Univerzita Pardubice
Fakulta elektrotechniky a informatiky

# Uživatelská nadstavba pro backoffice systém

Diplomová práce

2023 Bc. Lukáš Semorád

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2021/2022

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

| | |
|---|---|
| Jméno a příjmení: | **Bc. Lukáš Semorád** |
| Osobní číslo: | **I20215** |
| Studijní program: | **N0613A140007 Informační technologie** |
| Téma práce: | **Uživatelská nadstavba pro backoffice systém** |
| Zadávající katedra: | **Katedra softwarových technologií** |

## Zásady pro vypracování

Cílem diplomové práce je návrh a tvorba potřebných modulů pro efektivní správu, přehled a řízení celé aplikace. Klíčovým požadavkem je tvorba přehledných modulů v projektu, které budou plnit potřeby uživatele. Nadstavba bude komunikovat s plně RESTovým API.
Součástí bude implementace autentizace pomocí JWT tokenu, a bude kladen důraz na striktní validaci formulářů pro omezení chybovosti. Systém bude rozdělen do jednotlivých modulů, které budou samostatné, nebo dle potřeby komunikovat mezi sebou.
V teoretické části budou také popsány technologie, které se využívají pro efektivní vývoj webových aplikací a dále technologie využité pro samotnou tvorbu systému.
Praktická část bude obsahovat analýzu řešené oblasti, pomocí jazyka UML budou modelovány podstatné části systému a bude také uveden popis implementace jednotlivých technologií při realizaci systému, včetně prvotních návrhů. Pro realizaci budou využity převážně jazyky TypeScript a HTML5 s frameworkem Angular.

Rozsah pracovní zprávy: **40-50 normostran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

BASL, J., BLAŽÍČEK R. Podnikové informační systémy: podnik v informační společnosti. 3., aktualiz. a dopl. vyd. Praha: Grada, 2012. Management v informační společnosti. ISBN 978-80-247-4307-3.

EELES, P., CRIPPS P. Architektura softwaru. Brno: Computer Press, 2011. ISBN 978-80-251-3036-0.

DEELEMAN, P. Learning Angular 2. 1. Birmingham: Packt Publishing Limited, 2016. ISBN 1785882074.

Vedoucí diplomové práce: **doc. Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **8. listopadu 2021**
Termín odevzdání diplomové práce: **20. května 2022**

L.S.

**Ing. Zdeněk Němec, Ph.D.** v.r.
děkan

**prof. Ing. Antonín Kavička, Ph.D.** v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2021

Prohlašuji:

Práci s názvem „Uživatelská nadstavba pro backoffice systém“ jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 1. 8. 2023

Lukáš Semorád

# PODĚKOVÁNÍ

Tímto bych rád poděkoval svému vedoucímu práce panu doc. Ing. Michaeli Bažantovi, Ph.D. za ochotu a odbornou pomoc při vedení této práce. Veliké poděkování patří mým přátelům a rodině, za trpělivost a podporu během mého studia. V neposlední řadě bych také rád poděkoval společnosti TechCrowd s.r.o. za možnost téma vypracovat a zveřejnit praktickou část této práce.

# ANOTACE

Tato diplomová práce je zaměřena na analýzu a vývoj klientské části backoffice systému, který je možné dále rozšířit a využít pro správu ve firmách s různým zaměřením. Cílem teoretické části je analýza nástrojů a technologií, které byly využity při realizaci systému. Autor v teoretické části dále rozebírá aktuální možnosti řešení backoffice správy. V praktické části se autor věnuje klíčovým požadavkům a návrhem systému, který tyto požadavky bude splňovat.

# KLÍČOVÁ SLOVA

Obslužný systém, informační systém, intranet, klientská část, vývoj, TypeScript, Angular

# TITLE

User interface for backoffice system.

# ANNOTATION

This diploma thesis is focused on the analysis and development of the client-side of a backoffice system, which can be further extended and utilized for management in companies with various focuses. The aim of the theoretical part is the analysis of tools and technologies used in the system's implementation. The author also discusses current options for backoffice management solutions in the theoretical part. In the practical part, the author addresses key requirements and designs a system that will meet these requirements.

# KEYWORDS

Service system, information system, intranet, Frontend, development, TypeScript, Angular.

# OBSAH

# SEZNAM ILUSTRACÍ A TABULEK

# SEZNAM ZKRATEK A ZNAČEK

| | |
|---|---|
| API | Application Programming Interface |
| BO | BackOffice |
| CD | Continuous Deployment |
| CI | Continuous Integration |
| EP | End Point |
| HTML | Hypertext Markup Language |
| JS | JavaScript |
| JSON | JavaScript Object Notation |
| JWT | JSON Web Token |
| NPM | Node.js package manager |
| REST | Representational State Transfer |
| PR | Pull request |
| TS | TypeScript |

# ÚVOD

Tato diplomová práce se zabývá návrhem a vývojem frontendové části rozšiřitelné webové aplikace. V dnešní době je poptávka po webových aplikacích, jako je ta, kterou se tato práce zabývá, velmi vysoká. S narůstající digitalizací a automatizací procesů se stále více společností rozhoduje přecházet na moderní a efektivnější softwarová řešení. Tyto systémy poté usnadňují řízení firmy, umožnují správu uživatelských účtů, poskytují přístup k datům a zvyšují efektivitu firmy.

Tato práce se zaměřuje na vývoj aplikace, která bude snadno upravitelná a rozšiřitelná. To bude zajištěno modulárním vývojem. Aplikace bude rozdělena do jednotlivých modulů, které bude možné přidávat či upravovat dle potřeb daného zákazníka. Toto řešení bude také obsahovat vlastní komponenty, které poté do budoucna usnadní vývoj následujících modulů. Tato architektura bude hrát klíčovou roli ve schopnosti rychle a pružně reagovat na nové klientské požadavky a bude podporovat dlouhodobou udržitelnost systému, což bude přinášet další konkurenční výhodu.

Rozhodnutí zaměřit se v rámci této diplomové práce pouze na vývoj frontendové částí aplikace bylo motivováno rozdělením práce, což je způsob v praxi převládající. Jedná se o efektivnější způsob vývoje, kdy na projektu spolupracuje několik lidí, kteří se zaměřují na jednotlivé části vývoje. Díky tomu je možné se na jednotlivé části aplikace zaměřovat podrobněji a pečlivěji. U větších projektů je to již přímo nezbytné. Vývoj projektu je poté dále závislý na efektivní komunikaci. To je problematika, která je v práci také dále rozebrána.

Úvod této diplomové práce je nejprve věnován analýze aktuálního stavu backoffice systémů na trhu a poté se zabývá rozborem základních potřeb, které je pro tento systém nutné zvážit. Dále jsou v teoretické částí práce popsány a analyzovány technologie, které jsou potřeba pro efektivní vývoj takovýchto systémů.

V druhé části diplomové práce jsou již představeny základní požadavky na vývoj a představení návrhů moderních řešení. Tato část se také zabývá problematikou efektivní správy a komunikace v rámci vývoje projektu. Dále je popsána realizace důležitých prvků a částí aplikace a jejich samotná implementace.

# 1 PŘEDSTAVENÍ ŘEŠENÉ OBLASTI

Práce se pohybuje převážně v oblasti backoffice systémů. Ty se v poslední době staly nedílnou součástí firemního prostředí. S velikou poptávkou se rychle rozvijí trh a vznikají nové technologie pro efektivnější vývoj takovýchto systémů. V této kapitole autor přiblíží tyto systémy a požadavky, které jsou na ně kladeny.

## 1.1 Backoffice systémy

Backoffice systém je počítačový systém, který podporuje a spravuje různé operace, zpravidla v rámci firmy, která jej využívá. Typicky jsou využívány pro správu interních procesů a funkcí. Mezi backoffice systémy se řadí například správa skladového hospodářství, správa účetnictví, nebo správa zaměstnanců. Primárními cíli těchto systémů je zvýšit efektivnost procesů ve firmě a co nejvíce tyto procesy zautomatizovat. Ideálně tyto systémy sjednotí firemní data a umožní různým oddělením a pracovníkům ve firmě tyto data bezpečně a jednoduše sdílet.

Tyto systémy jsou oddělené od uživatelských systémů, které využívají zákazníci a klienti pro komunikaci s firmou. Tyto systémy se od sebe velmi liší, jelikož mají zcela odlišnou funkci. K takzvaným front office systémům má totiž přístup veřejnost a slouží zpravidla zákazníkům dané firmy. Takovéto systémy jsou u firem většinou přednější než backoffice systémy, jelikož mývají mnohem nižší náklady. Střední a větší firmy se ovšem bez nějaké formy backoffice systému zpravidla neobejdou.

Klíčovou vlastností backoffice systémů je, že mohou dále využívat systémy třetích stran. Tím se rozumí že integrují data z jiných systémů, a dále je využívají. K tomu je nutné systém přizpůsobit, aby tyto přenosy dat probíhaly bez nutnosti častých zásahů. I díky této vlastnosti jsou to systémy, které mají zásadní vliv na běžné fungování firmy, a hrají klíčovou roli při rozhodování o dalším vývoji firmy.

### 1.1.1 Současný stav

Na trhu je dnes vcelku široké množství řešení pro správu firmy. Ovšem ne každému podniku vyhovují obecná řešení a je nutné, aby se buď systému přizpůsobil, nebo aby se upravil daný systém. Pokud si podnikatel zvolí již hotové řešení backoffice systému, může s tím souviset okamžité používání a bezproblémové zařízení. Realita ovšem bývá často rozdílná. Hotový systém je potřeba správně implementovat v dané firmě a řádně vyškolit příslušné osoby. Další nevýhoda může nastat, pokud by podnikatel požadoval nějaké změny v systému, jako je například úprava ovládacích prvků. V těchto případech to často buď není vůbec možné,

popřípadě se jedná o nákladné záležitosti vzhledem k nízké náročnosti úpravy. I přesto je to velmi populární řešení a v základu bývá nejméně nákladné.

Další možností je nechat si systém vytvořit přesně podle svých potřeb. Tento přístup není vhodný pro každého, ale přináší určitě výhody. Nejdůležitější je uvědomit si, že tato možnost je vhodná, pokud již nějaký systém firma vlastní, nebo dokáže fungovat určitý čas bez něj. Vývoj takového systému totiž zabere čas, se kterým by měla být firma předem obeznámena a počítat s tím.

Výhoda nastává ve skutečnosti, že firma může zasahovat do vývoje systému již od samotného začátku, tedy může specifikovat své potřeby a nezbytné funkce, které budou klíčovou součástí systému. Dále může vyjádřit své preference k ovládání, vzhledu, a ostatním dílčím vlastnostem vyvíjeného systému.

Celkově dnes firmy mají mnoho možností, pokud shánějí řešení backoffice řešení pro jejich fungování. Téměř všechny moderní systémy dnes nabízí modulární řešení. To znamená že firma si může vybrat z několika připravených balíčků, podle svých potřeb a neplatit zbytečně za něco co nevyužije. Pokud firma zjistí že potřebuje spravovat další procesy ve firme, je zpravidla možné zakomponovat do backoffice systému další modul, který vyhoví daným požadavkům.

### 1.1.2 Konkurenční systémy

Mezi backoffice systémy, které jsou dnes populární patří například MySmartPlace. Ten aktuálně nabízí 5 různých balíčků s možností jejich kombinace. Baličky obsahují 3 až 5 modulů, které jsou vždy částečné provázány. Služby jsou nabízeny za měsíční předplatné, jehož výše se odvíjí od vybraných balíčků. [1]

Další zajímavou možností je MACROS od firmy FormSoft. Ti nabízí především skladové hospodářství, správu e-shopu a adresář zákazníků. Nabízí také doplňkové služby jako je individualizace aplikace či zaškolení. Zároveň si jejich systém lze pořídit za jednotnou cenu, bez dalšího měsíčního předplatného. [2]

## 1.2 Analýza potřeb

Backoffice systémy bývají klíčovou částí firem, a je tedy nezbytné dbát velký důraz při jejich návrhu a implementaci. Ze získaných informací a analýze potřeb bylo sestaveno několik základních požadavků a vlastností, které je potřeba brát v potaz při vývoji obecných backoffice systémů, jakým je i ten, kterým se zabývá tato práce.

### 1.2.1 Škálovatelnost

První základní vlastností, kterou se je potřeba zabývat je škálovatelnost. Je potřeba vytvořit systém tak, aby mohl růst a vyvíjet se spolu s danou firmou a stále plnit veškeré její potřeby. Tento problém se v dnešních systémech řeší často modularitou. To v praxi znamená, že se systém rozdělí na dílčí části, a ty se poté dají již vcelku snadno propojit mezi sebou. Tím se zajistí možnost vyvinout nový modul pro další potřeby firmy.

### 1.2.2 Pružnost systému

S modularitou systému souvisí i jeho pružnost. Systém by neměl být statický a těžkopádný. Již při návrhu, ale převážně při samotné implementaci bychom se měli snažit, aby byl systém a samotné moduly snadno upravitelné dle specifických potřeb firmy. Pokud například firma upraví logo, tak může zjistit, že je systém navržený tak, že změna v jednotlivých modulech způsobí problémy a v některých případech takovou změnu ani nepůjde provést. Takovým problémům lze tedy často předcházet dodržováním správných technik při vývoji.

### 1.2.3 Přívětivé rozhraní

Nesmíme také zapomenout na to, že systém budou obsluhovat pověření lidé, kteří nemusí být často vyškoleni a v takových systémech nejsou zvyklí pracovat. To platí zpravidla v menších firmách. Z těchto důvodů nesmíme opomenout přívětivé prostředí, a především intuitivní ovládání celého systému. To se může týkat například správných a výstižných popisů funkcí v systému, využití obecně známých ovládacích prvků, či jednoduché ikony u textů pro větší přehlednost. V krajních případech, pokud nelze prvek zjednodušit, je vhodné poskytnout uživateli vhodně umístěnou nápovědu.

### 1.2.4 Spolehlivost

Jak již bylo zmíněno, backoffice systémy bývají páteří dané organizace, a je nutné, aby se na takový systém mohli plně spolehnout. Systém by měl disponovat minimální chybovostí. Je tedy důležité předvídat různé scénáře které mohou v rámci systému nastat, a správně je ošetřit. Je také nezbytné mít k systému stálý přístup, a tedy minimalizovat neplánované výpadky.

### 1.2.5 Integrace

Integrace softwaru třetích stran je důležitým aspektem při vývoji backoffice systémů, jelikož umožní vývojářům přidávat další funkce efektivněji a snadněji. To díky tomu, že mohou využívat již hotové komponenty, které jsou již vyzkoušené a uživatelé s nimi mohou být již seznámeni. Tato vlastnost systému může poté ušetřit mnoho prostředků a pomoct ke vzniku efektivnějšího systému. Je také potřeba zvážit, zda závislost na jiném systému nemůže v budoucnu způsobit problémy. Systémy, které integrujeme do našeho, bychom si tedy měli předem řádně prověřit. [3]

### 1.2.6 Zabezpečení

Zabezpečení je kritický aspekt při vývoji backoffice systémů. Backoffice systémy zpravidla obsahují velmi citlivé informace o celé organizaci, jejím chodu a o jejích zaměstnancích. Je tedy klíčové vytvořit co nejbezpečnější prostředí a pokusit se zamezit úniku dat, nebo jejich nekonzistentnosti. Do systému by měli mít přístup pouze pověřené osoby, což zajišťuje správná autentizace. Následně v systému často figurují uživatelé, co mají různá oprávnění. To se řeší ve fázi autorizace, kde se zjistí, jakou má uživatel v systém roli, a na základě toho, jaké má v systému samotné oprávnění.

# 2 TECHNOLOGIE VYUŽITELNÉ PRO VÝVOJ

Backoffice systémy se rychle rozšiřují a firmy požadují více a více funkcí. Díky tomu také rychle vznikají nové technologie a způsoby, které zefektivňují vývoj takovýchto systémů. V této kapitole se seznámíme s typy backofficových systémů a technologiemi, které se využívají pro jejich vývoj.

## 2.1 Architektura aplikace

Dnes se při realizaci informačních systémů využívají dva základní typy architektur. Obě mají své výhody a nevýhody. Při vývoji nových systémů je potřeba zvážit veškeré požadavky a zvolit nejvhodnější architekturu.

### 2.1.1 Desktopová aplikace

První z možných řešení jsou takzvané desktopové aplikace. Ty jsou potřeba nainstalovat na počítač uživatele. Takové aplikace uživatel spustí v rámci svého operačního systému. Spuštěná aplikace poté zpravidla nabízí uživatelské rozhraní, díky kterému jí uživatel ovládá. Mezi hlavní výhody desktopových aplikací se řadí například plný přístup k hardwaru klientského počítače. To znamená snadnou možnost využití mikrofonu či připojené čtečky kódů. [4]

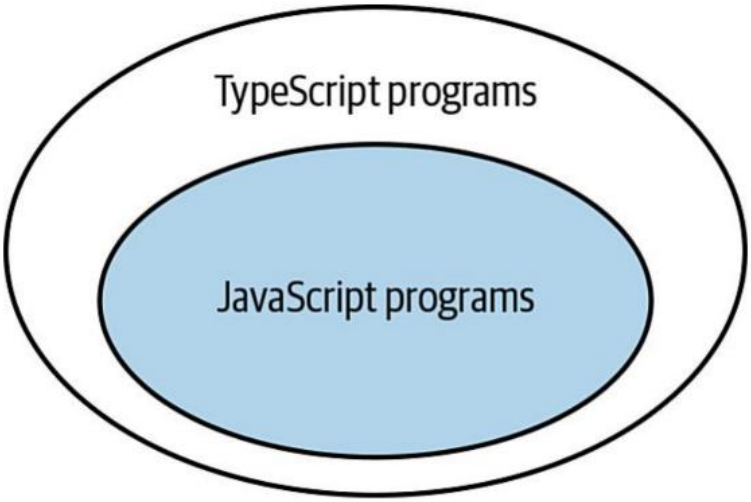
### 2.1.2 Webová aplikace

Oproti tomu jsou v poslední době více a více populárnější webové aplikace. Webová aplikace využívá architekturu klient-server. Aplikace je tedy umístěna na vzdáleném serveru a uživatel k ní přistupuje vzdáleně, typicky skrze webový prohlížeč. Zásadní výhodou webových aplikací je, že díky skutečnosti že jsou spuštěné na vzdáleném serveru, mají velmi malé nároky na klientský počítač. Webový prohlížeč je již základní součástí každého uživatelského počítače a není tedy nutné žádné jiné specifické vybavení. [4]

## 2.2 TypeScript

TypeScript je programovací jazyk, který vznikl již v roce 2012. TypeScript je unikátní v tom, že se nechová jako interpret, ani nekompiluje kód do nižšího programovacího jazyka. Oproti tomu opět kompiluje do vyššího programovacího jazyka, kterým je JavaScript. [5]

Vztah TypeScriptu s JavaScriptem je tedy zásadní. Pochopení jejich provázání je zásadní pro tvorbu silnějších a efektivnějších aplikací. TypeScript tvoří nadstavbu nad známějším a populárnějším JavaScriptem.




Obrázek 1: TypeScript vs JavaScript. Zdroj: [5]

To v praxi znamená, že pokud napíšeme bezchybnou část programu v jazyku JavaScript, bude se stejně tak jednat o část kódu, kterou je možné spustit v jazyku TypeScript. Je ovšem možné, že k danému kódu může vznést TypeScript varování.

TypeScript využívá příponu souborů .ts, obdobně jako JavaScript využívá .js. Podpora jejich konverze je velmi užitečná. To je výhoda, pokud přepisujeme program z JavaScriptu do TypeScriptu. To ovšem neplatí, pokud bychom chtěli konvertovat program naopak, jelikož TypeScript přináší určité změny, které JavaScript nepodporuje. [5]

TypeScript, jak může být patrné již z jeho názvu, přináší především efektivnější typovou kontrolu a změny s tím spojené.

```
let city = 'New york city';
console.log(city.toUppercase());
```

Obrázek 2: JavaScript kód.

Pokud se například pokusíme v JavaScriptu spustit jednoduchý kus kódu z obrázku výše, tak narazíme na error: „TypeError: city.toUppercase is not a function“.

```
let city = 'New york city';
console.log(city.toUppercase());
// TS2551: Property 'toUppercase' does not exist on type
// 'string'. Did you mean 'toUpperCase'?
```

Obrázek 3: JavaScript kód v TypeScriptu.

Pokud ovšem stejný kód spustíme v TypeScriptu, zjistíme, že jeho kontrola dokázala snadno nalézt přesný problém a upozornit na chybu, i s návrhem možného řešení. A to i když v programu nejsou specifikované žádné typy. [5]

## 2.3 Angular

Angular je JavaScriptový framework vyvinutý a podporovaný firmou Google. Je předně optimalizován pro TypeScript. [7] Pokud se dnes bavíme o frameworku Angular, zpravidla se odkazujeme na Angular 2. Ten sice vychází z Angular 1.x, ovšem jedná se o kompletně novou architekturu vyvinutou zcela od nuly v jazyce TypeScriptu, a stále dostává nové aktualizace a nové funkce. Základem v Angular architektuře jsou komponenty. Navíc využívá velmi dobrý systém detekce změn, pro rychlou odezvu a propagaci vazeb v komponentách. [17]

### 2.3.1 Modularita

Aplikace psané ve frameworku Angular zpravidla dodržují koncept modulárního programování. Modulární programování je technika rozdělování kódu na logicky nezávislé a zaměnitelné moduly. K tomu Angular využívá systém zvaný „ngModules“ Každý modul poté může obsahovat různé komponenty a služby, které jsou definované v daném modulu. Kořenový modul Angular aplikace se nazývá „AppModule“. Tento modul mají všechny Angular aplikace. Pokud je aplikace rozsáhlejší, zpravidla jsou rozděleny do více pod modulů, které obsahují. [6]

Každý modul je definován třídou s anotací „@NgModule()“. Třída s touto anotací poté musí splňovat několik vlastností a obsahovat určitě parametry. Mezi ty patří například vlastnost „declarations“. V té se deklarují veškeré komponenty v daném modulu. Každá komponenta musí být definována v příslušném modulu, aby bylo možné ji využívat. Další vlastností je „imports“. Zde je možné zapsat jiné moduly, které jsou potřeba, a chceme je využívat v aktuálním modulu. Je možné využít i další vlastnosti, jako „exports“, nebo „providers“. [6]

### 2.3.2 Komponenty

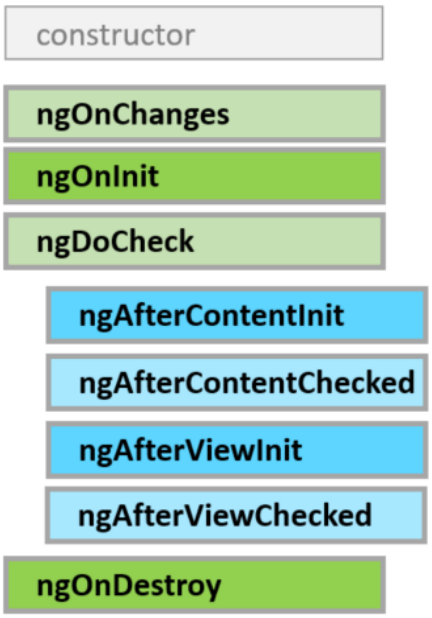
V Angularu můžeme u tříd využít anotaci „@Component“. Tím třídu definujeme jako komponentu, což je základní stavební kámen Angular aplikací. Každá komponenta k sobě většinou dále váže HTML šablonu a CSS styly. Sama komponenta poté zpravidla obsahuje logiku a spolu s šablonou a styly poté tvoří takzvaný pohled.

Komponenty obsahují několik základních vlastností, které se zde mohou definovat. Takzvaný „selector“ slouží k definování vlastního html tagu pro danou komponentu. Vlastností „templateUrl“ poté můžeme definovat relativní cestu k naší šabloně komponenty.

Angular pro propojení šablony a komponenty využívá takzvanou „two way binding“, neboli oboustrannou vazbu. Její použití poté dává vzniku komplexnějším pohledům. Nejjednodušším způsobem zobrazení dat z komponenty v šabloně je pomocí složených závorek. Tedy pokud máme v komponentě proměnnou, v šabloně použijeme zápis {{název_proměnné}} pro její zobrazení. Další způsob je využití direktivy ngModel. Díky té můžeme proměnnou z komponenty svázat s HTML ovládacím prvkem, jako je například textové pole. Takto svázaná proměnná se dá poté upravovat jak logikou komponenty, nebo zásahem uživatele z šablony. [6]

### 2.3.3 Životní cyklus

Každá Angular komponenta obsahuje několik metod životního cyklu. Při vytvoření nové instance třídy v Angularu se nejprve inicializuje konstruktor. Ten ovšem není metodou životního cyklu. Není také vhodnou praktikou konstruktor třídy využívat pro logiku aplikace nebo jako získávání dat. Konstruktor by měl sloužit pro tvorbu komponent a direktiv třídy. Po konstrukci třídy Angular volá metody životního cyklu při různých situacích.




Obrázek 4: Metody životního cyklu v Angularu. [8]

První metoda životního cyklu, která se volá po konstrukci, je „ngOnChanges()“. Tato metoda se volá pokaždé, kdy se detekuje změna ve vstupních parametrech komponenty. Může se jednat například o stisk klávesy. Jedná se o první metodu, které se volá ihned po konstrukci.

Další důležitou metodou je „ngOnInit()“. Tato metoda se v rámci životního cyklu komponenty volá právě jednou. Tato metoda se využívá pro inicializační logiku komponenty. Je vhodné zde načíst data, potřebná pro funkci komponenty.

Před samotným uvolněním komponenty se zavolá metoda „ngOnDestroy()“. Zde je prostor pro upozornění jiné částí aplikace o rušení dané komponenty. Slouží také pro uvolnění využívaných prostředků.

### 2.3.4 Služby

Angular je navržen, aby podporoval vkládání služeb do komponent. Komponenty jako takové by měly obsahovat především funkce a logiku vztažené k danému pohledu. Ostatní pomocné funkce, jako rozhraní pro komunikaci s API, je dobré definovat v pomocných službách, mimo samotné komponenty. Pro tyto účely je vhodné využívat „Dependency injection“. Jedné se o součást Angularu, která mimo jiné poskytuje právě komponentám možnost přístupu ke službám a ostatním prostředkům. Pro správnou funkčnost musí být komponenta, která má představovat službu, označena anotací „@Injectable()“. Takovou službu poté musíme vložit do konstruktoru komponenty, která jí poté bude moct využívat. [6]

```
constructor(private service: Service) { }
```

Obrázek 5: Definice služby v komponentě.

### 2.3.5 Alternativy

Angular není jediný JavaScriptový framework, mezi další se řadí například React. React je aktuálně mezi JS frameworky nejpopulárnější. Je spravován společností Facebook. Oproti Angularu, který rozděluje komponenty na logickou část a šablonu, v Reactových komponentách tyto dvě části splývají. Respektive jsou obsaženy ve stejném souboru. K tomu React využívá JSX. JSX je rozšíření syntaxe JavaScriptu. Na první pohled se jeho syntaxe podobá šablonovacímu jazyku, ovšem disponuje plnou podporou JavaScriptu. [6], [7]

```
const name = 'My name';
const element = <h1>Hello, {name}</h1>;
```

Obrázek 6: Ukázka JSX zápisu.

V neposlední řadě je více a více populární také Vue. Vue je rychle se rozvíjející JavaScriprový framework. I když není rozšířen, tak jako React či Angular, těší ve veliké přízni, a to především u nových programátorů. Vue využívá velmi podobný způsob vazby dat jako Angular. K proměnným se v šabloně přistupuje také pomocí zápisu: {{název_proměnné}}. [6]

## 2.4 Representational State Transfer

REST (Representational State Transfer) je softwarový architektonický styl, který definuje sadu pravidel a standardů pro tvorbu flexibilních a lehce spravovatelných webových služeb. REST je postaven na HTTP protokolu, a využívá standardní metody, jako GET, POST, PUT a DELETE. [11]

Důležitým pojmem je poté REST API. API (Application Programing Interface) je definováno jako soubor protokolů a nástrojů které se využívají pro vývoj softwarových aplikací, které umožnují více aplikacím komunikovat mezi sebou. Definuje také způsoby, jak mezi sebou dané aplikace mohou komunikovat. Pokud takové API navíc splňuje principy REST architektury, říkáme mu RESTful API. [10], [11]

### 2.4.1 Client-Server

Jeden ze základních principů je dodržování Client-Server architektury. REST rozděluje systém na klienta a server. Ve kterém klient zasílá požadavky na server, a server zasílá zpět odpovědi. Jedná se o rozdělení zodpovědnosti, které umožnuje snadnou rozšiřitelnost a rozdělení na jednotlivé části systému. [10], [11]

### 2.4.2 Cacheability

Pro splnění RESTful principů by webové služby měly být kešovatelné. To je proces ukládání výsledků operací do dočasné paměti, jako je mezipaměť webového prohlížeče. Při dalších dotazech na stejnou službu se odpovědi načítají již z mezipaměti, což může výrazně redukovat zatíženost serveru a zlepšit výkon celé aplikace. Ovšem ne všechny odpovědi jsou vhodné pro kešování. Některé odpovědi, obsahující například osobní údaje by kešovány být neměly, aby se zabránilo ohrožení bezpečnosti těchto dat. Stejně tak, jako odpovědi, které se často mění, u těch kešování také není žádoucí. [10], [11]

### 2.4.3 Statelessness

RESTfull webové služby jsou bez stavové. To znamená, že každý požadavek od klienta obsahuje veškeré informace, potřebné pro splnění požadavku, a nevzniká závislost na stavu uchovávaném na straně serveru. Každý požadavek z toho důvodu obsahuje hlavičku, parametry, autorizační detaily, tělo a URL adresu. [11], [12]


Obrázek 7: Struktura HTTP požadavku.[11]

### 2.4.4 Layered system

Díky vrstvené architektuře je možné zajistit vyšší spolehlivost, přehlednost a bezpečnost systému. Jednotlivé vrstvy mohou poté komunikovat pouze se sousedící, čím se zajistí více stabilní a předvídatelný systém. Sám klient poté ani nemusí vědět, zda komunikuje s cílovým serverem, nebo zda je mezi ním proxy server, nebo jiná podobná vrstva. [10], [11]

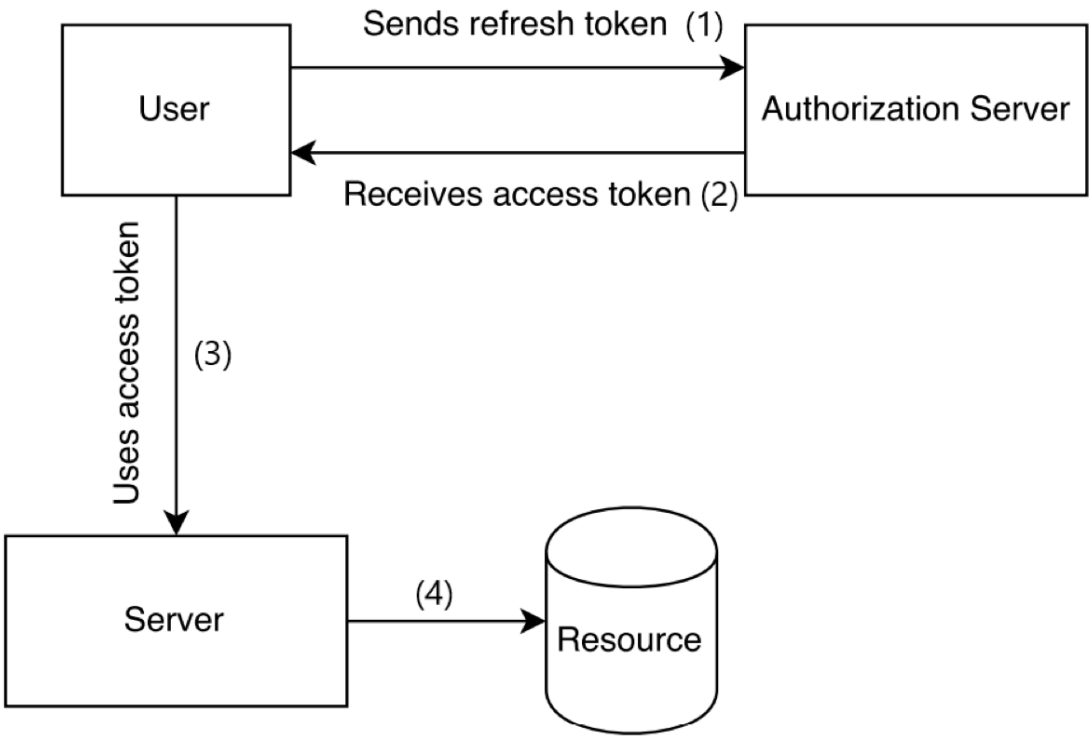
## 2.5 Autentizace pomocí JSON Web Token

JWT neboli JSON Web Token je standard pro přenos autentizačních údajů mezi serverovou a klientskou částí aplikací. Tyto údaje jsou ve formátu JSON a zpravidla jsou dále zašifrovány. V této podobě se poté přenáší v hlavičce HTTP požadavku.

Hlavní vlastností JWT je poté skutečnost, že se ukládá na straně klienta. To znamená, že se přenáší mezi klientem a serverem během každého požadavku na daný server. Server poté na základě informací v JWT dokáže ověřit identitu klienta. Tento způsob má několik výhod, mezi které se řadí například efektivnost ve velkých aplikacích. Server pro ověření platnosti JWT totiž nepotřebuje prohledávat databázi, a oproti jiným metodám se jedná se o relativně rychlý a snadný proces. [9]

### 2.5.1 Použití

V samotných aplikacích se JSON Web Token může implementovat pomocí takzvaných přístupových a obnovovacích tokenů (access a refresh). Tyto dva typy tokenů se využívají společně. Přístupový token se využívá pro autentizaci na serveru, ale jeho platnost je ovšem omezena na určitou dobu. Po uplynutí této doby je nutné využít obnovovací token, který má většinou dlouhou platnost. Ten se zašle spolu s požadavkem na server, který poté vrátí nový přístupový token. Tímto způsobem má server větší kontrolu nad platností a bezpečností tokenů. [9]




Obrázek 8: Princip přístupového a obnovovacího tokenu. [9]

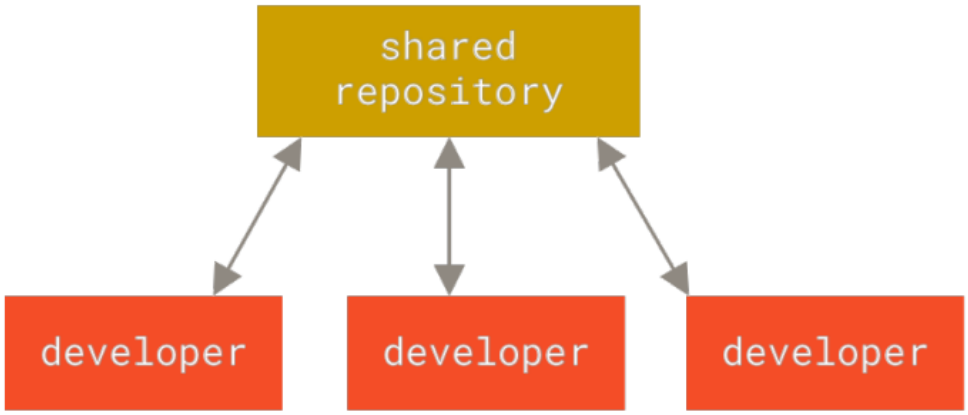
## 2.6 Verzovací systémy

Verzovací systémy slouží pro zaznamenávání změn jednoho nebo více souborů za určitý časový úsek. Pro vývoj softwaru jsou verzovací systémy zásadní. Hlavní funkcí je poté možnost zobrazit a popřípadě se vrátit ke specifické historické verzi souboru. Verzovací systémy nám dále umožnují snadnou správu spolupráce více lidí na jednom projektu. Verzovací systém totiž zaznamenává každou změnu a informace o tom, kdo a v jakém čase změnu provedl. [13] Verzovací systémy mohou být lokální, nebo centralizované na sdíleném serveru. Nejjednodušší metoda verzování je zkopírovat soubory do jiné složky, pojmenované například podle datu a času. Jedná se o snadný způsob verzování, ovšem je velmi náchylný k chybám a při větším množství souborů se může stát velmi nepřehledným. Existují samozřejmě specializované programy, jako například Revision Control System. [13], [14]

### 2.6.1 Centralizované systémy

Pokud chceme udržovat projekt, na kterém spolupracuje více lidí, nebo jenom potřebujeme mít k souborům přístup z více míst, lokální verzovací systém nám již stačit nebude. Po mnoho let byl v těchto případech standart právě centralizovaný verzovací systém. Mezi tyto systémy se

řadí například stále populární Subversion. Při jeho využívání poté existuje jeden centrální server, který zaznamenává veškerou historii sledovaných souborů, ke které mohou klienti přistupovat. [13], [14]

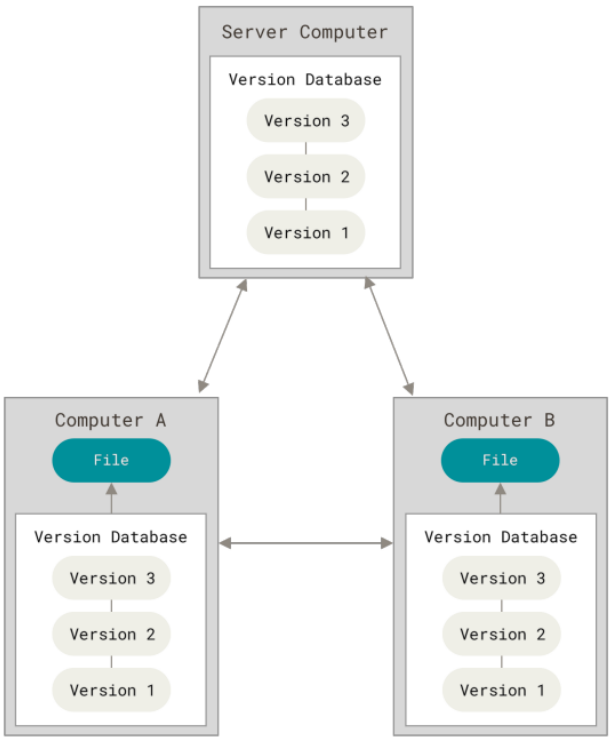



Obrázek 9: Centralizovaný verzovací systém. [14]

Tento systém má mnoho výhod, zvláště oproti lokálním verzovacím systémům. Ovšem obsahuje také určité problémy a nedostatky. Hlavní je, že se spoléhá na daný server. Pokud server přestane fungovat, k souborům ani k jejich historii se nikdo během té doby nedostane. Pokud se data na serveru dokonce poškodí, tak je celá historie ztracena. Jediná zachovalá data jsou části souborů, které zrovna někdo využíval a má je uložené na svém zařízení. [13], [14]

### 2.6.2 Distribuované verzovací systémy

Distribuované systémy jsou dnes nejpopulárnější skupinou pro verzování. Jedná se systémy jako Git, nebo Mercurial. Klienti využívající tyto systémy nemají pouze poslední verzi systému, ale zcela zrcadlí repositář, a to včetně historie změn. Tím se eliminuje centrální bod, který může selhat.




Obrázek 10: Distribuovaný verzovací systém. [14]

Tyto systémy také výborně fungují se vzdálenými repositáři. Poskytují nástroje a možnosti pro řešení konfliktů a zajištění správné integrity dat. Tyto systémy také velmi podpořili koncept „forkingu“, neboli větvení, především v rámci otevřeného sdílení. Jedná se o možnost naklonovat jiný repositář k vytvoření alternativních změn a úprav. Hlavní výhoda poté spočívá v možnosti aktualizace repositáře z jiného. Ostatní mohou udělat to samé s našimi změnami. [13], [14]

### 2.6.3 Implementace Git

Nejčastěji se dnes můžeme setkat s využitím Gitu spolu s hostovaným repositářem. Jedna z nejpopulárnějších a největších hostovacích platforem je GitHub. Jedná se o platformu, které mimo hostování Git repositářů, umožňuje mnoho dalších nástrojů. GitHub sám o sobě hostuje velké procento existujících Git repositářů, ale vývojáři zde mohou využít například správu a sledování problémů v kódu, kontrolu kódu, nebo pokročilou možnost kooperace. Mezi další známe platformy pro Git se řadí například BitBucket, nebo GitLab. [13], [14]

## 2.7 Pipeline

Mezi naprogramováním aplikace a jejím vydáním je několik důležitých kroků, mezi které patří sestavení, testování a nasazení samotné aplikace. Mnoho moderních aplikací jsou veliké

a náročné pro manuální správu těchto kroků. Při manuální správě je také aplikace náchylnější vůči lidským chybám. [15]

Z toho důvodu existují nástroje, které dokáží tyto kroky plně automatizovat. Dnes se využívá takzvaný CI / CD Pipeline. CI (Kontinuální integrace) slouží k automatickému sestavení aplikace. To umožnuje častější úpravy a včasné odhalení problémů. Během kroku CD (Kontinuální nasazení) aplikace projde dalšími testy a dále je automaticky nasazena do předem definovaného prostředí. [16]

## 2.8 Řízení projektu

Při vývoj softwaru, na kterém se podílí více lidí, je řízení daného projektu zásadní částí. Zahrnuje to plánování, organizaci, a přehled nad časem, zdroji a aktivitami potřebnými pro úspěšný vývoj. Je potřeba mít přehled na možnými riziky a včas je identifikovat.

Mezi úspěšné nástroje, které nám s tím mohou pomoci, se řadí Jira. Jira je populární nástroj pro řízení a sledování projektů. Jedná se webovou aplikaci, která nabízí širokou škálu funkcí. Při každém projektu je možné si zvolit z dvou oblíbených metod, a to kanban, nebo scrum. Jednou z klíčových vlastností je dále sledování požadavků, v rámci celého životního cyklu. Požadavky je možné přiřazovat jednotlivým členům týmu, upravovat priority, přidávat komentáře a zpravovat čas strávený nad nimi. Jira poté nabízí souhrny a přehledy které obsahují přehledné vizualizace skrz celý životní cyklus projektu. [18]

# 3 ANALÝZA ŘEŠENÉHO SOFTWARU

V této kapitole jsou rozebrány požadavky a očekávané funkce vyvíjeného systému. Jsou zde popsány jednotlivé funkce a jejich možné řešení v moderních systémech.

## 3.1 Požadavky aplikace

V rámci projektu byly definovány funkční a nefunkční požadavky, které slouží jako základní směrnice pro vývoj aplikace. Funkční požadavky popisují očekávané funkcionality a chování aplikace, zatímco nefunkční požadavky se zaměřují na aspekty jako bezpečnost, uživatelskou přívětivost a další.

Mezi funkční požadavky patří například možnost registrace nových uživatelů, přihlašování do systému, správa uživatelských účtů, správa dat a další specifické funkce, které mají být implementovány v rámci aplikace. Tyto požadavky jsou přesně popsány ve funkčním seznamu a slouží jako základní kameny pro vývoj.

Nefunkční požadavky se zaměřují na kvalitu, strukturu a bezpečnost aplikace. Mezi ně mohou patřit požadavky na rychlost aplikace, škálovatelnost, použití bezpečnostních mechanismů jako autentizace a autorizace, a také přehlednost a udržovatelnost kódu. Tyto požadavky jsou rovněž důležité pro úspěšné fungování aplikace a pro splnění očekávání uživatelů.

Bližší seznam funkčních a nefunkčních požadavků je zobrazen na obrázku níže a slouží jako klíčový prvek pro přesné porozumění požadavkům klienta a úspěšnou realizaci projektu.

| Funkční požadavky | |
|---|---|
| FR.1. | Modulární architektura |
| FR.2. | Správa uživatelů |
| FR.3. | Rozšířitelná lokalizace |
| FR.4. | Znovupoužitelné komponenty |
| FR.5. | Zabezpečený přístup do systému |
| FR.6. | Automatická aktualizace dat |
| FR.7. | Chybové hlášky |
| FR.8. | Změna oprávnění |
| FR.9. | Blokování uživatelů |
| FR.10. | Vyhledávání adresy na mapě |
| FR.11. | Registrace nového uživatele |
| FR.12. | Přihlašování do systému |

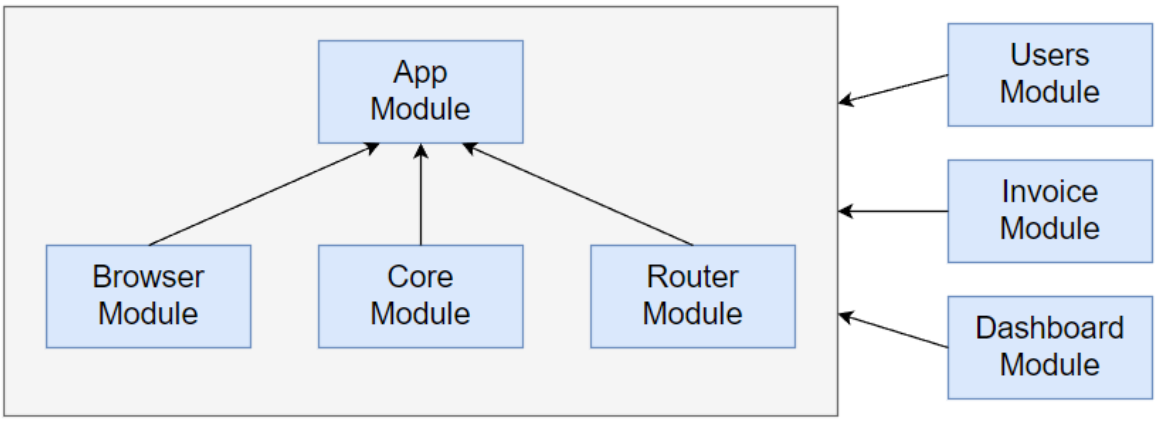
| Nefunkční požadavky | |
|---|---|
| NR.1. | Snadná integrace nových modulů |
| NR.2. | Flexibilní úprava funkcí |
| NR.3. | Konfigurace modulů |
| NR.4. | Ošetřené chyby |
| NR.5. | Konzistentní data |
| NR.6. | Ošetřené vstupy uživatele |
| NR.7. | Napojení na Google mapy |
| NR.8. | Ošetřené vstupy uživatele |
| NR.9. | Řízení chyb a zpětná vazba |
| NR.10. | JWT Autentizace |

Obrázek 11: Funkční a nefunkční požadavky.

## 3.2 Struktura aplikace

Jednou ze základních vlastností navrhovaného systému je modularita. Jednotlivé moduly bude možné nezávisle upravovat a popřípadě nahrazovat dle požadavků klienta. K tomu bude nutné navrhnout strukturu projektu pro snadnou a přehlednou správu modulů. V rámci Angular, ve kterém je aplikace vyvíjena, je možné nalézt mnoho způsobů organizace. Ovšem na každý projekt je nutné nahlížet zvlášť a řídit se dle jeho požadavků.

Angular aplikace mají vždy alespoň jeden modul. Takzvaný AppModule, kterému se také říká root. Ten je zodpovědný za start aplikace a nahrání ostatních potřebných modulů, pokud aplikace nějaké má.

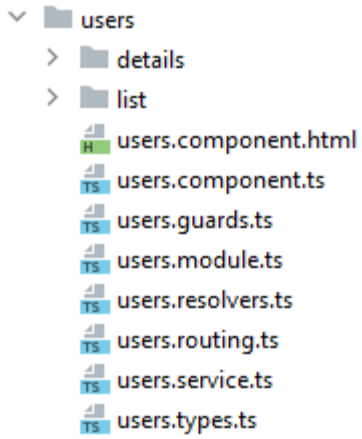



Obrázek 12: Ukázka struktury modulů v aplikaci.

Jak je vidět na obrázku výše, hlavním modulem aplikace je AppModule, který se načte jako první a dále načte další závislé moduly. Mezi ty důležité se řadí i RouterModule. Ten obsahuje nastavení všech hlavních cest a umožnuje tedy navigaci v systému.

### 3.2.1 Struktura modulu

Jednotlivé moduly v aplikaci musí podléhat určitým standardům, a to především pro lepší srozumitelnost a přehlednost aplikace. Pokud dodržujeme jednotnou strukturu našich modulů, je poté snadnější vytvářet nové a zároveň se orientovat v těch starých.




Obrázek 13: Struktura modulu users.

Obrázek výše nám ukazuje ideální strukturu pro typický modul v aplikaci. Zde je vidět uživatelský modul, který je jeden z nejzákladnějších v aplikaci.

Jako první můžeme vidět další podsložky, ve kterých jsou potřebné komponenty pro různé detaily a potřebné karty pro správu jednotlivých uživatelů. Dále je složka list, která obsahuje komponentu pro zobrazení stránky se seznamem uživatelů. Tyto komponenty se mohou velmi lišit, v závislosti na typu a určení modulu.

Pokud se ovšem podíváme na strukturu odspodu, vidíme soubor users.types. Ten slouží pro definici typů objektů v rámci modulu. V příkladu uživatelů je zde tedy vytvořen například objekt uživatele. Ten se poté jednotně využívá v rámci celého modulu.

Následuje třída users.service. Ta se využívá pro správu veškerých služeb, které spadají do daného modulu. Třída služeb tedy bude obsahovat veškeré definice pro komunikaci s RESTovým API. V modulu uživatelů to budou veškeré metody týkající se uživatelů, které se v daném modulu budou využívat.

Další zmíněná třída bude users.resolvers. Ta využívá princip takzvaných resolverů. Ten spočívá v tom, že pokud přijde požadavek na načtení určité stránky, která implementuje resolver, tak se prvně zavolá její resolver. Ten dokáže asynchronně načíst potřebná data pro načítanou stránku. Vrácená data jsou poté předána načítané stránce, která je může použít pro inicializaci. Výhodou resolverů je především rychlejší odezva, jelikož data se mohou načítat před samotným načítáním komponenty, která data potřebuje. Dále můžou napomáhat ke konzistenci dat, jelikož ty se načítají pouze jednou při načítání komponenty. Jednotlivé definice resolverů bývají jednoduché a obsahují zpravidla odkaz na třídu services v daném modulu, které obsahuje patřičné volání dat.

Další třída, které se v modulech využívá je guards. Ta implementuje rozhraní canDeactivate, a slouží k provedení určitých akcí před přesměrováním na jinou stránku. V případě potřeby dokáže přesměrování úplně zastavit. Ve vyvíjené aplikaci se využívá především pro drobné úpravy při přesměrování mezi moduly, jako například pro uzavření bočního menu modulu.

```
export const usersRoutes: Route[] = [
    {
        path     : '',
        component: UsersComponent,
        children : [
            {
                path     : '',
                component: UsersListComponent,
                resolve  : {
                    contacts : UsersResolver,
                },
                children : [
                    {
                        path         : 'new',
                        component    : UsersDetailsComponent,
                        resolve      : {
                            contact  : UsersNewUserResolver,
                        },
                        canDeactivate: [CanDeactivateContactsDetails]
                    },
                    {
                        path         : ':id',
                        component    : UsersDetailsComponent,
                        resolve      : {
                            contact  : UsersUserResolver,
                        },
                        canDeactivate: [CanDeactivateContactsDetails]
                    }
                ],
            }
        ]
    }
];
```

Obrázek 14: Ukázka nastavení cest v rámci uživatelského modulu.

Na obrázku výše můžeme vidět hlavní část souboru users.routing. Jedná se definici veškerých cest v rámci daného modulu.

Soubor users.routing je jedno z nejdůležitějších v každém modulu, jelikož definuje jeho strukturu a chování, a to především, pokud je modul rozsáhlejší. Na předchozím obrázku můžeme vidět hlavní konfiguraci cest v modulu uživatelů. Při přechodu do tohoto modulu přesměrujeme na výchozí komponentu UsersComponent. Ta neobsahuje žádná data a slouží

pouze jako základní komponenta. Z té je výchozí přechod na UsersListComponent. Ta již obsahuje například svůj resolver, pro načtení seznamu uživatelů.

Ze seznamu uživatelů je poté možné přistoupit na tvorbu nového uživatele, nebo detail již exitujícího. Obě tyto cesty mají resolver pro asynchronní načtení dat. Dále obsahují příznak canDeactivate, který se volá bezprostředně před změnou stránky. Záznam pro přechod na detail uživatele ve své cestě také obsahuje parametr, viz „:id“. To znamená, že se cesta aktivuje, pokud parametr existuje a v komponentě je s ním poté možné dále pracovat.

Jako poslední je potřeba také zmínit soubor users.module. Každý angular modul musí tento soubor obsahovat. Jedná se o třídu s anotací @NgModule, která obsahuje veškeré deklarace a importy, které se v daném modulu využívají. Je to nejdůležitější součást modulu, která samotný modul tvoří.

## 3.3 Překlady

Jeden z požadavků bylo také, aby aplikace byla připravena na budoucí možné rozšíření lokalizace. K tomu se v rámci Angular aplikací využívá velmi verzatilní a užitečná funkčnost, které se říká Angular Internationalization, která je také často nazývaná i18n. Jedná se o klíčovou funkci, která byla implementována v rámci. Je to nezbytný nástroj pro snadné lokalizování a překlad různých typů aplikací.

Použitím i18n v Angularu poté umožnuje vytvářet aplikace, které jsou snadno přizpůsobitelné různým jazykům. Je důležité začít používat tuto funkcionalitu již na začátku vývoje nového projektu. Angular i18n umožňuje snadno přidávat nové jazyky do projektu a díky tomu je připravený na snadné rozšiřování. S pomocí i18n lze snadno lokalizovat texty, formátování dat a další jazykově závislé části aplikace, což zlepšuje uživatelskou přívětivost a přístupnost aplikace pro uživatele mluvící jinými jazyky.

Implementace Angular Internationalization je poměrně jednoduchá. V rámci projektu je vytvořen soubor pro každou lokalizaci, který obsahuje přeložené řetězce pro daný jazyk. Tyto soubory jsou typicky označeny zkratkou jazyka, například „cz.json“ pro českou lokalizaci. V těchto souborech jsou definovány klíče, které párují seřazené řetězce ve zvoleném jazyce. Klíče slouží jako identifikátory, které propojují původní řetězce v kódu s jejich překlady v lokalizačních souborech.

Ve zdrojovém kódu aplikace jsou tyto klíče použity jako direktivy pro Angular. Například, v HTML šablonách, místo přímo psaných řetězců, používáme direktivy typu {{ 'KEY' | translate }}, kde 'KEY' je klíč odpovídající požadovanému přeloženému řetězci. Angular poté vyhledá překlad tohoto klíče v odpovídajícím lokalizačním souboru a zobrazí přeložený text ve výsledném uživatelském rozhraní.

Celkově tedy implementace i18n ve frameworku Angular zahrnuje definování lokalizačních souborů pro každý jazyk, ve kterém chceme aplikaci lokalizovat, a použití klíčů v kódu pro párování řetězců s jejich překlady. Tímto způsobem umožňuje Angular snadno a efektivně přeložit aplikaci do různých jazyků a poskytnout uživatelům přístup k aplikaci ve zvoleném podporovaném jazyce.

## 3.4 Bezpečnost

Jeden z prvních a základních modulů v aplikaci je modul s názvem auth. Jedná se o nezbytnou součást tvořeného systému, která zajišťuje bezpečnost a správu přístupu do systému. Obsahuje komponenty pro přihlášení (sign-in), registraci (sign-up), odhlášení (sign-out) a dále komponenty pro obnovu hesla. Tento modul je navržen tak, aby poskytoval zabezpečení a efektivní správu přístupových práv.

Při autentizaci se využívá JWT (JSON Web Token) princip. Po úspěšném přihlášení je uživateli přidělen přístupový token (accessToken) a obnovovací token (refreshToken). Tyto tokeny jsou uloženy v lokálním úložišti pro pozdější použití. Tato část přihlášení je zobrazena na obrázku níže.

```
return this._accessService.loginUserWeb(credential).pipe(
    switchMap( project: (response : LoginResource ) => {

        this.accessToken = response.accessToken;
        this.refreshToken = response.refreshToken;

        this._authenticated = true;

        return of(response);
    })
);
```

Obrázek 15: Přidělení tokenů v metodě přihlášení.

Přístupový token se využívá k ověření identity uživatele a je odesílán v hlavičce každého žádosti na server. Obnovovací token je použit pro obnovení přístupového tokenu v případě jeho vypršení.

## 3.5 Autentizace a autorizace

Pro omezení přístupu k aplikaci a ověřování přihlašovacích údajů bude implementován systém, který zabezpečí, že pouze oprávnění uživatelé budou mít přístup k chráněným částem aplikace.

Při přístupu k aplikaci bude uživatel přesměrován na stránku pro přihlášení. Zde bude vyžadováno zadání přihlašovacích údajů, tj. uživatelského jména a hesla. Při zadávání údajů bude na klientské straně probíhat ověřování správnosti formátů zadávaných údajů. Po odeslání údajů bude provedena validace těchto údajů na serverové straně.

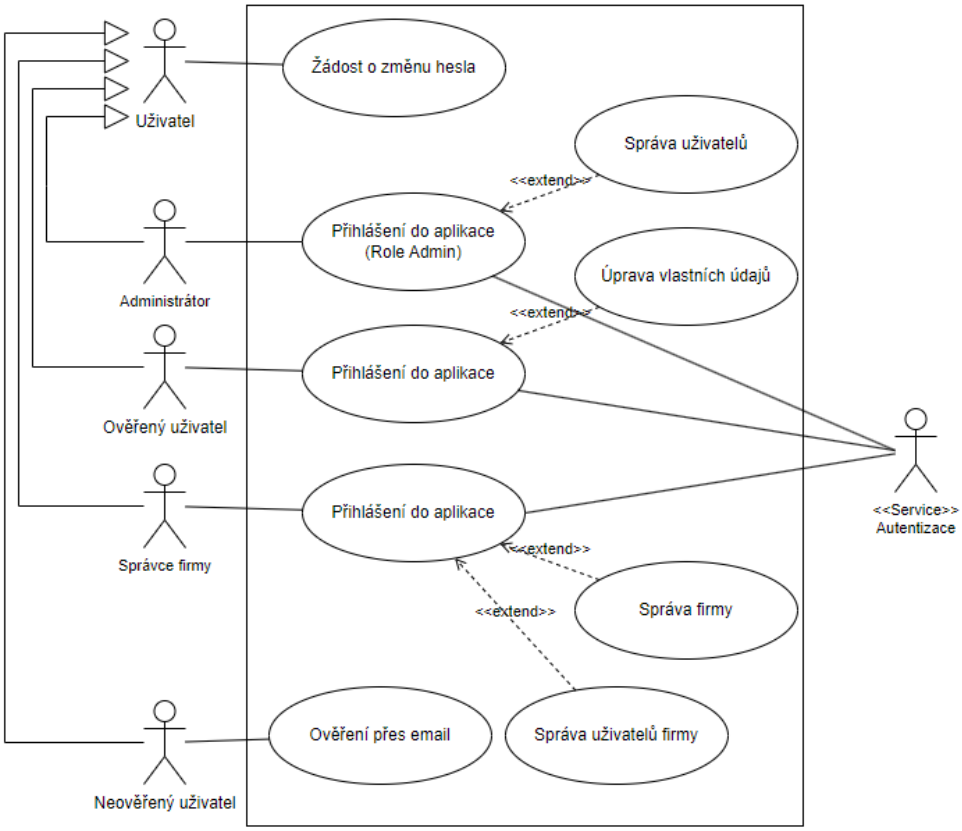
Pro validaci bude využito zabezpečeného API, které ověří zadané přihlašovací údaje. Pokud jsou údaje platné, server vrátí odpovídající přístupový token JWT, který bude uložen v lokálním úložišti. Tím bude uživatelovi umožněn přístup k chráněným částem aplikace.

Kromě přihlášení bude v rámci aplikace také implementována možnost registrace nových uživatelů. Uživatelé budou vyžádáni, aby poskytli své osobní údaje a vytvořili si jedinečné přihlašovací údaje. Mezi podmínky bude patřit vynucování zadání silného hesla, které se bude skládat z několika různých typů znaků a dostatečné délky. Takové heslo je nutné vynucovat i z důvodu absence dnes již rozšířeného dvou faktorového ověřování. Po úspěšné registraci bude uživatel dále vyzván pro ověření účtu potvrzením které bylo zasláno na zadaný email.

Dále bude v aplikaci implementována funkce pro obnovu zapomenutého hesla. Uživatelé budou mít možnost zadat svou e-mailovou adresu a požádat o odkaz pro resetování hesla. Na základě zadané e-mailové adresy bude vygenerován odkaz, který bude odeslán na uživatelem poskytnutý e-mail. Po kliknutí na odkaz bude uživatel přesměrován na stránku, kde bude moci zadat nové heslo.

### 3.5.1 Role

V systému jsou definovány různé role, které určují, jaké možnosti a oprávnění mají jednotliví uživatelé po přihlášení. Po úspěšném přihlášení vrátí endpoint pro přihlášení objekt obsahující informace o přihlášeném uživateli, včetně jeho rolí a oprávnění. Níže můžeme vidět diagram případů užití vyobrazující jednotlivě definované role s danými oprávněními v aplikaci.




Obrázek 16: Diagram případů užití zobrazující role a oprávnění v aplikaci.

Základní rolí je uživatel, který může zažádat o obnovu zapomenutého hesla. Tuto možnost mají všichni uživatelé systému. Po obdržení žádosti o obnovu hesla bude uživateli zaslán email s odkazem pro resetování hesla.

Pokud uživatel provede registraci, musí si nejprve ověřit svou identitu kliknutím na odkaz v potvrzovacím e-mailu. Teprve po této verifikaci bude mít přístup k přihlášení do systému. Registrovaný uživatel bude moci upravovat své vlastní údaje a nahlížet do svých sekcí.

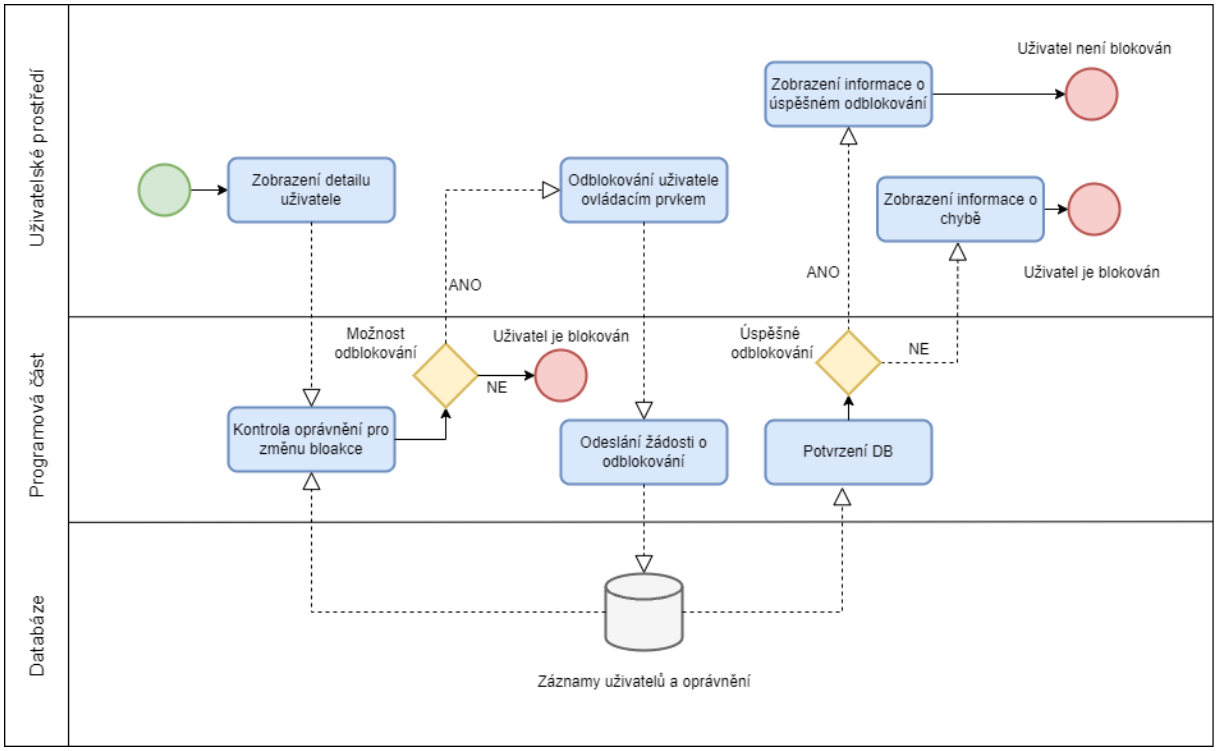
Administrátor systému bude mít možnost upravovat údaje všech uživatelů. Jeho role mu poskytne vyšší oprávnění a přístup ke všem administrativním funkcím.

Kromě toho existuje role správce firmy, která umožňuje správu dané společnosti a uživatelů v rámci této společnosti. Správce firmy bude mít oprávnění k úpravám a správě údajů uživatelů, kteří jsou přiřazeni ke stejné společnosti.

## 3.6 Funkce blokace

Na základě požadavků byla dále implementována funkce pro blokování uživatelů, která umožňuje přihlášenému uživateli s rolí administrátora blokovat/odblokovat jiné uživatele. Pokud administrátor zablokuje konkrétního uživatele, bude mu s okamžitou platností znemožněn přístup do aplikace. Blokovaný uživatel se poté při následném pokusu o přihlášení do systému setká s oznámením, že mu byl přístup do systému zablokován.

Na obrázku níže je zobrazen proces blokování uživatele pomocí procesního diagramu, který zobrazuje kroky a postupy, které se musí provést k provedení blokování uživatele.

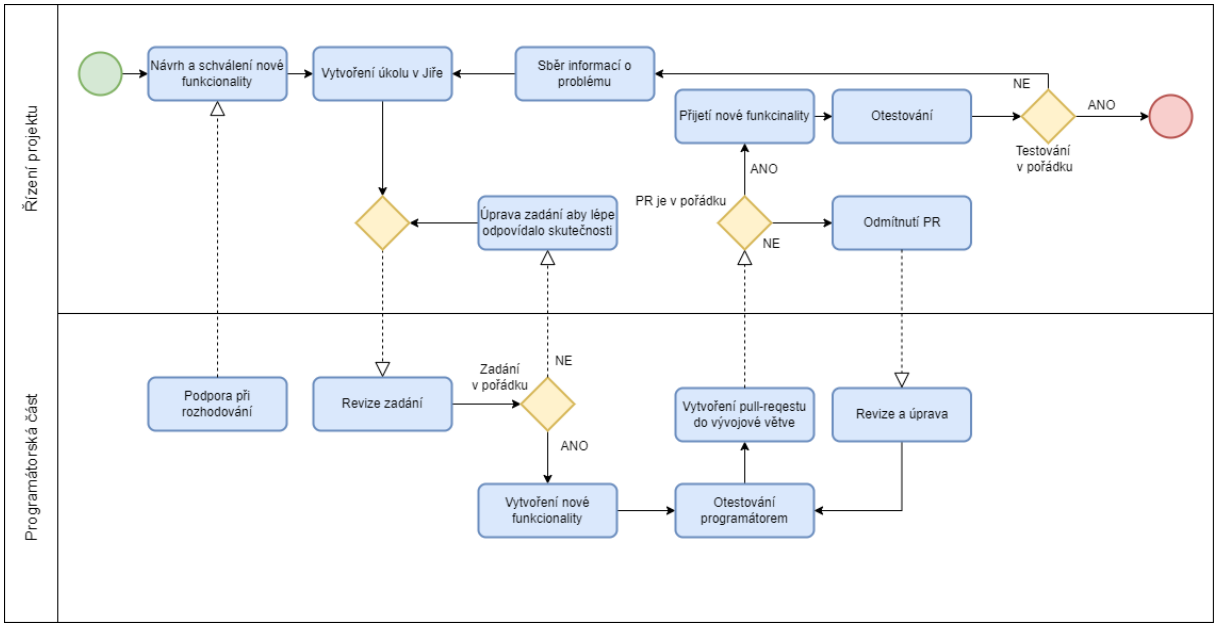



Obrázek 17: Diagram procesů blokování uživatele.

Implementace této funkce blokování přináší administrátorům aplikace možnost efektivně a rychle řídit přístup uživatelů a zajistit, že blokovaní uživatelé nemají možnost vstupu do systému či aplikace. Tím je zajištěna vyšší bezpečnost a kontrola nad přístupovými právy v projektu.

# 4 REALIZACE ŘEŠENÍ

Realizace zadané aplikace probíhala v různých etapách a v této kapitole jsou popsány hlavní body, kterými se bylo potřeba při vývoji zabývat. Především se dbalo na dodržování standartních postupů, jako při jiných projektech vyvíjených pro zákazníky. Tyto postupy zahrnují tvorbu a schvalování nových funkcionalit a jejich postupné přidávání za pomoci verzovacího systému. Na obrázku níže je vidět diagram který tento cyklus zachycuje.




Obrázek 18: Procesní diagram postupu vývoje.

## 4.1 Architektura aplikace

Architektura aplikace byla navržena s využitím moderního a efektivního nástroje NX (Extensible Build System). NX poskytuje rozšířený systém pro sestavení aplikací, který umožňuje strukturovat a spravovat složité aplikace a projekty. Aplikace je rozdělena do modulů a komponent, což zajišťuje lepší organizaci a snadnější správu kódu. Využívání pluginu NX umožnilo získat přehlednou a rozšiřitelnou strukturu projektu.

NX používá monorepozitářovou strukturu, což znamená, že celá aplikace je uložena v jednom repozitáři a je rozdělena do několika podsložek, jako jsou apps (aplikace), libs (knihovny), tools (nástroje) atd. Tato struktura umožňuje snadné sdílení kódu mezi různými částmi aplikace a usnadňuje správu a organizaci projektu. NX je také navržen tak, aby byl škálovatelný pro velké a komplexní projekty. Díky tomu lze bez problémů také přidávat nové funkcionality, moduly a komponenty, aniž by to ovlivnilo celkovou stabilitu a výkon aplikace.

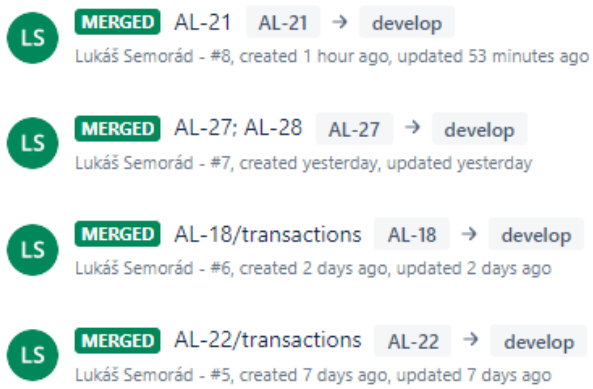
NX je potřeba implementovat již od samého začátku vývoje. Pomocí balíčkovacího systému Nmp, který byl využívám se NX snadno nainstaloval, a poté již bylo možné využívat příkazy samotného pluginu.

## 4.2 Správa aplikace

Při práci na tomto projektu byla využívána platforma Bitbucket jako správce verzí a kolaborační nástroj. Pro organizaci vývoje byl využit klasický model s větvemi „master“ a „develop“ pro produkční a vývojové prostředí. Každý úkol nebo funkce se implementuje ve vlastní větvi, která je následně začleněna do vývojové větve pomocí pull-requestu a následného sloučení (merge). Tímto způsobem je zajištěna lepší přehlednost a kontrola nad změnami v kódu.

Pro další zvýšení přehlednosti a správy úkolů se využívá software Jira, který je propojený s Bitbucketem. Dané nové funkcionality či jiné typy úprav se díky Jira integraci automaticky propojují s odpovídajícími větvemi v Bitbucketu. Tímto propojením je možné snadno sledovat pokrok jednotlivých úkolů, komunikovat s týmem a sledovat stav jejich dokončení. Spolupráce mezi Bitbucketem a Jirou usnadňuje synchronizaci mezi správou úkolů a správou verzí, a tím poskytuje celkově lepší organizaci a přehlednost při vývoji projektu.

Na obrázku níže je možné vidět samotné pull-requesty do hlavní vývojové větve. Názvy jednotlivých větví obsahují identifikátor úkolu, který je definovaný v Jiře. Samotný commit tedy díky tomuto systému vždy obsahuje novou funkcionalitu, které je popsána a zdokumentována.




Obrázek 19: Ukázka pull-requestů do hlavní vývojové větve.

Dále byly v rámci Bitbucketu využívány takzvané Pipelines. Ty byly využívány k instalaci, ověření kódu, sestavení a nasazení projektu. Tato automatizace těchto kroků se ukázala jako

skvělý nástroj, protože šetří čas a usnadňuje vývojový proces. Při každé změně v repozitáři byla spuštěna příslušná pipeline, která prováděla definované kroky. Například, při každém commitu byla spuštěna fáze ověření kódu a v případě úspěšného ověření byl vygenerován nový artefakt pro nasazení. Poté byla spuštěna fáze nasazení, která zajišťovala automatické nasazení nové verze projektu.

## 4.3 Kontrola kódu

V rámci projektu byl od samého začátku také využíván analytický nástroj pro kontrolu kódu Lint. Ten byl použit k ověřování chyb, nedostatků a nesrovnalostí v kódu. Kontrola kódu byla prováděna pravidelně během vývoje a poskytovala zprávy o nalezených chybách a nedodržení kódových standardů.

Pro implementaci a spuštění kontroly kódu pomocí nástroje Lint v projektu bylo zapotřebí několik kroků. Nejprve jsem musel nainstalovat balíček typescript-eslint, který slouží jako rozšíření pro Lint specificky pro TypeScript kód. Tento balíček je nezbytný pro provádění kontroly kódu v TypeScript projektu.

Poté bylo potřeba provést konfiguraci Lintu a typescript-eslint pro projekt. To zahrnovalo vytvoření souboru .eslintrc.json, ve kterém byly definovány pravidla a nastavení pro kontrolu kódu. V konfiguračním souboru se dá specifikovat například preferované formátování, pravidla pro kontrolu syntaxe a další parametry podle potřeb projektu. Na obrázku níže je vidět způsob zápisu pravidla ve výše zmíněném souboru .eslintrc.json. V tomto případě se jedná o pravidlo pro kontrolu stylu zápisu selektorů šablon a vynucování stylu kebab-case.

```
"@angular-eslint/component-selector": [
    "error",
    {
        "type": "attribute",
        "prefix": "",
        "style": "kebab-case"
    }
]
```

Obrázek 20: Pravidlo pro kontrolu nástrojem Lint.

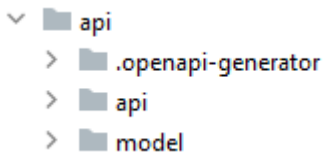
Lint obsahuje základní sadu pravidel, které se dají upravit dle specifických preferencí projektu či vývojáře. Po úspěšné konfiguraci se mohl spustit příkaz „npx nx lint tc“, k provedení kontroly kódu. Příkaz „npx nx lint“ spouští Lint pro konkrétní projekt (v tomto případě „tc“) v rámci projektového balíčkovacího systému NX. Tento příkaz projde zdrojový kód v projektu a porovná ho s definovanými pravidly a nastavením v konfiguračním souboru.

Výstupem příkazu jsou zprávy o nalezených chybách, varováních nebo nedodržení kódových standardů v projektu „tc“. Tyto zprávy jsou zobrazeny v příkazovém řádku a umožňují programátorům identifikovat a opravit problémy v kódu.

Kontrola kódu pomocí Lintu byla integrována do procesu vývoje a byla také součástí pipeline před samotnou kompilací a vytvořením výsledného buildu. Tím bylo zajištěno, že před každým sestavením aplikace byla vždy bez výjimky provedena striktní kontrola kódu a potenciální chyby byly odhaleny a opraveny. Tento přístup zvýšil kvalitu kódu a zabezpečil dodržování stanovených standardů v rámci projektu.

## 4.4 OpenAPI

V projektu byl využit OpenAPI Generator. Jedná se o rozšířený a populární nástroj pro generování potřebných závislostí pro komunikaci s databází pomocí dokumentace API. Proces generování byl vždy spuštěn pomocí skriptu, který stáhl dokumentaci z platformy Swagger a vygeneroval soubory přímo v projektu. Díky tomuto nástroji byla usnadněna a zrychlena tvorba potřebných artefaktů na základě poskytnuté dokumentace.




Obrázek 21: Struktura OpenAPI artefaktů.

Na obrázku výše můžeme vidět již vygenerovanou strukturu z OpenAPI. Složka model obsahuje veškeré databázově definované modely, které obsahují patřičně definované atributy a vlastnosti. Tyto modely se poté dají využívat v celém projektu a díky tomu lze docílit větší přehlednosti a nižší chybovosti v komunikaci se serverem. Složka api poté obsahuje především samotné služby, které implementují jednotlivé sekce s definovanými rozhraními.

## 4.5 Slovník

Pro implementaci mezinárodní lokalizace bylo potřeba nejprve nastavit službu v globální konfiguraci projektu a vytvořit lokalizační soubory pro jednotlivé jazyky. V konfiguraci bylo potřeba nastavit výchozí jazyk a příslušný soubor jenž jej překládá. Na obrázku níže je vidět toto nezbytné nastavení.

```
translate.addLangs( langs: ['cz']);
translate.setDefaultLang('cz');
translate.use( lang: 'cz');
```

Obrázek 22: Globální výchozí nastavení překladové služby.

Jedná se o instanci translate služby TranslateService, kterou poskytuje přímo framework Angular. Tuto službu bylo poté nutné pomocí dependency injection (DI) vkládat do komponent, které jí potřebovali využívat, tedy obsahovaly řetězce, které byly potřeba překládat.

```
this.dialogData = {
    title: this.translateService.instant( key: 'section.users.remove_avatar'),
    message: this.translateService.instant( key: 'section.users.really_remove_avatar', this.contact),
    icon: {
        show: true,
        name: 'heroicons_outline:exclamation'
    },
    actions: {
        confirm: {
            show: true,
            label: this.translateService.instant( key: 'section.users.delete'),
            color: 'warn'
        },
        cancel: {
            show: true,
            label: this.translateService.instant( key: 'section.users.cancel')
        }
    }
};
```

Obrázek 23: Použití služby translateService pro překlady.

Na obrázku výše můžeme vidět službu translateService, která využívá překladové klíče a dle nastavené konfigurace je poté zamění za požadované zprávy pro uživatele. Tyto klíče musí být definované v překladových souborech, viz následující obrázek.

```
"section.users.remove_avatar": "Odstranit avatar",
"section.users.really_remove_avatar": "Opravdu chcete odstranit avatar uživatele {{firstname}} {{lastname}}?",
"section.users.delete": "Odstranit",
"section.users.cancel": "Zrušit",
```

Obrázek 24: Ukázka překladového souboru.

Takto bylo nutné službu využívat všude, kde se počítá s překlady daných řetězců. Jelikož se se službou počítalo již od návrhu, byla tedy nastavena a implementována již od samého začátku vývoje.

## 4.6 Routes

V aplikaci byl vytvořen globální soubor rout, ve kterém byly deklarovány všechny routy pro jednotlivé moduly „první úrovně“. Ukázka této definice je vidět na obrázku níže.

```
{path: 'users',     loadChildren: () => import('libs/vipbc/src/lib/users/users.module').then(m => m.UsersModule),
{path: 'dashboard', loadChildren: () => import('libs/vipbc/src/lib/dashboard/project/project.module').then(m => m.ProjectModule),
{path: 'support',   loadChildren: () => import('libs/vipbc/src/lib/pages/support/support.module').then(m => m.SupportModule),
{path: 'changelog', loadChildren: () => import('libs/vipbc/src/lib/pages/changelog/changelog.module').then(m => m.ChangelogModule),
```

Obrázek 25: Ukázka definice hlavních tras v projektu.

Každý modul má poté svůj vlastní soubor rout, který definuje specifické cesty pro tento modul. Tyto cesty mohou zahrnovat podstránky, komponenty nebo další moduly, které jsou specifické pro daný modul. Tento přístup poskytuje jednoduchý a modulární způsob správy rout, který usnadňuje rozšiřování a údržbu aplikace. Například po zadání cesty users, dojde k přesměrování na modul UserModule. V rámci toho jsou definované lokální routy pro tento modul, jak můžeme vidět na následujícím obrázku. Můžeme zde vidět základní cestu směřující na seznam uživatelů a dále pod cesty, sloužící pro vytvoření nového uživatele, a jeho detail.

```
export const usersRoutes: Route[] = [
    {
        path     : '',
        component: UsersComponent,
        children : [
            {
                path     : '',
                component: UsersListComponent,
                children : [
                    {
                        path         : 'new',
                        component    : UsersDetailsComponent,
                        canDeactivate: [CanDeactivateContactsDetails]
                    },
                    {
                        path         : ':id',
                        component    : UsersDetailsComponent,
                        canDeactivate: [CanDeactivateContactsDetails]

                    }
                ],
            }
        ]
    }
];
```

Obrázek 26: Soubor tras v modulu users.

## 4.7 Interceptory

V rámci celého projektu jsem využíval tzv. interceptory v Angularu/TypeScriptu. Interceptory jsou důležitou součástí aplikační logiky, která umožňuje zásahy do každého HTTP požadavku nebo odpovědi. Interceptory jsou vytvořeny jako samostatné třídy a slouží k vykonávání specifických úkolů před odesláním požadavku na server nebo po obdržení odpovědi.

Interceptory jsou velmi užitečné, protože poskytují jednotné místo pro zpracování opakujících se úkolů, které se týkají HTTP komunikace. V mém projektu jsem je využíval(a) například k přesměrování uživatele na stránku s chybou 404 v případě neplatného požadavku nebo neexistujícího obsahu.

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    return next.handle(req).pipe(
        tap(
            next: () : null  => null,
            error: error => this.handleError(req, error)
        ));
}
```

Obrázek 27: Implementace Interceptoru.

Na obrázku výše je vidět implementace metody intercept(), která se stará o zachycení jakékoliv chyby v http dotazu či odpovědi. Tato chyba je poté dále zpracována v metodě handleError(), definované dále v dané třídě.

Pro správnou funkčnost interceptorů je nutné je také začlenit do modulu, který je používá. Byl tedy vytvořen samostatný modul, ve kterém je zapouzdřena funkcionalita interceptorů. V rámci tohoto modulu se dále musí importovat potřebné závislosti, zejména HttpClient a samotné třídy interceptorů.

```
@NgModule({
    imports: [
        HttpClientModule
    ],
    providers: [
        ErrorService,
        {
            provide: HTTP_INTERCEPTORS,
            useClass: ErrorInterceptor,
            multi: true
        }
    ]
})
```

Obrázek 28: Modul obsahující definici interceptoru.

V modulu se definuje provide pro daný interceptor, který je potřeba využít, a to pomocí příslušného tokenu, kterým bude interceptor identifikován. Tímto způsobem Angular ví, který interceptor má použít pro daný typ zásahu. Poté je nutné přidat tyto providery do sekce providers v rámci modulu.

Nakonec je potřeba začlenit modul s interceptory do aplikace. Toho se dosáhne tak, že se tento modul přidá do pole imports v rámci modulu hlavní aplikace, který definuje celkovou strukturu aplikace.

Po správném začlenění modulu s interceptory je Angular schopen automaticky provádět zásahy v rámci každého HTTP požadavku nebo odpovědi. Interceptory jsou spouštěny v předdefinovaném pořadí, které lze nastavit podle potřeby.

### 4.7.1 Odhlášení

Během procesu odhlášení uživatele bylo potřeba zajistit bezpečné ukončení relace uživatele a odstranění příslušných dat z paměti. Akci odhlášení může vyvolat uživatel stisknutím příslušného ovládacího prvku, nebo může být tato akce vyvolána z jiných důvodů automaticky. Mezi ty může patřit například invalidace relace po zablokování účtu. V tu chvíli, pokud uživatel provolá jakékoliv data ze serveru, dojde k jeho okamžitému odhlášení a informování o dané situaci.

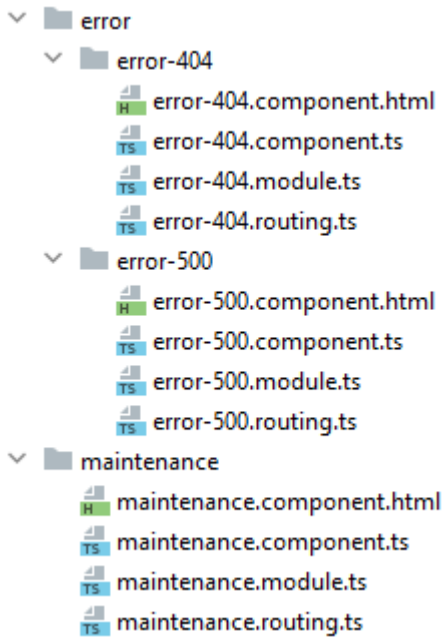
Jelikož aplikace implementuje bezpečnostní opatření pomocí JSON Web Tokenu, bylo tedy potřeba v samotném procesu odhlášení odebrat uložené přihlašovací tokeny z lokálního úložiště, kde jsou během relace uloženy. To zajistí, že při dalším přístupu, bude vyžadováno přihlášení a následně dojde k opětovnému přiřazení nových tokenů. Dále byla hodnota _authenticated nastavena na false, což signalizuje neplatnost aktuálního přihlášení uživatele. Tím je zajištěno, že uživatel bude v dalších krocích přesměrován na přihlašovací stránku a nebude mít přístup k chráněným částem aplikace. V neposlední řadě bylo potřeba odebrat dočasně uložené informace o přihlášeném uživateli, které slouží jako užitečná mezipaměť při platné relaci. Tato implementace je vidět na následujícím obrázku.

```
signOut(): Observable<any> {

    localStorage.removeItem( key: 'accessToken');
    localStorage.removeItem( key: 'refreshToken');

    this._authenticated = false;
    this.clearLoggedUserCache();

    return of( value: true);
}
```

Obrázek 29: Metoda pro odhlášení uživatele.

## 4.8 Chybové stránky

V rámci projektu byly vytvořeny speciální stránky pro zpracování chyb, které zahrnují stránky s chybovými kódy jako je 404, 500 a stránky pro údržbu. Jejich vytvořená struktura je zachycena na následujícím obrázku.




Obrázek 30: Struktura modulů chybových stránek.

Tyto stránky jsou přístupné mimo jiné také prostřednictvím tzv. wildcard routování, které zachytává všechny neznámé nebo neexistující cesty a směruje je na odpovídající chybovou stránku. Například, pokud uživatel zadá neplatnou adresu URL nebo odkaz na neexistující stránku, je přesměrován na stránku 404, která signalizuje, že hledaná stránka nebyla nalezena.

Na obrázku níže je vidět již implementovanou část kódu, která je zodpovědná za přesměrování na chybovou stránku, po zavolání neplatné url adresy.

```
// maintenance, error & Catch all
{path: 'maintenance', loadChildren:
{path: '500-internal-server-error', loadChildren:
{path: '404-not-found', pathMatch: 'full', loadChildren:
{path: '**', redirectTo: '404-not-found',
```

Obrázek 31: Přímé routování na chybové stránky.

Pro implementaci tohoto chování byly také využity interceptory, které zachytávají chyby v rámci HTTP komunikace. Pokud například dojde k chybě při načítání dat ze serveru, interceptor může přesměrovat uživatele na stránku s chybovým kódem 500. Tímto způsobem je zajištěna správná reakce na chybové situace a uživatel je informován o výskytu problému.

Díky těmto chybovým stránkám a jejich integrovanému routování je zajištěno, že uživatelé budou přiměřeně informováni a navedeni, pokud se vyskytnou chyby nebo neplatné požadavky. Tím se zvyšuje uživatelská přívětivost a poskytuje se lepší uživatelský zážitek v rámci aplikace.

## 4.9 Uživatelský modul

Modul pro správu uživatelů je základní uživatelský modul aplikace. Tento modul je navržen speciálně pro uživatele s administrátorskými právy, aby mohli spravovat uživatelské účty v systému. Tento modul zajišťuje efektivní a bezpečné řízení uživatelských účtů a poskytuje funkcionalitu pro jejich správu.

Struktura modulu pro správu uživatelů je pečlivě navržena a zahrnuje několik důležitých komponent a služeb. Mezi tyto komponenty patří například komponenta pro zobrazení seznamu uživatelů, formulář pro vytváření a úpravu uživatelů, a také komponenta pro detaily jednotlivých uživatelských účtů. Každá komponenta je odpovědná za specifickou část správy uživatelů a spolupracuje s odpovídajícími službami. Detail jednotlivého uživatele dále obsahuje další karty, které umožnují další funkcionality, jako je změna hesla, role, či nahlédnutí do historie blokace uživatele.

Při inicializaci modulu bylo nutné načíst seznam všech uživatelů ze serveru. Při těchto operacích se často využívají Observables. Observables jsou klíčovou součástí programování v Angularu, protože nabízejí silný a flexibilní způsob práce s asynchronními událostmi a daty. Observables poskytují mechanismus pro asynchronní získávání, transformaci a zpracování dat.

V implementaci bylo použito rozhraní users$, které je observable poskytovaný službou daného modulu. Tímto rozhraním bylo možné naslouchat asynchronním událostem a získávat data ze serveru. Příslušná data byla získána prostřednictvím odběru na observablu pomocí metody subscribe. Tuto část logiky zachycuje obrázek níže.

```
this._usersService.users$
    .pipe(takeUntil(this._unsubscribeAll))
    .subscribe( next: (contacts : UserResource[] | null ) : void  => {

        // Update the counts
        this.usersCount = contacts?.length;
        this.users = contacts;
        // Mark for check
        this._changeDetectorRef.markForCheck();
    });
```

Obrázek 32: Načtení uživatelů ze serveru.

Pomocí operátoru takeUntil bylo dále zajištěno správné ukončení odběru dat, například při zrušení komponenty. Při provádění aktualizace uživatelů, jsou vždy prováděny další operace, jako například aktualizace proměnné usersCount, která uchovává aktuální počet uživatelů. Pro zajištění aktualizace uživatelského rozhraní byla na konci také použita metoda, která vyvolá manuální spuštění detekce změn pro danou komponentu.

Použití observables v tomto případě umožnilo efektivní a asynchronní získávání dat ze serveru. Díky observables byla zajištěna správná synchronizace dat a jejich aktualizace v uživatelském rozhraní. Tímto způsobem bylo dosaženo interaktivního a dynamického chování aplikace, které reaguje na změny v datech ze serveru.

### 4.9.1 Detail uživatele

Pro přehledné zobrazení všech funkcionalit v rámci detailu uživatele, byla zvolena komponenta mat-drawer. Ta poskytuje sadu nástrojů a možností, jak vytvořit například vysouvací panel, který se hodí přesně pro tyto účely.

V rámci implementace komponenty mat-drawer bylo vytvořeno několik komponent pro zobrazení obsahu v závislosti na vybraném panelu. Tento přístup umožňuje dynamicky zobrazovat a měnit komponenty v rámci prvku mat-drawer na základě výběru uživatele.

```
<div class="mt-8">
    <ng-container [ngSwitch]="selectedPanel">
        <ng-container *ngSwitchCase="'info'">
            <app-contact-detail [(contact)]="contact"></app-contact-detail>
        </ng-container>
        <ng-container *ngSwitchCase="'role'">
            <app-contact-role [(contact)]="contact"></app-contact-role>
        </ng-container>
        <ng-container *ngSwitchCase="'password'">
            <contact-password [contact]="contact"></contact-password>
        </ng-container>
        <ng-container *ngSwitchCase="'block-history'">
            <app-contact-block-history [userId]="contact?.id"></app-contact-block-history>
        </ng-container>
    </ng-container>
</div>
```

Obrázek 33: Výběr mezi komponenty v detailu uživatele.

Na obrázku výše je možné vidět část HTML šablony, ve které byl vytvořen kontejner <ng-container> s použitím direktivy ngSwitch, která vyhodnocuje hodnotu selectedPanel. Na základě hodnoty selectedPanel je určen aktuálně vybraný panel v rámci mat-drawer.

V rámci jednotlivých ng-container se nachází další ng-container s direktivou ngSwitchCase, která slouží k přiřazení konkrétního obsahu k danému panelu. V rámci ngSwitchCase byly poté umístěny jednotlivé komponenty, které budou zobrazeny v rámci mat-drawer v závislosti na vybraném panelu.

Například, v případě, že je vybrán panel „info“, bude zobrazena komponenta app-contact-detail, která zobrazí detaily uživatele. Pokud je vybrán panel 'role', bude zobrazena komponenta app-contact-role, která umožňuje správu rolí uživatele.

Tímto způsobem byla vytvořena flexibilní a dynamická struktura pro zobrazení různých komponent v rámci mat-drawer na základě výběru uživatele. Uživatelé mohou jednoduše přepínat mezi různými panely a pracovat s odpovídajícím obsahem. Tím se zlepšuje uživatelská interakce a umožňuje efektivní správu a zobrazení informací v rámci projektu. Tento styl detailu je poté také využit v dalších modulech s podobnou funkcionalitou.

Při implementaci komponent v rámci mat-drawer byla také využita funkce „two-way“ data binding pro propojení komponenty v mat-drawer s rodičovskou komponentou. Jeden z příkladů je použití [(contact)]="contact", kde contact je objekt uživatele, který slouží k předávání dat mezi rodičovskou komponentou a komponentami v mat-drawer. Tímto způsobem je zajištěno, že změny provedené v komponentách uvnitř mat-drawer jsou propagovány zpět do rodičovské komponenty.

Podobně, další komponenty v rámci mat-drawer mohou využívat dvousměrný binding pro předávání a aktualizaci dat s rodičovskou komponentou. To umožňuje synchronizovat data mezi komponentami a správně zobrazovat aktuální informace v rámci mat-drawer.

## 4.10 Validace hesel

V projektu bylo na několika místech potřeba ošetřit validaci hesel. Tato validace musela zajišťovat, že uživatelé zadávají hesla s odpovídajícími požadavky. Tato validace je důležitá z hlediska bezpečnosti a zajištění správného nastavení hesel. Jelikož se jedná o kritickou část, která se týká bezpečnosti aplikace a samotných účtů uživatelů, byly vytvořeny centrální metody, která se využívá napříč celou aplikací.

Nejprve bylo potřeba vytvořit kontrolní mechanismus pro zadávání nového hesla. V tomto případě musí uživatel zadat heslo vždy dvakrát, aby se snížila šance překlepu uživatele. Tyto dvě hesla se validují na straně aplikace a případě uživateli oznámí, že se dopustil chyby. Na obrázku níže je možné vidět FormBuilder, který obsahuje dvě pole pro hesla. Každé pole je samostatně validováno, ovšem dále formulář obsahuje validátor, který zajišťuje správnost obou hesel zároveň.

```
this.passForm = this._formBuilder.group( controls: {
    password: ['', [Validators.required, this._authService.validatePassword.bind(this._authService)]],
    passwordConfirmation: ['', [Validators.required]]
},  options: {validator: this._authService.checkPasswords.bind(this._authService)});
```

Obrázek 34: Formulář s validací hesel.

Na obrázku výše je také vidět využití validátoru na první pole pro zadávaní heslo. To se stará o komplexnost zadávaného hesla. Opět se tato validace neprovádí v dané komponentě, ale je využita služba, která je přístupná a využívá se z více míst v aplikaci. Tento přístup umožnuje snadnou a rychlou kontrolu nad validací napříč celou aplikací, která přispívá k přehlednosti a bezpečnosti aplikace.

Heslo musí splňovat několik kritérií, aby bylo považováno za dostatečně silné. V aplikaci se validace implementovala, aby vyžadovala, aby heslo obsahovalo alespoň jedno číslo, jedno velké písmeno, jedno malé písmeno a mělo minimální délku osmi znaků. Pokud heslo neodpovídá těmto požadavkům, je uživateli zobrazena chybová zpráva s vysvětlením, jaké kritéria heslo nesplňuje.

Metoda nejprve získá hodnotu zadaného hesla pomocí control.value. Poté postupně kontroluje, zda heslo obsahuje požadované prvky. Pokud se některé kritérium nevyhovujícího hesla

identifikuje, přidává příslušnou chybovou zprávu do proměnné errorMessage. Chybové zprávy jsou lokalizovány pomocí translateService, což dále umožňuje jejich jednoduchou lokalizaci do různých jazyků.

Po kontrole všech kritérií se provede kontrola proměnné error, která indikuje, zda se vyskytly nějaké chyby. Pokud je error nastaven na true, vytvoří se chybová zpráva, která obsahuje všechny nevyhovující kritéria. Poté dojde k vytvoření objektu s chybovou zprávou invalidPassword, který obsahuje tuto zprávu. V případě, že heslo splňuje všechna kritéria, metoda vrátí hodnotu null, což znamená, že heslo je platné. Tato metoda přispívá k lepšímu uživatelskému rozhraní a poskytuje uživatelům konkrétní zpětnou vazbu ohledně jejich hesla, což zvyšuje uživatelskou spokojenost a zajišťuje bezpečnost systému. Implementace této metody je zobrazena na následujícím obrázku.

```
validatePassword(control: AbstractControl): { [key: string]: any } | null {
    const value: string = control.value;
    const hasNumber : boolean  = /\d/.test(value);
    const hasUppercase : boolean  = /[A-Z]/.test(value);
    const hasLowercase : boolean  = /[a-z]/.test(value);
    const hasMinimumLength : boolean | "" = value && value.length >= 8;
    let errorMessage = this.translateService.instant( key: 'section.pass.mustContain');
    let error : boolean  = false;
    if (!hasNumber) {
        error = true;
        errorMessage += this.translateService.instant( key: 'section.pass.number');
    }

    if (!hasUppercase) {
        error = true;
        errorMessage += this.translateService.instant( key: 'section.pass.uppercase');
    }

    if (!hasLowercase) {
        error = true;
        errorMessage += this.translateService.instant( key: 'section.pass.lowercase');
    }

    if (!hasMinimumLength) {
        error = true;
        errorMessage += this.translateService.instant( key: 'section.pass.minimumLength');
    }

    if (error) {
        errorMessage = errorMessage.slice(0, -2);
        errorMessage+= '.';
        return { invalidPassword: errorMessage };
    }

    return null;
}
```

Obrázek 35: Metoda pro zajištění a kontrolu komplexnosti hesla.

## 4.11 Validace formulářů

Validace formulářů je klíčovou součástí této aplikace, která umožňuje zabezpečit a zajistit správnost vstupních dat od uživatelů. V rámci implementace byl kladen důraz na důkladnou a komplexní validaci formulářů, aby byla zajištěna správnost a konzistence dat v aplikaci.

```
<mat-form-field class="w-full">
    <mat-label>{{"auth.signup.email" | translate}}</mat-label>
    <input
        id="user-email"
        matInput
        [formControlName]="'userEmail'">
    <mat-error *ngIf="signUpForm.get('userEmail').hasError( errorCode: 'required')">
        {{"auth.signup.fill_email" | translate}}
    </mat-error>
    <mat-error *ngIf="signUpForm.get('userEmail').hasError( errorCode: 'email')">
        {{"auth.signup.fill_valid_email" | translate}}
    </mat-error>
</mat-form-field>
```

Obrázek 36: Formulářový prvek pro email.

Ve vzorovém kódu na obrázku výše je vidět formulářový prvek pro zadání emailové adresy. Ten je propojen s odpovídajícím kontrolním prvkem v rámci formulářové skupiny. Tímto propojením je umožněno provádět různé typy validací na základě zadaných pravidel.

Jsou využívané vestavěné validační funkce, které poskytuje přímo Angular, jako je Validators.required, která zajišťuje, že pole je povinné vyplnit, a Validators.email, která ověřuje, zda má zadaná hodnota platný tvar emailové adresy. Tyto validační funkce jsou aplikovány na příslušný formControl v rámci formulářové skupiny.

Chybové zprávy jsou poté zobrazovány v elementu <mat-error> v závislosti na výsledku validace. Například, pokud pole emailu nebylo vyplněno, zobrazí se zpráva, která je přiřazena překladovému klíči „auth.signup.fill_email“. To uživatelům jasně indikuje, jaká chyba se ve vyplněných datech vyskytuje, a jak jí opravit.

Při odesílání formuláře je zajištěno a kontroluje se, že formulář je v platném stavu a byl upraven. To znamená, formulář může být odeslán až v případě, že veškerá data jsou v souladu s formulářovými pravidly. Tato skutečnost se dá kontrolovat pomocí příznaku invalid u objektu formuláře, jak je vidět na následujícím obrázku.

```
[disabled]="signUpForm.invalid || signUpForm.disabled"
(click)="signUp(submitBtn)">
```

Obrázek 37: Prvek pro odesílání formuláře.

Do té doby to není uživateli odeslání umožněno, aby se snížilo riziko odeslání chybných či neplatných dat. Jedná se o jednu z kontrol, kterou je možné tímto způsobem v této části aplikace ošetřit. Pokud jsou tedy data v pořádku, formulář je umožněno odeslat a dojde k zavolání metody signUp(), která se stará o zpracování zadaných dat.

## 4.12 Znovupoužitelné komponenty

V aplikaci byly využívané znovupoužitelné komponenty, které vypadají jako standartní komponenty a je poté možné využívat je dále na více místech a hrály důležitou roli při vývoji škálovatelné a snadno udržovatelné aplikace. Ve složce určené pro tyto komponenty byly deklarovány a poté byly použity na různých místech projektu. Tím bylo dosaženo znovupoužitelnosti kódu, zlepšené konzistence a efektivity vývoje.

Jednou z výhod opakovaně použitelných komponent je snížení duplicity kódu. Namísto opakovaného zápisu podobného kódu pro podobnou funkčnost v různých částech projektu je možné tuto funkčnost zapsat do opakovaně použitelné komponenty. To podporuje čistý kód a usnadňuje údržbu a aktualizaci aplikace.

Opakovaně použitelné komponenty také zvyšují přehlednost kódu. Díky tomu, že jsou komponenty umístěny ve zvláštní složce, je snadné je najít a znovu použít na různých místech v projektu. To zjednodušuje přidávání nových funkcionalit a úpravy existujícího kódu.

### 4.12.1 Našeptávač adres

Mezi časté znovupoužitelné části kódu se řadí například části formulářů, které se využívají na více místech. V této aplikaci se můžeme na několika místech setkat s vyplňováním adresy. Styl vyplňování a formátu adresy je dobré udržovat napříč celým systémem jednotný, což byl hlavní důvod pro vytvoření znovupoužitelné komponenty. Základem takové komponenty je možnost vstupních a výstupních parametrů.

Na obrázku níže můžeme vidět vlastnosti znovupoužitelné komponenty. Jedná se zprvu o definici vstupních parametrů, které bude daná znovupoužitelná komponenta využívat pro své funkce. Zde můžeme vidět vstupní adresu pro výchozí vyplnění formuláře, souřadnice pro správné zobrazení mapy a nakonec příznak, zda je adresa povinná.

```
@Input() fullAddress?: string;
@Input() inLatitude?: string;
@Input() inLongitude?: string;
@Input() mandatory?: boolean;
@Output() addressEmitted: EventEmitter<AddressObject> = new EventEmitter<AddressObject>();
```

Obrázek 38: Vstupní a výstupní parametry znovupoužitelné komponenty.

Následuje direktiva @Output(), která značí výstupní parametr. Jedná se o takzvaný EventEmmiter, díky kterému můžeme poslat aktuální hodnotu chtěného výstupu vnější komponentě.

Daná komponenta se poté deklaruje v jakémkoliv modulu, v rámci kterého jí chceme využívat a díky definovanému selektoru, který je zde „app-google-geo“, jí využijeme v naší šabloně, jak je ukázáno na obrázku níže. V rámci vložené komponenty se definují vstupní parametry a také se zapracuje již dříve zmíněný EventEmmiter, díky kterému získáme aktuální adresu z vnořené komponenty.

```
<app-google-geo
    [fullAddress]="contact?.addressFull"
    [inLatitude]="this.contact?.gps?.[0]?.latitude"
    [inLongitude]="this.contact?.gps?.[0]?.longitude"
    (addressEmitted)="addressObtained($event)">
</app-google-geo>
```

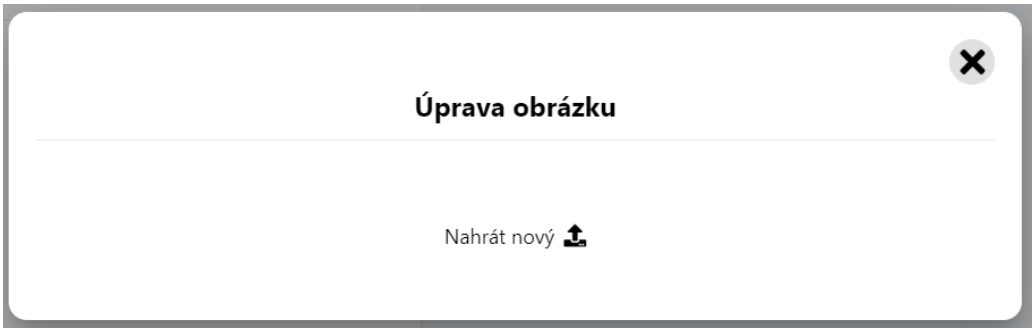
Obrázek 39: Použití vlastní sdílené komponenty.

Takto se komponenta může dále využívat na všech potřebných místech, kde je potřeba pracovat s adresou. A další velmi důležitá výhoda, která plyne z takto sdílených komponent je skutečnost, že pokud bude potřeba v budoucnu upravit styl nebo způsob zadávání, stačí jej upravit pouze v rámci sdílené komponenty a pokud se nemění vstupně výstupní parametry, není potřeba upravovat komponenty které jej využívají.

### 4.12.2 Výběr obrázků

Další funkce, která se v podobných systémech často využívá, a zde tomu není výjimkou, je nahrávání vlastního obrázku. Ten se dá využít v mnoha případech. Například jako avatar, či ilustrace k různým produktům.

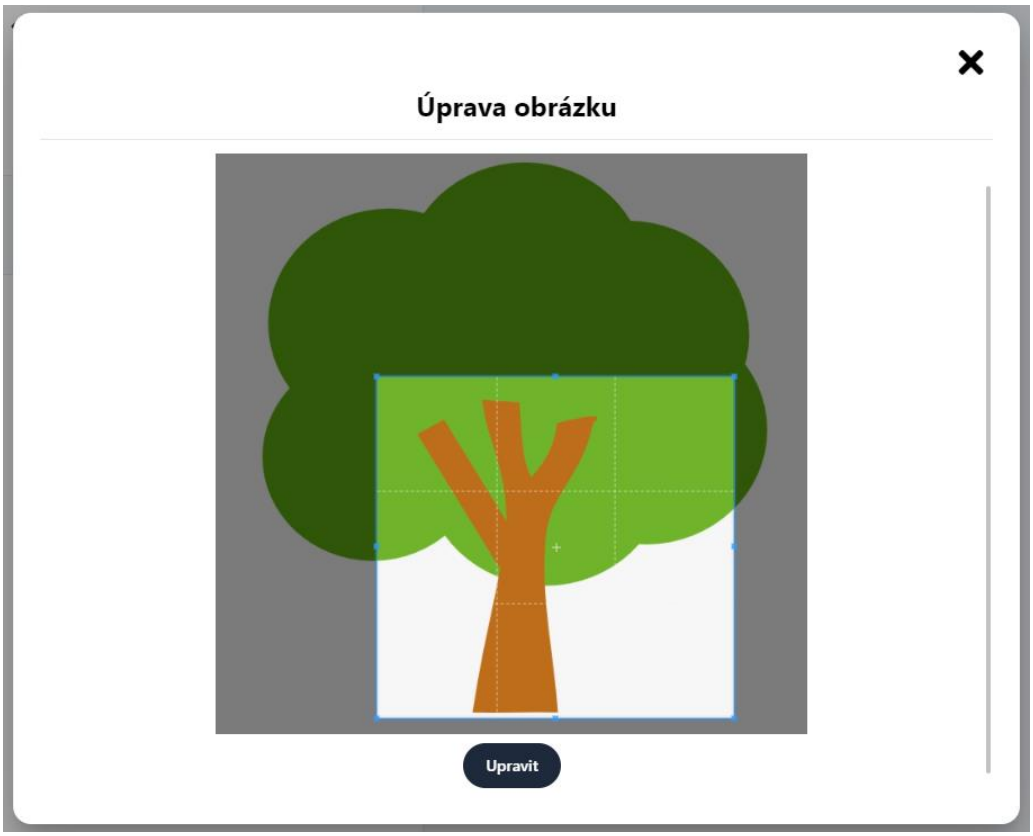
Pro tuto funkci byl vytvořen znovupoužitelný dialog. Jedná se opět o samostatnou nezávislou komponentu, která se dá využívat v rámci projektu na více místech, popřípadě se dá i snadno implementovat napříč projekty. Dialog je navržen tak, aby mohl být snadno začleněn do různých částí aplikace, kde je potřeba manipulace s obrázky. Jeho využití je mnohostranné a umožňuje uživatelům aplikace upravovat obrázky dle svých preferencí. Komponenta je navržena s ohledem na uživatelskou přívětivost a poskytuje jednoduché a intuitivní ovládání. Po vyvolání dialogu se ihned zobrazí pole pro možnost nahrání nového obrázku, voz obrázek níže.




Obrázek 40: Dialog pro nahrání obrázku.

Podporované obrázky pro nahrání a úpravu jsou aktuálně s příponou .jpeg a .png. Ostatní formáty jsou skryté z výběru, a pokud se uživatel i přesto pokusí takovýto nepodporovaný obrázek nahrát, je informován, že tento formát není podporován a obrázek není přijmut.

Pokud se nahraje správný formát obrázku, uživatel má poté možnost v náhledu obrázek snadno oříznout a upravit, dle svých preferencí. Viz následující obrázek.




Obrázek 41: Dialog pro úpravy obrázku.

V neposlední řadě, po potvrzení úpravy, je uživateli zobrazen náhled upraveného obrázku a vyzvání k uložení, popřípadě k vrácení se k dalším úpravám.

Celý dialog má svojí sekci v překladovém souboru, a při implementaci v jiném projektu, je nutné počítat s tím, že se využívá překladová služba a komponenta tedy obsahuje překladové klíče. V dialogu se pro samotné ořezávání obrázku využívá velmi rozšířená a podporovaná knihovna angular-cropperjs, která obsahuje komponentu angular-cropper. Na obrázku níže je vidět implementace tohoto prvku v samotném kódu aplikace.

```
<angular-cropper #angularCropper [cropperOptions]="config" *ngIf="!result"
                 (export)="resultImageFun($event)" (ready)="checkStatus($event)"
                 [imageUrl]="imageSrc">
</angular-cropper>
```

Obrázek 42: Implementace ořezu nahraného obrázku.

# ZÁVĚR

Práce se zabývala návrhem a vývojem dále rozšiřitelné funkční kostry aplikace pro backoffice systém firmy. V první části práce je provedena analýza stávajícího trhu s podobnými systémy. Dále autor představuje aktuální technologie, které přispívají k efektivnímu vývoji podobných aplikací.

Následuje důležitá část práce, která byla věnována samotnému návrhu a realizaci daného systému. Nejprve bylo potřeba analyzovat požadavky a navrhnout samotnou architekturu systému. Pro dílčí části systému byly využity UML modely pro přehlednou vizualizaci procesů, které byly potřeba implementovat.

Realizované řešení bylo externě otestováno, což je velmi důležitý krok při vývoji systému pro produkční využití. Je důležité si uvědomit, že systémy dále vyhotovené na základě tohoto budou využívat firmy a uživatelé, kteří nemusí mít se systémy tohoto typu mnoho zkušeností. Je tedy nutné, aby byly s tímto vědomím testovány. To je něco, co mi opakovaně dělalo problémy a často bylo potřeba určité prvky předělávat, aby byly vhodné pro koncové uživatele.

Celý projekt byl od samého začátku kladně podporován v rámci firmy, ve které jsem ho realizoval, jelikož se jedná o veliký přínos pro tvorbu dalších projektů a usnadnění jejich vývoj. Obecně byl tento projekt velmi přínosný a aplikace která díky tomu vznikla se již využívá pro vývoj několika produkčních systémů pro různé společnosti a zákazníky. Veliká výhoda spočívá v tom, že není nutné vždy začínat od prázdného projektu, ale je možné využít tuto aplikaci a dle potřeb zákazníka jí snadno začít upravovat, aby co nejlépe vyhovovala jeho požadavkům.

Tímto ovšem projekt samozřejmě nekončí. Jelikož se jedná o obecné a univerzální řešení, je možné do něj přidávat další a další funkce. Tyto funkce jsou vždy pečlivě vybírány dle analýzy trhu. Pokud se zjistí, že se v několika projektech dodělávají podobné funkcionality, jsou poté zváženy a pro budoucí usnadnění, popřípadě doplněny do tohoto projektu. Ten se tedy postupně více rozšiřuje, jelikož zefektivňuje vývoj všech dalších projektů firmy, ve které pracuji.

POUŽITÁ LITERATURA


[1] *MySmartplace* [online]. c2023 [cit. 2023-01-15]. Dostupné z: https://www.mysmartplace.cz/cs

[2] *Macros* [online]. c2011 - 2023 [cit. 2023-01-15]. Dostupné z: https://www.macros.cz/

[3] *Iinterchange Systems* [online]. November 30, 2021 [cit. 2023-01-24]. Dostupné z: https://www.iinterchange.com/importance-third-party-software-integrations/

[4] STRELEC, Ing. Michal. Výběr vhodného řešení pro informační systém. *Strelec.pro* [online]. Praha 6, c2023 [cit. 2023-01-24]. Dostupné z: https://www.strelec.pro/napsal/webova-aplikace-vs-desktopova-aplikace

[5] VANDERKAM, Dan. Effective TypeScript. In: *Effective TypeScript: 62 Specific Ways to Improve Your TypeScript*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 2020, s. 1-6. ISBN 978-1-492-05374-3.

[6] CINCOVIĆ, Jelica a Marija PUNT. *Comparison: Angular vs. React vs. Vue.: Which framework is the best choice?*. Belgrade, Serbia, c2020. University of Belgrade, School of Electrical Engineering.

[7] VYAS, Rishi. Comparative Analysis on Front-End Frameworks for Web Applications. In: *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*. 2022. ISSN 2321-9653. Dostupné z: doi:https://doi.org/10.22214/ijraset.2022.45260

[8] Lifecycle hooks. *Angular v2 Archive* [online]. Google, c2010-2017 [cit. 2023-02-08]. Dostupné z: https://v2.angular.io/docs/ts/latest/guide/lifecycle-hooks.html

[9] E. PEYROTT, Sebastián. *The JWT Handbook* [online]. 0.14.1. Auth0, 2018 [cit. 2023-02-08]. Dostupné z: https://auth0.com/resources/ebooks/jwt-handbook

[10] MAURYA, Richa a Keerthi NAMBIAR. Application of Restful APIs in IOT: A Review. In: *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*. 2022. ISSN 2321-9653. Dostupné z: doi:https://doi.org/10.22214/ijraset.2021.33013

[11] RICHARDSON, Leonard, Mike AMUNDSEN a Sam RUBY. *RESTful Web APIs*. O'Reilly Media, 2013. ISBN 9781449359737.

[12] BUCKLER, Craig. What Is a REST API?. *SitePoint* [online]. c2000–2023, August 24, 2022 [cit. 2023-02-08]. Dostupné z: https://www.sitepoint.com/rest-api/

[13] TSITOARA, Mariot. *Beginning Git and GitHub*. Antananarivo, Madagascar: Apress, c2020. ISBN 978-1-4842-5312-0.

[14] CHACON, Scott a Ben STRAUB. *Pro Git*. 2nd ed. Apress, 2014. ISBN 1484200772.

[15] HUMBLE, Jez a David FARLEY. Continuous delivery: reliable software releases through build, test, and deployment automation. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978–0–321–60191–9.

[16] IBM CLOUD EDUCATION. What Are CI/CD and the CI/CD Pipeline?. IBM [online]. New York, c2023, 27 September 2021 [cit. 2023-02-28]. Dostupné z: https://www.ibm.com/cloud/blog/ci-cd-pipeline

[17] DEELEMAN, Pablo. Learning Angular 2. Birmingham: Packt Publishing, 2016. ISBN 978-1-78588-207-4.

[18] SALEH, Sarah. Everything you need to know about Jira. JEXO [online]. c2023, 24 JAN 2022 [cit. 2023-03-01]. Dostupné z: https://jexo.io/blog/beginners-guide-to-jira/