Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Srovnání JavaScriptových frameworků

2023

Bc. David Hrůza

Vedoucí práce:
RNDr. Martin Trnečka Ph.D.

Studijní program: Aplikovaná informatika,
prezenční forma

Bibliografické údaje

Autor:	Bc. David Hrůza
Název práce:	Srovnání JavaScriptových frameworků
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	RNDr. Martin Trnečka Ph.D.
Počet stran:	42
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. David Hrůza
Title:	Comparison of JavaScript frameworks.
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, full-time form
Supervisor:	RNDr. Martin Trnečka Ph.D.
Page count:	42
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Cílem této diplomové práce je srovnání reaktivních přístupů JavaScriptových frameworků Vue, Svelte a SolidJS. Součástí je jednoduchá aplikace představující poštovního klienta v každém frameworku pro demonstartivní účely.

Synopsis

The goal of this thesis is to compare reactive data approach of JavaScript frameworks Vue, Svelte and SolidJS. Part of thesis is simple email client application written in each framework for demonstration purpose.

Klíčová slova: reaktivita; frontend; frameworky

Keywords: reactivity; frontend; frameworks

Mnohorát děkuji panu RNDr. Martinu Trnečkovi, Ph.D. za vedení a invostovaný čas.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

S mírnou nadsázkou lze tvrdit, že není týden, kdy by se neobjevil nový frontendový framework. Oblast frontendu je neustále velmi aktivně zlepšována. Jednou z hlavních charakteristik frameworků je reaktivní systém frameworku. V této práci se podíváme na tři odlišené přístupy reaktivity. Reaktivitou se obecně myslí deklarativní přístup k programování. Pouhou změnou hodnoty proměnné se daná změna projeví všude, kde je proměnná použita.

1.1 O autorovi

V rychlosti si uvedeme autora, jelikož má předešlé zkušenosti v tomto oboru. Autor diplomové práce pracuje pět let jako full–time full–stack developer. Vyvíjí interní administrační portály a eshopové jádro. Po celou dobu kariéry využívá na frontendu framework Vue [7]. V počátcích začínal na Vue 2 [6] a v posledních dvou letech Vue 3 [7]. Na backendu pracuje pouze v jazyce PHP [1] s pomocí frameworku Laravel [5].

1.2 O aplikaci Thessenger

Zadání diplomové práce je demonstrovat přístupy několika frontendových frameworků. Pro tento účel vznikla aplikace Thessenger. Jedná se aplikaci simulující poštovního klienta.

Uživatelé se mohou registrovat do systému pomocí uživatelského jména a hesla. Následně mohou ze systému odesílat v rámci systému ostatním uživatelům zprávy. Zpráva může mít více příjemců. Místo konverzačních vláken se drží klasického přístupu k emailu, kde každá zpráva udržuje historii konverzace v sobě. Aplikace na pozadí hlídá existenci nových zpráv a upozorní na novou zprávu červeným puntíkem v menu u položky „Přijaté zprávy“.

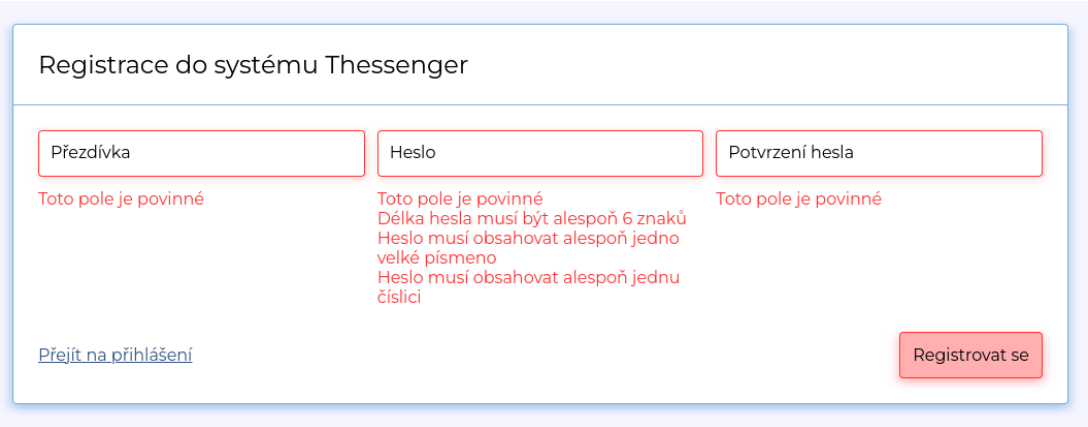
1.2.1 Mapa aplikace

Stručný seznam stránek aplikace je následující:

- Přihlášení
- Registrace
- Odeslání zprávy
- Přijaté zprávy
- Odeslané zprávy
- Detail zprávy
- Koš

1.2.2 Registrace účtu

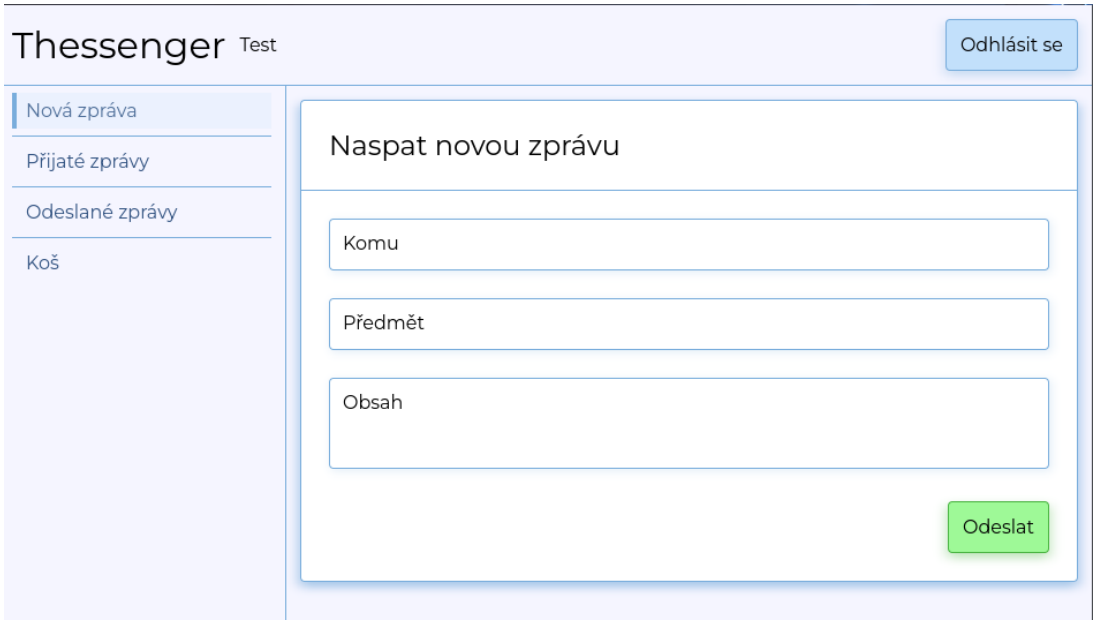
Prvním krokem k získání přístupu do aplikace je registrace účtu. Jak můžeme vidět na obrázku 1, stačí vymyslet uživatelské jméno a dostatečně silné heslo. Po vytvoření účtu je provedeno automatické přihlášení a přesměrování do zabezpečené sekce aplikace.



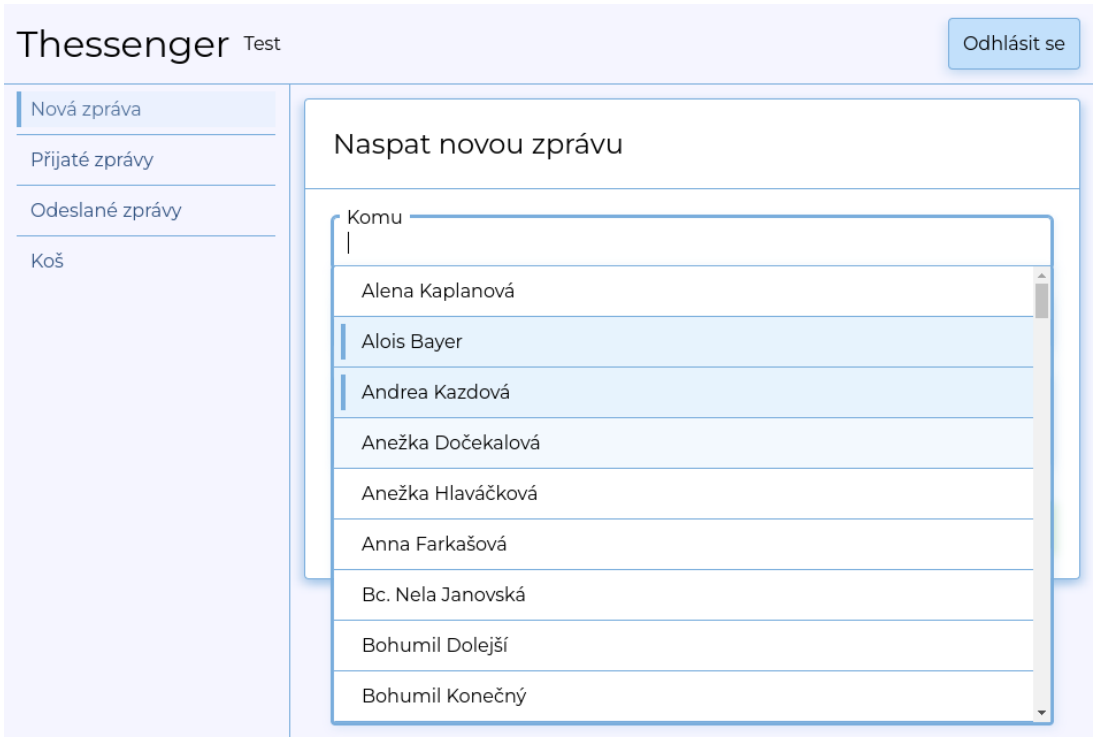
Obrázek 1: Registrace účtu.

1.2.3 Nová zpráva

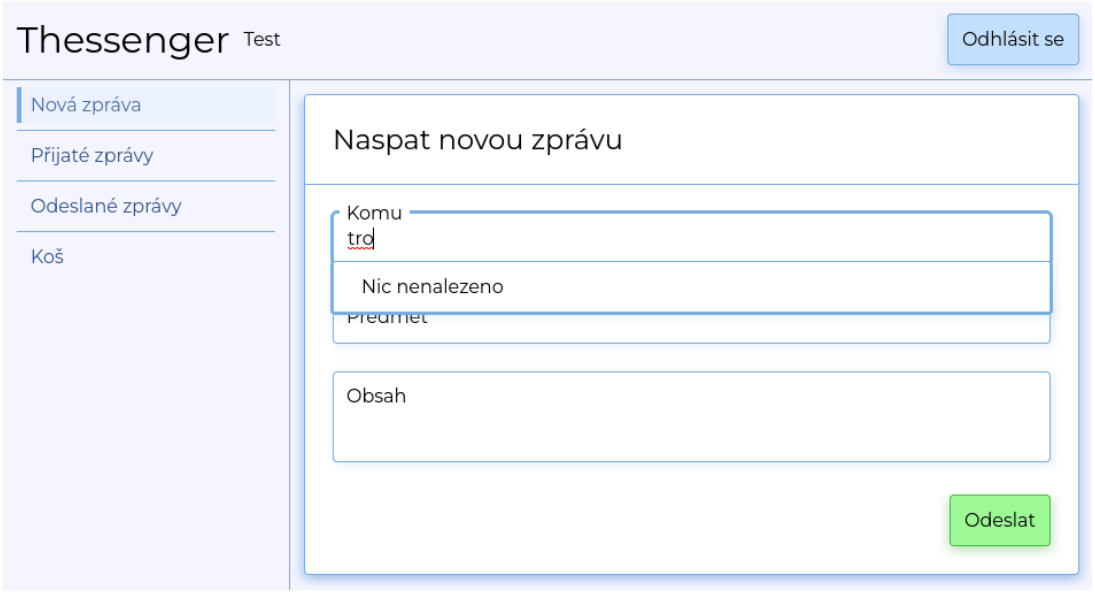
Na obrázku 2 vidíme stránku nová zpráva, na které můžeme vytvořit a odeslat zprávu. Každá zpráva má svého odesílatele – účet, ze které byla odeslána, několik příjemců, předmět a obsah. Na obrázku 3 můžeme vidět možnost výběru více příjemců. Seznam příjemců lze také filtrovat pro zúžení výběru, jak jde vidět na obrázku 4.



Obrázek 2: Nová zpráva.



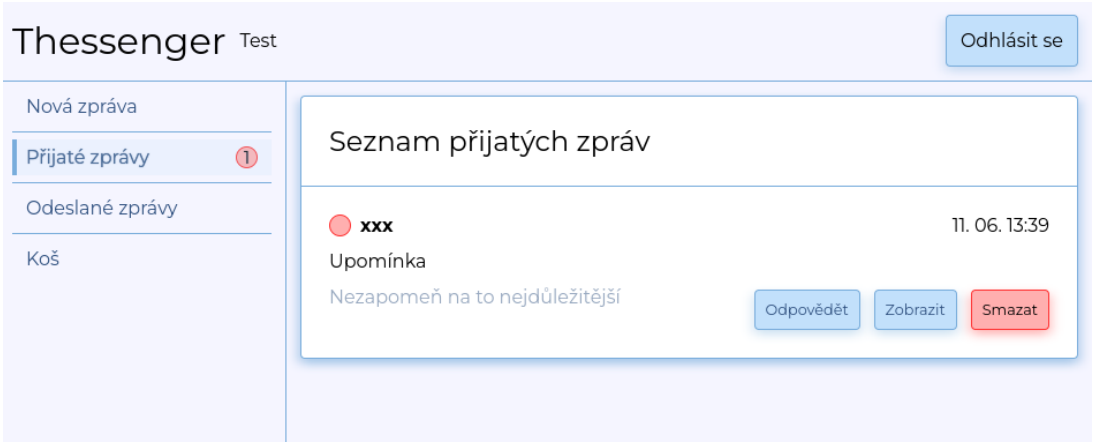
Obrázek 3: Nová zpráva výběr více příjemců.



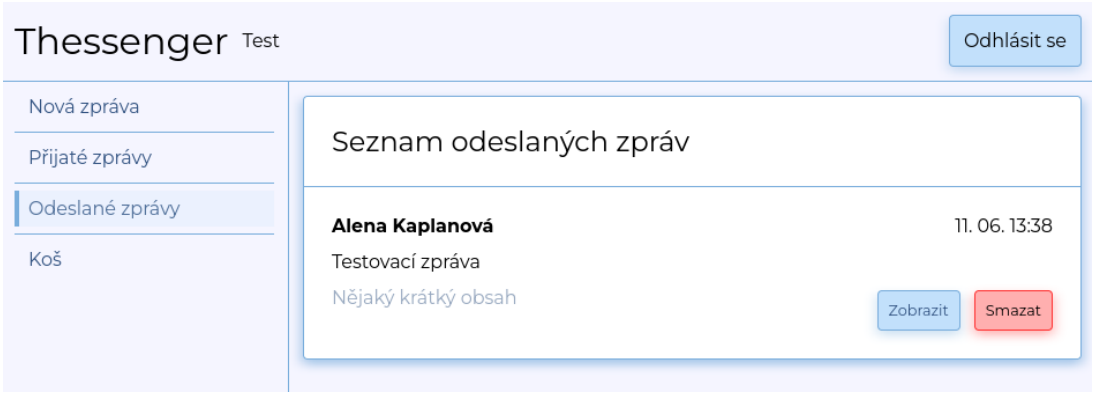
Obrázek 4: Nová zpráva filtrace.

1.2.4 Přijaté a odeslané zprávy

Jedná se o totožné stránky s filtrovanými daty. Zobrazí se zde seznam všech přijatých obrázek 5 / odeslaných obrázek 6 zpráv s možností přechodu na detail zprávy a možnosti smazání zprávy.



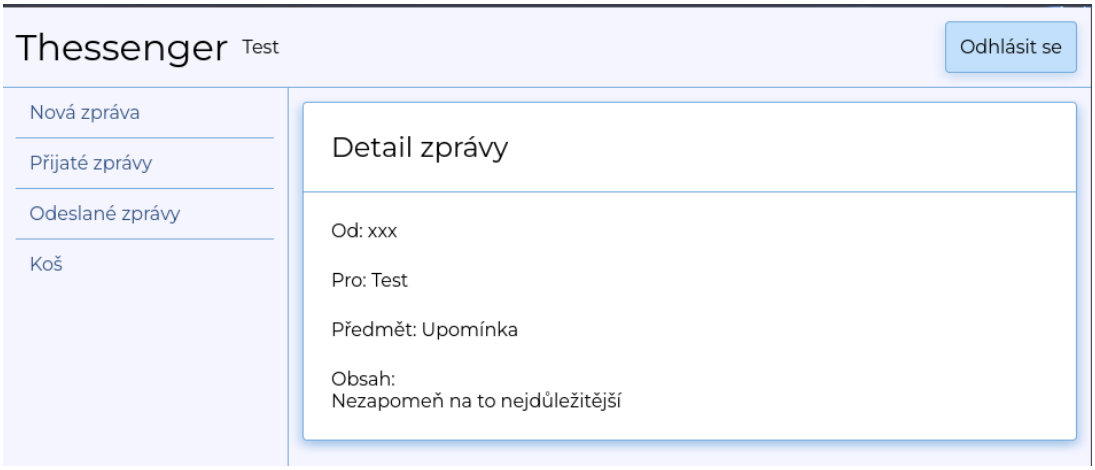
Obrázek 5: Přijaté zprávy.



Obrázek 6: Odeslané zprávy.

1.2.5 Detail zprávy

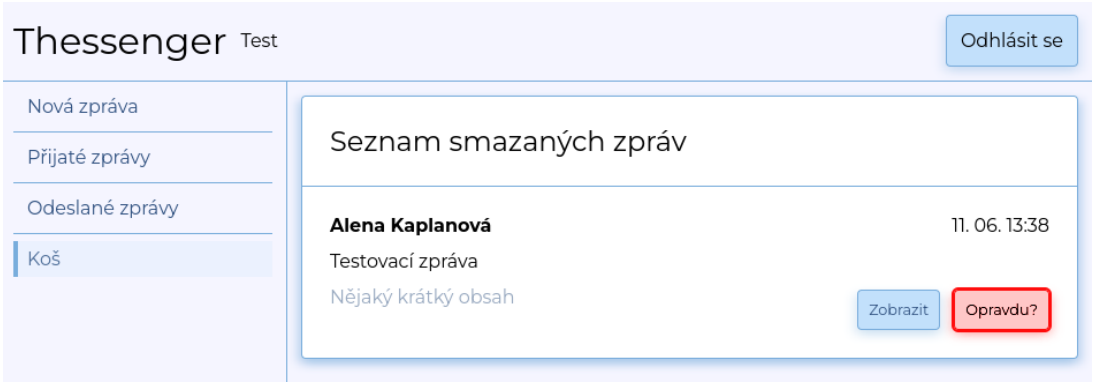
Detail zobrazí celý obsah zprávy, jak můžeme vidět na obrázku 7. Obsah zprávy může být triviálně formátován pomocí nových řádků a mezer. Samotný výpis je obalen v pre značce a tudíž zachovává mezery.



Obrázek 7: Detail zprávy.

1.2.6 Koš

Stránka koše, zobrazená na obrázku 8, nám umožňuje smazat položky navždy.



Obrázek 8: Koš.

2 Implementace

V rámci implementace byl dbán velký důraz na redukci redundantního kódu. Vzhledem k tomu, že máme tři instance té samé aplikace psané pouze v různých frameworcích je očekáváno velké množství stejného kódu.

2.1 Adresářová struktura

Aplikace je logicky rozdělena podle frameworků s dodatečnou složkou pro společný kód.

- `backend` - slouží pro manipulaci a ukládání dat
- `common` - slučuje stejnou funkcionalitu frontendových frameworků
- `solidjs` - implementace frontendu aplikace
- `svelte` - implementace frontendu aplikace
- `vue` - implementace frontendu aplikace

2.2 Backend

Na backend byl zvolen framework Laravel [5]. Vývoj začal právě tímto projektem a k ověření funkčnosti byly využity unit testy. Jediná zajímavá záležitost z pohledu frontendu je vystavení všech dostupných endpointů do JSON API. Po zavolání `api/route/routes` dostaneme odpověď ve formátu zobrazeném ve zdrojovém kódu 1:

```
{
  "auth.login":{
    "url": "api/auth/login",
    "method": "POST"
  }
}
```

Zdrojový kód 1: JSON seznam jednotlivých endpointů.

Z těchto dat se pomocí skriptu vytvoří soubor `routes.ts`, který obsahuje objekt ve formátu zobrazeném ve zdrojovém kódu 2.

```
  export const roues = {
    "message.send": {
      url: (message: string|number) => `/api/message/message/
          ${message}/send`,
      method: "POST"
    }
  }
```

Zdrojový kód 2: Definice jednotlivých endpointů.

Díky tomu mají všechny frontendy staticky definované endpointy. Zároveň se z jednotlivých URL pomocí regexu získá i informace o parametrech. Díky tomu se zmenšuje množství informací, které je nutné znát pro člověka na frontendu. Pohledem na soubor s endpointy získá přehled o tom, co backend umožňuje a jaké parametry jednotlivé endpointy požadují. Při změně endpointů by nám statická analýza řekla, jestli využíváme všechny endpointy korektně. Navíc máme možnost na backendu změnit v případě potřeby URL mez nutnosti úpravy frontendu.

2.3 Common

Adresář společných funkcionalit jednotlivých frontendových implementací. Dělí se na tři větší celky.

- `css`
- `ts/validation`
- `ts/api`

Setkáváme se zde s nutností psaní framework-agnostického kódu. Nejprve nastíníme problém, se kterým se musíme vypořádat. Každý framework řeší reaktivní aktualizaci dat jinak.

```
1  // Vue příklad
2  const count = ref(0);
3  count.value = 42;
4
5  // Svelte příklad
6  let count = 0;
7  count = 42;
8
9  // SolidJS příklad
10 const [count, setCount] = createSignal(0);
11 setCount(42);
```

Zdrojový kód 3: Definice reaktivních proměnných.

Jak je ze zdrojového kódu 3 vidět, Vue [7] využívá funkci `ref` pro vytvoření reaktivního objektu, SolidJS [9] používá funkci `createSignal` pro získání getteru a setteru a Svelte [12] využívá reaktivitu na základě přiřazení.

Zároveň má každý framework řešené komponenty[1] a komunikaci mezi nimi také jinak. Primárně nás pro naše účely zajímá kontextuální předávání informací[2] mezi komponentami. Jednotlivé implementace jsou demonstrovány ve zdrojovém kódu 4.

[1] Komponenty slouží jako základní stavební blok. Definují, jak se mají vykreslit, mají svůj interní stav a dokáží upravit své chování na základě vstupních parametrů nebo oznámit změny pomocí událostí.

[2] V složitější struktuře komponent by mohlo být velmi náročné přeposílat data a události N úrovní hluboko. Kontextuální předávání dat pak umožňuje v komponentě poskytnout data a každá komponenta, která je jakkoli vzdáleným potomkem smí tato data získat.

```
// Vue příklad v nadřazené komponentě
provide('key', 42)
// Vue příklad v podřazené komponentě
const count = inject('key')

// Svelte příklad v nadřazené komponentě
setContext('key', 42)
// Svelte příklad v podřazené komponentě
const count = getContext('key')

// SolidJS příklad v nadřazené komponentě
export const useCount = useContext(createContext(42))
// SolidJS příklad v podřazené komponentě
const count = useCount()
```

Zdrojový kód 4: Kontextuální API.

Řešení samotného problému pak spočívá pouze ve využití getterů a setterů v případě reaktivních dat. Pro potřeby kontextuálního api poskytneme pouze klíč a funkce, které mají být sdíleny s potomky komponenty.

2.3.1 CSS

Všechny projekty by měly vypadat totožně. Dává tedy smysl mít pouze jednu sdílenou sadu CSS pravidel. Je zajištěno, aby byla aplikace použitelná jak na mobilních zařízeních, tak na desktopu.

2.3.2 Ts/validation

Definujeme systém pro validování formulářů. Primárně je tato funkcionalita využita během registrace. Každý formulář poskytne svým potomků možnost registrovat se jako vstupní políčko. Při odeslání formuláře jsou všechna pole zkontrolována, zda je jejich validace platná. Pokud není, formulář nebude odeslán.

3 Porovnávané frameworky

Nejprve si představíme jednotlivé frameworky. Všechny mají společné jedno – snaží se sblížit datovou a vizuální stránku. Pokud bychom tvořili větší interaktivní stránku, velmi rychle dojdeme k závěru, že je vcelku náročné sledovat kde se která data zobrazují a jakým způsobem se modifikují. Představme si jednoduchý příklad, kdy chceme po stisknutí tlačítka navýšit hodnotu interního čítače, který začíná na nule a zobrazit tuto hodnotu v divu. V čistém JavaScriptu [2] bychom museli po načtení stránky získat podle statických id referenci na tlačítko a zobrazovací `div`. Vytvořili bychom proměnnou `count` a nastavili ji na nulu. Museli bychom zaregistrovat odposlech události stisku tlačítka myši na elementu tlačítka. V této události bychom mohli zároveň upravit hodnotu zobrazovacího divu na současnou hodnotu proměnné `count`. Už v takto jednoduchém případě vidíme, že tento přístup není nejpřímočařejší. Při pohledu na html nevidíme nic, co by nám říkalo, že tlačítko má navázanou událost stisknutí tlačítka, nebo že obsah zobrazovacího divu bude upraven. Naopak při pohledu na JavaScript [2] zase nevidíme, s jakými prvky manipulujeme.

Zde přichází na pomoc frameworky. Každý definuje způsob, jak deklarativně pracovat s daty přímo v šabloně. Základním stavebním blokem jsou komponenty. Komponentu si můžeme představit jako třídu. Každá komponenta má svůj interní stav a funkci, kterou se vykreslí. Když pak používáme komponenty je to jako vytvoření instance třídy. Komponenty mají většinou životní cyklus. Tedy funkce, které se volají při vytvoření nové instance (konstruktor) při vyjmutí z DOMu (destruktor) a při aktualizaci DOMu (update).

Samotné komponenty nám umožňují vytvářet znovupoužitelné části kódu, ale abychom opravdu vyřešili problém s neviditelným tokem dat, potřebujeme ještě reaktivní data. Reaktivita je způsob, jak při změně proměnné implicitně aktualizovat všechna místa, kde je tato proměnná použita. Nejedná se pouze o šablony, ale například i proměnné odvozené.

Dejme tomu, že máme reaktivní proměnnou `x`. Můžeme vytvořit proměnnou `y`, která představuje například dvojnásobek proměnné `x`. Pokud změníme hodnotu proměnné `x`, musí se přepočítat i `y` a všechna místa, kde je proměnná `y` použita. Způsob, jakým jednotlivé frameworky tento problém řeší se u každého zvoleného frameworku liší.

3.1 Vue 3

Vue [7] využívá k dosažení rekativity virtuální DOM a proxy objekty [14]. K jednotlivým pojmům se dostaneme po ukázce jednoduchého příkladu Vue komponenty.

3.1.1 Komponenty

Vue [7] má dva způsoby, jak definovat komponenty. Pomocí funkce a pomocí souboru s koncovkou `.vue`. V případě souboru se jedná o takzvanou SFC.[3] Preferovaným způsobem dle dokumentace je psaní SFC komponent. Každá komponenta se skládá ze tří částí zobrazených ve zdrojovém kódu 5. První je šablona, následuje kód a v poslední můžeme definovat styly.

```
1  <template>Šablona</template>
2  <script>Kód</script>
3  <style>Styl</style>
```

Zdrojový kód 5: Vue SFC rozložení.

V rámci kódu komponenty existují dvě API, jakými můžeme definovat reaktivní data. V době Vue 2 [6] existovalo pouze jediné – takzvané Options API (zdrojový kód 6). V části kódu byl zadefinovaný objekt, na kterém mohly existovat vlastnosti `data`, `computed`, `methods` a jiné. Vlastnost `data` definovala funkci, která byla zodpovědná ze vytvoření počátečního stavu komponenty. Všechny vlastnosti vrácené v objektu této funkce byly automaticky reaktivní a dostupné v šabloně. `Data` byla dostupná z metod přes klíčové slovo `this`.

Options API je zastaralé. Reaktivní data nemohla být definována mimo komponentu a logické celky[4] byly fragmentovány na více místech.

```
1    <template>
2      <span>{{a}}</span>
3      <button @click="print">Print!</button>
4    </template>
5
6    <script>
7      return default {
8        data: () => ({
9          a: ""
10       }),
11
12       methods: {
13         print: () => {
14           console.log(this.a);
15         }
16       }
17     }
18   </script>
```

Zdrojový kód 6: Vue Options SFC rozložení.

S příchodem Vue 3 [7] přišlo Composition API (zdrojový kód 7). Definice reaktivních dat byla přesunuta do funkcí. Tento samotný krok přináší několik výhod. Můžeme definovat reaktivní data i mimo komponentu a nemusíme mít

[3] Z anglického Single File Component
[4] Data a metody pracující se těmito daty

fragmentovanou strukturu SFC. Vše, co je zadefinované v části kódu, je dostupné v šabloně.

```
<template>
  <span>{{a}}</span>
  <button @click="print">Print!</button>
</template>

<script setup>
  const a = ref("");

  const print = () => {
    console.log(a.value);
  }
</script>
```

Zdrojový kód 7: Vue Composition SFC rozložení.

Pro nás je nejzajímavější funkce `ref`, která definuje reaktivní proměnnou. Vytvoří to pro nás zástupce, na kterém je vlastnost `value`, přes kterou můžeme získat obsah proměnné nebo jej změnit. V šabloně není nutné volat `.value` na reaktivních proměnných - je to zajištěno automaticky.

Aby mohlo Vue [7] hlídat změny reaktivních dat, vrací funkce `ref` Proxy objekt [14]. Proxy objekty [14] jsou hojně využívány pro sledování změn reaktivních proměnných i mimo framework Vue [7], proto si je v rychlosti představíme.

3.1.2 Proxy

V jednoduchosti má každý objekt v jazyce JavaScript [2] základní množinu metod [4]. Část této množiny můžeme vidět ve zdrojovém kódu 8. Mezi ně patří metody `get` a `set`, které jsou volány při čtení a zápisu vlastností.

```
const test = {a: 42};
console.log(test.a); // Na pozadí spustí metodu get
test.a = 21; // Na pozadí spustí metodu set
```

Zdrojový kód 8: Základní metody objektů.

Proxy nám umožní vytvořit zástupce objektu. Zástupce je nový objekt, u kterého můžeme přetížit chování všech základních metod objektu. Tedy kromě pozorování změn můžeme i změnit nastavovaná data. Proxy je primitivum jazyka JavaScript [2], tedy pro vytvoření nové proxy nám satčí zavolat `new Proxy`, jak můžeme vidět ve zdrojovém kódu 9

```
const target = {};

const proxy = new Proxy(target, options);
```

Zdrojový kód 9: Vytvoření proxy.

Druhým parametrem konstruktoru Proxy je objekt definující přetížení základních metod [4]. Definici takového objektu můžeme vidět ve zdrojovém kódu 10.

```
1  const options = {
2    get(target, prop, receiver) {
3      return Reflect.get(...arguments);
4    },
5
6    set(obj, prop, newval) {
7      Reflect.set(...arguments)
8    }
9  };
```

Zdrojový kód 10: Přetížení metod pomocí Proxy.

V případě, že chceme zachovat výchozí chování přetížené metody, máme k dispozici objekt `Reflect`, který definuje některé z metod, které umožňují Proxy přetížit.

3.1.3 Proxy ve Vue

Se znalostí chování Proxy definujeme naivní implementaci funkce `ref` ve zdrojovém kódu 11.

```
1  const ref = (value) => {
2    return new Proxy({ value: value }, {
3      set(obj, prop, newval) {
4        rebuildComponent();
5        return Reflect.set(...arguments);
6      }
7    })
8  }
```

Zdrojový kód 11: Naivní implementace funkce ref.

Při zavolání naší funkce `ref` dostáváme objekt, na kterém je definovaná vlastnost `value`. Každá změna této vlastnosti zavolá funkci `rebuildComponent`, která je zodpovědná za opravení všech míst, kde se s proměnnou pracuje. V případě Vue [7] tato funkce způsobí kontrolu virtuálního DOMu oproti reálnému DOMu a sestaví se sada opravných změn, které opraví reálný DOM tak, ať odpovídá virtuálnímu.

3.1.4 Options API

Mimo funkci `ref` máme k dispozici ještě `computed` a `reactive`. Funkce `computed` bere jako parametr funkci, která musí vrátit hodnotu. Kdykoli je použita v rámci této funkce reaktivní proměnná, dojde znovu ke spuštění funkce vždy, když se některá z reaktivních proměnných změní. Použití můžeme vidět ve zdrojovém kódu 12.

```
1  const x = ref(2);
2  const xPow2 = computed(() => x.value * x.value)
```

Zdrojový kód 12: Použití computed funkce.

V případě, kdy potřebujeme vytvořit několik reaktivních proměnných najednou, můžeme pomocí funkce `reactive` zreaktivnit všechny vlastnosti na objektu, jak můžeme vidět ve zdrojovém kódu 13.

```
1  const data = reactive({
2    x: 42,
3    y: 41
4  })
5
6  data.y = 42;
```

Zdrojový kód 13: Použití reactive funkce.

U reaktivních objektů nepotřebujeme volat `.value`. Samotná modifikace i jakkoli hlubokých objektů je automaticky detekována.

Nyný máme k dispozici reaktivní data, odvozené proměnné, ale občas potřebujeme reagovat na změnu reaktivní proměnné nějakým vedlejším efektem. Například vždy, když se změní proměnná, vypsat její hodnotu do konzole. K tomu nám slouží funkce `watch`. Ta bere jako první argument reaktivní proměnnou a druhý argument funkci, která se spustí vždy při změně prvního argumentu. Použití můžeme vidět ve zdrojovém kódu 14.

```
1  const x = ref(0);
2  watch(x, (value) => console.log(value));
3  x.value = 42;
```

Zdrojový kód 14: Hlídání změn.

3.2 Svelte

Svelte [12] přistupuje k celému problému reaktivity jinak. Místo virtuálního DOMu přidává kompilační krok. Místo Proxy [14] hlídá změny na základě přiřazení.

3.2.1 Komponenty

Svelte [12] definuje komponenty podobným způsobem jako jsou Vue SFC. Každá komponenta se skládá, stejně jako Vue [7], ze tří částí definovaných ve zdrojovém kódu 15 – kód, šablona a styl. S tím rozdílem, že šablona není obalená v template značce.

```
<script>Kód</script>
<div>Šablona</div>
<style></style>
```

Zdrojový kód 15: Svelte SFC rozložení.

3.2.2 Reaktivita

Na první pohled vypadá kód Svelte komponent ve zdrojovém kódu 16 jako klasický JavaScript [2]. Deklarujeme proměnnou `count`, klasicky ji můžeme upravit a následně ji použít v šabloně.

```
<script>
  let count = 0;

  function handleClick() {
    count += 1;
  }
</script>

<button on:click={handleClick}>
  Clicked {count} {count === 1 ? 'time' : 'times'}
</button>
```

Zdrojový kód 16: Svelte příklad.

Až téměř magicky bude vše fungovat přesně dle očekávání. Stejně jako Vue [7] umožňuje Svelte [12] definovat odvozené proměnné, které můžeme vidět ve zdrojovém kódu 17. Zápis je poněkud nezvyklý, ale umožňuje nám definovat jak odvozené proměnné, tak hlídat změny stavu stejnou syntaxí.

```
<script>
  let count = 0;
  $: doubled = count * 2;

  $: {
    console.log('the count is ' + count);
  }
</script>
```

Zdrojový kód 17: Svelte odvozené proměnné.

Kdykoli se proměnná `count` změní, nejen že se přepočítá proměnná `doubled`, ale zároveň se do konzole vypíše text.

Avšak ne všechno jde takto hladce. Musíme si dávat pozor na to, abychom vždy provedli přiřazení. Nejvíce na toto trpí pole, kde přidání nového záznamu musí být následováno přiřazením pole samo do sebe, nebo s pomocí ES6[5]. Oba

[5]Z anglického ECMAScript 6, což je specifikace definující standard jazyka JavaScript

přístupy jsou demonstrovány ve zdrojovém kódu 18.

```
<script>
  let numbers = [];

  function addNumber() {
    numbers.push(numbers.length + 1);
    numbers = numbers;
  }
  // nebo
  function addNumber() {
    numbers = [...numbers, numbers.length + 1];
  }
</script>
```

Zdrojový kód 18: Svelte problémové části.

U objektů si taktéž musíme dávat pozor na to, aby se vždy vyskytl aktualizovaný objekt na levé části přiřazení. Jinými slovy následující zdrojový kód 19 je v pořádku.

```
<script>
  const x = { y: { z: 42 }};
  x.y.z = 21;
</script>
```

Zdrojový kód 19: Svelte přiřazení objektů správně.

Ale v tomto zdrojovém kódu 20 by změna detekována nebyla.

```
<script>
  const x = { y: { z: 42 }};
  const temp = x.y;
  temp.z = 21;
  // nebo
  function test(thing) {
    thing.y.z = 21;
  }
  test(x);
</script>
```

Zdrojový kód 20: Svelte přiřazení objektů špatně.

3.2.3 Reaktivita mimo komponenty

Aby toho nebylo málo, změny přiřazením jsou detekovány pouze v rámci Svelte komponent. Pokud bychom chtěli mít reaktivní proměnnou mimo komponentu, má pro to Svelte [12] připravené takzvané sklady. [6]

Sklady jsou obyčejné objekty s metodou `subscribe`, která umožní komukoli být upozorněn v případě změny hodnoty skladu. Metoda `subscribe` vrací

[6] Z anglického názvu stores

funkci k zrušení sledování změn. Existují tři typy skladů – pro zápis (writable), pro čtení (readable) a odvozené (derived).

Zapisovatelné sklady jsou obdobou deklarace reaktivní proměnné pomocí funkce `ref` z Vue [7]. Kromě povinné metody subscribe obsahují také metody `set` a `update` k modifikaci stavu. Jednoduché využití skladu můžeme vidět ve zdrojovém kódu 21.

```
// js soubor
export const count = writable(0);
export const addOne = () => count.update((value) => value + 1);
// svelte komponenta
<script>
  let countValue;

  count.subscribe(value => {
    countValue = value;
  });
</script>
<div>{countValue}</div>
```

Zdrojový kód 21: Svelte zapisovatelný sklad.

Tento způsob zápisu je docela zdlouhavý a zároveň náchylný k chybě. V ukázkovém příkladu jsme totiž zapomněli při zničení komponenty zavolat `unsubscribe`. Mohlo by tedy dojít postupem času k úniku paměti. Svelte [12] má pro použití skladů syntaktický cukr. Místo manuálního zápisu a odpisu pro sledování změn můžeme v rámci komponenty použít sklad s prefixem `$`, jak je demonstrováno ve zdrojovém kódu 22. Ukázkový příklad 21 se tím zjednoduší a automaticky se zajistí volání metod `subscribe` a `unsubscribe`.

```
// js soubor
export const count = writable(0);
export const addOne = () => count.update((value) => value + 1);
// svelte komponenta
<script>
</script>
<div>{$countValue}</div>
```

Zdrojový kód 22: Svelte zapisovatelný sklad se syntaktickým cukrem.

Dále můžeme vytvářet sklady, které slouží pouze pro čtení. Tento typ skladů se může využít například pro reaktivní čas nebo pozici myši. Funkce `readable` bere dva parametry. Prvním je výchozí hodnota a druhým funkce, která dostane jako parametr aktualizační funkci a vrací funkci pro zrušení skladu. Definici takového skladu vidíme ve zdrojovém kódu 23, kde definujeme sklad pro reaktivní zobrazení aktuálního času.

```
export const time = readable(new Date(), (set) => {
  const interval = setInterval(() => {
    set(new Date());
  }, 1000);

  return () => {
    clearInterval(interval);
  };
});
```

Zdrojový kód 23: Svelte sklad pro čtení poskytující aktuální čas.

Posledním typem je odvozený sklad. Funkce `derived` opět bere dva parametry. Prvním je sklad, ze kterého má být hodnota odvozená a druhý je funkce pro výpočet odvozené hodnoty.

3.3 SolidJS

Poslední framework je v určitém smyslu na pomezí mezi předchozími frameworky. Jako Vue [7] využívá Proxy objektů [14] k hlídání změn[7] a jako Svelte [12] se vyhýbá virtual DOMu.

3.3.1 Komponenty

Komponenty jsou zde prosté funkce, které vrací JSX. Z tohoto pohledu je SolidJS [9] velmi podobný Reactu [15]. Nemáme tedy žádné speciální soubory s koncovkou `.solid`. Místo toho máme soubory s koncovkou `.tsx` pro TypeScript [3] nebo `.jsx` pro JavaScript [2]. Každý takový soubor může obsahovat i několik komponent. Samotná funkce definující komponentu slouží jako konstruktor. Je volaná pouze jednou. Komponentu pro zobrazení čítače můžeme vidět ve zdrojovém kódu 24.

```
export const Counter = () => {
  const [count, setCount] = createSignal(0);

  const addOne = () => setCount((value) => value + 1);

  return <button onClick={addOne}> {count()} </button>;
}
```

Zdrojový kód 24: SolidJS komponenta.

3.3.2 Reaktivita

Máme k dispozici tři hlavní primitiva, které nám umožňují definovat reaktivní data. Obdobou funkce `ref` z Vue [7] je zde funkce `createSignal`. Místo

[7] Pro hlídání zanořených změn

toho, aby nám vrátila objekt, vrací dvojici `getter` a `setter`. Dále máme funkci `createMemo`, které nám umožňuje definovat odvozenou proměnnou a `createEffect`, kterou můžeme využít pro sledování změn a aplikování vedlejšího efektu. Použití základních funkcí můžeme vidět ve zdrojovém kódu 25.

```
const [count, setCount] = createSignal(0);
createEffect(() => console.log(count()));
const doubleCount = createMemo(() => count() * 2);
```

Zdrojový kód 25: SolidJS reaktivita.

Dále si přiblížíme způsob, jakým SolidJS [9] hlídá změny. Jádrem celého systému je globální zásobník, který se využívá pro automatické detekování závislostí. Ve výsledku to znamená, že není zapotřebí proxy objektů k docílení reaktivity.

Samotná funkce `createSignal` vytvoří lexikálně uzavřené prostředí, ve kterém existuje hodnota signálu a seznam odběratelů změn. Vrací nám zmíněné dvě funkce pro čtení a zápis interní hodnoty. Triviální implementace funkce `createSignal` může být například definována, jako ve zdrojovém kódu 26.

```
function createSignal(value) {
  const subscribers = new Set();

  const read = () => {
    return value;
  };

  const write = (nextValue) => {
    value = nextValue;
  };

  return [read, write];
}
```

Zdrojový kód 26: SolidJS createSignal.

Nyní ke způsobu, jakým dochází k registrování odběratelů změn. Vždy, před spuštěním funkce poskytnuté do funkce
`createEffect` nebo `createMemo` se zaregistrují jako odběratelé do globálního zásobníku. Každé spuštění `getteru` signálu následně způsobí zkontrolování, zda všichni ze zásobníku jsou registrování v interním seznamu odběratelů daného signálu. Rozšířená verze tedy může být definována jako ve zdrojovém kódu 27.

```
function createSignal(value) {
  const subscribers = new Set();

  const read = () => {
    const listener = getCurrentListener();
    if (listener) subscribers.add(listener);
    return value;
  };

  const write = (nextValue) => {
    value = nextValue;
    for (const sub of subscribers) sub.run();
  };

  return [read, write];
}
```

Zdrojový kód 27: SolidJS createSignal s automatickou registrací odběratelů.

Tím máme zajištěné, že při každé změně signálu jsou obeznámeni všichni odběratelé a jejich hodnoty jsou přepočítány.

Samotné šablony komponent se chovají velmi podobně. JSX je v rámci kompilace převedeno na renderovací funkci, která je schopná detekovat, které části DOMu mají být upraveny při změně kterého signálu. Jediná podmínka, která musí být splněna pro správné chování komponent je volání funkce `createRoot`. Ta je zodpovědná za úklid po zaniklých komponentách aby nedocházelo k úniku paměti. Tato funkce se automaticky volá v rámci `render` funkce, která je doporučeným způsobem, jak vytvořit vstupní bod SolidJS [9] aplikace.

Na rozdíl od frameworků využívajících virtuální DOM, kde dochází k renderovacímu cyklu, při němž může dojít během jedné aktualizace k více změnám, SolidJS [9] využívá synchronní aktualizace. Tedy například pokud bychom aktualizovali dvě proměnné `name` a `surname` ve zdrojovém kódu 28, ihned po aktualizaci `name` bude již změna aplikována do DOMu.

```
export const Greeter = () => {
  const [name, setName] = createSignal("Honza");
  const [surname, setSurname] = createSignal("Novák");

  const fullName = () => {
    const result = `${name()} ${surname()}`;
    console.log(`Full name is: ${result}`);
    return result;
  }

  const update = () => {
    setName("Pepa");
    setSurname("Dvořák");
  }

  return <button onClick={update}>{fullName()}</button>
}
```

Zdrojový kód 28: SolidJS synchronní reaktivní aktualizace.

To může mít za následek vícenásobné vyhodnocení odvozených funkcí. Z uvedeného zdrojového kódu 28 bychom po stisknutí tlačítka viděli v konzoli dva záznamy. Prvním by byl „Pepa Novák“ a druhý „Pepa Dvořák“. Abychom se této situaci vyvarovali, můžeme využít funkci `batch`, která odloží aktualizace po skončení všech modifikací. Viz zdrojový kód 29.

```
export const Greeter = () => {
  ...
  const update = () => batch(() => {
    setName("Pepa");
    setSurname("Dvořák");
    })
    ...
    }
```

Zdrojový kód 29: SolidJS synchronní reaktivní aktualizace s batchingem.

Pokud bychom opomenuli odvozené funkce, problém okamžitého projevení do DOMu zmizí. Naopak se občas dostáváme do situace například ve Vue [7], kdy si musíme počkat na aktualizaci DOMu například abychom mohli správně určit šířku nebo výšku elementu. Díky tomu, že SolidJS [9] nemusí porovnávat virtuální DOM s reálným, si může dovolit provést změnu okamžitě, protože cena této operace je stejná, ať se jedná o samostatnou změnu nebo o sadu změn.

3.3.3 Proxy objekty

Na začátku jsme se řekli, že je SolidJS [9] podobný Vue [7] právě v tom, že využívá Proxy objekty. Pokud ale signály nevyužívají Proxy objekty [14] a signály jsou základním stavebním blokem reaktivního systému jako funkce `ref` z Vue [7], k

čemu jsou zapotřebí?

Pravdou je, že nejsou striktně potřebné. Jsme schopni psát celý systém bez Proxy objektů [14]. Někdy nám mohou ale ušetřit práci. Představme si aplikaci, která eviduje nákupní seznam. Můžeme přidat záznam a následně jej v seznamu označit za hotový. Existují z pohledu reaktivity tři způsoby, jak takovou aplikaci implementovat.

Prvním je naivní implementace zobrazena ve zdrojovém kódu 30. Vytvoříme signál `list` reprezentující seznam. Přidání záznamu spočívá ve vytvoření nového pole, kde na konec přidáme nový záznam. Dosud je vše v pořádku. SolidJS [9] se postará o to, aby byl DOM modifikován minimalisticky, tedy pouze přidá nový element na konec. Dále po kliknutí na záznam chceme tento záznam označit za vyřešený. Můžeme stávající seznam projít, a element, na který jsme klikli, nahradit novým elementem, který bude mít nastavený příznak, že byl vyřešen. Pro SolidJS [9] to je ale vytvoření nového elementu a zrušení původního. Místo označení textu například pomocí css bude v DOMu nahrazen celý element. Toto není nejefektivnější, jelikož je celá komponenta zahozena a znovu vytvořena kvůli minimální změně.

```
1   export const App = () => {
2     const [getList, setList] = createSignal([])
3     ...
4     const addItem = (text) => {
5       setList([...getList(), { id: ++itemId, text, completed: false
            }]);
6     }
7     const toggleItem = (id) => {
8       setList(getList().map((item) => (
9         item.id !== id ? item : { ...item, completed: !item.
              completed }
10      )));
11    }
12
13    return (
14      <>
15        ...
16        <For each={getList()}>
17          {(item) => {<span>{item.text} {item.completed ? ' - Hotovo'
                : ''</span>} }}
18        </For>
19      </>
20    );
21  };
```

Zdrojový kód 30: SolidJS naivní modifikace pole.

Druhým způsobem, zobrazeným ve zdrojovém kódu 31, jak se s touto situací vypořádat je využití signálu na úrovni záznamu. Každé vytvoření záznamu vytvoří signál `completed`. Místo samotné hodnoty pošleme do objektu `getter` a `setter` této hodnoty. Při přepnutí stavu záznamu můžeme zavolat poskytnutý

`setter`, tím dojde pouze k úpravě potřebné pro projevení změny stavu. Celá komponenta nebude zničena a znovu vytvořena. Toto řešení je lepší, ale musíme kvůli němu upravit také komponentu pro výpis záznamů. Místo toho abychom pouze vypsali hodnotu `completed` musíme volat `getter`.

```
export const App = () => {
  const [getList, setList] = createSignal([])
  ...
  const addItem = (text) => {
    const [completed, setCompleted] = createSignal(false);
    setList([...getList(), { id: ++itemId, text, completed,
        setCompleted }]);
  }
  const toggleItem = (id) => {
    const item = getList().find((item) => item.id === id);
    if (item) item.setCompleted(value => !value)
  }

  return (
    <>
      ...
      <For each={getList()}>
        {(item) => {<span>{item.text} {item.completed() ? ' -
            Hotovo' : ''</span>} }}
      </For>
    </>
  );
};
```

Zdrojový kód 31: SolidJS modifikace pole s pomocí signálu.

Posledním způsobem, zobrazeným ve zdrojovém kódu 32, je využití skladu. I SolidJS [9] má své sklady, pouze z jiného důvodu, než Svelte [12]. Ve Svelte [12] jsme je využívali ke zprovoznění reaktivity mimo komponenty. Zde takový problém nemáme. Funkci `createSignal` můžeme volat odkudkoli. Sklady nám v rámci SolidJS [9] pomohou s hloubkovou reaktivitou. V předchozím řešení jsme museli upravit komponentu pro zobrazení záznamů, jelikož jsme místo obyčejné hodnoty dostali `getter`. Pokud místo signálu pro reprezentaci seznamu využijeme skladu, můžeme pracovat se záznamem jako v naivní implementaci. SolidJS [9] se postará o vytvoření potřebných signálů na pozadí. Vyváření signálu se navíc provádí pouze pro vlastnosti záznamu, které jsou sledovány, líně až v okamžiku potřeby. Tímto máme řešení, které je definováno podobně přímočaře jako naivní řešení, ale je stejně efektivní jako druhé řešení.

```
1    export const App = () => {
2      const [list, setList] = createStore([])
3      ...
4      const addItem = (text) => {
5        setList([...list, { id: ++listId, text, completed: false }]);
6      }
7      const toggleItem = (id) => {
8        setList(list => list.id === id, "completed", completed => !
             completed);
9      }
10
11     return (
12       <>
13         ...
14         <For each={getList()}>
15           {(item) => {<span>{item.text} {item.completed ? ' - Hotovo'
                 : ''</span>} }}
16         </For>
17       </>
18     );
19   };
```

Zdrojový kód 32: SolidJS modifikace pole s pomocí skladu.

Tedy programátor může rozhodnout, zda chce využít Proxy objektů k zjednodušení definice komponent, nebo zda se chce využití Proxy objektů vyhnout. Starší prohlížeče Proxy objekty nemusejí podporovat, proto se SolidJS [9] hodí i pro vývoj aplikací do různých embed řešení, které mají zastaralý software.

4 Srovnání frameworků

V této kapitole srovnáme jednotlivé frameworky mezi sebou. Nejprve se na ně podíváme z pohledu programátora. Následně se podíváme na vygenerovaný kód jednotlivých frameworků a srovnáme výsledné velikosti.

4.1 Z pohledu programátora

Při psaní kódu je vhodné dodržovat určitý styl. Čím jasněji nám framework diktuje, jakým způsobem bychom měli při psaní kódu postupovat, tím budou existovat menší rozdíly mezi projekty nebo dokonce v rámci projektu.

4.1.1 Vue

Z tohoto pohledu je Vue [7] nejvíc benevolentní. Umožňuje nám definovat komponenty jako funkce nebo SFC a existují 2 reaktivní API z historických důvodů. Nad možností úniku z SFC se dá polemizovat. Možnost definovat anonymní komponenty nebo využívat celé síly JavaScriptu [2] k definování šablon je užitečné, ale Options API je již přežitek, který pouze překáží.

4.1.2 Svelte

Svelte [12] razí svou myšlenku lépe. Existují pouze SFC. Ale díky tomu, že reaktivita funguje implicitně pouze v rámci komponent musíme mít nějaký jiný způsob, jak si poradit s reaktivními daty mimo komponenty. Následně si musíme pamatovat která proměnná je sklad a která ne, jelikož se od toho odvíjí způsob práce s takovými proměnnými.

4.1.3 SolidJS

Nejpřesněji definovanou myšlenku zastává SolidJS [9]. Komponenty jsou pouze funkce, pro definici šablon se využívá JSX. Reaktivita funguje stejně mimo komponentu jako v komponentě. Preferuje imutabilní data, která mohou ve velkých systémech výrazně zjednodušit způsob přemýšlením nad tokem dat.

4.2 Reaktivita

Všechny reaktivní systémy dokáží vyřešit na začátku definovaný problém s neviditelným tokem dat. Některé nesou na svých bedrech roky vývoje, jiné jsou postavené na chybách svých předchůdců. Podíváme se na rychlý přehled jednotlivých reaktivních systému, jejich výhod a nevýhod.

4.2.1 Vue

Ve Vue [7] mohou být reaktivní hodnoty definované pomocí `ref` nebo `reactive`. Ve výsledku to znamená, že mohou být reaktivní data i bez typického `.value`

přístupu. Při použití reaktivní proměnné v šabloně dochází k automatickému rozbalení. Což znamená, že se nemusí volat `.value` na reaktivního hodnotách v šabloně, ale v kódu ano. Takových drobných odchylek skrývá Vue [7] docela hodně.

4.2.2 Svelte

Svelte [12] má problém s detekováním změn. Programátor si musí dávat pozor na to, aby se vždy upravovaný objekt objevil na levé straně přiřazení. Společně s rozdílným způsobem definováním reaktivních dat uvnitř a mimo komponenty je na tom lépe než Vue [7], ale vyžaduje od programátora také neustálou ostražitost a znalost fungování reaktivního systému. Zde přichází další nevýhoda Svelte [12]. Je relativně náročné přesně definovat, co se na pozadí vlastně děje. Ve Vue [7] i SolidJS [9] je vyhodnocovací proces triviální a pokud je potřeba, může se člověk podívat do zdrojového kódu. V případě Svelte [12] jsme schopni říct, jak reaktivní systém funguje, ale zkoumat hlouběji fungování překladače je výrazně náročnější. Navíc dochází k největšímu rozchodu mezi psaným kódem a vyprodukovaným kódem.

4.2.3 SolidJS

SolidJS [9] vychází opět nejlépe. Máme jednotný přístup k reaktivitě pomocí signálů. Je jasně rozlišeno čtení od zápisu. I ve složitějších strukturách můžeme přesně definovat, co je reaktivní a co ne. Synchronní aktualizace DOMu nepřináší nutnost ručního vynucení aktualizace[8].

4.3 Vygenerovaný kód

Využití frameworků nám na jednu stranu ušetří práci, na druhou mohou přinést i nevýhody. Jednotlivé frameworky se mohou lišit ve třech hlavních kategoriích. CPU využití při změně DOMu, paměť a datová velikost přenášených souborů. První dvě kategorie jsou velmi závislé na typu aplikace. U jednoduchého webu jsou téměř až zanedbatelné. Co ale může hrát vcelku velkou roli je výsledná datová velikost aplikace. Zároveň se můžeme podívat, jak jednotlivé frameworky pracují pod pokličkou.

Pro potřeby vygenerovaného kódu byl připraven triviální projekt pro každý framework. Projekt obsahuje pouze čítač s tlačítkem pro inkrementování hodnoty a odvozenou hodnotu.

4.3.1 Vue

Dle očekávání je výsledný soubor největší ze všech tří frameworků. Kromě kódu aplikace potřebujeme celý systém pro generování změn získaných rozdílem vir-

[8] Vue má funkci `nextTick` a Svelte funkci `tick`

tuálního a reálného DOMu. Toto přichází s minimální fixní velikostí vygenerovaného kódu. V našem případě je finální verze minifikovaného kódu 50,3 kB.

Každá komponenta je transformována jako objekt, který udržuje název komponenty a funkce `setup`. Kód zůstává téměř netknutý, pouze šablona se překompilovává do renderovací funkce. Pro srovnání můžeme vidět komponentu před sestavením ve zdrojovém kódu 33 a po sestavení ve zdrojovém kódu 34.

```
<script setup>
  import { ref, computed } from "vue";

  const value = ref(0);
  const double = computed(() => value.value * 2);

  const increase = () => (value.value = value.value + 1);
</script>

<template>
  <div>Counter: {{ value }}, Double: {{ double }}</div>
  <button type="button" @click="increase">Increase</button>
</template>
```

Zdrojový kód 33: Srovnání kódu Vue před sestavením.

```
...
const _sfc_main = {
  __name: "App",
  setup(__props) {
    const value = ref(0);
    const double = computed(() => value.value * 2);
    const increase = () => value.value = value.value + 1;
    return (_ctx, _cache) => {
      return openBlock(), createElementBlock(Fragment, null, [
        createBaseVNode("div", null, "Counter: " + toDisplayString(
            value.value) + ", Double: " + toDisplayString(double.
            value), 1),
        createBaseVNode("button", {
          type: "button",
          onClick: increase
        }, "Increase")
      ], 64);
    };
  }
};
```

Zdrojový kód 34: Srovnání kódu Vue po sestavení.

U složitějších aplikací může dojít k rozkouskování kódu do menších celků potřebných pro fungování konkrétní stránky v případě, že knihovna pro simulování navigace mezi stránkami podporuje dynamické importování. Aplikace Thessenger se skládá z 21 malých souborů, kde celková velikost činí 113,8 kB.

4.3.2 SolidJS

U SolidJS [9] jsme schopní se dostat velikostí výsledného souboru výrazně níž. Minifikovaná verze má pouze 8kB. Obdobně jako u Vue [7] dochází k větším úpravám pouze v šabloně. Komponentu před sestavením můžeme vidět ve zdrojovém kódu 35 a po sestavení ve zdrojovém kódu 36.

```
function App() {
  const [getValue, setValue] = createSignal(0);
  const getDouble = () => getValue() * 2;
  const increase = () => setValue((value) => value + 1);

  return (<>
    <div>Counter: { getValue() }, Double: { getDouble() }</div>
    <button type="button" onClick={increase}>Increase</button>
  </>);
}
```

Zdrojový kód 35: Srovnání kódu SolidJS před sestavením.

```
...
const _tmpl$ = /*#__PURE__*/template(`<div>Counter: <!>, Double:
    `),
  _tmpl$2 = /*#__PURE__*/template(`<button type="button">Increase
     `);
function App() {
  const [getValue, setValue] = createSignal(0);
  const getDouble = () => getValue() * 2;
  const increase = () => setValue(value => value + 1);
  return [(() => {
    const _el$ = _tmpl$(),
      _el$2 = _el$.firstChild,
      _el$4 = _el$2.nextSibling;
      _el$4.nextSibling;
    insert(_el$, getValue, _el$4);
    insert(_el$, getDouble, null);
    return _el$;
  })(), (() => {
    const _el$5 = _tmpl$2();
    _el$5.$$click = increase;
    return _el$5;
  })()];
}
```

Zdrojový kód 36: Srovnání kódu SolidJS po sestavení.

Aplikace Thessenger vygeneruje 15 souborů s celkovou velikostí 57 kB.

4.3.3 Svelte

Svelte [12] dokázal vyprodukovat nejmenší soubor s velikostí 4,4kB. Dochází zde k větším úpravám jak samotného kódu, tak šablony. Komponentu před sestaveím

můžeme vidět ve zdrojovém kódu 37 a po sestavení ve zdrojovém kódu 38.

```
<script>
  let value = 0;
  $: double = value * 2;

  const increase = () => (value = value + 1);
</script>

<div>Counter: {value}, Double: {double}</div>
<button type="button" on:click={increase}>Increase</button>
```

Zdrojový kód 37: Srovnání kódu Svelte před sestavením.

```
...
function instance($$self, $$props, $$invalidate) {
  let double;
  let value = 0;
  const increase = () => $$invalidate(0, value = value + 1);
  $$self.$$.update = () => {
    if ($$self.$$.dirty & /*value*/
    1) { $$invalidate(1, double = value * 2); }
  };
  return [value, double, increase];
}
class App extends SvelteComponent {
  constructor(options) {
    super();
    init(this, options, instance, create_fragment, safe_not_equal,
        {});
  }
}
```

Zdrojový kód 38: Srovnání kódu Svelte po sestavení.

Svelte [12] router nepodporoval roztříštění kódu dle jednotlivých stránek, máme proto pouze jediný soubor s velikostí 69,9 kB.

Závěr

Se svojí zkušeností ve Vue [7] jsem aplikaci začal psát právě v tomto frameworku. Přesně jsem věděl co očekávat a na co si dát pozor. Svelte [12] byl framework, který jsem delší dobu sledoval se vyvíjet, proto byl druhý v pořadí. Přepis z Vue [7] do Svelte [12] byl poněkud krkolomný. Přístup k reaktivitě je dost odlišný, takže bylo nutné některé části více upravit i logicky. U SolidJS [9] jsem byl nejvíce překvapen. Překlopení z Vue [7] probíhalo velmi jednoduše. Reaktivní přístupy obou frameworků jsou si velmi podobné.

Od Svelte [12] jsem očekával víc, než co doručil. Hlídání změn pomocí přiřazení sice zní dobře na papíře, ale v realitě jsem se s tím vcelku pral. Oproti tomu SolidJS [9] předčil veškerá očekávání. Jasný přístup velmi podobný Vue [7], jen bez nutnosti virtuálního DOMu. Tedy menší velikostně a méně výpočetně náročné aplikace. Zároveň řeší reaktivitu velmi elegantně a precizně. Ani jeden z ostatních frameworků neumožňuje tak přesně a chirurgicky definovat reaktivní data a změny způsobené do DOMu.

SolidJS [9] a Svelte [12] je zatím mladou konkurencí a je to znát na kvalitě a množství balíčků oproti Vue [7].

Conclusions

Thesis conclusions in “English”.

A První příloha

Text první přílohy

B Druhá příloha

Text druhé přílohy

C Obsah elektronických dat

Na samotném konci textu práce je uveden stručný popis obsahu elektronických dat odevzdaných v systému katedry informatiky spolu s textem. Tato data jsou nedílnou součástí práce a tvoří (datovou) přílohu textu práce. Povinné položky struktury dat jsou:

`text/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu (případně v ZIP archivu), tj. zdrojový text textu a příloh, vložené obrázky, apod.

`README.*`
: Textový soubor (s příponou např. `.txt`) s informacemi o opakovatelném způsobu použití ostatních dat práce – typicky plně reprodukovatelný co nejúplnější funkční postup zprovoznění software vytvořeného v rámci práce, tzn. jeho případné instalace/nasazení a spuštění, včetně uvedení všech požadavků pro bezproblémový provoz; za zprovoznění software se nepovažuje zpřístupnění (např. po Internetu) již někde zprovozněného software.

`*`
: Adresáře a soubory s veškerými ostatními autorskými daty práce (případně v ZIP archivu) – typicky spustitelné a další soubory software vytvořeného v rámci práce potřebné pro bezproblémový provoz software, případně jeho instalační program, a kompletní zdrojové texty software a další data nutná pro plně reprodukovatelné korektní vytvoření spustitelných souborů.

Dále mohou data obsahovat například:

- ukázková a testovací data použitá v práci nebo pro potřeby posouzení práce v rámci její obhajoby,
- položky bibliografie v elektronické podobě, příp. jiná relevantní literatura a dokumentace vztahující se k práci,

- cizí data (software) potřebná pro bezproblémové použití autorských dat práce (software), která nejsou standardní součástí předpokládaného (softwarového) vybavení uživatele.

U veškerých cizích obsažených materiálů jejich zahrnutí dovolují podmínky pro jejich veřejné šíření nebo přiložený souhlas držitele práv k užití. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy, je uveden jejich zdroj, např. webová adresa, v bibliografii nebo textu práce nebo souboru `README.*`.

Literatura

[1] Php. Dostupný z: https://www.php.net/

[2] JavaScript. Dostupný z: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[3] TypeScript. Dostupný z: https://www.typescriptlang.org/

[4] Základní metody objektu z jazyka JavaScript. Dostupný z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy/Proxy#handler_functions

[5] Laravel framework. Dostupný z: https://laravel.com/

[6] Vue 2 framework. Dostupný z: https://v2.vuejs.org/

[7] Vue framework. Dostupný z: https://vuejs.org/

[8] Vue – reactivity fundamentals. Dostupný z: https://vuejs.org/guide/essentials/reactivity-fundamentals.html

[9] SolidJS. Dostupný z: https://www.solidjs.com

[10] SolidJS – reactivity. Dostupný z: https://www.solidjs.com/guides/reactivity

[11] SolidJS – rendering. Dostupný z: https://www.solidjs.com/guides/rendering

[12] Svelte. Dostupný z: https://svelte.dev/

[13] Svelte – reactivity. Dostupný z: https://learn.svelte.dev/tutorial/reactive-assignments

[14] Proxies. Dostupný z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

[15] React. Dostupný z: https://react.dev/