Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Vývoj mobilnej aplikácie pre iOS

Bakalárska práca

2022 **Martin Varga**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Vývoj mobilnej aplikácie pre iOS

Bakalárska práca

Študijný program:	Inteligentné systémy
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra kybernetiky a umelej inteligencie (KKUI)
Školiteľ:	Ing. Erik Kajáti, PhD.

Košice 2022 **Martin Varga**

Abstrakt v SJ

Bakalárska práca opisuje vývoj aplikácie pre iOS. V práci sú taktiež opísané najpopulárnejšie zdravotné databázy, ich výhody, nevýhody a porovnanie. Čo sa týka vývoja aplikácie pre iOS tak je priblížené a podrobne popísané vývojové prostredie Xcode. V práci sa takisto opisuje programovací jazyk Swift, ktorý je najpoužívanejším programovacím jazykom pre vývoj iOS aplikácií. Ďalej sa venujeme návrhu riešenia, všetky návrhy v tejto práci sú vytvorené pomocou dizajnového nástroja Figma. Následne je opísaný vývoj konkrétneho riešenia v ktorom užívateľ môže nové zdravotné dáta pridávať, zobraziť za zadané obdobie a synchronizovať s lokálnou databázou podľa potreby. V práci sú časti kódu ohľadom týchto funkcionalít vysvetlené.

Kľúčové slová v SJ

iOS, aplikácia, programovanie, Swift, Xcode, zdravie, synchronizácia

Abstrakt v AJ

The bachelor thesis describes the development of an iOS application. The thesis also describes the most popular health databases, their advantages, disadvantages and compares them. As far as iOS app development is concerned, the Xcode development environment is described in detail. The thesis also describes the Swift programming language, which is the most widely used programming language for iOS application development. Next we discuss the design of the solution, all the designs in this thesis are created with the help of Figma design tool. Then the development of a specific solution is described in which the user can add new health data, view data from a specified period of time and synchronize it with the local database as required. In the thesis, parts of the code regarding these functionalities are explained.

Kľúčové slová v AJ

iOS, application, programming, Swift, Xcode, health, synchronization

Bibliografická citácia

VARGA, Martin. *Vývoj mobilnej aplikácie pre iOS*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2022. 60s. Vedúci práce: Ing. Erik Kajáti, PhD.

66008

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra kybernetiky a umelej inteligencie

ZADANIE BAKALÁRSKEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Inteligentné systémy**

Názov práce:

Vývoj mobilnej aplikácie pre iOS

iOS Mobile Application Development

Študent: **Martin Varga**
Školiteľ: **Ing. Erik Kajáti, PhD.**
Školiace pracovisko: **Katedra kybernetiky a umelej inteligencie**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie bakalárskej práce:

1. Vypracovať prehľad použiteľných zdravotných databáz pre iOS.
2. Návrh aplikácie pre synchronizáciu dát medzi zvolenou zdravotnou databázou a externým úložiskom.
3. Realizácia zvolenej aplikácie.
4. Zhodnotenie výsledkov vytvoreného riešenia a súhrn získaných poznatkov.
5. Vypracovať dokumentáciu podľa pokynov vedúceho práce (hlavná časť minimálne 40 strán, užívateľská a systémová príručka).

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 27.05.2022
Dátum zadania bakalárskej práce: 29.10.2021

.....
prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 27.5.2022 .....

Vlastnoručný podpis

Poďakovanie

Na tomto mieste by som rád poďakoval svojmu vedúcemu práce Ing. Erikovi Kajátimu, PhD za jeho čas, cenné rady, pripomienky a usmernenie pri písaní tejto bakalárskej práce.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

V dnešnej dobe veľká väčšina ľudí vlastní a pravidelne používa nejaký druh mobilného zariadenia. V týchto zariadeniach máme nainštalované rôzne aplikácie, ktoré nás dokážu zabaviť, a tak nám spríjemňujú čas alebo nám svojou funkcionalitou pomáhajú a uľahčujú život. Aplikácie sú naozaj pestré v tom, čo ponúkajú, a v tom ako vyzerajú z grafického hľadiska. Žiadne medze sa pre vývojárov a grafických dizajnérov nekladú. S pribúdajúcim počtom senzorov v mobilných zariadeniach, a taktiež s pribúdajúcim počtom zariadení, ktoré vieme s mobilným zariadením prepojiť sa otvárajú stále nové možnosti pre vývoj mobilných aplikácií.

Jedným z hlavných typov aplikácií a zároveň pre nás najviac užitočným sú aplikácie, ktoré sa venujú zdraviu a pracujú so zdravotnými dátami, ktoré my zadáme alebo o nás rôzne zariadenia dokážu zbierať. Najlepším príkladom takého zariadenia sú inteligentné hodinky. Tie o nás dokážu zbierať informácie ako sú srdcový tep alebo krvný tlak, a to všetko nahrávajú do prepojeného mobilného zariadenia v reálnom čase ak je dané zariadenie v dosahu.

Existuje veľké množstvo aplikácií, ktoré sa venujú práci so zdravotnými dátami. Väčšinou tieto aplikácie ponúkajú možnosť nové dáta pridať alebo sledovať a analyzovať určité zdravotné dáta za určité obdobie. Ná základe tejto analýzy následne ponúkajú nejaký záver pre užívateľa ako napríklad zvýšenie intenzity pri tréningu alebo odporúčanie pre prijímanie viac tekutín. Len malý počet aplikácií tohto typu ponúka možnosť nejakého typu synchronizácie s databázou, a ak už aj ponúkajú, tak užívateľ nemá na výber s akou databázou by chcel dané údaje zosynchronizovať, ale je nútený sa prispôsobiť danej aplikácii.

Cieľom našej práce je vytvoriť aplikáciu, ktorá práve takúto možnosť synchronizácie podľa výberu užívateľa ponúka a zároveň umožniť užívateľovi mať o svojich zdravotných dátach perfektný prehľad a podľa potreby údaje o sebe pridať do svojho zariadenia, a takýmto spôsobom mať všetko prehľadne na jednom mieste.

1 Formulácia úlohy

- Zdravotné databázy - v tejto kapitole budeme riešiť to aké zdravotné databázy sú dostupné pre iOS zariadenia. Najpopulárnejšie popíšeme a porovnáme medzi sebou, spomenieme ich výhody alebo nevýhody a čo je ich hlavným cieľom.

- Vývoj aplikácií pre systém iOS - v rámci tejto kapitoly budeme riešiť možnosti vývoja aplikácie pre tento operačný systém. Popíšeme si vývojové prostredie Xcode, jeho možnosti, ako funguje a jeho konkrétne časti a na čo slúžia. V krátkosti spomenieme čo to je Xamarin. Následne sa už v tejto kapitole budeme venovať najpopulárnejšiemu jazyku pre vývoj iOS aplikácií, ktorým je Swift a jeho fungovanie a syntax si popíšeme do hĺbky. Taktiež si spomenieme SwiftUI čo je relatívne nový *framework* na tvorbu užívateľského rozhrania.

- Návrh riešenia - V tejto kapitole popíšeme ako prebiehal návrh nášho riešenia z pohľadu užívateľského rozhrania. Priblížime si to akým vývojom návrh prebiehal od prvotného návrhu až po finálny a odôvodníme si prečo a aké zmeny boli vykonané.

- Implementácia riešenia - V tejto kapitole opíšeme proces pri vývoji aplikácie a to od nainštalovania prostredia až po vytvorenie finálneho produktu. Je tu spomenutých niekoľko problémov na ktoré sme v rámci implementovania riešenia narazili a ako sme ich vyriešili. Taktiež tu je spomenutá štruktúra projektu a popis toho, ktorá trieda alebo súbor sa čomu venuje a popis niektorých základných funkcionalít napísaného kódu.

- Vyhodnotenie - V tejto kapitole sa budeme venovať hlavne vykonaným meraniam a zhodnoteniu výsledkov ku ktorým sme vďaka meraniam došli.

2 Zdravotné databázy

V dnešnej dobe dokážeme pomocou zariadení, ktoré používame každý deň zbierať ohromné množstvo dát ohľadom nášho zdravia. Medzi takéto zariadenia patria mobily alebo inteligentné hodinky. Tieto hodinky môžu byt zosynchronizované s mobilom, ktorý používame, a to nám umožní zbierať dáta ako náš krvný tlak a srdcový tep. Taktiež mobil dokáže samostatne zbierať dáta napríklad o krokoch, ktoré sme vykonali vďaka zabudovaným senzorom. Samozrejme dokážeme zbierať aj mnoho iných dát, ale tieto patria medzi najzákladnejšie.

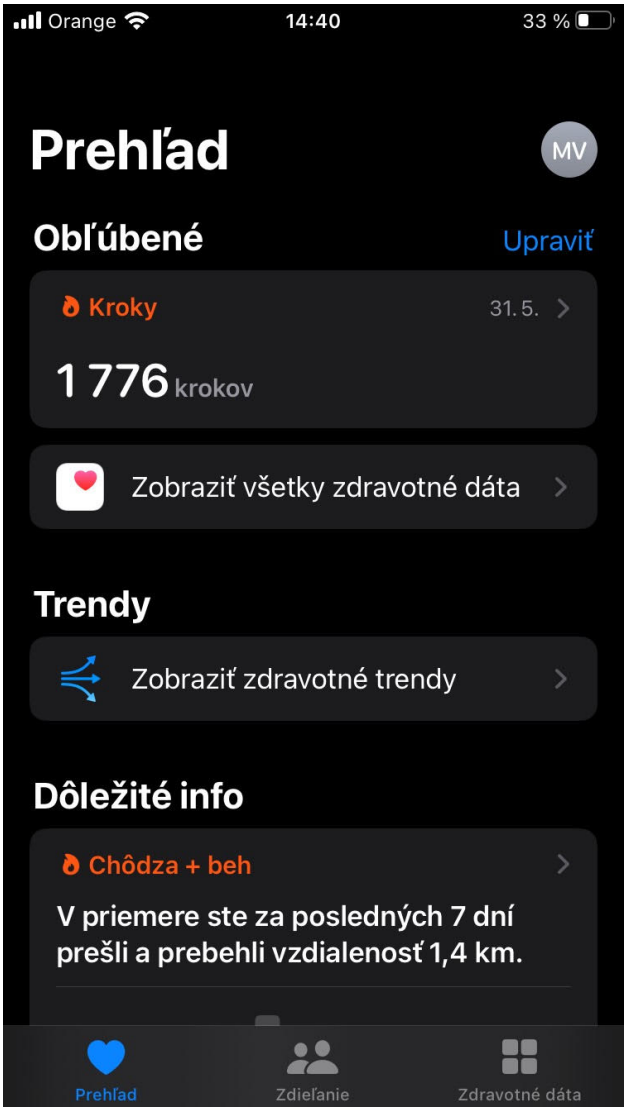
Niekedy keď senzory nestačia, tak existuje aj možnosť do našich zariadení dáta o nás pridať manuálne. Medzi takéto dáta napríklad patrí množstvo prijatých tekutín, a na to už slúžia rôzne aplikácie.

Tieto dáta o našom zdraví sú veľmi citlivé, keďže sa jedná o osobné informácie a veľa ľudí nepovažuje za komfortné tieto dáta zdieľať s kýmkoľvek, a práve preto sú tieto dáta šifrované. Prístup k týmto dátam majú len ľudia a aplikácie, ktorým to povolíme [1][2].

V tejto kapitole sa budeme venovať tomu, aké sú možnosti zdieľania, zálohovania a synchronizácie týchto šifrovaných a citlivých dát.

2.1 Apple Zdravie

Apple už dlhšiu dobu ponúka aplikáciu s názvom Zdravie. Jedná sa o aplikáciu, ktorá ponúka prehľad o všetkých zdravotných dátach zozbieraných z rôznych zdrojov ako napríklad fitnes doplnky rôzneho druhu alebo z aplikácií tretích strán, kde si užívateľ pridá nejaké dáta manuálne. Dá sa povedať, že táto aplikácia je takým jadrom pre zdravotné dáta v Apple zariadeniach. Dáta si môžeme jednoducho prezerať podľa potreby, ale najpodstatnejšie je to, že sú bezpečne uložené a zašifrované. Jednou z najpodstatnejších sekcií je sekcia aktivity. V tejto sekcii si môžeme prezrieť rôzne zaujímavé dáta o našej vykonanej aktivite ako o počte krokov, ktoré sme prešli alebo aký sme mali spánok. Vieme si tu pozrieť dennú aktivitu alebo aj aktivitu za určité obdobie podľa výberu. Ďalšou zaujímavou sekciou



Obr. 2.1: Apple Zdravie aplikácia

je jedlo a výživa. V tejto sekcii vieme manuálne pridať náš denný príjem potravy. To ale môže byť náročné a zdĺhavé, keďže musíme všetky podrobnosti zadať manuálne. Tento problém riešia aplikácie tretích strán, v ktorých sú rôzne databázy, ktoré už obsahujú najčastejšie jedlá a častokrát nám stačí vybrať niečo zo zoznamu ako predvoľbu, a to sa už zosynchronizuje s aplikáciou Zdravie. Manuálne zadávanie váhy alebo zosynchronizovanie s inteligentnou váhou je ďalšou zaujímavou vlastnosťou aplikácie Zdravie, dokáže to pomôcť ľudom sledovať si svoju váhu a upozorniť, ak napríklad niekto za krátky čas príliš veľa váhy priberie alebo stratí. S aktualizáciou iOS 13 prišla taktiež možnosť zadávať priamo do aplikácie Zdravia informácie o menštruačnom cykle. Pred touto aktualizáciou to bolo možné len cez aplikácie tretích strán. Keďže zdravotných údajov, ktoré môžeme zbierať je veľa, tak aplikácia Zdravie ponúka možnosť vybrať si najobľúbenejšie zdravotné dáta a tie potom máme v jednej časti prehľadne spolu. Jednou z najdôležitejších

funkcionalít tejto aplikácie je Medical ID, jedná sa o profil alebo teda zdravotnú kartu, ktorú si užívateľ musí sám nastaviť a povoliť. V prípade potreby je potom možné k týmto informáciám pristúpiť bez nutnosti odomknúť telefón a tieto zdravotné informácie o užívateľovi môžu častokrát zachrániť život daného užívateľa v prípade núdze. Donedávna táto aplikácia slúžila len na analýzu zdravotných dát, ich ukladanie a extrahovanie, ale s príchodom aktualizácie iOS 15 prišla možnosť tieto dáta zdieľať v rámci blízkeho kruhu príbuzných alebo s doktormi a rôznymi inými pracovníkmi v medicínskom odvetví. Všetko sa to dá nastaviť v rámci aplikácie. Pri nastavovaní toho s kým chceme zdravotné dáta zdieľať je aj možnosť vybrať, ktoré konkrétne dáta sme ochotní zdieľať s daným človekom. V prípade zdieľania s príbuznými to môže byť prospešné v tom, že si naši príbuzní môžu všimnúť nejaké malé zmeny v zdravotných dátach, ktoré generujeme a upozorniť nás na to alebo nás nejako podporiť podľa toho, o čo sa jedná. Táto funkcionalita je stále relatívne nová a veľa doktorov túto možnosť zdieľať s nimi zdravotné dáta zatiaľ neponúka [3][4].

2.1.1 iCloud

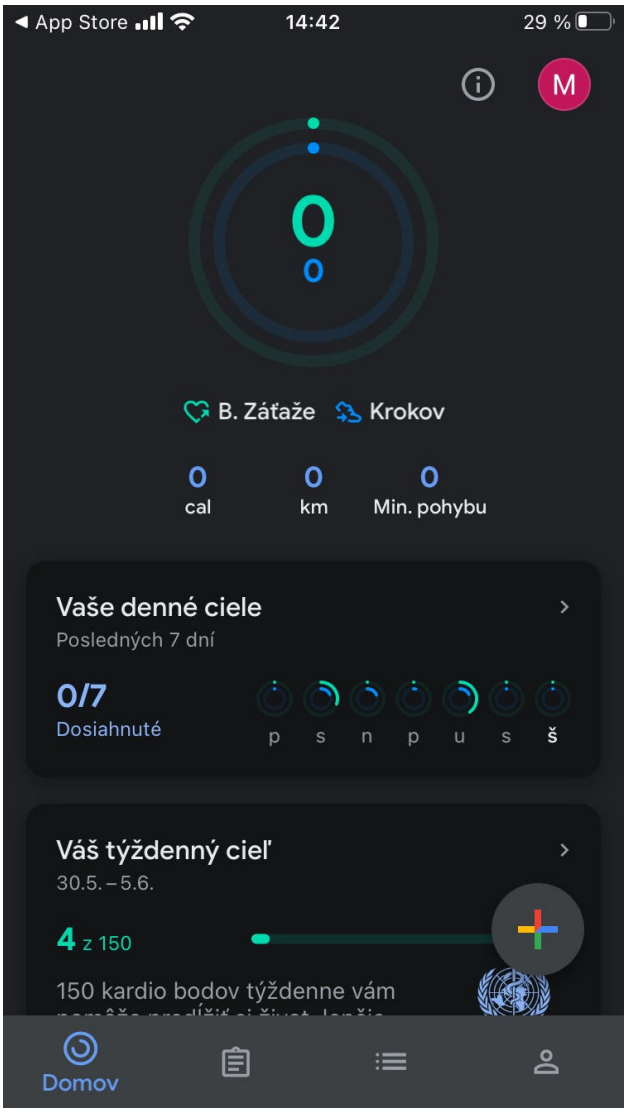
iCloud je cloudová služba od Applu pre používateľov iOS zariadení. iCloud je poskytnutý každému užívateľovi zadarmo s obmedzením na využitie 5 GB ukladacieho priestoru a možnosťou využitia len základných ponúkaných funkcií.

Keďže iCloud je služba priamo od Applu, tak zapnutie synchronizácie je veľmi jednoduché. Potrebujeme mať vytvorený Apple ID účet. Pomocou tohto účtu sa vieme prihlasiť do viacerých Apple zariadení, ktoré zbierajú zdravotné dáta a následne synchronizovať tieto dáta nie len s cloudom, ale aj medzi všetkými našimi zariadeniami. Všetko, čo je potrebné spraviť je prejsť do nastavení na danom zariadení, následne stlačiť Apple ID a potom zvoliť iCloud. Na obrazovke sa nám objavia možnosti, čo všetko môžeme synchronizovať na cloude a tu len zvolíme Zdravie. Tento spôsob synchronizácie a zálohovania je veľmi bezpečný keďže využíva *end-to-end* šifrovanie. K ďalším bezpečnostným prvkom patrí to, že táto možnosť synchronizácie je dostupná len pre ľudí, ktorí používajú dvojfaktorové overenie.

Ak teda chceme stiahnuť dáta, ktoré máme uložené na iCloude, tak budeme musieť poznať prihlasovacie meno, heslo, musíme zadať kód z dvojfaktorového overenia, ktorý nám dôjde v podobe SMS a ešte budeme musieť zadať prístupový kód. Takto na sebe navrstvené bezpečnostné prvky spôsobujú to, že naše dáta sú naozaj v bezpečí [5].

2.2 Google Fit

Google Fit je fitnes aplikácia od Googlu, ktorá je relatívne jednoduchá. Umožňuje nám zbierať a sledovať základné zdravotné dáta o nás na jednom mieste. Podporuje zber dát z rôznych populárnych zariadení ako sú inteligentné hodinky priamo od Googlu alebo ich Nest Hub 2, ktorý ponúka možnosť sledovať spánok za predpokladu, že je umiestnený napríklad na nočnom stolíku a má priamy výhľad na matrac na ktorom spíme. Taktiež sa spojili so spoločnosťou Polar, ktorá ponúka inteligentné hodinky pre sledovanie športových výkonov profesionálnych atlétov.



Obr. 2.2: Google Fit aplikácia

Pre používanie aplikácie je nutné prihlásiť sa do nej pomocou Google účtu. Následne môžeme vyplniť nejaké informácie o nás do profilu aplikácie ako napríklad výška a váha. Keďže aplikácia je relatívne jednoduchá tak navigovanie v

nej je celkom intuitívne. Hlavným cieľom tejto aplikácie je slúžiť ako fitnes aplikácia a podporovať nás v dosiahnutí našich športových cieľov, ktoré si stanovíme. To je jedným z hlavných rozdielov medzi touto aplikáciou a aplikáciou Zdravie od Applu a práve kvôli tomu niektoré zdravotné dáta nie sú dostupné v tejto aplikácii, ale v aplikácii Zdravie sú. Následkom toho je to, že Google Fit nie je certifikovaná ako lekárska aplikácia, ale len ako fitnes aplikácia a tým pádom bezpečnosť či ochrana dát, ktoré zbiera nespadajú pod tak striktné podmienky ohľadom bezpečnosti a šifrovania ako je to v prípade aplikácie Zdravie od Applu.

Jednou z hlavných výhod Google Fit aplikácie je integrácia Google kalendára. Priamo v Google kalendári si vieme pridať plánované cvičenie, aký typ cvičenia to je, a ako často sa má opakovať a to sa automaticky zosynchronizuje s Google Fit, kde následne po splnení cvičenia bude tento cieľ považovaný za splnený.

Budúcnosť tejto aplikácie je otázna, keďže Google v roku 2021 kúpil spoločnosť Fitbit a s ňou aj populárnu fitnes aplikáciu s rovnakým názvom. Zatiaľ tieto dve aplikácie koexistujú, ale vzniká otázka toho, či Google má potrebu pre dve aplikácie s podobnou či až priam rovnakou funkcionalitou [6][7].

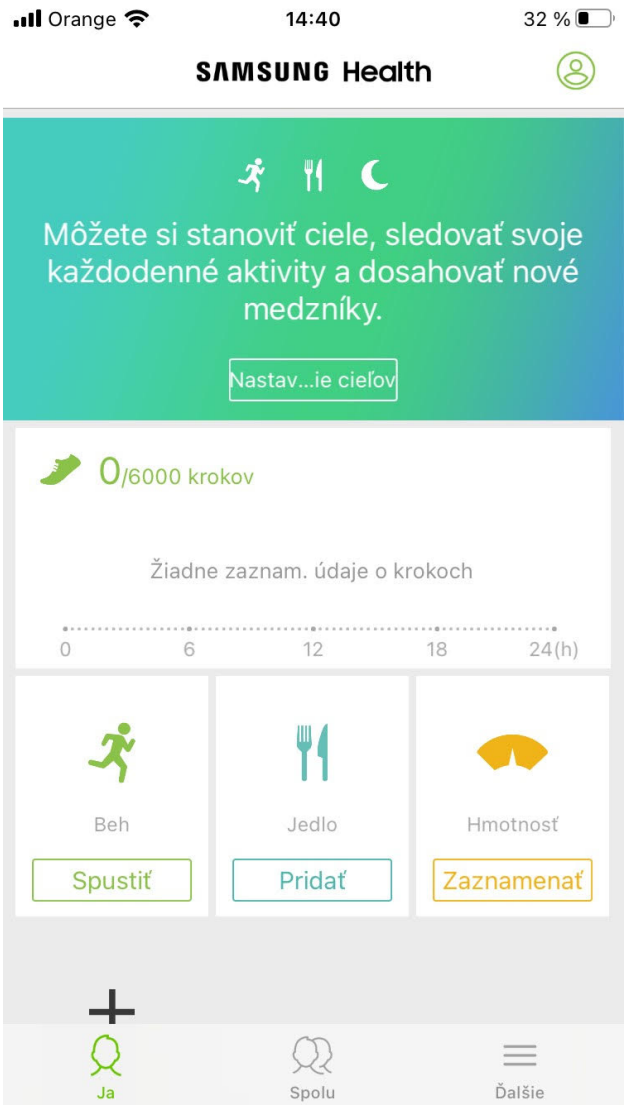
2.3 Samsung Health

Samsung Health je ďalšou aplikáciou, ktorá umožňuje sledovať a analyzovať rôzne zdravotné dáta ako počet krokov, typ cvičenia a jeho trvanie, príjem tekutín, jedlá a ich nutričné hodnoty, spánok, tlak, srdcový tep a rôzne iné. Táto aplikácia je vyvíjaná spoločnosťou Samsung. V dnešnej dobe využívajú mobilné zariadenia od Samsungu operačný systém Android, no aj napriek tomu je táto aplikácia taktiež dostupná na iOS zariadeniach, a to je hlavne kvôli tomu, že Samsung Health ponúka podporu pre synchronizáciu s rôznymi podporovanými zariadeniami. Hlavne sa jedná o ich vlastné produkty ako sú inteligentné hodinky Samsung Galaxy Watch a rôzne iné, ale taktiež podporujú inteligentné zariadenia od rôznych výrobcov ako napríklad SD Biosensor GlucoNavii, Polar alebo Garmin.

Pre čo najlepšie využitie tejto aplikácie sa musíme zaregistrovať a vytvoriť si Samsung účet. Po prihlásení si vieme do profilu zadať údaje o nás ako výška alebo váha. Na domovskej obrazovke môžeme nájsť všetky sledované údaje alebo manuálne pridať dáta podľa potreby. Môžeme tu taktiež spravovať aké dáta sa nám na domovskej obrazovke zobrazujú.

Veľkou súčasťou tejto fitnes aplikácie tak ako aj iných je podpora užívateľa v dosiahnutí cieľov, ktoré si sám určí alebo ktoré mu odporučí aplikácia. K tomuto prispieva možnosť spájať sa s ostatnými používateľmi Samsung Health. Po

ťuknutí na tlačidlo Spolu si môžeme pridať ostatných používateľov tejto aplikácie medzi ktorými môžu napríklad byť naši známi. Táto funkcionalita je jednou z podstatných výhod Samsung Health, určiť si podobné ciele s niekým koho poznáme a následne sa predbiehať k dosiahnutiu cieľa môže byť pre užívateľov tejto aplikácie extrémne motivujúce.



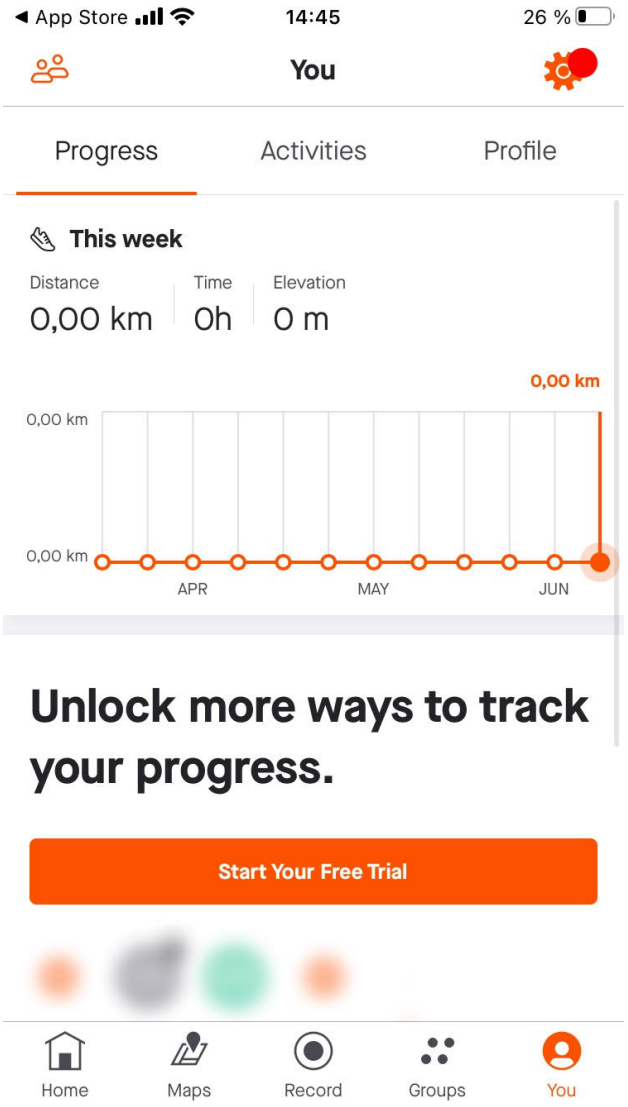
Obr. 2.3: Samsung Health aplikácia

Jednou z najväčších nevýhod aplikácie Samsung Health je to, že oproti dvom vyššie spomenutým aplikáciam táto aplikácia nepodporuje synchronizáciu s aplikáciami tretích strán až na pár výnimiek ako je Strava alebo Technogym [8][9].

2.4 Strava

Strava je platforma pre atlétov. Po stiahnutí aplikácie a zaregistrovaní sa v tejto platforme je užívateľovi ponúkaná možnosť sledovať a synchronizovať niektoré

vybrané dáta v databáze, ktorú ponúka Strava. Táto platforma využíva NoSQL databázu s názvom Apache Cassandra. Táto databáza je vhodná pre aktívne datasety a to dáta ohľadom zdravia sú, keďže neustále pribúdajú nové dáta [10].



Obr. 2.4: Strava aplikácia

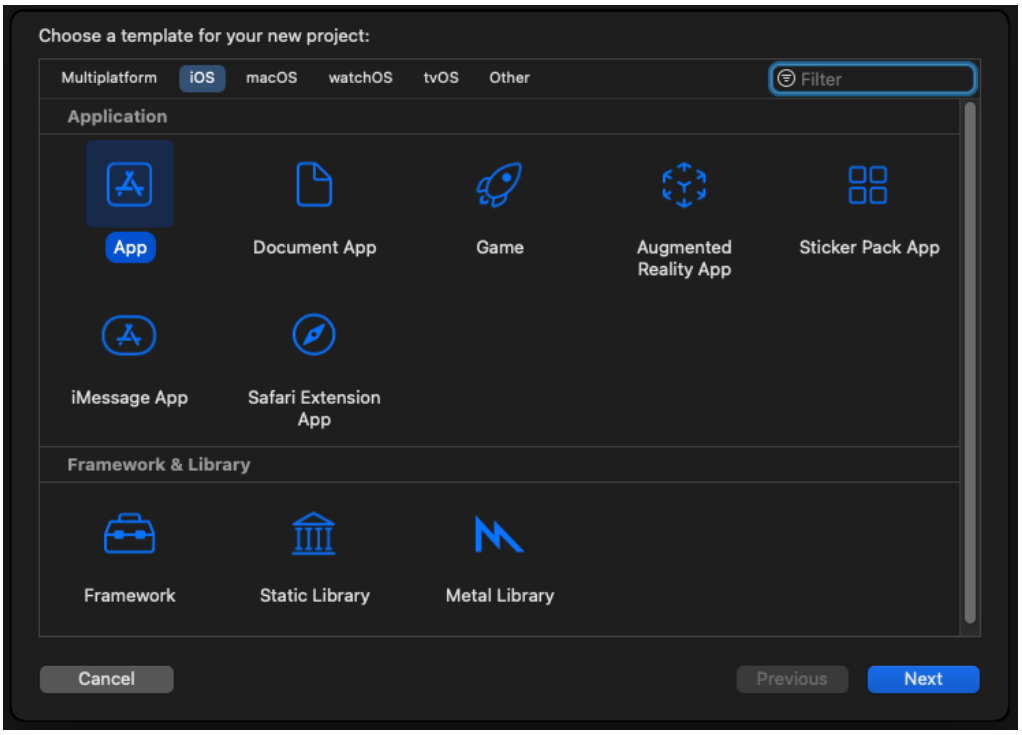
3 Vývoj aplikácií pre systém iOS

Vyvíjať aplikáciu pre zariadenia so systémom iOS je možné rôznymi spôsobmi. Najideálnejším je, ak máme zariadenie od spoločnosti Apple s macOS. Tieto zariadenia sú najlepšie optimalizované pre tento systém, ale taktiež existuje možnosť rozbehať tento systém s pomocou virtualizácie vo virtualizačných programoch ako je VMware Workstation Pro alebo VirtualBox. V takomto prípade je ale potrebné spĺňať minimálne požiadavky na rozbehnutie konkrétnej verzie macOS. AMD procesory majú v tomto určitú nevýhodu oproti Intel procesorom aj napriek rovnakej alebo lepšej výkonnosti, a to z dôvodu toho, že ešte do roku 2020 Apple využíval vo svojich zariadeniach hlavne Intel procesory. Apple na konci roka 2020 začal používať svoje vlastné procesory vo svojich laptopoch a počítačoch. macOS je potrebný z toho dôvodu, že Xcode, čo je vývojové prostredie pre vývoj iOS aplikácií, je exkluzívne dostupný zadarmo na AppStore. Pre stiahnutie tohto IDE(Integrated Development Environment) je nutné mať vytvorený Apple ID účet, následne sa pomocou svojho Apple ID prihlásime a jednoducho stiahneme.

3.1 Xcode

Xcode je IDE od Apple a kvôli tomu je primárne využívaný na vývoj iOS a macOS aplikácií v jazyku Swift. Dá sa ale využiť aj na vývoj aplikácií pre iné operačné systémy a taktiež podporuje veľa rôznych programovacích jazykov ako napríklad C, Objective-C, C++, Java a mnohé ďalšie. Obsahuje veľa *frameworkov* a prídavných nástrojov, ktoré sú potrebné pri vývoji aplikácií a kvôli tomu patrí medzi populárne vývojové prostredia. Ako už bolo spomenuté Xcode je zadarmo, ale ak chceme našu aplikáciu nahrať na AppStore, tak budeme potrebovať vývojárske konto za ktoré sa platí 99$ ročne. Aplikácie môžeme aj bez tohto konta vyvíjať bez obmedzenia, stačí nám obyčajné Apple ID konto. Vyvinuté aplikácie môžeme spustiť na simulátore, ktorý je integrovaný v Xcode alebo na vlastnom zariadení s potrebným operačným systémom [11].

3.1.1 Vytvorenie projektu

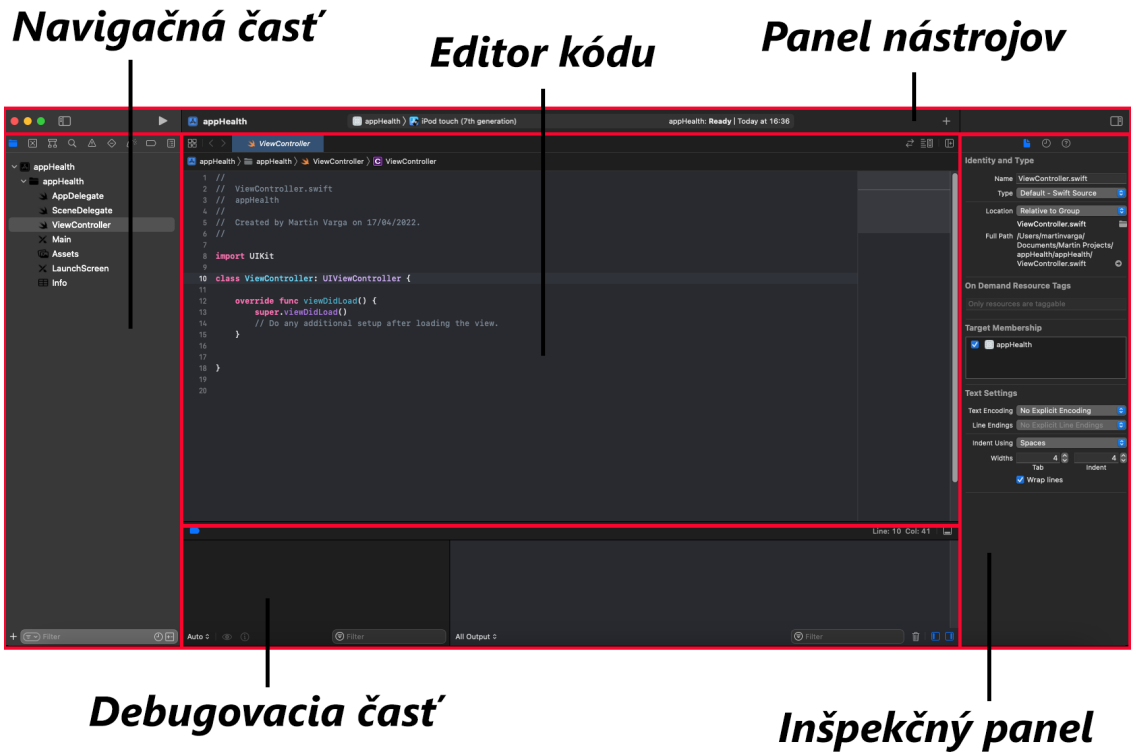


Obr. 3.1: Xcode vytvorenie projektu

Vytvorenie projektu spočíva v jednoduchom výbere šablóny a na základe tejto šablóny Xcode pripraví potrebné veci pre ďalší vývoj. Ďalej je potrebné zadať názov produktu, názov organizácie pod ktorou budete projekt vyvíjať a unikátny identifikátor organizácie. .Následne vyberieme z dostupných programovacích jazykov. Ak si vyberieme konkrétne aplikáciu pre iOS, tak ďalej máme možnosť výberu rozhrania, ktoré budeme používať. Na výber máme Storyboard alebo SwiftUI a v prípade jazykov to je medzi Swift a Objective-C. V ďalšom okne je potom možnosť vytvoriť Git repozitár pre správu verzií. Git repozitár nám umožňuje náš kód zálohovať v prípade, že sa niečo prihodí zariadeniu na ktorom aplikáciu vyvíjame. Taktiež nám umožňuje vidieť rozdiely medzi verziami spolu s komentármi, ktoré sme k zmene pridali [12].

3.1.2 Popis prostredia

Po otvorení projektu sa nám zobrazí takáto obrazovka, ktorú vieme rozdeliť na päť hlavných častí, a to panel nástrojov, navigačná časť, kód editor, debugovacia oblasť a inšpekčný panel. Každá z týchto častí má podstatnú úlohu pri vývoji aplikácií. V pravom hornom horu vieme nejaké tieto časti skryť, ak ich momentálne nepotrebujeme a prajeme si mať viac priestoru na písanie kódu.



Obr. 3.2: Xcode oblasti

Navigačná časť je rozdelená na deväť ďalších podčasti, ale takými najpodstatnejšími sú navigácia v rámci projektu, hľadanie a problémy. V navigácii projektu môžeme vidieť všetky súbory spojené s projektom. V tejto časti vieme tieto súbory organizovať, vytvárať nové priečinky, spájať existujúce súbory do skupín a to všetko s cieľom lepšej prehľadnosti projektu. Ak na nejaký súbor klikneme, tak sa nám okamžite otvorí v editore a tu ho vieme editovať podľa potreby. Napríklad ak klikneme na modrú ikonu projektu úplne hore, tak sa nám zobrazia vlastnosti projektu, kde vieme meniť meno projektu, ID, definovať najnižšie verzie operačného systému na ktorom aplikácia bude fungovať a pridať rôzne knižnice a *frameworky* využívané v aplikácii. Ďalšou podstatnou časťou v navigačnom panely je hľadanie, to sa hlavne využíva ak hľadáme nejakú konkrétnu funkciu, premennú alebo iné a kvôli rozsiahlosti projektu si nepamätáme, kde konkrétne to je. Navigátor pre problémy zobrazuje problémy, ktoré vznikli pri pokuse o zostavenie aplikácie. Pri kliknutí na konkrétny problém nás to presmeruje na miesto v kóde, kde tento problém vznikol. Taktiež zobrazuje varovania, ktoré síce nie je potrebné pre zostavenie a spustenie aplikácie vyriešiť, ale je dobré, ak ich poriešime pred tým ako aplikáciu skúsime nahrať na AppStore.

V editore kódu strávime najviac času ako programátor, a práve preto to je aj najpodstatnejšia časť. Samozrejmosťou je farebné odlíšenie funkcií od premen-

ných a taktiež kľúčových slov. Ako v každom editore aj tu je možnosť číslovania riadkov. Vo vrchnej časti editora máme celú cestu ku súboru, ktorý máme aktuálne otvorený. Tiež tu sú dve tlačidlá *dopredu* a *späť*, ktoré nám umožňujú rýchle prepínanie medzi súbormi, ktoré sme mali nedávno otvorené. Po stlačení *Command* tlačidla a kliknutí na určitú funkciu alebo triedu nás to presunie do konkrétnej časti kódu, kde je táto trieda alebo funkcia definovaná. Ku každému riadku kódu si vieme pridať *breakpoint*, čo je vlastne označenie daného riadku a ak pri exekúcií kódu dôjdeme k tomuto riadku, tak sa exekúcia pozastaví. *Breakpoint* je označený modrým indikátorom v ľavej časti daného riadku. Ak máme týchto *breakpointov* viac, tak si ich vieme všetky zobraziť v navigačnom panely s *breakpointmi* a ak sa nám ich nechce všetky manuálne odstraňovať, tak existuje globálne tlačidlo na zrušenie všetkých *breakpointov*. Asistenčný editor je vlastne druhý editor, ktorý sa nám zobrazí zároveň s tým pôvodným. Týmto spôsobom vieme editovať dva rôzne súbory v rovnakom čase alebo môžeme mať v jednom okne kód a v druhom simulátor, ktorý beží v reálnom čase nech vidíme zmeny, ktoré vykonáme v kóde hneď ako ich vykonáme, táto druhá možnosť je asi najpoužívanejšia.

Inšpekčný panel má rôzne možnosti zobrazenia detailov o súboroch, elementoch alebo časti kódu na ktorý sa pozeráme. Súborový inšpektor zobrazuje informácie ako konkrétne umiestnenie súboru a cestu k nemu a iné vlastnosti súboru. Inšpektor rýchlej pomoci zobrazuje dokumentáciu k metódam, triedam alebo kľúčovým slovám, ktoré sme vybrali, ak sme vybrali niečo k čomu dokumentácia nie je, tak táto časť bude prázdna. Inšpektor vlastností nám umožňuje meniť rôzne vlastnosti elementu, ktorý je momentálne vybraný.

Debugovacia časť nám zobrazuje konzolový výstup a status rôznych premenných ak aplikáciu rozbehneme napríklad v simulátore. Táto časť je podstatná, ak náš kód nefunguje podľa predstáv a potrebujeme pomoc s tým kde a prečo nastala chyba.

V panely nástrojov úplne hore máme dve tlačidlá a to na spustenie a zastavenie aplikácie. Xcode potrebuje vedieť, kde má našu aplikáciu vlastne spustiť a od toho slúži výber napravo od týchto tlačidiel. Tu si vieme zvoliť simulátor zariadenia podľa našej preferencie alebo ak máme pripojené vlastné zariadenie, tak je tu tiež možnosť aplikáciu spustiť na vlastnom zariadení. *Status bar* nám hovorí o stave Xcodu a čo v danom momente Xcode robí, taktiež nás v tejto časti upozorňuje ak sú s aplikáciou problémy [13][14].

3.2 Xamarin

Xamarin je vývojová platforma, ktorá umožňuje vývoj aplikácií pre iOS a Android prostredníctvom písania kódu v C#. Cieľom tejto platformy je to, aby sa aplikácia vyvinula len raz a nebolo ju potrebné vyvíjať dvakrát v dvoch rôznych prostrediach pre dva rôzne operačné systémy. Pod vrstvou Android a iOS sa nachádza *Mono runtime*. To je most medzi jazykom C# a natívnymi rozhraniami API(Application Programming Interface) systémov Android a iOS. To umožňuje prístup vývojárom k používateľskému rozhraniu Android a iOS, oznámeniam a všetkým funkciám obsiahnutým v mobilnom zariadení. To všetko s účelom vytvoriť vývojárske prostredie podobné tomu natívnemu. Xamarin využíva výhody *frameworku* .NET pre funkcie týkajúce sa dátových typov, generických typov, uvoľňovania zbytočne zaplnenej pamäte, jazykovo integrovaných dotazov, asynchrónnych programovacích vzorov a WCF(Windows Communication Foundation) komunikácie. Xamarin.Forms je multiplatformová UI(User Interface) knižnica, ktorá umožňuje vytvoriť používateľské rozhranie, zahŕňa v sebe aj možnosť využiť platformou špecifické vlastnosti ak to je potrebné [15][16].

3.3 Swift

Swift je kompilovaný programovací jazyk pre iOS, macOS, watchOS, tvOS a Linux aplikácie od spoločnosti Apple. Predovšetkým sa používa na vyvíjanie aplikácií pre Apple zariadenia, a to v kombinácií s Xcode vývojovým prostredím. Apple prvýkrát predstavil tento programovací jazyk na WWDC(Apple Worldwide Developers Conference) v roku 2014. S príchodom Swiftu, ktorý bol vyvinutý s účelom byť moderným a hlavne bezpečným programovacím jazykom, ale prišla aj dilema čo spraviť s Objective-C. Nikto nechcel nútiť všetkých vývojárov, ktorí doteraz vyvíjali aplikácie pre Apple zariadenia prejsť na nový, neznámy programovací jazyk, a tak padlo rozhodnutie ďalej podporovať Objective-C, ale zároveň sa plne venovať vývoju Swiftu. Do dnešného dňa sa obidva jazyky používajú.

Prvé dojmy vývojárov ohľadom Swiftu boli zmiešané. Swift si získal veľa priaznivcov, ktorí ocenili jeho jednoduchosť, flexibilitu, a aké možnosti ponúkal, ale zároveň ho aj veľa ľudí kritizovalo a väčšina vývojárov sa zhodla, že je príliš skoro tento jazyk začať používať v produkčnom prostredí. V roku 2015 sa Apple rozhodol urobiť tento jazyk *open-source*, čo výrazne pomohlo jeho rastu a zlepšovaniu. V Marci 2019 Apple vydal Swift 5.0, ktorý priniesol stabilnú verziu ABI(Application

Binary Interface) naprieč všetkými platformami, ktoré Apple ponúka. Ďalšou veľkou novinkou v tejto verzii bol príchod SwiftUI, čo je vlastne knižnica pre dizajn užívateľského rozhrania. Toto vydanie so sebou taktiež prinieslo vynovenú dokumentáciu a Swift sa stal spätne kompatibilným so svojimi predchádzajúcimi verziami. Swift rozšíril svoju oficiálnu podporu na niekoľko dodatočných Linux distribúcii a na Windows v Septembri 2020 s príchodom Swift 5.3.

Podľa oficiálneho príspevku od Applu Swift kombinuje výkonnosť a efektivitu kompilovaných jazykov, ale zároveň si ponecháva jednoduchosť a interaktivitu skriptovacích jazykov. Čo to v praxi ale reálne znamená?

Výhody Swiftu

- Je to expresívny jazyk so zjednodušenou syntaxou, čo znamená, že sa ľahko píše a číta. V porovnaní s Objective-C potrebujeme na vykonanie tej istej úlohy napísať o niečo menej kódu, čo umožňuje vyvíjať aplikácie rýchlejšie.

- O manažovanie a správu pamäte sa stará ARC(Automatic Reference Counting). Zjednodušene povedané to znamená, že vývojári vôbec nemusia riešiť, čo sa kedy deje s pamäťou a všetko bude fungovať vo väčšine prípadov bez problémov. Pred príchodom ARC museli iOS vývojári riešiť manuálne, ktoré inštancie tried sa už nepoužívajú a manuálne ich vymazávať. To zvyšovalo počet riadkov kódu a častokrát to nebolo ideálne z pohľadu optimalizácie, keďže to zaberalo zbytočne veľa výpočtového výkonu procesora.

- Medzi ďalšie výhody Swiftu patrí ľahká škálovateľnosť. Do našej aplikácie môžeme voľne pridávať novú funkcionalitu a nemusíme sa báť toho, že to prestane zo dňa na deň fungovať, keďže Swift bude ešte dlho podporovaný a kód, ktorý napíšeme dnes bude fungovať aj v budúcich verziách Swiftu.

- Swift bol navrhnutý tak, aby bol rýchlejší a výkonnejší oproti svojmu predchodcovi a to bolo dosiahnuté hlavne vďaka využitiu LLVM(Low Level Virtual Machine) kompilátoru. LLVM je vlastne knižnica na vytváranie binárneho kódu, ktorému zariadenie na ktorom program spúšťame rozumie. LLVM poskytuje svoje vlastné implementácie pre funkcie, globálne premenné a ďalšie rôzne koncepty, ktoré sú v programovacích jazykoch bežné, takže vývojárom, ktorí vytvárajú nový jazyk to dokáže veľmi uľahčiť prácu. Automatická optimalizáciu kódu je ďalšou výhodou, ktorou disponujú jazyky, ktoré využívajú LLVM.

- Pri vývoji aplikácie môžeme využívať knižnice tretích strán, a ak sú tieto knižnice statické tak v momente skompilovania nášho kódu sa stanú súčasťou našej aplikácie a tým pádom veľkosť aplikácie rastie. Swift predstavil dynamické knižnice. Niektoré štandardné knižnice sa stali priamou súčasťou operačných systémov Apple zariadení a tým pádom nemusí naša aplikácia zaberať zbytočne viac miesta a jednoducho k týmto knižniciam dokáže pristupovať.

- Perfektne kompatibilný s Objective-C. V rámci jedného projektu môžeme využívať obidva jazyky bez akýchkoľvek problémov. Toto je hlavne užitočné pre staršie projekty, ktoré treba rozšíriť o novú funkcionalitu.

- Swift je *open-source* projekt s veľkou a aktívnou komunitou vývojárov, ktorú si získal vďaka veľkej podpore od Applu.

- Swift na to, že to je relatívne nový programovací jazyk má veľmi veľa materiálov, ktoré pomáhajú vývojárom sa tento jazyk naučiť. Existuje veľa e-kníh, komunitných návodov a videí a taktiež dokumentácia je dosť detailne napísaná. Swift Playgrounds je vývojové prostredie, ktoré slúži na interaktívne naučenie toho, ako kód funguje. Swift Playgrounds je možné stiahnuť ako aplikáciu napríklad na iPad. Je to vynikajúca pomôcka pre niekoho, kto ešte len začína s programovaním.

Nevýhody Swiftu

- Časté aktualizácie a zmeny v jazyku môžu niektorých vývojárov odradiť od používania tohto jazyka.

- V porovnaní s najpopulárnejšími jazykmi je počet vývojárov, ktorí ovládajú Swift stále malý, takže hrozí, že ak napríklad máme nejaký komplexnejší problém, tak možno budeme pomoc hľadať dlhšie ako keby sme mali problém na podobnej úrovni v inom populárnejšom jazyku.

- Nekompletná podpora pre všetky platformy. Aj napriek tomu, že Swift už podporuje Windows platformu a väčšinu Linux distribúcií, tak tento jazyk bol vyvinutý pre najlepší vývoj iOS aplikácií a nie pre vývoj multiplatformových aplikácií.

- Nefunguje na zariadeniach, ktoré využívajú iOS 6 alebo nižšie verzie tohto operačného systému. Tu by som rád podotkol, že len veľmi malé percento Apple zariadení beží na iOS verzii 6 alebo nižšie [17][18][19].

3.3.1 Premenné, konštanty a typy kolekcií

Každý programovací jazyk umožňuje vývojárom ukladať hodnoty do pamäte a následne k týmto hodnotám pristupovať alebo ich prepisovať a Swift v tomto nie je iný.

Premenné a konštanty

Zdrojový kód 3.1: Deklarácia premennej

```
var pi : Double = 3.14159
```

Takto 3.1 vyzerá deklarácia premennej v Swifte s priradením hodnoty. Kľúčové slovo *var* sa používa pre deklarovanie premennej akéhokoľvek základného dátového typu a po priradení hodnoty Swift automaticky určí, o aký dátový typ sa jedná ak ho pri deklarovaní nešpecifikujeme sami.

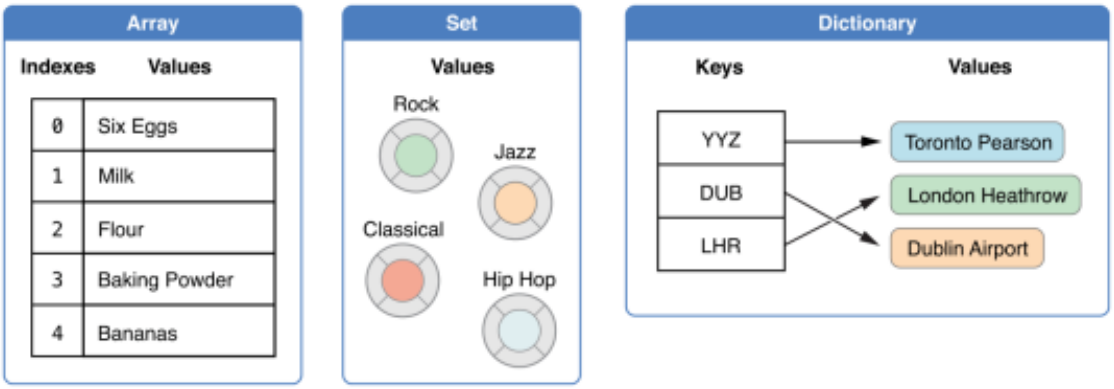
Swift má tieto základné dátové typy:

- *Character* - jedná sa 16 bitový Unicode charakter
- *String* - *string* slúži na uloženie dát v podobe textu
- *Integer* - tento dátový typ slúži na uloženie celého čísla. Minimálna a maximálna veľkosť závisí od platformy na ktorej sme. Pri 32 bitovej platforme je rozsah čísla, ktoré vieme uložiť od $-(2^{31})$ do $2^{31}-1$ a pri 64 bitovej platforme to je od $-(2^{63})$ do $2^{63}-1$.
- *Float* - *float* má 32 bitov a slúži na uloženie desatinného čísla s presnosťou na 6 desatinných miest. Jeho rozsah je od 1.2×10^{-38} do 3.4×10^{38}.
- *Double* - rovnaký dátový typ ako *float*, len s tým rozdielom, že je 64 bitový a dokáže uložiť číslo s presnosťou na 15 desatinných miest. Jeho rozsah je od 2.3×10^{-308} do 1.7×10^{308}.
- *Boolean* - jedno bitový dátový typ používaný na uloženie pravdivostnej hodnoty. Môže nadobudnúť len hodnoty *true* alebo *false* teda buď 1 alebo 0. Ak mu nepriradíme pri vytvorení hodnotu, tak je predvolený *false*.

Konštanty sa deklarujú úplne rovnako ako bežné premenné, len s tým rozdielom, že namiesto kľúčového slova *var* použijeme kľúčové slovo *let*. Hlavnou vlastnosťou konštánt narozdiel od bežných premenných je to, že po priradení hodnoty sa ich hodnota nemôže zmeniť počas celého behu programu.

Typy kolekcií

Slúžia na uloženie viacerých premenných do jednej zbierky. Medzi tri základne typy kolekcií, ktoré Swift ponúka patria *array*, *set* a *dictionary*. Počas behu programu s nimi vieme voľne pracovať a to tak, že ich vieme čítať, dokážeme z nich odstrániť premenné alebo pridať nové premenné. Samozrejme z nich vieme spraviť aj konštanty, a potom ich nebudeme schopní meniť, ale len čítať hodnoty vo vnútri.



Obr. 3.3: Typy kolekcií

- *Array* - je to vlastne zoradené pole, do ktorého vieme uložiť každý typ premennej alebo objektu. K týmto zoradeným hodnotám vieme pristupovať pomocou indexu.
- *Set* - *set* je to isté ako *array* len s tým rozdielom, že to nie je zoradené a kvôli tomu je pri *sete* nutnosť toho, aby obsahoval len unikátne hodnoty.
- *Dictionary* - do tohto vieme uložiť unikátne kombinácie kľúča s priradenou hodnotou. Tieto kľúč hodnota páry nie sú zoradené, takže ak chceme cez *dictionary* iterovať v nejakom poradí, tak musíme použiť funkciu *sorted* na stĺpec s kľúčmi alebo hodnotami.

3.3.2 Kontrolovanie behu programu

Swift tak isto ako väčšina moderných programovacích jazykov ponúka základné prvky pre kontrolu toho, čo sa má v konkrétnom bode programu vykonať.

Medzi základné prvky na kontrolu behu programu patria:

- *For-In* cyklus - tento cyklus je užitočný ak chceme iterovať cez nejaké pole, sekvenciu alebo napríklad len charaktermi v *stringu*.

Zdrojový kód 3.2: For-In cyklus

```
for game in games {
    print("Game: \(game)!")
}
```

- *While* cyklus - v tomto type cyklu je na začiatku určená podmienka a dokedy podmienka platí, tak sa blok kódu vo *while* cykle vykonáva. Existuje aj verzia, kde podmienka je určená na konci. V tomto prípade sa cyklus vykoná minimálne jedenkrát kvôli tomu, že platnosť podmienky sa kontroluje až po vykonaní cyklu. Tieto cykly sa využijú hlavne ak dopredu nevieme koľkokrát budeme potrebovať aby sa cyklus vykonal. Pri používaní tohto cyklu si treba dať pozor na to, aby sme sa nezacyklili v nekonečnom cykle.

- *If-Else* - tieto podmienky sa využívajú ak chceme aby sa určitý blok kódu vykonal len pri splnení nami danej podmienky, a ak podmienka splnená nie je tak sa ide v kóde ďalej. Dokážeme za sebou zreťaziť viacero podmienok a tie sa zaradom kontrolujú, ak Swift narazí na podmienku, ktorá je splnená, tak blok kódu k nej priradený sa vykoná a ďalšie podmienky za ňou nekontroluje, ale ide v programe ďalej. Ak potrebujeme nejak ošetriť to, že ani jedna podmienka nie je splnená, ale aj tak potrebujeme aby sa niečo vykonalo, tak na konci vieme použiť *else* odstavec, ten je plne voliteľný a vykoná sa len ak všetky ostatné podmienky nie sú splnené.

- *Switch* - pri *switchy* určíme na začiatku premennú, ktorú porovnávame s rôznymi prípadmi, a ak porovnanie platí, tak blok kódu príslušného prípadu je vykonaný. Taktiež pre každý prípad vieme určiť rôzne podmienky na to aby bol platný. Ak žiaden prípad nie je platný, tak v *switchy* existuje *default* prípad. Ten sa vykoná ak žiadny nad ním neplatí. Funkcionalita *switchu* je veľmi podobná *If-Else*, ale je tam pár rozdielov. Napríklad pri *switchy* sa hodnota číta len raz a to je z optimalizačného hľadiska lepšie. *Switch* je častokrát prehľadnejší.

- *Continue* - toto kľúčové slovo sa používa vo vnútri cyklov. Ak program dôjde ku *continue* v cykle, tak tá daná konkrétna iterácia sa v tom momente prestane vykonávať a preskočí sa priamo k ďalšej iterácií v cykle.

- *Break* - *break* sa používa na vyskočenie z cyklu, podmienky alebo *switchu*.

- *Fallthrough* - v iných programovacích jazykoch sa za každým prípadom v *switchy* dáva *break*, aby sa nevykonali všetky prípady za prvým úspešným. Swift toto rieši bez potreby *breaku* v každom prípade. Ak ale chceme replikovať funkcionalitu z iných jazykov, tak môžeme použiť *fallthrough*. Ak sa prípad vykoná a je v ňom *fallthrough*, tak sa vykonajú aj všetky prípady za ním.

- *Guard* - *guard* je skoro to isté ako *if*, len s tým rozdielom, že vždy vyžaduje *else* blok. Pomocou *guard let* vieme inicializovať premennú a tá je dostupná v danom bloku. *Guard* bol navrhnutý na to, aby sme skontrolovali podmienku a ak neplatí, tak vyskočili mimo danú funkciu alebo cyklus.

3.3.3 Funkcie

Funkcia je blok kódu, ktorý má vykonať nejakú špecifickú úlohu. Po definovaní funkcie ju môžeme zavolať v inom bode kódu aby sa vykonala. V Swifte sú funkcie veľmi flexibilné. Dokážu prijať niekoľko vstupných parametrov zároveň. Parametre môžu byť rôzneho typu. Vieme určiť predvolenú hodnotu pre každý parameter funkcie, to je dobré na to keď nechceme pri každom zavolaní funkcie určovať všetky parametre manuálne. Taktiež môžeme ako parameter funkcie zavolať inú funkciu. Existuje možnosť písať vnorené funkcie, to sú také funkcie, ktoré sú napísané vo vnútri inej funkcie.

Pri definovaní funkcie taktiež musíme určiť typ vrátenej hodnoty. Funkcia bez určeného výstupného typu sa nazýva *void* a takýto typ funkcie nič nevracia. V Swift funkcii vieme určiť viac ako jeden výstupný typ, ale v takom prípade aj musíme viac týchto hodnôt na konci funkcie vrátiť. To platí za predpokladu, že na začiatku neurčíme výstupný typ ako voliteľný, v prípade ak ho tak určíme tak môžeme vrátiť *nil*. *Nil* je výraz pre chýbajúcu alebo teda prázdnu hodnotu [20].

3.3.4 Štruktúry, triedy a enumeračné typy

Štruktúry a triedy

Štruktúry a triedy sú vlastne šablóny pre vytvorenie objektu. Obidve ponúkajú podobnú funkcionalitu. Dokážu v sebe uchovávať rôzne premenné, môžeme pre nich vytvoriť metódy. Metódy sú funkcie pre prácu s týmito inštanciami. Napríklad ak máme v rámci triedy nejakú privátnu premennú, tak vieme napísať metódu, ktorá túto hodnotu prečíta a vráti. Pri oboch vieme využiť inicializátor na určenie počiatočného stavu objektu po vytvorení. Taktiež môžu podliehať rôz-

nym protokolom pre získanie určitej funkcionality. Hlavným rozdielom medzi štruktúrou a triedou je to, že pre vytvorenie štruktúry použijeme kľúčové slovo *struct* a pre vytvorenie triedy kľúčové slovo *class*. Čo sa týka rozdielu vo funkcionalite, tak trieda vie navyše oproti štruktúre zdediť základné charakteristiky z inej triedy.

Enumeračné typy

V Swifte vieme vytvoriť enumeračný typ pomocou kľúčového slova *enum*. Takýto enumeračný typ je vlastne zoznam predom určených hodnôt. Po definovaní enumeračného typu vieme vytvoriť premennú tohto nami vytvoreného typu a môžeme jej určiť jednu hodnotu z možného zoznamu, ktorý sme vytvorili. Enumeračný typ vytvárame vtedy ak chceme aby premenná tohto typu nadobúdala len jednu z hodnôt, ktoré sme si dopredu určili. Často sa to kombinuje s používaním *switchu*. Používa sa to hlavne kvôli bezpečnosti pri písaní kódu.

Properties

Properties sú premenné, ktoré prislúchajú alebo teda patria určitej inštancii triedy, štruktúry alebo enumeračného typu.

Dedenie

Ako už bolo spomenuté tak trieda vie zdediť základné charakteristiky z inej triedy. Takúto triedu nazývame podtrieda. Dedenie funguje tak, že napríklad máme jednu triedu, ktorá v sebe obsahuje len jednu premennú. Ďalej definujeme podtriedu, ktorá z tejto triedy bude dediť. V tejto triede tiež môžeme mať definovanú nejakú premennú. Po vytvorení inštancie tejto podtriedy budeme mať prístup k obom premenným z obidvoch tried.

Ak nám nejaké charakteristiky z triedy z ktorej chceme dediť nevyhovujú, tak ich môžeme prepísať kľúčovým slovíčkom *override*. S pomocou *override* vieme implementovať rovnako nazvané metódy nejak inak pre našu podtriedu a to isté platí pre premenné [21][22].

3.3.5 Rozšírenia a protokoly

Rozšírenia

Rozšírenia dokážu pridať nejakú prídavnú funkcionalitu k už existujúcim triedam, štruktúram alebo enumeračným typom. Rozšírenia dokážu len pridať novú

funkcionalitu, nedokážu modifikovať funkcionalitu už existujúcu.

Protokoly

Protokoly definujú funkcionalitu, ktorú je nutné implementovať. Ak napríklad trieda zodpovedá určitému protokolu, tak si môžeme byť istí, že v danej triede sú implementované všetky veci, ktoré protokol určil ako napríklad vyžadované premenné alebo metódy [23].

3.3.6 Ošetrenie chýb

Ošetrenie chýb je proces pri ktorom na vzniknuté chyby alebo problémy v programe nejako odpovedáme alebo obnovíme funkcionalitu programu. Niekedy sa stane, že úloha, ktorú aktuálne program vykonáva sa nedokončí. V takých prípadoch je užitočné vedieť prečo sa daná úloha nedokončila. Na to vo Swifte slúži *Error* protokol. Tento protokol je prázdny a označuje to, že enumeračný typ, ktorý tomuto protokolu zodpovedá je použitý na zaobchádzanie s chybami.

Hlásenie o chybe hodíme vtedy, ak sa niečo nečakané stane a normálny beh programu nemôže pokračovať ďalej. Na hodenie takéhoto hlásenia sa používa kľúčové slovo *throw*.

Zdrojový kód 3.3: Do, catch, try príklad

```
do{
    try readRandomFile(path: "D:\Martin\file.png")
    print("Success!")
}catch fileRead.invalidPath{
    print("Invalid File Path.")
}catch{
    print("Default Error Message.")
}
```

Existuje viacero spôsobov ako vyriešiť chybové hlásenie. Takým najbežnejším spôsobom je použitie *do-catch* a *try*. Do *do* bloku napíšeme kód, ktorý chceme aby sa vykonal a vo vnútri tohto bloku použijeme kľúčové slovo *try*. To nám zabezpečí, že v prípade chyby nám nespadne celý program, ale prejde sa do *catch* bloku, v ktorom ošetríme čo sa má stať v prípade, že úloha zlyhala. Týchto *catch* blokov môžeme mať viacero a to pre každú jednu konkrétnu chybu, ktorá môže vzniknúť.

Ďalšou možnosťou je použitie kľúčového slova *try?*. To môžeme napríklad použiť ak chceme do nejakej premennej nahrať dáta zo servera, zavoláme funkciu, ktorá sa stará o získanie týchto dát a keď zlyhá, tak *try?* nám zabezpečí to, že namiesto spadnutia programu sa do premennej zapíše hodnota *nil*.

Kľúčové slovo *defer* sa používa na označenie bloku, ktorý sa má vykonať tesne predtým ako sa vykonávanie kódu posunie z daného bloku kódu do ďalšieho. Napríklad ak v rámci funkcie otvoríme nejaký súbor, tak do *defer* bloku by sme dali jeho zatvorenie. To nám zabezpečí to, že všetko čo sme chceli so súborom vykonať v rámci funkcie bolo vykonané a po dokončení funkcie neostal tento súbor zbytočne otvorený [24].

Type Casting

Type casting slúži na skontrolovanie toho, či to je inštancia takého typu aký vyžadujeme a to vieme docieliť pomocou kľúčové slova *is*. Ďalej existuje kľúčové slovo *as* a to sa využíva na zmenu typu objektu na iný. Zmena typu môže byť vynútená alebo len ak to je možné. Vynútená je ak použijeme *as!* a voliteľná ak použijeme *as?*. Pri vynútenej vznikajú chyby ak nie je daný typ možné premeniť na nami určený, a preto sa to využíva len keď sme si úplne istí, že daný typ vieme na nami určený zmeniť [25].

3.4 SwiftUI

SwiftUI je nový *framework* pre tvorbu užívateľského rozhrania od Applu. Bol vydaný v roku 2019 s cieľom umožniť vývojárom aplikácií pre Apple zariadenia vyvíjať užívateľské rozhranie deklaratívne. To znamená, že písanie kódu pre užívateľské rozhranie je teraz jednoduchšie a prirodzenejšie, taktiež to vyžaduje menej napísaného kódu pre dosiahnutie toho istého výsledku v porovnaní s predchádzajúcimi riešeniami. Jednou zo slabých stránok Xcodu bolo to, že funkcionalita náhľadu neumožňovala náhľad zložitejších zmien, ale podporovala len náhľad jednoduchých UI prvkov, takže pre zobrazenie kompletného UI bolo nutné úplne načítanie aplikácie v simulátore alebo na fyzickom zariadení. Tento problém SwiftUI úplne odstránilo, keďže teraz je možné každú zmenu UI v kóde sledovať v reálnom čase v náhľade bez nutnosti spustenia aplikácie. Ďalšou výhodou je, že ak chceme vidieť ako aplikácia bude vyzerať v tmavom alebo svetlom móde, tak si to vieme jednoducho pod kanvasom prepnúť. Písanie kódu je ale pri vývoji UI častokrát redundantné, preto pri SwiftUI je možnosť interaktívne vybrať UI komponent a Xcode nám automaticky vygeneruje kód pre tento komponent. To šetrí

čas a hlavne robí vývoj príjemnejším a hlavne priamočiarejším. Týmto spôsobom sú kód a užívateľské rozhranie vždy synchronizované a to je pre vývojárov veľká pomôcka [26].

Každý UI komponent má dostupné modifikátory. Tie slúžia na to aby sme mohli daný komponent graficky upraviť, tak ako nám to vyhovuje. Modifikátory pre každý konkrétny komponent si nemusíme pamätať, keďže Xcode nám poskytne zoznam všetkých dostupných modifikátorov aj s popisom pre komponent na ktorý si klikneme.

Medzi také najzákladnejšie modifikátory patria:

- *Font color* - určuje farbu písma v komponente
- *Font size* - určujú veľkosť písma
- *Background color* - určuje farbu pozadia
- *Corner radius* - určuje ako veľmi zaoblené budú rohy komponentu

Užívateľské rozhranie zobrazuje užívateľovi súčasný pohľad s rôznymi typmi UI komponentov. Ak sa hodnota alebo teda stav nejakého UI komponentu zmení, tak pohľad by sa mal aktualizovať a zobraziť užívateľovi aktuálnu hodnotu. Toto sa deje pri používaní SwiftUI automaticky vďaka *property wrappers*, ktoré viažu pohľad s konkrétnymi dátovými premennými. Zároveň tieto dátové premenné môžu byť použité, ak užívateľ zadá vstup ako napríklad svoje meno a to je uložené do tejto premennej. Tento koncept nesie názov *zdroj pravdy* [27][28].

Zdrojový kód 3.4: @State príklad

```
struct StateExample: View {
  @State private var intValue = 0
  var body: some View {
    VStack {
      Text("intValue equals \(intValue)")
      Button("Increment") {
        intValue += 1
      }
    }
  }
}
```

Ako môžeme vidieť na ukážke kódu 3.4, tak pridanie *property wrapperu @State* pri deklarácii premennej je veľmi priamočiare. Následne tu máme *Text* kompo-

nent, ktorý nám zobrazuje hodnotu nami zadeklarovanej premennej a *Button* komponent, ktorý pri každom stlačení pripočíta k našej premennej číslo jedna. Po každom stlačení tlačidla sa teda zmení hodnota premennej a kvôli tomu, že naša premenná má *property wrapper @State*, tak pohľad sa automaticky vykreslí na novo s aktuálnom hodnotou danej premennej a to všetko bez akéhokoľvek iného zásahu od nás.

@Binding property wrapper sa používa pre premenné, ktorých hodnotu získame z iného pohľadu. Tým pádom pohľad v ktorom je táto *binding* premenná dokáže reagovať na zmenu tejto premennej z iných zdrojov. Taktiež má právo na zápis do tejto premennej a to znamená to, že pri zmene v tejto premennej sa aktualizuje táto premenná aj v pohľade z ktorého sme pôvodnú hodnotu získali.

Zdrojový kód 3.5: @StateObject príklad

```
class DataProvider: ObservableObject {
  @Published var currentValue = "a value"
}
struct DataOwnerView: View {
  @StateObject private var provider = DataProvider()
  var body: some View {
    Text("provider value: \(provider.currentValue)")
  }
}
```

@StateObject je viacmenej rovnaký *property wrapper* ako *@State*, ale s tým rozdielom, že môže byť použitý len na nejaký pozorovateľný objekt ako je napríklad trieda. Táto trieda v sebe musí obsahovať premennú *@Published* a pri každej zmene tejto premennej bude SwiftUI o tom informovaný, aby sa mohol pohľad vykresliť s aktuálnou hodnotou. Týchto *@Published* premenných môže byť v jednej triede samozrejme viac a pohľad sa aktualizuje pri zmene akejkoľvek z nich. Ako aj môžeme vidieť na ukážke kódu 3.5, tak až na to, že pristupujeme k premennej, ktorá je v triede je tento kód takmer identický s tým predchádzajúcim.

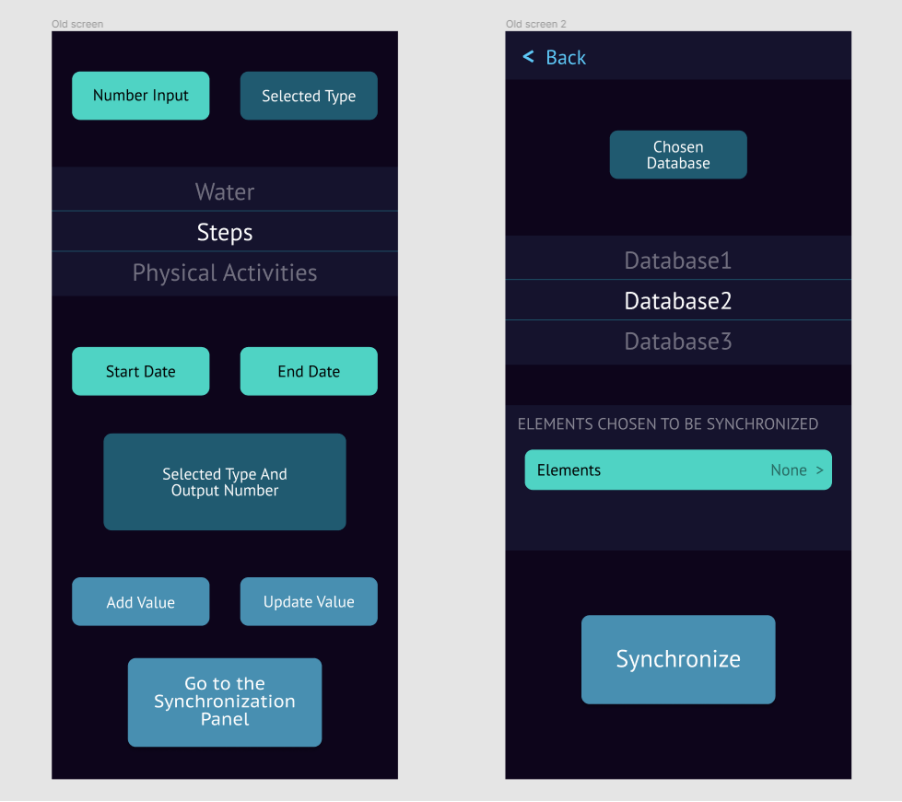
@ObservedObject property wrapper sa používa pri pozorovateľných objektoch, ktoré nevlastní pohľad v ktorom sú použité ale referencia na ne je poslaná z iného pohľadu. Väčšinou to je referencia na nejaký existujúci *@StateObject*.

@EnvironmentObject je tiež *property wrapper* pre pozorovateľný objekt, ale tento typ využívame ak potrebujeme pristupovať k premenným tohto objektu na viacerých miestach v rámci aplikácie [29].

4 Návrh riešenia

Figma je nástroj, ktorý slúži na dizajnovanie aplikácií. Je dostupná ako webová služba a návrh vyvíjame cez grafické prostredie. V tejto kapitole popíšem ako sa vyvíjal dizajn aplikácie od počiatočného návrhu až po finálny.

4.1 Prvotný dizajn UI



Obr. 4.1: Prvý návrh UI

Na Obr. 4.1 môžeme vidieť úplne prvý návrh používateľského rozhrania pre našu aplikáciu. Tento návrh bol vytvorený bez ohľadu na to ako budú používatelia reagovať pri používaní aplikácie a aká bude ich prvotná skúsenosť s aplikáciou. Jediná vec, ktorú tento návrh spĺňa je tá, že obsahuje väčšinu požadovanej funkcionality aplikácie.

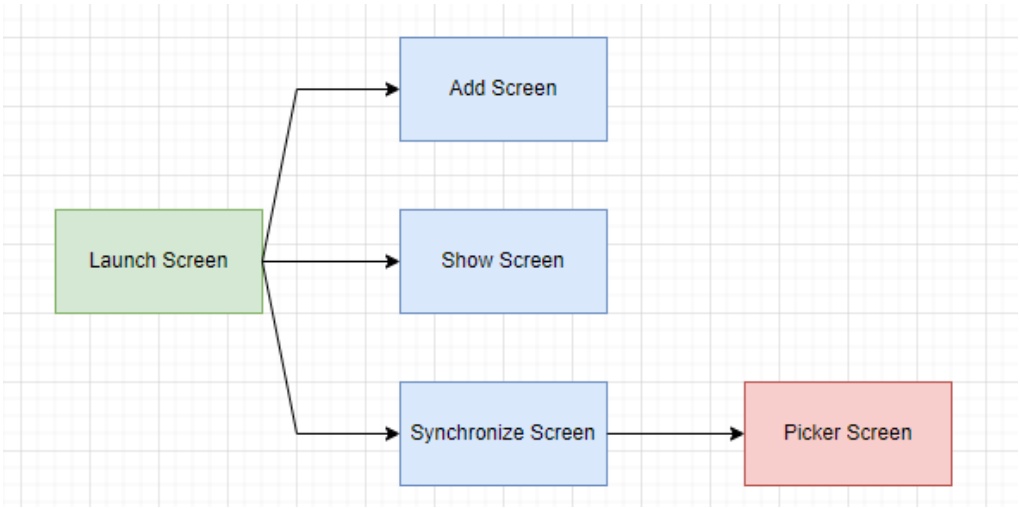
Medzi hlavné problémy tohto návrhu patria:

- Farebná schéma - farebná schéma je z veľkej väčšiny podľa môjho názoru v poriadku. Jediný problém, ktorý si vyžaduje do finálneho dizajnu úpravu je to, že niektoré prvky majú nudnú farbu, ktorá nepôsobí tak dobre so zvyškom návrhu.
- Veľa informácii naraz - úvodná obrazovka obsahuje príliš veľa na sebe natlačených prvkov, takže užívateľ musí naraz spracovať veľké množstvo informácii, ktoré od neho aplikácia vyžaduje a to môže spôsobiť nepríjemný prvý dojem z aplikácie.
- Mätúca navigácia - tento problém je spojený so snahou o natlačenie množstva informácií na jednu obrazovku. Jednoducho povedané užívateľovi nie je jasné čo má v ktorom kroku vykonať, keďže mu to aplikácia neprezentuje čo najjednoduchšie. Nevie, ktoré vstupné pole s čím súvisí a aké tlačidlo čo spôsobí a za akých podmienok. Taktiež veľkým problémom je, že dátum vyberáme na prvej obrazovke, ale ak užívateľ prejde na obrazovku so synchronizáciou tak táto funkcionalita pracuje s dátumom vybraným na predošlej obrazovke a nikde ho ani nemôžeme skontrolovať alebo zmeniť bez nutnosti vrátiť sa na pôvodnú obrazovku.
- Konzistentnosť - návrh nemá nejaký zavedený vzor, ktorého sa drží, ale prvky sú všelijako porozhadzované po obrazovke.

4.2 Finálny dizajn UI

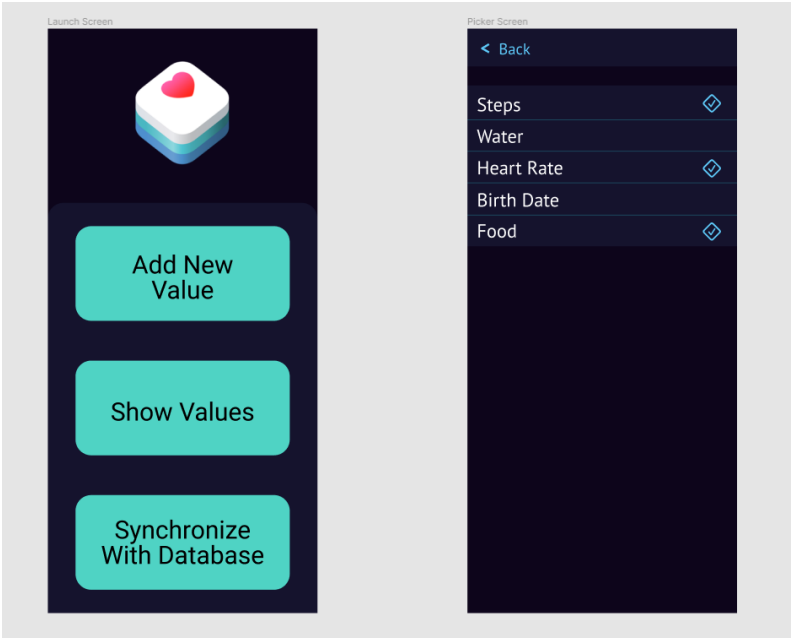
Cieľom finálneho návrhu bolo napraviť chyby, ktoré obsahoval prvotný návrh. Chyby boli odhalené vďaka viacerým ľuďom, ktorým som tento návrh ukázal. Medzi ľuďmi, ktorí prvotný návrh videli a vyjadrili sa k nemu boli aj menej technicky zdatní a starší ľudia, ale taktiež aj mladší či už menej alebo viac technicky zdatní ľudia. Po zhodnotení všetkých názorov som bol schopný nájsť, aké chyby vznikli v prvotnom návrhu a napraviť ich ako som najlepšie vedel. Jedným z hlavných problémov prvotného návrhu bolo to, že užívateľ si nie je istý, čo má kedy vykonať. Tento problém sme vyriešili pridaním viacerých obrazoviek medzi ktorými sa vie užívateľ intuitívne orientovať. Na Obr. 4.2 vidíme navigačný diagram, ktorý sme neskôr implementovali aj do dizajnu UI.

Na úvodnej obrazovke na Obr. 4.3 vidíme jednoduchú plochu s troma tlačidlami a HealthKit logom, ktoré označuje to, že aplikácia využívate tento *framework*.



Obr. 4.2: Návrhový diagram

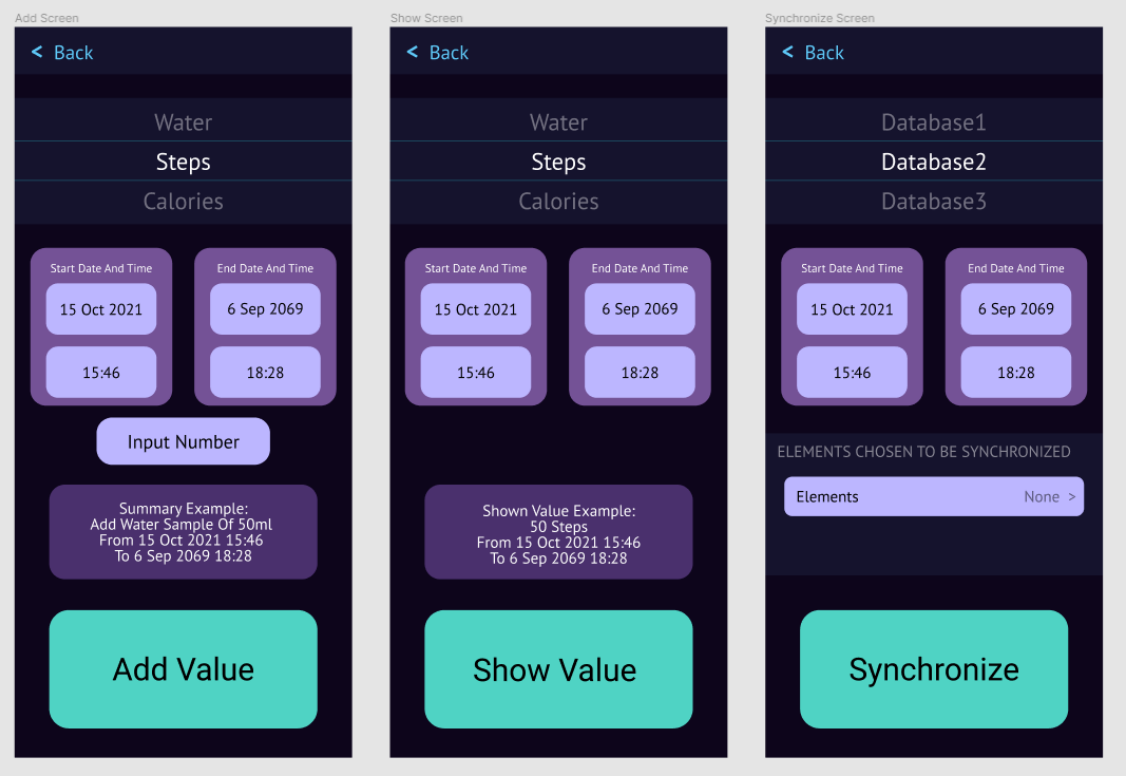
Jediné, čo je teda od užívateľa v tomto kroku vyžadované je to, že má kliknúť na tlačidlo podľa toho čo si praje v aplikácii vykonať.



Obr. 4.3: Finálny návrh UI 1

Po ťuknutí na jedno z týchto tlačidiel bude presmerovaný na jednu z obrazoviek na Obr. 4.4. Tieto tri obrazovky spĺňajú všetky podmienky, ktoré prvý dizajn nespĺňal. Sú konzistentné, keďže všetky tri sa zhodujú v tom, čo sa približne deje v danej časti obrazovky. V hornej časti vyberáme z výberového pohľadu hodnotu, ktorú chceme zobraziť, čítať alebo databázu. V strednej časti obrazovky sú vstupy užívateľa ako dátum odkedy a dokedy alebo konkrétnu hodnotu, ktorú chce pridať a taktiež tam je súhrn toho čo vybral. Úplne dole na obrazovke je potvrdzujúce

tlačidlo, ktoré sa od všetkého na danej obrazovke svojim dizajnom líši aby bolo zrejmé, že sa jedná o tlačidlo, ktoré má vykonať nejakú úlohu.



Obr. 4.4: Finálny návrh UI 2

Sú nenáročné na prezentovanú informáciu. Od užívateľa sa vyžadujú len absolútne nutné informácie, takže užívateľovi stačí len vyplniť všetko čo mu obrazovka ponúka a stlačiť potvrdzujúce tlačidlo v dolnej časti obrazovky. Na každej obrazovke je taktiež tlačidlo "Back", ktoré slúži na vrátenie sa na predchádzajúcu obrazovku a ako už býva pri iOS aplikáciach zvykom, tak je umiestnené v ľavej hornej časti obrazovky v navigačnom panely.

Konzistentnosť spolu s prehľadným prezentovaním informácie na každej danej obrazovke a dobrou navigáciou medzi obrazovkami spôsobuje to, že aplikácia už nie je vôbec mätúcou ale plne intuitívnou.

Farebná schéma bola trochu obmenená, zelená farba ostala len tlačidlám na úvodnej obrazovke a tlačidlám, ktoré slúžia na potvrdenie a vykonanie úlohy, ktorú prislúchajúca obrazovka rieši. Odtiene modrej boli nahradené odtieňmi fialovej a to spôsobuje, že aplikácia pôsobí na užívateľa živšie a prívetivejšie. Zároveň vyzerá lepšie v kombinácii s tmavým pozadím.

Väčšina problémov bola týmto dizajnovým návrhom vyriešená a aplikácia bola takto prispôsobená pre používanie aj pre menej technicky zdatných užívateľov.

5 Implementácia riešenia

V tejto kapitole sa budem bližšie venovať tomu ako som k riešeniu zadania pristúpil, popíšem aké kroky som vykonal, popíšem použité technológie a vysvetlím základné časti kódu, a to hlavne čo robia a ako fungujú.

5.1 Výber prostredia

Existujú rôzne možnosti ako môžeme vyvíjať iOS aplikáciu, či už natívnu alebo multiplatformovú. Keďže som nemal žiadne predchádzajúce skúsenosti s vývojom mobilnej aplikácie, tak som pozorne zhodnotil väčšinu dostupných možností. Po konzultácii so školiteľom a následným vzdelaním sa o rôznych možnostiach, o výhodách a nevýhodách daných možností som sa rozhodol pre použitie Xcodu. Medzi hlavné dôvody prečo som sa rozhodol pre toto vývojové prostredie patrí to, že Xcode je priamo od spoločnosti Apple, takže je v ňom obsiahnuté všetko čo vývojár pri vyvíjaní iOS aplikácie bude potrebovať.

Čo sa týka programovacieho jazyka tak Xcode síce podporuje vývoj v rôznych programovacích jazykoch, ale ja som sa rozhodol pre Swift, keďže to je najnovší programovací jazyk plne podporovaný spoločnosťou Apple a vyvinutý priamo na vývoj iOS aplikácií.

5.1.1 Inštalácia prostredia

Inštalácia Xcode sama o sebe nepredstavovala žiadny problém. Ako som už vyššie spomínal stačí toto prostredie stiahnuť cez AppStore. Keďže ale nevlastním zariadenie s macOS na ktorom je AppStore dostupný tak som sa rozhodol použiť VM(Virtual Machine) a do tejto VM nainštalovať najnovšiu verziu macOS. Najprv som musel v BIOSe(Basic Input Output System) zapnúť AMD-V čo je riešenie pre AMD procesory, ktoré má pomôcť pri virtualizácii. Virtualizácia je proces spúšťania virtuálnej inštancie počítačového systému vo vrstve abstrahovanej od skutočného hardvéru. Zapnúť túto funkcionalitu nie je nutné, ale napomáha to výkonu VM, keďže bez zapnutého AMD-V by musel softvér na virtuálizáciu

veľa úloh emulovať. Následne som nainštaloval VMware Workstation Pro, čo je vlastne virtualizačný program, ktorý umožňuje beh viacerých VM v rovnakom čase. Pre tento program som sa hlavne rozhodol preto, lebo podporoval virtualizáciu macOS aj keď je Windows 10 nainštalovaný ako hosťovský systém. Po pár neúspešných pokusoch sa mi potom podarilo zohnať macOS Catalina ISO. Následne som sa pustil do nastavenia a nainštalovania VM s týmto systémom. Bolo nutné prepísať nejaké nastavenia v rámci VMware aplikácie aby som mohol odomknúť možnosť inštalácie macOS systému. To sa mi síce podarilo, ale kvôli tomu, že mám stolný počítač s procesorom AMD Ryzen 5 2600 som narazil na ďalší problém a to ten, že mi táto VM nepôjde spustiť a vypisuje mi to chybu. Tento problém som vyriešil vďaka návodu na internete, ktorý opísal ako treba zmeniť nastavenie VM, jednalo sa o pár riadkov textu v konfiguračnom súbore, ktoré sa venovali inštrukciám procesora.

5.2 Prvý pokus o realizáciu

Obr. 5.1: Aplikácia s jedným pohľadom

Ako môžeme vidieť na Obr. 5.1 tak zo začiatku vyzerala aplikácia len takto

jednoducho. Ide o aplikáciu s jedným pohľadom a na vytvorenie užívateľského rozhrania bol použitý *framework* SwiftUI. Pre SwiftUI som sa rozhodol hlavne kvôli tomu, že to Apple propagoval ako veľkú novu vec pre vývoj užívateľského rozhrania a taktiež v pár tutoriáloch, ktorými som si prešiel bolo odporúčané SwiftUI ako jednoduchý UI *framework* pre začiatočníkov a keďže to bol môj prvý pokus o vývoj mobilnej aplikácie, tak pri rozhodnutí som neváhal.

Čo mi najviac ako začiatočníkovi pomohlo bolo to, že počas toho ako som menil kód, tak som mohol sledovať zmeny v užívateľskom rozhraní v reálnom čase. Akcelerovalo to moje pochopenie toho ako funguje vývoj UI v mobilných aplikáciach.

Zdrojový kód 5.1: SwiftUI tlačidlo

```
HStack{
Button(action: {
   healthStore?.writeSteps(stepCountValue: 200,date: date)
}, label: {
    Text("Add 200 steps")
        .foregroundColor(Color.black)
        .padding(.all)
        .background(Color.green)
        .opacity(0.8)
        .cornerRadius(10)
})
Spacer()
```

V mojom prvom pokuse o aplikáciu vyzerá väčšina kódu, tak ako je ukázané v ukážke kódu 5.1. Väčšina elementov je uložených v takzvaných *Stacks*, tie sa vlastne starajú o to aby v rámci jedného *stacku* boli všetky elementy na rovnakej úrovni z pohľadu vertikálneho alebo horizontálneho. V tomto danom prípade tu je len jeden element, ktorý predstavuje tlačidlo. Je špecifikované aká funkcia sa má zavolať po stlačení tlačidla a následne už len definujeme ako má tlačidlo vyzerať a aký má mať popis. *Spacer*() slúži na pridanie rovnomernej medzery medzi elementami v *stacku*, týchto medzier môžeme pridať viac a vždy bude každá rovnomerná.

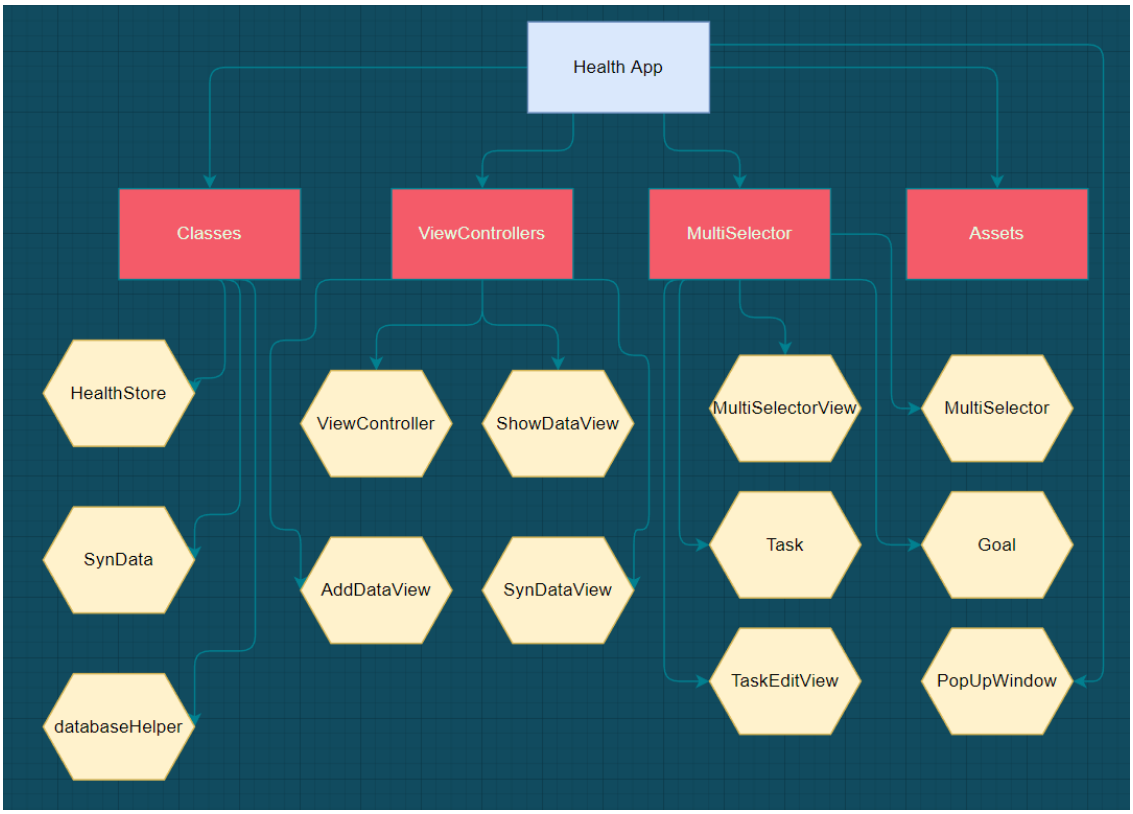
Táto aplikácia má samozrejme veľa problémov a väčšina funkcionality nefunguje vôbec alebo nie tak ako má. Napríklad hodnota, ktorá sa po stlačení tlačidla zobrazuje nie je vôbec presná a to je spôsobené tým, že funkcia, ktorá sa volá po stlačení nie je správne napísaná. Prvky užívateľského rozhrania sú umiestnené

celkom v poriadku, ale aj napriek tomu pôsobia rozhádzane. Niektoré prvky ako zobrazenie dátumu alebo tlačidlo pre aktualizovanie zobrazovanej hodnoty sú zbytočne redundantné.

5.3 Realizácia riešenia

V rámci tejto časti bakalárskej práce sa budem venovať konkrétnemu finálnemu riešeniu a technológiám použitým v rámci tohto riešenia.

5.3.1 Štruktúra projektu

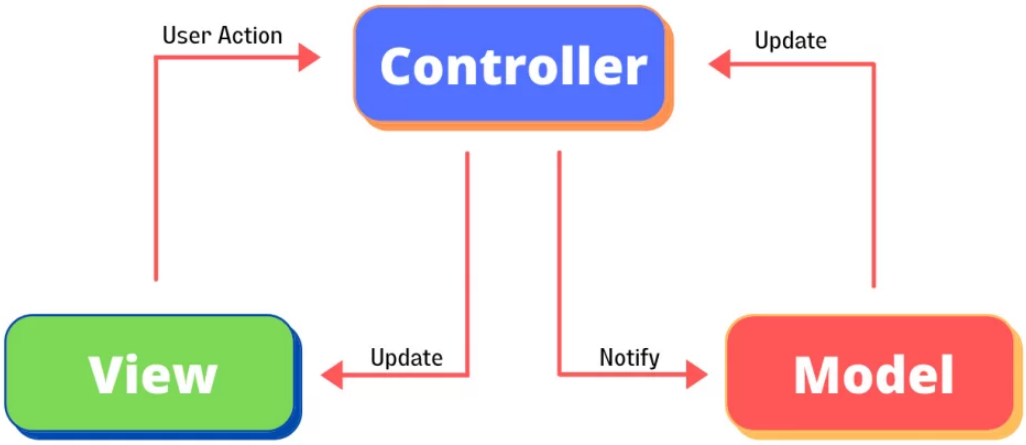


Obr. 5.2: Štruktúra projektu

Na Obr. 5.2 môžeme vidieť stromovú štruktúru projektu obsahujúcu priečinky a súbory, ktoré im patria. V Classes sú pomocné triedy v ktorých sa hlavne pracuje s dátami. Vo ViewControllers je kontrolér pre každý pohľad a v MultiSelectore sú súbory, ktoré riešia funkcionalitu pre výber viacerých možností z poľa, keďže Apple zatiaľ nevydal svoje vlastné riešenie ohľadom tejto problematiky. V Assets sú obrázky alebo fonty, ktoré aplikácia využíva a nie sú dostupné bez ich manuálneho pridania.

Pri vývoji aplikácie pomocou SwiftUI som narazil na pár problémov, ktoré boli hlavne spojené s tým, že to je relatívne nová technológia, preto som sa po nejakom čase rozhodol prerobiť všetko od začiatku cez UIKit *framework* a implementovať všetku funkcionalitu tak, aby zodpovedala MVC(Model-View-Controller) návrhovému vzoru.

MVC návrhový vzor



Obr. 5.3: MVC návrhový vzor

MVC je návrhový vzor, ktorý sa používa vo väčšine dnešných iOS aplikácií. Tento návrhový vzor nie je exkluzívny pre vývoj aplikácií pre iOS, ale je možné ho použiť aj v iných programovacích jazykoch pre vývoj rôznych iných aplikácií alebo riešení.

- *Model* - V modeli sú uložené všetky dáta s ktorými aplikácia pracuje. Taktiež sú v ňom obsiahnuté triedy a v nich funkcie, ktoré s dátami pracujú, napríklad tu môžeme mať funkciu pre komunikáciu so serverom, ktorá sa dopytuje na dáta potrebné pre chod aplikácie.
- *View* - V pohľade sa riešia veci ohľadom užívateľského rozhrania z grafického pohľadu. Neobsahuje žiadnu logiku ohľadom toho ako má aplikácia fungovať, ale len opisuje grafické prvky kódom ako napríklad farba tlačidla. Jednoducho povedané pohľad sa stará o to, čo užívateľ vidí keď otvorí aplikáciu.
- *Controller* - Kontrolér sa stará o komunikáciu medzi pohľadom a modelom.

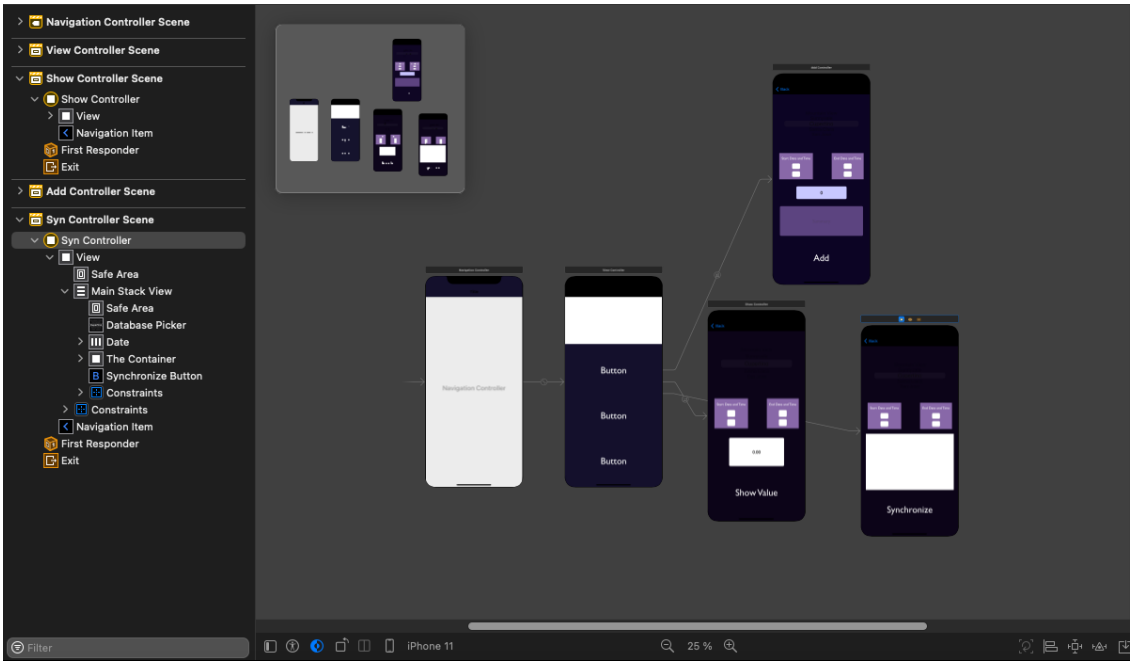
Túto časť častokrát nevieme použiť viackrát, keďže väčšinou obsahuje špecifické pravidlá ako zvyšok aplikácie má medzi sebou komunikovať.

Na Obr. 5.3 môžeme vidieť ako taký bežný tok programu funguje pri použití MVC návrhového vzoru. Užívateľ vykoná nejakú akciu v pohľade, kontrolér následne dá vedieť modelu čo sa má vykonať, model vykoná čo je potrebné a dá vedieť kontroléru, že skončil a kontrolér už len podľa potreby aktualizuje pohľad, ktorý je zobrazovaný užívateľovi.

V mojej implementácii tohto návrhového vzoru je kontrolér a pohľad spojený v rámci jedného súboru, keďže mi to tak prišlo prehľadnejšie a práca s konkrétnymi grafickými prvkami bola jednoduchšia. V komplikovanejších aplikáciach, ktoré obsahujú viac kódu by to ale nebol dobrý nápad, keďže sa v tom jednoducho dá stratiť. Už aj v mojom prípade začala byť práca s kódom trochu komplikovanejšia s tým, ako sa aplikácia rozrastala o novú funkcionalitu.

UIKit

UIKit je *framework*, ktorý nám umožňuje postaviť a spravovať iOS aplikáciu. Na Obr. 5.4 môžeme vidieť *storyboard* našej aplikácie, ktorý obsahuje navigačný kontrolér, ten sa vlastne stará o hierarchiu pohľadov a poskytuje navigačný panel v každom pohľade umiestnený vo vrchnej časti s tlačidlom *Back*.



Obr. 5.4: Storyboard

V *storyboarde* sme do každého pohľadu pridali UI prvky podľa potreby a prechod medzi pohľadmi je docielený pomocou definovaných sekvencii.

Medzi základné použité UI prvky patria:

- *UIButton* - tlačidlo, ktoré po stlačení vykoná nejakú s ním súvisiacu.
- *UIStackView* - kontajner do ktorého vieme nahádzať rôzny počet UI prvkov. Tento kontajner sa stará o to, aby prvky boli uložené v osi či už horizontálnej alebo vertikálnej, taktiež rieši to ako sa dané prvky voči sebe správajú. To znamená to, že koľko priestoru, ktorý zaberie a aké sú medzery medzi nimi a ich pozície.
- *UIPickerView* - koliesko v ktorom máme na výber jednu hodnotu z viacerých nami určených hodnôt.
- *UILabel* - jedná sa len o prvok, ktorý zobrazuje nejakú textovú informáciu.
- *UITextField* - zobrazuje taktiež text, ale v tomto prípade tento prvok predstavuje objekt, ktorý je možné editovať.
- *UIView* - obdĺžnikový kontajner predstavujúci objekt starajúci sa o pohľad.

Ďalej sú v tomto storyboarde pohľady, ktoré využívajú vyššie spomenuté UI prvky. Každý pohľad je spojený s kontrolérom, ktorý vlastní prislúchajúci pohľad.

5.3.2 Classes

V tejto sekcii popíšem modelové triedy v rámci aplikácie. Budem sa hlavne venovať tomu k čomu slúžia a ako fungujú.

HealthStore

Táto trieda pracuje s *frameworkom* HealthKit. Tento *framework* je vlastne jadrom celej aplikácie. Ponúka repozitár všetkých dát ohľadom zdravia s ktorými aplikácia pracuje. K týmto dátam aplikácia dokáže pristupovať len za predpokladu, že užívateľ prístup k týmto dátam povolil.

Zdrojový kód 5.2: HealthStore init

```
static let shared = HealthStore()
var healthStore: HKHealthStore?
var waterLabel: String?
var stepLabel: String?

private init() {
```

```
        if HKHealthStore.isHealthDataAvailable() {
            healthStore = HKHealthStore()
        }
    }
```

V ukážke kódu 5.2 vidíme zadeklarovanie pár základných premenných tejto triedy. Podstatnou časťou ale je vytvorenie *HKHealthStore* objektu. Táto trieda je implementovaná ako *singleton*, keďže v rámci chodu jednej aplikácie budeme vždy potrebovať len jeden *HKHealthStore* objekt. Takže v každom prípade budeme mať len jednu inštanciu HealthStore triedy, ktorá v sebe obsahuje jeden *HKHealthStore* objekt. To nám zabezpečí hneď prvý riadok v ktorom zadefinujeme *shared property*. Funguje to tak, že inštancia tejto triedy sa vytvorí pri prvom zavolaní a potom sa k tomu dá už len pristupovať. K takejto inštancii je potom možné súčasne pristupovať z viacerých vlákien aplikácie bez problémov. V inicializácii tejto triedy sa vlastne už len opýtame či *HKHealthStore* existuje a s ním aj dostupné zdravotné dáta, ak áno tak vytvoríme *HKHealthStore* objekt.

Zdrojový kód 5.3: Metóda žiadosti o autorizáciu

```
func requestAuthorization(completion: @escaping (Bool)
-> Void) {
    let stepType = HKQuantityType.quantityType
    (forIdentifier: HKQuantityTypeIdentifier.stepCount)!
    let waterType = HKObjectType.quantityType
    (forIdentifier: HKQuantityTypeIdentifier.dietaryWater)!

    let healthKitTypesToWrite: Set<HKSampleType> =
    [stepType,
    waterType]

    let healthKitTypesToRead: Set<HKObjectType> =
    [stepType,
    waterType]

    guard let healthStore = self.healthStore else
    { return completion(false) }

    healthStore.requestAuthorization
    (toShare: healthKitTypesToWrite,
```

```
        read: healthKitTypesToRead) { (success, error) in
            completion(success)
        }
}
```

V ukážke kódu 5.3 môžeme vidieť metódu, ktorá sa zavolá pri otvorení aplikácie. Na začiatku zadefinujeme typy dát ku ktorým chceme pristupovať, následne tie typy, ktoré chceme len čítať dáme do *setu* s typmi na čítanie a spravíme to isté aj pre typy s ktorými budeme chcieť vykonávať operáciu zapisovania. Následne skontrolujeme či v našej inštancii triedy je vytvorený *HKHealthStore* objekt a ak áno tak zavoláme na tento objekt metódu, ktorá už existuje v HealthKit *frameworku* s názvom *requestAuthorization*. Do tejto metódy dáme ako vstupné parametre vytvorené sety typov pre čítanie a zápis dát. Ak užívateľ povolí prístup ku všetkým dátam, tak aplikácia bude fungovať podľa očakávania.

Zdrojový kód 5.4: readSteps metóda

```
func readSteps(dateStart: Date, dateEnd: Date, completion:
@escaping(Bool) -> Void){
    guard let stepType = HKQuantityType.quantityType
    (forIdentifier: HKQuantityTypeIdentifier.stepCount)else{
        print("Quantity type not available")
        return
    }
    let sPredicate = HKQuery.predicateForSamples
    (withStart:dateStart,end:dateEnd,options:.strictEndDate)
    let stepQuery = HKStatisticsQuery(
            quantityType: stepType,
            quantitySamplePredicate: sPredicate,
            options: .cumulativeSum
        ) { _, result, _ in
            guard let result = result,
            let sum = result.sumQuantity()else{
                self.stepLabel = String
                (format: "Steps: %.2f", 0)
                completion(false)
                return
            }
            print(result)
```

```
                self.stepLabel = String
                (format: "Steps: %.2f",sum.doubleValue
                (for: HKUnit.count()))
                completion(true)
            }
        self.healthStore?.execute(stepQuery)
}
```

V ukážke kódu 5.4 môžeme vidieť jednu z najhlavnejších metód celej aplikácie. Jedná sa o metódu, ktorá slúži na čítanie počtu krokov v danom časovom rozpätí. Časové rozpätie získame vďaka dvom vstupným parametrom *dateStart* a *dateEnd* reprezentované ako *Date* objekt. Následne zadefinujeme aké dáta chceme čítať. V tomto prípade to budú kroky a skontrolujeme, či takéto dáta alebo takýto typ dát je dostupný. Ak áno tak sa posunieme k zadefinovaniu časového rozpätia pre dopyt, tu len dosadíme do metódy dátumy zo vstupných parametrov a ako tretí parameter pridáme *.strictEndDate*, to spôsobí to, že v záznamoch, ktoré nám dopyt vráti na konci budú len záznamy, ktorých konečný dátum je rovný alebo skôr ako nami zadaný konečný dátum. Ďalej už len zadefinujeme dopyt, v ňom spočítame hodnotu zo všetkých záznamov do jednej a do premennej, ktorá patrí tejto triede túto hodnotu zapíšeme. Na konci metódy tento dopyt zavoláme na objekte *healthStore*. Pri písaní tejto metódy vznikol jeden problém kvôli tomu, že HealthKit dopyty nebežia na rovnakom vlákne ako aplikácia a v dôsledku sú asynchrónne, takže vždy keď som túto metódu zavolal a hneď za ňou som v kontroléry aktualizoval UI tak zobrazovaná hodnota nebola aktuálna, keďže tento dopyt ešte neskončil. Pre zobrazenie správnej hodnoty bolo nutné túto metódu zavolať dvakrát. Tento problém bol vyriešený pomocou *completion handleru*. *Completion handler* je Swift funkcia, ktorá sa volá po vykonaní nejakej úlohy, v tomto prípade po dokončení dopytu. Týmto spôsobom viem kedy metóda skončila a môžem aktualizovať UI. Taktiež to pomáha pri ošetrení chýb, keďže aktualizujem UI len ak metóda skončila správne, ak nie tak stále UI aktualizujem ale s nulovou hodnotou.

Pár ďalších metód v tejto triede vyzerá veľmi podobne, len sa mení typ čítanej alebo zapisovanej hodnoty a zároveň aj dopyt vyzerá trochu inak pri zápise. Ak chceme hodnotu zapísať tak definujeme typ dát aké budeme zapisovať, následne zadefinujeme odkedy dokedy tento záznam trval a hodnotu záznamu. Ak to všetko máme pripravené tak pomocou týchto parametrov vytvoríme záznam a na *healthStore* objekt zavoláme metódu *.save()*, do ktorej ako vstupný parameter vložíme vytvorený záznam. Taktiež pri metódach tohto typu využívam *comple-*

tion handler aby som vedel či bol záznam už úspešne uložený.

Zdrojový kód 5.5: Štruktúra pre zdravotné dáta

```
struct healthModelSample{
    var uuid : String
    var sampleType : String
    var sampleValue : Double
    var startDate : Date
    var endDate : Date
}
```

Štruktúra s názvom *healthModelSample* v ukážke kódu 5.5 slúži na to aby sme nejako pripravili záznam pre synchronizáciu s databázou v nejakom jednotnom formáte. Zatiaľ obsahuje len týchto päť najzákladnejších premenných pre identifikovanie záznamu. Prvá premenná *uuid* je unikátny identifikátor daného záznamu, ďalej tu je typ záznamu, hodnota záznamu od akého dátumu po aký záznam trval. Táto štruktúra mala pôvodne byť v triede SynData, ale ostala nakoniec v tejto triede, lebo pomocné metódy pre získavanie záznamov z *healthStore* objektu sú pisané v tejto triede. Tieto pomocné metódy vyzerajú podobne ako metódy pre čítanie zdravotných dát, ale s tým rozdielom, že sa hodnota nijako nespočítava, ale každý záznam sa prída do poľa objektov, kde jeden objekt predstavuje jednu inštanciu *healthModelSample* štruktúry. Všetky získané záznamy sa prejdú jednoduchým for cyklom a v každej iterácii sa daný záznam pomocou funkcie *.append*() pridá do poľa.

SynData

Táto trieda v konečnom dôsledku slúži len na uchovanie hodnôt ohľadom toho, čo budeme chcieť synchronizovať. Taktiež je implementovaná ako *singleton*, keďže nechceme viac ako jednu inštanciu tejto triedy.

Zdrojový kód 5.6: SynData trieda

```
class SynData{
    let goalNone: [Goal]
    var task: Task
    var selectedGoals: Set<Goal>
    static let shared = SynData()

    init(){
        self.goalNone=[Goal(name:"None")]
```

```
        self.task=Task(name:"",servingGoals:[goalNone[0]])
        self.selectedGoals=Set(arrayLiteral: goalNone[0])
    }
}
```

databaseHelper

Táto trieda v sebe obsahuje metódy na vytvorenie lokálnej SQLite3 databázy a tabuľky s ktorou budeme synchronizovať vybrané zdravotné dáta. Taktiež tu sú metódy na pridanie záznamu do tabuľky v databáze a čítanie hodnôt z databázy.

Zdrojový kód 5.7: SQLite3 metóda vložiť

```
func insert(uuid : String, sampleType : String,
sampleValue: Double, startDate: Date, endDate: Date){
    let formatter=DateFormatter()
    formatter.dateFormat="dd-MM-yyyy HH:mm:ss"
    let dateStartString=formatter.string(from:startDate)
    let dateEndString=formatter.string(from:endDate)
    let query = "INSERT INTO health_db9 (uuid,sampleType,
    sampleValue,startDate,endDate) VALUES (?,?,?,?,?);"
    var statement : OpaquePointer? = nil
    if sqlite3_prepare_v2(db, query, -1, &statement, nil)
    == SQLITE_OK{
        sqlite3_bind_text(statement, 1, (uuid
        as NSString).utf8String, -1, nil)
        sqlite3_bind_text(statement, 2, (sampleType
        as NSString).utf8String, -1, nil)
        sqlite3_bind_double(statement, 3, sampleValue)
        sqlite3_bind_text(statement, 4, (dateStartString
        as NSString).utf8String, -1, nil)
        sqlite3_bind_text(statement, 5, (dateEndString
        as NSString).utf8String, -1, nil)
        if sqlite3_step(statement) == SQLITE_DONE {
            print("Data inserted success")
        }else {
            print("Data is not inserted in table")
        }
    } else {
```

```
        print("Query is not as per requirement")
    }
}
```

Metóda v ukážke kódu 5.7 slúži na pridanie hodnôt do tabuľky. Ako vstupné parametre sú všetky hodnoty, ktoré budeme vyžadovať pre pridanie jedného záznamu. Následne upravíme obidva dátumy do formátu, ktorý je ľahko čitateľný a môže byť uložený do databázy. Vytvoríme SQL(Structured Query Language) dopyt a zadeklarujeme *statement* ukazovateľ. Pripravíme spojenie k databáze a s pomocou pripraveného *statement* ukazovateľa ku ktorému priradíme hodnoty ich zapíšeme do tabuľky do stĺpcov, ktoré sme definovali. Ak všetko prebehlo v poriadku, tak vypíšeme správu o úspechu a v opačnom prípade správu o chybe.

Metóda pre čítanie všetkých hodnôt v tabuľke vyzerá podobne, len s tým rozdielom, že je použitý *while* cyklus aby sme mohli prejsť každý riadok v tabuľke a tieto hodnoty sú následne priraďované do premenných aplikácie. Taktiež proces úpravy dátumu je opačný, keďže z nejakého formátu to naspäť premieňame na *Date* objekt a na konci táto metóda vracia list všetkých riadkov premenených na pole objektov, ktoré zodpovedajú štruktúre *healthModelSample*.

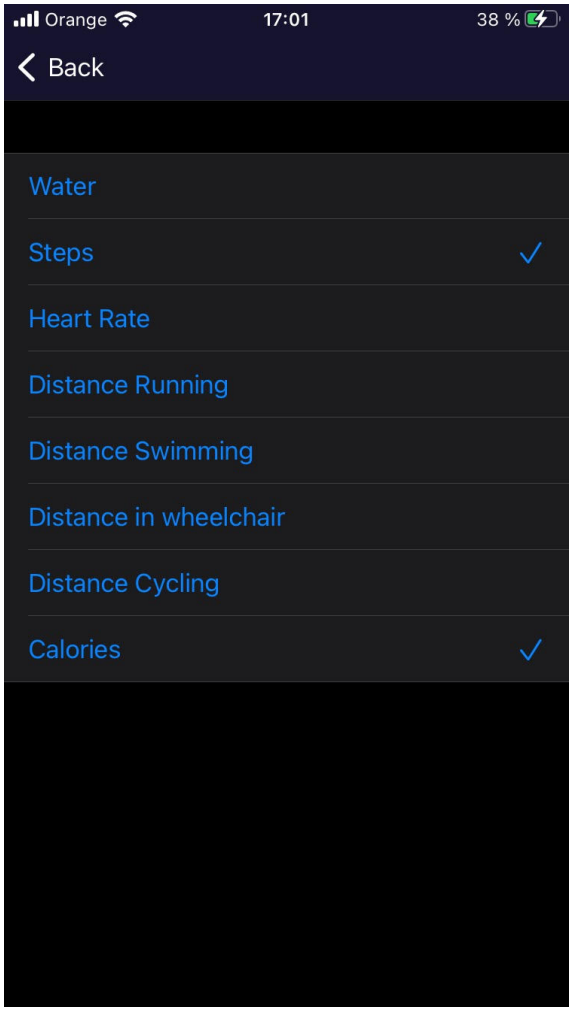
5.3.3 MultiSelector

Táto časť aplikácie sa venuje vytvoreniu poľa v ktorom si môžeme vybrať viacero hodnôt.

Zdrojový kód 5.8: MultiSelector

```
Section(header:Text("Elements chosen to be synchronized")){
    MultiSelector(
        label: Text("Elements"),
        options: allGoals,
        optionToString: { $0.name },
        selected: $task.servingGoals
    )
}
```

Celé toto riešenie je naprogramované pomocou *frameworku* SwiftUI. Do kontajneru v *storyboarde* sme pridali toto SwiftUI riešenie ako samostatný pohľad. Tento pohľad je jediným v tejto aplikácii, ktorý nemá vlastný kontrolér. Všetky zdravotné dáta, ktoré sú na výber sú zadefinované v poli ako štruktúrny objekt *Goal*. V ňom je vlastne len zadefinovaná textová premenná, ktorá predstavuje názov. Pomocou cyklu *ForEach* prejdeme každý tento štruktúrny objekt a pre každý



Obr. 5.5: Obrazovka pre výber dát na synchronizovanie

vytvoríme samostatný *Stack* v ktorom je názov z daného objektu, medzera vytvorená pomocou *Spacer*() a fajka, ktorá je zobrazená, len ak daný objekt bol vybraný. Vybrané objekty si ukladáme do triedy SynData. To je kvôli tomu, aby naša voľba zostala v takom stave v akom je aj keď sa rozhodneme daný pohľad opustiť a vrátiť sa k nemu neskôr. Zbavíme sa tým zbytočného výberu toho istého.

Na Obr. 5.5 vidíme ako táto obrazovka pre výber zdravotných dát na synchronizáciu vyzerá v aplikácii.

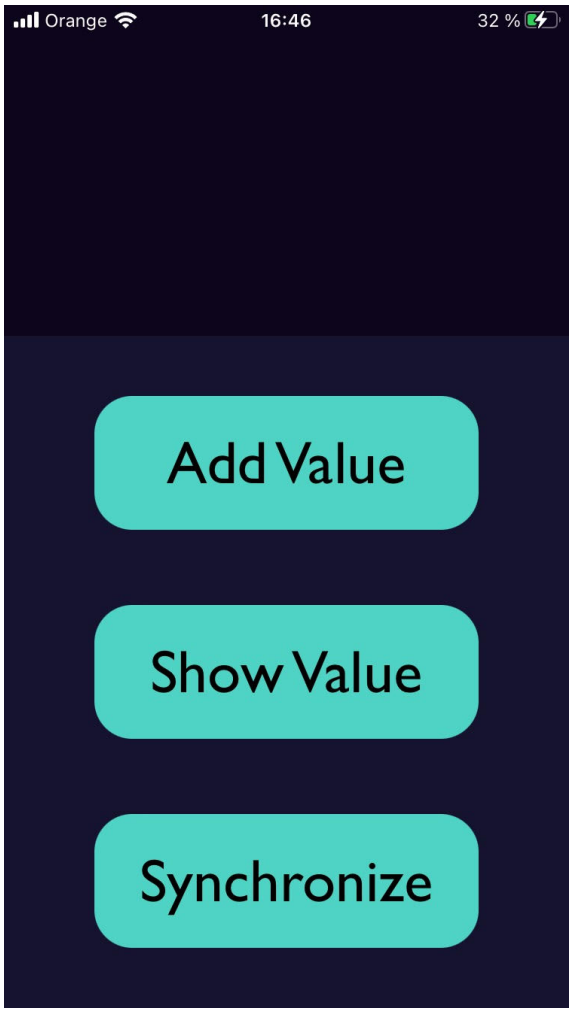
5.3.4 Controllers

ViewController

Zdrojový kód 5.9: UIButton prispôsobenie

```
addButton.setTitle("Add Value", for: .normal)
addButton.setTitleColor(.black, for: .normal)
```

```
addButton.backgroundColor=hexStringToUIColor(hex:"#4FD3C4")
addButton.layer.cornerRadius = 25
addButton.titleLabel?.font=UIFont(name: "GillSans",size:40)
addButton.titleLabel?.minimumScaleFactor = 0.2
addButton.titleLabel?.numberOfLines = 1
addButton.titleLabel?.adjustsFontSizeToFitWidth = true
addButton.titleLabel?.textAlignment = .center
```



Obr. 5.6: Úvodná obrazovka

Toto je kontrolér pre úvodnú obrazovku. Je v ňom len kód pre grafické prispôsobenie tlačidiel, ktoré sú úplne identické až na zobrazený text a taktiež sa tu volá metóda z triedy HealthStore, ktorá zabezpečuje, že sa od užívateľa vyžiada povolenie o prístup k zdravotným dátam.

Na Obr. 5.6 vidíme ako táto úvodná obrazovka vyzerá v aplikácii.

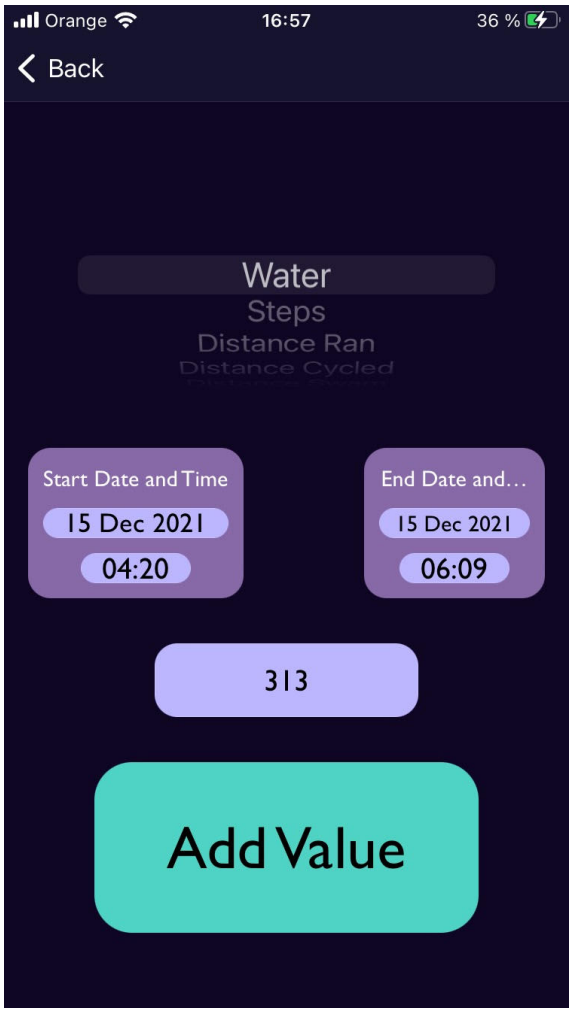
AddDataView

Zdrojový kód 5.10: Pridanie dát

```
@IBAction func addClicked(_ sender: UIButton) {
    switch selectedType {
    case healthTypes[0]:
        HealthStore.shared.writeWater(waterConsumed:
        (inputText.text! as NSString).doubleValue,
        dateStart:startDateFinal1!,dateEnd:endDateFinal1!)
        {success in
            if success{
                print("Water saved successfully")
            }else{
                print("Water was not saved successfully")
            }
        }
    case healthTypes[1]:
        HealthStore.shared.writeSteps(stepCountValue:
        (inputText.text! as NSString).doubleValue,
        dateStart:startDateFinal1!,dateEnd:endDateFinal1!)
        {success in
            if success{
                print("Steps saved successfully")
            }else{
                print("Steps were not saved successfully")
            }
        }
    default:
        print("Error")
    }
}
```

Úlohou tohto kontroléru je starať sa o pohľad v ktorom užívateľ vie manuálne pridať zdravotné dáta podľa potreby. Obsahuje inicializované premenné užívateľského rozhrania a spolu s nimi aj kódom definované grafické úpravy týchto elementov. V ukážke kódu 5.10 vidíme *switch* blok, ktorý sa stará o to, ktorá funkcia z HealthStore triedy sa má zavolať podľa výberu používateľa. Je tu definované pole možných hodnôt, ktoré sú užívateľovi dostupné na výber cez *UIPickerView* v hor-

nej časti obrazovky. Kvôli funkcionalite tohto elementu je vždy možné vybrať len jednu hodnotu.



Obr. 5.7: Obrazovka pre pridanie novej hodnoty

Vstupné parametre ako počiatočný dátum, konečný dátum a hodnota pre zápis získame z ostatných elementov v tomto pohľade. Každý výber dátumu a času, ktorý je s ním spätý je vytvorený samostatne po kliknutí na dané textové pole spojené s konkrétnou hodnotou. Po kliknutí na dané textové pole sa zobrazí výber dátumu alebo času a následne po kliknutí na tlačidlo *Done*, ktoré je súčasťou tohto vytvoreného elementu sa hodnota zapíše do daného textové poľa z ktorého vieme túto hodnotu následne čítať. Keďže celkový dátum pozostáva z dátumu a času tak tieto dve hodnoty musíme pomocou dátumového *formatteru* spojiť naspäť do jedného dátumového objektu. To sa deje nad už vyššie spomínaným *switch* blokom a takto upravený dátumový objekt ide ako vstupný parameter do funkcie *writeWater*.

Sú tu taktiež implementované *delegate* funkcie pre textové pole vstupnej hodnoty a pre *UIPickerView*, kde si užívateľ vyberá aký typ hodnoty chce zapísať. *De-*

legate funkcia je vlastne funkcia, ktorá mení správanie nejakého objektu. *Delegate* funkcia pre textové pole vstupnej hodnoty funguje tak, že po každom editovaní daného poľa sa skontroluje či je vstup validný, a ak nie je tak užívateľovi vymaže to čo napísal, respektívne mu nedovolí dopísať neplatný vstup. Platný vstup je definovaný ako číslo s jednou desatinnou čiarkou a okrem toho sú za platné znaky považované len čísla.

Na Obr. 5.7 môžeme vidieť ako vyzerá obrazovka v aplikácii za ktorú zodpovedá tento kontrolér.

ShowDataView

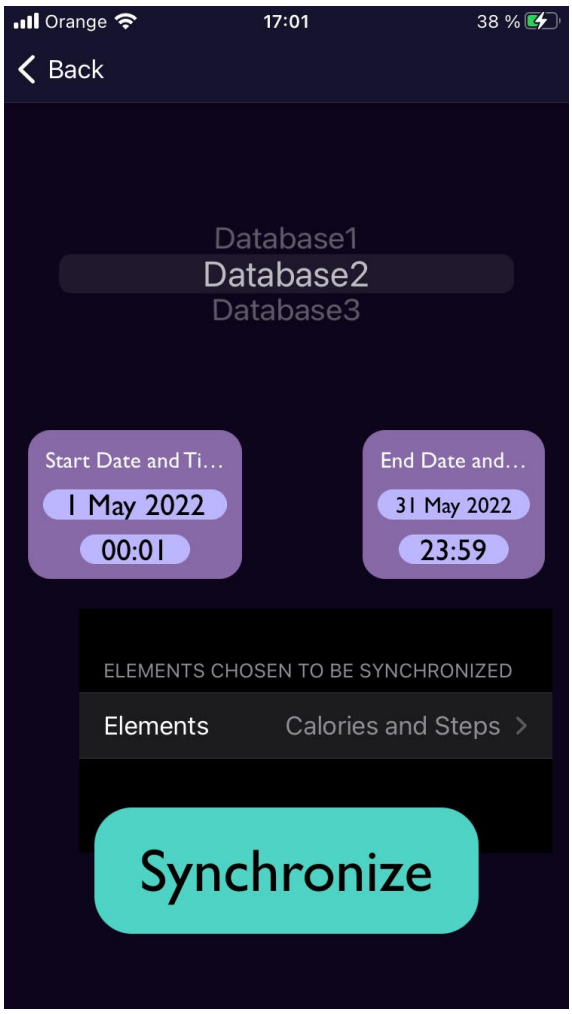
Obr. 5.8: Zobrazenie hodnoty

Tento kontrolér vyzerá rovnako ako AddDataView, len s tým rozdielom, že funkcie, ktoré sa volajú z HealthStore triedy po stlačení tlačidla sú iné a taktiež mu chýba *delegate* funkcia pre vstupnú hodnotu, keďže v tomto pohľade žiadnu vstupnú hodnotu okrem výberu toho čo chceme vidieť a dátumu nemáme. Čo sa

týka funkcionality toho ako funguje výber dátumu a následné spojenie dátumu a času do jedného dátumového objektu sú tieto dva kontroléry identické.

Na Obr. 5.8 vidíme obrazovku za ktorú zodpovedá tento kontrolér. Táto obrazovka je takmer identická s obrazovkou na Obr. 5.7 s tým rozdielom, že namiesto hodnoty, ktorú chceme zapísať tu je hodnota, ktorú chceme zobraziť podľa nášho výberu za dané obdobie. Pri zmene zobrazenej hodnoty sa menia aj jednotky hodnoty podľa toho, čo vyberieme.

SynDataView



Obr. 5.9: Obrazovka pre synchronizáciu

Tento kontrolér sa stará o logiku ohľadom synchronizovania dát s lokálne vytvorenou SQLite databázou. Čo sa týka výberu dátumu a jeho spájania, tak táto funkcionalita je vyriešená identicky s vyššie spomenutými kontrolérmi. Čo sa týka funkcionality navyše, tak výber zvolenej databázy je uložený do užívateľských predvolieb a po každom novom načítaní tohto pohľadu sa z týchto pred-

volieb načíta pôvodná hodnota a zvolí sa daná databáza ešte pred zobrazením pohľadu. Tento výber databázy je vytvorený len pre budúcu funkcionalitu, keďže funkcionalita synchronizácie zatiaľ funguje len pre jednu lokálnu databázu, ktorú vytvoríme pri zapnutí aplikácie. Vytvárame tu *UIView* kontajner, ktorému priraďujeme vyššie spomínaný MultiSelector. Týmto vlastne kombinujeme dva rôzne prístupy pre vývoj grafického rozhrania, keďže väčšina aplikácie je vyvíjaná pomocou UIKit a *Storyboardu*, ale tu pridávame aj prvky SwiftUI.

Zdrojový kód 5.11: Synchronizácia dát

```
if(SynData.shared.selectedGoals
.contains(Goal(name: "Water"))){
HealthStore.shared.readWaterForDB(dateStart:
startDateFinal!, dateEnd: endDateFinal!){success in
    if success{
        DispatchQueue.main.async {
        self.db.listOfHealthData.removeAll()
        self.db.listOfHealthData = self.db.read()
        for healthSample in HealthStore.shared
        .samplesToSyncWater{
            insertOrNot = true
            for healthEntry in self.db.listOfHealthData{
                if(healthSample.uuid == healthEntry.uuid){
                    insertOrNot = false
                }
                //print(healthSample.sampleValue)
            }
            if(insertOrNot){
                self.db.insert(uuid: healthSample.uuid,
                sampleType: healthSample.sampleType,
                sampleValue: healthSample.sampleValue,
                startDate: healthSample.startDate,
                endDate: healthSample.endDate)
            }
        }
        print("Synchronized Water")
        }
    }else{
        DispatchQueue.main.async {
```

```
                print("Big Sync Error")
            }
        }
}
}
```

V ukážke kódu 5.11 môžeme vidieť ako funguje logika synchronizácie v rámci tejto aplikácie. Najprv skontrolujeme, či hodnota, ktorú chceme synchronizovať bola vybraná v rámci MultiSelector pohľadu. V tomto konkrétnom prípade kontrolujeme len to, či chceme synchronizovať príjem tekutín. Ak áno, tak zavoláme metódu *readWaterForDB* z HealthStore triedy, ktorá sa postará o načítanie všetkých záznamov v zadanom období do poľa *samplesToSyncWater*. Následne máme *for* cyklus, ktorý každou touto hodnotou prejde a vo vnorenom *for* cykle porovná s každou už existujúcou hodnotou v databáze. Ak sa takýto záznam v databáze nenachádza, tak ho tam na konci danej iterácie prvého *for* cyklu pridáme a to vlastne spravíme pre každý záznam v danom období.

Na Obr. 5.9 vidíme obrazovku za ktorú zodpovedá tento kontrolér. Nad tlačidlom *Synchronize* je *UIView* kontajner pomocou, ktorého si vieme vybrať viacero typov zdravotných dát na synchronizovanie.

6 Vyhodnotenie

V tejto časti bakalárskej práce opíšem spôsob merania, vykonané merania, prínos tejto práce a spíšem zhrnutie tejto práce v pár bodoch.

V ukážke kódu 6.1 je ukázané ako prebiehali všetky merania. Na začiatku funkcie, ktorá sa zavolá po stlačení tlačidla napríklad na synchronizáciu sa vytvorí premenná do ktorej sa uloží štartovací čas, teda aktuálny čas v danom momente kliknutia na tlačidlo. Na konci meranej funkcie sa vytvorí premenná, kde sa uloží aktuálny čas po vykonaní všetkého potrebného v rámci funkcie. Následne tieto dve premenné odčítame, aby sme získali rozdiel medzi časmi a to všetko je vypočítané s presnosťou na nanosekundy. Túto hodnotu prevedieme na sekundy a vypíšeme výsledok do konzoly.

Zdrojový kód 6.1: Spôsob merania

```
let start = DispatchTime.now()//Start Time
//Merana funkcia
let end = DispatchTime.now()//End Time
let nanoTime=end.uptimeNanoseconds-start.uptimeNanoseconds
let timeInterval = Double(nanoTime) / 1_000_000_000
print("Time: " + timeInterval)
```

Tabuľka 6.1: Porovnanie hosťovského systému s VM

	Host	VM
CPU	AMD Ryzen 5 2600 12 vlákien 4 GHz	AMD Ryzen 5 2600 8 vlákien 4 GHz
RAM	16 GB 3200 MHz	12 GB 3200 MHz
Disk	Samsung 970 EVO Plus NVMe M.2 SSD 500 GB	Kingston A400 SATA SSD 100 GB

Merania boli vykonané na fyzickom zariadení iPhone 6S s nainštalovaným

operačným systémom iOS 15.4 a na simulátore iPhone 11, ktorý ponúka Xcode s rovnakým operačným systémom. Je nutné podotknúť, že tento simulátor beží v rámci VM. Technické špecifikácie stolného počítača na ktorom beží hosťovský operačný systém Windows 10 a technické špecifikácie VM s nainštalovaným macOS sú zobrazené v tabuľke 6.1.

6.1 Prvé meranie

V prvom meraní sme zmerali ako dlho trvá zápis alebo pridanie jednej hodnoty do *HKHealthStore* objektu s použitím našej aplikácie. Jednalo sa o pridanie hodnoty prijatých tekutín, konkrétne o 1998 ml s časovým ohraničením, ktoré predstavovalo približne 12 hodín. V tabuľke 6.2 vidíme, že namerané časy sú veľmi nízke. Ak porovnáme časy medzi simulátorom a fyzickým zariadením tak vidíme, že pridávanie hodnoty je rýchlejšie v simulátore, čo je pravdepodobne spôsobené tým, že vykonaný kód beží na oveľa rýchlejšom procesore ako je použitý v iPhone 6S, ktorý je už relatívne zastaralý a kvôli obmedzeniam ohľadom veľkosti menej výkonný. Každopádne tieto rozdiely sú minimálne a celkový čas, ktorý daná úloha zaberie je tak nízky, že tieto rozdiely sú úplne zanedbateľné z pohľadu používania.

Tabuľka 6.2: Pridanie hodnôt

Č.	iPhone 11 Simulator	iPhone 6S
1.	10.971 ms	32.239 ms
2.	4.454 ms	16.137 ms
3.	5.917 ms	17.115 ms
4.	7.381 ms	18.012 ms
5.	4.455 ms	11.302 ms
6.	4.521 ms	6.044 ms
7.	4.776 ms	18.336 ms
8.	5.011 ms	17.090 ms
9.	4.601 ms	7.285 ms
10.	3.898 ms	16.164 ms

6.2 Druhé meranie

V prípade druhého merania sme zmerali ako dlho trvá čítanie hodnoty pri použití našej aplikácie. V tomto konkrétnom prípade sme chceli zobraziť celkový počet krokov za určité obdobie. Ako v predchádzajúcom meraní, tak aj tu budeme porovnávať simulátor a fyzické zariadenie. Počet záznamov z ktorých finálnu zobrazenú hodnotu vypočítame sa samozrejme mení podľa rozsahu zadaného obdobia z ktorého chceme hodnotu získať a zvýšený počet záznamov s ktorými pracujeme znamená vyššiu náročnosť na výpočtovú silu.

Tabuľka 6.3: Čítanie hodnôt

	A		B	
Č.	iPhone 11 Simulator	iPhone 6S	iPhone 11 Simulator	iPhone 6S
1.	10.008 ms	13.970 ms	16.593 ms	38.989 ms
2.	6.498 ms	17.593 ms	12.301 ms	35.147 ms
3.	6.199 ms	10.788 ms	12.891 ms	29.518 ms
4.	5.219 ms	12.651 ms	11.591 ms	35.272 ms
5.	5.253 ms	10.285 ms	11.991 ms	22.546 ms
6.	5.287 ms	31.225 ms	11.897 ms	38.956 ms
7.	5.529 ms	19.180 ms	10.108 ms	33.993 ms
8.	6.971 ms	36.336 ms	11.729 ms	34.482 ms
9.	7.390 ms	18.711 ms	12.951 ms	34.344 ms
10.	5.744 ms	12.786 ms	12.518 ms	41.985 ms

Cieľom tohto merania teda bolo nie len porovnať simulátor a fyzické zariadenie, ale aj to ako sa simulátor a fyzické zariadenie popasujú s väčším počtom záznamov. Tabuľka 6.3 je rozdelená na časť A a časť B. V časti A sú hodnoty namerané ak sme ako požiadavku zadali počet krokov za posledné dva mesiace. Celkový počet záznamov pre výpočet danej hodnoty predstavoval 265 záznamov v prípade simulátoru a 293 záznamov v prípade fyzického zariadenia, kde väčšina týchto záznamov bola úplne identická. V časti B sme zmerali hodnotu pre rovnakú požiadavku, len v tomto prípade časový rozsah bol zväčšený na posledné štyri roky. V prípade simulátora sa tu pracovalo s 4273 záznamami a v prípade fyzického zariadenia bol počet záznamov 4310 z ktorých väčšina bola taktiež úplne identická. Tak ako v predchádzajúcom meraní, tak aj tu je simulátor v obidvoch prípadoch zase rýchlejší kvôli dôvodom už spomenutým. Ako bolo očakávané, tak so zvýšeným počtom záznamov stúpol aj čas za ktorý sa funkcia vykonala.

Ako tomu bolo aj v predchádzajúcom meraní, tak všetky tieto rozdiely sú z užívateľského pohľadu zanedbateľné, keďže čas za ktorý sa funkcia vykonala je veľmi malý v každom prípade, či už na fyzickom zariadení alebo na simulátore a taktiež aj so zvýšeným počtom záznamov v obidvoch prípadoch.

6.3 Tretie meranie

V rámci tretieho merania sme zmerali ako dlho trvá synchronizácia hodnôt s lokálnou databázou s použitím synchronizačného algoritmu v našej aplikácií. Synchronizované boli len záznamy, ktoré obsahovali informácie o prijatých tekutinách a počtu prejdených krokov. Tabuľka je zase rozdelená na časť A a časť B, kde v časti A je zapísaný čas za aký sa zosynchronizujú údaje za posledné dva mesiace a v časti B za posledné 4 roky. Celkový počet záznamov, ktoré sa mali zosynchronizovať pre simulátor v časti A bol 275 a pre fyzické zariadenie 299. V časti B bol počet záznamov určených na synchronizáciu 4283 v prípade simulátora a 4326 v prípade fyzického zariadenia. Samozrejme ako aj v predchádzajúcom meraní, tak aj v tomto boli záznamy z veľkej časti identické ak porovnáme simulátor a fyzické zariadenie či už v časti A alebo v časti B.

Tabuľka 6.4: Synchronizovanie hodnôt

	A		B	
Č.	iPhone 11 Simulator	iPhone 6S	iPhone 11 Simulator	iPhone 6S
1.	0.479 s	0.533 s	9.914 s	12.778 s
2.	0.218 s	0.322 s	11.214 s	14.627 s
3.	0.202 s	0.325 s	11.013 s	14.686 s
4.	0.230 s	0.326 s	10.998 s	14.646 s
5.	0.205 s	0.332 s	10.950 s	14.681 s
6.	0.227 s	0.328 s	11.013 s	14.433 s
7.	0.198 s	0.352 s	11.087 s	14.561 s
8.	0.228 s	0.324 s	10.934 s	14.638 s
9.	0.196 s	0.334 s	10.992 s	14.648 s
10.	0.206 s	0.326 s	11.117 s	14.507 s

Simulátor je zase rýchlejší za približne rovnakých podmienok. Pri veľkom počte záznamov na synchronizovanie je dokonca približne o 3 sekundy rýchlejší ako fyzické zariadenie. Ďalšiu vec, ktorú môžeme z tabuľky vyčítať je tá, že v porovnaní s predchádzajúcimi dvoma meraniami proces synchronizácie zaberá oveľa

viac času aj pri malom počte záznamov. So zvyšujúcim sa počtom záznamov na synchronizáciu prudko stúpa čas za ktorý sa to zosynchronizuje. Zaujímavé je, že čas nameraný v rámci prvého riadku sa líši od zvyšku nameraných hodnôt v ďalších riadkoch. Dokonca v časti A je väčší ako vo zvyšku a v časti B je menší ako vo zvyšku. Tento rozdiel je spôsobený tým, že pri prvom meraní je vždy lokálna databáza prázdna a kvôli tomu sa tieto hodnoty len zapisujú do databázy. V ďalších meraniach sa už žiadne hodnoty do databázy nepridali kvôli tomu, že všetky tam už sú a jediné čo sa dialo je to, že sa kontrolovalo či tam dané hodnoty naozaj sú a netreba ich pridať.

6.4 Zhodnotenie

Vo vyššie popísaných troch meraniach sme si ukázali, že funkcionalita pridávania a čítania hodnôt je rýchla priam až okamžitá aj na trochu zastaralejšom zariadení a, že ani v porovnaní so simulátorom vo výkonnej VM veľmi nezaostáva. Synchronizácia je náročnejší proces a je pochopiteľné, že trvá najdlhšie a vôbec nie je okamžitá. Čo sa týka porovnania výkonu medzi fyzickým zariadením a simulátorom, tak simulátor bol aj napriek môjmu očakávaniu rýchlejší v každom meranom prípade. Moje očakávania pre simulátor boli nižšie kvôli tomu, lebo beží vo VM na systéme, ktorý nie je najlepšie optimalizovaný na daný hardvér, ale merania ukázali, že čistý výkon procesora v stolnom počítači je postačujúci na to aby bol dostatočne rýchly pri takomto použití. Predpokladám, že novšie zariadenia od Applu by boli pravdepodobne najrýchlejšie vďaka dobrej optimalizácií, novej architektúre procesoru a tým pádom vyšším frekvenciám v procesore, čo znamená vyšší výkon.

V rámci teoretického prínosu tejto bakalárskej práce sme si popísali najpopulárnejšie zdravotné databázy, povedali si o možnostiach ohľadom vývoja aplikácií pre systém iOS, popísali vývojové prostredie Xcode a do hĺbky priblížili programovací jazyk Swift a nový *framework* na tvorbu užívateľského rozhrania s názvom SwiftUI.

Čo sa týka praktického prínosu tejto bakalárskej práce, tak sme ukázali ako je možné vyvíjať iOS aplikáciu aj bez zariadenia s macOS. Ďalej sme navrhli dizajn riešenia pomocou nástroja Figma a najväčším praktickým prínosom tejto práce je samotné naprogramovanie celej aplikácie.

7 Zhrnutie

Teoretická časť

- Opis najpopulárnejších zdravotných databáz
- Oboznámenie sa s možnosťami vývoja aplikácií pre iOS
- Popis vývojového prostredia Xcode
- Vysvetlenie programovacieho jazyku Swift
- Priblíženie *frameworku* na vývoj grafického rozhrania s názvom SwiftUI

Praktická časť

- Návrh užívateľského rozhrania pre naše riešenie s pomocou nástroja Figma
- Inštalácia potrebného softvéru k vývoju iOS aplikácie
- Návrh a realizácia funkcií na zápis a čítanie zdravotných dát cez HealthKit *framework*
- Návrh a realizácia funkcionality synchronizácie s lokálnou SQLite databázou
- Realizácia aplikácie z pohľadu implementovania grafického užívateľského rozhrania
- Merania a zhodnotenie výsledkov

8 Záver

Cieľom tejto bakalárskej práce bolo navrhnúť a naprogramovať iOS aplikáciu pre synchronizáciu s externým úložiskom podľa výberu užívateľa. Aplikáciu sa podarilo navrhnúť a naprogramovať. Z pohľadu návrhu pôsobí aplikácia intuitívne a prívetivo. Čo sa týka funkcionality, tak okrem synchronizácie je v aplikácii aj možnosť pridať nové dáta do zariadenia a čítať dáta už uložené v zariadení. Synchronizáciu s externým úložiskom sa v konečnom dôsledku nepodarilo implementovať, ale podarilo sa implementovať synchronizáciu s lokálnou SQLite databázou, takže algoritmus synchronizácie je naprogramovaný a bol taktiež otestovaný a táto funkcionalita funguje, ale ešte to treba prepojiť s SQLite databázou, ktorá by bola mimo zariadenia. Vyhodnotenie ukázalo, že naša aplikácia funguje relatívne rýchlo aj na staršom zariadení.

Najväčším problémom pri implementácií bola celková práca s HealthKit *frameworkom* a práca so zdravotnými dátami v rámci tohto *frameworku*, napríklad zápis a čítanie rôznych typov dát, ktoré HealthKit ponúka. Taktiež problémom bolo implementovanie UI, ale po prejdení na UIKit *framework* sa tento problém vyriešil do značnej miery.

Do budúcnosti má táto aplikácia veľa priestoru na zlepšenie. Napríklad ponúknuť nejaké východzie možnosti výberu pre užívateľa ohľadom zvoleného dátumu ako za posledný deň, týždeň alebo mesiac. Optimalizovať algoritmus synchronizácie a celkovo implementovať aj možnosť zápisu, čítania a synchronizácie pre ďalšie rôzne typy dát, ktoré momentálne nie sú implementované. Ďalším užitočným vylepšením tejto aplikácie by bolo pridanie funkcionality na analýzu dát a to tak, že sa užívateľovi zobrazia dáta za určité obdobie v ľahko čitateľných grafoch a aplikácia vygeneruje informáciu ohľadom trendov z týchto grafov v podobe správy pre užívateľa.

Literatúra

1. REEDER, Blaine; DAVID, Alexandria. Health at hand: A systematic review of smart watch uses for health and wellness. *Journal of biomedical informatics*. 2016, roč. 63, s. 269–276.
2. KATALOV, Vladimir. Apple Health Is the Next Big Thing: Health, Cloud and Security. 2018. Dostupné tiež z: https://blog.elcomsoft.com/2018/11/apple-health-is-the-next-big-thing-health-cloud-and-security/.
3. SINHASANE, Shailendra. Significance of iOS 15 Health Data App Features for Sharing Health Records. 2021. Dostupné tiež z: https://mobisoftinfotech.com/resources/blog/health-data-app-for-sharing-health-records/.
4. CAPRITTO, Amanda. The complete guide to Apple's Health app. 2019. Dostupné tiež z: https://www.cnet.com/health/the-complete-guide-to-apples-health-app/.
5. How to sync your Health data to iCloud. 2017. Dostupné tiež z: https://www.imore.com/how-sync-your-health-data-ios-11-and-how-it-works.
6. WESTENBERG, Jimmy. Google Fit guide: Everything you need to know about Google's fitness platform. 2022. Dostupné tiež z: https://www.androidauthority.com/google-fit-393110/.
7. SINGH, Rahul. Apple's HealthKit vs Google Fit: Who is winning the next important domain invaded by technology? 2018. Dostupné tiež z: https://www.promaticsindia.com/blog/apples-healthkit-vs-google-fit-who-is-winning-the-next-important-domain-invaded-by-technology/.
8. Samsung Health: Keď chcete mať svoje zdravie a športové aktivity kompletne pod kontrolou. 2021. Dostupné tiež z: https://www.mojandroid.sk/samsung-health-sport-zdravie/.

9. CIMINO, Kaitlyn. What is Samsung Health? Everything you need to know. 2021. Dostupné tiež z: https://www.androidauthority.com/samsung-health-3037491/.

10. HALL, Chris. What is Strava, how does it work and is it worth paying for? 2022. Dostupné tiež z: https://www.pocket-lint.com/apps/news/154854-what-is-strava-and-how-does-it-work.

11. CHUNG, Eddy. What is Xcode and why do I need it? [B.r.]. Dostupné tiež z: https://www.zerotoappstore.com/what-is-xcode-and-why-do-i-need-it.html.

12. Creating an Xcode Project for an App. [B.r.]. Dostupné tiež z: https://developer.apple.com/documentation/xcode/creating-an-xcode-project-for-an-app.

13. Learn how to build apps: Magic 8-Ball. [B.r.]. Dostupné tiež z: https://makeschool.org/mediabook/oa/tutorials/learn-how-to-build-apps--magic-8-ball/new-xcode-project/.

14. CHING, Chris. Xcode Tutorial For Beginners. 2019. Dostupné tiež z: https://codewithchris.com/xcode-tutorial/.

15. AVDIC, Dervis. React native vs xamarin-mobile for industry. 2019.

16. Dokumentácia Xamarin.iOS. [B.r.]. Dostupné tiež z: https://docs.microsoft.com/en-us/xamarin/ios/.

17. FOJTIK, Rostislav. Swift a new programming language for development and education. In: *The 2018 International Conference on Digital Science*. 2019, s. 284–295.

18. REBOUÇAS, Marcel; PINTO, Gustavo; EBERT, Felipe; TORRES, Weslley; SEREBRENIK, Alexander; CASTOR, Fernando. An empirical study on the usage of the swift programming language. In: *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*. 2016, zv. 1, s. 634–638.

19. The Good and the Bad of Swift Programming Language. 2021. Dostupné tiež z: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-swift-programming-language/.

20. HOFFMAN, Jon. *Mastering Swift 5.3: Upgrade your knowledge and become an expert in the latest version of the Swift programming language*. Packt Publishing Ltd, 2020.

21. Swift Dokumentácia. 2022. Dostupné tiež z: `https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html`.

22. Swift - The powerful programming language that is also easy to learn. 2022. Dostupné tiež z: `https://developer.apple.com/swift/`.

23. Swift - Protocols. [B.r.]. Dostupné tiež z: `https://www.tutorialspoint.com/swift/swift_protocols.htm`.

24. CASSEE, Nathan; PINTO, Gustavo; CASTOR, Fernando; SEREBRENIK, Alexander. How swift developers handle errors. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018, s. 292–302.

25. ABHIMURALIDHARAN. Type casting in swift : difference between is, as, as?, as! 2017. Dostupné tiež z: `https://abhimuralidharan.medium.com/typecastinginswift-1bafacd39c99`.

26. VARMA, Jayant. *SwiftUI for Absolute Beginners*. Springer, 2019.

27. CAHILL, Bear. *UI Design for iOS App Development: Using SwiftUI*. Springer, 2021.

28. Mastering SwiftUI. 2021. Dostupné tiež z: `https://www.appcoda.com/learnswiftui/`.

29. WALS, Donny. SwiftUI Property Wrappers. 2020. Dostupné tiež z: `https://swiftuipropertywrappers.com/`.

Zoznam skratiek

ABI Application Binary Interface.

API Application Programming Interface.

ARC Automatic Reference Counting.

BIOS Basic Input Output System.

IDE Integrated Development Environment.

LLVM Low Level Virtual Machine.

MVC Model-View-Controller.

SQL Structured Query Language.

UI User Interface.

VM Virtual Machine.

WCF Windows Communication Foundation.

WWDC Apple Worldwide Developers Conference.

Zoznam príloh

Príloha A CD médium – záverečná práca v elektronickej podobe

Príloha B Používateľská príručka

Príloha C Systémová príručka

Príloha D Dokumentácia podľa pokynov výskumnej skupiny IKS - https://tukesk.sharepoint.com/sites/Archiv_ZP_IKS_KKUI