

Mobilní aplikace pro plánování nabíjení elektrických vozidel

Bc. Petr Jelínek

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Petr Jelínek
Osobní číslo:	A22399
Studijní program:	N0613A140022 Informační technologie
Specializace:	Softwarové inženýrství
Forma studia:	Kombinovaná
Téma práce:	Mobilní aplikace pro plánování nabíjení elektrických vozidel
Téma práce anglicky:	Mobile Application for Planning Electric Vehicle Charging

Zásady pro vypracování

- Vypracujte literární rešerši na téma vývoje mobilních aplikací pro operační systém iOS, zaměřte se na vhodné programovací jazyky, popis důležitých frameworků a vývojářské nástroje.
- Navrhněte UX/UI mobilní aplikace pro plánování nabíjení elektrických vozidel.
- Využijte cloudových služeb Firebase zejména pro různé způsoby autentizace uživatele a použité služby popište.
- Provedte implementaci prototypu mobilní aplikace dle návrhu.
- Ověřte funkčnost aplikace.

Forma zpracování diplomové práce: tištěná/elektronická

Seznam doporučené literatury:

1. MARTIN, Robert C. Clean code: a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, c2009. ISBN 9780132350884.
2. MARTIN, Robert C. Clean architecture: a craftsman's guide to software structure and design. London, England: Prentice Hall, [2018]. ISBN 0134494164.
3. FOWLER, Martin. Refaktoring: zlepšení existujícího kódu. Praha: Grada, 2003. Moderní programování. ISBN 80-247-0299-1.
4. APPLE. *Swift – Apple Developer*. Online. 2024. Dostupné z: <https://developer.apple.com/swift>. [cit. 2024-09-06].
5. APPLE. *SwiftUI – Xcode – Apple Developer*. Online. 2024. Dostupné z: <https://developer.apple.com/xcode/swiftui>. [cit. 2024-09-06].
6. APPLE. *Apple Developer Documentation*. Online. 2024. Dostupné z: <https://developer.apple.com/documentation>. [cit. 2024-09-06].
7. GOOGLE. *Firestore | Firebase*. Online. 2024. Dostupné z: <https://firebase.google.com/docs/firestore>. [cit. 2024-09-06].
8. GOOGLE. *Firebase Authentication*. Online. 2024. Dostupné z: <https://firebase.google.com/docs/auth>. [cit. 2024-09-06].
9. GOOGLE. *Cloud Functions for Firebase*. Online. 2024. Dostupné z: <https://firebase.google.com/docs/functions>. [cit. 2024-09-06].

Vedoucí diplomové práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval samostatně a použitou literaturu jsem řádně citoval; v případě publikace výsledků budu uveden jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.
- Prohlašuji, že při tvorbě této práce jsem použil nástroj generativního modelu AI ChatGPT (Model 4o) za účelem úpravy stylistiky, kontroly gramatiky a generování základních ukázek kódu. Po použití tohoto nástroje jsem provedl kontrolu obsahu a přebírám za něj plnou zodpovědnost.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Hlavním cílem této diplomové práce je usnadnit přechod uživatelů z vozidel se spalovacím motorem na elektromobily prostřednictvím mobilní aplikace Volture. Aplikace poskytuje uživatelům nástroj pro plánování nabíjení, čímž minimalizuje dodatečnou zátěž spojenou s plánováním a provozem elektromobilu. Díky aplikaci se mohou uživatelé efektivně rozhodovat o tom, kdy a kde nabíjet, odhadnout potřebnou úroveň nabití a určit dobu potřebnou k dosažení cílového stavu baterie. Kromě toho aplikace nabízí statistiky a přehledy, které pomáhají uživatelům optimalizovat nabíjecí návyky a celkovou správu vozidla.

Klíčová slova: elektromobil, nabíjení, baterie, iOS, Apple, SwiftUI

ABSTRACT

The main goal of this thesis is to facilitate the switch from combustion engine vehicles to electric vehicles with the help of the Volture application. The app provides a simple tool for charge planning, reducing the extra effort needed to manage and operate an electric vehicle. With Volture, users can easily decide when and where to charge, estimate the required battery level, and calculate the time needed to reach their target state of charge. In addition, the app offers useful statistics and insights to help users improve their charging habits and overall vehicle management.

Keywords: electric vehicle, charging, battery, iOS, Apple, SwiftUI

Tímto bych chtěl poděkovat vedoucímu mé diplomové práce, panu Ing. Radkovi Valovi, Ph.D., za jeho čas, ochotu, cenné rady, vedení a celkovou pomoc při psaní a implementaci této práce. Dále bych chtěl poděkovat panu doc. Ing. Martinu Pospíšilíkovi, Ph.D., za jeho cenné rady a odbornou pomoc při kontrole správnosti fyzikálních principů v algoritmu této práce.

OBSAH

ÚVOD.....	13
TEORETICKÁ ČÁST.....	14
1 EKOSYSTÉM APPLE.....	15
1.1 OPERAČNÍ SYSTÉMY APPLE	15
1.1.1 iOS, iPadOS, macOS – specifika jednotlivých systémů.....	15
1.1.2 Ekosystém Apple a funkce mezi zařízeními.....	17
1.2 VÝVOJ APLIKACÍ PRO APPLE ZAŘÍZENÍ	18
1.2.1 Xcode pro vývoj aplikací.....	18
1.2.2 Podpora technologií v Xcode.....	19
1.2.3 Simulátory a debugging na reálných zařízeních.....	22
2 TECHNOLOGIE A ARCHITEKTURA.....	24
2.1 PROGRAMOVACÍ JAZYK SWIFT.....	24
2.1.1 Hodnoty a datové typy	24
2.1.2 Řízení toku programu	25
2.1.3 Funkce.....	27
2.1.4 Třídy, struktury a objekty	27
2.1.5 Rozšíření	29
2.1.6 Ošetření chyb	29
2.2 FRAMEWORK SWIFTUI	30
2.2.1 Pohledy a modifikátory.....	30
2.2.2 Správa stavu.....	32
2.2.3 Navigace	34
2.2.4 Zpřístupnění	36
2.2.5 Async/Await	37
2.3 ARCHITEKTONICKÉ VZORY	37
2.3.1 MVVM (Model-View-ViewModel).....	37
2.3.2 MVC (Model-View-Controller)	39
2.3.3 Srovnání a rozdíly	41
3 BACKENDOVÉ TECHNOLOGIE A AUTENTIFIKACE.....	42
3.1 PŘEDSTAVENÍ FIREBASE	42
3.2 NABÍZENÉ SLUŽBY	43

3.2.1	Firestore Authentication	43
3.2.2	Firestore a Realtime Database.....	43
3.2.3	Firestore Cloud Functions.....	44
PRAKTICKÁ ČÁST.....		46
4	PRŮZKUM TRHU A KONKURENCE	47
4.1	MISE A CÍL APLIKACE	47
4.2	KONKURENČNÍ ŘEŠENÍ.....	47
4.2.1	EV CALC.....	47
4.2.2	EV Charge Calculator – Offline.....	48
4.2.3	EV Range Calculator	49
4.2.4	EV Calculator – Cost Tracker	50
4.2.5	EVrc	52
4.2.6	EV Charge Calculator and News	53
5	ARCHITEKTURA A ROZDĚLENÍ PROJEKTU	55
5.1	IMPLEMENTACE MVVM.....	55
5.2	VÝZVY PŘI IMPLEMENTACI.....	55
5.3	HLAVNÍ STRUKTURA PROJEKTU	56
5.4	SDÍLENÉ KOMPONENTY PROJEKTU	57
5.4.1	Servisy.....	57
5.4.2	UI komponenty	59
5.4.3	Utilities a Extensions	62
5.4.4	Assety a Konstanty.....	63
5.5	DIAGRAM SLOŽEK A SOUBORŮ	64
6	FUNKCE APLIKACE.....	66
6.1	NEFUNKCIONÁLNÍ POŽADAVKY	66
6.2	FUNKCIONÁLNÍ POŽADAVKY	66
6.3	POPIS JEDNOTLIVÝCH FUNKCÍ	69
6.3.1	Uživatelské účty	69
6.3.2	Nastavení.....	74
6.3.3	Domovská obrazovka.....	76
6.3.4	Správa vozidel a nabíječek.....	78
6.3.5	Výpočty.....	87
7	UŽIVATELSKÝ ZÁŽITEK A OPTIMALIZACE.....	93

7.1	UX/UI APLIKACE.....	93
7.2	PROTOTYP PRO TESTOVÁNÍ APLIKACE.....	94
7.3	KOMPONENTY A NÁVRH VE FIGMĚ	95
7.4	OPTIMALIZACE PRO RŮZNÉ VELIKOSTI OBRAZOVKY	98
8	VERZOVÁNÍ, REFACTORING A TESTOVÁNÍ	100
8.1	VERZOVACÍ SYSTÉM	100
8.2	VERZOVÁNÍ APLIKACE	101
8.3	REFACTORING A ČISTÝ KÓD	101
8.4	OVĚŘOVÁNÍ VÝPOČTŮ	102
	ZÁVĚR	104
	SEZNAM POUŽITÉ LITERATURY.....	105
	SEZNAM OBRÁZKŮ	108
	SEZNAM TABULEK.....	109
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	110
	SEZNAM PŘÍLOH	112

ÚVOD

Hlavním cílem této diplomové práce je usnadnit přechod uživatelů z vozidel se spalovacím motorem na elektromobil tím, že pomůže eliminovat obavy spojené s nutností plánování cest a výpočtů souvisejících s provozem a každodenním používáním elektromobilu. Aplikace poskytne uživatelům nástroje, které jim umožní lépe porozumět nabíjení, spotřebě a nákladům, díky čemuž minimalizuje nejistotu spojenou s touto změnou a usnadní jejich adaptaci na nový způsob mobility.

Spousta současných aplikací se snaží usnadnit život uživatelům přecházejícím na elektromobily, nicméně v řadě aspektů nejsou dostatečně efektivní – ať už jde o uživatelský zážitek a design, přesnost a množství vypočtených hodnot nebo celkovou použitelnost. Aplikace přímo od výrobců automobilů potom často nenabízejí nástroje pro plánování a kalkulace před samotným usednutím do vozidla nebo připojením k nabíječce, což omezuje možnosti uživatelů při přípravě na cestu. Tato práce se proto zaměří na všechny tyto oblasti s cílem vytvořit komplexní řešení, které poskytne užitečné informace, usnadní rozhodování a plánování nabíjení i cestování a celkově podpoří pohodlnější přechod na elektromobilitu.

Aplikace nabídne velkou škálu výpočtů, které umožní uživatelům lépe plánovat nabíjení a získat podrobné informace – mezi hlavní metriky budou patřit dopad na životní prostředí, tedy množství CO₂ vyprodukovaného během nabíjení, podrobnosti o nabíječce, vozidle a stavu nabití, množství přidané energie, celkový čas nabíjení včetně počátečního a koncového času, a také náklady spojené s nabíjením – ať už za jedno nabití, za kilometr nebo za plné dobití baterie. Uživatelé získají přehled i o dojezdu, a to jak o přidaném dojezdu na jedno nabití, tak o celkovém dojezdu při plně nabité baterii. Součástí výsledků bude také graf nabíjecího výkonu v závislosti na úrovni nabití baterie, který poskytne lepší představu o dynamice nabíjení. V neposlední řadě aplikace bude umožňovat výsledky snadno sdílet s dalšími uživateli díky vygenerovanému PDF reportu.

Aplikace nabídne také uživatelské účty a jejich správu, které umožní synchronizaci dat mezi různými zařízeními a tím usnadní její používání. Kromě toho poskytne možnosti přizpůsobení, včetně výběru barevného režimu (světlý nebo tmavý), preferovaného systému jednotek a jazyka aplikace. Aplikace bude dostupná pro zařízení od Apple, včetně iPhone, iPadu a Macu.

I TEORETICKÁ ČÁST

1 EKOSYSTÉM APPLE

Následující kapitoly budou věnovány ekosystému Apple a jeho specifickým, včetně rozdílů mezi operačními systémy iOS, iPadOS a macOS a funkcím umožňujícím spolupráci mezi zařízeními. Dále budou zaměřeny na vývoj aplikací pro Apple zařízení, kde bude popsáno použití Xcode, podporované technologie, možnosti simulace aplikací a ladění na reálných zařízeních.

1.1 Operační systémy Apple

Apple je tvůrcem několika operačních systémů, které jsou jedinečné pro jeho různá zařízení a které jsou vytvořeny tak, aby nabízely co nejlepší uživatelský zážitek a zároveň využily všechna specifika a určení daného zařízení. V této kapitole budou rozebrány hlavní operační systémy, které jsou pro tuto práci relevantní.

Operační systém iPhonu, iOS, se zabývá především snadnou a intuitivní navigací spolu s bezpečností a rychlým přístupem k aplikacím, které jsou využívány v každodenním životě uživatelů. Mobilní telefon iPhone je tak zařízení, které používá velké množství uživatelů různého věku po celém světě, což u konkurence s Androidem ne vždy platí, primárně kvůli fragmentaci Androidu a velkému množství variant tohoto operačního systému. Může to působit nejednotně. [1]

Na základě iOS představila společnost Apple později iPadOS, mobilní operační systém, který přidává funkce speciálně pro větší obrazovku iPadu, jako je multitasking a podpora Apple Pencil. Díky tomu je iPad ideální pro uživatele, kteří jsou umělci nebo studenti a používají ho například k digitálním poznámkám anebo také jen pro zábavu. [2]

Operační systém pro Mac počítače, macOS, se zaměřuje na profesionální a produktivní využití a podporuje také aplikace náročné na výkon, ať už se jedná o střih videa, úpravu fotografií anebo nástroje pro vývoj software. [3]

1.1.1 iOS, iPadOS, macOS – specifika jednotlivých systémů

Jednotlivé operační systémy jsou vyvinuty tak, aby zaručovaly optimální výsledky pro konkrétní zařízení z hlediska uživatelského použití a jednoduchosti, na které si Apple u svých produktů zakládá.

Operační systém iOS je navržen tak, aby uživatelům poskytoval intuitivní prostředí s důrazem na rychlost, bezpečnost a snadný přístup k aplikacím. Přesun důležitých

ovládacích prvků na obrazovku a kompaktní rozvržení celého systému jsou vhodné pro menší, ale i velké mobilní telefony – Apple totiž u svých zařízení nabízí několik velikostí a variant, jako je například Pro verze, která má vždy o nějaké funkce navíc anebo také Plus verze iPhone, která je zase větší než ta základní. Uživatelé mohou tedy ovládat iPhone jednou rukou, protože samotný systém disponuje plynulým a pohodlným rozhraním, kde je toto možné. Na druhou stranu se některým uživatelům, zvláště technicky založeným, může zdát, že operační systém neposkytuje takové možnosti nastavení, jako konkurenční Android, což se ale Apple v posledních aktualizacích iOS snaží změnit a přidává více funkcí, které dokáží rozhraní iPhone lépe konfigurovat podle představ uživatele. [1]

iPadOS, operační systém pro iPady, se vyvinul z iOS, který původně poháněl také iPady. Apple však v roce 2019 poslechl uživatele a přizpůsobil operační systém pro iPad větší obrazovce a potřebám uživatelů, kteří iPad používají k práci, tvoření i zábavě. Přidává funkce jako multitasking, podpora pro Apple Pencil nebo vylepšené ovládání pomocí gest, což iPad činí ideálním nástrojem pro kreativce a studenty. Funkce jako Split View nebo Slide Over umožňují pracovat s více aplikacemi zároveň, což je specifické pro iPadOS a není na iOS dostupné. Díky oddělení iOS a iPadOS může nyní Apple přidávat více funkcí specifických pro větší obrazovky bez toho, aniž by musel upravovat kód iOS, ale zároveň tak sdílet funkcionalitu, která je dostupná na iPhone. [2]

Poslední operační systém, který bude v této kapitole popsán je macOS, operační systém navržený pro stolní počítače a laptopy Mac, kde Apple klade důraz na produktivitu a podporu náročných aplikací. Tento operační systém nabízí klasické rozhraní se souborovým systémem, který je uživatelům známý z Windows nebo Unixových operačních systémů, na kterém je mimo jiné založen také macOS. Dále láká uživatele také širokými možnostmi pro přizpůsobení pracovní plochy a nástroje pro profesionály, jako je Final Cut Pro na střih videa nebo Xcode pro vývoj software, které jsou optimalizovány pro výkon počítačů Mac. macOS má navíc funkce jako plnohodnotný Finder, Dock a možnost práce s více okny a pracovními plochami, což jsou prvky určené pro efektivní pracovní tok na větší obrazovce. Díky velké spolehlivosti a jednoduchosti použití je tedy počítač s operačním systémem macOS vhodný nástroj pro profesionály anebo studenty, zvláště pokud uživatelé již vlastní nějaké další zařízení od Applu, jako například výše zmíněný iPhone anebo iPad. [3]

Pokud má počítač Mac také čip série „M“ vyvíjený Apple, je možné si nainstalovat a používat aplikace, které byly vyvinuty původně pro iOS nebo iPadOS, pokud vývojář tuto možnost nezakáže (některé aplikace určené pro telefony nemusí vypadat na větší obrazovce

dobře, což by mohlo vést ke špatnému hodnocení v App Store). Používání těchto aplikací na macOS je nyní možné díky stejné ARM architektuře čipů, která je použita také v iPhoneu a iPadu. Díky přechodu na tuto architekturu se stala z počítačů Mac velmi výkonná a úsporná zařízení se spoustou výhod, mezi které se právě řadí i používání aplikací původně vyvinuté pro mobilní zařízení. Stejně tak jako aplikace, které byly napsány na iOS jsou kompatibilní s iPadOS, pokud to však umožní vývojář ze stejných důvodů jako na macOS – některé aplikace mohou využívat specifické funkce daného zařízení. Je-li to možné a pro větší dosah, dostupnost a větší počet uživatelů je však dobré aplikace zpřístupnit na všechny platformy. [4]

1.1.2 Ekosystém Apple a funkce mezi zařízeními

Celý Apple ekosystém je navržen tak, aby zajišťoval bezproblémovou funkci a integraci mezi všemi zařízeními, která Apple vyrábí. Díky vzájemné kompatibilitě iPhoneu, iPadu, Macu, Apple Watch, AirPods a dalších zařízení mohou uživatelé jednoduše přecházet mezi různými zařízeními a pokračovat v práci, aniž by ztratili jejich progres anebo změny. Funkce jako Handoff umožňuje uživatelům začít psát na jednom zařízení a plynule pokračovat na jiném, například začít psát poznámku na iPhoneu a dokončit ji na iPadu nebo Macu. AirDrop zase usnadňuje sdílení souborů mezi zařízeními, aniž by bylo nutné si soubory předávat přes nějaké jiné úložiště, stejně tak i univerzální schránka, která umožňuje kopírování textu, obrázků nebo souborů na jednom zařízení a jejich okamžité vložení na jiném, což je opět velmi pohodlná funkce, která odstraní nutnost pro další službu na sdílení. V neposlední řadě také třeba sluchátka AirPods, která jsou spárována s iPhoneem uživatele, což znamená, že jsou automaticky spárována i s dalšími zařízeními (iPad, Mac), které jsou na uživatelově iCloud účtu a automaticky se připojí po nasazení do uší k zařízení, které právě uživatel používá.

Online cloudová služba od Apple, iCloud, pak propojuje všechna zařízení do jednoho ekosystému, ve kterém jsou automaticky synchronizovány soubory, fotky, poznámky, kontakty a zprávy (pokud si uživatel synchronizaci nevypne), což zaručuje, že všechny informace jsou vždy dostupné, bez ohledu na to, které zařízení právě uživatel používá. Dále se v rámci iCloudu synchronizují data o zařízeních nebo další metadata, ke kterým nemá přímo uživatel přístup. Tento vzájemně propojený ekosystém výrazně zvyšuje produktivitu a pohodlí uživatelů, kteří mohou jednoduše pracovat, komunikovat a sdílet obsah mezi zařízeními bez jakýchkoliv komplikací. Proto je potom velmi těžké z Apple

ekosystému přecházet ke konkurenčním řešením, které tuto synchronizaci a jednoduchost nenabízí do takové míry, jako právě Apple. [5]

1.2 Vývoj aplikací pro Apple zařízení

Vývoj aplikací pro zařízení od Applu je primárně charakteristický svou unikátní filozofií, která staví na integraci, uživatelské přívětivosti a optimalizaci výkonu. Apple ekosystém zahrnuje nejen hardware, kam spadá například iPhone, iPad, Mac nebo třeba Apple Watch, ale také odladěný software v podobě operačních systémů. Některé z nich byly popsány v kapitole výše. Tyto operační systémy jsou navrženy tak, aby spolu bezproblémově spolupracovaly a poskytovaly vývojářům konzistentní platformu, na které mohou stavět aplikace s vysokou mírou funkčnosti a spolehlivosti. Tato integrace umožňuje vývojářům snadno vytvářet aplikace, které fungují napříč Apple zařízeními a celým ekosystémem, což nejen šetří čas při vývoji, ale také přináší větší uživatelský komfort. Například aplikace původně navržená pro iPhone může být relativně snadno přizpůsobena pro iPad nebo Mac.

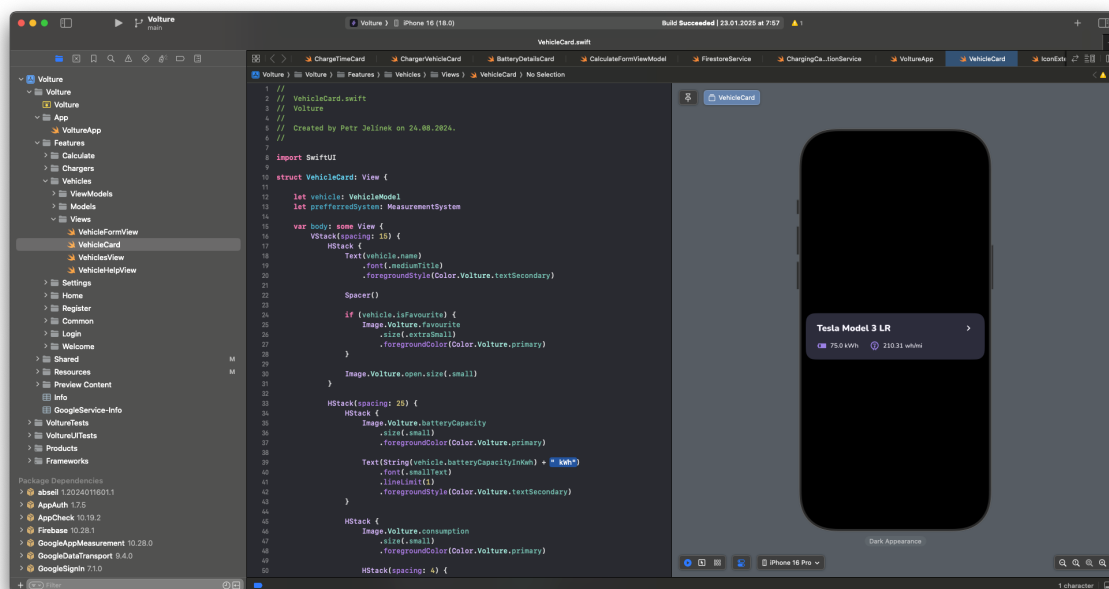
1.2.1 Xcode pro vývoj aplikací

Vývoj aplikací pro Apple zařízení není jen o specifikách operačních systémů a celé Apple platformy, ale také o zvládnutí a pochopení vývojového prostředí, které tento proces vývoje zjednodušuje a zefektivňuje. Xcode je vývojové prostředí (Obrázek 1 - Xcode), které je vyvinuté přímo firmou Apple a nabízí vše potřebné k návrhu, vývoji, testování a nasazení aplikací, což z něj dělá klíčový a nezbytný nástroj pro každého vývojáře v tomto ekosystému. Nejedná se pouze o textový editor kódu, ale spíše o komplexní sadu nástrojů navrženou s ohledem na optimalizaci celého vývojového procesu aplikací pro Apple platformy. Xcode obsahuje například simulátory pro testování aplikací na různých zařízeních, debugovací nástroje pro hledání a opravu chyb v kódu nebo třeba rozhraní pro práci s uživatelskými rozhraními anebo správce SDK pro jednotlivé operační systémy a jejich verze. [6]

Jednou z velkých výhod Xcode je také jeho možnost integrace s dalšími službami a technologiemi od Applu – například rozhraní SwiftUI umožňuje vytvářet uživatelská rozhraní v moderním a intuitivním jazyce Swift. Dalším příkladem je třeba integrace s nástrojem CloudKit, který usnadňuje práci na backendu, díky službám, které poskytuje anebo třeba nástroj TestFlight pro distribuci verzí aplikace k testerům pro její kontrolu. Nástrojů, služeb a dalších API je zde od Applu připraveno opravdu mnoho a díky tomu

je zaručeno to, že mezi všemi aplikacemi na Apple zařízeních je jakási konzistentnost a povědomost, kterou uživatelé už znají z jiných aplikací a je jim do jisté míry blízká. [6]

Dalším důležitým faktorem Xcode je důraz na kvalitu a výkon aplikací. Díky vestavěným nástrojům pro analýzu a optimalizaci výkonu mohou vývojáři zajistit, že jejich aplikace budou nejen funkční, ale také rychlé či úsporné, což je obzvlášť důležité na mobilních zařízeních, kde je potřeba brát ohled na to, že uživatel může mít starší zařízení se slabší baterií anebo výkonem. Xcode díky tomu podporuje filozofii Applu zaměřenou na uživatelskou spokojenost a uživatelský zážitek. Kvalita aplikace je mimo jiné také kontrolována v momentě, kdy vývojář chce svou aplikaci publikovat pro uživatele do App Store. Před samotným publikováním (mimo jiné i tento proces lze udělat přes Xcode) a distribucí prochází aplikace kontrolou, zda splňuje všechny pokyny, které Apple popsal a připravil pro vývojáře na svých stránkách. [7] Aplikace, které budou distribuovány přes App Store, si musí držet definovaný standard a kvalitu. Nároky jsou kladeny primárně na bezpečnost, optimalizaci výkonu a uživatelského rozhraní. [6]



Obrázek 1 - Xcode

(zdroj: vlastní)

1.2.2 Podpora technologií v Xcode

Vývoj aplikací pro zařízení a operační systémy od Apple je možný díky podpoře tří hlavních technologií – SwiftUI, UIKit a Objective-C. Každá z těchto technologií má své specifické vlastnosti a určení, které umožňují vývojářům přizpůsobit vývoj aplikace podle konkrétních

požadavků a specifik projektu, od návrhu uživatelského rozhraní až po udržování starších aplikací. Programovací jazyk Swift a framework SwiftUI budou podrobněji rozebrány v následujících kapitolách, protože jsou pro tuto práci nejvíce relevantní.

SwiftUI je deklarativní framework, který byl představený Applem v roce 2019. Primárně se snaží zjednodušit vývoj uživatelských rozhraní díky své čitelné a efektivní syntaxi. Umožňuje přímé propojení vizuálních prvků na obrazovce uživatele s logikou aplikace prostřednictvím anotací jako `@State`, `@Binding` nebo `@Environment`. Mezi hlavní funkce SwiftUI patří podpora tzv. live previews, které umožňují sledovat změny, které vývojář provede v rámci kódu v uživatelském rozhraní v reálném čase přímo v Xcode na obrazovce simulátoru zařízení dané platformy. Framework je mimo jiné také optimalizovaný pro všechny cílové platformy, což velmi usnadňuje tvorbu univerzálních aplikací. Na ukázce kódu níže je vidět jednoduchá aplikace `Hello, World!`, která pouze zobrazí tento text na displeji zařízení. Ukázková aplikace demonstuje minimalistický přístup SwiftUI, kde kód definuje obsah obrazovky pomocí jednoduchých metod a struktur, což se více podobá přirozenému jazyku. [8]

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .font(.largeTitle)
            .padding()
    }
}

@main
struct HelloWorldApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

UIKit je tradiční a starší framework používaný pro vývoj uživatelských rozhraní na Apple platformy, známý svým imperativním přístupem. Framework byl představený v roce 2008 a nabízí detailní a mnohem širší kontrolu nad chováním a interakcemi v rámci uživatelského rozhraní. Vývojáři zde mohou využívat nástroje jako Storyboards a Interface Builder pro vizuální návrh aplikací. UIKit je však zpětně kompatibilní, což z něho dělá vhodnou technologii pro projekty, které musí podporovat starší verze iOS. Dnes je však často nahrazován novějším SwiftUI, zvláště na nových projektech, které jsou psané úplně od nuly. V každém případě však zůstává nezbytným nástrojem pro správu a údržbu existujících aplikací. Níže následuje ukázka stejné aplikace, v tomto případě však psané pomocí UIKit frameworku. V porovnání se SwiftUI je kód v UIKit mnohem obsáhlejší. [9]

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let label = UILabel()
        label.text = "Hello, World!"
        label.textAlignment = .center
        label.frame = view.bounds
        view.addSubview(label)
    }
}

@main
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
        window = UIWindow(frame: UIScreen.main.bounds)
        window?.rootViewController = ViewController()
        window?.makeKeyAndVisible()
        return true
    }
}
```

Poslední technologie zmíněná v rámci této kapitoly je Objective-C, což je objektově orientovaný programovací jazyk, který vychází z jazyka C, obohacený o koncepty Smalltalku. I když byl postupně nahrazen novějším a modernějším jazykem Swift, stále je důležitý díky své dynamické povaze a velkému množství existujících knihoven. Vývojáři mohou například při běhu aplikace provádět dynamické operace, jako je introspekce objektů nebo manipulace s vlastnostmi tříd. Objective-C je stále používán při práci na starších projektech nebo při integraci s legacy kódem, takže se s ním stále v rámci vývoje můžeme setkat i dnes. Na ukázce kódu níže je jednoduchá funkce `main`, která je napsaná pomocí programovacího jazyka Objective-C. [10]

```
#import <Foundation/Foundation.h>

int main() {
    ... // Kód aplikace
    return 0;
}
```

Tyto výše zmíněné technologie tvoří základ vývoje software v rámci Apple ekosystému. SwiftUI představuje moderní a intuitivní přístup, UIKit naopak poskytuje více detailní kontrolu a Objective-C zůstává důležitým nástrojem pro práci s historickým kódem. Xcode zajišťuje hladkou integraci všech tří technologií, což vývojářům umožňuje vytvářet aplikace odpovídající jejich potřebám a požadavkům.

1.2.3 Simulátory a debugging na reálných zařízeních

Simulátory v Xcode nabízejí rychlost a pohodlí – umožňují snadné přepínání mezi různými verzemi zařízení a operačními systémy bez potřeby fyzického zařízení, které by musel vývojář vlastnit. Jsou ideální pro rychlé zkontrolování změn v kódu, testování uživatelského rozhraní nebo základní funkčnosti aplikace. Problém je však ten, že simulátory nedokážou přesně replikovat chování skutečných zařízení, primárně v hardwaru a specifickém chování při jeho použití. Jeden z příkladů je třeba výkon aplikace, který je na simulátoru ovlivněn výpočetní kapacitou Macu, na kterém je spuštěn, nikoli výkonem konkrétního iPhoneu nebo iPadu. Aplikace na simulátoru také nedokážou realisticky používat hardwarové funkce jako kamery, GPS nebo akcelerometry. Další problém může být také to, že jsou často jejich síťové podmínky lepší než na fyzickém zařízení, což neodráží reálné situace. Například uživatel může být v odlehlé krajině, kde signál není vůbec anebo je velmi slabý.

Na druhé straně debugging aplikací na reálných zařízeních poskytuje přesnější výsledky, protože odráží skutečné chování aplikace v prostředí koncového uživatele na jeho zařízení a za specifických podmínek. Tento způsob je při vývoji aplikací nezbytný pro ladění hardwarových funkcí, testování výkonu v omezených podmínkách, jako je třeba nízký výkon anebo malá paměť, nebo simulaci reálných síťových situací (např. pomalé připojení). Nevýhodou je však delší proces kompilace a nahrávání aplikace na samotné zařízení, což může vývoj značně zpomalit. Dále také nutnost spravovat certifikáty, ale hlavně fyzické zařízení vlastnit, což není vždy možné, zvlášť pokud se jedná o menší firmu nebo vývojáře, kteří nemají velké finanční zdroje na vývoj. Kombinace obou přístupů je nicméně klíčem k úspěchu. Simulátory (Obrázek 2 - Simulátor iPhoneu 16) se hodí na rychlé iterace a základní testování nových funkcí a změn, zatímco reálná zařízení jsou nepostradatelná pro finální ladění, optimalizaci výkonu a zajištění nejlepšího uživatelského zážitku.



Obrázek 2 - Simulátor iPhoneu 16

(zdroj: vlastní)

2 TECHNOLOGIE A ARCHITEKTURA

V této části budou představeny programovací jazyky, frameworky a architektury používané při vývoji aplikací pro Apple zařízení. Nejprve bude popsán programovací jazyk Swift a framework SwiftUI, následně budou rozebrány architektonické vzory v iOS vývoji, jejich srovnání a klíčové rozdíly.

2.1 Programovací jazyk Swift

Swift je moderní programovací jazyk vyvinutý společností Apple, který je navržen tak, aby byl přívětivý pro začátečníky, ale zároveň nabízel i pokročilé funkce pro zkušené vývojáře. Swift je ideální jazyk pro vývoj softwaru pro telefony, tablety, počítače, servery nebo cokoli jiného, kde běží kód. Díky jeho jednoduchosti, bezpečnosti a také rychlosti je vynikající volbou pro všechny aplikace od „Hello, world!“ až po celý operační systém. Kód, který je napsaný pomocí Swiftu, je zkompilován a optimalizován tak, aby co nejlépe využíval moderní hardware. Swift kombinuje moderní, jednoduchou a čitelnou syntaxi, která je povědomá a známá vývojářům, kteří přicházejí z jiných populárních jazyků (Java, Kotlin, JavaScript), spolu s výkonnými funkcemi, jako je například odvozování typů. To umožňuje vyjádřit složité myšlenky anebo převést požadavky projektu jasným a stručným způsobem, což potom vede ke snazšímu čtení, psaní a údržbě kódu. [11]

2.1.1 Hodnoty a datové typy

Pro vytvoření konstanty se v rámci Swiftu používá klíčové slovo `let` a pro vytvoření proměnné `var`. Hodnota konstanty nemusí být při kompilaci vždy známa, je nicméně nutné jí přiřadit hodnotu, a to jen jednou, což znamená, že dále v kódu se její hodnota nesmí změnit. Naopak u proměnné lze hodnotu v průběhu programu přepsat.

```
var foo = 1      // vytvoření proměnné
foo = 2          // nastavení nové hodnoty do proměnné
let bar = 10     // nemůže být změněna
```

Díky funkci automatického odvozování typu není u proměnných nutné psát jejich datový typ, což může někdy pomoci také v čitelnosti. Nicméně je potřeba myslet na to, že u komplikovanějších struktur anebo datových typů je někdy lepší datový typ uvést, aby bylo z kontextu pro vývojáře jasné, o jaký typ proměnné se jedná. Zápis by vypadal následovně – `let value: Int = 10`. V rámci Swiftu jsou obsaženy všechny standartní

datové typy jako je například `Int`, `Double`, `Float`, `String` atd., které jsou také v ostatních programovacích jazycích.

Swift obsahuje i pole pro sekvenční ukládání dat anebo třeba slovníky, což je standardní datová „klíč-hodnota“ struktura pro konstantní přístup. K jednotlivým elementům se potom přistupuje buď pomocí indexu v poli, nebo pomocí klíče v rámci slovníku. Na ukázce níže lze vidět vytvoření pole a následný přístup k prvku na indexu 1, dále vytvoření slovníku, kde se následně přistupuje k prvku pod klíčem `Jayne`. [12]

```
var fruits = ["strawberries", "limes", "tangerines"]
fruits[1] = "grapes"
var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

2.1.2 Řízení toku programu

Stejně jako v ostatních programovacích jazycích, tak i ve Swiftu jsou zde připraveny konstrukty k řízení toku programu, ať už se jedná o podmínky a větvení anebo cykly. Na ukázce níže lze vidět použití podmínky pomocí klíčového slova `if`.

```
if value > 50 {
    ...
} else {
    ...
}
```

U Swiftu lze použít tento příkaz v mnoha formách, nicméně se stále jedná o rozhodování – pouze se zjednoduší syntaxe, aby se například při přiřazení hodnoty nemusel psát kód navíc, což by mohlo zhoršit čitelnost kódu. U některých programovacích jazyků lze tento zápis však realizovat pomocí „ternary“ operátoru.

```
let result = if value > 10 { ... } else { ... }
```

Vzhledem k tomu, že má Swift nullable typy, tak je často nutné kontrolovat, jestli se náhodou v dané proměnné `nil` (neexistující hodnota) nenachází. K tomu potom slouží `?` operátor, který říká, zda v proměnné může být prázdná hodnota, anebo ne – `var value: String? = "Foo"`. Pokud by bylo nutné dále v kódu tuto proměnnou použít a zavolat na ni například nějakou metodu, před volání samotné metody se dává taktéž symbol `?`, který, pokud je hodnota `nil`, metodu nezavolá a tím se vyhneme vyhození výjimky. Zápis potom vypadá takto – `value?.uppercased()`. Pomocí `??` pak lze přiřadit do proměnné výchozí hodnotu, která se použije, pokud je původní hodnota `nil` – `value?.uppercased() ?? "EMPTY"`.

Dalším konstruktem pro řízení toku programu v rámci Swiftu je `switch-case`, který přesměruje tok programu na větev, která jako první splní podmínku.

```
let color = "Red"
switch color {
case "Red":
    print("...")
case "Green", "Blue":
    print("...")
default:
    print("...")
}
```

Pro opakované operace nebo například pro iterování přes nějakou kolekci se používají cykly `for` a `while`, které jsou taktéž standartní tak, jak je známe z ostatních programovacích jazyků. I zde je však mírně upravená a vylepšená syntaxe, nicméně samotná funkce zůstává stejná, jako všude jinde. [12]

```
let numbers = [75, 43, 103, 87, 12]
for number in numbers { ... }

let n = 1
while n < 100 { ... }
```

2.1.3 Funkce

U funkcí, stejně jako u proměnných, funguje ve Swiftu také automatické odvození návratového typu, což může udělat čtení kódu o něco lepší. Platí zde však stejné pravidlo, že u komplexnějších funkcí je lepší návratový typ zapsat, aby se potom lépe určilo, co metoda dělá. Na ukázce níže je definice a volání funkce, která má dva parametry `person` a `day`. Tato jednoduchá funkce potom vrací `string`, který je složen z dosazených hodnot `John` a `Wednesday`. Na ukázce si lze také všimnout operátoru `_` před parametrem `person`, který umožní předat hodnotu bez toho, aniž bychom psali název parametru. Pokud bychom tento operátor nepoužili, vypadalo by volání metody následovně `greet(person: "John", on: "Wednesday")`. Tato funkcionalita se používá pro čitelnější zápis a je zvlášť velmi vhodná u metod, které mají pouze jeden parametr, a předávání pomocí tzv. `named arguments` nemá moc velký význam. Další funkcí je potom použití externího jména parametru, v tomto případě `on`, který se hodí taktéž ke zlepšení čitelnosti, díky čemuž je potom kód více sebe-dokumentovaný. Z toho důvodu není potřeba používat při volání funkce název parametru, ale tento alias.

```
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}

greet("John", on: "Wednesday")
```

Ve Swiftu je možné psát také složené funkce, kde definice jedné funkce je uvnitř další. Scope takové vnořené funkce je ale pouze v rámci jeho rodiče. Dále je možné funkce použít například k vrácení z jiné funkce, použití funkce jako argument atd. Stejně tak jako v dalších jazycích, i ve Swiftu jsou podporované tzv. `closures`, které se často používají pro jednodušší zápis například při iterování anebo transformaci prvků v poli `numbers.map({ number in 3 * number })`. [12]

2.1.4 Třídy, struktury a objekty

Třída ve Swiftu se vytváří pomocí klíčového slovíčka `class`, za kterým potom následuje název třídy. Třída, potažmo její instance, je vždy referenční datový typ. Pokud by bylo nutné vytvořit objekt, jehož typ je hodnotový, tedy je uložen na zásobníku na rozdíl od haldy v případě třídy, používá se k tomu struktura s klíčovým slovem `struct`. Mezi třídou

a strukturou jsou potom ještě další specifika jako například to, že struktura nepodporuje dědičnost, nemá deinicializátory, je imutabilní ve výchozím stavu a je také o něco méně náročná z hlediska výkonu či paměti.

Vlastnosti ve třídě se zapisují stejným způsobem jako například deklarace konstant anebo proměnných popsanych v kapitole výše, s tím rozdílem, že je nyní deklarace uvnitř třídy. To samé platí i u deklarace metod a funkcí. Samotné vlastnosti ve třídě mohou mít ještě vlastní gettery a settery pro další logiku. Na ukázce níže lze vidět vytvoření třídy a také následnou instanci dané třídy. Po vytvoření instance třídy je možné přistoupit k jejím veřejným vlastnostem anebo metodám. Funkce `init` v tomto případě slouží jako konstruktor, což je metoda, která se volá při vytváření instance třídy a provádí se zde inicializace proměnných, které nemají výchozí hodnotu. Ve Swiftu existuje také metoda `deinit`, což je v podstatě destruktorka a obsahuje logiku pro čištění při dealokaci.

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}

var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

V rámci objektově orientovaného programování je zde samozřejmě i funkcionalita dědičnosti, tzn. funkce jako dědění, přepisování metod, rozhraní (protokoly), atd. To umožní vývojářům psát kód podle různých architektonických pravidel anebo návrhových vzorů, pro lepší design a strukturu kódu, což vede k větší čitelnosti nebo znovu-použitelnosti. V rámci Swiftu se používá označení protokolů, což jsou v podstatě rozhraní (interface), které známe z jiných programovacích jazyků. [12]

2.1.5 Rozšíření

Podobně jako například v Kotlinu, tak i ve Swiftu existují rozšíření, což umožní obohatit funkcionalitu již existující třídy nebo struktury, která je například z nějaké knihovny, nad kterou nemáme plnou kontrolu. Díky rozšířením můžeme vytvořit funkcionalitu, která bude potom dostupná v rámci kódu tam, kde budeme používat daný typ, na kterém je rozšíření definováno. Na ukázce níže lze vidět rozšíření typu `String`, které po zavolání udělá první písmeno velké. V rámci rozšíření lze psát jak funkce, tak i vlastnosti. [12]

```
extension String {  
    var capitalizedFirst: String {  
        return prefix(1).uppercased() + dropFirst()  
    }  
}  
  
print(example.capitalizedFirst)
```

2.1.6 Ošetření chyb

Pro vytvoření vlastní chyby (z jiných jazyků známé jako výjimky) je nutné, aby třída, struktura anebo enum implementovaly protokol `Error`, díky čemuž se potom vlastní chyba dá „vyhodit“ z kódu, což se dělá pomocí klíčového slova `throw CustomErrorName`. Pro ošetření situace, kdy kód může chybu „vyhodit“ se používá například `do-catch` blok, jako to je na ukázce níže. V případě, že nastane chyba, se provede daná logika, jako například logování. I zde je možné přidat více `catch` bloků pro zachycení specifické chyby a provedení adekvátní akce, pokud tento případ nastane. [12]

```
do {  
    let response = try send(job: 1040, toPrinter: "Bi Sheng")  
    print(printerResponse)  
} catch {  
    print(error)  
}
```

2.2 Framework SwiftUI

V této části se bude práce věnovat frameworku SwiftUI – nejprve bude popsána práce s pohledy a modifikátory, poté bude zaměřena na správu stavu a navigaci mezi obrazovkami. Dále bude popsáno zpřístupnění aplikací a nakonec asynchronní zpracování.

2.2.1 Pohledy a modifikátory

Ve SwiftUI, každý pohled nebo prvek, musí implementovat rozhraní, tedy protokol, `View`, který se následně používá pro definici jednotlivých prvků na obrazovce. Každý prvek v aplikaci, který implementuje tento protokol, musí mít také implementovanou vlastnost `body`, která se později v kódu volá při vykreslování prvků na obrazovku. Na ukázce níže je vidět jednoduchý view, který pouze zobrazuje text na obrazovce. Tento text má ještě nastavený styl, v tomto případě velikost fontu a padding. Na této ukázce lze hezky vidět moderní a čitelnou, deklarativní syntaxi, která je použita napříč celým SwiftUI frameworkem.

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .font(.title)
            .padding()
    }
}
```

SwiftUI nabízí také několik variant, jak jednotlivé elementy na obrazovce skládat – je zde možné skládat je horizontálně či vertikálně vedle sebe resp. pod sebe, anebo třeba na sebe tak, aby se překryly. K tomu slouží kontejnery jako `VStack` (vertikální), `HStack` (horizontální) a `ZStack` (překrytí na „z“ ose). Na ukázce níže lze vidět kód, který využívá všechny tyto kontejnery a jednotlivé prvky adekvátně poskládá.

```
struct StackedViews: View {
    var body: some View {
        ZStack {
            Color.blue.edgesIgnoringSafeArea(.all)
        }
    }
}
```

```
VStack {  
    Text("...")  
    HStack {  
        Text("Left")  
        Text("Right")  
    }  
    .padding()  
    .background(Color.white)  
    .cornerRadius(10)  
}  
}  
}
```

Jak již bylo zmíněno výše, jednotlivé prvky mají také modifikátory, které upravují to, jak bude element vypadat anebo jak se bude chovat. Modifikátor však vrací nový pohled použitím změn na stávající pohled, přičemž zachovává jeho neměnnost (immutability). V následující ukázce lze vidět text, který bude mít změněný font, barvu a také zaoblení rohů spolu s barvou pozadí.

```
struct CustomText: View {  
    var body: some View {  
        Text("Hello, World!")  
        .font(.headline)  
        .foregroundColor(.green)  
        .padding()  
        .background(Color.black)  
        .cornerRadius(8)  
    }  
}
```

Jednotlivé pohledy mohou být dynamicky měněny například na základě odpovědi ze serveru anebo po interakci uživatele, což umožní větší dynamiku a responsivnost celé aplikace. V příkladu níže se po interakci s textem změní barva pozadí na zelenou nebo šedou. Správa stavů bude více popsána v další kapitole. [13]

```
struct ConditionalView: View {  
    @State private var isHighlighted = false
```

```
var body: some View {  
    Text("Tap Me!")  
        .padding()  
        .background(isHighlighted ? Color.green : Color.gray)  
        .onTapGesture {  
            isHighlighted.toggle()  
        }  
}
```

2.2.2 Správa stavu

Správa stavu je velmi důležitým prvkem deklarativního frameworku SwiftUI, který umožňuje automatickou reakci prvků a pohledů na změnu dat. SwiftUI používá k efektivní správě stavu tzv. „property wrappers“, což umožní vývojářům přenechat kontrolu nad stavem přímo frameworku. První, nejjednodušší anotací je `@State`, která slouží primárně ke správě jednoduchého lokálního stavu v rámci jednoho pohledu. Když se hodnota proměnné s anotací `@State` změní, SwiftUI automaticky aktualizuje prvek anebo pohled, který je na této proměnné závislý. V kapitole výše lze vidět použití této anotace v akci, kdy se po kliknutí na text změní hodnota vlastnosti `isHighlighted` a tím se také změní barva pozadí textu.

Další často používanou anotací je `@Binding`, která umožňuje sdílení stavu mezi rodičem a potomkem v rámci pohledu. Díky tomu je umožněna obousměrná komunikace mezi rodičem a potomkem, což lze využít k zachování jednoho zdroje pravdy, pokud máme například nějakou hodnotu, která by měla být jak u rodiče tak u potomka vždy stejná – díky tomu ji tedy není nutné definovat dvakrát, ale pouze si ji předat díky této funkcionalitě. Na ukázce níže jsou dva pohledy, kdy rodičovský pohled „vlastní“ vlastnost `isOn`, nicméně potomek s ní potom pracuje dále. Tento přístup je považován za velmi vhodný i tehdy, když je cílem udržet kód v přehledném stavu s menšími pohledy, nebo pokud je potřeba některé prvky v kódu opakovaně využívat.

```
struct ParentView: View {  
    @State private var isOn = false  
  
    var body: some View {
```

```
        ToggleView(isOn: $isOn)
    }
}

struct ToggleView: View {
    @Binding var isOn: Bool

    var body: some View {
        Toggle("Switch", isOn: $isOn)
    }
}
```

Pokud zobrazení některých elementů anebo pohledů závisí na externím datovém modelu a jeho vlastnostech, musí tento datový model implementovat protokol `ObservableObject`. Následně se potom používá spolu s anotací `@ObservedObject`. Změny jednotlivých vlastností tohoto modelu potom vyvolávají překreslení uživatelského rozhraní. U jednotlivých vlastností, na kterých bude zobrazení uživatelského rozhraní závislé, je nutné také ještě přidat anotaci `@Published`. Na ukázce níže je jednoduché počítadlo na inkrementování stavu, které však svůj stav drží uvnitř modelu `CounterModel`.

```
class CounterModel: ObservableObject {
    @Published var count = 0
}

struct CounterObservedView: View {
    @ObservedObject var model = CounterModel()

    var body: some View {
        VStack {
            Text("Count: \(model.count)")
            Button("Increment") {
                model.count += 1
            }
        }
    }
}
```

Pro jednoduché sdílení dat mezi jednotlivými pohledy a bez nutnosti předávání těchto dat do konstruktoru se používá `@EnvironmentObject`. Kód níže ukazuje, jak lze sdílená data předat do prostředí pohledu pomocí modifikátoru `.environmentObject`. K tomuto objektu mají potom přístup všichni potomci daného pohledu. Velkou výhodou tohoto přístupu je právě automatická správa stavů a odpadnutí nutnosti předávat data přes konstruktor, což umožní ještě větší dynamičnost v rámci aplikace. [13]

```
class SharedData: ObservableObject {
    @Published var message = "Hello, World!"
}

struct ParentEnvironmentView: View {
    @StateObject var data = SharedData()

    var body: some View {
        ChildEnvironmentView().environmentObject(data)
    }
}

struct ChildEnvironmentView: View {
    @EnvironmentObject var data: SharedData

    var body: some View {
        Text(data.message)
    }
}
```

2.2.3 Navigace

SwiftUI nabízí jednoduché a pohodlné rozhraní pro navigaci mezi jednotlivými obrazovkami aplikace, což zahrnuje také vytváření hierarchií pohledů, mezi kterými se může uživatel pohybovat. Veškeré prvky jako `NavigationView` anebo `NavigationLink` stejně jako ostatní struktury používají moderní, deklarativní zápis. První struktura `NavigationView` slouží jako jakýsi kontejner pro správu hierarchií. Poskytuje základ pro zobrazování potomků (pohledů) a také správu navigační lišty. V kódu níže lze vidět, jak se v rámci této struktury zabalí další pohled, v tomto případě jen `Text`, nicméně to může

být další, komplexnější struktura. Modifikátor `.navigationTitle` potom nastaví název do navigační lišty.

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            Text("Home Screen")
                .navigationTitle("Home")
        }
    }
}
```

Pro navigaci mezi jednotlivými pohledy se potom používá `NavigationLink`, který po kliknutí na nějaký element může uživatele přesměrovat dále. Často se tato komponenta používá v rámci nějakého listu, tlačítka anebo dalších interaktivních komponent. Na ukázce níže bude po kliknutí na text uživatel přesměrován na `DetailView` pohled. Veškeré animace, přechody anebo třeba gesta jako „přejetí zpět“ jsou řešeny automaticky v rámci SwiftUI, což zajistí konzistentnost a povědomost pro uživatele. Je zde samozřejmě také možnost použít vlastní text pro tlačítko zpět nebo také třeba vlastní animace.

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                NavigationLink("...", destination: DetailView())
            }
                .navigationTitle("Home")
        }
    }
}

struct DetailView: View {
    var body: some View {
        Text("...")
            .navigationTitle("Details")
    }
}
```

V rámci nového iOS (od verze 16+) je doporučeno používat místo `NavigationView` novější `NavigationStack`, který nabízí ještě o něco větší kontrolu nad navigací v aplikaci, hlavně díky jeho větší flexibilitě v rámci programovatelné navigace. K programové navigaci se zde používá pole cest. V příkladu níže se potom pomocí `.navigationDestination` zobrazí příslušný pohled. [13]

```
struct ContentView: View {
    @State private var path = [String]()

    var body: some View {
        NavigationStack(path: $path) {
            VStack {
                Button("Go to Page 1") { path.append("Page1") }
                Button("Go to Page 2") { path.append("Page2") }
            }
            .navigationDestination(for: String.self) { page in
                Text("This is \(page)")
            }
        }
    }
}
```

2.2.4 Zpřístupnění

SwiftUI zvyšuje inkluzivitu aplikací tím, že poskytuje pro vývojáře nástroje, modifikátory a rozhraní jako jsou například `accessibilityLabel` nebo `accessibilityHint`, které dokážou popsat jednotlivé prvky pro uživatele, kteří mají například problém se zrakem. Pokud jsou aplikace a její prvky a pohledy označeny těmito modifikátory, funkce jako `VoiceOver` dokáže potom aplikaci popsat a přečíst to, co například konkrétní tlačítko dělá. Díky tomu je potom aplikace přístupnější pro všechny uživatele. Na ukázce níže lze vidět, jak to potom vypadá v kódu. [13]

```
struct AccessibleView: View {
    var body: some View {
        Button(action: { ... }) {
            Image(systemName: "star")
        }
    }
}
```

```
    }  
    .accessibilityLabel("Favorite")  
    .accessibilityHint("Marks this item as a favorite")  
  }  
}
```

2.2.5 Async/Await

Ve SwiftUI lze také využít funkce pro podporu souběžnosti, tedy neblokovaní jednou operací další. Mohou se používat například pro zpracování dlouhotrvajících úloh, jako je třeba načítání dat ze serveru anebo generování dokumentu. V rámci SwiftUI, podobně jako u jiných jazyků, lze použít syntaxi `async / await`, díky které se potom kód stává čitelnější a přehlednější. Na ukázce lze vidět použití těchto dvou klíčových slov, kdy je potřeba nejdříve metodu označit jako `async` a následně před volání samotné dlouhotrvající funkce použít `await`. [13]

```
func fetchData() async -> String {  
    try? await Task.sleep(nanoseconds: 1_000_000_000)  
    return "Data Loaded"  
}
```

2.3 Architektonické vzory

V následujících kapitolách bude práce zaměřena na architektonické vzory používané v iOS vývoji. Budou popsány vzory MVVM (Model-View-ViewModel) a MVC (Model-View-Controller) jako nejčastěji využívané přístupy. Na závěr bude provedeno jejich vzájemné srovnání, včetně klíčových rozdílů a vhodnosti jejich použití v různých scénářích.

2.3.1 MVVM (Model-View-ViewModel)

Tento architektonický návrhový vzor se stal velmi rozšířeným přístupem k vytváření čistých, udržovatelných a testovatelných aplikací pro iOS, resp. pro všechny operační systémy od Apple. Tento architektonický vzor lze však použít kdekoliv, není nutnou podmínkou ho použít pouze pro vývoj na Apple platformách. Díky jasnému oddělení funkcí jednotlivých komponent je potom mnohem snadnější škálovat a udržovat aplikaci, pokud je potřeba ji v budoucnu upravovat a přidávat další funkce. Tento návrhový vzor je pro SwiftUI obzvláště efektivní díky jeho deklarativní povaze a vestavěným funkcím, jako jsou například

@State a ObservableObject, popsané v kapitole výše. Samotný návrhový vzor se skládá ze tří hlavních komponent.

Model reprezentuje data a business logiku samotné aplikace – může obsahovat strukturu reprezentující odpověď ze serveru, databázové objekty, datové kontejnery anebo jednoduché struktury v rámci Swiftu. Model by neměl vědět nic o uživatelském rozhraní.

View potom reprezentuje to, co vidí uživatel. V rámci SwiftUI jsou zde zahrnuty komponenty jako Text, Button, List a další elementy, ze kterých se potom skládá komplexnější pohled. Pohledy (View) jsou deklarativní a zobrazují pouze data poskytnutá ViewModelem.

ViewModel je potom jakýmsi mostem mezi Modelem a View, který připravuje data z Modelu do formátu vhodného pro zobrazení a také řídí logiku potřebnou pro interakci s uživatelem.

Na ukázce kódu níže lze vidět implementaci jednoduchého počítadla, které je vytvořeno za použití tohoto návrhového vzoru. Model, Counter, drží data pro výpočet, CounterViewModel řeší jednoduchou logiku počítadla a CounterView zobrazuje výsledek a řeší interakci s uživatelem. Tento kód je potom velmi jednoduchý pro rozšíření a přidání dalších funkcí díky tomu, že logika a určení komponent je rozděleno do adekvátních struktur a není vše smíchané v rámci jednoho souboru. [14]

```
struct Counter {
    var value: Int
}

class CounterViewModel: ObservableObject {
    @Published var counter = Counter(value: 0)

    func increment() {
        counter.value += 1
    }
    ...
}

struct CounterView: View {
    @StateObject private var viewModel = CounterViewModel()
```

```
var body: some View {
    VStack(spacing: 20) {
        Text("Count: \(viewModel.counter.value)")
            .font(.largeTitle)
            .fontWeight(.bold)

        Button("Increment") {
            viewModel.increment()
        }
    }
}
```

2.3.2 MVC (Model-View-Controller)

Architektonický návrhový vzor Model-View-Controller (MVC) patří mezi základní přístupy k vývoji aplikací na platformě iOS. Apple tento vzor široce využívá ve svých knihovnách, převážně však v rámci UIKit frameworku. Díky tomuto přístupu mohou vývojáři také organizovat aplikace do dobře oddělených a udržitelných částí. Tento vzor je navržen tak, aby oddělil uživatelské rozhraní (View), logiku aplikace a samotná data (Model) a řídicí část (Controller). Díky tomuto oddělení lze aplikace jednodušeji testovat, rozšiřovat a udržovat. Tento přístup je vhodný primárně pro menší až středně velké projekty, kde je jeho struktura jednoduchá na implementaci. Stejně jako v případě MVVM se tento vzor skládá ze tří hlavních komponent.

Model zajišťuje správu dat a business logiku aplikace. Obsahuje struktury nebo objekty, které reprezentují data aplikace, a také metody pro jejich manipulaci. Model by měl být plně nezávislý na uživatelském rozhraní.

View i zde reprezentuje to, co uživatel vidí na obrazovce. Obsahuje prvky uživatelského rozhraní, jako jsou tlačítka, textová pole nebo seznamy a slouží k zobrazování dat z Modelu.

Controller potom funguje jako prostředník mezi View a Modelem. Zpracovává uživatelské vstupy z View, manipuluje s daty v Modelu a zajišťuje aktualizaci uživatelského rozhraní.

Na ukázce kódu níže je opět jednoduchá implementace počítadla, tentokrát za použití MVC architektury. Pro názornou demonstraci je tento ukázkový příklad implementován

pomocí UIKit frameworku, který je pro tuto architekturu vhodnější než novější SwiftUI framework. Kód obsahuje model `Counter`, který se stará o business logiku aplikace – v tomto případě inkrementace hodnoty počítadla. Controller `CounterViewController` řeší volání metod na modelu a následnou aktualizaci view – zároveň ale také zpracovává události vyvolané uživatelem prostřednictvím view. V UIKit projektu by samotná definice view byla součástí `.storyboard` souborů, nicméně pro zjednodušení zde nejsou uvedeny a řeší se zde pouze jednoduchá aktualizace textu počítadla.

Díky oddělení komponent umožňuje jasnou organizaci kódu, což přispívá k lepší udržovatelnosti a testovatelnosti. I když může být pro menší aplikace ideální, u větších projektů je často nahrazován pokročilejšími vzory, jako je například MVVM. Tento příklad jednoduché aplikace ukazuje, jak má každá část jasně definovanou zodpovědnost. [15]

```
class Counter {
    private(set) var value: Int = 0

    func increment() { value += 1 }
}

class CounterViewController: UIViewController {
    @IBOutlet weak var counterLabel: UILabel!

    private var counter = Counter()

    ... // Další kód

    @IBAction func incrementTapped(_ sender: UIButton) {
        counter.increment()
        updateUI()
    }

    private func updateUI() {
        counterLabel.text = "\(counter.value)"
    }
}

// View definován v rámci .storyboard souborů
```

2.3.3 Srovnání a rozdíly

Oba architektonické vzory, MVC i MVVM, patří mezi nejčastěji používané přístupy při vývoji iOS aplikací. Nabízejí strukturovaný způsob, jak oddělit uživatelské rozhraní od datové logiky, což zvyšuje udržitelnost a usnadňuje testování aplikací. Tyto přístupy však mají své specifické výhody a nevýhody, díky kterým jsou vhodné pro různé typy projektů.

MVC je velmi jednoduchý na implementaci a je ideální pro menší aplikace – odděluje data a logiku (Model), uživatelské rozhraní (View) a řídicí část aplikace (Controller). Tento vzor je intuitivní, což je velmi dobré pro začátečníky a projekty využívající UIKit, kde Apple tento přístup lépe podporuje. Nevýhodou MVC je však tendence k přetěžování controlleru (velké a obsáhlé controller třídy), což vede ke vzniku tzv. Massive View Controller, který může ztěžovat udržitelnost kódu u složitějších aplikací. [15]

MVVM naopak poskytuje lepší oddělení odpovědností díky zavedení komponenty ViewModel. Ta funguje jako most mezi modelem a view, což usnadňuje přípravu dat pro uživatelské rozhraní a zpracování uživatelských interakcí. MVVM je obzvláště vhodný pro projekty ve SwiftUI, kde má díky deklarativnímu stylu lepší podporu. I když je MVVM složitější na implementaci a vyžaduje více úsilí při návrhu, jeho výhody v lepší testovatelnosti a udržitelnosti se projeví primárně u komplexnějších a větších aplikací. [14]

Zatímco MVC je ideální pro menší projekty s jednoduchou logikou a jasně definovaným uživatelským rozhraním, MVVM je vhodný přístup u komplexních aplikací, kde je klíčová modularita, škálovatelnost a snadná rozšiřitelnost. Výběr mezi těmito dvěma vzory by měl být založen na velikosti projektu, použité technologii (UIKit vs. SwiftUI) a požadavcích na budoucí údržbu aplikace.

3 BACKENDOVÉ TECHNOLOGIE A AUTENTIFIKACE

V této části budou rozebrány backendové technologie a autentifikace, se zaměřením na platformu Firebase. Nejprve bude představena samotná Firebase a její nabízené služby, jako jsou Firebase Authentication, Firestore, Realtime Database a další. Následně se bude práce věnovat autentifikaci a zabezpečení, včetně přístupu třetích stran pomocí OAuth, podpory SSO nebo e-mailové autentifikace.

3.1 Představení Firebase

Firebase je platforma pro vývoj aplikací poskytovaná společností Google. Je primárně navržena tak, aby vývojářům umožnila efektivně vytvářet, spravovat a škálovat mobilní i webové aplikace. Firebase nabízí velké množství služeb, které se zaměřují na ukládání dat, autentizaci uživatelů, analytiku, cloudové funkce, AI/ML komponenty a další. Díky integraci s Google Cloud Platform (GCP) je často využívána jako komplexní řešení pro backendovou infrastrukturu aplikací.

Firebase konkrétně nabízí nástroje pro ukládání dat, autentizaci uživatelů, synchronizaci dat v reálném čase a provozování serverless funkcí. Mezi jednu z nejvyužívanějších služeb patří bezesporu Firestore a Realtime Database pro ukládání dat anebo Firebase Authentication pro správu uživatelů. Velkou výhodou je potom integrace a propojení jednotlivých služeb, kdy například při registraci uživatele pomocí Firebase Authentication se díky triggeru do databáze uloží informace anebo další metadata o uživateli. Firebase obsahuje také nástroje pro testování aplikací, jako je Test Lab, pro sledování jejich výkonu a stability pomocí Crashlytics nebo Performance Monitoring, což pomáhá při sledování a monitorování aplikace v produkci. Dále nabízí možnosti pro komunikaci a zapojení uživatelů, například pomocí Firebase Cloud Messaging pro zasílání notifikací nebo Remote Config pro dynamické konfigurování aplikace.

Firebase je vhodná pro malé i velké projekty. Pro malé projekty nabízí jednoduché nastavení – vše je integrováno do sebe a umožní tak malým projektům rychlý start, zatímco pro větší projekty je možné využít její škálovatelnost a integraci s GCP. Nicméně u větších projektů se může cena vyšplhat poměrně vysoko, zvláště pokud se některé služby špatně nastaví anebo nejsou optimalizovány (například čtení z databáze). [16]

Firebase funguje na principu freemium modelu – základní služby, jako je Firestore, Authentication nebo Analytics, jsou dostupné zdarma v rámci Spark plánu, který však

zahrnuje omezení, jako jsou například limity na množství uložených dat nebo počet operací. K dispozici je potom také plán Blaze, který je založen na platbě za skutečně využitý zdroj (pay-as-you-go). Tento platební model umožňuje přizpůsobit náklady velikosti projektu, nicméně při rostoucím objemu dat či uživatelů mohou náklady výrazně narůstat. [17]

3.2 Nabízené služby

V následujících kapitolách budou stručně popsány nabízené služby Firebase, které jsou zároveň považovány za relevantní v kontextu této práce.

3.2.1 Firebase Authentication

Služba Firebase Authentication umožňuje vývojářům snadno integrovat autentizační mechanismy do mobilních a webových aplikací. Google nabízí také mnoho SDK anebo knihoven pro velké množství programovacích jazyků, které stačí přidat do projektu a potom pouze zavolat adekvátní metodu – není tedy nutné vše implementovat manuálně od nuly. Firebase Authentication Podporuje různé metody přihlašování, včetně e-mailu a hesla, telefonního čísla, účtů třetích stran (např. Google, Facebook, Apple) - SSO anebo také anonymního přihlášení. Dále je zde možnost využít funkce pro ověření emailu nebo změny hesla. Hlavním cílem Firebase Authentication je zjednodušit proces ověřování uživatelů a správu jejich identit, aby vývojáři nebo firma nemuseli mít vlastní implementaci autentizačních mechanismů, které jsou velmi důležité a kladou velký důraz na bezpečnost. Služba poskytuje nástroje pro bezpečné uložení přihlašovacích údajů a správu přihlášených uživatelů prostřednictvím integrované Firebase konzole. Autentizace je úzce propojena s dalšími službami, jako jsou například databáze, což umožňuje snadno přistupovat k datům anebo spravovat oprávnění uživatelů v rámci aplikace. Tato služba je ideální pro projekty, které vyžadují jednoduchou a bezpečnou autentizaci bez nutnosti ruční implementace komplexní logiky na vlastním backendu. Díky bezplatnému plánu se hodí i pro menší aplikace. [18]

3.2.2 Firestore a Realtime Database

Obě služby, jak Firestore, tak i Realtime Database jsou databázové služby poskytované na Firebase, které umožňují ukládání a synchronizaci dat pro mobilní i webové aplikace. Obě databáze jsou navrženy tak, aby zjednodušily práci s daty a umožnily rychlou integraci s dalšími službami dostupnými na Firebase, jako například autentizace anebo cloud functions. Každá z nich má však své specifické vlastnosti a využití.

Realtime Database je původní databáze na Firebase, která ukládá data v JSON formátu a nabízí synchronizaci v reálném čase mezi klienty. Je ideální volbou pro aplikace, u kterých je potřeba okamžitá aktualizace dat, jako jsou chatovací aplikace nebo aplikace pro sdílení polohy. Realtime Database je však méně flexibilní při strukturování dat a neumožňuje složitější dotazy, což může být u komplexnějších aplikací problém, především z hlediska výkonu. Tato databáze se často využívá například při prototypování, kdy není prioritou optimalizace výkonu nebo velikosti, ale spíše rychlost samotného vývoje bez zdržování se složitým nastavením a konfigurací. Je tedy vhodná pro menší projekty a aplikace vyžadující jednoduchý datový model. [19]

Firestore je ve srovnání s Realtime Database novější databázová služba, která ukládá data ve struktuře kolekcí a dokumentů – jedná se tedy taktéž o NoSQL databázi. Nabízí podporu pokročilejších dotazů, offline režimu a škálovatelnost na velký objem dat. Firestore podporuje také synchronizaci v reálném čase, nicméně ve srovnání s Realtime Database umožňuje efektivněji pracovat s hierarchicky strukturovanými daty. Díky tomu je vhodnější pro složitější aplikace, které vyžadují flexibilní datový model. Firebase SDK, které si lze přidat do projektu, potom nabízí již implementované funkce jako offline režim a synchronizaci dat, když je aplikace zpět online, správu mezipaměti, optimalizaci a metody potřebné k připojení se k databázi, což značně ulehčí celý proces vývoje. Hodí se tedy více pro aplikace s komplexnějšími požadavky na data, větším objemem uživatelů a potřebu složitějších dotazů. [20]

3.2.3 Firebase Cloud Functions

Jedná se o tzv. serverless službu, což znamená, že umožňuje vývojářům spouštět kód na Google serverech, bez nutnosti správy vlastní infrastruktury. Cloud Functions se nejčastěji používají k implementaci logiky na backendu, která reaguje na různé události, jako jsou změny v databázi (Firestore, Realtime Database), příjem HTTP požadavků, autentizace uživatelů nebo události v rámci dalších Firebase služeb. Příkladem může být odeslání notifikace při přidání nového záznamu do databáze nebo validace dat před jejich uložením. Mezi hlavní výhody patří automatické škálování, bezpečnost (kód běží v izolovaném prostředí) a samozřejmě i integrace s ostatními službami v rámci Firebase ekosystému. Na rozdíl od tradičních backendů není potřeba spravovat servery ani se starat o jejich dostupnost, což také ušetří spoustu času a peněz firmám anebo projektům. Použití je vhodné hlavně u aplikací, které vyžadují flexibilní backendové funkce, ale nemají dostatečné zdroje pro provoz vlastního serveru. Je však nutné podotknout, že je potřeba mít

také aktivovaný Blaze plán (pay-as-you-go model), což znamená, že náklady odpovídají počtu vykonaných funkcí a doby jejich běhu. Firebase samozřejmě nabízí také variantu s bezplatným limitem pro menší aplikace. [21]

II PRAKTICKÁ ČÁST

4 PRŮZKUM TRHU A KONKURENCE

V následujících kapitolách bude provedena analýza konkurenčních aplikací, jejich nabízených funkcí a porovnání s vlastnostmi a přínosy této práce. Bude zdůrazněno, v čem se tato práce odlišuje, a popsány specifické výhody, které nabízí uživatelům oproti konkurenčním řešením.

4.1 Mise a cíl aplikace

Hlavním cílem a misí této práce a aplikace (dále Vulture) je primárně pomoci původním uživatelům a řidičům vozidel se spalovacím motorem plynule přejít na elektrický pohon a zbavit je nadbytečného přemýšlení a plánování, které je spojeno s vlastněním elektrického vozidla. Pomáhá jim lépe zvládat další režijní náklady spojené s plánováním nabíjení jejich elektrického vozidla díky poskytování užitečných informací a statistik, přehledně a na jednom místě, aby nemuseli informace čerpat z různých zdrojů.

Mezi hlavní přednosti a jedinečné konkurenční výhody aplikace patří její velké množství výpočtů, které nabízí na jednom místě, ať už se jedná o samotný čas nabíjení, včetně startu a konce, přidané kWh a vzdálenosti, cenu za nabití, dopad na životní prostředí anebo také graf rychlosti nabíjení v čase. Dále byl v rámci aplikace kladen velký důraz na jednoduchost a intuitivnost celého uživatelského prostředí – aplikace nabízí moderní a minimalistický vzhled, který uživatele provede celým procesem každého případu užití. Velkou výhodou je také online synchronizace uživatelských dat – pokud by se tedy uživatel rozhodl migrovat na jiné zařízení, data jako svá auta anebo nabíječky tam bude mít po přihlášení dostupná, což spousta konkurenčních aplikací nedělá. Je zde samozřejmě dostupný i offline režim s následnou synchronizací dat ve chvíli, kdy je uživatel znovu online.

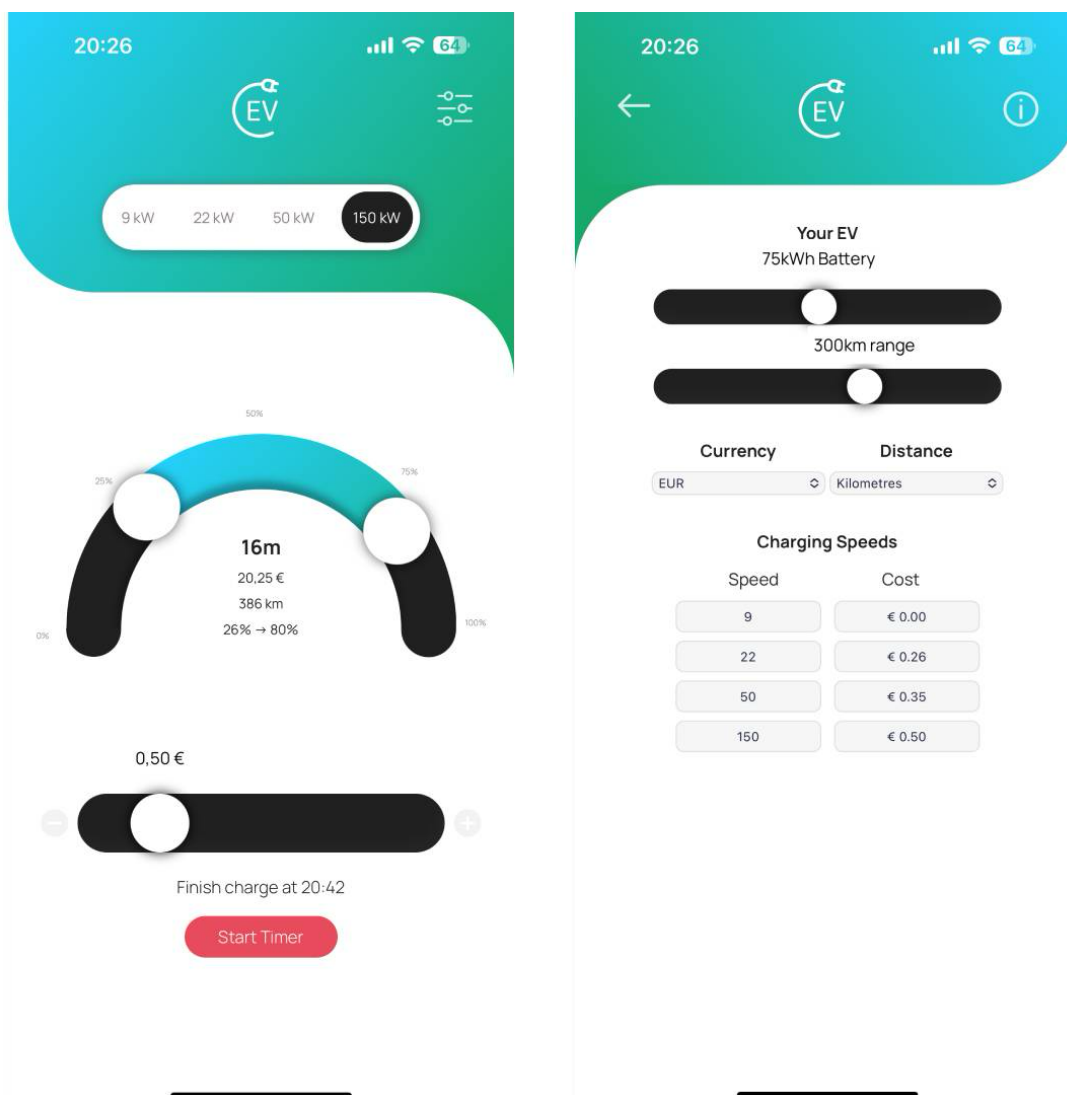
4.2 Konkurenční řešení

Na trhu již existuje několik řešení, které se věnují podobnému tématu. V následujících kapitolách bude provedena analýza těchto aplikací a také funkcí, které nabízí.

4.2.1 EV CALC

Aplikace funguje spíše jako simulátor nabíjení, který uživateli zobrazuje odpočítávání času během simulovaného procesu. EV CALC obecně zaujme líbivým a minimalistickým designem, nicméně orientace v aplikaci může být pro uživatele místy neintuitivní – není

zcela jasné, kam a jak přidat některé potřebné údaje pro výpočty. Jednou z hlavních funkcí aplikace je zobrazení různých rychlostí nabíjení včetně odhadovaných nákladů. Výpočty aplikace však obsahují pouze stanovení času nabíjení, jeho ceny a přidaného dojezdu, což je relativně málo informací. Mezi zajímavé funkce patří možnost zapnout „realistickou křivku nabíjení“, která během simulace reflektuje reálné průběhy nabíjení podle stavu nabití baterie (Obrázek 3 - EV CALC – Výpočty a nastavení).



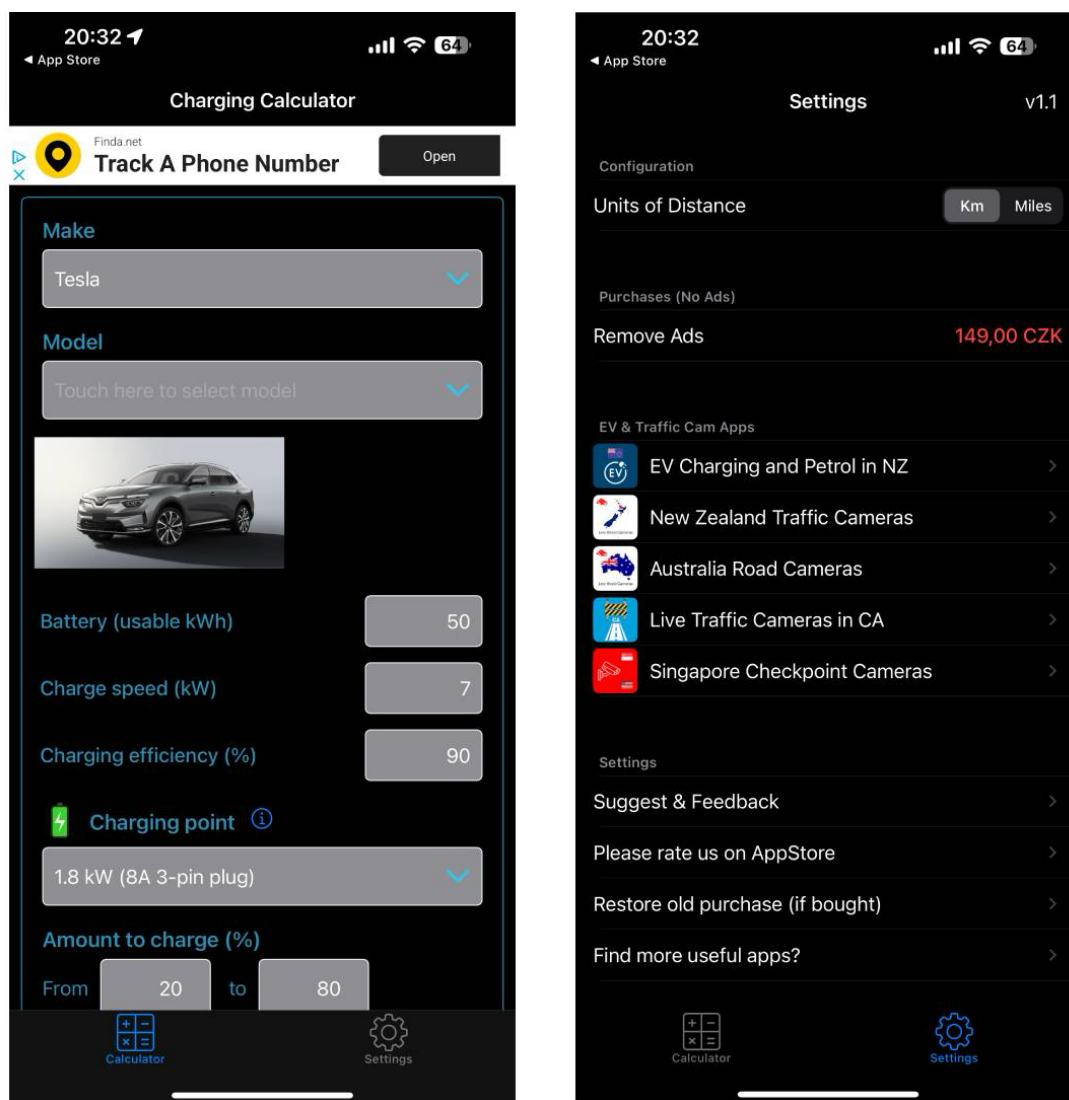
Obrázek 3 - EV CALC – Výpočty a nastavení

(zdroj: vlastní)

4.2.2 EV Charge Calculator – Offline

Tato aplikace působí zastarale a nemá příliš intuitivní design, což může uživatele odrazovat. K negativnímu dojmu přispívají i časté celobrazovkové reklamy, které lze odstranit za poplatek 149 Kč. Funkcionalita aplikace je poměrně omezená – nabízí pouze základní

výpočty, jako jsou náklady na nabíjení, dodaná energie, přidaný dojezd a čas potřebný k nabití. Dalším nedostatkem je podivný výběr nabíječky, který působí velmi nepřehledně – je zde možnost vybrat z předdefinovaného seznamu dat, kde však chybí spousta možností. Celkově aplikace zaostává jak v uživatelské přívětivosti, tak v kvalitě zpracování, čímž je méně atraktivní pro uživatele, kteří hledají jednoduché a efektivní řešení (Obrázek 4 - EV Charge Calculator – Výpočty a nastavení).



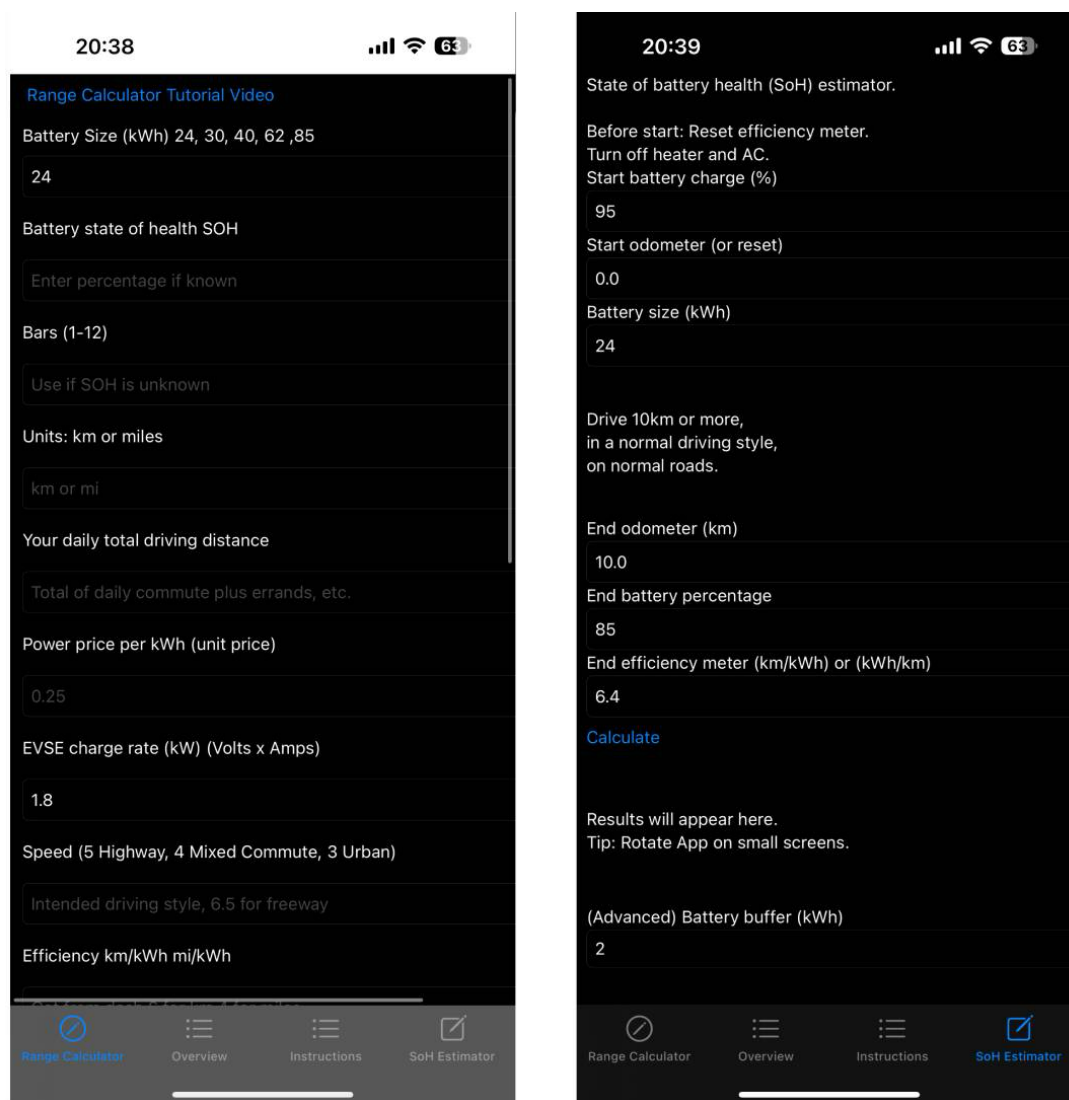
Obrázek 4 - EV Charge Calculator – Výpočty a nastavení

(zdroj: vlastní)

4.2.3 EV Range Calculator

Aplikace EV Range Calculator má taktéž velmi zastaralý a neintuitivní design, který působí spíše jako přizpůsobená webová stránka než plnohodnotná mobilní aplikace. Některé funkce se dokonce vůbec nenačítají, což může uživatele odradit od dalšího používání.

Aplikace je zaměřena spíše na výpočet vzdálenosti a plánování trasy než na detailní plánování nabíjení. Výsledky výpočtů a zobrazované hodnoty však působí zmateně a nejasně – uživatelé často nevědí, co přesně zadávají, ani co jednotlivé parametry znamenají. Celkově aplikace nenabízí příliš použitelný nástroj pro efektivní plánování nabíjení nebo správu elektromobilu, za což mohou primárně nedostatky v designu, funkčnosti i logice výpočtů (Obrázek 5 - EV Range Calculator – Výpočty vzdálenosti a zdraví baterie).



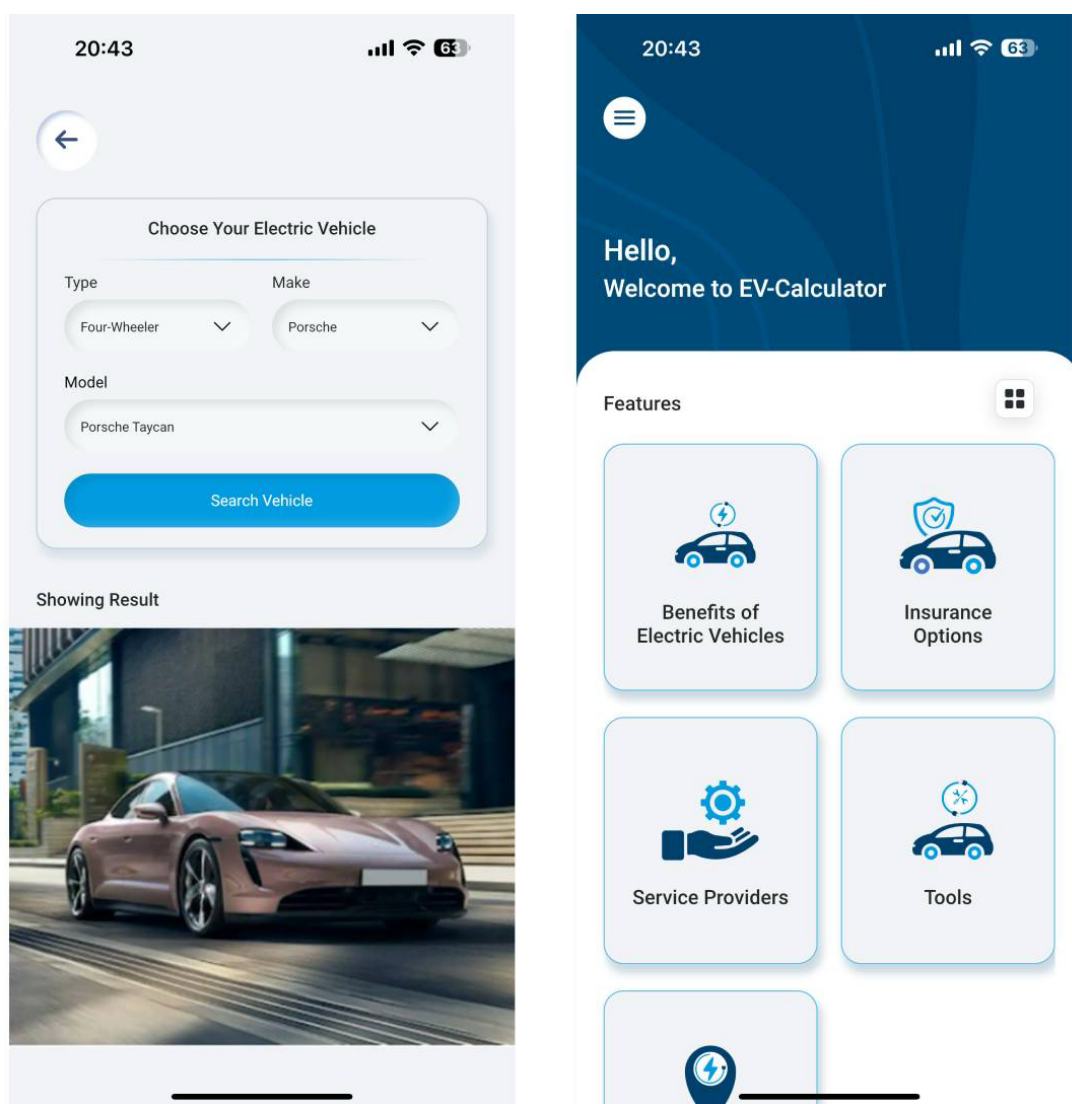
Obrázek 5 - EV Range Calculator – Výpočty vzdálenosti a zdraví baterie

(zdroj: vlastní)

4.2.4 EV Calculator – Cost Tracker

Aplikace EV Calculator může uživatele oslovit minimalistickým designem, který působí moderně, nicméně uživatelský zážitek z aplikace narušují časté a všudypřítomné reklamy, včetně těch celobrazovkových. Jedním z hlavních a velkých

nedostatků je omezený seznam vozidel, který postrádá populární značky jako je třeba Tesla. Není zde ani možnost zadat informace o vozidle manuálně. Aplikace sice nabízí širokou škálu nástrojů, jako jsou kalkulátory pro pojištění, daňové úspory, emise CO₂ nebo úspory ropty, ale výpočty a jejich výsledky jsou často matoucí. Uživatelé nemusí jasně chápat význam zadaných parametrů a výstupy z nabíjecích kalkulaček obsahují informace převážně o tom, kolik peněz by uživatel ušetřil, namísto podrobnějších údajů o nabíjení. Navigace v samotné aplikaci může být pro některé uživatele trochu matoucí, protože je nutné se proklikat přes několik stránek a ikon, než se uživatel dostane tam, kam chce – chybí zde nějaké jednoduché a přehledné menu (Obrázek 6 - EV Calculator – Výběr vozidla a nástroje pro výpočet).

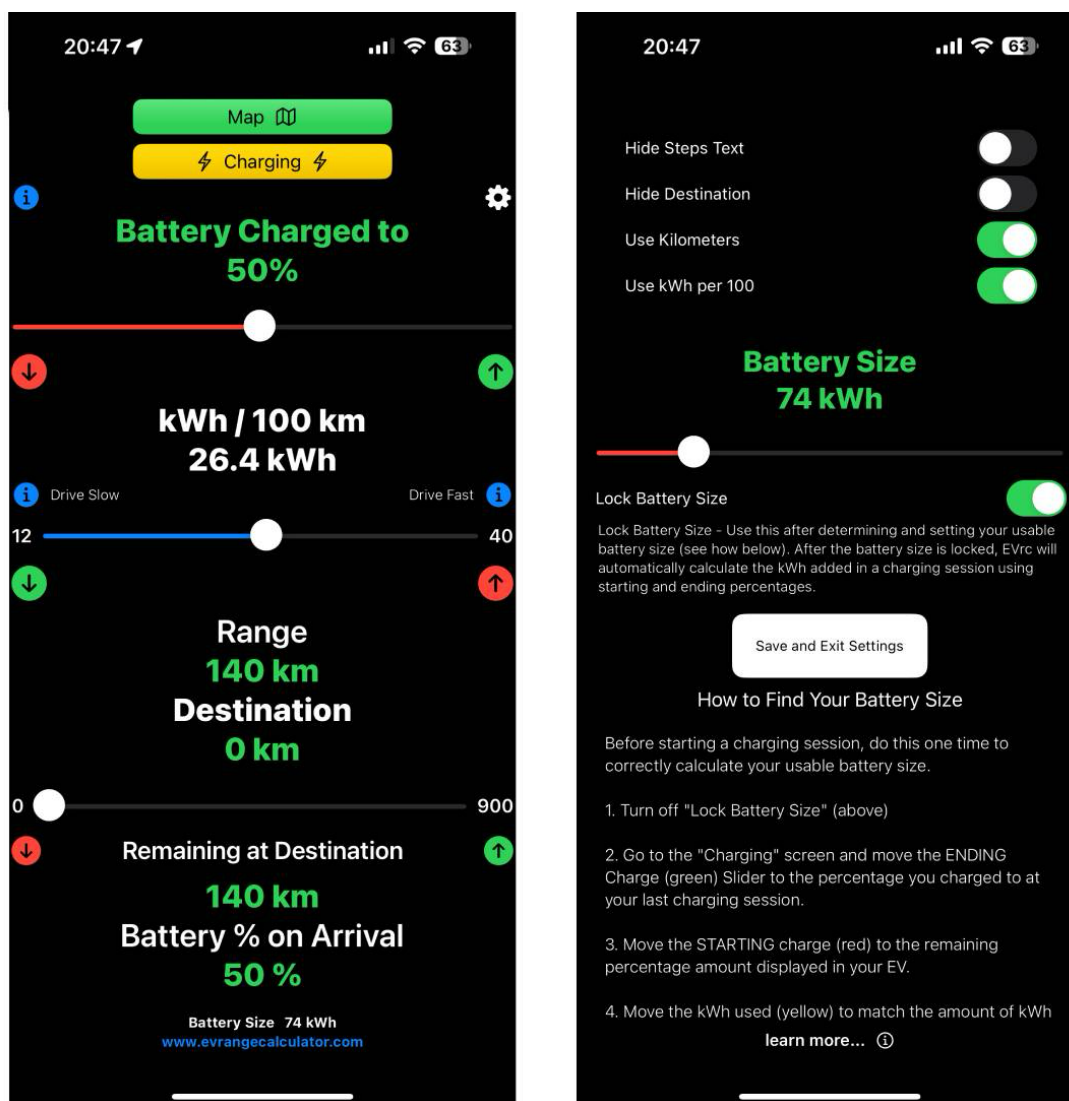


Obrázek 6 - EV Calculator – Výběr vozidla a nástroje pro výpočet

(zdroj: vlastní)

4.2.5 EVrc

Tato aplikace se vyznačuje velmi špatným designem, který uživatele odrázuje hned na první pohled. Jednou z hlavních funkcí je integrovaná mapa nabíjecích stanic, která však není aktuální. Navíc aplikace často padá při pokusu o zobrazení detailů nabíjecích míst. Zadávaní vstupů pro výpočty je v rámci této aplikace a kontextu neobvyklé a spoléhá pouze na posuvníky, což může být neintuitivní a nepřesné. Aplikace se navíc zaměřuje spíše na plánování tras – například na to, zda lze dojet na určité místo – než na podrobné parametry nabíjení. EVrc zaostává v klíčových oblastech, jako je uživatelská přívětivost. Ačkoliv nabízí základní funkce pro plánování cest, její technické nedostatky a zastaralý design ji dělá pro uživatele velmi neatraktivní (Obrázek 7 - EVrc – Výpočty a nastavení).



Obrázek 7 - EVrc – Výpočty a nastavení

(zdroj: vlastní)

4.2.6 EV Charge Calculator and News

Poslední aplikace, EV Charge Calculator and News, představuje asi nejbližší konkurenci pro Vulture. Aplikace nabízí například možnost přidat vozidlo s manuálním zadáním průměrné spotřeby, kapacity baterie a názvu, nicméně bez možnosti výběru ze seznamu a automatického doplnění dat. Design aplikace je na první pohled příjemný a moderní, avšak občasné chyby v uživatelském rozhraní, jako nereagující vstupy nebo zvláštní chování při scrollování, kazí celkový dojem. Výpočty doby nabíjení jsou někdy nepřesné, zejména při zapnutí volby realistické křivky nabíjení (je zde však možnost ji vypnout). Kromě základních parametrů, jako jsou náklady, čas a dodané kWh, nejsou k dispozici žádné další rozšířené výpočty. Aplikace obsahuje také sekci s novinkami, ty však nejsou aktuální ani dostupné. Zajímavou funkcí je možnost porovnání elektromobilů s vozy se spalovacím motorem, což může být přínosné pro uživatele zvažující přechod na elektromobilitu. Správa vozidel je v aplikaci dobře zpracovaná a je přehledná, ale chybí zde online synchronizace mezi zařízeními. Aplikace je zcela bez reklam a stojí 29 Kč, což je příjemná cena za funkce, které nabízí, avšak pořizovací cena může některé uživatele odradit (Obrázek 8 - EV Charge Calculator – Výpočty a výsledky).

The image displays two screenshots of the 'EV Charge Calc & News' mobile application interface.

Left Screenshot (20:50):

- Vehicle data:**
 - Vehicle: Tesla Model 3
 - Battery capacity (kWh): 75.0
 - Consumption (kWh/100km): 12.5
- Charging parameters:**
 - €/kWh: (empty)
 - €/minute: (empty)
 - Fixed fee (€): (empty)
 - Charging speed (kW): (empty)
 - Range: from 20% → to 80%
 - Simulate efficiency loss and charge curve: ☐
- Result:** (empty)
- Bottom navigation:** Calculator (selected), Drive, Compare, News

Right Screenshot (20:51):

- Charging parameters:**
 - €/kWh: (empty)
 - €/minute: (empty)
 - Fixed fee (€): (empty)
 - Charging speed (kW): (empty)
 - Range: from 20% → to 70%
 - Simulate efficiency loss and charge curve: ☒
- Result:**
 - Finish charge in: 0m
 - at: 20:51
 - Cost: 0.00 €
 - kWh: 37.36
 - €/kWh: 0.00 €
 - Kmkm available: 419 Km
- Bottom navigation:** Calculator, Drive, Compare, News

Obrázek 8 - EV Charge Calculator – Výpočty a výsledky

(zdroj: vlastní)

5 ARCHITEKTURA A ROZDĚLENÍ PROJEKTU

V následujících kapitolách bude popsána zvolená MVVM architektura v kontextu tohoto projektu, včetně její konkrétní implementace, výhod a problémů či výzev, které se během implementace objevily. Dále bude rozebrána samotná struktura projektu, včetně tříd a dalších souborů.

5.1 Implementace MVVM

Pro tuto práci byla zvolena architektura MVVM (Model-View-ViewModel) hlavně z důvodů, které vycházely jak z povahy projektu, tak z charakteristik použitého frameworku SwiftUI. Největší výhodou je v tomto kontextu to, že MVVM architektura svou strukturou přirozeně podporuje oddělení datové logiky od uživatelského rozhraní, což je zásadní pro udržitelnost a rozšiřitelnost aplikace do budoucna. Dalším důvodem zvolení MVVM architektury je způsob, jakým SwiftUI přistupuje k tvorbě uživatelského rozhraní. Samotný framework je založen na deklarativním přístupu, což znamená, že uživatelské rozhraní přímo reaguje na změny dat. MVVM architektura tento přístup efektivně podporuje tím, že například ViewModel slouží jako zprostředkovatel mezi modelem a pohledem. To umožňuje SwiftUI využívat vlastnosti a funkce jako jsou například data bindingy (`@ObservedObject`, `@State`, `@EnvironmentObject`) k automatické synchronizaci dat mezi jednotlivými částmi aplikace, takže vývojáři odpadá nutnost implementace další logiky, která by se o to musela starat. MVVM architektura byla zvolena také s ohledem na budoucí rozšiřitelnost aplikace – díky oddělení jednotlivých vrstev bude možné tyto funkce přidávat bez zásadního zásahu do existujícího kódu, což zjednodušuje údržbu a vývoj.

5.2 Výzvy při implementaci

Během implementace zvolené MVVM architektury v této práci se jako výzva ukázaly nejen správné rozdělení logiky mezi View a ViewModel (potažmo servisními třídami), ale také efektivní správa stavových objektů a jejich předávání napříč aplikací. SwiftUI nabízí funkce jako `@EnvironmentObject`, které umožňují sdílení dat mezi různými obrazovkami, ale jejich nesprávné použití může vést ke komplikacím. Pokud není například `@EnvironmentObject` správně inicializován nebo předán do hierarchie obrazovek, dochází k runtime chybám, které se často velmi obtížně ladí. Možnost je potom také ještě předat samotné objekty mezi třídami manuálně pomocí konstruktorů. Nevýhodou je zde to,

že vytvoření instance objektu se musí provést manuálně na místě, kde se používá daný pohled a předat mu tento objekt hned při vytváření.

Dalším, podobným problémem, který byl během implementace zaznamenán, bylo předávání hodnot nebo objektů, inicializovaných přímo ve View, do ViewModelu. Typickým příkladem bylo opět použití `@EnvironmentObject`, konkrétně třeba `UserPreferences`, které slouží k uchovávání uživatelských nastavení a jsou z části řízeny frameworkem SwiftUI. Tyto hodnoty byly potřebné pro výpočty ve ViewModelu, ale protože byly inicializovány ve View, nastala zde otázka, jak to správně udělat. Bylo tedy možné buď logiku výpočtů ponechat ve View, což by porušovalo principy MVVM, nebo předat hodnotu jako argument do setter metody ViewModelu, což byla nakonec také zvolená varianta, protože zajišťuje čistší oddělení odpovědností. Tato varianta zároveň přinesla komplikace v podobě potřeby správného řízení předávání dat a zvolený přístup vyžaduje pečlivou synchronizaci mezi View a ViewModelem, aby byly hodnoty vždy aktuální a zohledňovaly stav aplikace. Tato výzva také ukázala, že některé závislosti, jako například `UserPreferences`, by mohly být spravovány na vyšší úrovni, například pomocí Dependency Injection, což by zjednodušilo jejich celkovou dostupnost napříč aplikací. Během implementace bylo občas složité určit, která logika má zůstat ve ViewModelu a která přímo ve View. Nicméně ve většině případů bylo rozhodnuto minimalizovat logiku ve View a přesunout ji do ViewModelu nebo samostatných servisních tříd, aby byl kód přehledný a nedocházelo k porušování principů MVVM architektury.

5.3 Hlavní struktura projektu

Samotná struktura projektu, složek, tříd, assetů a dalších souborů byla vytvořena tak, aby byla, pokud možno, co nejvíce v souladu s vybranou MVVM architekturou, dala se v budoucnu jednoduše rozšířit a v projektu se dalo rychle vyznat, pokud by na něm pracovalo více vývojářů. Projekt obsahuje hlavní strukturu, která se skládá z adresářů `App` a `Features`. V následující kapitole budou popsány komponenty, které jsou sdílené mezi jednotlivými funkcemi aplikace.

V adresáři `App` se nachází pouze jeden soubor, `VultureApp`, který obsahuje hlavní funkci `main`, jakožto vstupní bod do aplikace. V rámci tohoto souboru je potom ještě obsažena inicializace Firebase knihovny a výchozí nastavení aplikace (například velikost písma).

V rámci adresáře `Features` jsou potom jednotlivé podadresáře, ve kterých jsou komponenty funkcí aplikace, které jsou v souladu s MVVM architekturou. Například funkce `Vehicles`, která se stará o správu vozidel, dále obsahuje pod-adresáře `Models`, `Views`, `ViewModels`, v nichž jsou korespondující soubory s kódem. Jednotlivé funkce aplikace budou potom detailněji rozebrány v následujících kapitolách. Díky této struktuře je možné aplikaci velmi jednoduše škálovat a rozšiřovat, protože pro integraci další funkce stačí pouze přidat nový adresář s korespondujícím kódem a není nutné vůbec modifikovat existující strukturu (pokud se samozřejmě nezmění požadavky). Za zmínku stojí ještě také adresář `Common`, který se neváže pouze k jedné z funkcí, ale obsahuje spíše obecné věci, jako je například `wrapper`, který zobrazí adekvátní obrazovku na základě toho, zdali je uživatel přihlášen, anebo ne – tedy buď obrazovku pro přihlášení anebo domovskou obrazovku. Dále také obsahuje komponenty, které jsou sice použity na více místech, nicméně jsou nějakým způsobem vázané s doménou aplikace a nejsou to obecné komponenty (obrázek s logem aplikace), jako například tlačítko anebo vstupní pole, které patří do sdíleného adresáře s komponenty, jak bude následně popsáno v kapitole níže.

5.4 Sdílené komponenty projektu

Protože aplikace obsahuje mnoho logiky a komponent, které se využívají napříč různými funkcemi, bylo potřeba tuto skutečnost zohlednit ve struktuře kódu. V rámci adresáře `Shared`, který je v root projektu, stejně jako například `App` anebo `Features`, popsané výše, bylo vytvořeno také několik pod-adresářů, které obsahují například servisní třídy, nástroje, UI elementy a další. Komponenty v tomto `Shared` adresáři mohou být použity v rámci více funkcí a není nutné je izolovat, jako právě jednotlivé doménové funkce aplikace, což umožní efektivní znovu-použití kódu bez nutnosti repetitivní deklarace anebo implementace.

5.4.1 Servisy

Servisní třídy slouží k implementaci logiky, která je často využívána ve `ViewModelech` jednotlivých funkcí aplikace. Jejich hlavním účelem je odlehčit `ViewModely` přenesením odpovědnosti a logiky do samostatných tříd, což přispívá k lepší přehlednosti kódu a usnadňuje jeho rozšiřování. Většina servisních tříd v tomto projektu implementuje protokol `ObservableObject`, což znamená, že poskytují reaktivní data pro `SwiftUI`, umožňují aktualizaci uživatelského rozhraní při změnách dat. Jejich životní cyklus závisí na tom, jak jsou spravovány (například jako `singletons` nebo prostřednictvím `@EnvironmentObject`), což často znamená, že žijí po celou dobu běhu aplikace.

Většina servisních tříd obsahuje statickou instanci definovanou jako `shared`, což zajišťuje, že v rámci celé aplikace existuje pouze jedna sdílená instance této třídy (návrhový vzor singleton).

```
public static let shared = FirestoreService()           // definice
let firestoreService = FirestoreService.shared         // použití
```

Mezi hlavní servisy patří třída `FirestoreService`, která slouží jako hlavní bod pro práci s Firestore databází v aplikaci. Třída obsahuje funkce jako odesílání uživatelské zpětné vazby, ukládání a načítání dat o vozidlech a nabíječkách spojených s konkrétními uživateli a správy dat o CO₂ emisích podle zemí. V rámci inicializačního bloku této třídy je potom ještě také konfigurace Firestore cache, která je použita pro offline přístup a následnou synchronizaci.

Servisní třída `AuthService` slouží pro správu autentizace v aplikaci. Nabízí podporu přihlašování a registrace prostřednictvím e-mailu, Google a Apple účtů, včetně funkcí pro resetování a změnu hesla. Třída zajišťuje také inicializaci stavu přihlášeného uživatele, což znamená, že ihned po vytvoření instance se určí, zda je uživatel přihlášen. Jakákoliv změna, například odhlášení uživatele, se automaticky promítne do uživatelského rozhraní. Tohoto chování je dosaženo pomocí listeneru, který sleduje změny v uživatelském účtu a dynamicky aktualizuje příslušný stav. Třída integruje všechny podporované způsoby přihlašování (e-mail a heslo, Apple a Google) tím, že deleguje specifickou logiku jednotlivých metod do samostatných tříd.

Mezi další servisní třídy patří také `ChargingCalculationService`, která slouží k provádění výpočtů souvisejících s procesem nabíjení elektromobilů. Zajišťuje například výpočet přidané kapacity baterie, trvání nabíjení, nákladů na nabití a environmentálního dopadu. Třída také poskytuje detailní analýzu, jako je cena za dojezdovou vzdálenost, odhadovaný dojezd po úplném nabití anebo dynamické modelování výkonu nabíjení v závislosti na stavu nabití baterie.

Dále se v tomto servisním adresáři nacházejí třídy jako například `UserPreferences`, která spravuje uživatelská nastavení v aplikaci (např. barevný režim nebo preferovaný systém jednotek), a `HapticFeedbackService`, která obsahuje metody pro vyvolání různých typů haptické odezvy telefonu.

5.4.2 UI komponenty

V rámci designového jazyka celé aplikace (detailněji popsáno později) bylo nutné vytvořit znovupoužitelné komponenty, které lze využít na více místech aplikace. Tento přístup odstraňuje potřebu opakovaného definování stejných prvků, což by mohlo způsobit problémy při jejich úpravách. Díky centralizované definici stačí změnu provést na jednom místě, což se automaticky promítne všude tam, kde je daná komponenta použita. Mnoho komponent má různé možnosti zobrazení, aby se přizpůsobily konkrétním potřebám. Například vstupní pole může zobrazovat text normálně, s maskováním (v případě hesla) nebo bez něj, anebo na konci pole může být ikona, případně nemusí. Proto bylo potřeba některé komponenty navrhnout jako konfigurovatelné, což umožňuje jejich vzhled a chování upravit pomocí předaných parametrů.

Jednou z nejdůležitějších komponent je `LargeButton` – tlačítko, které je navrženo v souladu s designovým jazykem aplikace a nabízí pokročilé vlastnosti a flexibilitu. Tlačítko může zobrazovat buď lokalizovaný text odpovídající zvolenému jazyku, nebo tzv. loading spinner během načítání. Má jasně definovanou maximální šířku, pevnou výšku, a dynamickou barvu textu, která se automaticky přizpůsobuje aktuálnímu barevnému režimu aplikace. Barva pozadí je konfigurovatelná podle toho, zda se jedná o primární nebo sekundární tlačítko. Dále je také možné k tlačítku přiřadit funkci, která se provede po jeho stisknutí. Na ukázce kódu níže je vidět definice tlačítka a také jeho použití v rámci aplikace. Ostatní komponenty potom následují podobnou strukturu.

```
struct LargeButton: View {

    let text: String
    let textColor: Color
    let backgroundColor: Color
    let showLoadingSpinner: Bool
    let action: () -> Void

    private let maxWidth: CGFloat = 400.0
    private let maxHeight: CGFloat = 55.0

    var body: some View {
        Button(action: action) {
            ZStack {
```

```

        if showLoadingSpinner {
            LoadingSpinner(spinnerColor: textColor)
                .transition(.opacity)
        } else {
            Text(LocalizedStringKey(text))
                .font(.mediumTitle)
                .foregroundColor(textColor)
                .transition(.opacity)
        }
    }
    .frame(maxWidth:      maxWidth,      minHeight:      maxHeight,
maxHeight: maxHeight)
    .background(backgroundColor)
    .cornerRadius(Constants.cornerRadius)
    .animation(.easeInOut(duration:      0.05),      value:
showLoadingSpinner)
    }
    .buttonStyle(PlainButtonStyle())
}
}

// Použití tlačítka v aplikaci
LargeButton(
    text: "get_started",
    textColor: Color.Vulture.textPrimary,
    backgroundColor: Color.Vulture.primary,
    showLoadingSpinner: true
) {
    print("Button pressed")
}

```

Jak již bylo zmíněno výše, tlačítko může místo textu zobrazit také loading spinner, například během čekání na odpověď ze serveru anebo dokončení jiné akce. Tato komponenta je taktéž speciálně navržena pro použití uvnitř tlačítka a umožňuje nejen plynulý přechod mezi textem a spinnerem, ale také přizpůsobení barvy tak, aby odpovídala barvě textu tlačítka, díky čemuž zůstává vzhled tlačítka konzistentní s designovým jazykem aplikace.

Pro sběr vstupních dat od uživatele byla vytvořena komponenta `InputField`, která taktéž umožňuje široké možnosti konfigurace. Je u ní možné nastavit lokalizovaný text placeholderu, který slouží jako nápověda nebo popis očekávaného vstupu od uživatele.

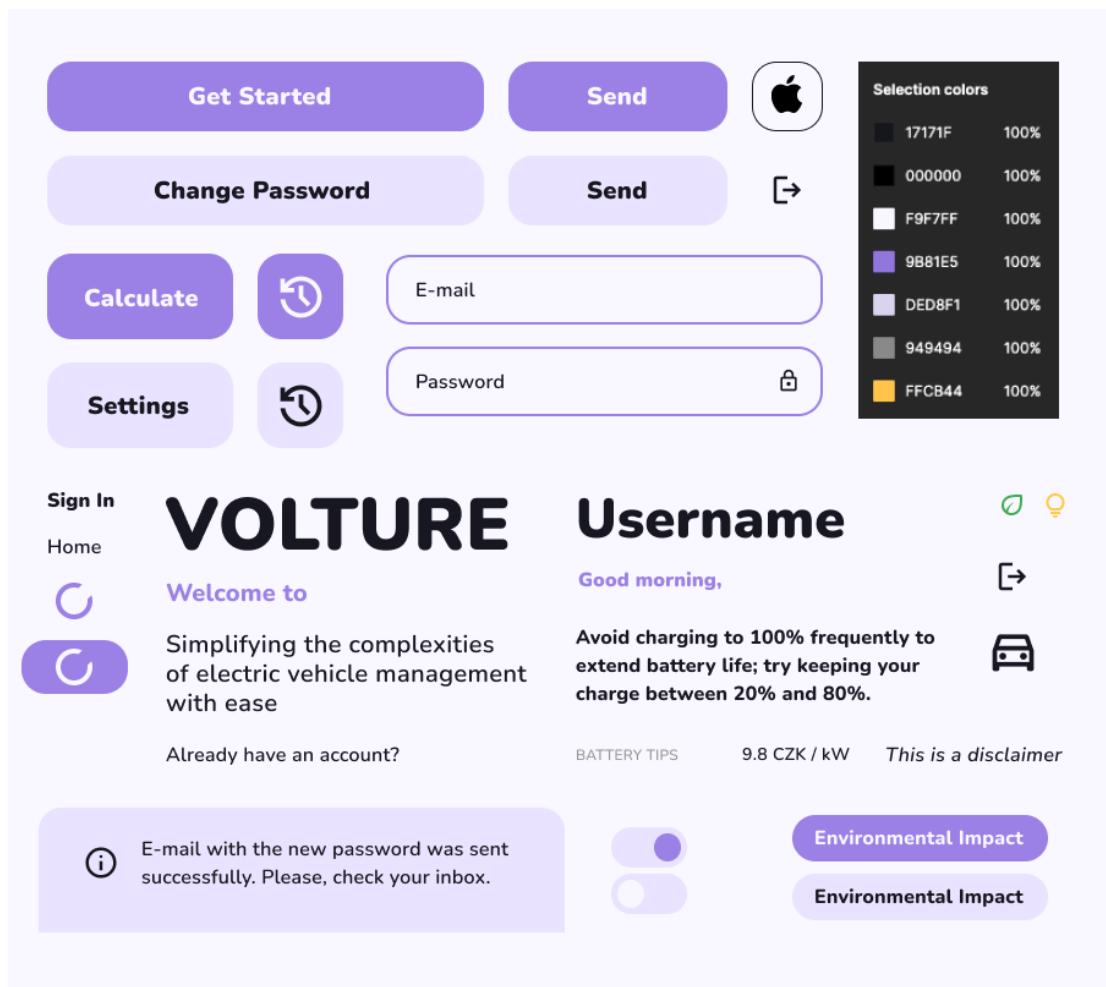
Komponenta podporuje zobrazení vstupních dat maskovaně díky parametru `isSecure`, který, pokud je nastaven na `true`, zamaskuje vstupní data za symbol tečky, což je ideální například pro zadávání hesel. Díky podpoře SwiftUI lze také definovat typ obsahu a klávesnice, což operačnímu systému umožňuje zobrazit vhodnou klávesnici nebo povolit zadávání specifických symbolů. Na konci vstupního pole může být také zobrazená ikona, která je součástí designového jazyka aplikace. Pro zajištění konzistentního designu a rozměrů napříč aplikací má i tato komponenta stejné rozměry jako například výše popsané tlačítko.

Protože SwiftUI a Cupertino design nenabízí nativní podporu pro komponentu `Snackbar`, která je nejčastěji spojována s Material Designem od Googlu, bylo nutné tuto komponentu vytvořit samostatně. `Snackbar` je ideální pro zobrazení informací, varování nebo chyb, například při validaci. Z hlediska UX je tento přístup mnohem přívětivější, protože nevyžaduje žádnou přímou interakci od uživatele, na rozdíl od modálního okna, které blokuje celý pohled. `Snackbar` naopak umožňuje aplikaci zůstat plně funkční, takže uživatel může okamžitě pokračovat v jejím používání. Tato komponenta je taktéž plně konfigurovatelná a umožňuje zobrazit tři úrovně závažnosti – informační, varovnou a chybovou. Před textem `Snackbaru` je zobrazena ikona, jejíž barva se dynamicky mění podle závažnosti (zelená pro informační zprávy, žlutá pro varování a červená pro chyby). Po zobrazení se `snackbar` potom automaticky zavře po třech sekundách. V případě potřeby lze do komponenty přidat volání metody pomocí tlačítka, které se zobrazí na konci řádku. Toto tlačítko může například provést přesměrování v aplikaci nebo potvrdit nějakou akci.

Jednou z dalších komponent aplikace je kombinace `OptionSheet` a `OptionButton`, které společně tvoří obsah modálního listu, umožňujícího uživateli vybírat například barevný motiv nebo systém jednotek v nastavení. `OptionSheet` přijímá jako parametr pole možností, které jsou poté zobrazeny uvnitř listu. Po kliknutí na některou z možností se automaticky zavolá metoda `onChange`, která nastaví aktuálně vybranou variantu a může například spustit haptickou odezvu. U vybrané možnosti je zobrazena ikona zaškrtnutí, která vizuálně zvýrazňuje aktuální volbu. Komponenta také využívá generiku, což umožňuje její použití s různými datovými typy.

V rámci aplikace existuje mnoho dalších komponent (Obrázek 9 - UI komponenty aplikace), které jsou použity na různých místech pro konzistentní uživatelský zážitek. V této kapitole byly popsány pouze ty nejdůležitější, nicméně mezi další významné

komponenty patří například `NavigationTileIconButton` a `NavigationTileButton`. Tyto dlaždice nebo tlačítka jsou umístěny na domovské obrazovce a slouží jako rozcestník k dalším funkcím aplikace. Dále je zde `SSOButton`, což je tlačítko určené pro přihlášení či registraci pomocí Apple nebo Google účtů, které zobrazuje příslušné logo. Mezi další znovupoužité komponenty patří tlačítka pro zavírání modálních oken nebo tlačítko zpět, které bylo taktéž specificky navrženo pro tuto aplikaci a využívá se na více obrazovkách.



Obrázek 9 - UI komponenty aplikace

(zdroj: vlastní)

5.4.3 Utilities a Extensions

V rámci kódu aplikace je často potřeba mít sdílené metody a logiku, které nejsou přímo vázané na konkrétní třídu anebo objekt. K tomuto účelu slouží různé pomocné nástroje, mezi které patří například funkce pro formátování data a času, detekce 12hodinového nebo 24hodinového formátu nebo pro integraci vestavěného Safari WebView, které umožňuje zobrazení webových stránek přímo v aplikaci bez nutnosti jejího opuštění, čímž se zlepšuje

uživatelský zážitek. Další důležitou komponentou je `ConnectivityChecker`, který ověřuje, zda je zařízení připojeno k internetu, a je využíván například pro zobrazování informačních hlášek ve snackbaru, když uživatel uloží data offline. Kromě toho zahrnují tyto nástroje i matematické metody, jako je konverze mezi mílemi a kilometry, převod textových hodnot na celá anebo desetinná čísla.

Programovací jazyk Swift umožňuje definovat rozšíření na existujících typech, což velmi zjednoduší a zpřehlední kód. Jedno z rozšíření v rámci této aplikace je například na typu `Font`, kde byly definovány různé velikosti fontů s vlastním stylem a názvy odpovídají jejich účelu. Například `extraLargeTitle` je definován jako `Font.custom("Nunito-Black", fixedSize: 60)`, což umožňuje použít tento font v kódu jednoduše jako `.font(.extraLargeTitle)`. Podobným způsobem byly definovány i barvy, které odkazují na hodnoty z asset katalogu, například `static let primary = Color("VulturePrimary")`. Stejný princip byl aplikován i na ikony, které jsou také referencovány z asset katalogu, což umožňuje jejich snadné použití, například `Image.Vulture.lowBattery`. Za zmínku stojí také rozšíření pro `View`, které obsahuje metodu pro dynamické ohraničení obsahu s adekvátním odsazením a je přizpůsobené velikosti zařízení. Tato metoda automaticky analyzuje rozměry obrazovky a nastavuje maximální šířku obsahu. U iPadů je maximální šířka nastavena na 500 bodů, což výrazně zjednodušilo návrh a implementaci aplikace, protože větší dostupná plocha obrazovky nebyla pro aplikaci nijak přínosná. U iPhoneů je šířka obsahu omezena na 90 % šířky obrazovky.

5.4.4 Assety a Konstanty

Pro zachování konzistentního vzhledu napříč aplikací byly vytvořeny speciální soubory a struktury obsahující globální konstanty. Tyto konstanty zahrnují například číselné hodnoty, jako je `maxScreenWidth` nebo `cornerRadius`, které se využívají ve více komponentách, aby bylo zajištěno jednotné zaoblení rohů nebo maximální šířka obsahu. Pokud se hodnota těchto konstant změní v rámci designového jazyka aplikace, změna se automaticky promítne na všechna místa, kde jsou tyto konstanty použity. Dále jsou zde také definované klíče pro chybové hlášky, které se následně překládají v rámci jednotlivých lokalizačních souborů podporovaných jazyků. Specifické konstanty pro jednotlivé komponenty jsou však definovány přímo v příslušné třídě, struktuře nebo souboru, díky čemuž odpadá potřeba přepínání do souboru s globálními konstantami. Jak již bylo zmíněno v kapitole o UI

komponentách, mezi tyto globální konstanty patří také ikony, barvy a fonty, které zajišťují jednotný styl aplikace a jsou v souladu s designovým jazykem.

Ačkoli SwiftUI nabízí rozsáhlý katalog ikon, pro tuto aplikaci byly zvoleny ikony z Material Icons od Googlu, které jsou vhodnější a více zapadají do designu aplikace. Tyto ikony byly staženy z oficiálních stránek Google, následně importovány do Asset Catalogu a referencovány prostřednictvím konstant, jak bylo popsáno výše. Stejný přístup byl zvolen také pro font aplikace. Vybraný font Nunito z Google Fonts byl importován do projektu jako celý balík a je v kódu používán prostřednictvím předem definovaných konstant.

V rámci této práce byl několikrát zmíněný také Asset Catalog v Xcode, který slouží k uspořádání a správě prostředků aplikace, jako jsou obrázky, barvy nebo datové soubory. Každý prostředek v katalogu může mít více variant přizpůsobených různým charakteristikám zařízení, jako například platformě, velikosti obrazovky, rozlišení, barevnému módu anebo jazyku. Při načítání prostředku za běhu potom systém automaticky vybere správnou variantu na základě aktuálních vlastností zařízení. [22]

Součástí projektu jsou také lokalizační soubory podporovaných jazyků aplikace, konkrétně angličtiny, češtiny a slovenštiny, přičemž angličtina je nastavena jako výchozí jazyk. Xcode poskytuje přehledné rozhraní, kde lze doplňovat překlady k jednotlivým klíčům, které jsou na pozadí uloženy v JSON souborech. Výhodou je, že Xcode automaticky upozorní na chybějící překlady nebo změny ve výchozích hodnotách. V kódu stačí potom například použít `LocalizedStringKey("calculate")`, a framework automaticky zobrazí správný překlad podle aktuálně nastaveného jazyka aplikace.

5.5 Diagram složek a souborů

Struktura adresářů tohoto projektu je navržena tak, aby byla přehledná a usnadňovala práci s jednotlivými částmi aplikace. Mezi hlavní adresáře patří `Features`, kde jsou uloženy všechny funkce aplikace (v souladu s MVVM architekturou), `Shared`, obsahující sdílené služby, komponenty a utility, a `Resources`, který obsahuje Asset Catalog, lokalizace a fonty. Níže je uveden kompletní diagram struktury projektu.

```
App
└─ Features
    ├── Calculate
    ├── Chargers
    ├── Vehicles
    ├── Settings
    ├── Home
    ├── Register
    ├── Common
    ├── Login
    │   ├── ViewModels
    │   ├── Models
    │   └─ Views
    └─ Welcome
└─ Shared
    ├── Services
    ├── Components
    └─ Utilities
        ├── Extensions
        │   ├── Constants
        │   └─ Types
        ├── ConnectivityChecker
        ├── MathUtils
        ├── SafariView
        └─ DateTimeUtils
└─ Resources
    ├── Assets
    ├── Localization
    │   └─ Localizable
    └─ Fonts
```

6 FUNKCE APLIKACE

V této části práce budou představeny systémové požadavky, případy užití a podrobný popis funkcí aplikace. Nejprve budou popsány funkcionální a nefunkcionální požadavky na aplikaci a potom také modelové případy užití aplikace uživatelem. Na závěr budou rozebrány hlavní funkce aplikace, včetně jejich primárního účelu a příkladů implementace.

6.1 Nefunkcionální požadavky

Tabulka 1 - Nefunkcionální požadavky

ID	Popis požadavku
NR_01	Aplikace by měla být rychlá a plynulá na všech podporovaných zařízeních
NR_02	Aplikace by měla být kompatibilní s mobilními telefony, tablety a počítači s operačními systémy iOS 17, iPadOS 17, macOS Sonoma a novějšími
NR_03	Aplikace by měla fungovat jak v online, tak i offline režimu
NR_04	Aplikace by měla mít minimální spotřebu energie pro delší výdrž baterie
NR_05	Aplikace by měla být zabezpečená a chránit citlivé informace a uživatelská data
NR_06	Aplikace by měla podporovat více jazykových mutací pro lepší přístupnost uživatelů
NR_07	Aplikace by měla mít přehledné a intuitivní uživatelské rozhraní
NR_08	Aplikace by měla mít tmavý a světlý barevný motiv
NR_09	Aplikace by měla dodržovat standardy přístupnosti pro osoby se zdravotním postižením
NR_10	Aplikace by měla poskytnout výpočty s co nejmenší odchylkou, aby výsledky odpovídaly reálnému světu
NR_11	Aplikace by měla synchronizovat uživatelská data na různých zařízeních

(zdroj: vlastní)

6.2 Funkcionální požadavky

Tato kapitola obsahuje funkcionální požadavky na aplikaci. Požadavky jsou rozděleny podle oblastí na správu uživatelských účtů, nastavení aplikace, domovskou stránku, práci s nabíječkami a vozidly a také na samotné výpočty.

Tabulka 2 - Funkcionální požadavky na uživatelské účty

ID	Popis požadavku
FR_01	Uživatel si může v aplikaci vytvořit účet pomocí e-mailu a hesla a nebo přes SSO (Google, Apple)
FR_02	Uživatel se může přihlásit do aplikace pomocí e-mailu a hesla a nebo přes SSO (Google, Apple)
FR_03	Uživatel si může obnovit zapomenuté heslo, pokud je registrován pomocí kombinace e-mailu a hesla

(zdroj: vlastní)

Tabulka 3 - Funkcionální požadavky na nastavení

ID	Popis požadavku
FR_04	Uživatel může změnit barevný motiv aplikace mezi světlým a tmavým režimem
FR_05	Uživatel může změnit jazyk aplikace na český, anglický nebo slovenský
FR_06	Uživatel může přepnout mezi metrickými a imperiálními jednotkami
FR_07	Uživatel může poslat zpětnou vazbu na aplikaci vývojáři
FR_08	Uživatel si může změnit heslo, pokud je přihlášen pomocí kombinace e-mailu a hesla
FR_09	Uživatel si může přečíst informace o aplikaci a ohodnotit ji na App Store
FR_10	Uživatel se může odhlásit z aplikace

(zdroj: vlastní)

Tabulka 4 - Funkcionální požadavky na domovskou obrazovku

ID	Popis požadavku
FR_11	Uživatel si může číst tipy k nabíjení, baterii a elektromobilu
FR_12	Uživatel si může zobrazit informace o dopadu na životní prostředí
FR_13	Uživatel se může přes domovskou obrazovku odhlásit z aplikace
FR_14	Uživatel může přes domovskou obrazovku změnit barevný motiv aplikace
FR_15	Uživatel se může přes domovskou obrazovku dostat na další funkce aplikace

(zdroj: vlastní)

Tabulka 5 - Funkcionální požadavky na nabíječky

ID	Popis požadavku
FR_16	Uživatel může přidávat informace o nabíječkách (název, rychlost nabíjení, průměrná cena za kWh, měna, lokaci nabíječky)
FR_17	Uživatel může spravovat seznam nabíječek včetně zobrazení detailu, mazání a úprav informací
FR_18	Uživatel může vyhledávat v seznamu nabíječek podle názvu
FR_19	Uživatel může nabíječky označit jako oblíbené

(zdroj: vlastní)

Tabulka 6 - Funkcionální požadavky na vozidla

ID	Popis požadavku
FR_20	Uživatel může přidávat informace o vozidlech (název, kapacita baterie, průměrná kombinovaná spotřeba)
FR_21	Uživatel může spravovat seznam vozidel včetně zobrazení detailu, mazání a úprav informací
FR_22	Uživatel může vyhledávat v seznamu vozidel podle názvu
FR_23	Uživatel může vozidla označit jako oblíbená
FR_24	Uživatel si může zobrazit nápovědu k zadávaným údajům o vozidle

(zdroj: vlastní)

Tabulka 7 - Funkcionální požadavky na výpočty

ID	Popis požadavku
FR_25	Uživatel může vybrat vozidlo a nabíječku, u kterých chce vypočítat informace o nabíjení
FR_26	Uživatel může zadat aktuální a konečný stav nabití baterie
FR_27	Uživatel může zadat koncový nebo počáteční čas nabíjení
FR_28	Uživatel si může zobrazit výsledky výpočtů pro zvolené vozidlo a nabíječku (dopad na životní prostředí, zvolené vozidlo a nabíječka, detaily o baterii, čas nabíjení, cena za nabíjení, informace o dojezdu, graf rychlosti nabíjení vzhledem ke stavu nabití)

FR_29	Uživatel může sdílet výsledky výpočtů pomocí nativního dialogu v rámci vygenerovaného PDF reportu
-------	---

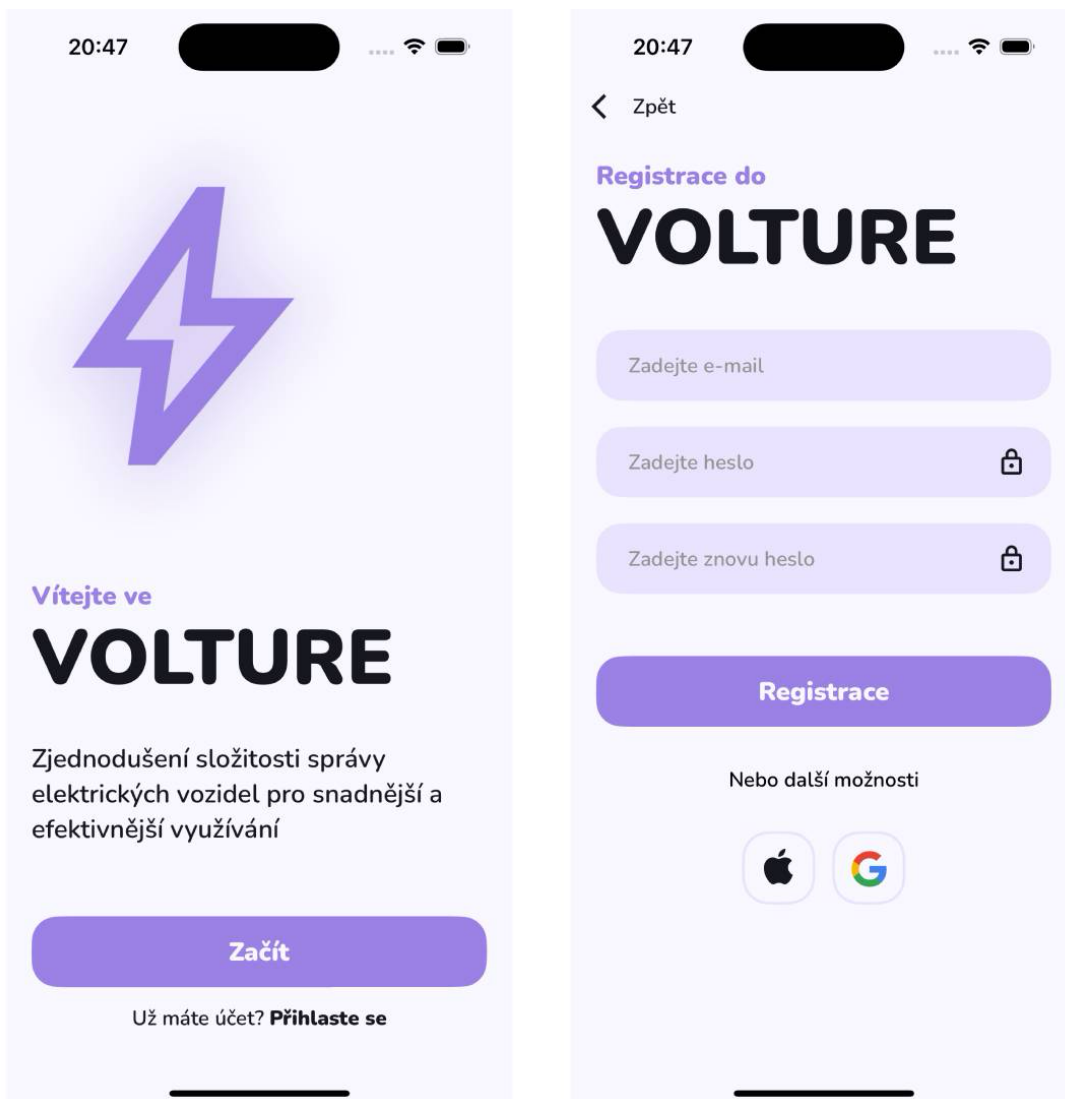
(zdroj: vlastní)

6.3 Popis jednotlivých funkcí

Tato kapitola se bude věnovat popisu funkcionality jednotlivých hlavních částí a funkcí aplikace. Bude popsáno, jak tyto funkce fungují, jaký je jejich účel a u některých budou uvedeny i implementační detaily.

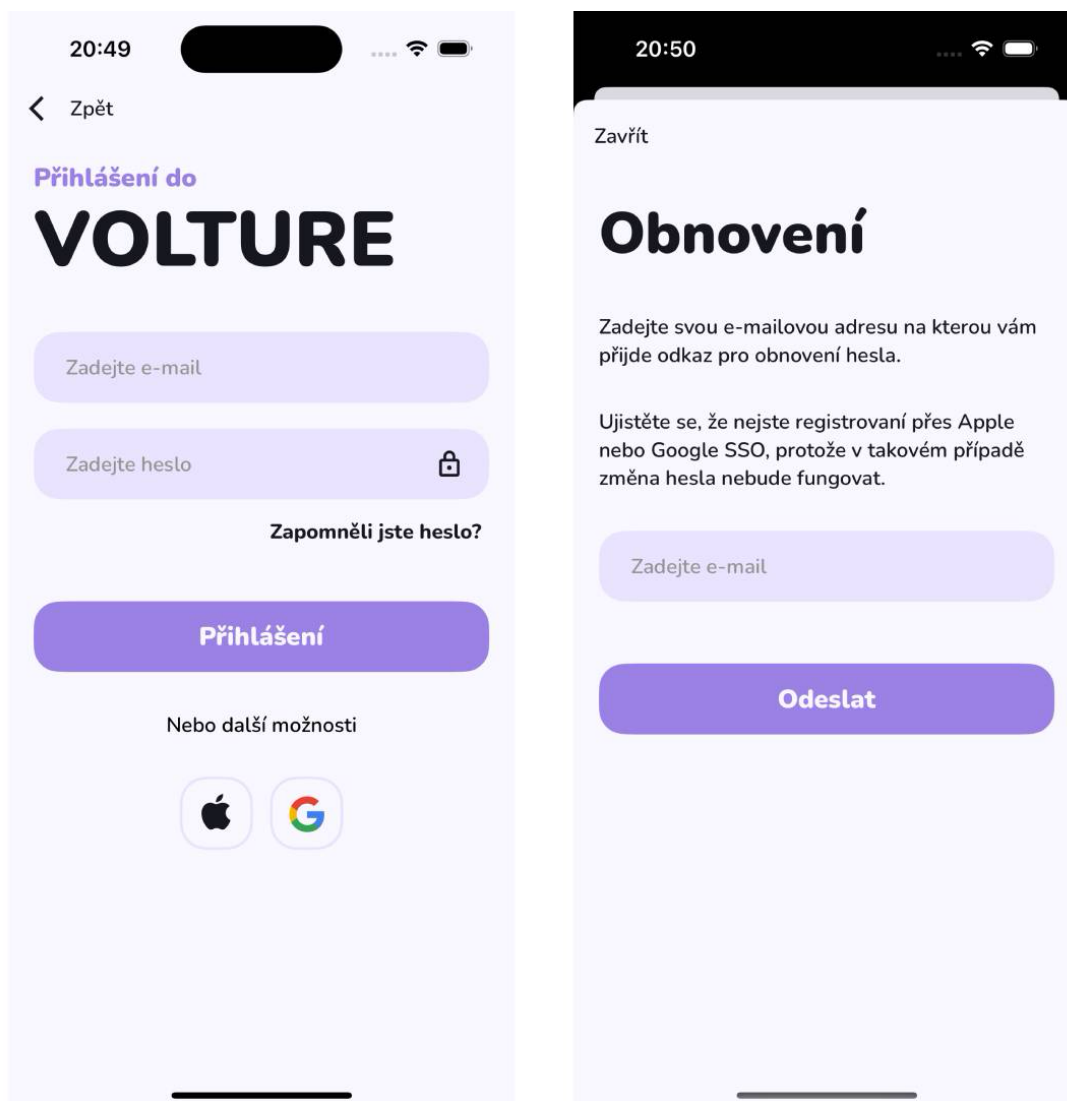
6.3.1 Uživatelské účty

Uživatelé aplikace si před jejím používáním musí vytvořit uživatelský účet, ke kterému budou potom připojena jejich data, což umožňuje synchronizaci mezi různými zařízeními a zajišťuje plynulý uživatelský zážitek. Při prvním spuštění aplikace se zobrazí uvítací obrazovka, která slouží jako rozcestník – uživatele následně přesměruje buď na stránku registrace, nebo na přihlášení, pokud již účet v aplikaci mají (Obrázek 10 - Uvítací obrazovka a registrace a Obrázek 11 - Přihlášení a obnova zapomenutého hesla). Aplikace podporuje přihlášení pomocí kombinace e-mailu a hesla, ale také pomocí Apple a Google SSO, což je dnes standardem ve většině moderních aplikací. Při implementaci SSO na Apple platformách je důležité zajistit podporu také pro Apple SSO, pokud aplikace nabízí jiné způsoby přihlášení třetích stran (např.: Google, Microsoft nebo GitHub). Bez této podpory by aplikace nemohla projít procesem schválení pro distribuci přes App Store. Pro uživatele registrované pomocí e-mailu a hesla, kteří zapomenou své heslo, je zde dostupná také funkce jeho obnovení (Obrázek 11 - Přihlášení a obnova zapomenutého hesla). Tato funkce zašle na zadaný e-mail odkaz, pomocí kterého si uživatel může nastavit heslo nové. Vzhled, zpráva anebo také jazykové mutace e-mailu pro obnovení hesla se dají konfigurovat v online Firebase konzoli.



Obrázek 10 - Uvítací obrazovka a registrace

(zdroj: vlastní)

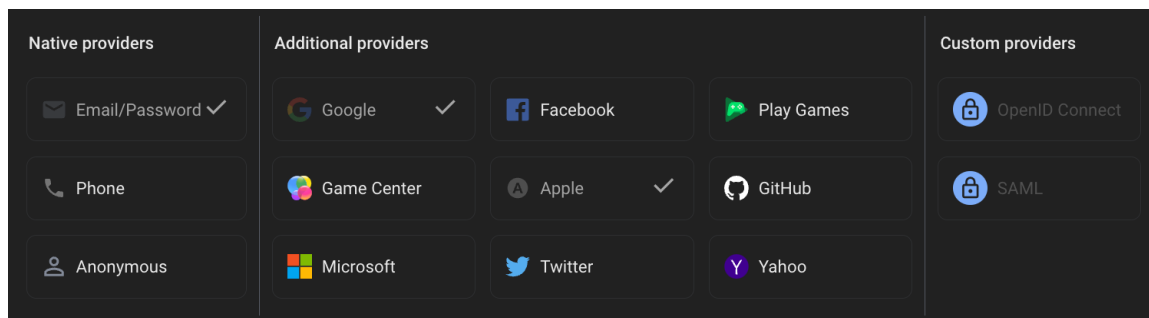


Obrázek 11 - Přihlášení a obnova zapomenutého hesla

(zdroj: vlastní)

Správa uživatelských účtů je řešena prostřednictvím Firebase – k uživatelským datům lze přistupovat pomocí online konzole Firebase, která umožňuje provádět administrativní operace, jako je mazání účtů, blokování uživatelů nebo manuální resetování hesel. Je však nutné zdůraznit, že k této konzoli mají přístup pouze vývojáři nebo administrátoři aplikace – uživatelé aplikace k ní přístup nemají. Pro zajištění maximální bezpečnosti jsou v konzoli dostupné pouze základní informace, jako je vygenerované ID uživatele, poskytovatel přihlášení, data první registrace anebo posledního přihlášení a e-mail. Žádné další citlivé údaje zde nejsou viditelné. Konzole také slouží k nastavení podporovaných metod přihlášení, respektive poskytovatelů, jako jsou e-mail a heslo, Apple, Google a další. Po dokončení konfigurace všech používaných služeb ve Firebase konzoli se vygeneruje konfigurační JSON soubor, který obsahuje všechny potřebné informace pro komunikaci

mobilní aplikace s Firebase službami. Tento soubor je poté integrován do projektu ve SwiftUI a referencován při spuštění aplikace, konkrétně v metodě pro inicializaci Firebase služeb. Tato metoda, stejně jako ostatní potřebné funkce, je součástí knihovny Firebase SDK, konkrétně modulu FirebaseAuth. Níže na snímcích obrazovky lze vidět ukázkou nastavení poskytovatelů přihlášení (Obrázek 12 - Poskytovatelé přihlášení) ve Firebase konzoli a záznam s údaji o uživateli aplikace (Obrázek 13 - Tabulka s uživateli).



Obrázek 12 - Poskytovatelé přihlášení

(zdroj: vlastní)

Search by email address, phone number, or user UID					Add user	⌂	⋮
Identifier	Providers	Created ↑	Signed In	User UID			
hwbcd4xy6b@privatere...		Jul 9, 2024	Jan 15, 2025	T6dgHyQ9IZctuntYMIurTE18B...			

Obrázek 13 - Tabulka s uživateli

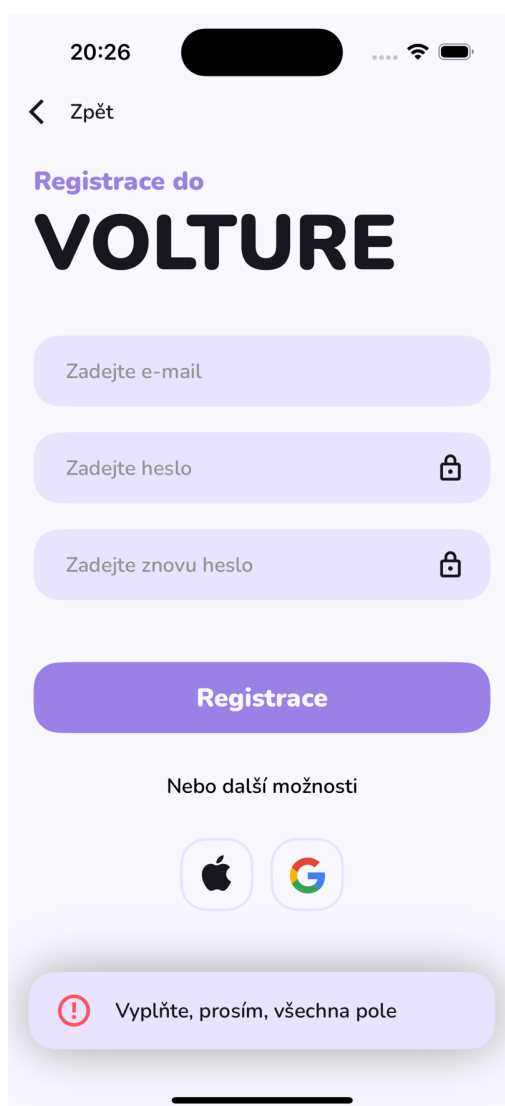
(zdroj: vlastní)

Aplikace využívá Firebase SDK pro jednoduchou a efektivní komunikaci s Firebase službami. Tato knihovna, potažmo jednotlivé moduly, obsahuje implementované metody, které přímo komunikují s backendem hostovaným na serverech Googlu. Na příkladu níže je potom zobrazen proces registrace uživatele pomocí e-mailu a hesla.

```
import Firebase

func registerWithEmail(email: String, password: String, completion:
    @escaping (Error?) -> Void) {
    Auth.auth().createUser(withEmail: email, password: password) {
        authResult, error in
            completion(error)
    }
}
```

Před samotným voláním této metody probíhá vlastní validace dat na straně aplikace. Firebase však vrací další chybové hlášky, například pokud uživatel s daným e-mailem již existuje nebo pokud zadané heslo nesplňuje požadavky (je příliš slabé nebo krátké). Všechny tyto chybové kódy, které Firebase vrací, jsou v aplikaci přemapovány na uživatelsky přívětivé zprávy a následně jsou zobrazeny ve snackbaru na spodní části obrazovky, jak je vidět na snímku obrazovky níže (Obrázek 14 - Snackbar a validace). Metoda `createUser` má kromě hlavních parametrů (e-mail a heslo) také tzv. completion handler – jedná se o callback funkci, která se zavolá po dokončení operace, v tomto případě po úspěšném, nebo neúspěšném vytvoření uživatele. Completion handler má také volitelný parametr `Error?`, který obsahuje informace o případné chybě, například o neúspěšné validaci.



Obrázek 14 - Snackbar a validace

(zdroj: vlastní)

6.3.2 Nastavení

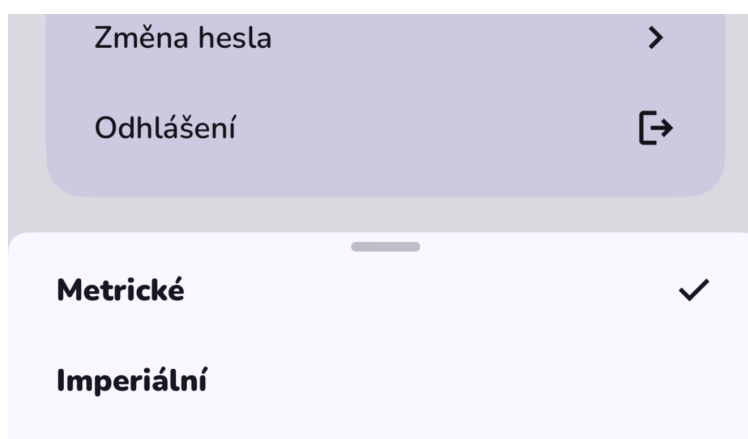
Uživatelé aplikace si mohou přizpůsobit její nastavení podle svých preferencí a potřeb (Obrázek 15 - Nastavení aplikace). Mezi dostupné možnosti patří výběr barevného režimu (světlý nebo tmavý), nastavení jazyka aplikace (angličtina, čeština, slovenština) a volba soustavy jednotek (metrická nebo imperiální). Pokud je uživatel přihlášen pomocí kombinace e-mailu a hesla, má navíc možnost změnit heslo ke svému účtu přímo v nastavení. Hodnoty nastavení jsou ukládány lokálně v paměti zařízení, nejsou však synchronizovány na úrovni uživatelského účtu. To znamená, že nastavení zůstane zachováno po zavření a znovuotevření aplikace, nicméně při přechodu na jiné zařízení bude nastavení obnoveno na výchozí hodnoty (metrické jednotky a tmavý režim).



Obrázek 15 - Nastavení aplikace

(zdroj: vlastní)

Možnosti pro výběr barevného motivu a soustavy jednotek jsou zobrazeny na modálním listu. V tomto seznamu je aktuálně vybraná varianta vizuálně odlišena ikonou zaškrtnutí. Na snímku obrazovky níže (Obrázek 16 - Modal sheet s výběrem jednotek) lze vidět příklad tohoto seznamu s dostupnými variantami. Nastavení jazyka aplikace však funguje odlišně – po kliknutí na tuto variantu zařízení uživatele přesměruje do systémového nastavení, kde si může vybrat preferovaný jazyk. Nevýhodou tohoto přístupu je nutnost aplikaci na chvíli opustit a také to, že se po návratu aplikace znovu načte, aby mohla aplikovat nové textové řetězce zvoleného jazyka. Tento přístup však zachovává konzistenci a umožňuje uživatelům toto nastavení jednoduše najít. Po instalaci aplikace automaticky nastaví jazyk podle aktuální lokalizace zařízení. Pokud je zařízení nastaveno například na češtinu, aplikace se automaticky zobrazí v tomto jazyce. Uživatel však má možnost toto nastavení manuálně přepsat. V případě, že zařízení používá jazyk, který aplikace nepodporuje, je jako výchozí jazyk nastavena angličtina.



Obrázek 16 - Modal sheet s výběrem jednotek

(zdroj: vlastní)

V nastavení aplikace se dále nachází několik důležitých funkcí a informací – uživatelé zde najdou kontaktní formulář, který slouží ke sdílení zpětné vazby, například pro nahlášení chyby nebo pro návrh nové funkce. Veškerá zpětná vazba je uložena přímo v databázi na Firebase, kde k ní má přístup pouze admin aplikace. Součástí nastavení je také sekce „O aplikaci“, která obsahuje stručný popis hlavní mise a vize projektu. Na této stránce se nachází i tlačítko pro hodnocení aplikace na App Store a informace o aktuální verzi aplikace. Poslední položkou v nastavení je možnost odhlášení uživatele, které po kliknutí uživatelé odhlásí a přesměruje na uvítací obrazovku.

6.3.3 Domovská obrazovka

Domovská obrazovka aplikace slouží jako hlavní rozcestník, který uživatelům umožňuje přístup ke všem hlavním funkcím aplikace. Po spuštění aplikace je uživatel automaticky přesměrován na domovskou obrazovku, pokud je přihlášen. Když Firebase knihovna zjistí, že uživatel není přihlášen, aplikace ho přesměruje na uvítací obrazovku. Na domovské obrazovce se pod úvodním textem a animovaným logem nachází sekce s tipy zaměřenými na baterie, elektromobily a nabíjení obecně. Tyto tipy jsou primárně určené a užitečné pro úplné začátečníky, kteří se s elektromobilitou teprve seznamují, a mohou jim pomoci lépe porozumět této oblasti. Aplikace obsahuje více než 50 tipů, které se zobrazují náhodně – po kliknutí na aktuální tip se automaticky načte nový. Stejně tak se stane i při každém novém načtení aplikace. Hlavní funkce aplikace jsou na domovské obrazovce reprezentovány dlaždicemi, které jsou rozděleny na dva typy – textové dlaždice a dlaždice s ikonami. Celkově jsou jednotlivé funkce aplikace uspořádány do tří hlavních sekcí – výpočty pro nabíjení, správa nabíjení a správa účtu. Kompletní vzhled domovské obrazovky je zobrazen na snímku obrazovky níže (Obrázek 17 - Domovská obrazovka).



Obrázek 17 - Domovská obrazovka

(zdroj: vlastní)

V první sekci „výpočty pro nabíjení“, se nachází primární textová dlaždice, která uživatele přesměruje na hlavní funkci výpočtů. Tato funkce bude detailně popsána v následující kapitole. Další funkcí v této sekci je zobrazení informací o emisích CO₂ na kWh pro jednotlivé země (Obrázek 18 - CO₂ na kWh pro dané oblasti). Tato funkce umožňuje uživatelům zjistit, kolik gramů CO₂ emisí na kWh produkuje jejich země, případně určitá oblast. Pokud nejsou k dispozici data pro konkrétní zemi, jsou k dispozici průměrné hodnoty pro EU nebo svět. Data pro tuto funkci jsou čerpána ze zdroje <https://www.nowtricity.com> a jsou uložena v databázi na Firebase. Díky tomu lze zajistit automatickou aktualizaci, jakmile budou k dispozici novější údaje. V současnosti jsou dostupná data za rok 2023 pro většinu zemí.

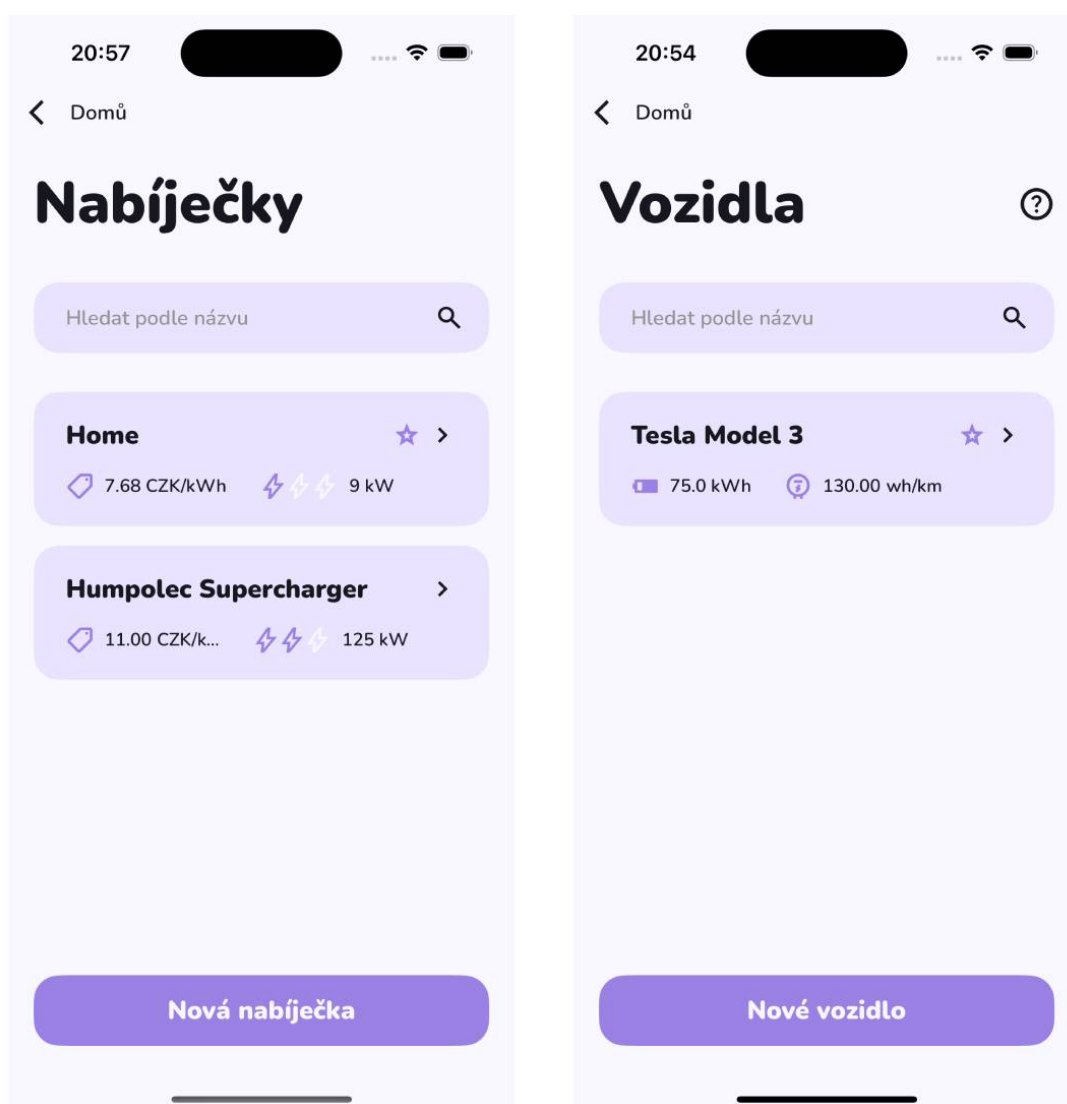
Obrázek 18 - CO₂ na kWh pro dané oblasti*(zdroj: vlastní)*

Další sekce, správa nabíjení, obsahuje dvě dlaždice s ikonami, které umožňují uživatelům přístup ke správě vozidel nebo nabíječek. Detailní popis těchto funkcí bude v následujících kapitolách. Poslední sekce, správa účtu, nabízí rychlé akce, jako je změna barevného režimu aplikace (světlý nebo tmavý) a možnost odhlášení uživatele z aplikace. Kromě toho obsahuje textovou dlaždici, která slouží k otevření nastavení, kde si uživatelé mohou upravit aplikaci podle svých preferencí.

6.3.4 Správa vozidel a nabíječek

Jednou z velmi důležitých funkcí aplikace je správa flotily vozidel a nabíječek uživatele (Obrázek 19 - Nabíječky a vozidla). Ačkoliv jsou tyto dvě funkce v aplikaci oddělené, jsou zde popsány společně, protože jejich implementace je velmi podobná. Přístup k těmto

funkcím je možný přímo z domovské obrazovky prostřednictvím dlaždic s ikonami. Každá z těchto funkcí ve výchozím stavu zobrazuje uživateli seznam položek, který je standardně seřazen abecedně vzestupně (od A do Z). Pokud však uživatel označí některá vozidla nebo nabíječky jako oblíbené, budou tyto položky v seznamu zobrazeny na začátku. To platí i v případě, že nesplňují podmínku abecedního řazení. Je to primárně kvůli tomu, aby byly oblíbené položky rychle na dosah, například i u výběru dat pro výpočty. Součástí seznamu vozidel a nabíječek je také možnost vyhledávání podle názvu, které probíhá lokálně na zařízení, kde jsou názvy položek i text zadaný uživatelem převedeny na malá písmena, aby bylo porovnání nezávislé na velikosti písmen.



Obrázek 19 - Nabíječky a vozidla

(zdroj: vlastní)

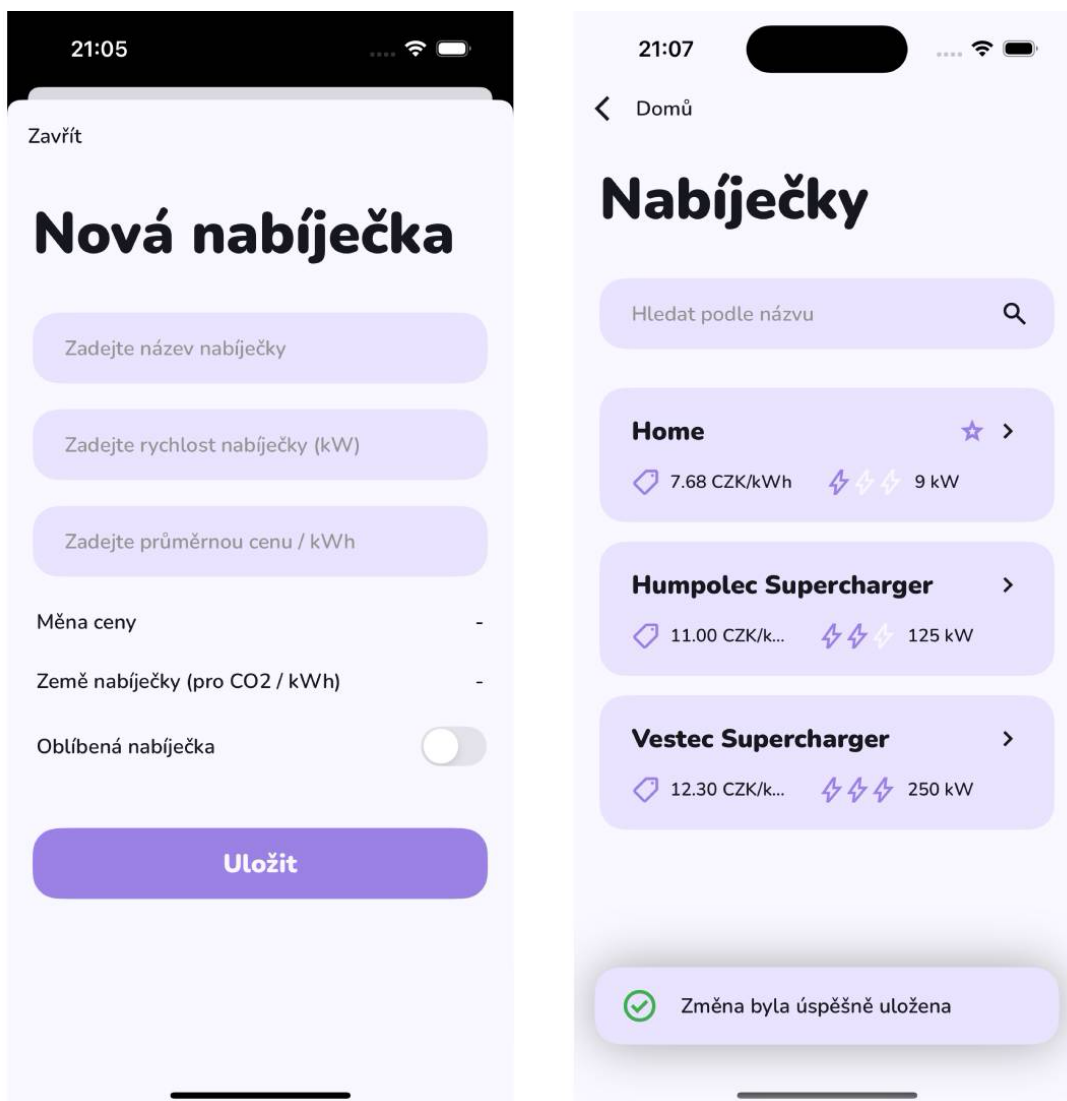
Uživatelé si mohou do aplikace přidat svá vozidla nebo nabíječky, které následně využijí pro výpočty spojené s plánováním nabíjení. K tomuto účelu slouží formulář se vstupními daty, která je nutné vyplnit. Při přidávání nového vozidla uživatel zadává jeho název, kapacitu baterie v kWh, průměrnou kombinovanou spotřebu v Wh na jednotku vzdálenosti (míle nebo kilometr podle preferencí) a může také označit vozidlo jako oblíbené. Všechna pole, kromě označení oblíbenosti, jsou povinná, a pokud se některé z nich nevyplní, aplikace při ukládání zobrazí snackbar s validační chybou informující uživatele o nutnosti vyplnit všechna povinná pole. Na snímku obrazovky níže je zobrazen jak formulář, tak validační hláška v snackbaru. V samotném seznamu vozidel je k dispozici také nápověda pro uživatele, kteří si nejsou jistí některými údaji (Obrázek 20 - Formulář pro vozidla s validační chybou a nápověda). Nápověda obsahuje odkaz na stránku EV Database, kde lze potřebné informace dohledat. Automatizace tohoto procesu bohužel nebyla možná, protože dostupná API s údaji o elektrických vozidlech jsou zpoplatněná a jejich financování přesahuje rámec této práce, proto je nutné údaje zadávat manuálně.

The image displays two mobile application screens side-by-side. The left screen, titled 'Nové vozidlo' (New vehicle), features a form with three input fields: 'Zadejte název vozidla' (Enter vehicle name), 'Zadejte kapacitu baterie (kWh)' (Enter battery capacity (kWh)), and 'Zadejte průměrnou spotřebu (Wh/km)' (Enter average consumption (Wh/km)). Below these is a toggle switch for 'Oblíbené vozidlo' (Favorite vehicle) and a large purple 'Uložit' (Save) button. A red error message at the bottom states 'Vyplňte, prosím, všechna pole' (Please fill in all fields). The right screen, titled 'Nápověda' (Help), contains text explaining that manual data entry is required and points to the 'Efficiency' and 'Usable Battery' sections of the 'EV Database Web'.

Obrázek 20 - Formulář pro vozidla s validační chybou a nápověda

(zdroj: vlastní)

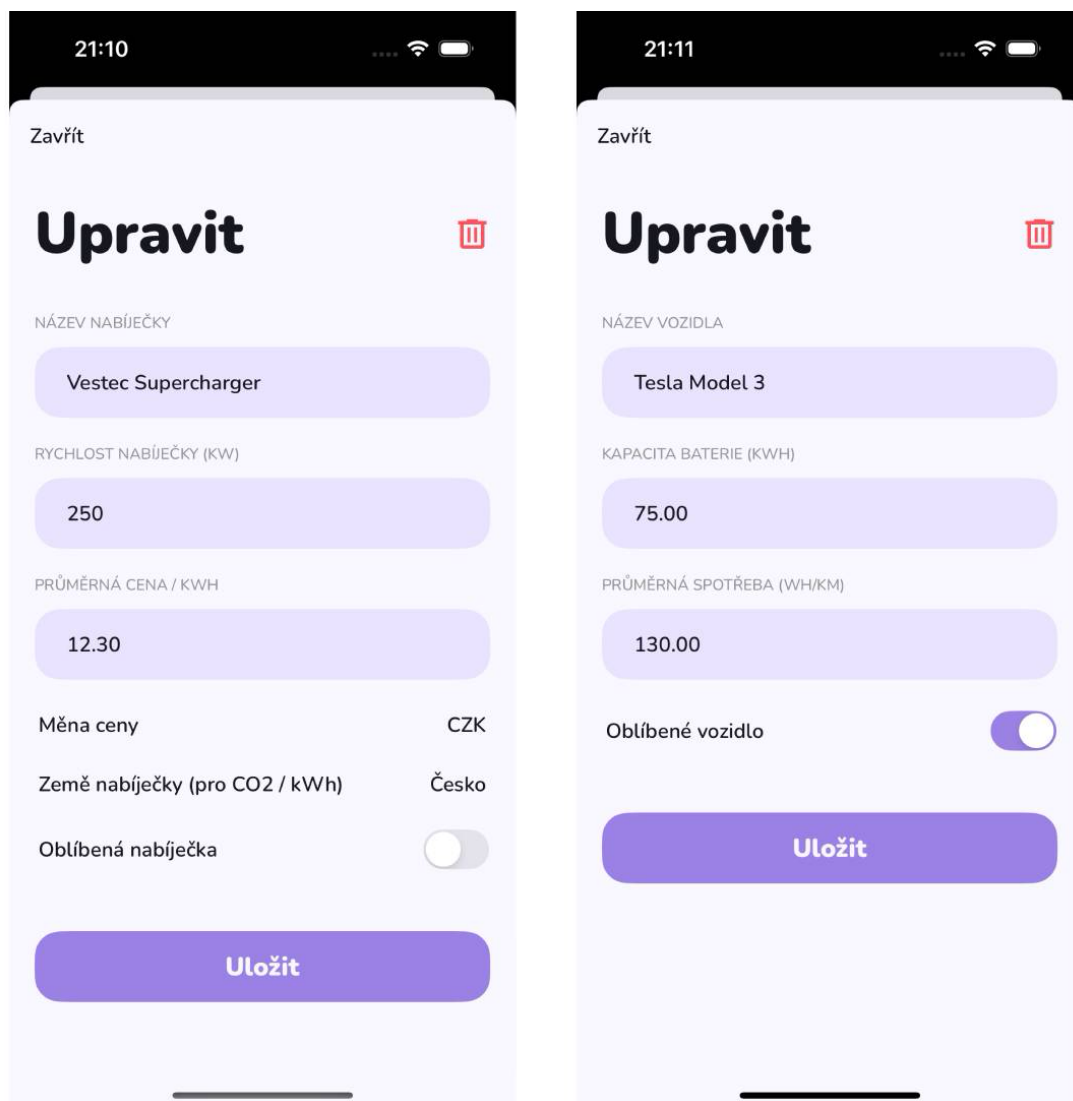
Podobně jako u vozidel, i při přidávání nové nabíječky se používá formulář se vstupními daty, která jsou dále validována. Mezi povinná pole patří název nabíječky, výkon v kW, průměrná cena za kWh, měna ceny a země, kde se nabíječka nachází pro následné výpočty dopadu na životní prostředí. Uživatel má i zde možnost označit nabíječku jako oblíbenou. Pokud některé z povinných polí zůstane nevyplněné, aplikace při ukládání zobrazí také snackbar s validační chybou. Po úspěšném uložení je uživatel přesměrován na seznam nabíječek a informován o dokončení operace pomocí snackbaru, jak je zobrazeno na snímku obrazovky níže (Obrázek 21 - Formulář pro nabíječky a snackbar po úspěšném uložení). Stejný postup a logika platí i pro přidávání nového vozidla.



Obrázek 21 - Formulář pro nabíječky a snackbar po úspěšném uložení

(zdroj: vlastní)

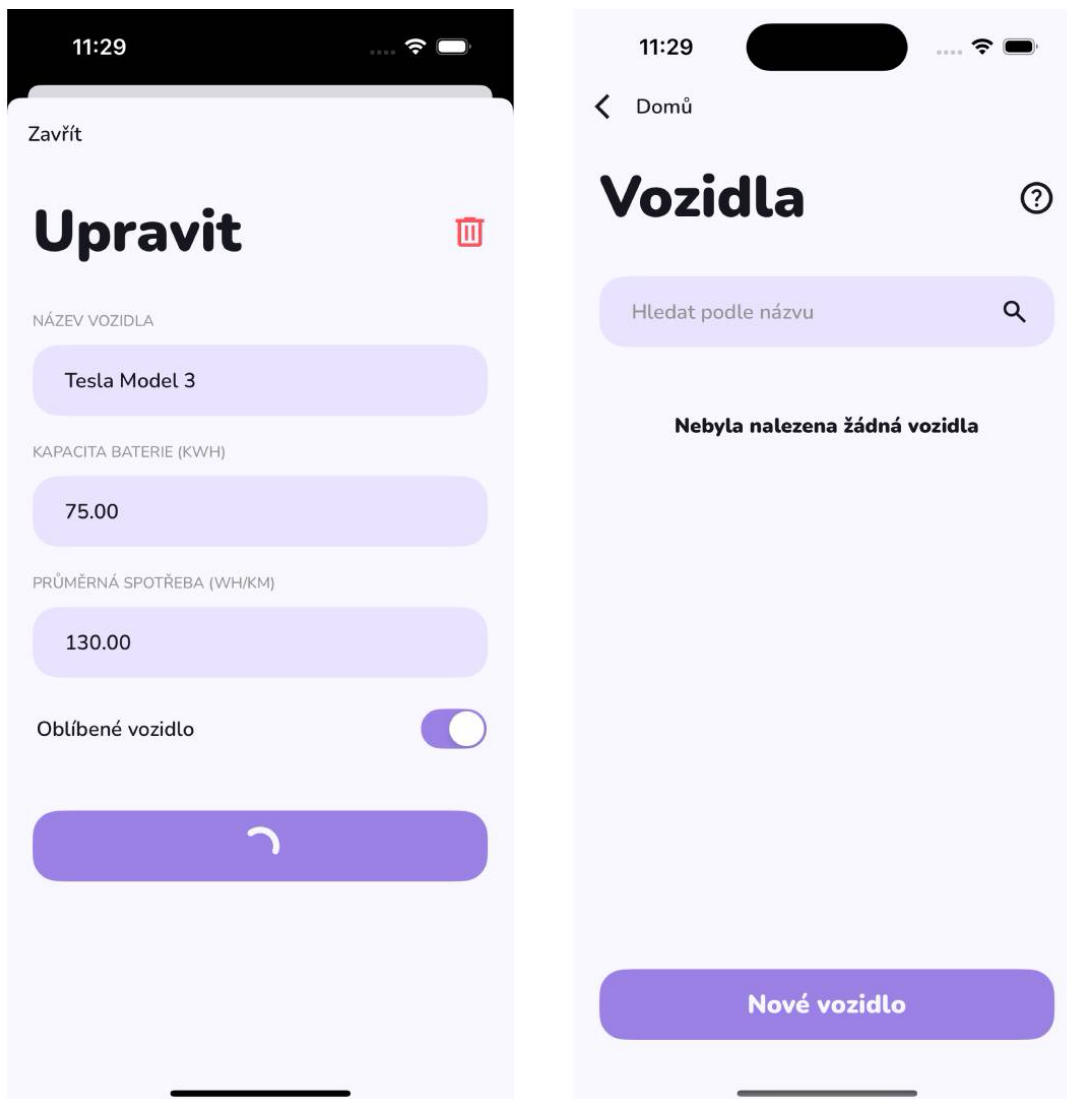
Uživatelé mají možnost upravit nebo smazat svá data, například pokud dojde ke změně údajů nebo vozidlo či nabíječku již nevyužívají. Po kliknutí na položku v seznamu se otevře stejný formulář jako při přidávání nové položky, nicméně v tomto případě jsou vstupní pole předvyplněna aktuálními hodnotami a doplněna popisy, aby bylo jasné, co jednotlivé údaje znamenají. V horní části obrazovky se nachází ikona koše, která umožňuje položku smazat, kdy po smazání je uživatel přesměrován zpět na seznam a informován o úspěšné akci snackbarem. Stejný proces platí i při úpravě hodnot – po kliknutí na tlačítko pro uložení aktualizovaných údajů je položka upravena, uživatel přesměrován zpět na seznam a zobrazen snackbar potvrzující dokončení operace. Obrazovky pro úpravu dat jsou zobrazeny níže (Obrázek 22 - Úprava nabíječky a vozidla).



Obrázek 22 - Úprava nabíječky a vozidla

(zdroj: vlastní)

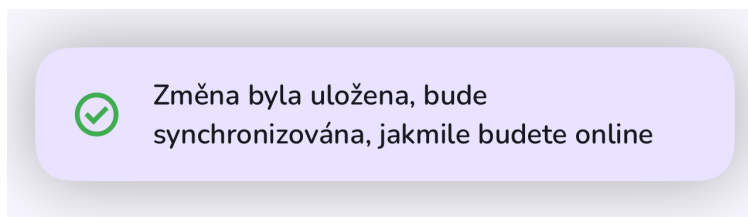
Pokud operace mazání nebo ukládání trvá delší dobu, tlačítko místo textu zobrazuje animovanou komponentu „loading spinner“, která indikuje probíhající operaci. V tomto stavu je tlačítko zároveň neaktivní, aby uživatel nemohl operaci opakovaně spustit. Stejný indikátor se zobrazuje i během načítání položek ze serveru při čekání na odpověď z Firebase. Pokud v seznamu nejsou žádné položky, například po prvním spuštění a přihlášení do aplikace, místo seznamu se zobrazí informační zpráva, která uživatele o této situaci informuje. Na snímcích obrazovky níže je zobrazena komponenta „loading spinner“ i příklad prázdného seznamu s informační hláškou (Obrázek 23 - Loading spinner a prázdný seznam vozidel).



Obrázek 23 - Loading spinner a prázdný seznam vozidel

(zdroj: vlastní)

Aplikace umožňuje uživatelům pracovat i v offline režimu díky Firebase knihovně, která ukládá data do mezipaměti, pokud zařízení zrovna nemá přístup k internetu. Pokud uživatel uloží, upraví nebo smaže data v offline režimu, zobrazí se mu v snackbaru jiná hláška než při online operaci, jak lze vidět na obrázku níže (Obrázek 24 - Snackbar v offline režimu). Po opětovném připojení k internetu aplikace automaticky synchronizuje data s Firebase při další interakci, takže uživatel o žádná data nepřijde. Při offline přístupu k seznamu položek, jako jsou vozidla nebo nabíječky, se zobrazí poslední uložená verze dat z lokálního úložiště a po připojení k internetu se data případně aktualizují. Pro zjišťování dostupnosti internetového připojení byla vytvořena speciální třída `ConnectivityChecker`, která monitoruje stav připojení a na základě toho zobrazuje adekvátní hlášky v snackbaru, aby byl uživatel správně informován o aktuálním stavu aplikace.



Obrázek 24 - Snackbar v offline režimu

(zdroj: vlastní)

Na ukázce kódu níže je potom metoda pro ukládání vozidel do Firestore – tato servisní metoda se volá až po validaci dat vozidla v příslušném ViewModelu a obsahuje další kontrolu, zda je přítomné ID uživatele, což je velmi důležitý parametr pro správné uložení dat. V rámci metody se poté sestaví objekt vozidla a vytvoří reference na kolekci, kam se data ve Firestore ukládají. Metoda je univerzální a zajišťuje jak ukládání nových vozidel, tak aktualizaci existujících – rozhodnutí probíhá na základě přítomnosti ID vozidla v datech. Pokud ID není přítomné, vytvoří se nový záznam, a pokud je, dojde k aktualizaci existujícího. Po dokončení operace se volá callback funkce, která obsahuje volitelný parametr `Error`, který pokud není `nil`, zobrazí snackbar s adekvátní chybovou hláškou. V poslední části metody probíhá kontrola dostupnosti internetového připojení – pokud zařízení není online, zobrazí se uživateli informační hláška o offline uložení s informací o pozdější synchronizaci. Poté se ukončí metoda, aby nedošlo k blokování uživatelského rozhraní.

```
func saveVehicle(vehicle: VehicleModel, userId: String?, isOnline: Bool, completion: @escaping (Error?) -> Void) {  
    guard let userId = userId else {  
        return completion(NSError(domain: Constants.ErrorCode.generalError, code: -1))  
    }  
  
    let vehicleData: [String: Any] = [  
        "name": vehicle.name,  
        "isFavourite": vehicle.isFavourite,  
        "batteryCapacityInKwh": vehicle.batteryCapacityInKwh,  
        "averageConsumptionInWhPerKm":  
        vehicle.averageConsumptionInWhPerKm,  
        "averageConsumptionInWhPerMi":  
        vehicle.averageConsumptionInWhPerMi  
    ]  
}
```

```

    let vehiclesCollection =
      db.collection("users").document(userId).collection("vehicles")
    if let id = vehicle.id {
      vehiclesCollection.document(id).updateData(vehicleData) {
        error in
          return completion(error)
      }
    } else {
      vehiclesCollection.addDocument(data: vehicleData) { error in
        return completion(error)
      }
    }
  }

  if !isOnline {
    DispatchQueue.main.asyncAfter(deadline: .now() + 1.0) {
      return completion(nil)
    }
  }
}

```

Pro zajištění bezpečnosti dat a ochrany aplikace jsou ve Firestore definována bezpečnostní pravidla, která přesně určují, k jakým datům mají uživatelé přístup. Tato pravidla zajišťují, že každý uživatel může číst, zapisovat, aktualizovat a mazat pouze svá vlastní data, a to na základě jejich jedinečného ID. Příklad pravidel je uveden níže, kde je logika nastavena pro kolekci `users` a její podkolekci `vehicles`. Pravidla specifikují, že přístup je povolen pouze přihlášeným uživatelům (`request.auth != null`) s odpovídajícím UID (`request.auth.uid == userId`). Veškeré ostatní operace mimo definovaná oprávnění jsou explicitně zakázány prostřednictvím pravidla, které nepovoluje žádný přístup k jiným dokumentům v databázi (v rámci tohoto příkladu pravidel není uvedena kolekce nabíječek). Samotná struktura Firestore je navržena tak, aby byla, pokud možno, co nejvíce efektivní a přehledná. Základem je kolekce `users`, která obsahuje dokumenty reprezentující jednotlivé uživatele. ID každého dokumentu odpovídá ID uživatele, což optimalizuje přístup k datům a minimalizuje počet operací s Firestore. Každý uživatelský dokument dále obsahuje dvě podkolekce – `chargers` a `vehicles`, ve kterých jsou uložena příslušná data o nabíječkách a vozidlech uživatele.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, write, update, delete: if request.auth != null &&
      request.auth.uid == userId;
      match /vehicles/{vehicleId} {
        allow read, write, update, delete: if request.auth != null &&
        request.auth.uid == userId;
      }
    }
    match /{document=**} {
      allow read, write, update, delete: if false;
    }
  }
}
```

6.3.5 Výpočty

Hlavní funkcí aplikace jsou výpočty, které uživatelům poskytují užitečné informace o vozidle a nabíjení, díky čemuž usnadňují plánování cestování a každodenního života s elektrickým vozidlem. Přístup k této funkci je opět možný přes domovskou obrazovku pomocí primární textové dlaždice, která otevře formulář s několika sekcemi. První sekce umožňuje uživatelům výběr vozidla a nabíječky, pro které aplikace následně provede výpočty. Používají se zde stejné komponenty ze správy vozidel a nabíječek, ale s tím rozdílem, že po kliknutí na položku se data vrátí jako vstup do výpočtů, místo aby se otevřelo jejich upravení, jako u jejich správy. Druhá sekce je zaměřena na baterii. Zde uživatel zadává aktuální a cílový stav nabití baterie v procentech (SoC). Poslední sekce se týká času nabíjení – uživatel si zde může vybrat čas zahájení nebo ukončení nabíjení. Je zde také možné rychle předvyplnit aktuální čas. Aplikace podporuje jak 12hodinový, tak 24hodinový formát času podle nastavení zařízení. Formulář také zahrnuje validaci vstupů, a pokud některé povinné údaje chybí, zobrazí se snackbar s upozorněním na jejich doplnění. Po kliknutí na tlačítko „Vypočítat“ se uživateli zobrazí výsledky výpočtů. Kompletní formulář pro zadávání vstupů je zobrazen na snímku obrazovky níže (Obrázek 25 - Vstupní formulář pro výpočty).

11:44

< Domů

Vypočítat

VOZIDLO A NABÍJEČKA

Vozidlo Vybrat >

Nabíječka Vybrat >

PODROBNOSTI O BATERII

Zadejte aktuální stav nabití (%)

Zadejte cílový stav nabití (%)

ČAS NABÍJENÍ

Začátek Konec

Čas zahájení nabíjení

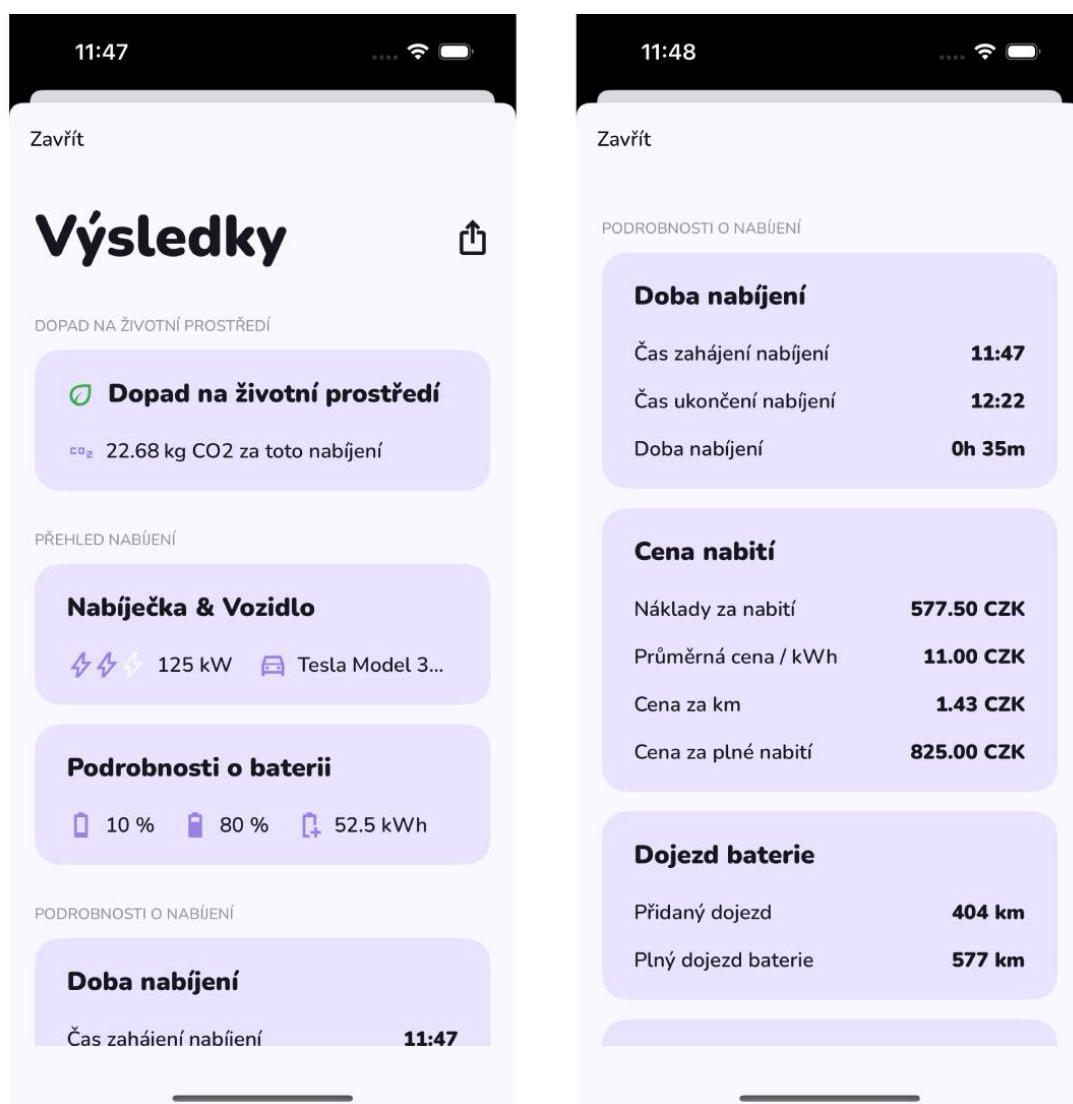
Vypočítat

Obrázek 25 - Vstupní formulář pro výpočty

(zdroj: vlastní)

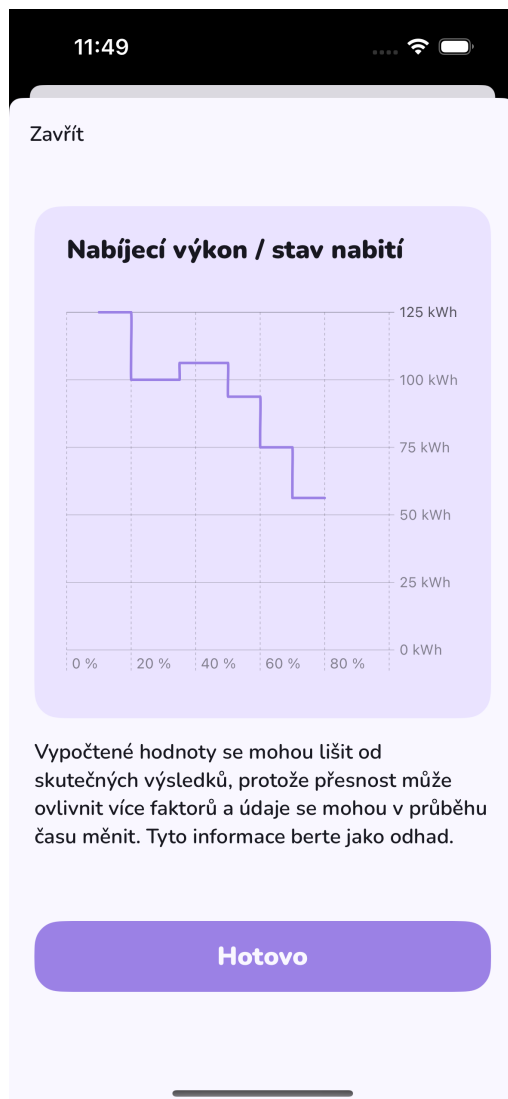
Výsledky výpočtů jsou taktéž rozděleny do několika přehledných sekcí, kde první sekce poskytuje informace o dopadu nabíjení na životní prostředí, konkrétně kolik kilogramů CO₂ bylo během nabíjení vyprodukováno. Tyto údaje se vypočítávají na základě emisních dat země, kde se nabíječka nachází. Další sekce obsahuje shrnutí parametrů, jako je vybrané vozidlo, rychlost nabíječky, procento nabití baterie a přidané množství energie v kWh. Ve třetí sekci jsou uvedeny detaily o čase nabíjení, včetně jeho začátku, konce a celkové doby, která je vypočítána pomocí algoritmu simulujícího realistický průběh nabíjení, který bude více popsán níže v této kapitole. Čtvrtá sekce se zaměřuje na náklady nabíjení, kde uživatel vidí cenu za konkrétní nabití, průměrnou cenu za kWh, cenu za jednotku vzdálenosti (kilometry anebo míle) a odhad nákladů na plné nabití baterie od 0 % do 100 %. Následují informace o přidaném dojezdu na aktuální nabití a o maximálním dojezdu vozidla

při zadané kombinované spotřebě. Pod tím se nachází orientační graf výkonu nabíječky vůči stavu nabití baterie, který ukazuje, jak například u rychlonabíječek výkon s časem postupně klesá, aby se chránila životnost baterie. Na závěr je uvedeno vyloučení odpovědnosti, které upozorňuje, že reálné hodnoty se mohou lišit v závislosti na různých faktorech. Po kliknutí na tlačítko „Hotovo“ je uživatel přesměrován zpět na vstupní formulář, který se vyresetuje na výchozí hodnoty, aby bylo možné ihned zadat nové výpočty. Kompletní přehled vypočtených hodnot lze vidět na snímcích obrazovky níže (Obrázek 26 - Výsledky výpočtů a Obrázek 27 - Výsledky výpočtů – graf).



Obrázek 26 - Výsledky výpočtů

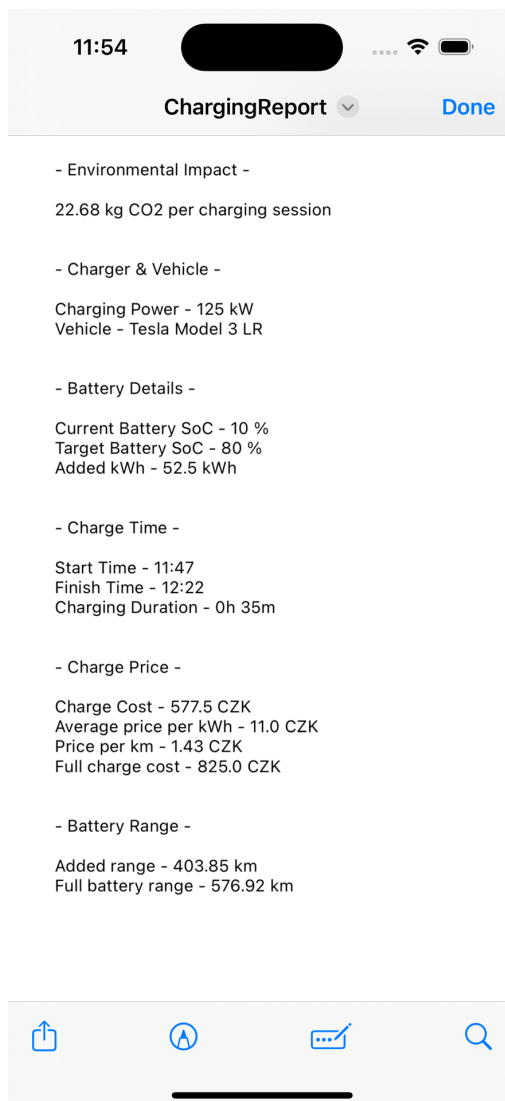
(zdroj: vlastní)



Obrázek 27 - Výsledky výpočtů – graf

(zdroj: vlastní)

Uživatelé mají také možnost sdílet výsledky výpočtů s dalšími osobami pomocí nativního dialogu pro sdílení dostupného v operačních systémech od Apple. Po kliknutí na tlačítko sdílení aplikace automaticky vygeneruje PDF dokument, který obsahuje všechny vypočtené hodnoty kromě grafu. Tento PDF soubor mohou uživatelé dále sdílet nebo uložit podle svých potřeb. Ukázka vygenerovaného PDF dokumentu je uvedena níže (Obrázek 28 - Vygenerovaný PDF report).



Obrázek 28 - Vygenerovaný PDF report

(zdroj: vlastní)

Jedním z primárních požadavků bylo, aby algoritmus pro výpočet času nabíjení co nejvěrněji odrážel reálný provoz. Vzhledem k tomu, že v reálném světě existuje mnoho faktorů ovlivňujících nabíjení, jako je počet uživatelů využívajících nabíjecí stanici, teplota baterie a okolí, technologie baterie nebo software vozidla, byl algoritmus navržen na základě pravidel obecné (průměrné) nabíjecí křivky [29]. Ta určuje, jak nabíjecí výkon postupně klesá v čase – na začátku se elektromobil nabíjí téměř plným výkonem (zde také záleží na maximální podporované rychlosti nabíjení samotného vozidla) a na konci, tedy když se stav nabití baterie (SoC) blíží k 100 %, se nabíjecí výkon sníží až na méně než jednu pětinu maximálního výkonu nabíječky. Simulace reálného rychlonabíjení v této práci byla tedy implementována podle těchto pravidel. Přesný a konkrétní algoritmus pro nabíjení má každé vozidlo, respektive výrobce, implementován odlišně. Algoritmus v této práci

v cyklech počítá přidávanou energii po sekundách, kdy pro nabíječky s vyšším výkonem (nad 20 kWh) postupně snižuje nabíjecí výkon v závislosti na aktuální úrovni nabití baterie, aby simuloval reálné zpomalování nabíjení při vyšších hodnotách SoC. Pro nabíječky s nižším výkonem (do 20 kWh) je výkon konstantní. Díky tomu je umožněno realistické modelování průběhu nabíjení, kde je výsledkem celkový čas nabíjení a také podrobné datové body obsahující změny výkonu a úrovně nabití během procesu, které jsou následně vyneseny do grafu. Samotná pravidla pro zpomalování a ubírání nabíjecího výkonu vůči stavu nabití jsou potom definována v následující tabulce (Tabulka 8 - Pravidla pro simulaci realistického nabíjení).

Tabulka 8 - Pravidla pro simulaci realistického nabíjení

Rozmezí stavu nabití baterie v %	Nabíjecí výkon
0-19	Výkon nabíječky * 1,0
20-34	Výkon nabíječky * 0,80
35-49	Výkon nabíječky * 0,85
50-59	Výkon nabíječky * 0,75
60-69	Výkon nabíječky * 0,60
70-79	Výkon nabíječky * 0,45
80-89	Výkon nabíječky * 0,30
90-100	Výkon nabíječky * 0,15

(zdroj: vlastní)

7 UŽIVATELSKÝ ZÁŽITEK A OPTIMALIZACE

V dalších kapitolách bude práce zaměřena na UX/UI design a optimalizaci aplikace. Budou popsány primární zásady tvorby jednoduchého a intuitivního uživatelského rozhraní, význam haptické odezvy a proces tvorby designových komponent ve Figma. Dále bude rozebrána práce s barvami, ikonami a texty, stejně jako návrh jednotlivých designových komponent aplikace. Kapitoly se rovněž zaměří na optimalizaci pro větší obrazovky, úpravy specifických komponent, jako jsou modální okna, a implementaci funkcí, jako je landscape lock a maximální šířka rozhraní.

7.1 UX/UI aplikace

Vzhledem k tomu, že většina konkurenčních aplikací analyzovaných v předchozích kapitolách postrádala jednoduché, intuitivní a minimalistické uživatelské rozhraní a jejich uživatelský zážitek nebyl optimální, byla pozornost zaměřena na to, aby Voltura v této oblasti vynikala. Návrhu uživatelského rozhraní, jeho stylu, vzhledu a celkové plynulosti ovládání proto bylo věnováno značné množství času.

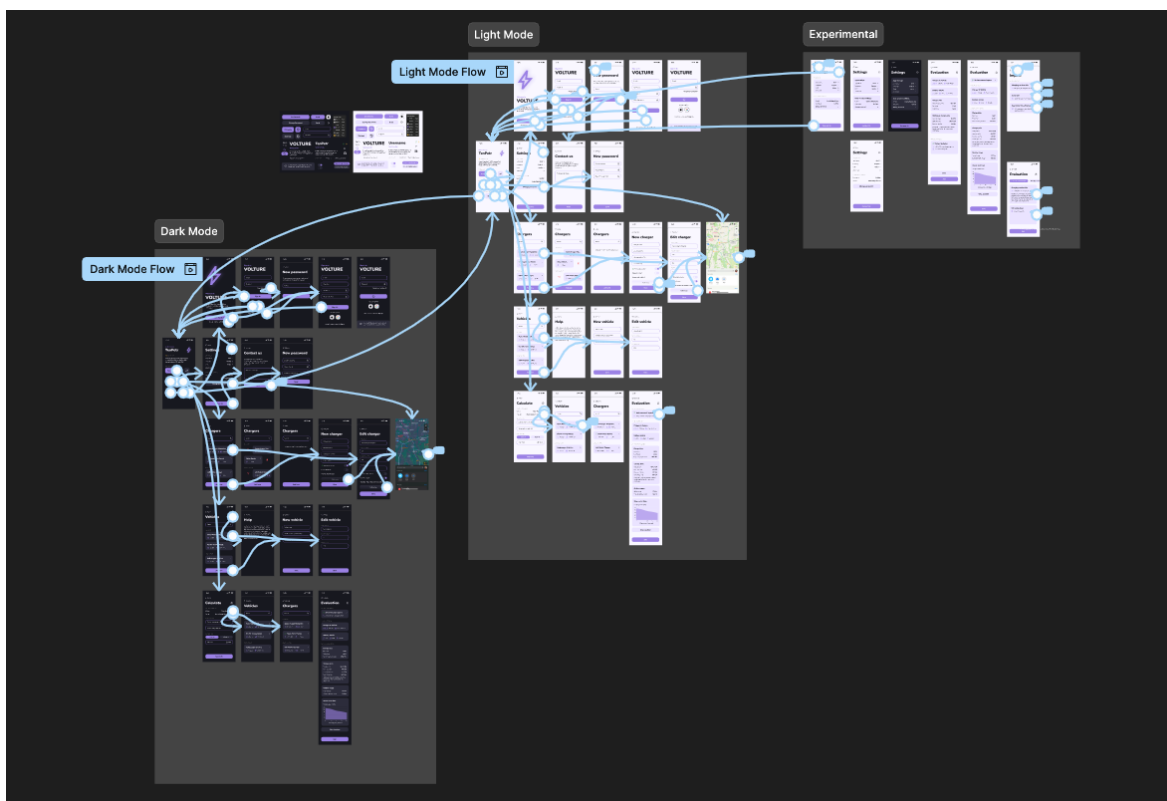
Cílem bylo vytvořit rozhraní, ve kterém se uživatelé rychle zorientují a intuitivně najdou požadované funkce, aniž by působilo přeplácane nebo naopak lacině. Zároveň bylo cílem také to, aby aplikace vytvářela v uživateli jakýsi prémiový pocit, na základě čehož byly také zvoleny adekvátní primární a sekundární barvy. Jednoduchost a přehlednost celé aplikace byly zásadní i proto, že cílová skupina obsahuje velmi široké věkové spektrum uživatelů. Důležitým aspektem návrhu uživatelského rozhraní bylo také zajistit, aby aplikace působila responzivně a agilně, proto byly do mnoha prvků aplikace implementovány animace a jemné přechody, které zvyšují vizuální atraktivitu a přispívají k plynulému a poutavému uživatelskému zážitku.

V neposlední řadě hraje důležitou roli haptická odezva jednotlivých komponent. Ať už uživatel kliká na primární nebo sekundární prvek, jako je tlačítko nebo tučný text spojený s další akcí, zařízení poskytne zpětnou vazbu prostřednictvím haptické odezvy, což má potom i přesah do reálného, fyzického světa. Tato funkce zlepšuje interaktivitu a potvrzuje uživateli jeho akci. Vzhledem k tomu, že aplikace je určena výhradně pro Apple zařízení, která haptickou odezvu standardně podporují, byla implementace této funkce snadnější. Je však potřeba zmínit, že u některých starších zařízení tato funkce není dostupná, což může mírně ovlivnit konzistenci uživatelského zážitku.

Součástí celého tohoto přístupu je také synchronizace uživatelských dat, která zajišťuje, že uživatelé mohou přecházet mezi různými zařízeními. Po přihlášení na svůj účet budou mít všechna svá data dostupná přesně tak, jak byli zvyklí z předchozího zařízení. Tato funkce přispívá nejen k pohodlnosti používání aplikace, ale také k vytvoření kontinuálního a bezproblémového uživatelského zážitku, který odpovídá moderním standardům dnešních aplikací.

7.2 Prototyp pro testování aplikace

Aby bylo zaručeno snadné a plynulé ovládání aplikace, byl v nástroji Figma vytvořen interaktivní prototyp ještě před samotnou implementací (Obrázek 29 - Prototyp a informační architektura aplikace). Tento prototyp byl testován na uživatelích v rámci cílové skupiny, což umožnilo získat cennou zpětnou vazbu. Na základě výsledků testování byly některé části uživatelského rozhraní upraveny tak, aby lépe odpovídaly potřebám uživatelů a zajišťovaly snadnější ovládání aplikace. Tento přístup zajistil, že finální verze designu reflektuje reálné požadavky a preference cílové skupiny.

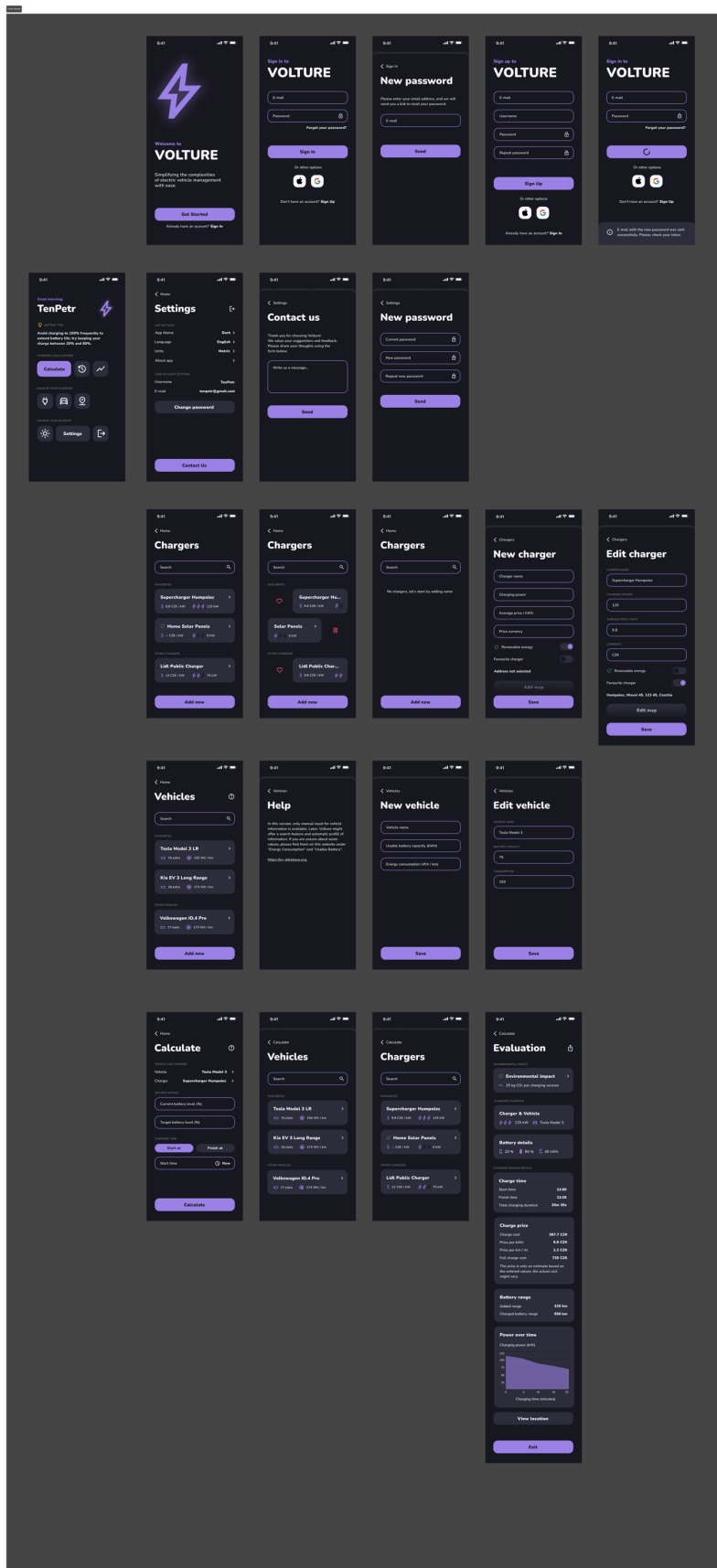


Obrázek 29 - Prototyp a informační architektura aplikace

(zdroj: vlastní)

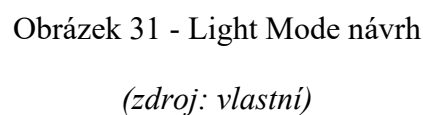
7.3 Komponenty a návrh ve Figmě

V online nástroji Figma byl před implementací aplikace vytvořen kompletní designový jazyk aplikace, který zahrnuje výčet základních komponent, jako jsou tlačítka, vstupní pole, velikosti a hierarchie textů, barvy, ikony a jejich velikosti, snackbar či další vlastní komponenty, mezery (spacing) a odsazení, karty a další prvky, které byly následně implementovány v rámci kódu, jak bylo popsáno v předchozích kapitolách. Tyto komponenty sloužily nejen jako základ pro tvorbu jednotlivých obrazovek aplikace ve Figmě, ale také jako designová reference během implementace (Obrázek 9 - UI komponenty aplikace). Všechny komponenty a obrazovky byly potom navrženy ve dvou barevných variantách – tmavé a světlé – aby odpovídaly preferencím uživatelů (Obrázek 30 - Dark Mode návrh a Obrázek 31 - Light Mode návrh). Primární barva aplikace, fialová, byla zvolena pro svůj pozitivní a klidný dojem, kterým na uživatele působí, ale také pro svůj luxusní a „královský“ nádech, který podporuje důvěryhodnost aplikace. [23]



Obrázek 30 - Dark Mode návrh

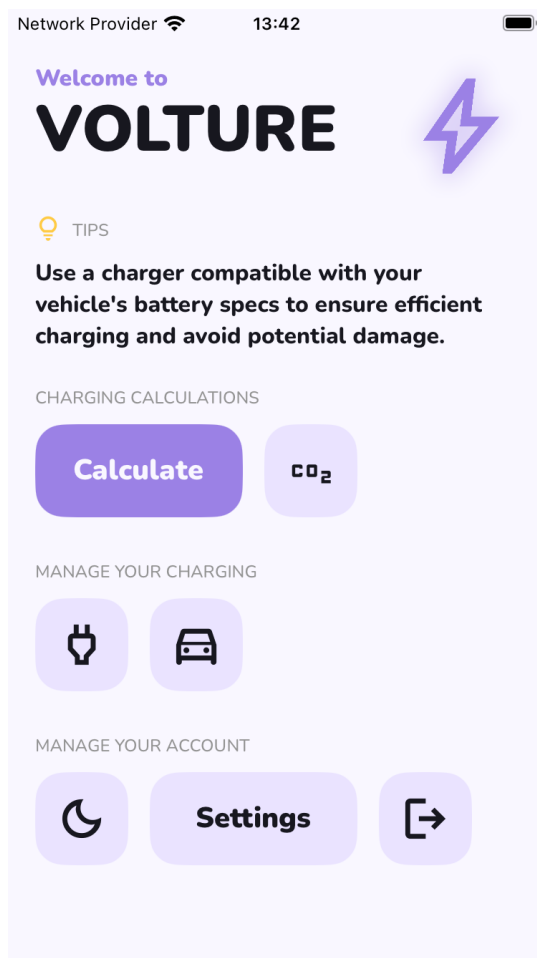
(zdroj: vlastní)



(zdroj: vlastní)

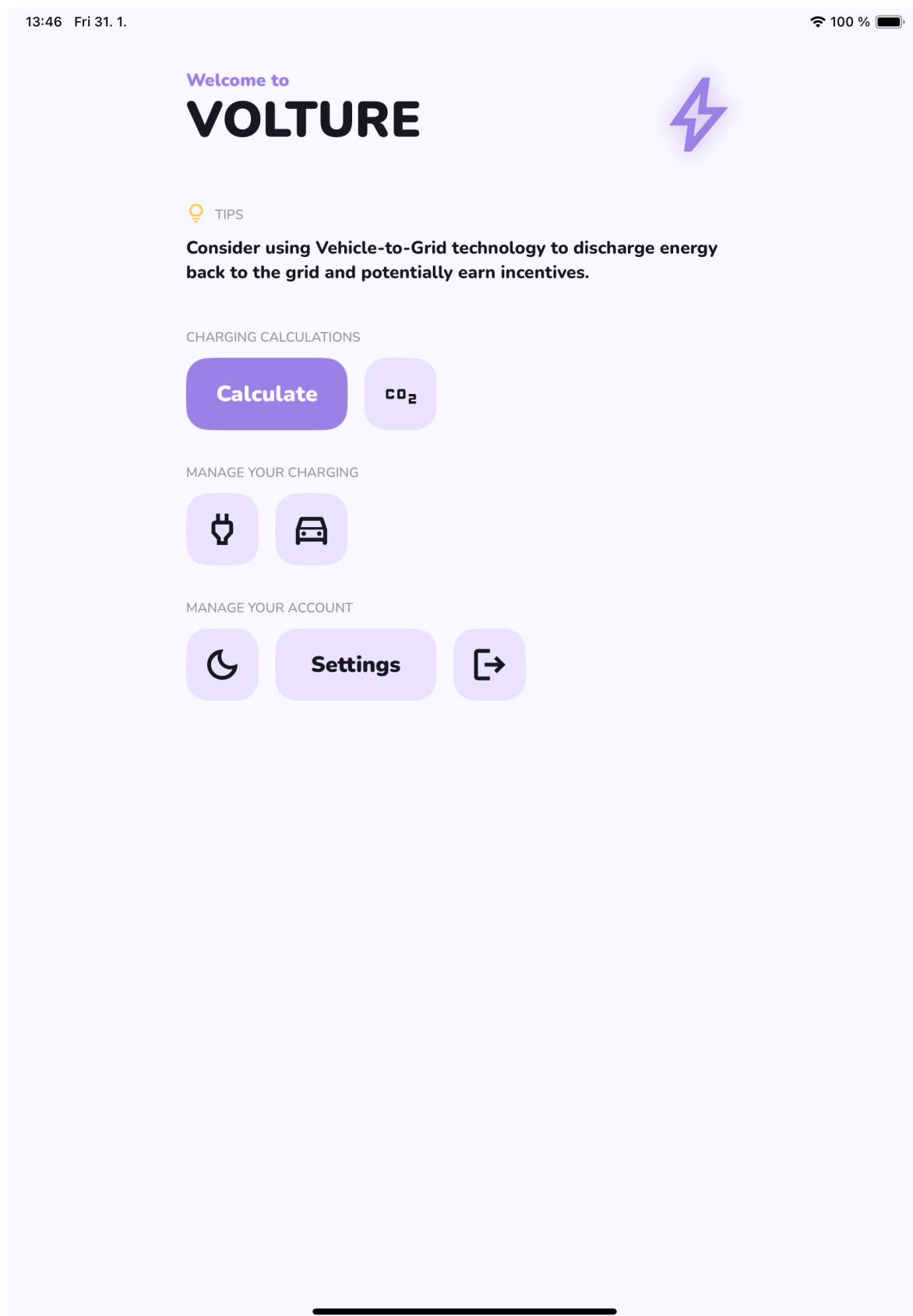
7.4 Optimalizace pro různé velikosti obrazovky

Aplikace Vulture je dostupná na Apple zařízeních s operačním systémem iOS, iPadOS a macOS, což zajišťuje větší přístupnost pro uživatele. Vzhledem k různým velikostem obrazovek bylo nutné zohlednit to, aby aplikace na všech zařízeních vypadala konzistentně a přehledně. Proto byla maximální šířka obsahu aplikace omezena na 500 bodů, což znamená, že na větších zařízeních, jako je iPad nebo Mac, je obsah aplikace zarovnán na střed s větším odsazením po stranách, což zabraňuje nepřehlednému rozložení komponent (Obrázek 33 - Aplikace na iPad Air). Na menších zařízeních, například na iPhone SE 3 (Obrázek 32 - Aplikace na iPhone SE 3), se odsazení dynamicky přizpůsobí a tvoří přibližně 5 % z každé boční strany, což také zajišťuje správné zobrazení obsahu. Aby byl zachován hezký a funkční design, byla navíc deaktivována možnost rotace aplikace do landscape módu. Vzhledem k tomu, že aplikace svou povahou z větší plochy na šířku nijak neprofituje, představovala by optimalizace pro tento režim zbytečné úsilí bez přidané hodnoty.



Obrázek 32 - Aplikace na iPhone SE 3

(zdroj: vlastní)



Obrázek 33 - Aplikace na iPad Air

(zdroj: vlastní)

8 VERZOVÁNÍ, REFACTORING A TESTOVÁNÍ

V této kapitole bude rozebrána správa verzí, refactoring a ověřování funkčnosti aplikace. Budou popsány základy práce s verzovacím systémem Git a pravidla pro správné sémantické verzování, včetně využití Gitu při spolupráci více vývojářů. Dále bude zaměřena pozornost na refactoring a principy psaní čistého a přehledného kódu, které přispívají k dlouhodobé udržitelnosti projektu. Kapitola se také bude věnovat ověřování výpočtů a testování na reálných nabíjecích stanicích a vozidlech.

8.1 Verzovací systém

V rámci tohoto projektu byl jako verzovací systém vybrán Git, což je moderní a velmi rozšířený nástroj pro správu verzí software, který se mezi vývojáři a projekty stal jakýmsi standardem. Git byl původně vytvořen Linusem Torvaldsem v roce 2005 jako nástroj pro správu vývoje jádra Linuxu. [24]

Při vývoji projektu bylo potřeba stanovit nějaký proces, který by vývoj a verzování projektu, potažmo zálohování kódu, usnadnil. Proto je využíván systém větví, pull requestů a issues, které jsou spravovány na GitHubu. Ten zároveň slouží jako vzdálený server / úložiště pro zálohování nebo sdílení kódu. Každá nová funkce nebo úprava je založena na issue, což je úkol či požadavek, který popisuje specifickou změnu nebo rozšíření aplikace. Pro vývoj konkrétní funkce je vytvořena samostatná tzv. „feature branch“ pojmenovaná podle čísla a popisu daného issue, například `feature/102-add-user-authentication`. Díky tomuto přístupu lze případně změny jednoduše vrátit zpět, aniž by se to dotklo zbytku existujícího kódu. Po dokončení implementace dané funkce je otevřen na GitHubu pull request, kde se provádí kontrola kódu. V rámci tohoto procesu se ověří, zda je kód bez chyb, alespoň do určité míry, a že odpovídá definovaným požadavkům a cílům. Když je pull request schválený, je větev integrována do hlavní větve `main` (nebo `master`), která představuje produkční verzi aplikace. Kód v této větvi by měl být vždy připraven k sestavení a distribuci aplikace. Pro opravy chyb je využíván systém tzv. „fix branch“, například `fix/112-correct-timestamp-format`. Tato větev obsahuje opravu chyby, která se po otestování také integruje do hlavní větve přes proces pull requestu a code review. Tento přístup vývoje je velmi škálovatelný a je připraven na potencionální rozšíření týmu o další vývojáře, pokud by v budoucnu tým rostl.

8.2 Verzování aplikace

Pro verzování aplikace, vyvíjené ve SwiftUI, byl zvolen systém sémantického verzování, který je mimo jiné také výchozí variantou. Tento systém je taktéž široce používán v open-source projektech, interních nástrojích i mobilních aplikacích. Jeho největší výhodou je srozumitelný a konzistentní přístup k číslování verzí, které se skládají ze tří částí. Typická verze vypadá například takto: 2.5.8-alpha. Podrobná pravidla a doporučení k sémantickému verzování jsou uvedena a popsána více v oficiální dokumentaci. [25] Sémantické verzování se dělí na následující části.

Major – tato část se inkrementuje při změnách, které nejsou zpětně kompatibilní. Typicky se jedná o zásadní přidání nebo úpravy funkcionalit, které ovlivňují chování aplikace způsobem, které vyžaduje změnu v jejím používání (2.x.x-x). [25]

Minor – tato část se zvyšuje, pokud je přidána nová funkcionalita nebo významná oprava chyb, která ale zachovává zpětnou kompatibilitu (x.5.x-x). [25]

Patch a Build – patch číslo se zvyšuje při menších opravách chyb nebo drobných úpravách, které nemají vliv na kompatibilitu (x.x.8-x). Build číslo není povinné a může se použít pro další specifikaci, například u testovacích nebo alfa verzí aplikace (x.x.x-alpha). [25]

V případě iOS aplikace, psané ve SwiftUI, se verze aplikace konfiguruje v souboru `Info.plist`. Tento soubor je součástí každého projektu ve Swiftu a kromě verze aplikace obsahuje další důležité informace, jako je například identifikátor balíčku (Bundle Identifier) nebo nastavení oprávnění (permissions). Pro verzi aplikace se typicky využívají klíče `CFBundleShortVersionString` (číslo verze) a `CFBundleVersion` (číslo buildu), které je možné upravovat v rozhraní Xcode nebo přímo v souboru. Verze aplikace je zásadní nejen pro vývoj, ale také pro publikaci v App Store, protože Apple vyžaduje, aby každá aktualizace aplikace měla unikátní číslo buildu, což zajišťuje konzistenci při správě verzí a distribuci. Díky tomuto přístupu je potom také řešení problémů v produkčním prostředí o něco jednodušší.

8.3 Refactoring a čistý kód

V rámci vývoje aplikace byl kladen důraz na čistotu, čitelnost a dodržování principů psaní kvalitního kódu. Tyto principy vycházejí z doporučení popsaných v literatuře autorů jako

jsou Martin Fowler anebo Robert Cecil Martin (zvaný také Uncle Bob), kde některé z nich budou stručně popsány v této kapitole v rámci kontextu této práce.

Při vývoji uživatelského rozhraní v rámci SwiftUI bylo primárně myšleno na modularizaci komponent, kde každý větší vizuální prvek aplikace, například tlačítko nebo vstupní pole, byl rozdělen do menších, snadno znovu použitelných komponent. Jedním z klíčových principů, který byl při návrhu dodržen, bylo pravidlo Open-Closed Principle, které říká, že moduly by měly být otevřené pro rozšíření, ale uzavřené pro modifikaci. Díky tomu je možné přidávat nové funkce bez nutnosti zasahovat do již existujícího kódu. [26]

Refactoring kódu byl pravidelně a průběžně prováděn během celého vývoje, aby se zajistila konzistentní kvalita kódu. Například při práci s některými UI elementy byly odstraněny duplicitní fragmenty kódu a nahrazeny generikou, která umožní jednodušší přizpůsobení různým datovým typům. Tento proces zahrnoval také úpravu názvů funkcí, struktur, tříd anebo proměnných tak, aby přesně vyjadřovaly jejich účel, což je základním požadavkem čistého kódu. Dále sem patří například přesouvání definic metod z View do ViewModelů, případně do servisních tříd, aby se dodrželo konkrétní určení dané komponenty. Refactoring byl prováděn v souladu s pravidlem „refaktorujte často a malými kroky“, které zajišťuje, že každý krok je snadno dohledatelný, například ve verzovacím systému, a méně náchylný k chybám. [27]

Architektura a struktura aplikace byla navržena tak, aby byla přehledná a dobře organizovaná. Logika aplikace byla rozdělená do vrstev, kde uživatelské rozhraní spolupracuje s ViewModely podle principu MVVM. Tento způsob práce udržel jednotlivé části aplikace nezávislé na sobě, což usnadní pozdější úpravy a vylepšení. [28] Tomuto tématu se práce věnovala více v kapitolách výše.

8.4 Ověřování výpočtů

Pro zajištění pokud možno co nejpresnějších výsledků a poskytnutí relevantních dat uživatelům bylo nutné provést testování a ověřování výpočtů v reálném provozu. Aplikace byla testována přímo na reálném vozidle Tesla Model 3 Long Range z roku 2023 s kapacitou baterie 75 kWh a maximální rychlostí nabíjení 250 kW. Testování probíhalo na různých typech nabíjecích stanic, včetně Tesla Superchargerů s vysokým výkonem (desítky až stovky kW) a pomalejších domácích nabíječek (jednotky kW). Cílem bylo určit

přesnost a odchylku vypočítaných hodnot, zvláště doby nabíjení, a porovnat je s hodnotami naměřenými v reálném provozu.

Z tabulky níže (Tabulka 9 - Měření nabíjení) lze vyčíst, že aplikace dosahuje vysoké přesnosti při výpočtech na pomalejších, domácích nabíječkách (v rámci tohoto měření konkrétně s výkonem 9 kW), kde je nabíjecí výkon téměř konstantní a méně ovlivněný vnějšími faktory než u rychlejších nabíjecích stanic, jako jsou Superchargery od firmy Tesla. U domácích nabíječek s výkonem 9 kW je průměrná přesnost výpočtu času nabíjení 98,96 %, zatímco u rychlejších nabíječek, použitých v rámci tohoto testování, dosahuje průměrná přesnost 76,26 %. Jak již bylo zmíněno v kapitole výše, predikce nabíjecího výkonu v čase vychází z obecné nabíjecí křivky, která však nemůže zohlednit faktory jako teplotu a zdraví baterie, obsazenost nabíjecích stanic nebo konkrétní implementaci nabíjecího algoritmu ve vozidle, což může značně ovlivnit reálný průběh nabíjení.

Tabulka 9 - Měření nabíjení

Číslo měření	Doba nabíjení (Vulture)	Doba nabíjení (Realita)	Start SoC	Konec SoC	Max. výkon nabíječky	Odchylka	Přesnost
1	21 min	23 min	17 %	65 %	125 kW	8,7 %	91,3 %
2	19 min	25 min	24 %	65 %	125 kW	24 %	76 %
3	280 min	288 min	24 %	80 %	9 kW	2,8 %	97,2 %
4	220 min	226 min	36 %	80 %	9 kW	2,7 %	97,3 %
5	195 min	195 min	41 %	80 %	9 kW	0 %	100 %
6	135 min	136 min	53 %	80 %	9 kW	0,7 %	99,3 %
7	250 min	252 min	30 %	80 %	9 kW	0,8 %	99,2 %
8	305 min	303 min	19 %	80 %	9 kW	0,7 %	99,3 %
9	245 min	247 min	31 %	80 %	9 kW	0,8 %	99,2 %
10	13 min	20 min	44 %	70 %	125 kW	35 %	65 %
11	270 min	270 min	16 %	70 %	9 kW	0 %	100 %
12	50 min	51 min	70 %	80 %	9 kW	2 %	98 %
13	285 min	288 min	23 %	80 %	9 kW	1 %	99 %
14	210 min	210 min	38 %	80 %	9 kW	0 %	100 %
15	8 min	11 min	35 %	70 %	250 kW	27,2 %	72,8 %

(zdroj: vlastní)

ZÁVĚR

Hlavním cílem a záměrem této aplikace, potažmo práce, bylo usnadnit přechod uživatelů z vozidel se spalovacím motorem na elektromobily a zjednodušit jim tak plánování nabíjení a s tím související výpočty či nadbytečné starosti. Součástí práce bylo také zaměřit se na nedostatky stávajících řešení, která nejsou dostatečně přehledná, intuitivní nebo neposkytují všechny potřebné informace a vypočtené hodnoty. Aplikace tak slouží jako nástroj, který je snadno použitelný a zároveň poskytuje uživatelům všechny důležité informace o nabíjení, spotřebě, nákladech nebo vlivu nabíjení na životní prostředí, díky čemuž mohou uživatelé činit informovanější rozhodnutí a plánovat tak efektivněji.

Aplikace nabízí velké množství výpočtů, které uživatelům pomáhají s plánováním nabíjení a poskytují detailní informace o jeho průběhu. Umožňuje tak sledovat dopad nabíjení na životní prostředí, zobrazit informace o nabíječce, vozidle a stavu baterie, vypočítat množství přidané energie za nabití, celkový čas nabíjení a související náklady, ať už za jedno nabití, cenu za kilometr nebo odhad pro plné dobití baterie. Součástí výsledků je také přehled dojezdu a graf znázorňující pokles nabíjecího výkonu v závislosti na úrovni nabití baterie, který je vypočítán a sestaven pomocí algoritmu simulujícího realistický průběh nabíjení. Uživatelé mohou výsledky výpočtů jednoduše sdílet prostřednictvím vygenerovaného PDF reportu. Pro intuitivnější a jednodušší použití aplikace nabízí uživatelské účty umožňující synchronizaci dat mezi zařízeními. Kromě toho poskytuje možnosti přizpůsobení, včetně volby barevného režimu (světlý nebo tmavý), preferovaného systému jednotek (metrický nebo imperiální) a jazyka (čeština, angličtina a slovenština). Design a struktura uživatelského rozhraní aplikace byla navržena v nástroji Figma a následně implementována pomocí frameworku SwiftUI v programovacím jazyce Swift. Samotná aplikace je dostupná pro zařízení od Apple, tedy na iPhone, iPad a Mac.

Do budoucna lze aplikaci rozšířit o další možnosti a funkce – například přidání dalších výpočtů pro ještě přesnější plánování, automatizace přidávání vozidel a nabíječek do aplikace pomocí dostupných API nebo veřejných dat. Dále je možné aplikaci rozšířit o webovou verzi pro jednodušší přístup z různých zařízení, integrace počasí a teploty do výpočtů pro realističtější odhady nabíjení a spotřeby nebo možnost zobrazit podrobný rozpis nákladů na nabíjení na základě tarifních plánů – například denní/noční sazba.

SEZNAM POUŽITÉ LITERATURY

- [1] IOS 18. Apple [online]. 2024 [cit. 2024-11-13]. Dostupné z: <https://www.apple.com/ios/ios-18/>
- [2] iPadOS 18. Apple [online]. 2024 [cit. 2024-11-13]. Dostupné z: <https://www.apple.com/ipados/ipados-18/>
- [3] MacOS Sequoia. Apple [online]. 2024 [cit. 2024-11-13]. Dostupné z: <https://www.apple.com/macos/macos-sequoia/>
- [4] Počítače Mac s čipem Apple. Apple Support [online]. 2024 [cit. 2024-11-13]. Dostupné z: <https://support.apple.com/cs-cz/116943>
- [5] Co je iCloud? Apple Support [online]. 2024 [cit. 2024-11-13]. Dostupné z: <https://support.apple.com/cs-cz/guide/mac-help/mh36832/mac>
- [6] Xcode. Apple Developer [online]. 2024 [cit. 2024-11-16]. Dostupné z: <https://developer.apple.com/xcode/>
- [7] App Review Guidelines. Apple Developer [online]. 2024 [cit. 2024-11-16]. Dostupné z: <https://developer.apple.com/app-store/review/guidelines/>
- [8] SwiftUI. Apple Developer [online]. 2024 [cit. 2024-11-16]. Dostupné z: <https://developer.apple.com/documentation/swiftui/>
- [9] UIKit. Apple Developer [online]. 2024 [cit. 2024-11-16]. Dostupné z: <https://developer.apple.com/documentation/uikit/>
- [10] The Objective-C Programming Language. Apple Developer [online]. 2013 [cit. 2024-11-16]. Dostupné z: <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html>
- [11] The Swift Programming Language. Swift Documentation [online]. 2024 [cit. 2024-11-30]. Dostupné z: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/>
- [12] A Swift Tour. Swift Documentation [online]. 2024 [cit. 2024-12-06]. Dostupné z: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/guidedtour/>

- [13] SwiftUI. Apple Developer [online]. 2024 [cit. 2024-12-08]. Dostupné z: <https://developer.apple.com/documentation/swiftui/>
- [14] MVVM in iOS Swift. Medium [online]. 2023 [cit. 2024-12-17]. Dostupné z: <https://medium.com/@zebayasmeen76/mvvm-in-ios-swift-6afb150458fd>
- [15] Understanding the Role of MVC in iOS Development: How Apple Utilizes the MVC Architecture in UI Libraries. Medium [online]. 2023 [cit. 2024-12-21]. Dostupné z: <https://medium.com/@viniciusnadin/understanding-the-role-of-mvc-in-ios-development-how-apple-utilizes-the-mvc-architecture-in-ui-4aa128ed59ad>
- [16] Firebase | Google's Mobile and Web App Development Platform. Firebase [online]. 2024 [cit. 2024-12-21]. Dostupné z: <https://firebase.google.com/>
- [17] Pricing plans. Firebase [online]. 2024 [cit. 2024-12-21]. Dostupné z: <https://firebase.google.com/pricing>
- [18] Firebase Authentication. Firebase [online]. 2024 [cit. 2024-12-21]. Dostupné z: <https://firebase.google.com/docs/auth>
- [19] Firebase Realtime Database. Firebase [online]. 2024 [cit. 2024-12-21]. Dostupné z: <https://firebase.google.com/docs/database>
- [20] Cloud Firestore. Firebase [online]. 2024 [cit. 2024-12-21]. Dostupné z: <https://firebase.google.com/docs/firestore>
- [21] Cloud Functions for Firebase. Firebase [online]. 2024 [cit. 2024-12-21]. Dostupné z: <https://firebase.google.com/docs/functions>
- [22] Managing assets with asset catalogs. Apple Developer [online]. 2024 [cit. 2024-12-29]. Dostupné z: <https://developer.apple.com/documentation/xcode/managing-assets-with-asset-catalogs>
- [23] Color Psychology: Purple. Treefrog Marketing [online]. 2025 [cit. 2025-01-02]. Dostupné z: <https://treefrogmarketing.com/color-psychology-purple/>
- [24] About. Git [online]. [cit. 2025-01-07]. Dostupné z: <https://git-scm.com>
- [25] Sémantické verzování 2.0.0. Sémantické verzování 2.0.0 [online]. 2024 [cit. 2025-01-07]. Dostupné z: <https://semver.org/lang/cs>
- [26] MARTIN, Robert C. Clean architecture: a craftsman's guide to software structure and design. London, England: Prentice Hall, [2018]. ISBN 0134494164.

- [27] FOWLER, Martin. Refactoring: zlepšení existujícího kódu. Praha: Grada, 2003. Moderní programování. ISBN 80-247-0299-1.
- [28] MARTIN, Robert C. Clean code: a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, c2009. ISBN 9780132350884.
- [29] What is a charging curve? Octopus Electroverse [online]. 2025 [cit. 2025-03-03]. Dostupné z: <https://electroverse.com/community/ev-blogs-and-guides/what-is-a-charging-curve>

SEZNAM OBRÁZKŮ

Obrázek 1 - Xcode	19
Obrázek 2 - Simulátor iPhone 16.....	23
Obrázek 3 - EV CALC – Výpočty a nastavení	48
Obrázek 4 - EV Charge Calculator – Výpočty a nastavení.....	49
Obrázek 5 - EV Range Calculator – Výpočty vzdálenosti a zdraví baterie	50
Obrázek 6 - EV Calculator – Výběr vozidla a nástroje pro výpočet.....	51
Obrázek 7 - EVrc – Výpočty a nastavení.....	52
Obrázek 8 - EV Charge Calculator – Výpočty a výsledky	54
Obrázek 9 - UI komponenty aplikace	62
Obrázek 10 - Uvítací obrazovka a registrace	70
Obrázek 11 - Přihlášení a obnova zapomenutého hesla.....	71
Obrázek 12 - Poskytovatelé přihlášení.....	72
Obrázek 13 - Tabulka s uživateli	72
Obrázek 14 - Snackbar a validace.....	73
Obrázek 15 - Nastavení aplikace	74
Obrázek 16 - Modal sheet s výběrem jednotek.....	75
Obrázek 17 - Domovská obrazovka.....	77
Obrázek 18 - CO ₂ na kWh pro dané oblasti.....	78
Obrázek 19 - Nabíječky a vozidla.....	79
Obrázek 20 - Formulář pro vozidla s validační chybou a nápověda.....	81
Obrázek 21 - Formulář pro nabíječky a snackbar po úspěšném uložení	82
Obrázek 22 - Úprava nabíječky a vozidla.....	83
Obrázek 23 - Loading spinner a prázdný seznam vozidel	84
Obrázek 24 - Snackbar v offline režimu	85
Obrázek 25 - Vstupní formulář pro výpočty.....	88
Obrázek 26 - Výsledky výpočtů.....	89
Obrázek 27 - Výsledky výpočtů – graf	90
Obrázek 28 - Vygenerovaný PDF report	91
Obrázek 29 - Prototyp a informační architektura aplikace	94
Obrázek 30 - Dark Mode návrh	96
Obrázek 31 - Light Mode návrh.....	97
Obrázek 32 - Aplikace na iPhone SE 3.....	98
Obrázek 33 - Aplikace na iPad Air	99

SEZNAM TABULEK

Tabulka 1 - Nefunkcionální požadavky	66
Tabulka 2 - Funkcionální požadavky na uživatelské účty	67
Tabulka 3 - Funkcionální požadavky na nastavení	67
Tabulka 4 - Funkcionální požadavky na domovskou obrazovku	67
Tabulka 5 - Funkcionální požadavky na nabíječky	68
Tabulka 6 - Funkcionální požadavky na vozidla	68
Tabulka 7 - Funkcionální požadavky na výpočty	68
Tabulka 8 - Pravidla pro simulaci realistického nabíjení	92
Tabulka 9 - Měření nabíjení	103

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CO ₂	Oxid uhličitý
PDF	Portable Document Format
SDK	Software Development Kit
ARM	Označení procesorů vyvinutých britskou firmou ARM Holdings
API	Application Programming Interface
GPS	Global Positioning System
MVVM	Model-View-ViewModel
MVC	Model-View-Controller
OAUTH	Open Authorization
SSO	Single Sign-On
AI	Artificial Intelligence
ML	Machine Learning
GCP	Google Cloud Platform
JSON	JavaScript Object Notation
NoSQL	No Structured Query Language
HTTP	HyperText Transfer Protocol
kWh	Kilowatthodina
Wh	Watthodina
kW	Kilowatt
min	Minuta
EV	Electric Vehicle
APP	Application
UX	User Experience
UI	User Interface
NR	Non-Functional Requirement

FR Functional Requirement

ID Identifier

UID User Identifier

EU Evropská unie

SoC State of Charge

SEZNAM PŘÍLOH

Příloha P I: CD disk se zdrojovým kódem a diplomovou prací