Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Rozšíření systému pro evidenci majetku ve frameworku Angular

Diplomová práce

Autor: Bc. Milan Knop
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Hradec Králové duben 2023

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedených zdrojů.

V Hradci Králové dne 20.4.2023 Bc. Milan Knop

Poděkování:

Děkuji vedoucímu diplomové práce Ing. Pavlu Křížovi, Ph.D. za metodické vedení práce, směrování a cenné rady při zpracování diplomové práce. Také bych chtěl poděkovat své rodině, za velikou trpělivost.

Anotace

Tato diplomová práce je zaměřena na rozšíření systému evidence majetku ve frameworku Angular pro organizaci s vysokou fluktuací aktiv a velkým počtem uživatelů. Aplikační logika systému je rozdělena na stranu klienta, tvořena ve frameworku Angular, a na stranu serveru, tvořenou frameworkem NestJS.

Tvorbě rozšíření předchází představení vlastní aplikace pro evidenci majetku v bakalářské práci, představení frameworku NativeScript. NativeScript je framework pro tvorbu mobilních aplikací, který dovoluje využívat k tvorbě programů přední frameworky pro tvorbu klientských aplikací např. React, Vue, Svelte a Angular. V rámci mobilní aplikace je z čistě osobních preferencí vybrán framework Angular. Vzhledem k typu aplikace, jenž NativeScript produkuje, jsou zmíněné odlišnosti v použití Angularu společně s NativeScriptem.

Praktickou část tvoří rozšíření systému evidence majetku ve frameworku Angular, ve kterém jsou implementovány funkce pro přesun majetku mezi jednotlivými správci majetku a vývoj aplikace pro mobilní zařízení, která umožňuje přesouvat majetek v terénu pomocí skenu QR kódů a NFC štítů, včetně provádění inventurních procesů. Celá aplikace je řešena, tak aby ji bylo možné provozovat i na uzavřené síti bez přístupu k internetu.

Title: **Extending the Feature Assets Management System in Angular framework**

Annotation

This master’s thesis is focused on expanding the asset management system in the Angular framework for an organization with high turnover of assets and a large number of users. The application logic of the system is divided into client-side, created in the Angular framework, and server-side, created in the NestJS framework.

The development of the extension is preceded by the presentation of the author's own application for asset management in a bachelor's thesis, as well as the introduction of the NativeScript framework. NativeScript is a framework for creating mobile applications that allows the use of leading client-side frameworks such as React, Vue, Svelte, and Angular. Within the mobile application, the Angular framework is selected purely based on personal preferences. Due to the type of application produced by NativeScript, there are differences in the use of Angular together with NativeScript.

The practical part consists of expanding the asset management system in the Angular framework, implementing functions for transferring assets between different asset managers, and developing a mobile application that allows for asset transfer in the field through scanning QR codes and NFC tags, as well as performing inventory processes. The entire application is designed to be able to operate on a closed network without internet access.

Obsah

1 Úvod ... 1

2 Analýza ... 3

2.1 Cíl diplomové práce ... 3

2.2 Výchozí stav ... 3

2.2.1 Prezentační vrstva ... 3

2.2.2 Aplikační vrstva ... 3

2.2.3 Datová vrstva ... 4

2.3 Požadavky na rozšíření aplikace ... 4

2.3.1 Přístupnost ... 4

2.3.2 Lokace ... 4

2.3.3 Změna správce majetku ... 4

2.3.4 Identifikace majetku ... 4

2.3.5 Mobilní aplikace ... 5

2.4 Typy mobilních aplikací ... 6

2.4.1 Nativní mobilní aplikace ... 7

2.4.2 Hybridní mobilní aplikace ... 7

2.4.3 Progressive Web Apps (PWA) ... 7

2.4.4 Interpretované mobilní aplikace ... 8

2.5 QR kódy a NFC štítky ... 8

3 Technologie použité pro tvorbu aplikace ... 9

3.1 Angular ... 9

3.2 NestJS ... 10

3.3 NativeScript ... 10

3.3.1 Rozdíly použití Angularu bez a s NativeScriptem ... 11

3.4 RxJS ... 18

3.4.1 Přihlášení k odběru ... 19

3.4.2 Operátory ... 20

3.4.3 Subjekty ... 23

3.5 TypeORM ... 24

3.5.1 Definice vztahů mezi entitami ... 24

4 Návrh a implementace ... 27

4.1 Lokace ... 27

4.1.1 Implementace lokace - Angular ... 27

4.1.2 Implementace lokace – NestJS ... 29

4.2 Převod majetku mezi správci ... 32

4.2.1 Implementace převod majetku - Angular ... 34

4.2.2 Implementace převod majetku – NestJS ... 35

4.3 Identifikace majetku ... 39

4.3.1 Implementace identifikace majetku – Angular ... 39

4.4 Mobilní aplikace 39
4.4.1 Zajištění požadavků aplikace 39
4.4.2 Implementace funkcí mobilní aplikace 42
4.4.3 Testování mobilní aplikace 52
5 Závěr 54
6 Seznam zdrojů 55
6.1 Knižní zdroje 55
6.2 Internetové zdroje 55
7 Přílohy 57
7.1 Příloha č. 1 – elektronická příloha 57

Seznam obrázků

Obrázek 1, Angular logo, zdroj: https://angular.io/ 9

Obrázek 2, NestJS logo, zdroj: https://nestjs.com/ 10

Obrázek 3, NativeScript logo, zdroj: https://docs.nativescript.org/api-reference/ 10

Obrázek 4, Angular NativeScript logo, zdroj: https://medium.com/@tj.vantoll/nativescript-angular-the-future-of-mobile-app-development-d83cfeecfebd 11

Obrázek 5, ukázka flexbox, zdroj: https://docs.nativescript.org/ui-and-styling.html#flexboxlayout 12

Obrázek 6, výsledek flexboxLayoutu vlastní tvorba; 12

Obrázek 7, ukázka zápisu pomocí HTML a CSS, vlastní tvorba 12

Obrázek 8, ukázka použití atributu, zdroj: https://docs.nativescript.org/ui-and-styling.html#flexboxlayout 13

Obrázek 9, ukázka zápisu gridLayout, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout 14

Obrázek 10, gridLayout rowSpan, colSpan, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout 14

Obrázek 11, zobrazení gridLayoutu, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout 15

Obrázek 12, použití * v gridLayoutu, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout 15

Obrázek 13, ukázka * v gridLayoutu: https://docs.nativescript.org/ui-and-styling.html#gridlayout ... 16

Obrázek 14. absoluteLayout, zdroj: https://docs.nativescript.org/ui-and-styling.html#absolutelayout ... 16

Obrázek 15, RxJS logo, zdroj: https://worldvectorlogo.com/logo/rxjs-1 ... 18

Obrázek 16, použití Async pipe - Angular, vlastní tvorba ... 19

Obrázek 17, ukázka metody subscribe, vlastní tvorba ... 20

Obrázek 18, operátor switchMap, vlastní tvorba ... 21

Obrázek 19, přehled subjektů v RxJS, vlastní tvorba ... 23

Obrázek 20. TypeORM logo, zdroj: https://typeorm.io/ ... 24

Obrázek 21, Ukázka @ManyToMany, vlastní tvorba ... 26

Obrázek 22, dvojí využití komponenty, vlastní tvorba ... 28

Obrázek 23, rekurze ve třídě Location, vlastní tvorba ... 28

Obrázek 24, ukázka výsledku dashboardu, vlastní tvorba ... 29

Obrázek 25, TypeORM entita - Location, vlastní tvorba ... 30

Obrázek 26, sekvenční diagram přesunu majetku, vlastní tvorba ... 33

Obrázek 27, žádost o převod, vlastní tvorba ... 34

Obrázek 28, DI - interface, vlastní tvorba ... 35

Obrázek 29, ukázka queryBuilderu, vlastní tvorba ... 37

Obrázek 30, výsledný QR kód ... 39

Obrázek 31, Ugreen USB-C hub, zdroj: https://www.alza.cz/ugreen-usb-c-hub-3x-usb-a-2-0-1x-ethernet-d6126479.htm 40

Obrázek 32, ukázka základního rozhraní mobilní aplikace, vlastní tvorba 42

Obrázek 33, ukázka přihlášení, vlastní tvorba 43

Obrázek 34, uchování tokenu ve službě, vlastní tvorba 44

Obrázek 35, metoda intercept v auth.interceptor.ts 44

Obrázek 36, průběh registrace místnosti, vlastní tvorba 45

Obrázek 37, ukázka přiřazení štítku k lokaci, vlastní tvorba 46

Obrázek 38, ukázka označení naskenovaného majetku, vlastní tvorba 47

Obrázek 39, ukázka map operátoru, vlastní tvorba 48

Obrázek 40, ukázka zápisu a čtení z ApplicationSettings, vlastní tvorba 48

Obrázek 41, ukázka detailu inventury, vlastní tvorba 50

1 Úvod

V dnešní době se využívají informační technologie, a to zejména počítače, ke všem možným úkonům. Většina lidí je s nimi v kontaktu denně, a to buď doma nebo v zaměstnání. Ovšem pouze zřídkakdy je jejich potenciál využíván tak, aby byl jejich přínos opravdu znatelný. Často jsou využívány pro tvorbu dokumentů a v případě evidence, „pouze“ jako papírová evidence s možností přemazávání dat.

Jednou z oblastí, která by mohla z informačních technologií těžit, a i těží, je i určitě oblast evidence majetku společnosti.

V oblasti evidence majetku existuje mnoho řešení, které nabízejí softwarové společnosti. Od robustních systémů, které jsou schopné pojmout stovky tisíc položek, jejichž cena převyšuje desítky milionů, po systémy, které se dají pořídit za zlomek ceny, ale s určitými limity.

Díky tomu se větší společnosti často uchylují k řešení „na míru“, buď celého softwaru, kdy si jej buď nechají zpracovat, a nebo si zřídí vlastní vývojové oddělení či personalizací stávajícího řešení. (Regli, 2009)

Ve společnostech, které spravují velké množství majetku, bývá často problém při inventarizaci s dohledáváním přemístěných položek. Obzvláště, pokud dojde ke změně jejich umístění, bez vědomí správce majetku, či jeho chybou, resp. nedůsledností při aktualizaci polohy majetku.

Hledáním majetku pak společnost zbytečně vynakládá prostředky ve formě platů zaměstnanců, kteří by mohli vykonávat jinou práci, a to jen díky tomu, že využívají konvenčních inventarizačních procesů.

Během nich je majetek pouze polepen štítky (čárovými kódy) a zaevidován, buď v listinné či elektronické podobě. Společnost ale dále nevyužívá plně potenciál informačních technologií. Inventurní procesy většinou probíhají v listinné podobě, a s tím je spjatá také řada nedostatků. Společnost tím např. může přijít o informaci umístění či stavu majetku. (Saptari, 2019)

Proto je vhodné při evidenci majetku používat mobilní zařízení s podpůrnými aplikacemi. Někteří výrobci takové aplikace nabízejí, ovšem přímo nainstalované na zařízení a často bez možnosti měnit umístění majetku v terénu.

Dříve se aplikace instalovali na uživatelské zařízení či servery, ale postupem času se převážně u všech těchto softwarových řešení přešlo na model „Software as service (SaaS)“. Což je pochopitelné a určitě i výhodné pro koncové zákazníky, kdy nejsou vázáni na určitý hardware či operační systém a tím je i částečně odstraněn nežádoucí „vendor lock-in“, kdy je zákazník odkázán na produkty jednoho výrobce či poskytovatele (Vidhyalakshmi, 2014}. Další výhoda je z hlediska aktualizací a údržby softwaru, kdy má poskytovatel daleko snazší přístup k softwaru, než v případě „on-premise“ řešení.

Problém nastává v případě, kdy ve společnosti existuje oddělená síť, bez přístupu na internet a je vyžadováno ono „on-premise“ řešení.

2 Analýza

V této kapitole je popsán výchozí stav aplikace, určení cílů a rozšíření pro aplikaci. Protože největší část je věnována části mobilní aplikace, je zde i výčet typů mobilních aplikací a jejich výhody či nevýhody.

2.1 Cíl diplomové práce

Cílem práce je vytvořit rozšíření pro vícevrstvou aplikaci pro správu majetku postavenou na frameworku Angular a NestJS. Aplikace bude rozšířena o lokace a dále bude vytvořena podpůrná mobilní aplikace prostřednictvím frameworku NativeScript, která bude umět aktualizovat umístění majetku v terénu, pomocí QR kódů a NFC štítků.

2.2 Výchozí stav

V aplikaci je možné definovat kategorie a do nich vkládat majetek a měnit jeho detaily, psát k materiálu poznámky a vkládat soubory či nahrávat fotografie. Aplikace umožňuje operace s uživateli včetně měnění práv jednotlivých uživatelů. Svěřovat majetek osobám do užívání. Každému uživateli dovoluje tvorbu vlastních sestav majetku, vlastních pohledů nad majetkem a tisknutí protokolů jako je např. předávací protokol. Aplikaci je možné provozovat na uzavřené síti.

2.2.1 Prezentační vrstva

Prezentační vrstva vícevrstvé aplikace je tvořena frameworkem Angular. V původní práci byla aplikace tvořena pomocí frameworku Angular verze 10. V rámci vývoje této práce byla verze povýšena na verzi 13.

2.2.2 Aplikační vrstva

Aplikační vrstva je vytvořena TypeScriptovým frameworkem NestJS. Její verze byla oproti původní povýšena z verze 6 na verzi 9.

2.2.3 Datová vrstva

Datová vrstva je zastoupena databází PostgreSQL, zde se jedná čistě o osobní preference autora. Operace s datovou vrstvou probíhají pomocí frameworku TypeORM, který je integrován v NestJS a umožňuje vyměnit databázi za jinou s minimálním zásahem do aplikace.

2.3 Požadavky na rozšíření aplikace

V této sekci jsou stanoveny hlavní funkční požadavky na rozšíření pro aplikaci.

2.3.1 Přístupnost

Zachování možnosti provozovat systém na oddělené síti, bez možnosti přístupu na internet. Veškerá komunikace bude probíhat prostřednictvím ethernetového připojení, včetně synchronizace dat mezi mobilní aplikací a aplikační vrstvou.

2.3.2 Lokace

Systém bude umožňovat vytvářet lokace a umísťovat do nich majetek. Tvorba lokací bude pomocí stromové struktury. Bude ji smět tvořit uživatel s oprávněním: „správce lokací“. Lokace budou identifikovány pomocí NFC štítků. Přiřazení štítků k lokacím bude probíhat pomocí mobilní aplikace. Tímto způsobem by měla být eliminována chyba lidského faktoru při ručním zadávání.

2.3.3 Změna správce majetku

Další rozšíření spočívá v převedení majetku z jednoho správce na správce druhého. Tato funkce by měla být obousměrná. Jeden správce může odeslat druhému správci žádost o převedení majetku k sobě, ale i opačně od sebe k druhému správci. Po potvrzení všech stran dojde k přesunu majetku.

2.3.4 Identifikace majetku

Z aplikace bude možné tisknout QR kódy pro identifikaci majetku. Ty budou následně využity pomocí mobilní aplikace k úkonům jako je změna umístění, tvorba sestavy či inventurní proces.

2.3.5 Mobilní aplikace

Rozšíření o mobilní aplikaci představuje největší rozšíření. Její funkce jsou rozebrány v podkapitolách této sekce. Aplikace bude vytvořená pomocí NativeScriptu a frameworku Angular. Musí umět přijmout data a odeslat data až na základě požadavku uživatele, tak aby bylo možné ji provozovat v „off-line“ režimu.

2.3.5.1 Požadavky na mobilní aplikaci

V této sekci jsou popsány požadavky na aplikaci, které musí splňovat, aby bylo možné realizovat funkce definované v následující kapitole.

2.3.5.1.1 **Připojení**

Načítání a odesílání dat bude probíhat na základě požadavku uživatele. Tímto způsobem bude možné provozovat zařízení i v off-line formě a bez připojení k internetu. Připojení k síti může být pomocí wifi routeru nebo pomocí ethernetového kabelu.

2.3.5.1.2 **Sken majetku**

Mobilní aplikace bude umožňovat skenovat QR kódy a zobrazovat tak detaily o majetku. Skenování QR kódů bude sloužit i pro inventurní procesy a aktualizaci umístění majetku.

2.3.5.1.3 **Využití NFC senzoru**

Aplikace bude umět využívat číst NFC štítky a tím získávat aktuální polohu. Tuto polohu bude dále využívat, jak pro aktualizaci umístění, tak pro inventurní proces.

2.3.5.1.4 **Persistence dat**

Data v aplikaci zůstanou i po restartu aplikace, případně zařízení, aby bylo zamezeno ztrátě dat, např. při vybití baterie mobilního zařízení.

2.3.5.2 Funkce mobilní aplikace

V této sekci jsou popsány funkce, které má mobilní aplikace umět.

2.3.5.2.1 **Přihlašování uživatelů**

Aby bylo možné identifikovat, který uživatel se zařízením pracuje, je třeba, aby v aplikaci byl mechanismus pro přihlášení uživatele.

2.3.5.2.2 **Registrace lokací**

Pro eliminování chyby při přepisu identifikátoru NFC štítku přiřazeného k lokaci (místnosti), bude probíhat registrace pomocí mobilní aplikace.

2.3.5.2.3 **Aktualizace umístění**

Na základě lokace, ve které se bude zařízení nacházet a skenovaném majetku, který se má nacházet jinde, nabídne uživateli možnost změnit umístění přímo na místě. Jakmile uživatel provede odeslání dat na server, změna umístění se provede automaticky.

2.3.5.2.4 **Tvorba sestav majetku**

V mobilní aplikaci bude možné v terénu vytvářet sestavy majetku, které budou následně nahrány k účtu přihlášeného uživatele.

2.3.5.2.5 **Provádění inventur**

V klientské části bude možné vytvořit inventuru a zvolit osobu, která bude inventuru provádět. V případě přihlášení osoby do mobilní aplikace jí bude tato inventura zpřístupněna a umožněno její provádění.

2.4 Typy mobilních aplikací

Pro lepší představu o zařazení aplikací tvořených pomocí NativeScriptu je v této sekci rozdělení mobilních aplikací z hlediska jejich technického provedení. Jsou zde zmíněny pozitivní i negativní stránky jednotlivých kategorií.

2.4.1 Nativní mobilní aplikace

Nativními mobilními aplikacemi jsou označovány takové aplikace, které jsou tvořeny přímo na míru, pro daný operační systém. Předností nativních aplikací je jejich rychlost a přímý přístup k hardwarovému API (Bjørn-Hansen, 2017).

K vytváření nativní aplikace pro Android je třeba využít jeden ze dvou programovacích jazyků, Javu nebo Kotlin. Pro IOS je třeba využít buď Swift nebo Objective-C (Zderic, 2021).

2.4.2 Hybridní mobilní aplikace

Hybridní mobilní aplikace jsou tvořeny pomocí frameworků nebo dodatečných knihoven k programovacím jazykům, které „obalují" aplikaci a vytváří jí jednotný prostor, ve kterém pracuje, nezávisle na operačním systému. V těchto frameworcích je pak možné využívat k tvorbě aplikací zejména UI, běžně používaných technologií (HTML, CSS, JavaScript). (Griffith, 2022)

Výhoda tohoto přístupu je v použití běžně používaných technologií pro tvorbu webových aplikací a možnost vydání aplikaci na IOS i Android zároveň, bez nutnosti tvorby zcela nové aplikace. Zároveň je možné pomocí přídavných balíčků získat přístupy k většině hardwarových senzorů.

Mezi hlavní nevýhody patří menší výkon v aplikacích, v porovnání s nativními aplikacemi a vývojář aplikace je odkázán na tvůrce knihoven, resp. frameworků, v případě přístupu k hardwarovému API.

Mezi představitele takovýchto frameworků je např. Ionic nebo Flutter.

2.4.3 Progressive Web Apps (PWA)

PWA je technologie, která umožňuje instalaci webové aplikace na zařízení s různými operačními systémy. K tomuto slouží tzv. manifest, který obsahuje nastavení aplikace. Dále je možné využít běžně používaných technologií k vývoji.

K výměně dat se serverem slouží service workery. Ty umožňují nejen výměnu, ale i správu dat.

Díky podpoře společností Microsoft a Google je hlavní předností tohoto přístupu možnost instalace aplikací i jako desktopové aplikace na novějších verzích Windows či Android, bez nutnosti návštěvy služby Google play, ale přímo z URL webové stránky.

Hlavní nevýhoda je velice omezený přístup k hardware API.

2.4.4 Interpretované mobilní aplikace

Jedná se o přístup, který umožnuje pomocí JavaScriptových frameworků vytvářet aplikace, které se skládají z JavaScript virtual machine, runtimu a mostu. Pomocí tohoto mostu pak aplikace používá volání přímo s konkrétním systémem a umožňuje tak komunikovat s API pomocí JavaScriptu místo Kotlinu (Android), resp. Swiftu (IOS).

Výhody tohoto přístupu je rychlost, která je téměř totožná s nativní aplikací, vzhled prvků nativní aplikace, přístup k hardwarovému API a možnost vývoje aplikace jak pro Android, tak na IOS, pomocí jednoho kódu.

Nevýhoda tohoto přístupu je v použití specifických prvků na view. Představitele tohoto přístupu jsou React-native a NativeScript.

2.5 QR kódy a NFC štítky

NFC technologie, která je známá širší veřejnosti, hlavně díky platbám v obchodech, se stává čím dál více dostupnější. Velké společnosti, jako je Google, Apple, Samsung do ní každoročně investují více a více finančních prostředků (Gegeckiene, 2022).

Méně jsou pak známé tzv. „NFC štítky". Ty umožňují pasivní přenos informací ze štítku na zařízení, které se nachází v jeho blízkosti. Existují různé typy a velikosti těchto štítků, některé na sebe dovolují zapisovat informace o různých délkách. Běžně je tato kapacita v řádu stovek bytů. Nejmenší velikost těchto štítků je pouhých

5mm x 5mm. Každý NFC štítek obsahuje jedinečné identifikační číslo, které se ve spolupráci s jinou aplikací stává mocným nástrojem. Jejich využití je velice univerzální, od připojení k wifi, odkazu na stránku či získání informací o hospotalizovaném, (Manimuthu, 2021) až po cílenou nabídku či reklamu v obchodě. (Bourg 2021).

Podobnou úlohu jako NFC štítky mohou zastávat QR kódy. Jejich minimální velikost je 2cm x 2cm. Jejich nesporná výhoda je, že je možné si je vytisknout a případně jednoduše nahradit stávající poškozené či nečitelné kódy.

3 Technologie použité pro tvorbu aplikace

V této kapitole jsou představeny použité technologie pro tvorbu vícevrstvé aplikace. Frameworky Angular a NestJS jsou zde zmíněny jen okrajově. Jejich podrobnější rozebrání je popsáno v Bakalářské práci, na kterou tato práce navazuje.

3.1 Angular

Obrázek 1, Angular logo, zdroj: https://angular.io/

Angular je jeden z předních frameworků pro tvorbu tzv. „single page apps“ (dále jen SPA). Hlavní rozdíl mezi SPA a standardní webovou stránkou je, že pokud uživatel klikne na odkaz, tak se načte celá nová stránka, kdežto u SPA se načte pouze požadovaný obsah. (Lim, 2017).

3.2 NestJS

Obrázek 2, NestJS logo, zdroj: https://nestjs.com/

NestJS je framework využívající platformu NodeJS pro tvoření „server-side" aplikací a stejně jako Angular, tak i NestJS využívá TypeScript.

3.3 NativeScript

Tato kapitola je věnovaná frameworku pro tvorbu mobilních aplikací. Pomocí

Obrázek 3, NativeScript logo, zdroj: https://docs.nativescript.org/api-reference/

NativeScriptu je možné tvořit interpretované mobilní aplikace buď pomocí JavaScriptu, resp. TypeScriptu, případně s využitím některého z předních javascriptových frontendových frameworků jako je React, Vue, Svelte a Angular, kde je třeba dodržet určité odlišnosti.

3.3.1 Rozdíly použití Angularu bez a s NativeScriptem

Obrázek 4, Angular NativeScript logo, zdroj: https://medium.com/@tj.vantoll/nativescript-angular-the-future-of-mobile-app-development-d83cfeecfebd

V této kapitole jsou popsány a názorně ukázány základní odlišnosti použití view v Angularu, který je využit pro tvorbu webové aplikace, oproti použití Angularu pro tvorbu mobilní aplikace pomocí NativeScriptu.

3.3.1.1 View

Ve vývoji webové aplikace se Angularu pro tvorbu view využívá zejména HTML. Tvorba layoutu pak probíhá pomocí několika do sebe vnořených HTML tagů s různými vlastnostmi nastavenými pomocí kaskádových stylů (Google, 2023).

Naproti tomu tvorba view ve spolupráci s NativeScriptem probíhá pomocí XML. Pro každou část existuje několik prvků, které má vývojář k dispozici a těch se musí držet.

3.3.1.2 Layout

V této sekci jsou popsány prvky, které slouží pro skládání prvků v uživatelském rozhraní aplikace. V NativeScriptu existuje celkem 7 typů těchto prvků: AbsoluteLayout, DockLayout, FlexboxLayout, GridLayout, RootLayout, StackLayout, WrapLayout (OpenJS Foundation, 2023a). Pro většinu aplikací si ovšem vývojář vystačí s kombinací tří resp. 4 prvků a to FlexBoxLayout, GridLayout, StackLayout a případně AbsoluteLayout. Proto jsou v této sekci popsány pouze tyto typy rozložení.

3.3.1.2.1 **FlexboxLayout**

Prvek FlexboxLayout dává vývojáři do rukou nástroj pro tvorbu layoutu, obdobný jako při použití technik flexboxu za pomocí použití HTML a CSS.

```
<FlexboxLayout backgroundColor="#3c495e">
  <label text="first" width="70" backgroundColor="#43b883" />
  <label text="second" width="70" backgroundColor="#1c6b48" />
  <label text="third" width="70" backgroundColor="#289062" />
</FlexboxLayout>
```

Obrázek 5, ukázka flexbox, zdroj: https://docs.nativescript.org/ui-and-styling.html#flexboxlayout

Jak je vidět na Obrázek 5, prvek obsahuje párový tag FlexboxLayout a uvnitř něj jsou pro ilustraci tři textové prvky. Toto je základní zápis bez použití jakýchkoliv nastavení. Výsledek je znázorněn na Obrázek 6.

Obrázek 6, výsledek flexboxLayoutu vlastní tvorba

Jen pro ilustraci, dosažení tohoto vzhledu prvků pomocí HTML a CSS by byl zápis tak, jak je znázorněno na Obrázek 7.

```
<style>
  .flexContainer {
    display: flex;
    height: 50px;
    background-color: #3c485e;
  }
  .flexContainer div {
    color: white;
    display: flex;
    align-items: center;
    justify-content: center;
  }
</style>
<div class="flexContainer">
  <div style="width: 70px; background-color:#43b883">first</div>
  <div style="width: 70px; background-color:#1c6b48">second</div>
  <div style="width: 70px; background-color:#289062">third</div>
</div>
```

Obrázek 7, ukázka zápisu pomocí HTML a CSS, vlastní tvorba

Vlastnosti se upravují pomocí atributů umístěných ve značce tagu. Například, pokud je třeba změnit orientaci prvků z řádkového rozložení na sloupcové, použije se atribut flexDirection s hodnotou „column“ (ukázka Obrázek 8). Pokud není nastaven, použije se „row“.

```
<FlexboxLayout flexDirection="column" backgroundColor="#3c495e">
  <label text="first" height="70" backgroundColor="#43b883" />
  <label text="second" height="70" backgroundColor="#1c6b48" />
  <label text="third" height="70" backgroundColor="#289062" />
</FlexboxLayout>
```

Obrázek 8, ukázka použití atributu, zdroj: https://docs.nativescript.org/ui-and-styling.html#flexboxlayout

3.3.1.2.2 **StackLayout**

StackLayout umožňuje skládat prvky uživatelského rozhraní přirozeně za sebou. A to v řadě za sebou, nebo ve sloupci pod sebou. Prvky se v tomto rozložení řadí za sebe nebo pod sebe v závislosti na hodnotě atributu "orientation“, která může být: „vertical“ nebo „horizontal“.

Rozdíl oproti FlexboxLayoutu je např. to, že ve stackLayout není možné např. zarovnat prvky na začátek a konec ve zvoleném rozložení.

3.3.1.2.3 **GridLayout**

GridLayout je hodně využívaný systém rozložení, jelikož umožňuje předem definovat, z kolika sloupců a řádků se bude rozložení skládat, a poté v něm uspořádávat prvky.

```
<GridLayout columns="115, 115" rows="115, 115">
  <label text="0,0" row="0" col="0" backgroundColor="#43b883" />
  <label text="0,1" row="0" col="1" backgroundColor="#1c6b48" />
  <label text="1,0" row="1" col="0" backgroundColor="#289062" />
  <label text="1,1" row="1" col="1" backgroundColor="#43b883" />
</GridLayout>
```

Obrázek 9, ukázka zápisu gridLayout, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout

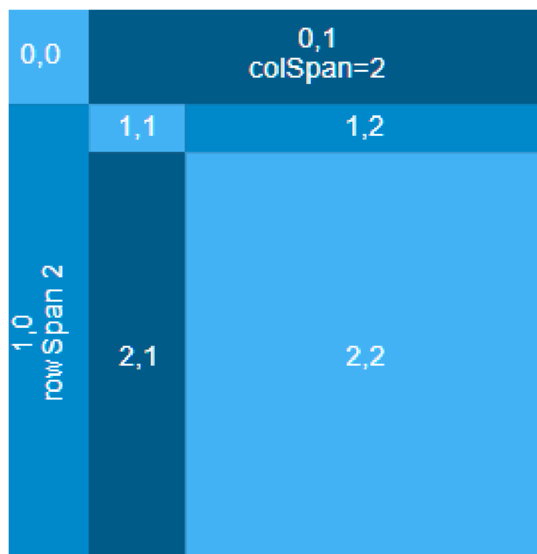
Ve výše uvedené ukázce Obrázek 9 jsou pomocí atributu columns a hodnotou „115, 115" definovány dva sloupce o šířce v hodnotě 115 bodů. Dále jsou definovány dva řádky o stejné výšce. Umístění prvků je definované přímo v atributu prvku, kde se nastavuje pomocí row a col indexu příslušného sloupce, resp. řádku.

Pokud má atribut přesahovat přes více řádků či sloupců nastavuje se mu atribut rowSpan reps. colSpan s hodnotou počtu přesahu (kód viz. ukázka Obrázek 10, výsledek zobrazení viz. Obrázek 11)

```
<GridLayout columns="40, auto, *" rows="40, auto, *" backgroundColor="#3c495e">
  <label text="0,0" row="0" col="0" backgroundColor="#43b883" />
  <label text="0,1" row="0" col="1" colSpan="2" backgroundColor="#1c6b48" />
  <label text="1,0" row="1" col="0" rowSpan="2" backgroundColor="#289062" />
  <label text="1,1" row="1" col="1" backgroundColor="#43b883" />
  <label text="1,2" row="1" col="2" backgroundColor="#289062" />
  <label text="2,1" row="2" col="1" backgroundColor="#1c6b48" />
  <label text="2,2" row="2" col="2" backgroundColor="#43b883" />
</GridLayout>
```

Obrázek 10, gridLayout rowSpan, colSpan, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout

Existují zde ještě hodnoty, které neurčují přesný počet bodů, ale vypočítávají např. hodnotu „auto", která určuje šíři sloupce, jakou potřebuje, aby se vykreslil korektně.



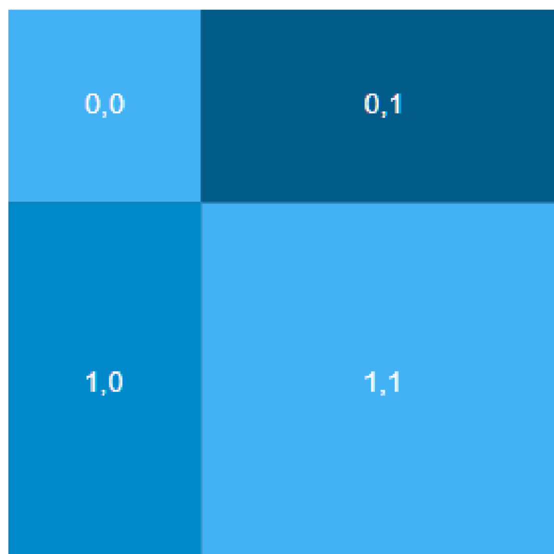
Obrázek 11, zobrazení gridLayoutu, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout

Další znak je „*“, který určuje, jak bude sloupec či řádek široký (vysoký), podle toho, kolik zbývá místa. tzn. roztáhne se a vyplní prázdný prostor, který vznikne po ostatních prvcích. Je možné jej nastavit i relativně. tzn. může být uvedeno i s hodnotou „2*“. Pro lepší pochopení je ukázka kódu na Obrázek 12.

```
<GridLayout columns="*, 2*" rows="2*, 3*" backgroundColor="#3c495e">
  <label text="0,0" row="0" col="0" backgroundColor="#43b883" />
  <label text="0,1" row="0" col="1" backgroundColor="#1c6b48" />
  <label text="1,0" row="1" col="0" backgroundColor="#289062" />
  <label text="1,1" row="1" col="1" backgroundColor="#43b883" />
</GridLayout>
```

Obrázek 12, použití * v gridLayoutu, zdroj: https://docs.nativescript.org/ui-and-styling.html#gridlayout

Zde jsou v atributu columns definovány dva sloupce s hodnotou „*, 2*. Tento zápis určuje, že výsledné sloupce se roztáhnou po celé šíři a druhý bude 2x širší než první. Zároveň jsou definovány dva řádky s hodnotou: „2*,3*“, která určuje, že první řádek bude mít výšku 2/5 místa stanoveného pro gridLayout. Druhý pak bude mít výšku 3/5. Výsledná ukázka je na obrázku č. Obrázek 13.

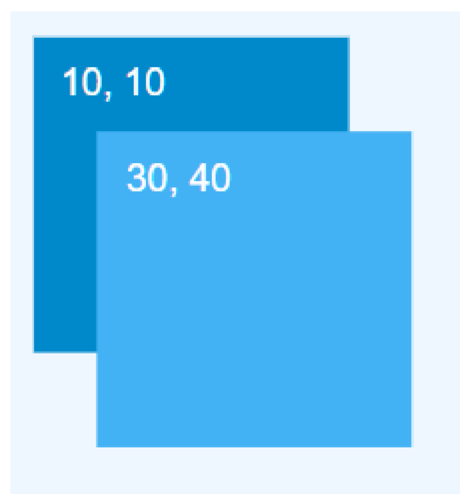


Obrázek 13, ukázka * v gridLayoutu: https://docs.nativescript.org/ui-and-styling.html#gridlayout

3.3.1.2.4 **AbsoluteLayout**

Pomocí AbsoluteLayout typu zobrazení se umisťují prvky na obrazovce pomocí atributů: „top, left“ a výškou a šířkou. Tento typ umožňuje přetékání prvků přes sebe.

Prvek, který je umístěný v kódu dále přeteče prvek předchozí viz. Obrázek 14.



Obrázek 14. absoluteLayout, zdroj: https://docs.nativescript.org/ui-and-styling.html#absolutelayout

3.3.1.3 Komponenty

V této sekci jsou zmíněny základní komponenty pro uživatelské rozhraní mobilní aplikace.

Label, slouží pro vypsání textového řetězce na obrazovku. Obdobou tohoto prvku v HTML by byl např. *span*.

Button, je komponenta, která vytváří tlačítka. Oproti „(click)“ události emituje „(tap)“, ale jinak je její použití stejné, jako v případě webové aplikace.

TextField, je obdoba inputu v HTML. Pomocí „data bindingu“ (i two-way) má pak Angular možnost přistupovat k hodnotě prvku, stejně jako v případě webové aplikace.

ScrollView komponenta přidává možnost scrollování prvku. V kódu se umísťuje jako „obalovací“ prvek, komponenty, které bude scrollování umožněno, zejména pak layoutu.

ListView, je komponenta, která slouží pro virtualizaci a zobrazení seznamu.

3.3.1.4 Kaskádové styly

NativeScript umožňuje využití kaskádových stylů. Jsou zde ovšem určitá omezení. Obecně platí, že jakákoli vlastnost, která má v css hodnotu velikosti nebo barvy, je možné využít i zde (OpenJS Foundation, 2023b).

Hlavní odlišností od běžného užití je, že se zde nenastavují hodnoty s jednotkou. Tzn. výška se zde zapíše pouze jako *„height: 50“* nikoli: *„height: 50px“* a má hodnotu 50 „bodů“.

Seznam podporovaných vlastností je možné najít v oficiální dokumentaci na adrese: https://docs.nativescript.org/ui-and-styling.html#supported-css-properties

3.3.1.5 Moduly

V této sekci se nachází odlišnosti v použití běžných modulů, které se využívají v téměř každé aplikaci tvořené Angularem.

3.3.1.5.1 **NativeScriptModule**

Aplikace založená na NativeScriptu se využívá místo BrowserModulu. Tento modul aplikaci zprostředkovává přístup ke službám, které umožňují běh aplikace, zejména pak renderer. BrowserModul se tedy v NativeScript aplikaci v Angularu nevyužívá.

3.3.1.5.2 **NativeScriptHttpClientModule**

NativeScriptHttpClientModule je modul, který umožňuje vytvářet http dotazy, pomocí služby HttpClient. Jedná se o obdobu modulu HttpClientModule.

3.3.1.5.3 **NativeScriptRouterModule**

NativeScriptRouterModule nahrazuje RouterModule. Pomocí tohoto modulu je možné využívat routingu v aplikaci. V aplikaci není možné využít atribut routerLink, ale nsRouterLink. Pokud je třeba využít router mimo view, je třeba router injektovat jako RouterExtensions.

3.3.1.5.4 **NativeScriptFormsModule**

K tomu, aby aplikace mohla využívat především two-way data bindingu, slouží modul NativeScriptFormsModule, který tuto vlastnost dovoluje využívat. Two-way data binding je pak přístupný pomocí [(ngModule)].

3.4 RxJS

Obrázek 15, RxJS logo, zdroj: https://worldvectorlogo.com/logo/rxjs-1

Při vývoji byla využita knihovna RxJS, která je součástí frameworku Angular. Pro lepší pochopení problematiky jsou zde zmíněny nejvíce používané techniky v rámci projektu. Knihovna slouží k práci se streamy dat (ty nazývá tzv. Observables, dle konvence jsou značeny $ na konci názvu). Umožňuje využít řadu funkcí na jejich kombinace a přetváření, a při každé změně jejich hodnoty informuje odběratele (tzv. Subscribery).

3.4.1 Přihlášení k odběru

Přihlášení k odběru záleží na typu aplikace, která RxJS využívá a o typ dat ke kterým se jako Subscriber přihlašuje. Každé přihlášení k odběru, které není ukončeno (není na něm volána metoda complete()), je třeba ukončit. Pokud by se tak nestalo může v aplikaci docházet k tzv. memory leakum.

3.4.1.1 Async pipe (Angular)

V případě použití RxJS a Observables ve frameworku Angular je nejčistší způsob, jak se přihlásit k odběru Observables pomocí tzv. „async pipe".

```
<div *ngIf="stockTaking$ | async as stockTaking" fxFlex fxLayout="column">
  <div...>
  <div fxLayout="column" fxLayoutGap="1rem" fxFlex...>
</div>
```

Obrázek 16, použití Async pipe - Angular, vlastní tvorba

Na Obrázek 16 je vidět použití na view. Na aplikační části je pak pouze přiřazen do položky „stockTaking$" ukazatel na Observable. Nejčistší způsob spočívá v tom, že napíšeme nejméně kódu, kód je lépe čitelný a není potřeba provádět odhlášení se z odběru u nekončících Observables.

3.4.1.2 Metoda subscribe

Pokud není možné využít metodu pomocí Async pipe, můžeme se přihlásit k odběru pomocí metody subscribe(). V takovém případě musíme myslet na to k jakým Observables se přihlašujeme, zda se jedná o typ, který končí či nekončí. Níže uvedené subjekty, které jsou velice využívané mohou být, a ve většině případů jsou, typ nekončících Observables. Naproti tomu volání http požadavku je typ Observales, který končí a není třeba se z jeho odběru odhlašovat.

Na následující ukázce (Obrázek 17) je vidět, že je potřeba před voláním metody subscribe nastavit, do jaké doby se má odběr používat (metoda takeUntil), do ní je potřeba vložit další typ Observables, který svým emitováním odběr ukončí. V tomto případě se odběr ukončí při zániku komponenty, kdy se volá na subjektu „_unsubscribe" metoda next().

Tělo metody subscribe se zavolá vždy, když se změní hodnota Observable, na které je přihlášena. S hodnotu je pak třeba dále pracovat. V uvedeném případě je emitovaná hodnota přiřazena to pole „entity".

```
export class MinimalisticShowCase implements OnDestroy {
  private _unsubscribe: Subject<void> = new Subject<void>();
  entity: EntityA[] = [];

  constructor(private dataProvider: DataService) {
    this.dataProvider.entityA$
      .pipe(takeUntil(this._unsubscribe))
      .subscribe( next: entityA => {
      this.entity = entityA;
    })
  }

  ngOnDestroy(): void {
    this._unsubscribe.next();
    this._unsubscribe.complete();
  }
}
```

Obrázek 17, ukázka metody subscribe, vlastní tvorba

3.4.2 Operátory

Knihovna poskytuje obrovské množství operátorů, které slouží pro práci s Observables. Aktuálně jich je více než 100 (Lesh, 2023a), proto jsou zde uvedeny ty, které jsou nejběžněji používané a využity v rámci této práce. Operátory je možné za sebou řetězit a transformovat tak emitovaný výsledek Observables. K řetězení slouží metoda „pipe()", do které se operátory vkládají oddělené čárkou.

3.4.2.1 CombineLatest

Do operátoru combineLatest se vkládá pole Observables, a tento operátor emituje, až když z Observable v poli emituje aspoň jednou. Hlavní využití tohoto operátoru je, když je více druhů Observable, jejichž hodnoty mají reference na ostatní.

Může se jednat např. o situaci, kdy jsou čtyři Observables: uživatelé, inventury, inventurní_položky a položky majetku. V tomto případě má inventurní položka referenční klíč na položku majetku a inventuru. Majetek obsahuje identifikátor na uživatele, který majetek užívá a inventura obsahuje ID řešitele a tvůrce. Jedno bez druhého nemá význam. Proto jsou tyto čtyři Observable volány operátorem

combineLatest. Jakmile je emitována hodnota z každého Observable, emituje operátor a jeho výsledkem je pole získaných hodnot z přiřazených operátorů.

3.4.2.2 ForkJoin

Operátor forkJoin podobně jako combineLatest vyžaduje pole Observables, ale emituje až v případě, kdy všechny streamy skončí (Lesh, 2023b).

3.4.2.3 TakeUntil

Operátor takeUntil, vyžaduje jako parametr jiný Observable, který když emituje hodnotu, tak ukončí běh Observable, ve kterém je použit. Používá se zejména v případech, kdy je potřeba zajistit odhlášení od odběru.

3.4.2.4 Map

Operátor map slouží ke změně hodnot výsledného streamu. Používá se zejména v případech, kdy chceme přetvořit výsledek nějakou transformační funkcí.

3.4.2.5 Tap

Operátor tap umožňuje vytvářet tzv. „side efekty“. Tento operátor nijak s daty nemanipuluje, pouze vidí jejich obsah a může volat jiné funkce a data jim předávat.

3.4.2.6 SwitchMap

Operátor switchMap má za úkol vrátit nový Observable, který původní nahradí. Jeho využití je zejména, pokud máme vytvořený Observable na změnu url adresy. Jeho výsledek je pak možné využít a vrátit nový Observable, který již ukazuje na konkrétní ID záznamu. Takový příklad je uvedený na Obrázek 18.

```
this.stockTaking$ = this.route.paramMap.pipe(
  switchMap( project: (paramMap : ParamMap ) => {
    const uuid = paramMap.get('uuid');
    if (uuid) {
      return this.stockTakingService.getOne$(uuid);
    }
  }
))
```

Obrázek 18, operátor switchMap, vlastní tvorba

V tomto případě se na základě změny hodnoty „uuid“, získané z cesty změní Observable na nový, který obsahuje pouze data pro položku s identifikátorem „uuid“.

3.4.2.7 Pairwise

Operátor pairwise je vhodné použít, pokud chceme porovnávat dva výsledky z jednoho streamu dat. Operátor emituje až při druhé hodnotě a s ní vrací hodnotu předchozí.

3.4.2.8 FirstValueFrom

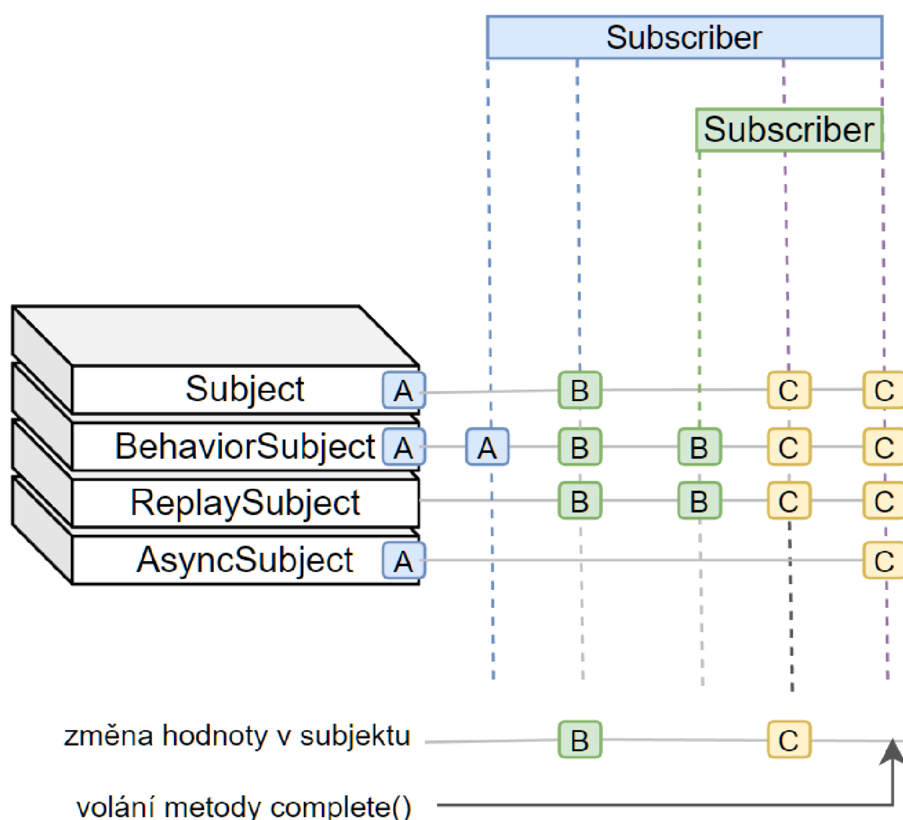
Operátor firstValueFrom byl uveden v nové verzi RxJS . Nahrazuje použití operátoru take(1) s metodou toPromise(). Jeho úkolem je vrátit Observable v typu Promise, ihned po první obdržené hodnotě.

3.4.2.9 StartWith

Operátor startWith se používá v případech, kdy je potřeba nastavit výchozí hodnotu ještě před tím, než bude Observable emitovat data.

3.4.3 Subjekty

Subjekty jsou speciálním typem Observables, které umožňují uchovávat data a informovat své Subscribery. Knihovna nabízí 4 druhy subjektů, které se liší především tím, kdy poskytují data svým Subscriberům, jak je vidět na Obrázek 19



Obrázek 19, přehled subjektů v RxJS, vlastní tvorba

3.4.3.1 Subject

Subject je typ, který informuje své Subscribery až v případě, když nastane změna jeho hodnoty. Nezávisle na tom, jakou hodnotu má v případě přihlášení Subscribera k odběru. Nevyžaduje nastavit hodnotu ve své iniciaci.

3.4.3.2 BehaviorSubject

BehaviorSubject je speciální typ subjektu, který poskytuje svojí aktuální hodnotu i nově zaregistrovaným Subscriberům. Ze své podstaty vyžaduje nastavit hodnotu při iniciaci.

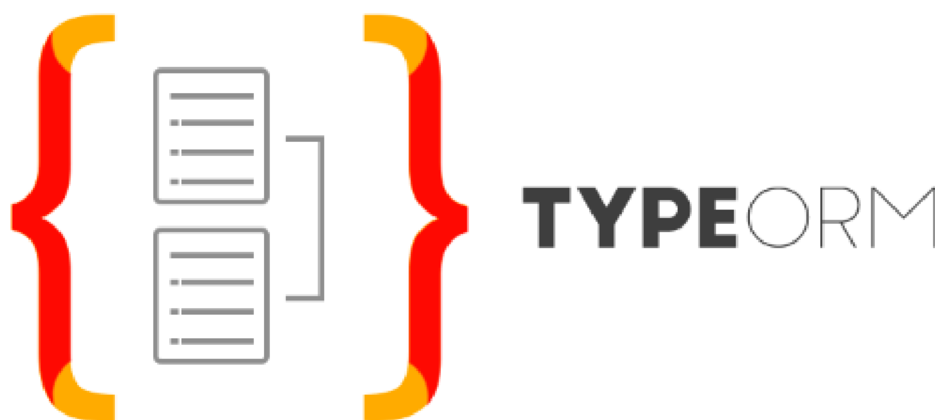
3.4.3.3 ReplaySubject

ReplaySubject je speciální typ subjektu, který poskytuje svojí hodnotu i nově zaregistrovaným Subscriberům, ale hodnotu oznamuje až v případě, že je mu nastavena. Zároveň umí držet i historii. V iniciaci se zadává kolik hodnot (verzí) si má subjekt uchovávat.

3.4.3.4 AsyncSubject

AsyncSubject je typ subjektu, který poskytuje svojí hodnotu až v případě, že je na něm zavolána metoda „complete()“

3.5 TypeORM



Obrázek 20. TypeORM logo, zdroj: https://typeorm.io/

Pro komunikaci s databází je využit objektově relačního mapování (ORM) pomocí TypeORM. TypeORM je objěktově relační mapper, který může být provozován na serveru postaveném na platformách NodeJS, Ionic, React Native, NativeScript a dalších. Podporuje architektonické návrhové vzory Data Mapper a Active Record. Dále umožňuje definovat vztahy mezi objekty, které jsou zde nazývány jako entity. Pomocí těchto entit jsou pak tvořeny tabulky v databázi.

3.5.1 Definice vztahů mezi entitami

V této kapitole jsou popsané definice vztahy mezi entitami v TypeORM. Ty jsou definovány pomocí níže uvedených dekorátorů. Výsledné vztahy pak mohou být reprezentovány buď ihned nebo až na vyžádání.

3.5.1.1 Vztahy mezi entitami

V této sekci je přehled možných vztahů tvořených v TypeORM.

1. Vztah 1:1 (OneToOne)

   Tato vazba se používá v případě, kdy mohou mít entity mezi sebou vztah právě maximálně 1:1. Většinou je tento vztah ne 1:1, ale 1:0-1. Jedna strana bývá často povolena jako nullable.

   V dekorátoru se na prvním místě uvádí type funkce, ve které je uvedena entita, ke které se vazba vztahuje, následuje označení atributu z opačné strany, který ukazuje na tuto entitu. Jako poslední je možné vložit do dekorátoru objekt s nastavením chování vztahu, zejména pak zda se bude vztah načítat ihned nebo až na vyžádání.

2. Vztah 1:N (OneToMany)

   Vztah 1:N se definuje pomocí dekorátoru @OneToMany a obsahuje parametry stejně jako vztah 1:1. Díky tomuto operátoru je pak možné vrátit pole záznamů v entitě uvedené v dekorátoru.

3. Vztah N:1 (ManyToOne)

   Vztah N:1 je možné definovat dekorátorem @ManyToOne, kde na prvním místě je taktéž typová funkce odkazující na entitu, ale na další pozici je označení atributu, který je primární klíč. Poslední atribut je opět objekt s nastavením.

4. Vztah M:N (ManyToMany)

   Vztah M:N je možné definovat dekorátorem @ManyToMany, ten má pouze dva vstupní parametry. První je typová funkce a druhý je nastavení. Tento dekorátor ovšem vyžaduje ještě další dekorátor, a tím je definice spojovací tabulky @JoinTable. V tomto dekorátoru se definuje název spojovací tabulky a sloupce zajišťující spojení dvou entit. Ukázka použití je vidět na Obrázek 21.

```
@ManyToMany( typeFunctionOrTarget: () => Rights,  options: { cascade: true, eager: true })
@JoinTable( options: {
  name: 'users_rights',
  joinColumn: { name: 'user_id', referencedColumnName: 'id' },
  inverseJoinColumn: { name: 'rights_id', referencedColumnName: 'id' },
})
rights: Rights[];
```

Obrázek 21, Ukázka @ManyToMany, vlastní tvorba

3.5.1.2 Načítání relací ihned vs na vyžádání

Načítání relací ihned, je možné pomocí vlastnosti eager nastavené na true. Opakem je pak načtení na základě vyžádání. To je nastaveno pomocí hodnoty lazy: true. Pokud by byla hodnota nastavena na false, tak by to znamenalo její opak, ale pro lepší čitelnost, se využívá tento zápis (Khudoiberdiev, 2023).

Využívání načítání ihned může mít dopad na rychlost načítání dat obzvláště pokud se jedná o rozsáhlé tabulky, o to více pokud je takových atributů v entitě více.

Doporučené je využívat především „lazy-loadingu“ a k entitám přistupovat jako k typu Promise. Rozdíl v rychlosti u větších tabulek je výrazný. V obou případech je potřeba dát pozor na to, jak je označen typ atributu. Pokud je označen návratový typ jako <Pole hodnot>, v případě, kdy je použit lazy-loading a ne Promise<Pole hodnot>, tak nás na to TypeScript neupozorní, a případná chyba se bude špatně dohledávat.

4 Návrh a implementace

V této kapitole jsou popsány implementace Angularu, NestJS a NativeScriptu. Jsou zde vysvětleny principy, myšlenky projektu a jejich implementace. Každé rozšíření má svojí sekci, ta je dále dělená na základní myšlenku, návrh a implementaci v každé z dotčených částí systému.

4.1 Lokace

Aby bylo možné lépe dohledat majetek, je třeba, aby systém obsahoval lokace. Ať už se jedná o budovy, místnosti či jiná umístění, je vhodné je tvořit stromovou strukturou tak, aby bylo možné získat lokace, které jsou uvnitř lokace vybrané.

V případě přidání lokací je třeba vyřešit několik úkolů:

- Přidání, editaci a smazání lokace
- Zobrazení seznamu a detailu
- Přidání lokace k majetku
- Přidání NFC štítků k lokaci – tento úkol je řešen v rámci mobilní aplikace

4.1.1 Implementace lokace - Angular

Lokace jsou na straně klienta reprezentovány samostatným modulem, který obsahuje službu, která se stará o obhospodařování požadavků a tři komponenty. Komponenta pro zobrazení seznamu lokací, komponenta sloužící pro úpravu jedné lokace a poslední je pomocná komponenta, která přidává k detailu možnosti tvorby nové lokace pomocí zdvojení předchozí komponenty (Obrázek 22).

```
<div fxFlex fxLayout="column" fxLayoutGap=".5rem" *ngIf="locations$ | async as locations">
  <ng-container *ngIf="location$ | async as location">
    <app-location-detail
      *ngIf="location.uuid"
      [location]="location"
      [locations]="locations"
      (saveEmit)="onSaveEmit($event)"
    ></app-location-detail>
  </ng-container>
  <app-location-detail
    [location]="newLoc"
    [locations]="locations"
    (saveEmit)="onSaveEmit($event)"
  ></app-location-detail>
</div>
```

Obrázek 22, dvojí využití komponenty, vlastní tvorba

Pro zobrazení seznamu, jako stromové struktury, je využita knihovna ag-grid. V případě nastavení parametru „treeData“, vyžaduje poskytnutí informace o zanoření objektu pro každý záznam. Stromová struktura byla využita v projektu již v modulu „jednotky“, kde byl využit algoritmus prohledávání do hloubky, protože data byla poskytována ze serveru ve stromové struktuře. V případě lokací byla využita rekurze ve třídě Location (Obrázek 23).

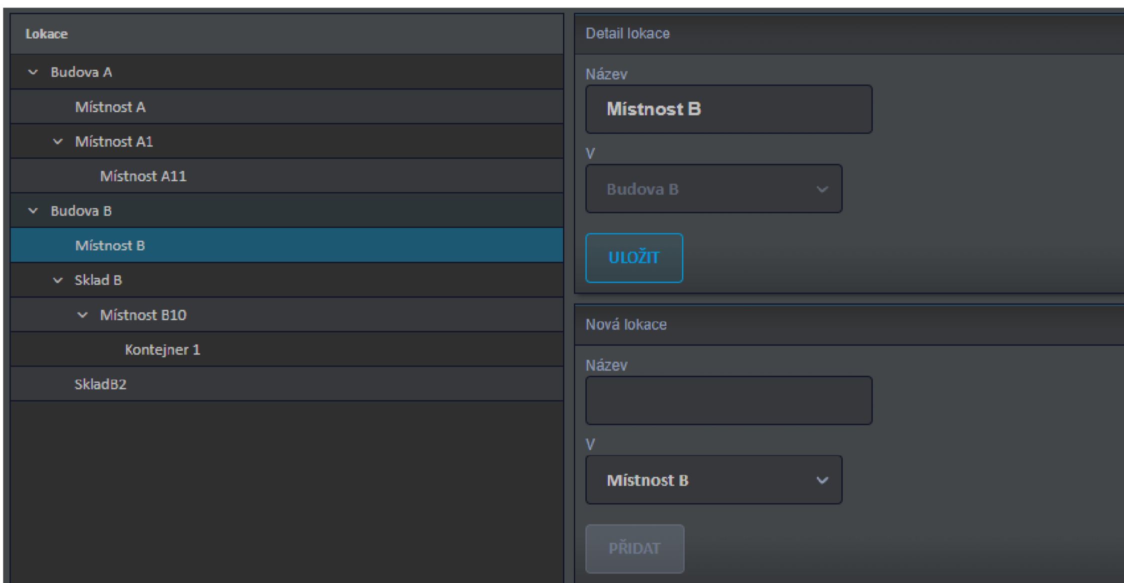
```
get treePath(): string[] {
  return this.recursiveSearch(this, treePath: []);
}

private recursiveSearch(parent: Location, treePath: string[]): string[] {
  if (parent.uuid) treePath.unshift(parent.uuid)

  if(parent.parent && parent.parent.uuid) {
    return this.recursiveSearch(parent.parent, treePath)
  }
  return treePath;
}
```

Obrázek 23, rekurze ve třídě Location, vlastní tvorba

Výsledný vzhled je složen v komponentě dashboard. Dále byly provedeny úpravy v komponentách zobrazující seznamy majetku a detail majetku, aby bylo možné majetek přiřadit do lokace i přes webové rozhraní aplikace (Obrázek 24).



Obrázek 24, ukázka výsledku dashboardu, vlastní tvorba

4.1.2 Implementace lokace – NestJS

Na serveru byly vytvořeny nové koncové body pro získání všech lokací, vytvoření, provedení změny a smazání lokace. Dále pak endpoint, který přiřadí lokaci k identifikátor z NFC štítku.

4.1.2.1 Koncové body

Metoda	Url	Popis
Get	locations	Získání pole všech jednotek
Post	locations	Tvorba lokace
Patch	locations/:uuid	Změna lokace s identifikátorem: uuid
Patch	locations/:uuid/nfc	Změna v přiřazení NFC štítků k lokaci
Delete	locations/:uuid	Smazání lokace s identifikátorem: uuid

4.1.2.2 Definice entity - Lokace

Lokace jsou reprezentovány v TypeORM jako entita tvořena pomocí stromové struktury: „closure table“. Pro využití stromové struktury je třeba využít dekorátoru @Tree(<specifikace typu>) viz. Obrázek 25

```
@Entity()
@Tree( type: 'closure-table')
export class Location extends BaseEntity {
  @PrimaryGeneratedColumn( strategy: 'uuid')
  uuid: string;

  @Column()
  @Generated()
  ord: number;

  @Column( options: { nullable: true })
  nfcId: string;

  @Column()
  name: string;

  @TreeChildren()
  children: Location[];

  @TreeParent()
  parent: Location;
  @RelationId( relation: (loc: Location) => loc.parent)
  parent_uuid: string;

  @ManyToOne( typeFunctionOrTarget: (type) => Unit,  inverseSide: (unit : Unit ) => unit.id, options: { eager: true })
  masterUnit: Unit;

  @VersionColumn( options: { default: 1 })
  version: number;
}
```

Obrázek 25, TypeORM entita - Location, vlastní tvorba

Typ „closure table“ je jeden z několika způsobů, který se dá v případě TypeORM využít. Pomocí tohoto typu jsou vztahy mezi nadřízenými a podřízenými zaznamenávány do oddělené tabulky. Entita lokace obsahuje tyto atributy:

1. <u>uuid</u>

   Jedná se o řetězec, který je unikátní a slouží jako jednoznačný identifikátor. Je automaticky gererován dekorátorem @PrimaryGeneratedColumn(‚uuid‘), kde ‚uuid‘ je specifikace metody, pomocí které bude klíč generován.

2. ord

   Automaticky generované číslo, které určuje pořadí, v jakém byla lokace vytvořena.

3. nfcId

   Jedná se o řetězec, který obsahuje identifikátor k NFC štítku. Tímto je možné identifikovat lokaci pomocí NFC štítku. Protože je prvně vytvořena lokace a následně se k ní štítek přidává, je potřeba, aby tato hodnota mohla obsahovat hodnoty „NULL“. To je nastaveno pomocí parametru: „nullable: true“ v nastavení sloupce pomocí dekorátoru @Column() nad definicí vlastnosti.

4. name

   Jedná se vlastnost, která obsahuje řetězec reprezentující název lokace

5. children

   Tato vlastnost je definována speciálním dekorátorem @TreeChildren(). Ten poskytuje vztah 1:N na entitu Location. Neznamená to, ale že se s atributem pracuje. Jde čistě o definici vztahu. Pokud bychom chtěli mít k dispozici její následovníky, bylo by potřeba ještě pod touto vlastností definovat vztah @OneToMany().

6. parent

   Touto vlastností se definuje vztah na rodiče dekorátorem @TreeParent(). Stejně jako ve výše uvedeném případě, neznamená to, že je hodnota jakkoli přístupná ve výsledném objektu. Toho je dosaženo až díky dekorátoru @RelationId(). V tomto dekorátoru specifikujeme třídu, ze které bude vlastnost vybrána. Hodnota reprezentující identifikátor je pak přístupná přes vlastnost parent_uuid.

7. masterUnit

   Vlastnost definovaná dekorátorem @ManyToOne zajišťuje vztah N:1. Zde se jedná o nejvyšší jednotku, pod kterou tvůrce lokality spadal. Vlastnost eager pak zajišťuje její načtení ihned.

8. version

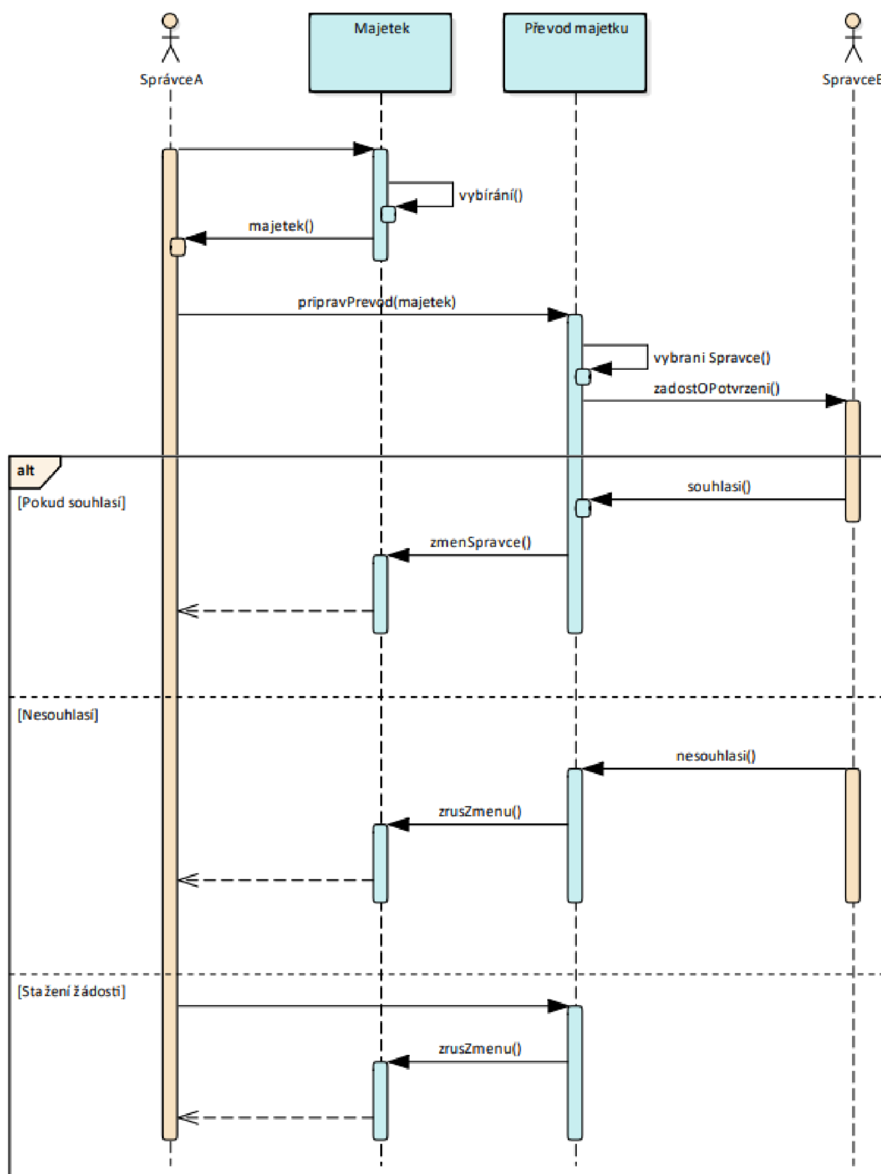
   Jedná se o sloupec reprezentující verzi záznamu v tabulce. S každým uložením záznamu se toto číslo zvyšuje. Tento sloupec může být užitečný v případě řešení konfliktů při změně záznamů.

V rámci implementace lokací byl taky řešen problém při odstraňování lokace, kdy je třeba nejdříve zkontrolovat, zda se v některé z lokacích nacházejících se v lokaci odstraňované, nenachází majetek. Pokud ne, tak nejdříve odstranit lokace, které jsou následovníky a až poté odstranit lokaci požadovanou.

4.2 Převod majetku mezi správci

Tato kapitola je věnovaná rozšíření, kdy jeden ze správců chce část spravovaného majetku přesunout pod správce jiného.

Přesun majetku je možné realizovat až po odsouhlasení obou stran. Zároveň je možné žádost stáhnout či zamítnout.



Obrázek 26, sekvenční diagram přesunu majetku, vlastní tvorba

Jak je vidět na Obrázek 26, tak zdánlivě banální problém může mít tři různé scénáře. První scénář uvažuje se situací, kdy obě strany souhlasí a majetek se následně převede ze SprávceA na SprávceB.

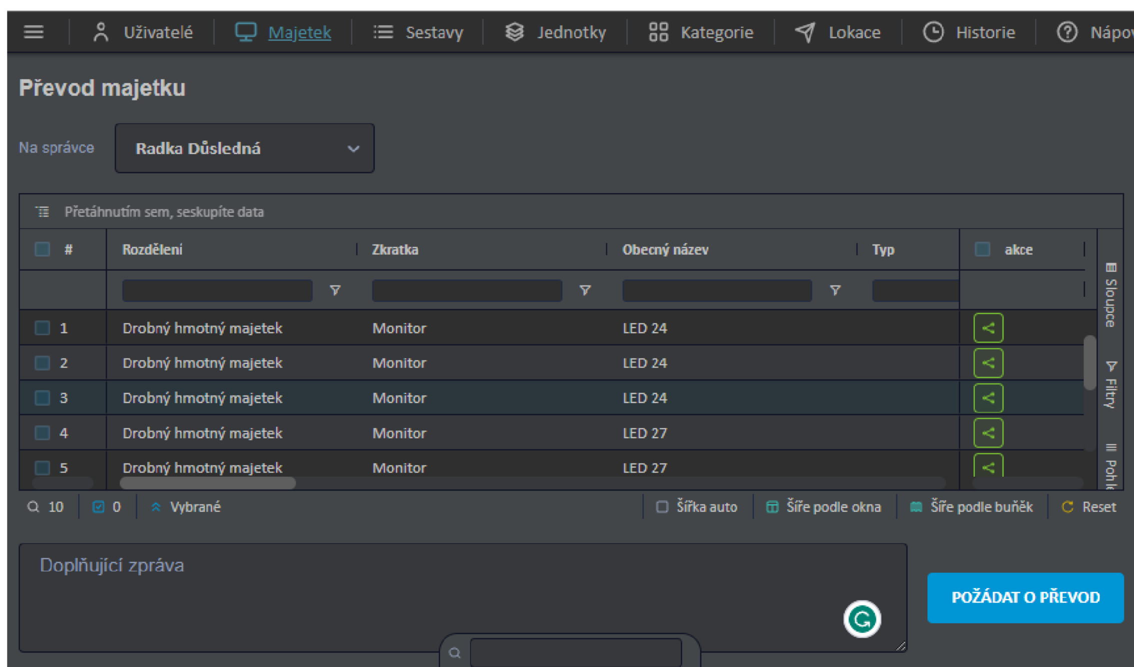
Druhý scénář uvažuje situaci, kdy SprávceB nesouhlasí se zaslaným převodem a převod zamítá. Poslední scénář uvažuje situaci, kdy SprávceA si své rozhodnutí

rozmyslel a chce jej stáhnout. V obou případech majetek zůstává pod správou SprávceA.

4.2.1 Implementace převod majetku - Angular

Pro přidání převodu majetku na klientské straně vznikla nová sekce v majetku, která obsahuje přehled převodů a tvorbu žádosti o převod.

Správce si v majetku vyhledá položky, které chce nechat převést. V následujícím kroku označí správce, na kterého chce majetek převést, a případně doplní zprávu, jak je vidět na Obrázek 27.

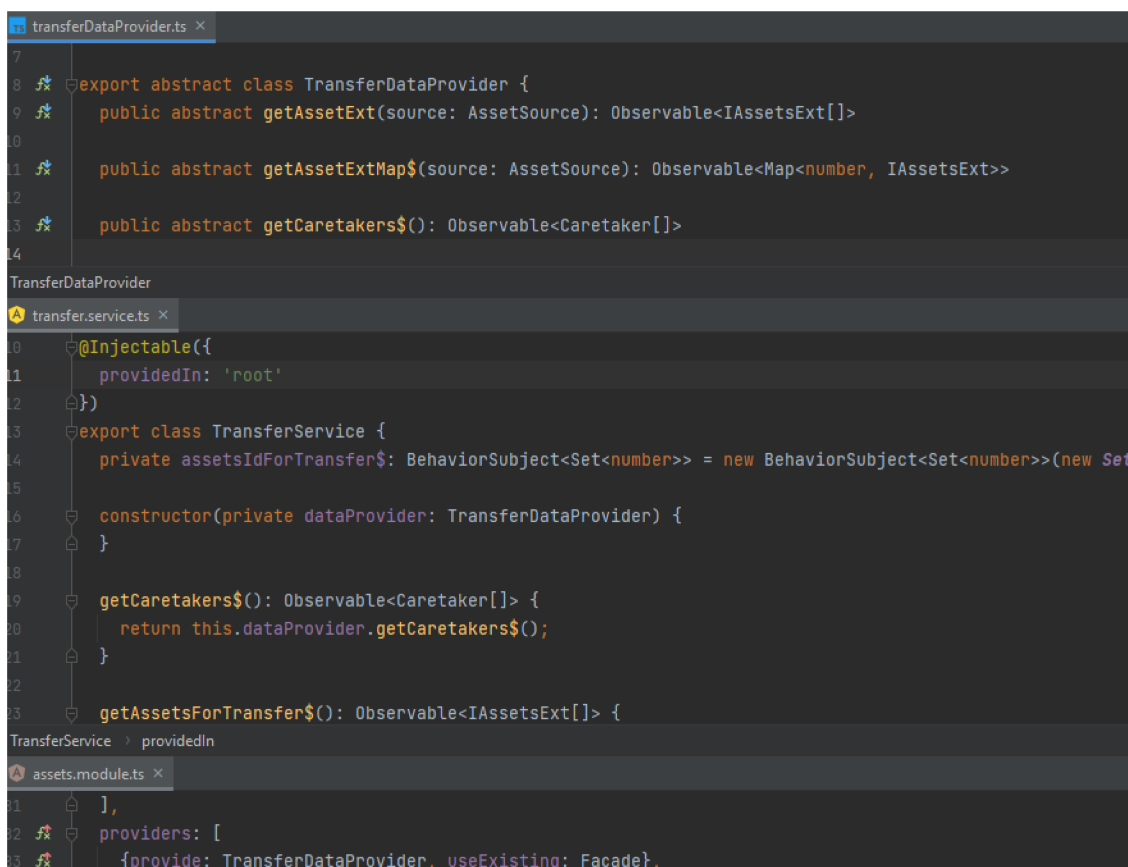


Obrázek 27, žádost o převod, vlastní tvorba

V rámci tohoto rozšíření byly přidány tři komponenty (tvorba, detail a seznam) a služba na které je ukázána Dependency Injection přes rozhraní.

Jak je vidět na Obrázek 28, v Angularu je možné tohoto způsobu docílit tak, že se místo rozhraní využije abstraktní třída. V našem případě se jedná o třídu TransferDataProvider. Tu je možné využít v konstruktoru komponenty nebo služby. V našem případě se jedná o službu TransferService. Další krok, který je potřeba udělat je nastavit providera v modulu pomocí objektu, kde atribut „provide“

odkazuje na předpis třídy (TransferDataProvider) a následuje rozhodnutí, zda je vyžadována nová instance tohoto providera (použijeme atribut useClass) či chceme již vytvořenou (atribut useExisting) a jako hodnotu vložíme třídu, která bude tyto metody zajišťovat. Poslední krok, který je ještě třeba udělat je implementovat rozhraní v takto zvolené třídě. Zde tedy ve třídě „Facade“.



Obrázek 28, DI - interface, vlastní tvorba

4.2.2 Implementace převod majetku – NestJS

Pro implementaci na straně serveru bylo nutné vytvořit koncové body na získání seznamu převodů, detailu převodu a endpoint na provedení akce. V rámci tohoto endpointu byl použit filtr tak, aby bylo možné získávat seznamy dle určitých kritérií, např. všechny převody pro správceA, dle stavu, a ty, ve kterých se nachází majetek s požadovaným ID.

4.2.2.1 Koncové body

Metoda	Url	Popis
Get	assets/transfers	Získání pole všech převodů na základě specifikace parametrů
Get	assets/transfers/:uuid	Detail převodu majetku s ID: uuid
Post	assets/transfers/:uuid/actions/:action	Vyvolání akce (potvrzení, stažení, vrácení a zamítnutí) k převodu majetku s ID: uuid

V rámci jednotlivých akcí se kontroluje, zda se uživatel nachází v roli správce, zda jsou shodné jednotky pro požadovaný převod a zda se již majetek nenachází v jiném aktivním převodu.

Pro získávání dat na koncovém bodu seznamu převodů majetku je možné využít specifikaci parametrů, kterým je výsledný seznam filtrován.

Načítání těchto dat z TypeORM je řešeno pomocí „queryBuilderu". Tento způsob dovoluje řetězit dotaz a přidávat tak jednoduše další podmínky do dotazu.

Nejdříve je třeba injektovat repositář pro převody (assetTransferRepository), to je provedeno v konstruktoru třídy. Následně můžeme z tohoto repositáře voláním metody „createQueryBuilder", query builder inicializovat, jak je vidět na Obrázek 29.

Následuje metoda leftJoin, která provede spojení tabulek. První uvedená hodnota odkazuje na definovaný vztah, případně cizí klíč, kterým je možno identifikovat záznam z tabulky, která je uvedena jako druhá hodnota této funkce.

Aby bylo možné vyhledávat pomocí ID majetku, je přidána funkce addSelect, která přidá do výběru ID majetku a umožní tím i jeho filtraci.

Za zmínku také stojí podmínka, která přidává omezení správce, od kterého vzešli požadavky. Zde je v metodě „andWhere" specifikován atribut, který je vybrán a

pomocí „@>" přiřazen do proměnné „:caretakerFrom", který je dále v metodě specifikován. Tím je možné docílit i vyhledávání v typu jsonb.

```
async getAssetTransferList(assetTransferQuery: AssetTransferQuery) {
  const { caretakerFrom, assetId, rejected, accepted, reverted } =
    assetTransferQuery;
  const query = await this.assetTransfersRepository.createQueryBuilder(
    alias: 'asset_transfers',
  );
  query.leftJoin( property: 'asset_transfers.assets',  alias: 'assets');
  query.addSelect( selection: 'assets.id');

  if (caretakerFrom) {
    query.andWhere( where: 'asset_transfers.caretakerFrom @> :caretakerFrom',  parameters: {
      caretakerFrom: {
        id: caretakerFrom,
      },
    });
  }

  if (assetId) {
    query.andWhere( where: 'assets.id = :assetId',  parameters: { assetId });
  }

  if (rejected !== undefined) {
    if (rejected) {
      query.andWhere( where: 'asset_transfers.rejectedAt IS NOT NULL');
    } else {
      query.andWhere( where: 'asset_transfers.rejectedAt IS NULL');
    }
  }
```

Obrázek 29, ukázka queryBuilderu, vlastní tvorba

4.2.2.2 Definice entity – Převod majetku

Převod majetku je v TypeORM reprezentován entitou, která obsahuje údaje o správcích, mezi kterými je převod uskutečněn, seznamem majetku, doplňující zprávou a datem, kterým byla akce realizována. Převod majetku obsahuje tyto atributy:

1. uuid

   Primární klíč generovaný pomocí uuid

2. caretakerFrom, caretakerTo

   Tyto atributy reprezentují správce majetku, mezi kterými je převod uskutečněn. Aby bylo docíleno zachování informace o správci a jednotky v danou chvíli

převodu, tak je správce uložen jako objekt obsahující: ID uživatele, jméno, příjmení a ID jednotky. Tento sloupec je definován jako typ „jsonb“. Díky tomu jsme pak schopni se dotazovat i na atributy uložené v této struktuře, jak je znázorněno na Obrázek 30

3. <u>createdAt, revertedAt, acceptedAt, rejectedAt</u>
   Jedná se o atributy, které reprezentují datum provedení akce.

4. <u>message</u>
   Tento atribut reprezentuje případnou doplňující zprávu.

```
@Entity( name: 'asset_transfers')
export class AssetTransfersEntity extends BaseEntity {
  @PrimaryGeneratedColumn( strategy: 'uuid')
  uuid: string;

  @Column( options: 'jsonb')
  caretakerFrom: Caretaker;

  @Column( options: 'jsonb')
  caretakerTo: Caretaker;

  @ManyToMany( typeFunctionOrTarget: () => Assets)
  @JoinTable()
  assets: Assets[];

  @CreateDateColumn()
  createdAt: Date;

  @Column( options: { nullable: true })
  revertedAt: Date;

  @Column( options: { nullable: true })
  acceptedAt: Date;

  @Column( options: { nullable: true })
  rejectedAt: Date;

  @Column( options: { nullable: true })
  message: string;
}
```

Obrázek 30, entita pro převod, vlastní tvorba

4.3 Identifikace majetku

Aby bylo možné využít automatizované procesy pomocí mobilního zařízení je potřeba majetek označit. Pro logické odlišení úkonů od skenu / vstupu do místnosti a majetku byla zvolena rozdílná technologie. V případě označení majetku se jedná o použití QR kódů.

4.3.1 Implementace identifikace majetku – Angular

Aby bylo možné tisknout z klientské aplikace QR kódy, byl použit balíček „angularx-qrcode“ (Jacob, 2023). Díky tomuto balíčku je možné z označených položek tisknout níže uvedené QR kódy. Každý kód obsahuje pouze ID majetku. (Obrázek 31)

Obrázek 31, výsledný QR kód

Takto označený majetek je pak možné použít v různých scénářích, které jsou podrobněji rozebrány v sekci zabývající se tvorbou mobilní aplikace.

4.4 Mobilní aplikace

Největší rozšíření je věnováno mobilní aplikaci, která je tvořena pomocí NativeScriptu a Angularu. Dělení této kapitoly je rozděleno podle dílčích funkcí, které aplikace nabízí. Ty jsou nejdříve popsané a pokud mají přesah i do jiné části aplikace, tak jsou uvedeny i implementace na straně klienta či serveru.

4.4.1 Zajištění požadavků aplikace

V této kapitole je popsáno, jakým způsobem jsou zajištěny požadavky na aplikaci. Každému požadavku je věnován vlastní bod.

4.4.1.1 Připojení

Pro připojení k oddělení síti je možné využít buď wifi router nebo ethernetový kabel. V rámci vývoje byla aplikace testována ve verzi pro zařízení se systémem Android na telefonu Samsung A20. K síti se zařízení připojovalo pomocí USB-C hubu od firmy Ugreen (Obrázek 322).

Obrázek 32, Ugreen USB-C hub, zdroj: https://www.alza.cz/ugreen-usb-c-hub-3x-usb-a-2-0-1x-ethernet-d6126479.htm

4.4.1.2 Sken majetku

K zajištění požadavku na sken QR kódů byl využit balíček: „nativescript-barcodescanner“ (Verbruggen, 2023a). Jedná se o balíček, který ke své funkci vyžaduje přístup ke kameře. V případě spuštění se dotáže o přístup k právům. Balíček zprostředkovává funkce pro sken různých typů čárových kódů, nejen QR kódů.

Balíček je možné využít jako komponentu přímo na view nebo jako třídu v aplikační části. V rámci projektu byl využit typ použití pomocí třídy.

Třídu je možné nastavovat různými parametry, především typ skenovaného kódu, a poskytuje metodu, která se zavolá po ukončení skenovacího procesu.

4.4.1.3 Využití NFC senzoru

Pro využití NFC senzoru a zajištění čtení NFC štítků byl využit balíček „nativescript-nfc“ (Verbruggen, 2023b). Tento balíček je možné použít jako třídu v aplikační části. Při spuštění kontroluje, jestli je NFC na zařízení přítomno a povoleno.

Pro účel označení místností byly použity NFC štítky typu NTAG213 a ULTRALIGHT. Na NFC štítky je možné zapisovat i vlastní údaje, ale v případě této aplikace není nutné zapisovat jakékoli údaje. Plně postačí identifikační údaj, který nám poskytne samotný NFC štítek.

4.4.1.4 Persistence Dat

K plnohodnotnému využití aplikace je potřeba, aby data zůstávala v mobilním zařízení i po vypnutí aplikace nebo případném restartu zařízení.

V NativeScriptu je možné uchovávat data třemi různými způsoby. Buď do dat aplikace, souboru či pomocí SQLite databáze.

1. <u>Uložení do dat aplikace</u>

   K zápisu dat do aplikace je třeba využít objektu ApplicationSettings (dále appSettings), který se nachází v „@nativescript/core“. Takto importovaný objekt je možný využít podobně jako localStorage nebo Redis. Do appSettings je možné ukládat data v podobě klíče: hodnota v řetězcové podobě. Kromě ukládání a načítání umožňuje dotazovat se na existenci klíče, získání seznamu všech klíčů či smazání veškerých záznamů.

2. <u>Uložení dat do souboru</u>

   Pro zápis dat do souboru nabízí NativeScript třídu FileSystemAccess. Pro získání třídy je třeba zavolat funkci getFileAccess(). Metoda se nachází taktéž v „@nativescript/core“. Tato metoda vrací instanci třídy FileSystemAccess ve formě singletonu. Ta poskytuje řadu metod, jak pro čtení a zápis dat do souboru, tak i pro práci s adresáři.

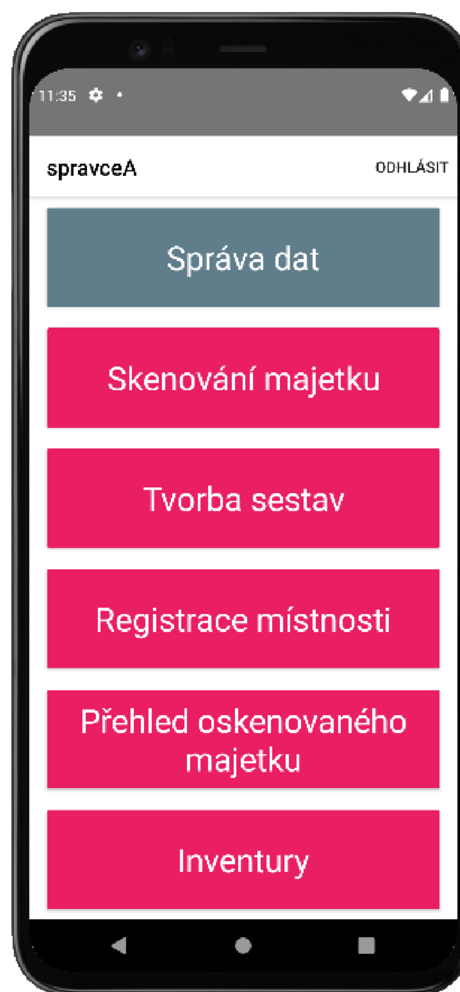
3. <u>Uložení do databáze SQLite</u>

   Pro uložení dat do databáze je třeba doinstalovat balíček: „nativescript-sqlite“. Balíček nabízí komplexní řešení pro ukládání dat a hodí se především pro aplikace, které existují jako samostatné celky. Balíček je nabízen pod free a komerční licencí, která nabízí více podpůrných funkcí (Anderson, 2023).

V mobilní aplikaci řešené v této práci se pracuje především s objekty, proto byla zvolena varianta 1 (Uložení dat do aplikace). Způsob uložení je popsán u každé části zvlášť.

4.4.2 Implementace funkcí mobilní aplikace

Tato sekce je věnována implementacím jednotlivých funkcí v mobilní aplikaci. V každé funkci je popsán její scénář, implementace a uchování dat. Každá funkce má svoji vlastní sekci v rámci aplikace, jak je vidět na hlavní straně mobilní aplikace viz. Obrázek 33.



Obrázek 33, ukázka základního rozhraní mobilní aplikace, vlastní tvorba

4.4.2.1 Přihlašování uživatelů

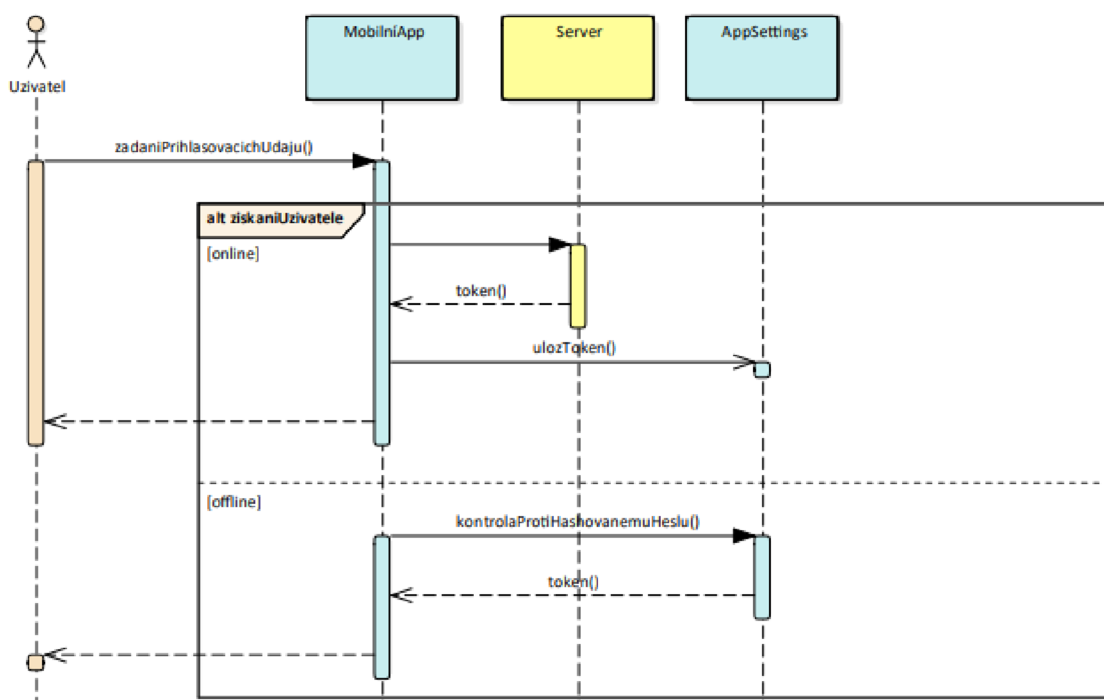
Tato funkce je věnována přihlašování a autentizaci uživatelů na mobilním zařízení.

4.4.2.1.1 Scénář

Pro identifikaci uživatele, který se zařízením pracuje, je potřeba, aby se uživatel přihlásil a k požadavkům na serveru odesílal identitu. Aby bylo možné využít již existující koncové body je využito JwtTokenů, stejně jako v případě klientské části určené pro PC, tvořené Angularem.

Pro přihlašování do zařízení v offline formě je potřeba, aby se uživatel prvně přihlásil, když je zařízení připojeno k síti. Tím je zajištěno, že systém uživatele zná a poskytne mu JwtToken (viz. Obrázek 34).

Následně je uložen token společně s uživatelským heslem, které je zašifrováno. Pokud se uživatel snaží přihlásit v offline formě, tak je kontrolováno vložné heslo a přihlašovací jméno oproti zahešované formě, která je uložena v AppSettings.



Obrázek 34, ukázka přihlášení, vlastní tvorba

Pro autentizaci uživatele slouží třída AuthInterceptor, která ke každému požadavku přidá token, který získá ze třídy auth.service.ts

4.4.2.1.2 **Implementace**

V této sekci jsou popsány klíčové prvky implementace přihlašování uživatelů. Triviální prvky, jako je zadávání dat pomocí textových polí, jsou vynechány.

1. <u>Uložení tokenu v rámci aplikace</u>

   Token je v rámci aplikace poskytován pomocí služby auth.service.ts, která má pouze jednu instanci. To je zajištěno nastavením atributu provideIn: „root“ v dekorátoru @Injectable() nad touto třídou (viz. Obrázek 35).

   Pro uchovávání tokenu, byla zvolena třída BehaviorSubject z balíčku RxJS. Dále byla ve službě vytvořena metoda, která vrací hodnotu atributu „accessToken“ jako Observable.

```
@Injectable({
  providedIn: 'root'
})
export class AuthService {
  token$: BehaviorSubject<{ accessToken: string, jwtToken: JwtToken } | null> =
    new BehaviorSubject<{ accessToken: string, jwtToken: JwtToken } | null>(null);

  constructor(private httpClient: HttpClient) {
  }

  accessToken$(): Observable<string | null> {
    return this.token$.asObservable().pipe(map(obj => obj?.accessToken ?? null))
  }
```

Obrázek 35, uchování tokenu ve službě, vlastní tvorba

2. <u>Autentizace uživatele</u>

   Autentizace je řešena pomocí JwtTokenů zasílaných spolu s požadavky. Toto je zajištěno pomocí interceptoru: Auth.interceptor.ts, což je JavaScriptová třída, která implementuje rozhraní HttpInterceptor. Musí tedy obsahovat metodu intercept, která má přístup k požadavku. Ten upraví a společně s dotazem odešle hlavičku s tokenem, jak je vidět na Obrázek 36.

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return this.authService.accessToken$().pipe(
    switchMap(accessToken => {
      if (accessToken) {
        const cloned = req.clone({
          headers: req.headers
            .set('Authorization', `Bearer ${accessToken}`)
        });
        return next.handle(cloned);
      } else {
        return next.handle(req);
```

Obrázek 36, metoda intercept v auth.interceptor.ts

3. <u>Uložení uživatele a tokenu pro offline použití</u>

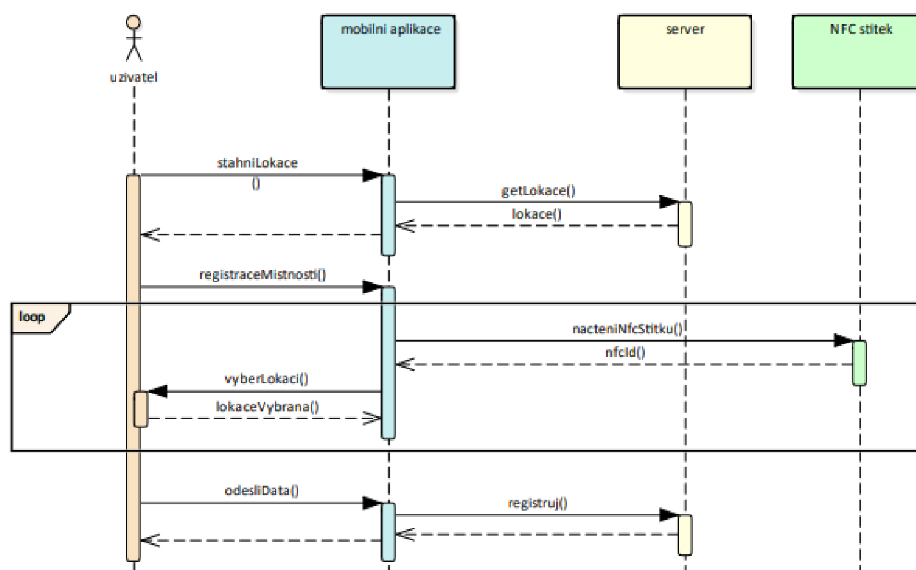
   Aby bylo možné přihlášení uživatele i v offline režimu je potřeba uložit heslo a token uživatele v rámci aplikace. To je zajištěno voláním metody při úspěšném přihlášení v auth.service.ts. Metoda uloží do úložiště aplikace token s uživatelským jménem a zahešovaným heslem, to je pak porovnáváno oproti hešované verzi zadaného hesla.

4.4.2.2 Registrace lokací

Funkce registrace lokací byla vytvořena z důvodu eliminace chyby při opisování identifikačního údaje z NFC štítku při párování s místností.

4.4.2.2.1 **Scénář**

Aby bylo možné k lokacím registrovat NFC štítky, je potřeba mít seznam lokací, ke kterým je možné štítky přiřadit. Data musí být prvně nahrána do aplikace. Následuje výběr funkce, která uživatele vyzve k naskenování NFC štítku a poté uživatel vybere ze seznamu lokací ke které bude štítek přiřazen. Takto může přiřadit libovolné množství štítků k lokacím. Jakmile úkon skončí, připojí zařízení k síti a odešle data ke zpracování na server, který registraci dokončí. Průběh je vidět na Obrázek 37.

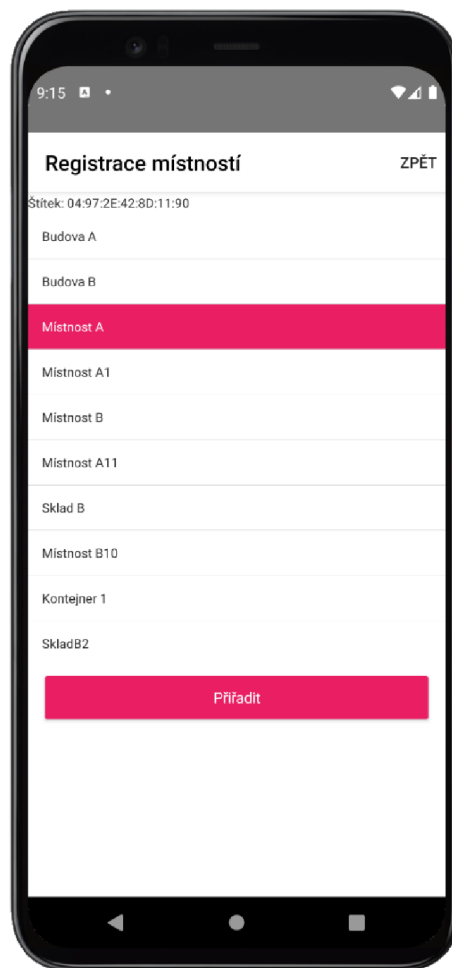


Obrázek 37, průběh registrace místnosti, vlastní tvorba

4.4.2.2.2 **Implementace**

Aby bylo možné načítat NFC štítky, bylo využito balíčku „nativescript-nfc“. Kromě kontroly funkce NFC, zápisu dat na štítky a výmazu dat balíček také obsahuje dvě metody pro čtení NFC štítů:

1. setOnNdefDiscoveredListener
   Tato metoda se zavolá v případě, že je zařízení přiloženo ke štítku, co obsahuje data.
2. setOnTagDiscoveredListener
   Tato metoda je naopak volána v případě, kdy je zařízení přiloženo ke štítku, který je prázdný



Obrázek 38, ukázka přiřazení štítku k lokaci, vlastní tvorba

To, že je štítek prázdný neznamená, že neposkytuje svoje unikátní identifikační číslo. Po naskenování uživatel vybere lokaci, ke které bude štítek přiřazen, viz. Obrázek 38.

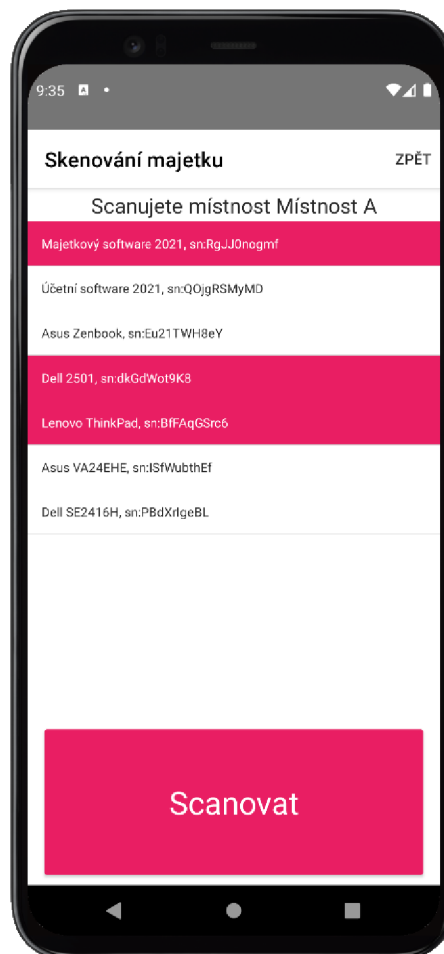
Pro dokončení celého procesu je potřeba mobilní zařízení připojit k síti a data odeslat metodou patch na server v podobě objektu obsahujícího jeden údaj, nfcId. Koncový bod se nachází na adrese „locations/:uuid/nfc“. (4.1.2.1 Koncové body)

4.4.2.3 Aktualizace umístění majetku

V této sekci je popsána implementace funkce k aktualizaci majetku. Pro její funkčnost bylo potřeba vytvořit i koncové body na serveru.

4.4.2.3.1 **Scénář**

Tato funkce slouží k průběžné změně umístění majetku. Správce nahraje data do mobilní aplikace. Při příchodu do místnosti si pomocí NFC štítku umístěného u vstupu načte místnost, kterou chce skenovat. Aplikace mu zobrazí, jaký majetek by se měl v lokaci nacházet. Při skenování položek, které by se měli nacházet v jiné oblasti, aplikace uživatele upozorní a vyzve k rozhodnutí, zda chce umístění v systému změnit či nikoli. Veškeré naskenované položky pak označuje podbarvením (Obrázek 39).



Obrázek 39, ukázka označení naskenovaného majetku, vlastní tvorba

Pro ukončení skenování lokace uživatel opět přiloží zařízení k NFC štítku nebo klikne na tlačítko zpět.

Poslední krok spočívá v nahrání dat na server.

4.4.2.3.2 **Implementace**

K implementaci této funkce bylo potřeba využít balíčků pro čtení NFC štítků i QR kódů.

Při skenování NFC štítku je mobilní aplikaci poskytnut identifikátor NFC štítku. Aplikace zjistí, jestli není ve stavu, kdy skenuje místnost, a zda skenovaný údaj zná. Pokud ano nastavuje místnost ke skenování.

Díky Observables zobrazuje uživateli majetek, který by se měl v oblasti nacházet. Výsledný Observable je získán pomocí filtrace položek majetku (viz. Obrázek 40), které jsou uložené v BehaviorSubjectu.

```
getItemsForRoom(uuid: string): Observable<AssetModel[]> {
  return this._items$.asObservable()
    .pipe(
      map( project: assets => assets.filter(i =>
        (i.locationOld && i.locationOld.uuid === uuid && i.locationConfirmed === undefined)
        || i.locationConfirmed?.uuid === uuid
      )))
}
```

Obrázek 40, ukázka map operátoru, vlastní tvorba

Persistence dat je řešena obdobně jako v případě uživatelů, kdy se při každé změně stavu data zapisují i do ApplicationSettings (Obrázek 41).

```
private saveLocallyItems(items: AssetModel[]) {
  ApplicationSettings.setString('items', JSON.stringify(items))
}

private loadLocallyItems(): AssetModel[] {
  const items = ApplicationSettings.getString( key: 'items')
  if (items) {
    return JSON.parse(items);
  }
  return []
}
```

Obrázek 41, ukázka zápisu a čtení z ApplicationSettings, vlastní tvorba

Opačně jsou pak při startu aplikace data z ApplicationSettings načtena a vložena do BehaviorSubjectu.

Posledním krokem je odeslání dat na server. Ty se odesílají v podobě pole, které obsahuje objekty s identifikátory majetku a lokace, ve které byl majetek nalezen. Server na základě těchto údajů provede změnu u umístění majetků.

4.4.2.4 Tvorba sestav majetku

V této sekci je popsána funkce tvorby sestav majetku. V systému pro evidenci majetku je možnost si vytvářet vlastní sestavy majetku, které je možné využít v budoucnu pro rychlejší práci, zejména pro hromadné úpravy, přehledy či pro

tvorbu dokumentů. V rámci tohoto rozšíření přibyla tato možnost i na mobilní zařízení.

4.4.2.4.1 **Scénář**

Uživatel se přihlásí do aplikace, vybere možnost tvorby sestav. Zde vidí seznam již vytvořených sestav, a má možnost si vytvořit i novou. Po vybrání sestavy může skenováním QR kódů přidávat majetek do sestavy. Majetek je možné smazat, stejně tak i sestavy. Po odeslání dat na server jsou uživateli přeneseny sestavy do hlavní aplikace.

4.4.2.4.2 **Implementace**

Pro tuto funkci bylo potřeba využít balíčku pro skenování QR kódů tak, aby bylo možné identifikovat majetek. Sestavy jsou ukládány v BehaviorSubjectu ve službě workingList, která se stará o změnu stavů sestav a poskytuje Observables pro rozšíření komponenty pro seznam a detail sestavy majetku.

Pro nahrání sestav do hlavní aplikace je důležité data odeslat na server. Na základě tokenu, který je zaslán spolu s požadavkem na vytvoření sestav se sestavy vytvoří k uživatelskému účtu.

4.4.2.5 Provádění inventur

Funkce provádění inventur zahrnovala úpravy na všech částech systému. V této sekci je zmíněn nejprve scénář pro tuto akci, následovaný implementacemi na jednotlivých částech aplikace.

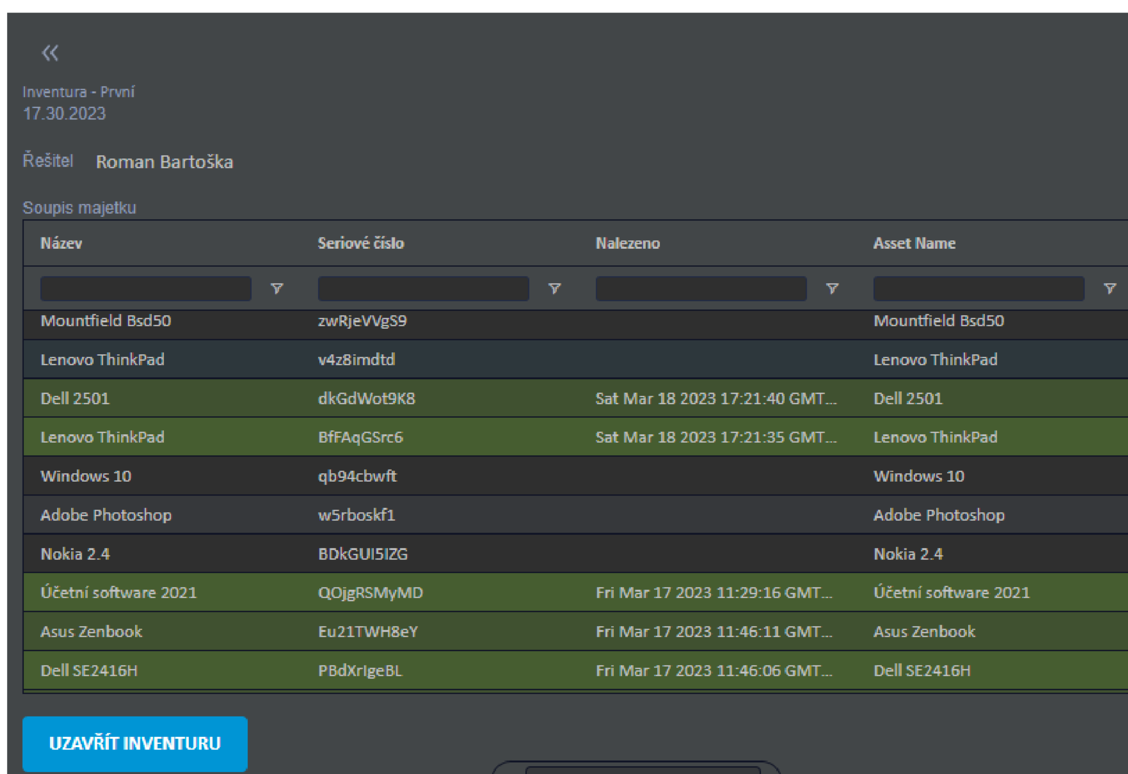
4.4.2.5.1 **Scénář**

Aby bylo možné provádět inventuru, tak uživatel s rolí správce vybere osobu, která bude tuto inventuru provádět a odešle požadavek na server. Server inventuru vytvoří a při připojení aplikace, ve které je uživatel s rolí řešitel, inventuru zpřístupní. Ta je následně uložena do mobilní aplikace. Její provádění pak funguje obdobně jako skenování majetku v lokacích.

Uživatel může kdykoli v průběhu inventury data aktualizovat se serverem až do doby, než je inventura uzavřena.

4.4.2.5.2 **Implementace – Angular**

V rámci implementace na klientské straně určené pro PC, bylo potřeba umožnit správcům zakládat inventurní proces. Byly vytvořeny tři komponenty: seznam inventur, detail inventury a tvorba nové inventury.



Obrázek 42, ukázka detailu inventury, vlastní tvorba

Seznam inventur je zobrazen pomocí knihovny ag-grid, stejně je řešen i seznam majetku, který je dohledáván v rámci inventury. Položky, které jsou nalezené jsou označeny zelenou barvou (viz. Obrázek 42). Správce má kdykoli možnost inventuru uzavřít, a tím znemožnit jakékoli dohrávání dat.

4.4.2.5.3 **Implementace – NestJS**

V rámci inventurního procesu bylo potřeba na serveru vytvořit dvě nové entity a šest nových koncových bodů.

4.4.2.5.3.1 Koncové body

Metoda	Url	Popis
Get	assets/stock-taking	Získání pole všech inventur
Post	assets/stock-taking	Vytvoření nové inventury
Get	assets/stock-taking/:uuid	Detail inventury s ID: uuid
Post	assets/stock-taking/:uuid/close	Uzavření inventury
Get	assets/stock-taking-in-progress	Získání aktivních inventur pro uživatele
Patch	assets/stock-taking-in-progress	Aktualizace aktivních inventur pro uživatele

4.4.2.5.3.2 Definice entity

Aby bylo možné provádět inventurní proces bylo potřeba vytvořit dvě nové entity:

1. Stock-taking

   Jedná se o entitu reprezentující inventuru. Entita obsahuje referenci na řešitele, autora, název, identifikátor, datum vytvoření, uzavření a vazbu 1:N na entitu „Stock-taking-item".

2. Stock-taking-item

   Entita stock-taking-item reprezentuje dohledávanou položku v inventurním procesu. Obsahuje identifilkátor, referenci na majetek, kterého se týká, referenci na Stock-taking, položku, kdy a v jaké lokaci byla nalezena a případné pole pro poznámku.

4.4.2.5.4 Implementace – NativeScript

Aby bylo možné provádět inventuru pomocí mobilní aplikace, bylo potřeba vytvořit komponentu pro seznam a pro detail inventury, dále pak službu, která se bude starat o změny stavů položek.

V rámci této funkce byly použity balíčky pro NFC štítky a QR kódy. Implementace této funkce je řešena obdobně jako u skenování majetku v místnostech. Sken se ovšem neukládá k majetku, ale k položce inventury. Po připojení zařízení k síti má uživatel kdykoli možnost data odeslat na server, případně si je doplnit nebo stáhnout verzi dat, která je na serveru. V rámci odesílání dat má uživatel možnost si vybrat, jaká data na server odešle.

4.4.3 Testování mobilní aplikace

V rámci vývoje mobilní aplikace byl proveden test a zkoumalo se, zda využití takovéto aplikace zrychlí průběh inventur.

4.4.3.1 Průběh testu

Bylo vytvořeno 100 ks QR kódů reprezentujících majetek a vloženo do 5ti obálek, na kterých byl NFC kód. Obálky reprezentovali lokace.

Test probíhal ve dvou fázích. Při prvním testování majetek v obálkách odpovídal umístění, ve kterém se měl nacházet. Při druhém testu byly vyndány z každé obálky čtyři kusy majetku, dány na společnou hromádku a z té vybrány náhodně 4 kusy, které se vrátily do každé obálky zpět. Test byl proveden na třech subjektech rozdílných věkových kategorií, pohlaví i vzdělaní. Každá osoba provedla 4 druhy dohledávání majetku a to:

1) Nalezení majetku ručně bez přehození
2) Nalezení majetku s aplikací bez přehození
3) Nalezení majetku ručně s přehozením
4) Nalezení majetku s aplikací s přehozením

Nalezení majetku ručně odpovídá situaci, kdy osoba musí majetek dohledávat v soupisu majetku určeného pro danou místnost v papírové podobě.

Sledovala se doba, po jakou trvalo dohledání majetku, ve výše uvedených hledáních, a zda se data, která byla aplikací odeslána na server shodovala s reálným umístění QR kódů.

4.4.3.2 Výsledky testování

měření	Subjekt 1	Subjekt 2	Subjekt 3
1)	15:43	13:51	11:25
2)	7:54	5:41	7:42
3)	23:27	39:33	17:30
4)	6:00	5:38	7:37

4.4.3.3 Zhodnocení výsledků

Z výsledků je patrné, že využití aplikace několikanásobně zrychlí průběh evidence, obzvláště, pokud je třeba majetek dohledávat. Po konci testování byl majetek z aplikace odeslán na server a porovnán se stavem, který byl odškrtán a poznamenán. Stavy mezi aplikací a papírovou podobou byly shodné. Můžeme tedy konstatovat, že využití takovéto aplikace usnadní evidenci, narovnání umístění majetku, a jak je patrné z měření 2) a 4), tak téměř anuluje rozdíl mezi osobou, která evidenci provádí, a eliminuje problémy s dohledáváním majetku.

5 Závěr

Během této práce byly definovány rozšíření pro systém evidence majetku, který byl vytvořen v bakalářské práci, na kterou tato práce navazuje.

Okrajově byly zmíněny technologie, které jsou podrobněji popsány v bakalářské práci. Tato práce se více zaměřuje na framework NativeScript v použití s frameworkem Angular, odlišnosti oproti běžnému použití a na vysvětlení prvků, které souvisí s velice využívanou knihovnou RxJS a TypeORM.

Každému rozšíření je věnována sekce, popisující scénář, kterému odpovídá daná funkce a implementace na konkrétních částech systému.

Během testování aplikace se došlo k závěru, že využití podobné aplikace eliminuje nejen chybu lidského faktoru, ale také rapidně snižuje čas strávený při vykonávání procesů, při dohledávání majetku.

Výsledkem práce je systém pro evidenci majetku, který je rozšířen o mobilní aplikaci. Ten umožňuje automatizovat řadu procesů, které by bylo jinak potřeba řešit ručně. Bez použití informačních technologií by mohly vznikat chyby a zmíněné úkony by byly časově náročnější.

Systém je možné provozovat na oddělené síti bez přístupu k internetu, a to včetně mobilní aplikace, kterou je možné k systému připojit pomocí mobilního zařízení, přes ethernetový kabel, případně wifi-router.

6 Seznam zdrojů

6.1 Knižní zdroje

BIØRN-HANSEN, Andreas; MAJCHRZAK, Tim A.; GRØNLI, Tor-Morten. Progressive web apps: The possible web-native unifier for mobile development. In: *International Conference on Web Information Systems and Technologies*. SciTePress, 2017. p. 344-351.

LIM, Greg. Beginning Angular 2 with TypeScript, CreateSpace Independent Publishing Platform

6.2 Internetové zdroje

ANDERSON Nathanel, NativeScript-sqlite. [online]. Market.nativescript.org, 2023 [cit. 2023-04-11]. Dostupné z: https://market.nativescript.org/plugins/nativescript-sqlite/

BOURG, Lorena, Thomas CHATZIDIMITRIS, Ioannis CHATZIGIANNAKIS, et al. Enhancing shopping experiences in smart retailing. *Journal of Ambient Intelligence and Humanized Computing* [online]. [cit. 2023-03-19]. ISSN 1868-5137. Dostupné z: doi:10.1007/s12652-020-02774-6

GEGECKIENĖ, Laura, Ingrida VENYTĖ, Justina KARPAVIČĖ, Torben TAMBO, Kęstutis VAITASIUS a Darius PAULIUKAITIS. Near field communication (NFC) technology in the packaging industry. In: *Proceedings - The Eleventh International Symposium GRID 2022* [online]. University of Novi Sad, Faculty of technical sciences, Department of graphic engineering and design, 2022, 2022-11-3, s. 495-501 [cit. 2023-03-19]. ISBN 9788660225339. Dostupné z: doi:10.24867/GRID-2022-p54

MANIMUTHU, Arunmozhi, Venugopal DHARSHINI, Ioannis ZOGRAFOPOULOS, M. K. PRIYAN a Charalambos KONSTANTINOU. Contactless Technologies for Smart Cities: Big Data, IoT, and Cloud Infrastructures. *SN Computer Science* [online]. 2021, 2(4) [cit. 2023-03-19]. ISSN 2662-995X. Dostupné z: doi:10.1007/s42979-021-00719-0

GOOGLE. Angular [online]. Silicon Valley: Google, 2010-2023 [cit. 2023-04-18]. Dostupné z: https://angular.io/guide/template-overview

JACOB, Andreas. Angularx-qrcode [online]. Github.com, 2023 [cit. 2023-04-18]. Dostupné z: https://github.com/cordobo/angularx-qrcode

KHUDOIBERDIEV Umed. Eager and Lazy Relations [online]. Typeorm.io 2023 [cit. 2023-04-11]. Dostupné z: https://typeorm.io/eager-and-lazy-relations

LESH, Ben, RxJS - operators list. [online]. *Rxjs.dev,* 2023a [cit. 2023-04-11]. Dostupné z: https://rxjs.dev/guide/operators

LESH, Ben, RxJS – forkJoin operator. [online]. *Rxjs.dev,* 2023b [cit. 2023-04-11]. Dostupné z: https://rxjs.dev/api/index/function/forkJoin

OPENJS FOUNDATION, NativeScript layout. [online]. Nativescript.org, 2023a [cit. 2023-04-11]. Dostupné z: https://docs.nativescript.org/ui-and-styling.html

OPENJS FOUNDATION, NativeScript styling. [online]. Nativescript.org, 2023b [cit. 2023-04-11]. Dostupné z: https://docs.nativescript.org/ui-and-styling.html

REGLI, Theresa. The state of digital asset management: An executive summary of CMS Watch's Digital Asset Management Report. *Journal of Digital Asset Management* [online]. 2009, 5(1), 21-26 [cit. 2023-03-22]. ISSN 1743-6559. Dostupné z: doi:10.1057/dam.2008.49

SAPTARI, Asep Yusup, S. HENDRIATININGSIH, Dony BAGASKARA a Levana APRIANI. IMPLEMENTATION OF GOVERNMENT ASSET MANAGEMENT USING TERRESTRIAL LASER SCANNER (TLS) AS PART OF BUILDING INFORMATION MODELLING (BIM). *IIUM Engineering Journal* [online]. 2019, 20(1), 49-69 [cit. 2023-03-19]. ISSN 2289-7860. Dostupné z: doi:10.31436/iiumej.v20i1.987

VERBRUGGEN, Eddy. NativeScript-barcodescanner [online]. Market.nativescript.org, 2023a [cit. 2023-04-11]. Dostupné z: https://market.nativescript.org/plugins/nativescript-barcodescanner/

VERBRUGGEN, Eddy. NativeScript-nfc [online]. Github.com, 2023b [cit. 2023-04-11]. Dostupné z: https://github.com/EddyVerbruggen/nativescript-nfc

VIDHYALAKSHMI, R. a Vikas KUMAR. Design comparison of traditional application and SaaS. In: *2014 International Conference on Computing for Sustainable Global Development (INDIACom)* [online]. IEEE, 2014, 2014, s. 541-544 [cit. 2023-03-22]. ISBN 978-93-80544-12-0. Dostupné z: doi:10.1109/IndiaCom.2014.6828017

ZDERIC, Mario [online]. Zagreb, Chorvatsko: DECODE HQ d.o.o., 2021 [cit. 2022-04-21]. Dostupné z: https://decode.agency/article/mobile-app-programming-languages/

7 Přílohy

7.1 Příloha č. 1 – elektronická příloha

Obsah elektronické přílohy (source.rar), CD-ROM

- zdrojové kódy pro Angular (složka frontend)
- zdrojové kódy pro NestJS (složka rest)
- zdrojové kódy pro NativeScript (složka mobile-app)

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. Milan Knop**
Studium:	I2100434
Studijní program:	N1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název diplomové práce:	**Rozšíření systému pro evidenci majetku ve frameworku Angular**
Název diplomové práce AJ:	Extending the Feature Assets Management System in Angular framework

Cíl, metody, literatura, předpoklady:

Cíl: Rozšíření existujícího systému o přesuny majetku mezi jednotlivými správci a správu lokací, tvorbu QR kódů a mobilní aplikace. Pomocí mobilní aplikace bude možné skenovat QR kódy majetku a načítat umístění čtecího zařízení pomocí NFC štítků, registrace místností (párování NFC štítků a lokace) a dále tvorbu pracovních sestav přímo v terénu. Mobilní aplikace bude vytvořena frameworkem NativeScript-Angular. Aplikace budou provozovány na uzavřené síti, bez přístupu na internet. Data v tomto případě budou do mobilní aplikace načteny a po provedení akce (např. části inventury) synchronizovány se systémem po připojení k síti. V případě změn systém umožní uživateli automatizované přesuny majetku mezi lokacemi, případné žádosti o převod na jiného správce majetku.

Osnova:

1. Úvod
2. Analýza
3. Návrh a implementace aplikace
    1. na straně klienta ve frameworku Angular,
    2. na straně serveru ve frameworku NestJS a platformě node.js
    3. na straně mobilní aplikace ve frameworku NativeScript
4. Výsledky a doporučení
5. Závěr

https://angular.io/

https://nestjs.com/

https://nativescript.org/

https://www.ag-grid.com/

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Pavel Kříž, Ph.D.

Datum zadání závěrečné práce: 26.1.2021