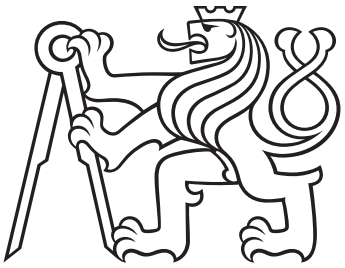
České vysoké učení technické v Praze
Fakulta elektrotechnická

Obor: Softwarové inženýrství a technologie



Modernizace aplikace Flagis.com

Modernization of Flagis.com

BAKALÁŘSKÁ PRÁCE

Vypracoval: Matouš Najman
Vedoucí práce: Bc. Petr Huťák
Rok: 2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Najman** Jméno: **Matouš** Osobní číslo: **499148**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Modernizace Webové Aplikace Flagis.com

Název bakalářské práce anglicky:

Modernization of Web Application Flagis.com

Pokyny pro vypracování:

Cílem projektu je modernizovat stávající webovou aplikaci a provést její přepsání s důrazem na aktualizaci Reactu na verzi 18 a integraci TypeScriptu pro zkvalitnění vývoje. Vedlejším cílem je migrace z Reduxu na jednodušší knihovnu na správu globálního stavu aplikace. Kromě toho je potřeba sledovat některé klíčové aspekty projektu, jako je stav balíčků a výkon aplikace.
Kroky projektu:
1. Analýza Stávající Aplikace: Proveďte detailní analýzu stávající aplikace, včetně její architektury, technologií a knihoven. Seznamte se s aplikací.
2. Migrace na nový bundler: Proveďte analýzu možných řešení. Odstraňte react-boilerplate knihovnu a převeďte aplikaci na nový bundler. Cílem je spustit aplikaci se starými knihovnami s novým bundlerem.
3. Aktualizace na React 18: Proveďte aktualizaci Reactu na verzi 18. Sledujte stav aktualizace a její dopady na chování aplikace.
4. Update Balíčků: Proveďte zdravotní kontrolu knihoven a identifikujte problematické knihovny. Měřitelným cílem bude snížení počtu starých, neudržovaných a nepoužívaných knihoven. Aktualizujte balíčky a vyřešte konflikty s problémovými knihovnami nahrazením nebo smazáním knihoven.
5. Přidání Typescriptu, GraphQL codegen: V projektu začněte efektivně používat Typescript. Začněte přemýšlet o zavedení GraphQL do projektu. Zvažte použití codegenu pro zavedení typů do projektu.
6. Omezení závislosti na knihovně Redux: Zkuste navrhnout postup jak omezit závislosti na knihovně Redux. Jaká jsou dostupná řešení? Cílem je postupný proces aktualizace na novou knihovnu.
7. Zhodnocení a Budoucí Vývoj: Zhodnoťte výsledky práce a navrhněte případné další funkcionality nebo zlepšení a další postup na modernizaci aplikace.
Cíle:
Výrazné zlepšení developer experience, vývoj nebrzdí staré a neudržované knihovny. Aplikace funguje na novém bundleru. Aplikace běží na React 18 a používá Typescript. Vedlejším cílem je omezení závislosti na knihovně Redux. Výkon aplikace by měl být lepší, nebo alespoň srovnatelný.

Seznam doporučené literatury:

[1] Roger S. Pressmann, Bruce Maxim: Software Engineering: A Practitioner's Approach, ISBN-10: 9780078022128
[2] Oficiální dokumentace technologií React, Typescript a příslušných knihoven
[3] Články a zdroje týkající se modernizace webových aplikací a vývoje v React s Typescriptem a GraphQL
[4] S. Mujahid, D. E. Costa, R. Abdalkareem and E. Shihab, "Where to Go Now? Finding Alternatives for Declining Packages in the npm Ecosystem," 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, Luxembourg, 2023, pp. 1628-1639, doi: 10.1109/ASE56229.2023.00119.
[5] Kruchten, Philippe, Robert Nord, and Ipek Ozkaya. Managing Technical Debt. Addison-Wesley Professional, 2019. http://books.google.ie/books?id=y4AMvgEACAAJ&dq=Managing+Technical+Debt:+Reducing+Friction+in+Software+Development&hl=&cd=1&source=gbs_api
[6] Cherny, Boris. Programming TypeScript. O'Reilly Media, 2019. http://books.google.ie/books?id=YmUDwAAQBAJ&printsec=frontcover&dq=Programming+TypeScript:+Making+Your+JavaScript+Applications+Scale&hl=&cd=1&source=gbs_api

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Bc. Petr Huťťák katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **16.02.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

Bc. Petr Huťťák podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne .....

.....
Matouš Najman

Poděkování

Děkuji Bc. Petru Hurťákovi za vedení mé bakalářské práce a za neocenitelné rady a pomoc při tvorbě práce. Dále bych chtěl poděkovat kolegovi Ing. Patriku Krhovskému za spolupráci na modernizaci aplikace Flagis.

Matouš Najman

Název práce:
Modernizace aplikace Flagis.com

Autor: Matouš Najman

Obor: Softwarové inženýrství a technologie
Druh práce: Bakalářská práce

Vedoucí práce: Bc. Petr Huťťák

Klíčová slova: React, technický dluh, aktualizace balíčků, Redux, migrace na GraphQL

Abstrakt: Tato práce se zaměřuje na modernizaci webové aplikace Flagis, která čelí významnému technologickému dluhu. Hlavním cílem byla aktualizace knihovny React z verze 16 na verzi 18. Pro dosažení tohoto cíle byl proveden postupný proces modernizace.

Prvním krokem v tomto procesu byla migrace ze šablonového řešení react-boilerplate na rychlý a moderní bundler Vite.

Dalším klíčovým krokem byla aktualizace balíčků. Tento proces zahrnoval pročištění používaných knihoven, nahrazení problémových a neudržovaných knihoven a aktualizaci verzí knihoven, aby byly kompatibilní s poslední verzí React 18. Během aktualizace byl proveden také refaktoring návrhového vzoru HOC na Hooks a do projektu byl zaveden TypeScript místo JavaScriptu.

Následovaly kroky k odstranění knihovny Redux z projektu. Tento proces byl složitý, ačkoli se podařilo omezit závislost na knihovně Redux a změnit komunikační technologii na GraphQL. Pro tento účel byla zvolena kombinace knihoven ApolloClient a Zustand jako alternativa k Reduxu. V práci byl také navržen budoucí postup pro úplné odstranění knihovny Redux.

Výsledkem této práce je modernizovaná aplikace využívající nejnovější technologie a přístupy, které zlepšují udržitelnost aplikace a podporují vývoj nových funkcionalit.

Obsah

Seznam použitých zkratek

HOC Higher-Order Components

JSX Rozšíření JavaScript syntaxe pro React

SPA Single Page Application

MVC Model View Controller

API Application Programming Interface

IDE Integrated Development Enviroment

UI User Interface

Seznam obrázků

Kapitola 1

Úvod

Frontend vývoj je proces tvorby prezentační vrstvy webové aplikace. V dnešním digitálním světě se webová aplikace neobejde bez JavaScriptu, jelikož přibližně 98,7 % všech webových stránek a aplikací využívá programovací jazyk JavaScript [1]. Moderní vývoj webových aplikací tlačí poptávku na sofistikovanější nástroje než jen základní trio HTML, CSS a JS. S narůstající komplexitou projektů se stává tento minimalistický přístup neudržitelným a neefektivním, což vyvolává potřebu využít frontendové frameworky [2]. Tyto frameworky poskytují strukturovaný a efektivní způsob práce. Mezi nejznámější frontendové frameworky patří React, Vue, Svelte a dalších 28 open-source frontendových frameworků, které jsou dostupné v GitHub kolekci [3]. Nepřibývaly jenom frameworky, v registru npm k roku 2022 můžeme napočítat přes 2 miliony knihoven, které lze použít k rozšíření funkcionality či řešení problému [4]. Pokud se podíváme například na framework React, lze vidět, že v komplexnějších aplikacích, které potřebují globální správu stavu, je potřeba využití externí knihovny jako je Redux či Zustand [5].

Se vznikem frameworků a nových knihoven ovšem také vznikají nové problémy, které musí frontend vývojáři řešit. Se zrychlujícím se tempem vývoje aplikací se zrychluje i tempo nárůstu technického dluhu, jelikož údržba frameworků a knihoven je stále složitější. V rámci studie z roku 2023 9 z 10 vývojářů souhlasilo, že práce se závislostmi na další knihovnách vnímají jako náročnou a frustrující [6].

1.1 Struktura práce

Tato práce se zaměřuje na modernizaci aplikace Flagis, softwarový nástroj pro plánování a organizaci úkolů, a vysvětluje postup modernizace této existující a fungující aplikace. Ve druhé kapitole představuje aplikaci Flagis a její funkcionality. Ve zbývajících kapitolách se postupně věnuje analýze současného stavu, migraci na nový bundler, aktualizaci Reactu a dalších knihoven, přidání TypeScriptu a postupnému omezování závislosti na knihovně Redux.

1.2 Cíle a očekávané výstupy projektu

Hlavním cílem tohoto projektu je modernizovat webovou aplikaci Flagis s důrazem na zlepšení vývojového prostředí a migraci na moderní technologie. Očekáváme, že tato modernizace umožní výrazně snažší vývoj nových funkcionalit a odstraní překážky spojené se zastaralými knihovnami a technologiemi. Přestože samotná uživatelská zkušenost zůstane nezměněná, modernizace položí pevný základ pro budoucí rozvoj a inovace v rámci aplikace Flagis.

Konkrétní cíle:

1. Aplikace běží na React 18.
2. Aplikace běží na moderním bundleru (Vite).
3. Aplikace má aktualizované balíčky kompatibilní s React 18. Aktuální aplikace využívá celkem 97 balíčků ve starší verzi:
   - Major: 72 balíčků
   - Minor: 18 balíčků
   - Patch: 7 balíčků

   Hlavní prioritu při aktualizaci mají knihovny React, Redux a Recompose. Součástí aktualizace by mělo být také odstranění nepoužívaných knihoven a balíčků.
4. Aplikace bude alespoň stejně výkonná, co se týče rychlosti načítání v prohlížeči.
5. Aplikace využívá TypeScript. Typy se generují automaticky z GraphQL schématu pomocí knihovny Codegen.
6. Aplikace využívá GraphQL s Apollo Clientem a granulárně se zbavuje závislosti na knihovně Redux.

Kapitola 2

Aplikace Flagis

V dnešní rychle se vyvíjející digitální éře, kde efektivní nástroje pro organizaci času a práce hrají klíčovou roli v každodenním životě, se projekt Flagis.com stal prostředkem, který nejen usnadňuje, ale také obohacuje pracovní prostředí uživatelů. Od svého vzniku v roce 2019 se tato webová aplikace stala sofistikovaným nástrojem pro organizaci úkolů, zdůrazňujícím jednoduchost použití a zároveň poskytujícím pokročilé funkce pro plánování a sdílení úkolů v týmu.

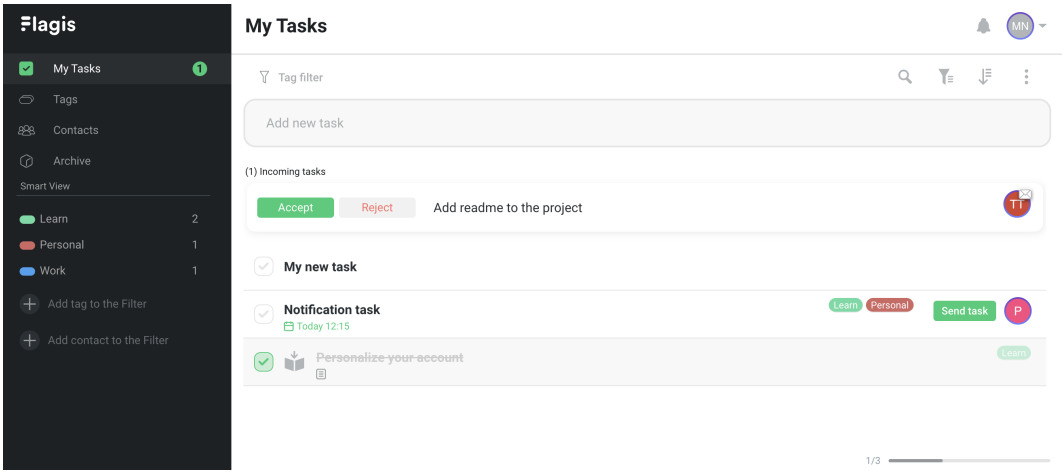
2.1 Seznámení s funkcionalitou a strukturou aplikace

2.1.1 Klíčové funkce aplikace

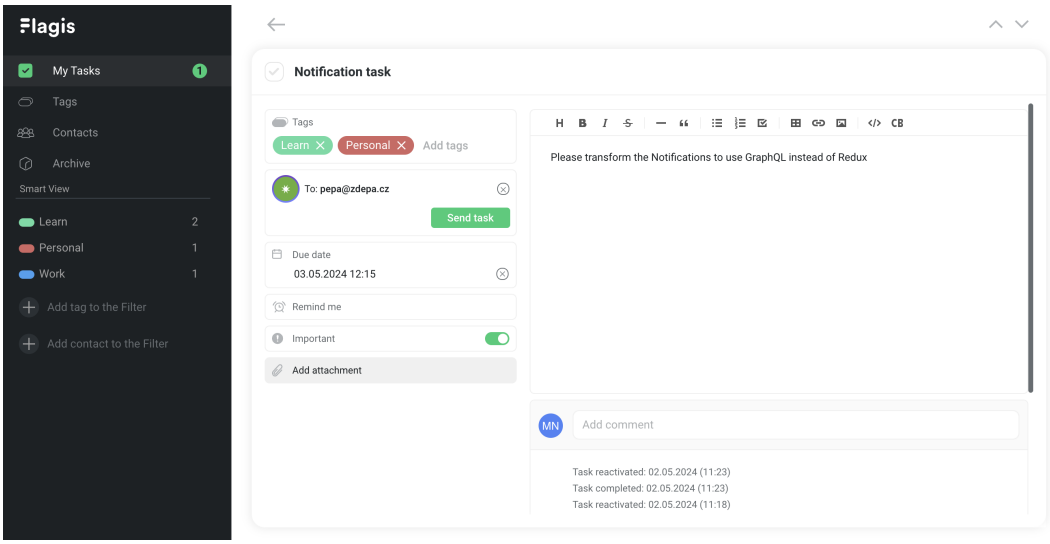
Aplikace Flagis poskytuje uživatelům robustní sadu funkcí pro správu úkolů a jednodušších projektů. Mezi klíčové patří možnost vytváření vlastních úkolů s detailními informacemi, jako jsou přidávání tagů, přiřazování úkolů jiným uživatelům, stanovení termínu splnění a připojení příloh. Dále umožňuje víceúrovňové řazení a filtrování úkolů, full-textové vyhledávání, archivaci úkolů a základní zobrazení statistik plnění úkolů.

2.1.2 Uživatelské rozhraní

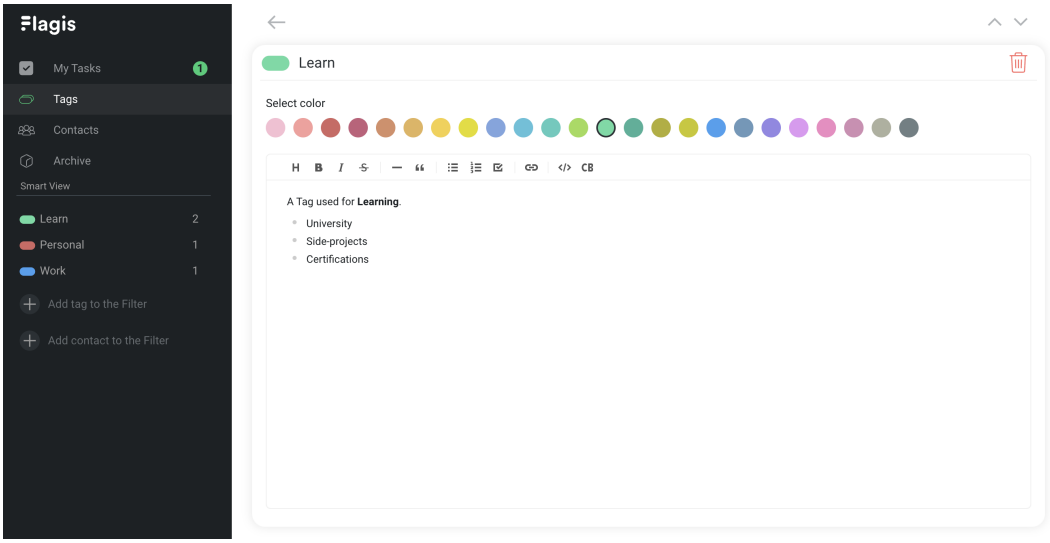
Uživatelské rozhraní je přehledné a moderní, s centrální obrazovkou pro úkoly, bohatými možnostmi filtrování a vyhledávání. Nicméně, v rámci modernizace je nutné zajistit, aby designové změny zůstaly minimální a uživatelsky přívětivé. Plánovaná modernizace by se měla soustředit pouze na aktualizaci použitých technologií, aniž by zásadně měnila uživatelskou zkušenost.



Obrázek 2.1: Hlavní stránka Flagis: My Tasks



Obrázek 2.2: Flagis: Detail úkolu



Obrázek 2.3: Flagis: Detail Tagu

2.2 Stav současné aplikace

Současný stav aplikace Flagis představuje kombinaci nových obchodních příležitostí a existujícího technického dluhu, který brání v jejím dalším rozvoji. S rozšířením možností aplikace se objevila i potřeba reagovat na nové výzvy a požadavky trhu. Bohužel, technický dluh, který se během času vytvořil, se stal překážkou pro efektivní implementaci nových funkcionalit. Tento dluh nejenže zpomaluje vývoj nových funkcí, ale také vyžaduje neustálé záplaty a řešení chyb, což zatěžuje vývojový tým a brání mu v plném využití jeho potenciálu.

2.3 Důvody pro modernizaci

Existují dva hlavní problémy spojené s nahromaděným technickým dluhem.

Prvním problémem je blokace implementace nových funkcionalit. Například přidání funkcí jako je přihlášení přes Google je zastaveno nedostatečnou kompatibilitou současných technologií s požadavky nových knihoven. To vytváří překážku ve vývoji a brání rychlému reagování na požadavky uživatelů.

Druhým problémem je nadměrný čas, který vývojáři musí trávit opravami chyb, které vznikají zastaralými knihovnami a technologiemi. Tento cyklus oprav zpomaluje vývoj nových funkcí a vede k frustraci v týmu.

Kapitola 3

Analýza dosavadního stavu aplikace

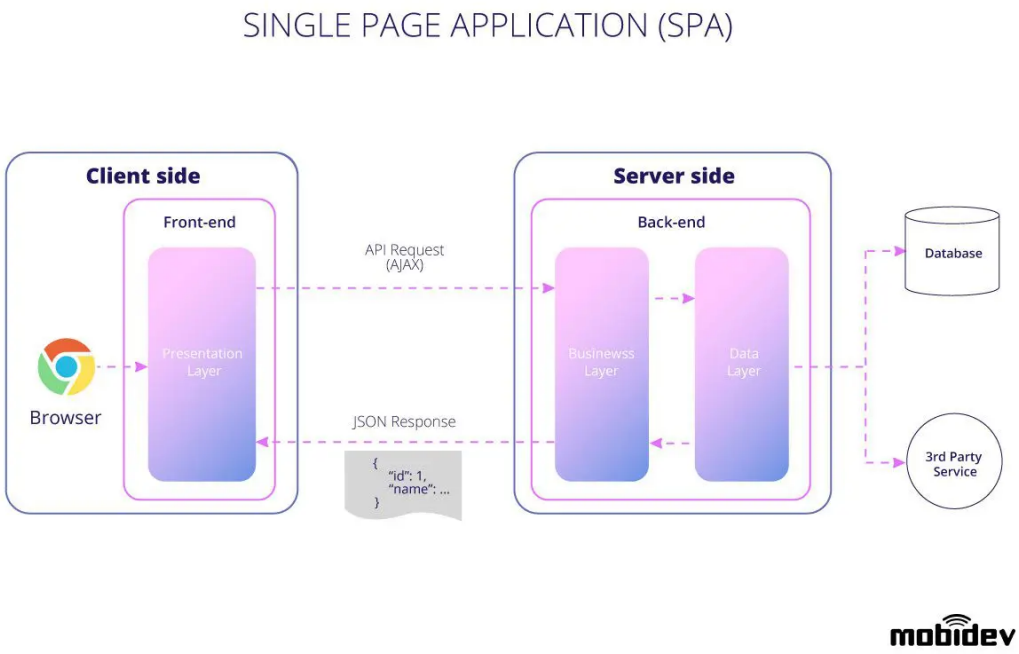
3.1 Popis dosavadní architektury aplikace

Aplikace Flagis je vyvinuta jako SPA[1] s využitím knihovny React. Architektura aplikace je postavena na konceptu HOC[2], který je rozšířen knihovnou Recompose. Aplikace využívá na state management knihovnu Redux, v rámci které je implementován také routing nebo validace formulářů. Ke komunikaci s backendovým API je použita knihovna Axios pro odesílání HTTP požadavků.

3.1.1 Koncept SPA

Aplikace Flagis využívá koncept SPA, což umožňuje uživatelům plynulý a interaktivní zážitek bez nutnosti načítání nových stránek. Tento přístup začíná tzv. "shell", což je jednoduchý HTML soubor, který slouží jako výchozí bod aplikace. Tento soubor je načten pouze jednou a slouží jako základ pro zbytek aplikace. Jedná se o jediné plné načtení v prohlížeči, které se provede při spuštění SPA. Následné části aplikace, jako jednotlivé stránky nebo JavaScript potřebný pro vykreslení těchto stránek, jsou pak načítány dynamicky pomocí JS. Tento HTML soubor má často minimální strukturu a obsahuje pouze prázdný script tag, který načte všechna data z API, vykreslí HTML a CSS. Tento přístup zvyšuje rychlost a efektivitu aplikace při routingu a načítání dalších stránek, ovšem SPA aplikace mají pomalejší úvodní načtení. Aplikace SPA vyžadují vyšší pozornost v oblasti optimalizace pro vyhledávače, ale toto je často řešeno oddělenou landing page [7].

[1] Single Page Application
[2] Higher-Order Components



Obrázek 3.1: SPA: diagram [8]

3.1.2 React

React je knihovna pro tvorbu uživatelských rozhraní webových aplikací. Pomáhá vytvářet interaktivní webové stránky a aplikace tak, že rozděluje je do malých, snadno spravovatelných částí nazývaných "komponenty". Tyto komponenty jsou jako stavební bloky, které můžete znovu používat na různých částech vaší aplikace [9].

React je oblíbený pro svou jednoduchou syntaxi a způsob práce s daty, který zjednodušuje psaní kódu. S Reactem můžete jednoduše aktualizovat obsah na stránce bez nutnosti obnovy celého webu. Nicméně, React je často popisován jako pouze View v rámci architektury MVC[3], což znamená, že neřeší globální správu stavu ani routing a další funkcionality. Z tohoto důvodu je často nutné použití další knihovny nebo frameworku v komplexních aplikacích.

3.1.3 Redux

Pro správu stavu aplikace je využita knihovna Redux, což nám umožňuje centralizovaně ukládat a aktualizovat data napříč aplikací pomocí tzv. akcí (actions), selektorů (selectors) a reducerů. Tento přístup poskytuje centralizovaný a předvídatelný způsob správy stavu, ale zvyšuje složitost aplikace pro menší projekty a vyžaduje dodatečnou konfiguraci [10].

[3]Model View Controller

3.2 Identifikace slabých míst v aplikaci

3.2.1 Problematika s React 18 a balíčky

Problém spočívající v zastaralé verzi Reactu (verze 16) a rozsáhlém množství závislých balíčků, které blokují přechod na novější verzi, představuje výzvu, která vyžaduje pečlivý a postupný přístup k řešení. Primárním krokem v této fázi je nejen odstranění nepoužívaných balíčků, což je klíčové pro uvolnění prostoru pro nové aktualizace, ale také identifikace knihoven, které již nepodporují React 18, ačkoliv byly v minulosti aktivně využívány. Tyto neudržované knihovny se stávají překážkou v procesu aktualizace a vyžadují podrobný přístup k jejich nahrazení modernějšími alternativami. Je třeba zdůraznit, že odstraňování nepoužívaných knihoven představuje pouze první krok v sérii, kdy následující fáze zaměřené na identifikaci a řešení problémů s knihovnami budou klíčovým prvkem. Tyto kroky jsou nezbytné pro plynulý postup směrem k Reactu 18 a zároveň představují nezbytný proces pro minimalizaci technického dluhu.

3.2.2 Inicializace projektu přes React-boilerplate

Dalším výrazným slabým místem je použití knihovny React-boilerplate, která byla využita při inicializaci projektu. Tato knihovna, přestože poskytuje komplexní řešení pro rychlý start React aplikace s vlastním serverem, testy, eslint a dalšími nástroji, představuje vážný problém v kontextu modernizace. Aktuální verze této knihovny je neudržovaná od roku 2019, což znamená, že nespadá pod aktivní vývoj a aktualizace. Její zastaralost se projevuje v nedostatečné podpoře novějších verzí Reactu a problémech s kompatibilitou s aktuálními balíčky. Toto slabé místo vyžaduje specifický přístup, a to migraci na nový bundler. Bundler je nástroj využívaný ve vývoji webových aplikací, který zpracovává zdrojové soubory aplikace do statických prostředků, které jsou využívány prohlížeči. Hlavním účelem bundlerů je organizovat moduly a závislosti ve frontend projektech, optimalizovat výkon webových stránek a snižovat velikost souborů [11].

Migrace na nový bundler je klíčovým krokem v procesu modernizace, umožní odstranění závislosti na neudržované knihovně a zároveň umožní efektivnější správu balíčků a závislostí, čímž podpoří celkovou stabilitu a udržitelnost aplikace. Migrace na nový bundler by měla také přinést jednoduchost konfigurace aplikace [12].

3.2.3 Zastaralý návrhový vzor HOC

Používání návrhového vzoru HOC (Higher Order Component) je v současném kontextu považováno za zastaralé a nepřináší výhody, které přináší modernější přístupy, jako jsou React Hooks. Od verze 16.8 byly Hooks přidány do Reactu, a oficiální dokumentace zdůrazňuje, že by se měly preferovat před HOC, zejména pokud HOC renderuje pouze jednoho potomka, což platí pro většinu případů v naší aplikaci. Většina našich komponent zpravidla renderuje pouze jednoho potomka, což podpořilo přechod na Hooks.

> *React 16.8 documentation (Adoption Strategy) [13]*
> **Do Hooks replace render props and higher-order components?**
> Often, render props and higher-order components render only a single child. We think Hooks are a simpler way to serve this use case. There is still a place for both patterns (for example, a virtual scroller component might have a renderItem prop, or a visual container component might have its own DOM structure). But in most cases, Hooks will be sufficient and can help reduce nesting in your tree.

Dalším klíčovým krokem v modernizaci aplikace je refaktoring HOC, zejména v případě používání knihovny Recompose, která již není udržována a brání aktualizaci na React 18. Při průzkumu HOC versus React Hooks jsem zvážil kritéria jako je čitelnost, manipulace s parametry, state management, přepoužitelnost, výkon. Průzkumem jsem došel k závěru, že migrace na Hooks je logický a nutný krok k modernizaci aplikace.

I přesto, že tato změna může být pracná, identifikovali jsme 10 komplexnějších komponent, které vyžadují náročnější refaktoring kvůli práci s Redux storem. Zbývající jednodušší komponenty lze téměř strojově refaktorovat do Hooks, což představuje efektivní cestu k odstranění závislosti na knihovně Recompose a přizpůsobení se moderním standardům Reactu. Tímto způsobem získáváme nejen výhody aktuálních technologií, ale také zlepšujeme čitelnost a udržovatelnost kódu.

```
import * as React from "react";
import { Card, Row, Input, Text } from "./components";
import ThemeContext from "./ThemeContext";

export default class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Harry",
      surname: "Potter",
      width: window.innerWidth
    };
    this.handleNameChange = this.handleNameChange.bind(this);
    this.handleResize = this.handleResize.bind(this);
    this.handleSurnameChange = this.handleSurnameChange.bind(this);
  }
  componentDidMount() {
    window.addEventListener("resize", this.handleResize);
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentDidUpdate(){
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentWillUnmount() {
    window.removeEventListener("resize", this.handleResize);
  }
  handleNameChange(name) {
    this.setState({ name });
  };

  handleSurnameChange(surname)  {
    this.setState({ surname });
  };

  handleResize() {
    this.setState({ width: window.innerWidth });
  };

  render() {
    let { name, surname, width } = this.state;
    return (
      <ThemeContext.Consumer>
        {theme => (
          <Card theme={theme}>
            <Row label="Name">
              <Input value={name} onChange={this.handleNameChange} />
            </Row>
            <Row label="Surname">
              <Input value={surname} onChange={this.handleSurnameChange} />
            </Row>
            <Row label="Width">
              <Text>{width}</Text>
            </Row>
          </Card>
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

```
import React, { useState, useContext, useEffect } from "react";
import { Card, Row, Input, Text } from "./components";
import ThemeContext from "./ThemeContext";

export default function Greeting(props) {
  let theme = useContext(ThemeContext);

  let [name, setName] = useState("Harry");
  let [surname, setSurname] = useState("Potter");
  useEffect(() => {
    document.title = name + " " + surname;
  });

  let [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    let handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  });

  return (
    <Card theme={theme}>
      <Row label="Name">
        <Input value={name} onChange={setName} />
      </Row>
      <Row label="Surname">
        <Input value={surname} onChange={setSurname} />
      </Row>
      <Row label="Width">
        <Text>{width}</Text>
      </Row>
    </Card>
  );
}
```

Obrázek 3.2: Porovnání HOC versus React Hooks [14]

3.2.4 State management a Redux

Práce s Reduxem v současné podobě je časově náročná a vyžaduje přílišné množství kódu pro jednoduché úpravy. Tento stav také blokuje možnost aktualizace na React 18, protože aktuální verze React-redux (v5) není kompatibilní s touto verzí Reactu. I když by přímé nahrazení Reduxu bylo ideální, je to příliš rozsáhlý úkol, který zahrnuje i další funkcionality jako je routing nebo validace formulářů. Je tedy rozumnější aktualizovat React-redux na verzi 8 a s ním i související balíčky, aby se odblokovala cesta k aktualizaci na React 18. I když přechod na alternativní knihovny pro state management, jako je Zustand nebo využití Apollo Clienta s GraphQL, by mohl nabídnout efektivnější a kompaktnější řešení, tento krok má nižší prioritu, protože používání Reduxu sám o sobě nebrání aktualizaci na React 18 a taková změna by vyžadovala rozsáhlejší přepis kódu. Omezení závislosti na Reduxu bude řešeno granulárně až po aktualizaci na React 18.

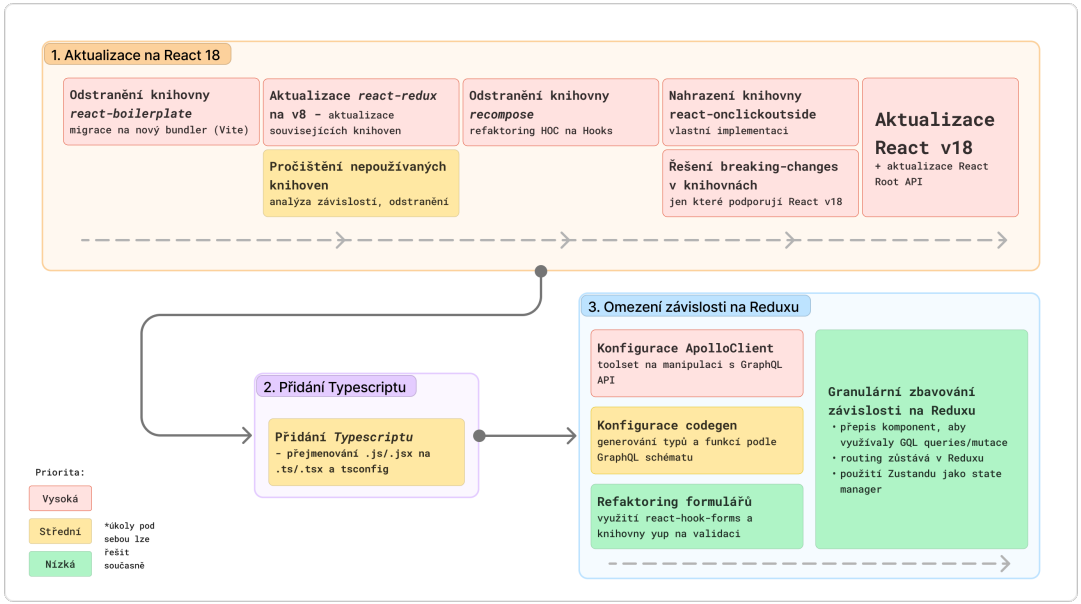
3.2.5 Používání JavaScriptu místo TypeScriptu

Dalším aspektem, který lze označit za slabé místo, je používání pouze JavaScriptu místo TypeScriptu v projektu. TypeScript nabízí silný statický typový systém, což vede k lepšímu odhalování chyb během vývoje, zlepšuje automatické doplňování kódu a umožňuje lepší refaktoring. Přechod na TypeScript by přinesl větší bezpečnost, srozumitelnost kódu a efektivnější spolupráci v týmu vývojářů.

Navíc, při používání GraphQL API z backendu lze naplno využít ponteciál TypeScriptu. S pomocí knihovny @graphql-codegen/typescript lze automaticky generovat typy pro frontendovou část aplikace na základě GraphQL schématu . Tímto způsobem lze získat kompletní a aktuální soubor typů, což výrazně zjednodušuje práci s daty a eliminuje možnost chyb způsobených nesouladem mezi frontendem a backendem. Přechod na TypeScript tedy nejen zvyšuje bezpečnost a srozumitelnost kódu, ale také usnadňuje a zrychluje vývoj frontendové části aplikace [15].

3.2.6 Řešení slabých míst aplikace

V následujícím diagramu jsou znázorněny kroky, které je potřeba provést pro vypořádání se se slabými místy aplikace. Diagram znázorňuje jaké kroky je možné řešit současně, pokud by na modernizaci pracovalo více vývojářů.



Obrázek 3.3: Postup a prioritizace kroků řešení

Kapitola 4

Migrace na nový bundler

4.1 Rozhodovací proces

Migrace na nový bundler byla vyvolána několika klíčovými faktory. Prvním z nich byla zastaralá podpora React-boilerplate, která přestala být udržována od roku 2019 a nedovolovala aktualizaci na novější verze Reactu, zejména na React 18. Dalším motivem byla potřeba adaptace na modernější nástroje a frameworky, a to především s ohledem na potřebu vývoje nových funkcionalit. Interní analýza možných alternativ se soustředila na porovnání šablony CRA (create-react-app) a Vite v kritériích jako jsou rychlost buildu, vývojářská zkušenost, konfigurace a customizace. Na základě této analýzy bylo rozhodnuto přejít na Vite, rychlý bundler optimalizovaný pro práci s moderními frameworky jelikož Vite nabízí HMR (hot module replacement) a tým vývojářů má s Vite zkušenosti z jiných projektů.

4.2 React-boilerplate vs Vite

React-boilerplate je šablona pro vývoj webových aplikací s knihovnou React. Cílem této knihovny je nabízet pevný základ pro webové aplikace, tak aby byly udržovatelné a škálovatelné. Součástí React-boilerplate je několik technologií a konfigurací včetně konfigurací Webpacku, Reduxu, Routingu, testovacích balíčků (Enzyme), CSS Preprocessorů a ESLintu s Prettierem.

React boilerplate, jako mnoho dalších boilerplate projektů, obvykle využívá kombinaci nástrojů a konfigurací, které mohou zahrnovat CommonJS moduly namísto ECMAScript Modules (ESM). CommonJS je modulární systém používaný v Node.js a starších prostředích JavaScriptu, který spoléhá na synchronní načítání modulů a nepodporuje některé pokročilé funkce nabízené ESM [12].

Na druhou stranu Vite je bundler, který se zaměřuje na rychlost a efektivitu při vývoji aplikací. Vite využívá moderní koncepty, kterými jsou například ES6 moduly a je optimalizován pro rychlé spuštění a hot-reloading během vývoje. Vite uplatňuje především nový standard ESM (ECMAScript Modules) v kombinaci s nástroji jako je například Rollup pro efektivní bundlování kódu [16].

Je důležité zdůraznit, že Vite reprezentuje modernější přístup k vývoji a je vhodným nástrojem pro ty, kteří chtějí využívat nejnovější technologie a optimalizace. Vite využívá ESM, což poskytuje několik výhod oproti CommonJS, včetně nativní podpory v moderních prostředích JavaScriptu, asynchronního načítání modulů, efektivnější statické analýzy a budoucí kompatibility se standardem ECMAScript. Tímto způsobem může Vite přinést efektivnější a snadněji udržovatelnou kódovou bázi ve srovnání s tradičním přístupem, jakým je použití React boilerplate.

4.3 Konkrétní kroky a postupy migrace na Vite

4.3.1 Analýza projektu a závislostí na React-boilerplate

Byla provedena systematická analýza stávajícího projektu s důrazem na identifikaci klíčových závislostí a konfigurace specifické pro React-boilerplate.

4.3.2 Založení nového Vite projektu

Při zakládání nového Vite projektu jsem následoval systematický postup s cílem zajistit kompatibilitu s moderními frameworky a optimalizaci pro efektivní vývoj. Pro tento účel jsem vytvořil nový Vite projekt s využitím předlohy pro React. Následující příkaz byl použit k vytvoření projektu:

```
npx create vite@latest flagis-vite -- --template react
```

Šablona byla vybrána z oficiální dokumentace k Vite, ačkoliv dokumentace nespecifikovala konkrétní verzi Reactu. Pro eliminaci pozdější migrace na Vite v4 jsem provedl testování v sandboxu s verzí React 16.10, kterou jsme aktuálně v projektu používali. V rámci tohoto testování jsem zohlednil změny v Client Rendering API, které byly představeny v React 18.

V důsledku těchto testů jsem upravil soubor main.jsx tak, aby byl kompatibilní s API pro React 16. Prokázalo se, že Vite v4 je funkční s verzí React 16, a tím byla zajištěna kompatibilita Vite v4 s React 16.

```
// React 16
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// React 18
import { createRoot } from 'react-dom/client';
const container18 = document.getElementById('app');
const root = createRoot(container18);
root.render(<App tab="home" />);
```

Ukázka kódu 4.1: React Root API in React 16 and React 18 [16]

4.3.3 Migrace kódu

Provedl jsem migraci kódu s přesunutím do nové struktury v *src* složce. Toto opatření nejenže usnadnilo a zlepšilo organizaci kódu, ale také zajistilo optimální kompatibilitu s Vite.

Před migrací byl kód rozdělen podle funkcionality do různých složek, jako jsou *components, redux, utils* a další. Tím, že byl kód migrován do adresáře *src*, se dosáhlo lepší přehlednosti a struktury projektu.

Přesun kódu do nové složky *src* vyžadoval aktualizaci cest importů. Protože JS moduly nepodporují absolutní cesty v projektu, je potřeba řešit tento problém na úrovni kompilátoru přes vlastní nastavení. Naším vlastním nastavením bylo vytvoření aliasu @ pro zlepšení čitelnosti a srozumitelnosti kódu. Ovšem je třeba zmínit, že každý z nástrojů (Prettier, ESLint, TypeScript a další), který poté chce využívat tento soubor, musí podporovat vlastní konfiguraci absolutních cest a musí být zvlášť nakonfigurován. Alias byl nakonfigurován v konfiguračním souboru *vite.config.js*. Funkce find&replace v IDE[1] byla využita k hromadné aktualizaci cest, což usnadnilo zachování struktury projektu.

```
1 resolve: {
2     alias: {
3       '@': path.resolve(__dirname, './src'),
4     },
5   },
```

Ukázka kódu 4.2: vite.config.js

4.4 Problémy a výzvy migrace na nový bundler

4.4.1 Vite a JSX in JS

Během migračního procesu se ukázalo, že existující .js soubory obsahují JSX[2] syntaxi neslučitelnou s požadavky nového bundleru Vite. Pro zachování kompatibility jsem vytvořil skript, který systematicky přejmenovával soubory z .js na .jsx. Tento script také strategicky ignoroval styles.js soubory a podobné.

I přestože toto řešení může dočasně zpomalit build, protože se nad každým .jsx souborem spustí transformace z JSX na JS, jedná se o opatření, které ušetří čas v současné fázi migrace. Přejmenování téměř všech souborů nemusí být optimální, ale vzhledem k plánovaným budoucím změnám v projektu, jako je přidání TypeScriptu a refaktorování z HOC na Hooky, je to praktické a efektivní řešení. Toto řešení umožnilo plynulý přechod na nový standard a eliminaci problémů spojených s JSX syntaxí.

[1] Integrated Development Enviroment
[2] Rozšíření JavaScript syntaxe pro React

4.4.2 Definování environmentálních proměnných

Nový bundler Vite vyžadoval také úpravu v práci s environmentálními proměnnými. Jelikož se v aplikaci nevyužívají environmentální proměnné, je potřeba pouze zřídit lokálního spuštění aplikace proti produkci. process.env.COMPILE_ENV se nahradilo použitím vystavené environmentální proměnné přímo z Vite. Pomocí této boolean proměnné se pak rozlišuje, jestli aplikace běží proti produkční verzi API[3]. Pro spuštění aplikace proti produkci byla přidána NODE_ENV do spouštěcího skriptu [17].

```
1 "dev": "NODE_ENV=develop vite",
```

Ukázka kódu 4.3: spuštění aplikace s Vite a environmentální proměnnou

4.4.3 Nekompatibilita knihovny React-onclickoutside

Posledním problémem byly pády aplikace po přihlášení. Analýzou problému jsem přišel na to, že problémy zřejmě způsobuje knihovna React-onclickoutside. Nebyl jsem schopný přesně odhalit proč tato knihovna přestala být funkční. Naštěstí byla použita jen ve 2 komponentách. Při zakomentování těchto komponent byla aplikace plně funkční s technologií Vite. Experimentoval jsem také s vlastní implementací, která by nahradila tuto knihovnu, ovšem v tomto případě bylo potřeba refaktoring komponent z HOC na funkční komponenty. Výměna knihovny React-onclickoutside za vlastní implementaci se tedy uskutečnila až po refaktoringu z HOC na Hooks v následující fázi modernizace.

```
1  import { useEffect } from 'react';
2
3  const useOnClickOutside = (ref, handler) => {
4    useEffect(() => {
5      const listener = (event) => {
6        if (!ref.current || ref.current.contains(event.target)) {
7          return;
8        }
9        handler(event);
10     };
11
12     document.addEventListener('mousedown', listener);
13     document.addEventListener('touchstart', listener);
14
15     return () => {
16       document.removeEventListener('mousedown', listener);
17       document.removeEventListener('touchstart', listener);
18     };
19   }, [ref, handler]);
20 };
21
22 export default useOnClickOutside;
```

Ukázka kódu 4.4: hook useOnClickOutside

[3]Application Programming Interface

Kapitola 5

Aktualizace Reactu a balíčků

5.1 Důvody pro aktualizaci na React 18

Aktualizace na verzi React 18 je hlavním krokem ve smazání technického dluhu. Používání zastaralých knihoven a frameworků může způsobovat nepředvídatelné problémy a blokovat vývoj nových funkcí. Aby aplikace byla konkurence schopná, je potřeba, aby běžela na moderních technologiích.

Konkrétním důvodem je potřeba implementace nových funkcionalit. Aktualizace knihoven a Reactu na aktuální verze umožní v projektu využívat další externí služby, jako je například Google API na přihlášení přes Google účet.

5.2 Očekávané výhody a vylepšení

- **Jednodušší vývoj:**
  Nové funkce v React 18 usnadňují implementaci nových prvků a zjednodušují každodenní vývojové úkony. React 18 je nejnovější verzí knihovny React a tak většina balíčků je již optimalizována pro React 18.

- **Odstranění problémů s knihovnami:**
  Eliminace potenciálních problémů spojených s nekompatibilitou knihoven.

- **Obecné výhody React 18:**
  Optimalizace výkonu (automatic batching), lepší podpora asynchronního renderingu, nové Hooks a další vylepšení.

5.3 Analýza balíčků

Zde je ukázka analýzy balíčků, která zachycuje nejdůležitější knihovny v kontextu modernizace. Jedná se tak o knihovny, které vyžadují nejvíce práce při řešení problémů nebo jsou kritické pro funkčnost aplikace.

Package	verze v projektu	poslední verze z data	údržba	podporuje React v18 v poslední verzi	poznámka
react-boilerplate	3.7.0	4.0.0 18.4.2019	neaktivní	ne - v16	migrace na Vite
react-onclickoutside	6.13.0	6.13.0 22.3.2023	udržitelná	ano	nefunkční s Vite, nahrazeno vlastní implementací
recompose	0.30.0	0.30.0 30.8.2018	neaktivní	ne - v16	řeší problémy HOC komponent, nahrazeno Hooky
react-redux	5.0.7	9.0.0 4.12.2023	aktivní	ano - vyžaduje react v18	potřeba updatovat alespoň na v8, pro update na react 18
redux	4.0.1	5.0.1 4.12.2023	aktivní	ano	
eslint	5.1.0	8.56.0 15.12.2023	aktivní	ano	vazba na react-boilerplate, dočasně vypnuto

Obrázek 5.1: Ukázka analýzy balíčků

5.4 Postup aktualizace

Před zahájením postupu aktualizace na React 18 jsem pečlivě zvažoval různé strategie aktualizace, včetně postupné aktualizace pomocí React 17. Po podrobném zkoumání možností jsem se rozhodl pro celkovou aktualizaci projektu najednou. Tato strategie byla preferována i samotnou dokumentací React 17, která zdůrazňuje její výhody pro většinu projektů.[18]

Konkrétní kroky aktualizace:

1. Analýza dokumentace upgrade na React 18:
   Studium oficiální dokumentace React 18 pro pochopení nových funkcí a změn.

2. Provedení depchecku a analýza nepoužívaných balíčků:
   Identifikace nepoužívaných balíčků pomocí nástroje depcheck a jejich odstranění.

3. Migrace na nový bundler Vite:
   Odstranění závislosti na knihovně React-boilerplate.

4. Aktualizace React-redux na v8:
   Aktualizace React-redux a s ním souvisejících knihoven k odblokování aktualizace na React 18.

5. Identifikace balíčků, které nepodporují React 18:
   Řešení problémů s knihovnami, které nejsou kompatibilní s React 18. U těchto knihoven se rozhodovalo, zda-li je smažeme či nahradíme modernějšími knihovnami. Příklady takových knihoven jsou Recompose, React-onclickoutside.

6. Řešení breaking changes balíčků a React 18:
   Aktualizace a úpravy kódu v souladu s breaking changes v knihovnách a v React 18.

7. Otestování aplikace:
   Důkladné manuální testování aplikace, zahrnující ověření funkcionality prvků, ve kterých se dělali úpravy při řešení změn.

5.5 Příklady řešení aktualizace knihoven

5.5.1 React-redux

Aktualizace knihovny React-redux se týkala především importu ConnectedRouter. Dříve byl ConnectedRouter importován z knihovny React-router-redux, která byla z projektu odstraněna a nahrazena knihovnou Connected-react-router/immutable. Kromě toho bylo potřeba použít ReactReduxContext a předat ho ConnectedRouteru, který zajišťuje integraci Reduxu s routingem v aplikaci. Další úpravy byly spíše drobného rázu. Jelikož je v plánu od Reduxu v budoucnu odejít, aktualizace React-redux byla provedena v minimální možné podobě, tak, aby zachovala funkčnost aplikace.

5.5.2 Recompose

Odstranění závislosti na knihovně Recompose se ukázalo být nejpracnější úpravou celé modernizace. Knihovna Recompose poskytuje sadu funkcí jako HOC, kterými lze rozšiřovat React komponenty. Tato knihovna již není udržována, jelikož Hooky, jako součást aktuálních verzí Reactu, tyto funkcionality pro HOC komponenty plně pokrývají. Knihovna Recompose je kompatibilní pouze s React 16 a blokuje tak upgrade na React 18.

Bohužel funkce této knihovny byla použita v každé komponentě v našem projektu. Za účelem odstranění knihovny z projektu se odstartoval rozsáhlý proces refaktoringu komponent pomocí Hooků. Naštěstí Hooky byly přidány do Reactu ve verzi 16.8 a v projektu jsme měli již verzi 16.10. Ty jednodušší šlo téměř strojově refaktorovat, ovšem idetifikovali jsme celkem 12 komponent, které vyžadovali detailnější přístup. Těchto 10 komponent bylo komplikovanějších z důvodu připojování Redux storu.

> **A Note from the Author (acdlite, Oct 25 2018):**
> Hi! I created Recompose about three years ago. About a year after that, I joined the React team. Today, we announced a proposal for Hooks. Hooks solves all the problems I attempted to address with Recompose three years ago, and more on top of that. I will be discontinuing active maintenance of this package (excluding perhaps bugfixes or patches for compatibility with future React releases), and recommending that people use Hooks instead. Your existing code with Recompose will still work, just don't expect any new features.[19]

Odstranění knihovny Recompose představovalo značnou práci, avšak byl to krok správným směrem. Přechod na používání hooků a moderních funkcí poskytuje mnohem čistší a efektivnější způsob psaní kódu. Aplikace, která využívá hooky a běží na Reactu 18, lze považovat za moderní a lépe připravenou na budoucí vývoj. Tato změna přinesla nejen technické výhody, ale i větší flexibilitu a udržitelnost celého projektu.

5.5.3 React

Jak bylo již zmíněno v předešlích kapitolách, React 18 představil změny v Client Rendering API. Upravil jsem tak *main.jsx* soubor, aby byl kompatibilní s poslední verzí Reactu.

Kapitola 6

Evaluace základní modernizace

První fáze modernizace naší aplikace Flagis spočívala v migraci zastaralých technologií a knihoven na moderní ekosystém nástrojů, který by poskytl stabilní základ pro další vývoj. Klíčovými kroky této fáze bylo nahrazení původní šablony React-boilerplate moderním nástrojem Vite a aktualizace knihovny React na poslední verzi 18.

Změny bundlovacího nástroje na Rollup (Vite) a aktulizace knihoven měly potenciál výrazně snížit celkovou velikost bundle aplikace a tím zrychlit proces načítání pro uživatele. Ke změření velikosti bundle jsem využil knihovny bundle-analyzer-plugin (Webpack) a vite-bundle-analyzer (Vite).Hodnoty Stat, Parsed a Gzipped size níže v tabulce představují různé metriky velikosti souborů: Stat je velikost souboru na disku, Parsed je velikost po parsování a Gzipped je velikost po Gzip kompresi.

Po provedení změn byly naměřeny následující hodnoty velikosti bundle:

	Stat	Parsed	Gzipped
Původní verze	3,44 MB	10,87 MB	2,66 MB
Modernizovaná verze	12,27 MB	6,42 MB	2,11 MB

Tabulka 6.1: Srovnání velikostí souborů

6.1 Důležitost bundle size pro rychlost načítání webu

Celková velikost bundle hraje klíčovou roli v rychlosti načítání webové aplikace. Menší bundle size znamená rychlejší načítání, což má přímý dopad na uživatelskou zkušenost. Při každém načítání stránky musí prohlížeč stáhnout všechny potřebné soubory, což zahrnuje HTML, CSS, JavaScript a další zdroje. Čím menší jsou tyto soubory, tím rychleji může prohlížeč zobrazit obsah uživateli.

Optimalizace bundle size je tedy klíčovým faktorem pro zajištění rychlého načítání aplikace, což přispívá k lepší uživatelské zkušenosti a zvyšuje pravděpodobnost, že uživatel zůstane na stránce.

6.2 Vliv nové funkcionality na bundle size

Je důležité poznamenat, že do modernizované aplikace byla přidána nová funkcionalita prémiových módů, která s sebou přinesla zvýšení množství obrázků a dalších dat. Tento faktor měl za následek nárůst hodnoty "Stat"velikosti souboru na disku. Nicméně, i přes toto zvýšení je výsledek migrace pozitivní, neboť celková bundle size byla snížena.

6.3 Testování výkonu aplikace - Lighthouse

Pro objektivní měření výkonu aplikace bylo využito nástroje Lighthouse. V rámci testování byly hodnoceny klíčové metriky výkonu, jako je čas prvního zobrazení obsahu (First Contentful Paint) a čas zobrazení největšího obsahového prvku (Largest Contentful Paint) [20].

Metrika	**Původní verze**	**Modernizovaná verze**
Performance score	63	78
First contentful paint	3.4 s	1.9 s
Largest contentful paint	3.5 s	2.3 s

Tabulka 6.2: Výkonnostní metriky srovnání

Výsledky testování ukázaly významné zlepšení výkonu aplikace po provedené modernizaci. Performance skóre se zvýšilo z hodnoty 63 na 78, zatímco čas prvního zobrazení obsahu klesl z 3,4 na 1,9 sekund a největší obsahový zobrazení z 3,5 na 2,3 sekund. Tato zlepšení přispívají k celkově lepší uživatelské zkušenosti a rychlejší odezvě aplikace.

6.4 Závěr první části modernizace

Modernizace úspěšně splnila cíl vytvoření stabilního základu aplikace, který poskytuje prostor pro vývoj nových funkcionalit. Zároveň se podařilo efektivně zmenšit velikost bundle a optimalizovat proces načítání aplikace, což bude mít dlouhodobě pozitivní dopad na uživatelskou spokojenost a výkonnost celého systému.

Kapitola 7

Přidání TypeScriptu

7.1 Motivace a důvody pro přidání TypeScriptu

Rozhodnutí o zavedení TypeScriptu do projektu bylo motivováno několika klíčovými faktory, které vycházejí z výhod TypeScriptu oproti JavaScriptu. Mezi hlavní důvody v kontextu aplikace Flagis patří:

1. Statické typování:
   Možnost odhalit chyby v době kompilace a snížit tak počet chyb v běhu aplikace.

2. Udržitelnost kódu:
   TypeScript přispívá k lepší čitelnosti kódu a snazšímu porozumění jeho struktuře díky definici typů.

3. Podpora IDE:
   TypeScript má vysokou míru podpory pro IDE a zvyšuje tak produktivitu vývojářů.

4. Budoucí kompatibilita:
   Použití TypeScriptu umožňuje kombinaci s ECMAScript standardem a zajišťuje tak připravenost na budoucí vývoj jazyka.

5. Integrace s GraphQL:
   TypeScript lze využít v kombinaci s dalšími nástroji. Lze tak například využít knihovnu Codegen pro generování typů z GraphQL schématu, což usnadňuje práci s GraphQL v projektu.

6. Monorepo struktura:
   Přidání TypeScriptu umožňuje sdílení typů mezi webovou a mobilní částí apliakce, což je výhodné pro monorepo strukturu projektu.

7.2 Zkušenosti týmu s TypeScriptem

Implementace TypeScriptu nebyla pro tým nová, jelikož všichni členové vývojového týmu, kteří se podílejí na vývoji aplikace, měli předchozí zkušenosti s TypeScriptem z jiných projektů. Tato zkušenost umožňila plynulé zavedení TypeScriptu do projektu a minimalizaci potenciálních problémů.

7.3 Proces zavedení TypeScriptu

Přidání TypeScriptu do projektu bylo provedeno s minimálními komplikacemi pomocí následujícího postupu [21]:

1. Přidání konfiguračního souboru tsconfig.json:
   Byl vytvořen soubor tsconfig.json s vhodnou konfigurací pro TypeScript, která zahrnuje základní nastavení pro projekt.

2. Aktualizace konfiguračního souboru vite.config.ts:
   Konfigurační soubor byl upraven tak, aby byl kompatibilní s použitím TypeScriptu a aby zahrnoval správné cesty pro aliasy a pluginy[16].

3. Přejmenování souborů na příponu .ts/.tsx:
   Existující soubory s příponami .js nebo .jsx byly přejmenovány na přípony .ts a .tsx

4. Postupné doplňování typů:
   TypeScript umožňuje postupné doplňování typů a proto byla migrace na plné používání TypeScriptu naplánována na další fázi vývoje. Odstranění knihovny propTypes, která umožňuje specifikovat typy u properties, a otypování celé aplikace se bude řešit postupně s přechodem na GraphQL při refaktorování komponent.

Kapitola 8

Omezení závislosti na knihovně Redux

8.1 Co je Redux a proč je problematický

Redux je knihovna pro správu stavu aplikace v JavaScriptových aplikacích. Hlavní role Reduxu spočívá v centralizovaném ukládání stavu aplikace v jednom tzv. store, který je dosažitelný z libovolné části aplikace, ovšem celá aplikace musí být obalena tzv. providerem. Redux implementuje jednosměrný tok dat, což znamená, že stav aplikace se aktualizuje pouze pomocí tzv. actions, které popisují události, jež se staly. Tyto akce jsou zpracovány tzv. reducery, které změní stav aplikace podle specifikace akce.

Přestože Redux nabízí systematický způsob správy stavu aplikace, má několik problémů, zejména v kontextu aplikací jako je Flagis.

1. Nadbytečná složitost:
   Redux vyžaduje hodně boilerplate kódu pro definici akcí, reducerů a spojení s UI[1]. Tento boilerplate kód zvyšuje složitost aplikace a zpomaluje vývoj. Může také vést k tomu, že vývojáři, kteří nedodržují správné nárhové vzory využití Reduxu, vytvoří špatně udržovatelný kód, kde vzniká například tzv. prop drilling proti kterému knihovna jako Redux bojuje.

2. Nedostatek integrace s asynchronními operacemi:
   Redux sám o sobě efektivně neřeší asynchronní operace. I přesto, že existují doplňky do Reduxu jako je například Redux Thunk nebo Redux Saga, přidávají další složitost a snižují přehlednost kódu.

[1] User Interface

8.2 Alternativy k Reduxu: Apollo Client + Zustand

8.2.1 Apollo Client

Apollo Client je vynikající alternativou k Reduxu, zejména pokud aplikace pracuje s GraphQL. Apollo Client nabízí kompletní řešení pro správu stavu a práci s daty, které jsou načítány a ukládány skrze GraphQL queries a mutace[22].

- **Integrovaná správa cache:** Apollo Client nabízí možnost ukládání dat v mezipaměti (cache). Data získaná ze serveru tak mohou být automaticky ukládána do cache a znovu být použita bez dotazování serveru.

- **Reaktivní aktualizace dat:** Apollo Client umožňuje reaktivně aktualizovat data pomocí GraphQL subscriptions. Lze tak zobrazovat změnu dat bez nutnosti ruční aktualizace stavu.

- **Jednoduchá integrace s Reactem:** Apollo Client poskytuje snadnou integraci s Reactem díky knihovně @apollo/react-hooks, která nabízí používat hooky pro provádění GraphQL queries a mutací přímo v React komponentách.

8.2.2 Zustand

Zustand je "lightweight" alternativou k Reduxu, který je založen na minimalismu a jednoduchosti. Od Reduxu se tak liší v několika ohledech[23]:

- **Jednoduchost použití:** Zustand je navržen tak, aby jeho použití bylo co nejjednodušší. Není to robustní nástroj jako Redux a nepotřebuje tak složitou konfiguraci či tvorbu akcí a reducers. V Zustandu stačí pouze vytvořit stav a funkce na jeho aktualizaci.

- **Centralizované ukládání stavu:** Podobně jako Redux, i Zustand poskytuje centrální úložiště stavu, které je dostupné kdekoliv v aplikaci. Ovšem není potřeba obalovat aplikaci *Providerem*, jelikož Zustand store lze konzumovat jako hook.

- **Reaktivní aktualizace stavu:** Zustand automaticky zajišťuje reaktivní aktualizaci stavu. To znamená, že jakmile se stav změní, všechny komponenty využívající stav se znovu překreslí.

V kontextu aplikace Flagis nabízí Zustand pár výhod:

- **Snadná integrace a minimalismus:** Zustand je vhodný pro jednoduché aplikace, které nepotřebují robustní řešení.

- **Reaktivní aktualizace stavu:** Pro aplikaci Flagis, kde je důležité okmažitě reagovat na změny stavu a aktualizovat uživatelské rozhraní, se využije built-in reaktivity Zustand stavu.

8.2.3 Spojení Apollo Client a Zustand

V aplikaci Flagis jsme se rozhodli využívat obě technologie zároveň. Kompetence jednotlivých knihoven lze rozdělit následovně:

- **Apollo Client:** Bude mít na starosti náčítání a ukládání dat skrz GraphQL queries a mutace. To zahrnuje i manipulaci s mezipamětí (cache). Zároveň se využívá GraphQL subscriptions, pokud reaktivita aplikace požaduje okamžité zobrazení změny dat.
- **Zustand:** Může být použit pro správu lokálního stavu aplikace, který není závislý na externím zdroji dat (serveru). Příkladem takových stavů jsou uživatelská nastavení, stav formulářů nebo stav UI komponent (filtrů).

8.3 Routing a formuláře v Reduxu

Aplikace Flagis je závislá na knihovně Redux i v kontextu tzv. "routingu". Jde o konfiguraci cest v aplikaci, která často zahrnuje v Reduxu ukládání do globálního stavu aplikace, což je neefektivní a komplikované. Migrace na alternativní knihovnu React-router zůstává jako jeden z posledních kroků k odstranění knihovny Redux z aplikace.

Aplikace využívá i knihovnu Redux-form pro správu stavu formulářů. Za účelem omezení závislosti na knihovně Redux se rozhodlo pro refaktoring a migraci formulářů na knihovnu React-hook-form.

- **Redux-form:** Knihovna pro React, která spravuje stav formulářů pomocí Reduxu. Umožňuje centrálně ukládat stav formulářů do Reduxu a sama řeší synchronizaci stavu mezi komponentami.
- **React-hook-form:** Knihovna pro React, která postkytuje hooky na správu formulářů. Je založena na minimalismu a jednoduchosti použití. Je ideální pro menší a střední formuláře a poskytuje snadnou integraci s hooky a funkcemi z Reactu.

Formuláře byly přepracovány, aby byly v souladu s principy knihovny React-hook-form. Na validaci formulářů byla použita knihovna Yup. Tato knihovna poskytuje možnosti komplexní validace formulářů včetně ověřování typů dat, rozsahu hodnot a dalších podmínek.Knihovny React-hook-form a Yup lze velice jednoduše integrovat, jelikož knihovna React-hook-form sama nabízí resolver pro knihovnu Yup. Lze také využít TypeScript a přímo otypovat výstupní hodnotu hooku useForm.

```
import { useForm } from 'react-hook-form';
import { yupResolver } from '@hookform/resolvers/yup';
import * as yup from 'yup';

const schema = yup
  .object()
  .shape({
    name: yup.string().required(),
    age: yup.number().required(),
  })
  .required();

type FormValues = {
  name: string;
  age: number;
}

const App = () => {
  const { register, handleSubmit } = useForm<FormValues>({
    resolver: yupResolver(schema),
  });

  return (
    <form onSubmit={handleSubmit((d) => console.log(d))}>
      <input {...register('name')} />
      <input type="number" {...register('age')} />
      <input type="submit" />
    </form>
  );
};
```

Ukázka kódu 8.1: ReactHookForm resolvers for Yup [24]

8.4 Migrace na GraphQL

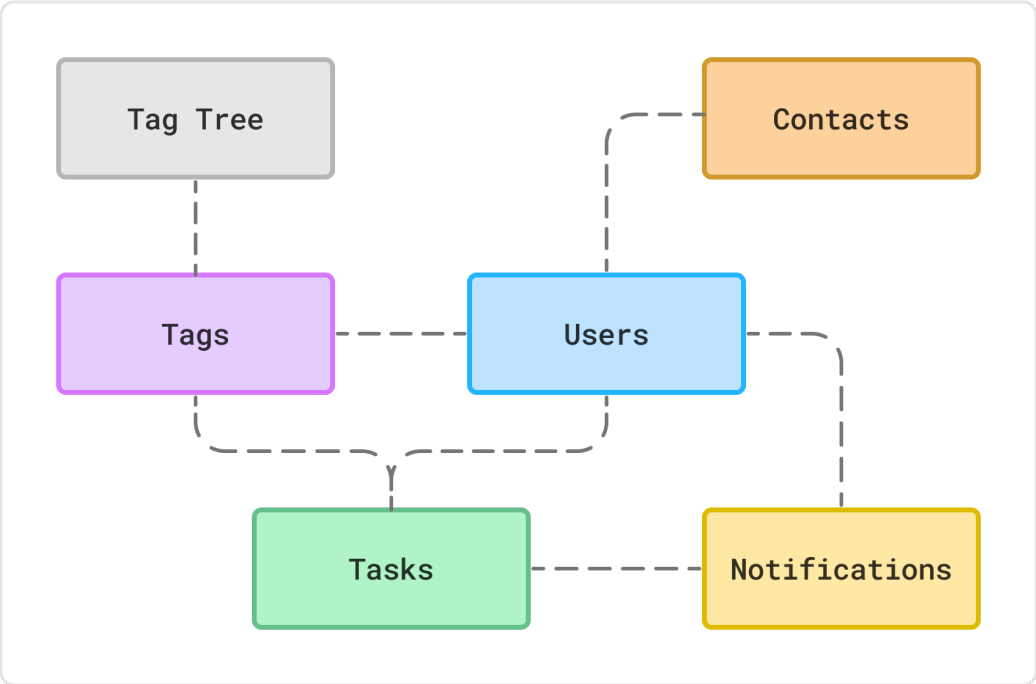
Migrace na GraphQL z jiné komunikační technologie je složitý a náročný proces. V této podkapitole bych tedy chtěl přiblížit jak probíhaly jednotlivé kroky. Dále ukážu, co bylo potřeba upravit ve webové části aplikace, aby mohla aplikace pracovat s GraphQL API. Představím také konkrétní ukázku modernizace jedné z jednodušších entit a komponent, které s toutu entitou pracují.

Je podstatné zmínit, že v rámci této práce jsem se zabýval pouze migrací webové části aplikace. Důležitou součástí migrace byla úzká spolupráce s kolegou na backendové straně, který připravoval GraphQL API a poskytoval nezbytné dotazy a mutace.

8.4.1 Proces migrace

Naší snahou bylo, aby proces migrace byl granulární a mohli jsme tak postupně nasazovat verze aplikace, kde už nějaká část fungovala přes GraphQL. Proto se komunikace přes HTTP requesty (Redux) nijak neomezovala a bylo potřeba vytvořit "mezistav" při procesu migrace, kdy některé části aplikace fungovaly přes Redux a některé již fungovaly přes GraphQL. To nám zároveň usnadnilo současně pracovat s kolegou na backendu, jelikož jsem například mohl konzumovat queries, které přidal na API, zatímco on připravoval mutace pro tu stejnou entitu.

1. **Analýza a plánování:** Identifikace částí aplikace a určení postupu migrace na GraphQL. Ve spolupráci s kolegou jsme velkým zjednodušením ER Diagramu z databáze jsme rozdělili aplikaci do následujících částí.
   Rozhodli jsme se nejdříve modernizovat izolovanější části aplikace, jako jsou například Contacs, Tags nebo Notifications. Tyto části mají vlastní stránky, kde je zobrazen například seznam kontaktů a je zde možné třeba přidat nový kontakt.



Obrázek 8.1: Identifikace hlavních částí aplikace

2. **Příprava backendu:** Kolega prováděl postupně se mnou modernizaci na backendové straně a přidával mi na API GraphQL queries a mutace pro práci s daty. Zároveň používáme na API Apollo Server. Lze si tedy spustit pro lokální vývoj Apollo Server, který má připravené webové rozhraní pro práci s queries a mutacemi.

3. **Příprava webové části:** Pro práci s GraphQL queries a mutacemi bylo potřeba nakonfigurovat Apollo Client. V rámci konfigurace Apollo Clienta je potřeba nastavit jak se bude pracovat s mezipamětí (cache) a také nastavit

autorizaci, aby se s requesty posílaly autorizační tokeny. Dále se ve webové části využívá knihovna Codegen, která z dostupného GraphQL schématu vygeneruje typy a funkce pro manipulaci s GraphQL queries a mutacemi.

4. **Migrace komponent:** Postupná migrace jednotlivých částí aplikace na použití GraphQL.

5. **Testování:** Manuální testování nově migrovaných komponent a ověření správné funkcionality.

6. **Postupné nasazování:** Postupné nasazování migrovanýách částí aplikace, aby nedošlo k narušení běhu celé aplikace.

8.4.2 Omezení při migraci

Při migraci se znovu projevilo omezení vycházející z Reduxu. Jelikož je v Reduxu implementován routing, rozhodli jsme se ho odebrat až bude celá aplikace fungovat datově na GraphQL. To znamená, že i při migraci komponent je potřeba zanechávat kód vztahující se k Reduxu v aplikaci a ovládat routing skrze Redux.

Omezení je zřejmé v konkrétní ukázce migrace zobrazení listu Tagů na GraphQL v další kapitole.

8.4.3 Ukázka migrace TagListu

V této kapitole bych chtěl popsat ukázku migrace TagListu, jedné z komponent aplikace. Jde o komponentu, kde lze ukázat granulární způsob migrace. Z této migrace je na první pohled vidět výhoda oproti Reduxu a zároveň zobrazuje limitaci v routingu v Reduxu.

Při migraci TagListu bylo potřeba upravit přímo komponentu TagListu a také menší komponentu, která představuje jednu položku ze seznamu.

```
export default function TagListContainer() {
const { t } = useComponentsTranslation();

const dispatch = useDispatch();

//ziskani listu Tagu z GraphQL
const { data, loading } = useGetTagListQuery({
  pollInterval: gqlPollInterval,
});

//navigace na detail Tagu (Redux)
const onHandleTagClick = useCallback(
  (tagId: string) => {
    dispatch(selectTag(tagId));
    dispatch(setDetail('tag'));
  },
  [dispatch],
);

if (loading) {
  return <Loader />;
}

if (!data || data.tags.length === 0) {
  return <EmptyList>{t('tagList.emptyList.text')}</EmptyList>;
}

return (
  <Scrollbar style={{ shadowHeight: 20, heightOffset: 198 }} type
     ="tag">
    <ul>
      {data.tags.map(
        (tag) =>
          tag && (
            <TagListItem
              key={tag.id}
              tag={tag}
              onClick={onHandleTagClick}
            />
          ),
      )}
    </ul>
  </Scrollbar>
);
}
```

Ukázka kódu 8.2: tag-list.tsx

```
type Props = {
  tag: Tag; //typ vygenerovan podle GQL schematu
  onClick: (tagId: string) => void; //navigace na detail Tagu Redux
};

export const TagListItem = ({ tag, onClick }: Props) => {
  const onHandleClick = useCallback(() => {
    onClick(tag.id);
  }, [onClick, tag.id]);

//... zbytek komponenty jednoho list-itemu
```

Ukázka kódu 8.3: tag-list-item.tsx

Migrace na GraphQL zjednodušila obecnou komponentu Detail, která původně získávala z Reduxu celý objekt aktuálního tagu, specifikovala všechny funkce manipulující s Tagem (změna description, smazání atd.) a následně vše přes *props* předala komponentě tag-detail, která je zobrazením detailu Tagu. Nyní si stačí získat pouze tagId z Reduxu a to předat komponentě na zobrazení detailu Tagu. Tato komponenta si následně získá data přes GraphQL query. To nám dále umožňuje pracovat s loading nebo error stavy přímo v komponentě detailu Tagu. Dále má v sobě také specifikované mutace na aktualizaci či smazání Tagu, takže není potřeba je získávat přes *props*.

Po refaktoringu routování na React-router si bude moct komponenta tag-detail získat tagId například z url a potřeba pro obecnou Detail komponentu odpadne kompletně.

```
//Obecna komponenta Detail, rozhoduje o zobrazeni detailu Tag/Task/
   Contact
const tagId = useSelector(getCurrentTagId);
```

Ukázka kódu 8.4: detail/index.tsx

```
  //GQL queries a mutace az v detailu
  const { data } = useGetTagQuery({
    variables: {
      tagId: tagId,
    },
  });
  const [updateTag] = useUpdateTagMutation({
    refetchQueries: [{ query: GetTagListDocument }],
  });

  const [deleteTag] = useDeleteTagMutation({
    refetchQueries: [{ query: GetTagListDocument }],
  });
```

Ukázka kódu 8.5: detail/tag-detail.tsx

8.4.4 Shrnutí migrace na GraphQL

Migrace na GraphQL se ukázala být správným krokem ke kompletní modernizaci a revitalizaci aplikace. Správným použitím GraphQL se eliminuje potřeba velké části boilerplate kódu způsobeným používání Reduxu. Postupná migrace má omezení, jelikož je potřeba využívat Redux i GraphQL zároveň. Po migraci získávání dat na GraphQL bude žádoucí migrace routingu na React-router a odstranit Redux úplně.

Kapitola 9

Závěr

9.1 Zhodnocení práce

Práce na modernizaci a rozvoji aplikace Flagis přinesla komplexní transformaci a zdokonalení, které posilují celkovou kvalitu a konkurenceschopnost aplikace.

V první části práce jsem se zaměřil na analýzu aplikace a hodnocení současného stavu. Identifikoval jsem klíčové oblasti potřebující aktualizaci a zdokonalení, včetně problémů spojených s udržitelností kódu, správou stavu a komunikačními technologiemi.

Za účelem odstranění závislosti aplikace na React-boilerplate byla provedena migrace na nový bundler Vite. Cílem této migrace bylo také optimalizovat vývojové prostředí a zmenšit počet závislostí v aplikaci. V rámci tohoto kroku byla taktéž aktualizována struktura projektu.

Následně jsem odstranil blok v podobě starých a neudržovaných verzí balíčků, což umožnilo aktualizaci na React 18. V rámci odstranění závislosti na knihovně Recompose jsem zrefaktoroval kód z HOC (Higher-Order-Components) na Hooky.

Během modernizace se podařilo odstranit 59 z původních 97 balíčků, kdy nejvíce se na odstranění balíčků podílela migrace na Vite. V projektu po modernizaci zůstává dalších 19 balíčků, které vyžadují úpravy kvůli breaking changes v nových verzích. Z těchto 19 balíčků se 9 týká knihovny Redux, kde je v plánu kompletní odstranění. Ostatní balíčky neblokují aktualizaci na React 18 ani celkovou modernizaci aplikace. Aktualizace balíčků celkově zlepšila udržitelnost a přehlednost kódu. Dále také umožnila vývoj nových funkcionalit, které byly blokovány zastaralými technologiemi.

Přidání TypeScriptu do projektu přineslo výrazné vylepšení, zejména díky statickému typování, které pomáhá odhalovat chyby v době kompilace a zvyšuje přehlednost kódu.

Dalším krokem bylo omezení závislosti na knihovně Redux a integrace alternativních nástrojů jako jsou Apollo Client a Zustand. Postupná migrace komunikační technologie na GraphQL s sebou přinesla snížení komplexity kódu, což vede ke zlepšení udržitelnosti aplikace a zrychlení vývoje. I přesto, že se Redux nepodařilo kompletně odstranit, odstartoval se proces odstranění. Lze tedy říci, že závislost na knihovně Redux byla omezena.

Celkově lze konstatovat, že práce na modernizaci aplikace Flagis přinesla významné vylepšení v oblasti udržitelnosti, vývojové zkušenosti a dalším rozvoji aplikace. Celá modernizace byla klíčovým prvkem pro posílení pozice produktu na trhu.

9.2 Budoucí kroky

I přes významný pokrok při modernizaci aplikace Flagis zůstává ještě několik klíčových úkolů, které je třeba vyřešit v budoucnosti.

Prvním z těchto úkolů je dokončení procesu odstranění závislosti na knihovně Redux. I když jsme již provedli kroky k omezení používání Reduxu, stále zůstávají části aplikace, které na něj spoléhají. Posledním krokem v tomto procesu bude migrace routingu z Reduxu na knihovnu React-router. Tento krok bude klíčovým krokem ke kompletnímu odstranění Reduxu.

Kromě toho je důležité neustále sledovat a aktualizovat balíčky a frameworky v rámci aplikace. Pravidelné aktualizace pomáhají zajištění bezpečnosti, výkonu a kompatibility aplikace s nejnovějšími technologiemi a standardy. Je však klíčové zdůraznit postupnou aktualizaci balíčků, aby se v budoucnu nemuselo přistoupit na takto rozsáhlý proces modernizace a aplikace se nedostala do takto zastaralého stavu.

S těmito budoucími kroky věřím, že aplikace Flagis bude nadále přinášet uživatelům vysokou hodnotu a spokojenost.

Bibliografie

1. *Usage Statistics of JavaScript as Client-Side Programming Language on Websites, May 2024* [n.d.]. [B.r.]. Dostupné také z: https://w3techs.com/technologies/details/cp-javascript.
2. *Does Your Web App Need a Front-End Framework? - Stack Overflow* [February 3, 2020]. [B.r.]. Dostupné také z: https://stackoverflow.blog/2020/02/03/is-it-time-for-a-front-end-framework.
3. *GitHub Collection of Front-End Frameworks* [Accessed May 4, 2024]. [B.r.]. Dostupné také z: https://github.com/collections/front-end-javascript-frameworks.
4. *Node.js — an Introduction to the Npm Package Manager* [n.d.]. [B.r.]. Dostupné také z: https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager.
5. ELROM, E. State Management. In: *React and Libraries*. Berkeley, CA: Apress, 2021. ISBN 978-1-4842-6695-3. Dostupné z DOI: 10.1007/978-1-4842-6696-0_5.
6. YUNITA, Andriani. *Challenges in Front-end JavaScript Development for Web Applications — Developers' Perspective* [December 11, 2023]. 2023. Dostupné také z: https://aaltodoc.aalto.fi/items/4d00dd50-9549-41d3-920c-f7912779dab3.
7. SCOTT, Emmit A. *SPA Design and Architecture*. Simon a Schuster, 2015. Dostupné také z: http://books.google.ie/books?id=fTkzEAAAQBAJ&printsec=frontcover&dq=SPA+Design+and+Architecture:+Understanding+single-page+web+applications&hl=&cd=1&source=gbs_api.
8. LUCHANINOV, Yurii. *Web Application Architecture in 2024: Moving in the Right Direction* [MobiDev]. 2024. Dostupné také z: https://mobidev.biz/blog/web-application-architecture-types.
9. *React* [n.d.]. [B.r.]. Dostupné také z: https://react.dev/.
10. *React Redux* [n.d.]. [B.r.]. Dostupné také z: https://react-redux.js.org/.
11. UNKNOWN. *JavaScript Bundlers: Is It Worth Switching From Webpack to Vite?* [October 2, 2023]. 2023. Dostupné také z: https://career.comarch.com/blog/javascript-bundlers-is-it-worth-switching-from-webpack-to-vite/.
12. STOIBER, Max. *React-boilerplate/react-boilerplate: :fire: A highly scalable, offline-first foundation with the best developer experience and a focus on performance and best practices.* 2018. Dostupné také z: https://github.com/react-boilerplate/react-boilerplate.

13. ABRAMOV, Dan; NABORS, Rachel. *Hooks FAQ*. 2019. Dostupné také z: https://legacy.reactjs.org/docs/hooks-faq.html#do-hooks-replace-render-props-and-higher-order-components.

14. PADMANABHAN, Arvind. *React hooks*. Devopedia Foundation, 2022. Dostupné také z: https://devopedia.org/react-hooks.

15. SIMHA, Dotan. *Dotansimha/graphql-code-generator: A tool for Generating Code based on a graphql schema and graphql operations (Query/mutation/subscription), with flexible support for custom plugins.* 2023. Dostupné také z: https://github.com/dotansimha/graphql-code-generator.

16. YOU, Evan. 2020. Dostupné také z: https://vitejs.dev/guide/.

17. YOU, Evan. 2020. Dostupné také z: https://vitejs.dev/guide/env-and-mode.

18. ABRAMOV, Dan; NABORS, Rachel. *React V17.0 – REACT BLOG*. 2020. Dostupné také z: https://legacy.reactjs.org/blog/2020/10/20/react-v17.html.

19. CLARK, Andrew. *Acdlite/recompose: A react utility belt for function components and higher-order components.* 2018. Dostupné také z: https://github.com/acdlite/recompose.

20. DEVELOPERS, Google for. *About PageSpeed Insights* [n.d.]. [B.r.]. Dostupné také z: https://developers.google.com/speed/docs/insights/v5/about.

21. *Documentation - Migrating from JavaScript* [n.d.]. [B.r.]. Dostupné také z: https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html.

22. *Introduction to Apollo Client* [n.d.]. [B.r.]. Dostupné také z: https://www.apollographql.com/docs/react/.

23. PMNDRS. *GitHub - Pmndrs/Zustand: Bear Necessities for State Management in React* [n.d.]. [B.r.]. Dostupné také z: https://github.com/pmndrs/zustand.

24. REACT-HOOK-FORM. *GitHub - React-Hook-Form/Resolvers: Validation Resolvers: Yup, Zod, AJV, Joi, Superstruct, Vest, Class-Validator, Io-Ts, Typanion, Ajv, TypeBox, Valibot and Nope* [n.d.]. [B.r.]. Dostupné také z: https://github.com/react-hook-form/resolvers?tab=readme-ov-file#yup.

Přílohy

A Script na přejmenování JS souborů na JSX

```
import fs from 'fs';
import path from 'path';

const rootDirectory = 'src';

function shouldRenameFile(fileName) {
  const excludedFiles = ['styles.js', 'global-styles.js', 'i18n.js'];
  return !excludedFiles.includes(fileName);
}

function processFilesInDirectory(directoryPath) {
  fs.readdirSync(directoryPath).forEach(file => {
    const filePath = path.join(directoryPath, file);
    console.log("FILE: ", file);

    if (fs.statSync(filePath).isDirectory()) {
      processFilesInDirectory(filePath); // Recursive call for subdirectories
    } else if (file.endsWith('.js') && shouldRenameFile(file)) {
      const newFilePath = path.join(directoryPath, file.replace('.js', '.jsx'));
      fs.renameSync(filePath, newFilePath);
      console.log(`Renamed: ${filePath} -> ${newFilePath}`);
    }
  });
}

processFilesInDirectory(rootDirectory);
```