>) komunikovat se svými odpovídajícími FormControl instancemi definovanými ve FormGroup. Díky tomu může být celý formulář spravován jako koherentní jednotka s centralizovanou logikou pro zpracování dat formuláře a validaci.

V rámci formuláře <form> se děje iterace přes pole „config?.fields“ pomocí *ngFor, kde každé pole formuláře definované v FormConfig se dynamicky zpracovává.

Pro různé typy vstupů („text“, „number“, „email“, „password“, „select“, „checkbox“) se využívají podmíněné bloky „*ngIf“ k určení, jaký druh HTML elementu bude pro dané pole použit. Toto rozhodnutí závisí na „field.type“

4.3.2 Angular Template-driven forms

Na rozdíl od reactive formulářů, kde je struktura formuláře vytvořena programově v TypeScriptu, template-driven formuláře definují strukturu přímo v HTML šabloně a používají direktivy Angularu pro vazbu dat a validaci.

```
constructor(
  private readonly formTransportService: FormTransportService,
  private readonly cdr: ChangeDetectorRef
) {
}
```

Obrázek 25 Template-driven forms konstruktor. Zdroj: vlastní zpracování

V konstruktoru jsou injektovány dvě služby: **formTransportService** pro získání dat formuláře a **ChangeDetectorRef** pro ruční spouštění detekce změn. Tato kombinace služeb je typická pro komponenty, které komunikují s externím zdrojem dat a potřebují reflektovat dynamické změny v šabloně.

```
ngOnInit() {
  this.formTransportService.getFormData()
    .subscribe( observerOrNext: (data : FormConfig ) => this.model = data);
}
```

Obrázek 26 Template-driven forms načtení dat ze serveru. Zdroj: vlastní zpracování

Metoda ngOnInit zajišťuje inicializační logiku. V rámci této metody dochází k volání **formTransportService.getFormData()** pro získání konfigurace formuláře, která se následně přiřazuje k vlastnosti **model**. Toto načítání probíhá asynchronně a výsledná data definují model pro Template-driven forms.

```
addControl(control: FormControlConfig) {
  this.model.fields.push(control);
  this.cdr.detectChanges();
}

removeControl(name: string) {
  this.model.fields = [...this.model.fields].filter(control => control.name !== name);
  this.cdr.detectChanges();
}
```

Obrázek 27 Template-driven forms přidání a odebírání prvků formuláře. Zdroj: vlastní zpracování

V přístupu Template-driven formulářů, na rozdíl od ReactiveForms, nedochází k přímé manipulaci s instancemi **FormGroup** nebo **FormControl** v TypeScript kódu. Místo toho je celá struktura formuláře a její logika definována přímo ve šabloně HTML pomocí direktiv, jako je **ngModel**.

Když metoda **addControl** v Template-driven přístupu "přidává" nový **FormControl**, ve skutečnosti nezařazuje nový **FormControl** do instance **FormGroup**, jelikož taková struktura explicitně v šabloně neexistuje. Namísto toho přidává konfigurační objekt do modelu, který je využit ve šabloně. Tento způsob je méně přímý a vyžaduje manuální aktualizace šablony a modelu.

Opačný proces, metoda **removeControl**, odebírá konfiguraci kontrolního prvku z modelu, čímž implicitně odstraňuje prvek z formuláře definovaného ve šabloně. Změny se poté v šabloně projeví díky mechanismu detekce změn Angularu.

```
<form #form="ngForm" class="my-form" (ngSubmit)="submitResult(form)">
  <ng-container>
    <ng-container *ngFor="let field of model?.fields; let i = index">
      <div
        class="field-container"
        *ngIf="[QuestionType.TEXT, QuestionType.NUMBER, QuestionType.EMAIL,QuestionType.PASSWORD].includes(field.type)"
      >
        <label for="{{field.name}}_{{i}}">{{field.label}}</label>
        <input [ngClass]="{'invalid': isInvalidFormControl(field.name, form)}"
               ngModel
               name="{{field.name}}"
               id="{{field.name}}_{{i}}"
               [type]="field.type"
               [placeholder]="field.placeholder"
               [required]="field?.['validations']?.required"
        >
      </div>
      <div class="field-container" *ngIf="field.type === 'select'">
        <label for="{{field.name}}_{{i}}">{{field.label}}</label>
        <select [ngClass]="{'invalid': isInvalidFormControl(field.name, form)}"
                ngModel
                name="{{field.name}}"
                id="{{field.name}}_{{i}}"
                [required]="field.['validations']?.required"
        >
          <ng-container *ngFor="let option of field.options">
            <option [style]="" [value]="option.value">{{option.display}}</option>
          </ng-container>
        </select>
      </div>
```

Obrázek 28 Template-driven forms implementace šablony. Zdroj: vlastní zpracování

Použití atributu **#form="ngForm"** v tagu **<form>** vytváří lokální proměnnou **form**, obsahující odkaz na instanci objektu formuláře. Direktiva **NgForm** od Angularu umožňuje monitorování celkového stavu formuláře, včetně validace a stavů všech vstupních polí.

Vložením **ngModel** do prvků formuláře, jako jsou **<input>** elementy, a definováním jména pomocí atributu **name** se vytváří dvoucestná vazba mezi modelem a vstupním polem. Tato vazba zajišťuje, že změny v modelu okamžitě ovlivňují hodnotu vstupního pole a změny provedené v poli uživatelem se okamžitě odrazí v modelu.

4.3.3 React Formik

Ve srovnání s Reactive Forms poskytují Formik a jeho template-driven přístup v React aplikacích více "deklarativní" metodu tvorby a správy formulářů, přičemž řada logiky a validace je delegována přímo na knihovnu nebo framework. Na druhé straně, Reactive Forms v Angularu nabízejí vývojářům rozšířenou kontrolu nad formuláři díky programovému přístupu, kde je veškerá logika a validace implementována v TypeScriptu.

Jako první krok vytvoříme komponentu **FormikSample** a využijeme vlastní hook **useFormTransport** pro načtení konfiguračních dat z serveru.

```
const FormikSample = () => {
    const { formData, getFormData } = useFormTransport();
    const fields = formData ?? {};
```

Obrázek 29 Formik načtení dat ze serveru. Zdroj: vlastní zpracování

Dalším krokem je vytváření validaci pro každý prvkem, pokud validační pravidla jsou skutečné v objektu.

```
const validationSchema = Yup.object().shape(generateValidationSchema( fields: fields.fields || []));
```

Obrázek 30 Formik vytvaření schématu pro validace. Zdroj: vlastní zpracování

```
const generateValidationSchema = (fields: FormControlConfig[]) => {
    const schema: { [key: string]: StringSchema | NumberSchema } = {};
    fields.forEach(field => {
        let validator = Yup.string();
        let numberValidator = Yup.number();
        if (field?.validations?.required) {
            validator = validator.required( msg: 'Toto pole je povinné');
        }
        if ([QuestionType.NUMBER].includes(field.type)) {
            if (field?.validations?.inputRestrictions) {
                if (field?.validations?.inputRestrictions?.min) {
                    numberValidator = numberValidator.min(field?.validations?.inputRestrictions?.min,
                        message: `Minimální číslo je ${ field?.validations?.inputRestrictions?.min }`);
                }
                if (field?.validations?.inputRestrictions?.max) {
                    numberValidator = numberValidator.max(field?.validations?.inputRestrictions?.max,
                        message: `Maximální číslo je ${ field?.validations?.inputRestrictions?.max }`);
                }
            }
            schema[field.name] = numberValidator;
        } else {
            if (field?.validations?.inputRestrictions) {
                if (field?.validations?.inputRestrictions?.minLength) {
                    validator = validator.min(field?.validations?.inputRestrictions?.minLength,
                        message: `Minimální délka je ${ field?.validations?.inputRestrictions?.minLength }`);
                }
                if (field?.validations?.inputRestrictions?.maxLength) {
                    validator = validator.max(field?.validations?.inputRestrictions?.maxLength,
                        message: `Maximální délka je ${ field?.validations?.inputRestrictions?.maxLength }`);
                }
                schema[field.name] = validator;
            }
        }
    });
    return schema;
};
```

Obrázek 31 Formik implementace funkce pro validace. Zdroj: vlastní zpracování

ValidationSchema je schéma definované pomocí knihovny Yup, které slouží k určení pravidel validace pro formuláře spravované knihovnou Formik. Funkce **generateValidationSchema** dynamicky vytváří toto validační schéma založené na konfiguraci formulářových polí (**fields**). Tento postup umožňuje přesně specifikovat validační pravidla pro různé typy vstupů, jako jsou textová nebo číselná pole, a definovat jejich omezení, například označení pole jako povinné, stanovení minimální nebo maximální délky, či hodnoty.

```
<Formik
    initialValues={ {...initialValues, dynamicFields: []} }
    validationSchema={ validationSchema }
    onSubmit={ (values : FormikValues ) => {
        alert(values);
    } }
>
```

Obrázek 32 Formik implementace komponenty <Formik />. Zdroj: vlastní zpracování

Implementaci dynamického formuláře začneme z využití komponenty Formik importovanou z knihovny.

Obrázek 33 Formik iterace skrz pole prvků formuláře. Zdroj: vlastní zpracování

Formik poskytuje objekt **formikProps** funkci render, obvykle prostřednictvím anonymní funkce, která vrací komponentu TSX **<Form>**. Tato komponenta **<Form>**, specifická pro knihovnu Formik, je navržena k automatickému zpracování procesu odesílání formuláře. V rámci těla komponenty **<Form>** probíhá iterace nad polem **fields.fields**, jež je definováno externě jako konfigurace polí formuláře, pomocí metody **map**. Během této iterace je pro každé specifikované pole vykreslena komponenta **<DynamicField>**. Komponenta **<DynamicField>** je připravena k zpracování různorodých typů vstupů formuláře, včetně textových a výběrových polí. Pro spojení hodnoty každého z těchto polí s odpovídající hodnotou v objektu **formikProps** se využívá **formikProps.values[field.name]**. Tímto způsobem je zajištěna dvoucestná vazba mezi stavem jednotlivých formulářových polí a celkovým stavem formuláře.

Obrázek 34 Formik implementace dynamického zobrazení prvků formuláře. Zdroj: vlastní zpracování

Komponenta **DynamicField** v Reactu je navržena pro flexibilní zobrazování různých typů polí formuláře podle zadaných props. Řídí se typem pole (**type**), který určuje, jaké vstupní pole bude zobrazeno. Props jako **name**, **placeholder**, **value** a **options** slouží k nastavení vlastností jednotlivých polí. Pro integraci s Formik formuláři využívá komponentu **<Field>** z knihovny Formik.

- Switch výraz vyhodnotí **type** pole a podle jeho hodnoty určí, jaký typ formulářového pole se má zobrazit.
- U typů jako Číslo, E-mail, Heslo, Text se použije standardní komponenta **<Field>** s vlastnostnmi **name**, **type**, **placeholder** a **value**. Tyto atributy specifikují základní charakteristiky pole, například jeho jméno pro identifikaci ve formuláři, typ (jako email nebo password), placeholder text a aktuální hodnotu.
- Pro typ Výběr (select) se **<Field>** komponenta použije s atributem **as="select"**, čímž se **<Field>** renderuje jako element select. V rámci tohoto select elementu se

iteruje přes props **options** pro vytvoření jednotlivých option elementů, kde **value** a zobrazovaný text optionů jsou určeny podle zadaných **options**.

- U Zaškrtávacího pole (checkbox) se rovněž využívá **<Field>**, tentokrát s **type="checkbox"**. Konkrétní konfigurace pro checkbox nejsou detailně popsány, ale mohou být přizpůsobeny stejně jako u jiných typů polí.
- Pokud **type** neodpovídá žádné z předdefinovaných možností, switch výraz vrátí **null**, což indikuje, že se žádné pole nevykreslí.

```
<FieldArray
    name="dynamicFields"
    render={ arrayHelpers => (
        <div>
            { formikProps.values.dynamicFields && formikProps.values.dynamicFields.length > 0 && (
                formikProps.values.dynamicFields.map((field: FormControlConfig, index: number) => (
                    <div key={ `${ index }_dynamic` } className="field-container">
                        <label htmlFor={ field!.name }>{ field?.label }</label>
                        <DynamicField type={ field.type } name={ field.name }
                                      placeholder={ field.placeholder }
                                      value={ formikProps.values[field.name] }
                                      options={ field?.options ?? [] }/>
                        { formikProps.errors[field.name] && formikProps.touched[field.name] &&
                            <ErrorMessage name={ field.name }/> }
                    </div>
                ))
            ) }
            <button
                type="button"
                data-testid="add-button"
                className="button add"
                onClick={ () => {
                    arrayHelpers.push(additionalControl)
                } }
            >
                Add Control
            </button>
            <button
                type="button"
                data-testid="remove-button"
                className="button remove"
                onClick={ () => arrayHelpers.pop() }
            >
                Remove Control
            </button>
        </div>
```

Obrázek 35 Formik dynamické přidávání a odebírání prvků formuláře. Zdroj: vlastní zpracování

Komponenta **<FieldArray>** z Formiku poskytuje možnost dynamického přidávání a odebírání prvků formuláře v poli hodnot, označovaném zde jako **dynamicFields**. Akce jako přidání nového prvku nebo odstranění stávajícího se provádějí pomocí tlačítek "Add Control" a "Remove Control", které aktivují metody **push** a **pop** na **arrayHelpers**. Tyto

metody jsou k dispozici díky komponentě **<FieldArray>** a umožňují efektivní manipulaci s datovým polem.

4.3.4 React Hook Form

React Hook Form, oblíbená knihovna pro tvorbu formulářů v Reactu, zahajuje implementaci podobně jako komponenta Formik. Klíčový rozdíl u React Hook Form spočívá v možnosti přímé registrace formulářů.

```
const {control, handleSubmit, register, formState: {errors}} = useForm();
```

Obrázek 36 React Hook Form inicializace useForm hooku. Zdroj: vlastní zpracování

Hook **useForm** z React Hook Form poskytuje funkce pro manipulaci s formulářem, jako jsou **control** pro správu polí, **handleSubmit** pro zpracování odeslání formuláře, **register** pro registraci polí a **errors** pro monitorování validačních chyb.

```
<form onSubmit={ handleSubmit(onSubmit) }>
    { formFields?.fields?.map((field : FormControlConfig , index : number ) => {
        switch (field.type) {
            case 'text':
            case 'number':
            case 'email':
            case 'password': {
                return (
                    <div key={ index }>
                        <label htmlFor={ field.name }>{ field.label }</label>
                        <input
                            type={ field.type }
                            id={ field.name }
                            data-testid={ `test_${ field.name }` }
                            placeholder={ field.placeholder }
                            { ...register(field.name,  options: {
                                required: field?.validations?.required,
                                min: field?.validations?.inputRestrictions?.min,
                                max: field?.validations?.inputRestrictions?.max,
                                minLength: field?.validations?.inputRestrictions?.minLength,
                            }) }
                        />
                        { errors[field.name]?.type === 'required' && <p>{ field.label } is required</p> }
                        { errors[field.name]?.type === 'minLength' && <p>min length
                            of { field.label } is { field.validations?.inputRestrictions?.minLength }</p> }
                        { errors[field.name]?.type === 'min' && <p>min number
                            of { field.label } is { field.validations?.inputRestrictions?.min }</p> }
                        { errors[field.name]?.type === 'max' && <p>max number
                            of { field.label } is { field.validations?.inputRestrictions?.max }</p> }
                    </div>
                );
            }
```

Obrázek 37 React Hook Form iterace skrz pole prvků formuláře. Zdroj: vlastní zpracování

Na rozdíl od Formiku, React Hook Form využívá funkci **register** k propojení vstupních polí s managementem formuláře. Tato funkce se aplikuje na každé pole formuláře pomocí spread operátoru **{...register(...)}**, což umožňuje deklarativně definovat validační pravidla pro jednotlivá pole.

Validace a řízení chyb je zabudované přímo do hooků a komponent nabízených React Hook Form. Chyby jsou dostupné prostřednictvím objektu **errors**, což umožňuje zobrazovat konkrétní validační zprávy pro každé pole na základě typu chyby.

```
case 'select':
    return (
        <div key={ index }>
            <label htmlFor={ field.name }>{ field.label }</label>
            <Controller
                name={ field.name }
                control={ control }
                rules={ {required: field?.validations?.required} }
                render={ ({field: {onChange, value ... }}) => (
                    <select onChange={ onChange } value={ value } id={ field.name }>
                        { field?.options?.map( callbackFn: (option :QuestionOptions , optionIndex :number ) => (
                            <option key={ optionIndex } value={ option.value }>
                                { option.display }
                            </option>
                        )) }
                    </select>
                ) }
            />
            { errors[field.name] && <p>{ field.label } is required</p> }
        </div>
    );
```

Obrázek 38 React Hook Form implementace zobrazení "select" pole. Zdroj: vlastní zpracování

```
case 'checkbox':
    return (
        <div key={ index }>
            <Controller
                name={ field.name }
                control={ control }
                rules={ {required: field?.validations?.required} }
                render={ ({field ... }) => (
                    <input
                        type="checkbox"
                        id={ field.name }
                        { ...field }
                    />
                ) }
            />
            <label htmlFor={ field.name }>{ field.label }</label>
            { errors[field.name] && <p>You must accept the terms</p> }
        </div>
    );
```

Obrázek 39 React Hook Form implementace zobrazení "checkbox" pole. Zdroj: vlastní zpracování

React Hook Form se zaměřuje na minimalizaci rerenderů a používá neřízené komponenty s výjimkou situací, kdy je nutné použít řízené komponenty prostřednictvím **Controller**. Tento přístup usiluje o větší výkon při práci s formuláři.

4.4 Unit testování

Unit testování je primární součástí vývoje softwaru, která zajišťuje, že jednotlivé komponenty aplikace fungují správně jak o samotě, tak ve spolupráci s ostatními částmi systému. V kontextu webových aplikací hrají formuláře klíčovou roli v interakci s uživateli, a proto je důležité, aby byly jejich funkce pečlivě otestovány. Tato kapitola se zaměří na specifika unit testování formulářů ve frameworkách Angular a React, využívajících testovací nástroje Jasmine pro Angular a Jest pro React.

Testování formulářů se soustředí na:

1. **Měření doby načtení formuláře**: Testy zaměřené na to, jak rychle je formulář připraven k interakci po načtení komponenty.
2. **Měření doby reakce na uživatelské vstupy**: Testy ověřující, jak efektivně formulář zpracovává vstupy, včetně validace a zobrazení chybových zpráv.
3. **Měření doby reakce na dynamické přidávání a odebírání prvků formuláře**: Testy zaměřené na flexibilitu formulářů při dynamických změnách, jako je přidávání nebo odebírání polí.

Počet opakování každého testovacího scénáře je 1000krát. Provedením velkého počtu testů a výpočtem průměru z těchto běhů se náhodné výkyvy minimalizují, což vede k přesnějšímu a spolehlivějšímu měření výkonu.

4.4.1 Doba načtení – Reactive Forms

Abychom úspěšně otestovali komponentu využívající Reactive Forms, je klíčové řádně nastavit testovací prostředí v testovacím souboru.

```
beforeEach( action: async () => {
  await TestBed.configureTestingModule( moduleDef: {
    schemas: [NO_ERRORS_SCHEMA],
    declarations: [ReactiveFormsSampleComponent],
    imports: [
      ReactiveFormsModule,
      HttpClientModule,
    ],
    providers: [
      FormBuilder,
      {provide: FormTransportService, useValue: transportService}
    ]
  }).compileComponents();
});
```

Obrázek 40 Angular nastavení testovacího prostředí. Zdroj: vlastní zpracování

Klíčovým krokem při nastavování testovacího prostředí je použití metody **TestBed.configureTestingModule**, která slouží k konfiguraci testovacího modulu pro danou sadu testů. Konfigurace zahrnuje definici komponent, které budou testovány nebo použity během testování, importy nezbytných Angular modulů a definici poskytovatelů služeb potřebných pro testy.

- **declarations**: Zde se specifikují komponenty, které se budou testovat nebo které jsou nezbytné pro testy. Například, **ReactiveFormsSampleComponent** je komponenta určená k testování.
- **imports**: V této sekci se uvádí seznam Angular modulů potřebných pro testovanou komponentu. Pro podporu reaktivních formulářů v testech se zde přidává **ReactiveFormsModule**, zatímco **HttpClientModule** je nezbytný pro komponenty, které využívají **HttpClient** pro volání API.
- **providers**: Zde jsou definováni poskytovatelé služeb, které testovaná komponenta vyžaduje. **FormBuilder** je služba používaná k vytváření instancí formulářů v Angular reaktivních formulářích. Mock instance **FormTransportService** se poskytuje pomocí **useValue: transportService**, kde **transportService** je mock objekt nebo služba, která nahrazuje reálnou instanci **FormTransportService** během testů a eliminuje tak závislost na externích API voláních.

- **compileComponents()**: Tento krok, obvykle používaný s **async** a **await** pro zajištění dokončení, je nezbytný pro kompilaci komponent a šablon před zahájením testů. Je to zásadní, zejména při práci s externími šablonami a styly, aby se zajistilo, že všechny zdroje jsou dostupné a správně načteny během testování.

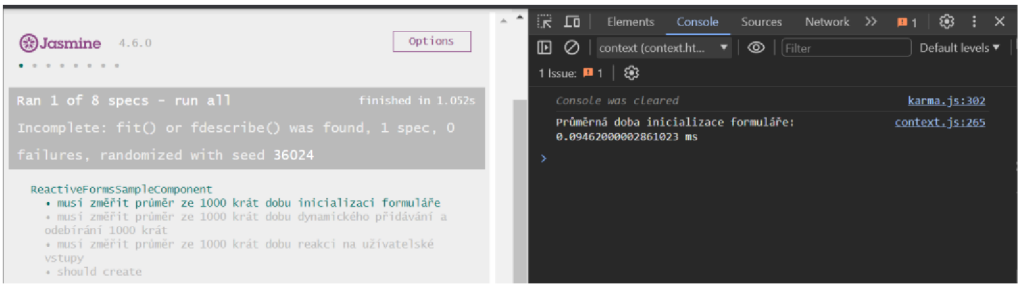
```
it( expectation: 'musí změřit průměr ze 1000 krát dobu inicializaci formuláře', fakeAsync( fn: () => {
  let totalTime = 0;
  expect(component.form).toBeDefined();
  expect(component.config).toBeDefined();
  for (let i = 0; i < iterations; i++) {
    const startTime = performance.now();
    component.ngOnInit();
    fixture.detectChanges();
    const checkboxElement = fixture.nativeElement.querySelector('#acceptTerms');
    expect(checkboxElement).toBeTruthy();
    const endTime = performance.now();
    totalTime += endTime - startTime;
  }

  initTime = totalTime / iterations;
  console.log(`Průměrná doba inicializace formuláře: ${ initTime } ms`);
}));
```

Obrázek 41 Reactive Forms testovací scénář – doba načtení. Zdroj: vlastní zpracování

V rámci cyklu **for** dochází k měření času nutného pro jednotlivé inicializace formuláře, což se realizuje pomocí metody **component.ngOnInit()** a následnou detekcí změn pomocí **fixture.detectChanges()**. Tento postup imituje běžný proces inicializace a zpracování formuláře v aplikaci. Čas před a po inicializaci se zaznamenává pomocí **performance.now()**, kde rozdíl mezi koncovým a počátečním časem (endTime - startTime) udává dobu potřebnou pro inicializaci v rámci jedné iterace.

Po každé inicializaci se pomocí **expect(checkboxElement).toBeTruthy()** ověřuje, zda byl element formuláře (v tomto případě checkbox) korektně vyrenderován.

Po absolvování všech iterací se pro výpočet průměrné doby inicializace celkový čas (totalTime) vydělí počtem iterací (iterations). Výsledná průměrná doba je poté zobrazena na konzoli.



Obrázek 42 Reactive Forms – doba načtení výsledky. Zdroj: vlastní zpracování

Výsledek vypočtu je: 0.09462000002861023ms.

4.4.2 Doba načtení – Template-driven forms

Template-driven forms a Reactive Forms se od sebe odlišují způsobem zpracování jednotlivých prvků formuláře. U Template-driven forms je klíčovým prvkem model, který může být reprezentován objektem nebo třídou. Změny stavu formuláře jsou řízeny přímo ve šabloně, přičemž hodnoty polí jsou vázány a zapisovány do odpovídajících atributů v modelu.

```
it( expectation: 'musí změřit průměr ze 1000 krát dobu inicializaci formuláře', assertion: () => {
  let totalTime = 0;
  expect(component.model).toBeDefined();
  for (let i = 0; i < iterations; i++) {
    const startTime = performance.now();
    component.ngOnInit();
    fixture.detectChanges();
    expect(component.model).toBeTruthy();
    const checkboxElement = fixture.nativeElement.querySelector('#acceptTerms_5');
    expect(checkboxElement).toBeTruthy();
    const endTime = performance.now();
    totalTime += endTime - startTime;
  }

  initTime = totalTime / iterations;
  console.log(`Průměrná doba inicializace formuláře: ${ initTime } ms`);
});
```

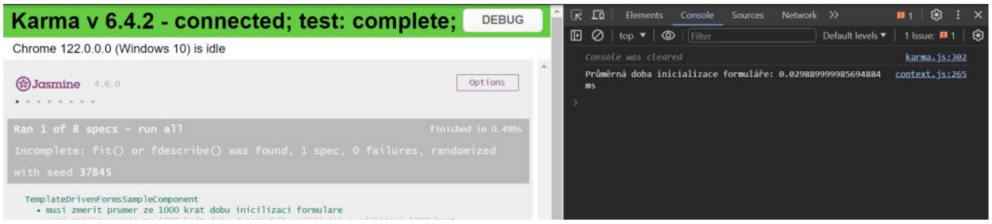
Obrázek 43 Template-driven forms testovací scénář – doba načtení. Zdroj: vlastní zpracování

V rámci **for** cyklu dochází k měření času potřebného pro inicializaci formuláře pomocí metody **component.ngOnInit()** a následné detekci změn skrze

fixture.detectChanges(). Tyto kroky imitují typický průběh inicializace a zpracování formuláře v reálné aplikaci.

Po každém spuštění inicializace se používá příkaz **expect(component.model).toBeTruthy()** k ověření, že vlastnost **model** obsahuje hodnoty, které nejsou nulové nebo undefined.

Po absolvování všech iterací se pro výpočet průměrné doby inicializace součet všech časů (**totalTime**) vydělí celkovým počtem iterací (**iterations**). Výsledná průměrná doba je poté zobrazena na konzoli.



Obrázek 44 Template-driven forms – doba načtení výsledky. Zdroj: vlastní zpracování

Výsledek vypočtu: 0.029889999985694884ms.

4.4.3 Doba načtení – Formik

V testování React aplikací se klade velký důraz na deklarativní určení očekávaného chování a interakci s DOM prostřednictvím nástrojů, jako je React Testing Library. Tento přístup je v souladu s principy "testování jako uživatel", které podporují přirozené interakce s komponentami, je prováděl koncový uživatel.

```
it( name: 'musí změřit průměr ze 1000 krát dobu inicializaci formuláře', fn: async () => {
    const iterations = 1000;
    let totalTime = 0;

    for (let i = 0; i < iterations; i++) {
        const start = performance.now();

        render(<FormikSample/>);

        await waitFor( callback: () => expect(screen.getByTestId('input-test-password')).toBeInTheDocument());

        const end = performance.now();
        totalTime += end - start;
        cleanup();
    }
    const averageTime = totalTime / iterations;
    console.log(`Průměrná doba inicializace formuláře: ${ averageTime }ms`);
});
```

Obrázek 45 Formik testovací scénář – doba načtení. Zdroj: vlastní zpracování

Funkce **render()** z React Testing Library se používá k vykreslení React komponenty v virtuálním DOM pro účely testování. Tím umožňuje simulovat chování komponenty v prohlížeči bez nutnosti skutečného renderování. Voláním **render(<FormikSample/>)** se v testovacím kódu vytváří prostředí, ve kterém se komponenta FormikSample chová, jako by byla vyrenderována v prohlížeči, a to v každé iteraci testovacího cyklu.

Objekt **screen**, který je součástí React Testing Library, poskytuje přístup k různým query funkcím pro vyhledávání elementů v DOM.

Použitím funkce **waitFor** a **expect(screen.getByTestId('input-test-password')).toBeInTheDocument()** test čeká, až se na stránce objeví formulářový prvek s testovacím ID 'input-test-password'. Tento postup zaručuje, že formulář byl úplně načten a je připraven pro interakci.

```
react-scripts test --testNamePattern=<Formik /> musí změřit průměr ze 1000 krát dobu inicializaci formuláře
 console.log
   Průměrná doba inicializace formuláře: 3.8767916999999894ms
```

Obrázek 46 Formik – doba načtení výsledky. Zdroj: vlastní zpracování

Výsledek výpočtu průměru z 1000krat opakování je 3.8767916999999894ms.

4.4.4 Doba načtení – React Hook Form

Kód určený k testování průměrné doby inicializace je identický s tím, který se používá pro Formik. V tomto kontextu dochází k načtení komponenty "ReactHookForm".

```
react-scripts test --testNamePattern=<ReactHookForm /> musí změřit průměr ze 1000 krát dobu inicializaci formuláře
 console.log
   Průměrná doba inicializace formuláře: 3.428051100000004ms
```

Obrázek 47 React Hook Form – doba načtení výsledky. Zdroj: vlastní zpracování

Výsledek výpočtu: 3.428051100000004ms.

4.4.5 Reakce na uživatelské vstupy – Reactive Forms

Reaktivní formuláře v Angularu využívají třídu FormGroup pro skupiny prvků a v ní zanořené třídy FormControl pro jednotlivé prvky formuláře. Klíčovým krokem při testování formuláře je lokalizace specifického prvku odpovídajícího poli, které se má modifikovat. Díky metodám nastaveným pro změnu hodnot polí dochází po jejich aplikaci

k aktualizaci DOM. Tím se zajišťuje synchronizace mezi hodnotou v HTML prvku a hodnotou v příslušné třídě FormControl.

```
it( expectation: 'musí změřit průměr ze 1000 krát dobu reakci na uživatelské vstupy', assertion: () => {
  component.ngOnInit();
  fixture.detectChanges();
  expect(component.form).toBeDefined();
  expect(component.config).toBeDefined();
  let totalTime = 0;
  const inputControl = component.form.get('name') as FormControl;
  for (let i = 0; i < iterations; i++) {
    const startTime = performance.now();
    inputControl.setValue( value: `Test Value ${ i }`);
    fixture.detectChanges();
    expect(fixture.nativeElement.querySelector(`#name`)?.value).toEqual( expected: `Test Value ${ i }`);
    expect(component.form.get('name')?.value).toEqual( expected: `Test Value ${ i }`);
    const endTime = performance.now();
    totalTime += endTime - startTime;
  }
  inputReactionTime = totalTime / iterations;
  console.log(`Průměrná doba reakce na uživatelský vstup: ${ inputReactionTime } ms`);
});
```

Obrázek 48 Reactive Forms testovací scénář – reakce na uživatelské vstupy. Zdroj: vlastní zpracování

Výsledek výpočtu průměru z 1000 opakování je: 0.06682999992370606ms

Obrázek 49 Reactive Forms – reakce na uživatelské vstupy. Zdroj: vlastní zpracování

4.4.6 Reakce na uživatelské vstupy – Template-driven forms

V Template-driven forms dochází k manipulaci s poli prostřednictvím šablony, na rozdíl od Reactive Forms, kde se manipulace provádí přímo ve třídě FormControl. Při testování se liší metodika tak, že u reaktivních formulářů se vybírá specifický atribut ve třídě FormControl, zatímco u formulářů řízených šablonou je nutné vybrat HTML prvek odpovídající poli, které měníme, a spustit na něm akci „input“ pro simulaci zadávání textu.

Následně se kontroluje, zda je nová hodnota přítomná ve šabloně.

```
fit( expectation: 'musí změřit průměr ze 1000 krát dobu reakci na uživatelské vstupy', waitForAsync( fn: () => {
  let totalTime = 0;
  expect(component.model).toBeTruthy();
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    for (let i = 0; i < iterations; i++) {
      const startTime = performance.now();
      const input = fixture.debugElement.query(By.css( selector: '#name_0')).nativeElement;
      input.value = 'Jan';
      input.dispatchEvent(new Event('input'));
      expect(component.form.form.get('name')?.value).toEqual( expected: 'Jan');
      const endTime = performance.now();
      totalTime += endTime - startTime;
    }
    inputReactionTime = totalTime / iterations;
    console.log(`Průměrná doba reakce na uživatelský vstup: ${ inputReactionTime } ms`);
  });
}));
```

Obrázek 50 Template-driven forms testovací scénář – reakce na uživatelské vstupy. Zdroj: vlastní zpracování

Výsledek výpočtu průměru z 1000 opakování je: 0.2924400000810623ms

Obrázek 51 Template-driven forms – reakce na uživatelské vstupy výsledky. Zdroj: vlastní zpracování

4.4.7 Reakce na uživatelské vstupy – Formik

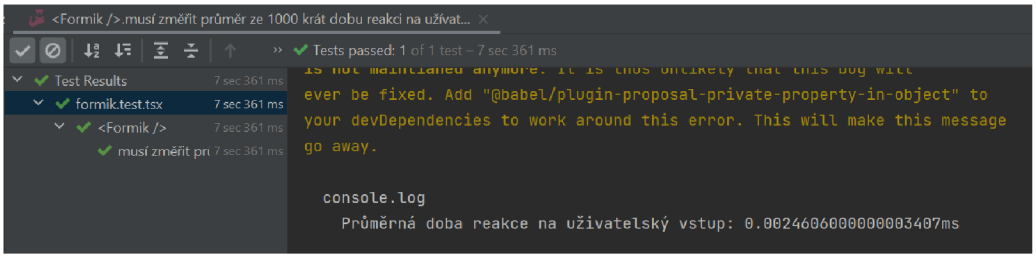
Při testování knihovny Formik ve srovnání s Angular Reactive Forms se používá funkce „fireEvent“ z React Testing Library k simulaci uživatelských interakcí, jako je například změna hodnoty ve vstupním poli. React Testing Library se zaměřuje na testování z uživatelské perspektivy, což znamená, že interaguje s komponentami tak, jak by to dělal uživatel, namísto přímé manipulace se stavem nebo interními aspekty

komponenty. komponenty.

```
it( name: 'musí změřit průměr ze 1000 krát dobu reakci na uživatelské vstupy', fn: async () => {
    const iterations = 1000;
    let totalTime = 0;
    for (let i = 0; i < iterations; i++) {
    const {getByTestId} = render(<FormikSample/>);
        const input = getByTestId( container: 'input-test-name');
        const start = performance.now();

        await act(() => {
            fireEvent.change(input, options: {target: {value: 'test'}});
        })

        await screen.findByDisplayValue('test');

        const end = performance.now();
        totalTime = end - start;
        cleanup();
    }
    const averageTime = totalTime / iterations;
    console.log(`Průměrná doba reakce na uživatelský vstup: ${ averageTime }ms`);
});
```

Obrázek 52 Formik testovací scénář – reakce na uživatelské vstupy. Zdroj: vlastní zpracování

Výsledek výpočtu průměru z 1000 opakování: 0.0024606000000003407ms.

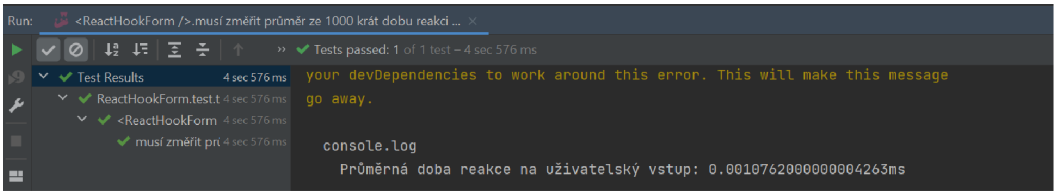


Obrázek 53 Formik – reakce na uživatelské vstupy výsledek. Zdroj: vlastní zpracování

4.4.8 Reakce na uživatelské vstupy – React Hook Form

Metoda testování React Hook Form je podobná metodě testování knihovny Formik.

Výsledek výpočtu průměru z 1000 opakování: 0.0010762000000004263ms.



Obrázek 54 React Hook Form – reakce na uživatelské vstupy výsledky. Zdroj: vlastní zpracování

4.4.9 Dynamické přídavní a odebíraní prvků – Reactive Forms

Formuláře využívající Reactive Forms jsou řízeny prostřednictvím tříd FormGroup a FormControl. Pro přidání nového pole je nezbytné vytvořit instanci FormControl. Metody definované v rámci komponenty umožňují přidávání a odebírání jednotlivých prvků formuláře. V daném testovacím scénáři se po přidání prvku ověřuje jeho existence v šabloně, následně se prvek odstraní a opětovně se kontroluje jeho absence.

```
it( expectation: 'musí změřit průměr ze 1000 krát dobu dynamického přidávání a odebírání 1000 krát', assertion: () => {
  let totalTime = 0;
  expect(component.form).toBeDefined();
  expect(component.config).toBeDefined();
  for (let i = 0; i < iterations; i++) {
    const controlName = `control${ i }`;
    const control = new FormControl(
      value: `Test přidávání control ${ i }`,
      validatorOrOpts: [Validators.required, Validators.minLength( minLength: 10)]
    );

    const startTime = performance.now();

    component.addFormControl(controlName, control);
    expect(component.form.get(controlName)).toBeDefined();
    fixture.detectChanges();

    expect(fixture.nativeElement.querySelector(`#control${ i }`)).toBeTruthy();
    component.removeFormControl(controlName);
    fixture.detectChanges();

    expect(fixture.nativeElement.querySelector(`#control${ i }`)).toBeFalsy();
    expect(component.form.get(controlName)).toBeNull();
    const endTime = performance.now();

    totalTime += endTime - startTime;
  }
  dynamicAddRemoveTime = totalTime / iterations;
  console.log(`Průměrný čas přidání a odebrání FormControl: ${ dynamicAddRemoveTime } ms`);
});
});
```

Obrázek 55 Reactive Forms testovací scénář – dynamické přidávaní a odebírání prvku. Zdroj: vlastní zpracování

Výsledek vypočtu dynamického přidáváni a odebíraní prvků: 1.384900001525879ms.

Obrázek 56 Reactive forms – dynamické přidávaní a odebírání výsledky. Zdroj: vlastní zpracování

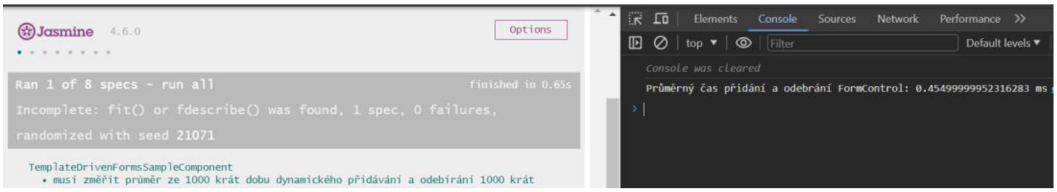
4.4.10 **Dynamické přídavní a odebíraní prvků – Template-driven forms**

V Template-driven forms je logika formuláře implementována přímo v šabloně. V příslušném testovacím scénáři byl vytvořen objekt, který má být zobrazen ve šabloně. Metody definované v komponentě umožňují dynamické přidávání a odebírání prvků. Způsob kontroly je podobný jako u Reactive Forms, tedy ověření přítomnosti prvku v DOM a jeho absence po odstranění.

```
it( expectation: 'musí změřit průměr ze 1000 krát dobu dynamického přidávání a odebírání 1000 krát', waitForAsync( fn: () => {
  let totalTime = 0;
  expect(component.model).toBeDefined();
  fixture.whenStable().then(() => {
    for (let i = 0; i < iterations; i++) {
      const control = {
        type: 'text',
        name: 'surname',
        label: 'Příjmení',
        placeholder: 'Zadejte své příjmení',
        validations: {
          required: true,
        }
      } as FormControlConfig;

      const startTime = performance.now();

      component.addControl(control);
      fixture.detectChanges();
      expect(component.form.form.get(control.name)).toBeDefined();
      fixture.detectChanges();
      expect(fixture.nativeElement.querySelector(`#surname_6`)).toBeTruthy();

      component.removeControl(control.name);
      fixture.detectChanges();
      expect(fixture.nativeElement.querySelector(`#surname_6`)).toBeFalsy();
      expect(component.form.form.get(control.name)).toBeNull();
      const endTime = performance.now();

      totalTime += endTime - startTime;
    }
    dynamicAddRemoveTime = totalTime / iterations;
    console.log(`Průměrný čas přidání a odebrání FormControl: ${ dynamicAddRemoveTime } ms`);
  });
```

Obrázek 57 Template-driven forms testovací scénář – dynamické přidávaní a odebírání prvku. Zdroj: vlastní zpracování

Výsledek vypočtu dynamického přidáváni a odebíraní prvků: 0.45499999952316283ms.



Obrázek 58 Template-driven forms – dynamické přidávání a odebírání prvku výsledky. Zdroj: vlastní zpracování

4.4.11 **Dynamické přídavní a odebíraní prvků – Formik**

Knihovna Formik nabízí komponenty a statické funkce pro dynamické přidávání a odebírání prvků. Tyto funkce jsou přiřazeny k tlačítkům „Add control“ a „Remove control“. V určeném testovacím scénáři se nejprve vyhledá identifikátor v šabloně, který odpovídá tlačítku pro přidání prvku. Následně se simuluje kliknutí na tlačítko a ověřuje se přítomnost nového pole podle jeho ID. Proces odebírání prvků se testuje obdobným způsobem.

```
it( name: 'musí změřit průměr ze 1000 krát dobu dynamického přidávání a odebírání 1000 krát', fn: async () => {
    const iterations = 1000;
    let totalTime = 0;
    for (let i = 0; i < iterations; i++) {
        const {getByTestId, queryByTestId} = render(<FormikSample/>);
        const addButton = getByTestId( container: 'add-button');
        const start = performance.now();
        await act(() => {
            userEvent.click(addButton);
        })

        await waitFor( callback: () => expect(queryByTestId( container: 'input-test-surname')).toBeInTheDocument());

        const removeButton = getByTestId( container: 'remove-button');
        await act(() => {
            userEvent.click(removeButton);
        })
        await waitFor( callback: () => expect(queryByTestId( container: 'input-test-surname')).not.toBeInTheDocument());
        const end = performance.now();
        totalTime = end - start;
        cleanup();
    }
    const averageTime = totalTime / iterations;
    console.log(`Průměrný čas přidání a odebrání FormControl: ${ averageTime }ms`);
});
```

Obrázek 59 Formik testovací scénář – dynamické přidávaní a odebírání prvku. Zdroj: vlastní zpracování

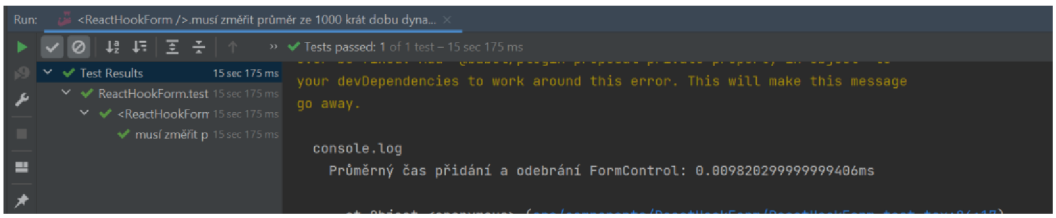
Výsledek vypočtu dynamického přidáváni a odebíraní prvků: 0.018230599999999896ms

Obrázek 60 Formik – dynamické přidávání a odebírání prvků výsledky. Zdroj: vlastní zpracování.

4.4.12 **Dynamické přídavní a odebíraní prvků – Hook Form**

Metoda testování React Hook Form se podobá metodě testování knihovny Formik.

Výsledek vypočtu dynamického přidáváni a odebíraní prvků: 0.009820299999999406ms



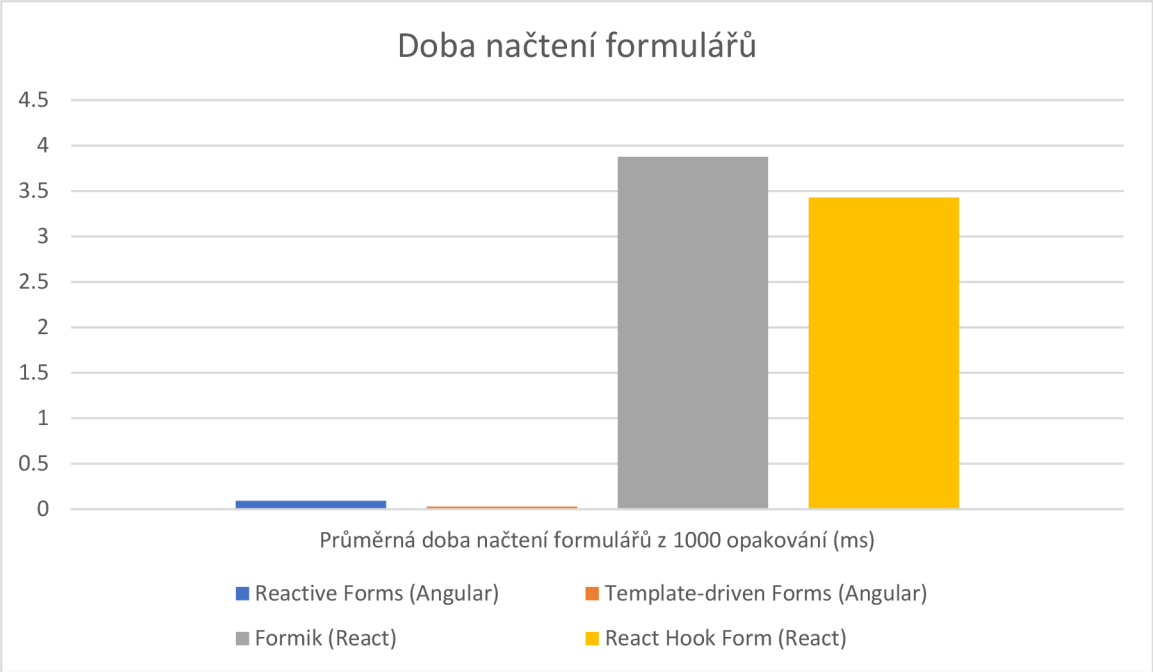
Obrázek 61 React Hook Form – dynamické přidávání a odebírání prvků výsledky. Zdroj: vlastní zpracování

5 Výsledky a diskuse

Následující analýza a diskuse jsou založeny na důkladné implementaci a testování výkonnosti různých formulářů v prostředí pro vývoj softwaru. Získané detailní výsledky nám poskytují možnost provést komparativní studii rychlosti formulářů vyvinutých v Angular a React. Tento oddíl je věnován rozboru uvedených výsledků a jejich implikací pro selekci technologických řešení v budoucích projektech. V rámci této analýzy jsou srovnávány různé metody práce s formuláři – Template-driven a Reactive forms v Angularu, a knihovny React Hook Form a Formik v React – z perspektivy jejich výkonnosti.

5.1 Doba načtení

Na základě informací získaných z testování výkonnosti jednotlivých přístupů k formulářům v Angular a React lze hodnotit, jakým způsobem tyto rozdílné technologie ovlivňují dobu načítání a celkovou reaktivitu aplikace. Cílem testů bylo změřit průměrnou dobu načítání formuláře po tisíci iteracích, což zajišťuje statistickou relevanci zjištěných výsledků. Prezentované hodnoty jsou zaznamenány v přiloženém grafu.



Graf 1 Doba načtení formulářů výsledky. Zdroj: vlastní zpracování

Výše uvedená data naznačují, že přístupy k formulářům implementované v Angular (jak Reactive, tak Template-Driven) jsou podstatně rychlejší ve srovnání s přístupy založenými na Reactu (Formik a React Hook Form). Tento rozdíl může být ovlivněn řadou faktorů, včetně způsobu, jakým tyto frameworky a knihovny řeší aktualizace DOM a management stavu formulářů.

Template-Driven Forms se v tomto testování ukázaly jako nejrychlejší, což může být pro některé překvapivé, jelikož se obecně považují za méně efektivní ve srovnání s Reactive Forms. Tento výsledek naznačuje, že pro jednoduché aplikace může být tento přístup vhodnější.

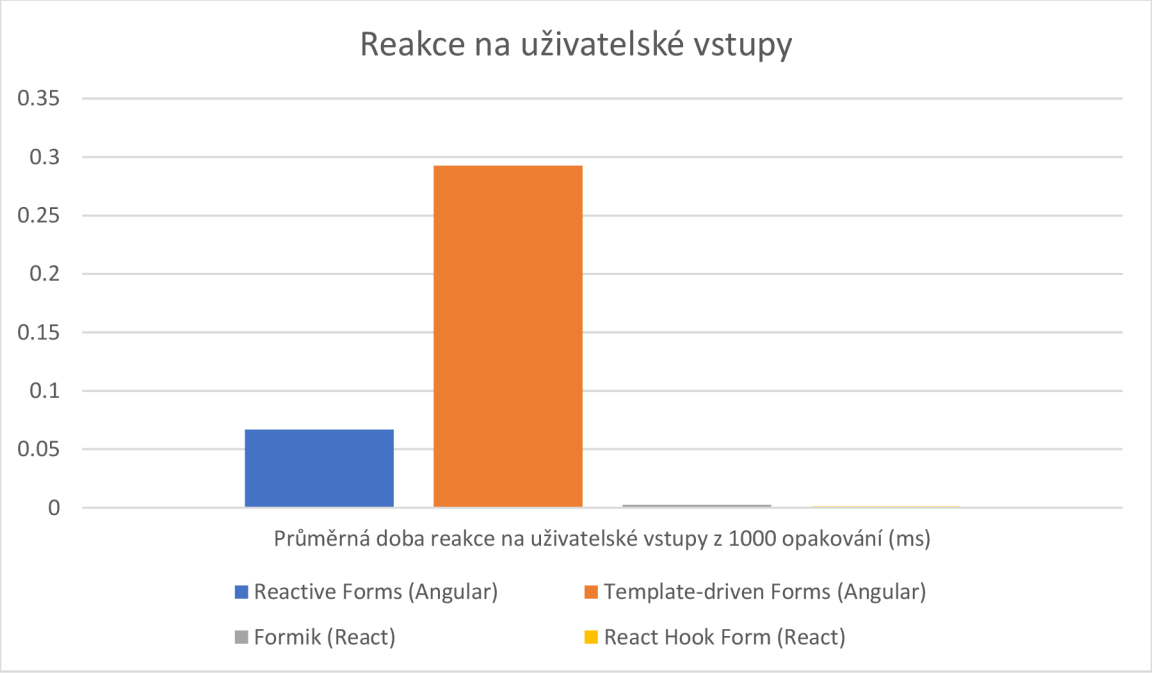
Reactive Forms vykazovaly o něco delší průměrnou dobu načítání než Template-Driven Forms, což může souviset s jejich komplexnějším způsobem správy stavu a validace formulářů. Přesto nabízí rychlou reakci a jsou preferovány pro komplexnější formulářové struktury.

Formik a React Hook Form vykázaly výrazně delší doby načítání ve srovnání s metodami pro formuláře v Angularu. Tyto knihovny nabízejí uživatelsky přívětivé API pro správu formulářů v Reactu a zároveň zvyšují režii při aktualizacích formulářů. Tyto výsledky jednoznačně ukazují na nutnost další optimalizace v případech, kdy je výkon aplikace klíčový.

5.2 Reakce na uživatelské vstupy

V rámci hodnocení průměrné reakční doby na uživatelské vstupy na základě tisíce iterací v kontextu různých technologií správy formulářů v Angularu a Reactu, analýza dat odhalila následující výsledky:

Porovnání dat o průměrných reakčních dobách na uživatelské vstupy odhaluje signifikantní změnu ve vztahu k dřívějšímu posouzení doby načítání formulářů. Přestože formulářové přístupy v Angular (tanto Reactive Forms jako i Template-Driven Forms) dosahovaly rychlejšího načítání, technologie využívané v React (konkrétně Formik a React Hook Form) prezentují v kontextu reakcí na uživatelské vstupy značně lepší výkonnost.



Graf 2 Reakce na uživatelské vstupy výsledky. Zdroj: vlastní zpracování

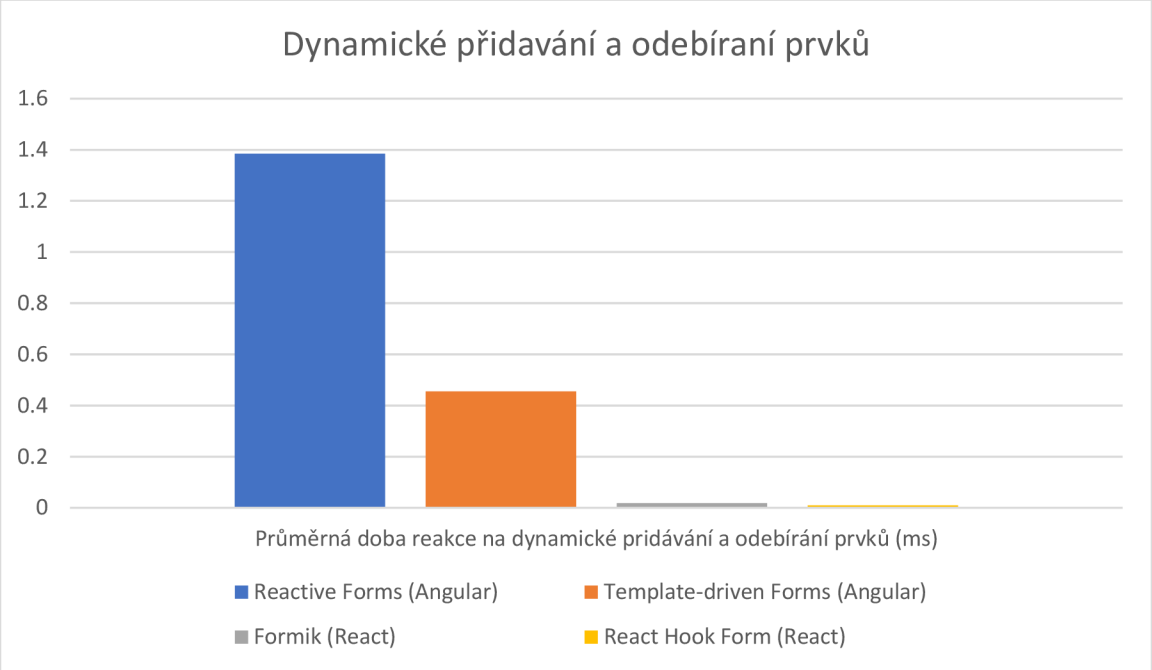
Reactive Forms vykazují rychlejší reakci na uživatelské vstupy ve srovnání s Template-Driven Forms, což je důsledek jejich imperativního přístupu a efektivnějšího managementu stavu formuláře. Nicméně, stále zaostávají za React knihovnami z hlediska reakční doby.

Template-Driven Forms se ukázaly jako nejpomalejší v reakci na uživatelské vstupy, což může být přičteno jejich vyšší úrovni abstrakce a způsobu, jakým Angular zpracovává aktualizace formulářů a data-binding.

Obě knihovny, Formik a React Hook Form, prokazují vysokou reakční rychlost na uživatelské vstupy, což naznačuje jejich schopnost efektivně manipulovat s DOM a minimalizovat počet re-renderů. Tyto vlastnosti jsou důležité pro aplikace s potřebou vysoké úrovně interaktivity uživatelů.

5.3 Dynamické přídavní a odebíraní prvků

V rámci analýzy průměrné doby reakce na procesy dynamického přidávání a odebírání prvků formuláře nabízejí revidované údaje podrobný vhled do rozdílů ve výkonnosti mezi aplikovanými technologiemi pro správu formulářů.



Graf 3 Dynamické přidávání a odebírání výsledky. Zdroj: vlastní zpracování

Reactive Forms vykazují nejpomalejší průměrnou dobu reakce při dynamickém přidávání nebo odebírání prvků. To může být důsledkem složitější správy stavu a aktualizací formulářů, které tento přístup vyžaduje.

Template-Driven Forms jsou v této operaci výrazně rychlejší než Reactive Forms, což naznačuje, že jejich abstrakce a zjednodušená manipulace s formuláři může být efektivnější pro dynamické změny ve formuláři.

Formik i React Hook Form prezentují extrémně nízké průměrné doby reakce na dynamické změny, což ukazuje na vysokou efektivitu a optimalizaci těchto knihoven pro rychlé manipulace s formulářovými prvky v React aplikacích.

React Hook Form vyniká jako nejrychlejší s upraveným výsledkem 0.00982ms, což podtrhuje jeho schopnost efektivně zpracovávat dynamické změny ve formuláři s minimální režií.

5.4 Zhodnocení výsledků

Analýza získaných dat ukázala, že i přes různé přístupy k implementaci formulářů v Angular a React existují významné rozdíly ve výkonnosti a uživatelské zkušenosti.

Reactive Forms a Template-Driven Forms v Angularu prokázaly rozdílnou výkonnost v různých testovaných scénářích. Zatímco Reactive Forms nabízejí lepší kontrolu a flexibilitu pro komplexní formulářové struktury, Template-Driven Forms se ukázaly být efektivnější v jednodušších situacích s nižšími latencemi při dynamických změnách. Obě metody však mohou být z hlediska doby reakce a efektivity za určitých okolností limitující, zejména ve srovnání s některými React řešeními.

V React, jak Formik, tak React Hook Form, předvedly vynikající výkonnost, zejména v rámci rychlosti reakce na dynamické změny ve formulářích. Tyto knihovny efektivně minimalizují dobu potřebnou pro zpracování aktualizací, což přispívá k rychlejší uživatelské interakci a zlepšuje celkovou uživatelskou zkušenost. React Hook Form se navíc jeví jako nejefektivnější řešení z testovaných knihoven, s nejnižšími latencemi a nejrychlejší reakcí na uživatelské vstupy.

6 Závěr

V rámci této bakalářské práce byl stanoven primární cíl navrhnout, implementovat a srovnat vybrané formuláře ve front-endových frameworkách Angular a React s cílem rozšířit pochopení jejich využití ve vývoji webových aplikací. Vedlejší cíle zahrnovaly porovnání metod, nástrojů a osvědčených postupů pro vytváření formulářů, implementaci vybraných formulářů a přípravu experimentu, a nakonec provádění hodnocení na základě výkonnosti formulářů.

Během realizace této práce byly dosaženy všechny stanovené cíle. Byl proveden důkladný přehled dostupných metod a nástrojů pro práci s formuláři v obou zvolených technologiích, což umožnilo detailní porovnání jejich přístupů, možností a omezení. Následná implementace vybraných formulářů a jejich testování v experimentálním prostředí poskytlo kvantitativní data, jež byla analyzována s ohledem na výkonnost jednotlivých řešení.

Závěrem lze říct, že poskytnutá data a provedená analýza představují cenný přínos k porozumění výkonnostním aspektům různých přístupů k formulářům v kontextu moderního vývoje webových aplikací.

7 Seznam použitých zdrojů

[1] BIN UZAYR, Sufyan. *HTML (The Ultimate Guide)*. CRC Press, 2023. ISBN 1032413255.

[2] Forms in HTML documents. In: *W3.org* [online]. *World Wide Web Consortium* [cit. 2024-03-15]. Dostupné z: https://www.w3.org/TR/html401/interact/forms.html

[3] *HTML Forms* [online]. c1999 - 2024 [cit. 2024-03-15]. Dostupné z: https://www.w3schools.com/html/html_forms.asp

[4] MEYER, Jeanine. *Essential Guide to HTML5*. 3rd ed. United States: APress, 2022. ISBN 9781484287217.

[5] BISWAS, Nabendu. *TypeScript Basics*. 1. CA: Apress Berkeley, 2023. ISBN 978-1-4842-9522-9.

[6] WELLMAN, Dan. *Ultimate Typescript Handbook: Build, scale and maintain Modern Web Applications with Typescript*. 1. AVA: Orange Education Pvt, 2023. ISBN 9388590783.

[7] BAUMGARTNER, Stefan. *TypeScript Cookbook: Real World Type-Level Programming*. O'Reilly Media, 2023. ISBN 1098136659.

[8] FREEMAN, Adam. *Pro Angular: Build Powerful and Dynamic Web Apps*. 5th. Apress, 2022. ISBN 1484281756.

[9] *Angular*. [Online] Google, ©2010-2022. [Citace: 14. 03 2024.] https://angular.io.

[10] RIPPON, Carl. *Learn React with TypeScript 3: Beginner's guide to modern React web development with TypeScript 3*. Packt Publishing, 2018. ISBN 1484281756.

[11] NARAYN, Hari. *Just React!: Learn React the React Way: Beginner's guide to modern React web development with TypeScript 3*. Apress, 2022. ISBN 1484282930.

[12] **Exbrayat, C.** Become A Ninja With Angular. Vydáno jako e-kniha: 2023. Dostupné z: https://coderbooks.ru/media/Become%20A%20Ninja%20With%20Angular/become-a-ninja-with-angular.pdf.

[13] CALDWELL, Ronald. *What is Formik?* [online]. 2021, 2021-07-19 [cit. 2024-03-15]. Dostupné z: https://www.liquidweb.com/kb/formik-react/

[14] *Formik: Build forms in React, without the tears* [online]. 2020 [cit. 2024-03-15]. Dostupné z: https://formik.org/

[15] Minnick, Chris. *Beginning ReactJS Foundations Building User Interfaces with ReactJS : An Approachable Guide*, John Wiley & Sons, Incorporated, 2022. *ProQuest Ebook Central*, https://ebookcentral.proquest.com/lib/czup/detail.action?docID=6883629.

[16] *React Hook Form* [online]. 2022 [cit. 2024-03-15]. Dostupné z: https://react-hook-form.com/

[17] HYGRAPH TEAM. *React Hook Form: A Complete Guide* [online]. 2022 [cit. 2024-03-15]. Dostupné z: https://hygraph.com/blog/react-hook-form

[18] Sriparasa, Sai Srinivas. *Javascript and Json Essentials*, Packt Publishing, Limited, 2013. *ProQuest Ebook Central*, https://ebookcentral.proquest.com/lib/czup/detail.action?docID=1481127.

[19] *JSON: A Complete Guide* [online]. [cit. 2024-03-15]. Dostupné z: https://www.json.org/json-en.html

[20] PATNI, Sanjay. *Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS*. Apress, 2017. ISBN 9781484226643.

[21] *JavaScript JSON* [online]. c1999 - 2024 [cit. 2024-03-15]. Dostupné z: https://www.w3schools.com/js/js_json.asp

[22] OSHEROVE, Roy. *The art of unit testing with examples in C#*. 2nd ed. New York: Manning, c2014. ISBN 978-1-617290-89-3.

[23] BROWSERSTACK. Jest Framework Tutorial [online]. [cit. 2024-03-10]. Dostupné z: https://www.browserstack.com/guide/jest-framework-tutorial

[24] HAHN, Evan. JavaScript Testing with Jasmine [online]. 1st edition. O'Reilly Media, 2013 [cit. 2024-03-10]. ISBN 9781449356378.

[25] FRANCESCO, Hugo Di. The Jest Handbook: Learn Advanced JavaScript Testing patterns with Jest [online]. Code with Hugo. [cit. 2024-03-10].

[26] META PLATFORMS. *Getting started - Jest* [online]. 2024 [cit. 2024-03-15]. Dostupné z: https://jestjs.io/docs/getting-started

[27] *Jasmine Documentation* [online]. [cit. 2024-03-15]. Dostupné z: https://jasmine.github.io/index.html

[28] React. *React.* [Online] Meta Platforms, Inc., 2022. [Citace: 2024-03-15.] https://reactjs.org

[29] HUSSAIN, Asim. *Angular: From Theory To Practice: Build the web applications of tomorrow using the Angular web framework from Google.* [online]. CodeCraft, 2017

[cit. 2024-03-15]. Dostupné z: https://github.com/JooYoo/Books-Angular/blob/master/angular-from-theory-to-practice.pdf

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

8.2 Seznam grafů

8.3 Seznam použitých zkratek

HTML – HyperText Markup Language
SPA – Single Page Application
HTTP – HyperText Transfer Protocol
CLI – Command Line Interface
DOM – Document Object Model
UI – User Interface
JSON – JavaScript Object Notation
API – Application Programming Interface
BDD – Behavior-Driven Development
TSX – TypeScript for JSX