FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Mobilní aplikace pro zaznamenávání odpracovaného času
Student:	Bc. Tomáš Batěk
Vedoucí:	Ing. Tadeáš Sosín
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem práce je analýza, návrh a implementace mobilní aplikace pro systém iOS (frontend i backend) pro zaznamenávání odpracovaného času, která umožňuje integraci s různými spouštěči akcí a propojení s dalšími systémy pro zaznamenávání a spravování odpracovaného času.

1. Proveďte rešerši různých řešení pro spouštěče měření času (např. fyzické ovladače nebo automatizace) a stanovte požadavky pro integraci těchto spouštěčů s aplikací.
2. Proveďte rešerši existujících systémů pro zaznamenávání odpracovaného času a navrhněte možnosti exportu dat z aplikace do těchto systémů. Stanovte požadavky pro integraci exportních funkcí s těmito systémy.
3. Navrhněte uživatelské rozhraní klientské iOS aplikace.
4. Navrhněte nástroje a jiné produkty třetích stran potřebné pro realizaci aplikace a vnitřní architekturu aplikace.
5. Po dohodě s vedoucím práce realizujte prototyp frontendu a backendu této platformy.
6. Vhodně aplikaci otestujte a proveďte uživatelské testování.
7. Navrhněte prostředí pro podporu provozu aplikace, které umožní její další rozvoj a podporu stávajících uživatelů.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 26. ledna 2024 v Praze.

Diplomová práce

MOBILNÍ APLIKACE PRO ZAZNAMENÁVÁNÍ ODPRACOVANÉHO ČASU

Bc. Tomáš Batěk

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Tadeáš Sosín
8. května 2024

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2024 Bc. Tomáš Batěk. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci: Batěk Tomáš. *Mobilní aplikace pro zaznamenávání odpracovaného času.* Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam výpisů kódu

Chtěl bych poděkovat svému vedoucímu práce, panu Ing. Tadeášovi Sosínovi, za pomoc a vstřícnost během tvorby práce. Také bych chtěl poděkovat své rodině za podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 citovaného zákona.

V Praze dne 8. května 2024 .....

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací mobilní aplikace pro systém iOS, která umožňuje zaznamenávání a správu odpracovaného času a integraci s dalšími systémy. Čtenáře v rámci analýzy seznamuje s problematikou zaznamenávání odpracovaného času, s možnostmi integrace se spouštěči měření času a s existujícími řešeními pro tuto potřebu, a zaměřuje se na platformu mobilních telefonů s operačním systémem iOS spolu se specifiky vývoje pro tuto platformu. Podle výstupů analýzy poté navrhuje uživatelské rozhraní iOS aplikace, její funkcionality, a také navrhuje architekturu aplikace a nástroje potřebné pro implementaci. Poté práce v rámci realizace popisuje, jak byla aplikace vyvíjena, popisuje architekturu a strukturu projektu a implementaci jednotlivých navržených funkcionalit. V navazující kapitole se práce věnuje testování, popisuje jak automatizované testy, tak metodiku a výsledky uživatelského testování. Práce se poté ještě věnuje prostředí pro budoucí podporu a provoz aplikace, ve kterém navrhuje metodiky budoucího vývoje. V závěru pak práce shrnuje a hodnotí provedené řešení.

Klíčová slova mobilní aplikace, iOS, měření odpracovaného času, správa odpracovaného času, Swift, Kotlin, Multi-platformní vývoj, Ktor

Abstract

This thesis focuses on designing and implementing an iOS mobile application for time tracking and integration with other systems. It describes the use for time tracking, the options for integration with tracking triggers and with existing time tracking systems, as well as the iOS system and the specifics of iOS application development. Based on the analysis, the thesis then proposes the design of the app's user interface, its features and also proposes the app architecture and tools necessary for the app's development. The thesis then describes the actual implementation of the application, its features and describes the app's architecture and structure. In the subsequent chapter, the thesis focuses on testing, it describers both automatic testing and the methodology and outputs of user testing. The thesis then also focuses on the environment for future support, where it then describes recommendations for future development. In the conclusion, the thesis sums up and evaluates the solution produced by it.

Keywords mobile application, iOS, time tracking, time tracking management, Swift, Kotlin, Multiplatform, Ktor

Seznam zkratek

API	Application Programming Interface
BLE	Bluetooth Low Energy
SDK	Software Development Kit
UI	User Interface
BaaS	Backend as a Service
DBaaS	Database as a Service
CSV	Comma-Separated Values
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
DTO	Data transfer object
OS	Operating System

Úvod

Zaznamenávání a správa odpracovaného času je problematikou, která může sloužit velkému množství uživatelů různých odvětví zaměstnání. Ať už se jedná o sledování pracovních hodin za účelem fakturace, za účelem sledování docházky, nebo za účelem analýzy produktivity, jedná se o službu, kterou vyžaduje a používá velké množství zaměstnanců a zaměstnavatelů.

Vzhledem k tomu, že je poptávka po službě zprostředkovávající měření a správu odpracovaného času poměrně vysoká, existuje řada řešení, které se tuto službu snaží s různými přístupy a funkcionalitami nabízet. Mohou proto nastávat situace, kdy budou uživatelé potřebovat nějakou formu integračního prvku, který bude moct propojovat více řešení a přístupů, jak odpracovaný čas měřit a jak ho analyzovat.

Tato práce nabízí implementaci vlastního řešení aplikace pro měření a správu odpracovaného času a jako přidanou hodnotu analyzuje možnosti integrace se spouštěči měření času (fyzické ovladače, automatizace, ...), možnosti integrace s existujícími systémy pro správu odpracovaného času, poté navrhuje, jaké typy integrací implementovat, a v realizované aplikaci tyto integrace nabízí.

Podkladem pro funkcionality realizované aplikace je analýza existujících řešení a přístupů pro měření a správu času, analýza možností pro integraci a návrh používání aplikace z výstupů analýzy.

Implementace aplikace zhotovená v rámci této práce nemá šanci se vyrovnat existujícím aplikacím řešícím problematiku měření a správy času, které mají mnohem větší časové a finanční rozpočty a stojí za nimi roky vývoje velkých týmů. Má ale šanci poskytnout základ aplikace, která mimo samotnou možnost měření a správy odpracovaného času dokáže nabídnout i některé vhodné formy integrace se spouštěči měření a zmíněnými existujícími aplikacemi. Na tomto základu poté může být navázáno dalším úsilím, které funkcionality a možnosti integrace aplikace rozšíří.

Mimo poskytnutí základu integrační aplikace pro měření a správu odpracovaného času má také tato práce potenciál nabídnout seznámení s aktuálními moderními přístupy vývoje mobilních aplikací a s nástroji k tomu potřebnými.

Kapitola 1
Cíl práce

Hlavním cílem diplomové práce je analýza, návrh a implementace mobilní aplikace pro systém iOS (frontend i backend) pro zaznamenávání odpracovaného času, která umožňuje integraci s různými spouštěči akcí a propojení s dalšími systémy pro zaznamenávání a spravování odpracovaného času. V rámci návrhu je cílem navrhnout vhodné technické řešení pro implementaci aplikace, funkcionality aplikace a jejich uživatelské rozhraní a tohoto návrhu se poté při realizaci aplikace držet. Následně je cílem práce realizované řešení vhodně otestovat a provést uživatelské testování.

Vedlejším cílem realizovaného řešení je flexibilita a rozšiřovatelnost technické implementace aplikace. Jelikož základ integrační aplikace pro měření a správu odpracovaného času, který tato práce realizuje, má široké možnosti budoucího rozšíření, ať už o aplikace pro další platformy (Android, Web, ...), tak o další funkcionality (propojení s mnoha dalšími řešeními), tak je přinejmenším vhodné, aby na tyto rozšíření byla co nejlepším způsobem připravena.

Výsledky diplomové práce budou přínosné pro potenciální budoucí uživatele nově realizované platformy, ale také pro čtenáře, kteří se chtějí seznámit s technickým přístupem řešení, které bylo v této práci zvoleno, proč bylo zvoleno, jaké byly alternativy a jaké je poté v závěru zhodnocení tohoto přístupu. Vývoj mobilních aplikací je rostoucím průmyslem, který je stále ve fázi objevování nových přístupů a řešení, a tato práce má potenciál do něj přispět zkušenostmi, které byly při její realizaci získány.

Kapitola 2
Analýza

Tato kapitola rozebírá, k čemu slouží zaznamenávání odpracovaného času. Poté popisuje možnosti, jak měření času spouštět a také se věnuje příkladům řešení, která již pro zaznamenávání odpracovaného času existují, a jaké jsou možnosti propojení s těmito řešeními. V dalších sekcích se poté obecně zaměřuje na vývoj aplikací pro mobilní platformu iOS včetně možností multi-platformního a cross-platformního vývoje.

2.1 Zaznamenávání odpracovaného času

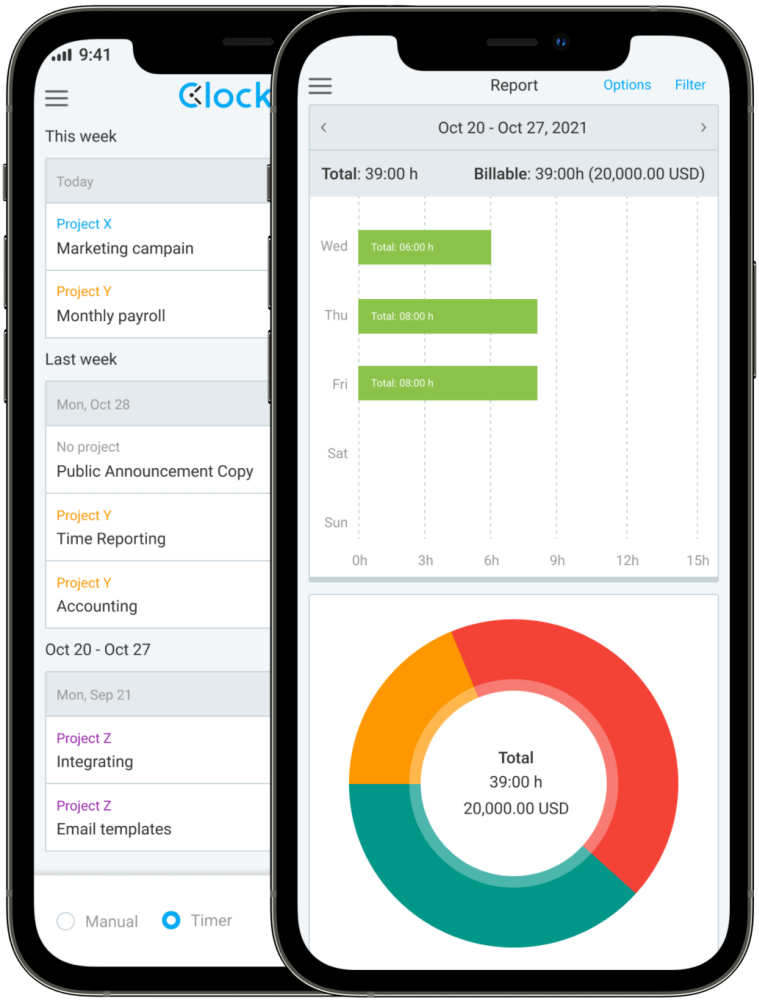
Zaznamenávání odpracovaného času může být potřeba z několika různých důvodů, buď z pohledu jednotlivce, nebo z pohledu spolupráce v nějakém týmu. Mezi tyto důvody může patřit například:

- **Sledování pracovních hodin:** Pomáhá zaměstnancům a podnikům sledovat, kolik času strávili na jednotlivých úkolech, projektech a aktivitách během pracovní doby. To může být užitečné pro sledování produktivity, hodnocení efektivity práce a plánování rozpočtu času.
- **Fakturace a účtování:** Pro profesionály a firmy, které účtují za své služby na základě odpracovaného času, umožňuje zaznamenávání snadné sledování času stráveného na určitých projektech a klientech pro účely fakturace.
- **Analýza produktivity:** Poskytuje data a statistiky o tom, jaký čas je věnován různým úkolům a projektům. To umožňuje identifikovat trendy v pracovních návycích, optimalizovat časové plánování a zlepšit efektivitu práce.
- **Správa projektů:** Pomáhá organizovat časové údaje spojené s různými projekty a úkoly, což usnadňuje plánování, delegování a monitorování pokroku.
- **Transparentnost a komunikace:** Pro týmy umožňuje transparentně sdílet informace o čase stráveném na různých aktivitách, což podporuje spolupráci a komunikaci v rámci týmu.

Celkově zaznamenávání slouží k lepšímu řízení času, sledování produktivity a optimalizaci využití pracovního času pro jednotlivce i organizace.

2.2 Spouštěče měření času

Jedním z cílů této práce je prozkoumat různá řešení pro spouštěče měření času. Nejjednodušším spouštěčem je samozřejmě ruční zapnutí nějaké formy časovače v samotné aplikaci, která pro měření času slouží. To ale umí kdejaké již existující řešení a přidanou hodnotou této práce



Obrázek 2.1 Clockify – iOS aplikace pro měření času [1]

by měly být možnosti, jak spouštění uživateli ulehčit. Tyto možnosti se dají rozdělit do dvou kategorií – fyzické a softwarové.

2.2.1 Fyzické

Za fyzické spouštěče měření času lze považovat jakoukoli formu fyzického ovladače, kterou může uživatel ovládat a tím spouštět časovač měření času.

Častým typem fyzického spouštěče je nějaká forma takzvaného platónského tělesa (krychle, osmistěn, dvanáctistěn, atd.), které může uživatel otáčet. Podle toho, na kterou stranu ho položí, se spustí časovač s danými parametry. Různé strany tělesa mohou sloužit například pro identifikování toho, na kterém projektu uživatel zrovna pracuje.

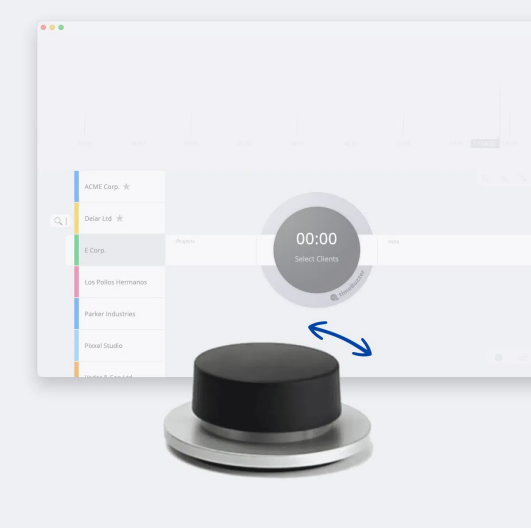
Mezi fyzické spouštěče patří například tyto produkty:

- **TIMEFLIP:** Dvanáctistěnné těleso, na které si uživatel může nalepovat cokoli, co bude identifikovat odpracovaný čas. Těleso se propojí s mobilní aplikací, která poté spravuje naměřený čas. [2]
- **TIMEULAR:** Stejný princip jako *TIMEFLIP*, akorát používá osmistěnné těleso. Jednotlivé

strany jdou také přizpůsobovat nálepkami. [3]

- **timeBuzzer:** Fyzické tlačítko, které lze umístit na stůl a zapojit do počítače. Stisknutí tlačítka otevře okno měřící aplikace, otočením tlačítka lze vybrat projekt a opětovným stisknutím se začne daný projekt měřit. [4]

(a) TIMEFLIP – Kostka a aplikace [2]



(b) timeBuzzer – tlačítko a aplikace [4]

Obrázek 2.2 TIMEFLIP a timeBuzzer

Všechny ze zmíněných produktů poskytují veřejné API pro aplikace, se kterými komunikují [5] [6] [7]. Pokud by tedy bylo potřeba produkty propojit s vlastní aplikací, která by mohla naměřený čas spravovat, bylo by potřeba, aby aplikace komunikovala s těmito nástroji. Pořád by byl potřeba prostředník, kterým by byla aplikace daného produktu. Výjimkou je v těchto produktech pouze *TIMEFLIP*, který poskytuje veřejný protokol pro BLE komunikaci [8]. Tento produkt by tedy mohl komunikovat s novou aplikací napřímo.

Pokud by nová aplikace měla umožňovat budoucí propojení s jakýmkoli dalším hardwarovým produktem, tak by také měla poskytovat veřejné API, které by takovou komunikaci umožňovalo. Vývoj takových produktů by mohl být předmětem návrhu pro budoucí vylepšení aplikace.

2.2.2 Softwarové

Za softwarové spouštěče měření času lze považovat určité formy integrace a automatizace v rámci zařízení, na kterém aplikace běží. V případě aplikace na platformě iOS to mohou být následující možnosti:

- **Automatizace podle polohy:** Aplikace by mohla reagovat na polohu uživatele a spouštět nebo vypínat časovač na základě informace, kde se uživatel nachází. Uživatel by si mohl například nastavit určitá místa, kde by chtěl, aby se automaticky spustil časovač. Mohl by se zapnout přímo s konkrétními přednastavenými parametry (projekt, klient, ...), nebo by se jen mohlo ukázat upozornění, aby si uživatel časovač zapnul sám, s parametry, které zadá. Podobně by aplikace mohla reagovat i na opuštění nějakého místa – například pokud uživatel opustí místo, které má označeno jako práci, tak by dostal upozornění, zda si nechce vypnout časovač.

- **Automatizace podle času:** Jednoduchá forma automatizace by mohla fungovat na základě času. V přednastavených časech by se mohl zapnout nebo vypnout časovač s danými parametry, nebo by aplikace mohla uživatele jen upozornit.
- **Automatizace podle kalendáře:** Užitečná by také mohla být integrace s kalendářem uživatele. Podle naplánovaných událostí by se časovač mohl automaticky zapínat, přepínat, nebo vypínat, s parametry z kalendáře. Nebo opět pouze uživatele upozornit, zda si kvůli nějaké události nechce měření času aktualizovat.
- **Automatizace podle režimu soustředění:** Systém iOS umožňuje od verze 15.0 uživatelům používat takzvané *Režimy soustředění* [9]. Uživatel si může konfigurovat různé režimy a nastavovat, jaké mu v tomto režimu budou chodit upozornění, jaké aplikace může používat, kdo mu může volat, apod. Od iOS verze 16 také Apple umožňuje vývojářům přizpůsobovat chování aplikace podle toho, jaký režim soustředění je zapnutý [10]. Toho by se dalo vhodně využít například tak, že by se mohl automaticky spouštět časovač (nebo by uživatel mohl být upozorněn) podle toho, do jakého režimu soustředění se uživatel právě přepnul.
- **Automatizace pomocí zkratek:** Apple nabízí uživatelům s iOS verze 13.0 nebo novější aplikaci *Zkratky* [11], která umožňuje tvorbu vlastních automatizujících procesů [12]. Tento způsob automatizace umožňuje uživatelům i vývojářům velkou míru přizpůsobitelnosti – zkratky lze propojovat s dalšími aplikacemi, s hlasovým asistentem a s mnoha různými akcemi. Všechny předchozí způsoby automatizace lze také nějakým způsobem vytvořit pomocí zkratek. [13]

2.3 Existující systémy pro měření a správu odpracovaného času

Systémů pro měření a správu odpracovaného času existuje mnoho. Existuje dokonce několik různých článků o tom, které systémy jsou nejlepší a jaké je jejich srovnání [14] [15]. Následující výběr příkladů je učiněn podle těchto článků, podle osobní preference a podle počtu stažení na iOS platformě. U každého příkladu jsou také rozebrány možnosti importu dat, od kterých se mohou odvíjet požadavky na export dat z nové aplikace.

2.3.1 Clockify

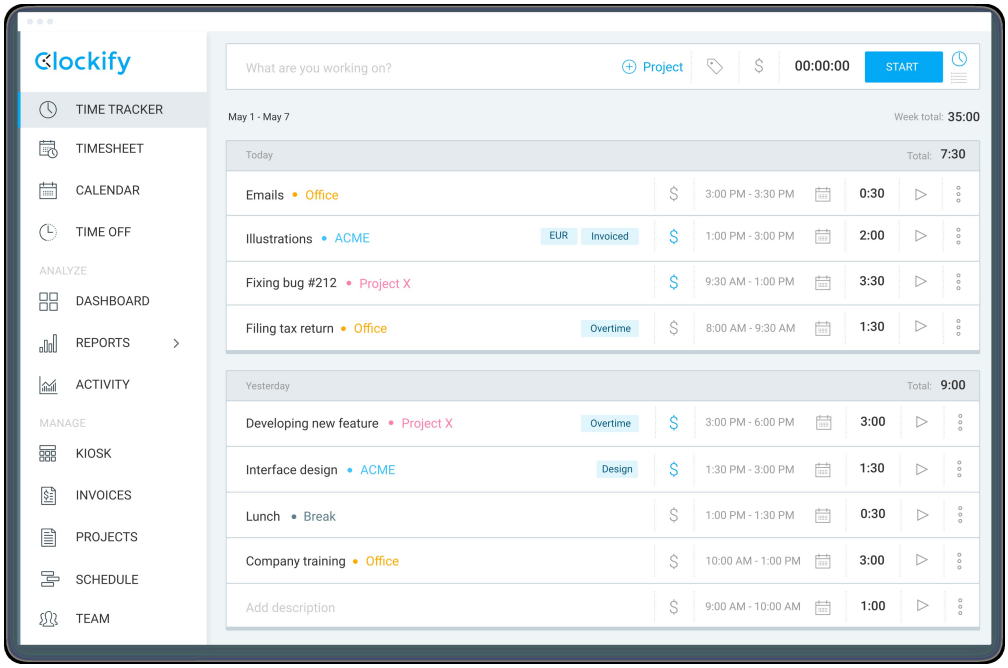
Clockify je jednoduchá a rozšířená aplikace primárně určená pro měření a správu odpracovaného času. Mobilní aplikace má více než 1 milion stažení [16] a celý systém používají miliony lidí [17].

V článku Forbes je hodnocena jako celkově nejlepší aplikace pro měření času pro rok 2024. Má široké spektrum funkcionalit – umožňuje měření a správu času pro různé projekty, klienty a zařízení. Mimo měření času nabízí také sledování prezence pro mzdy a účetnictví, optimalizaci produktivity zaměstnanců, měření vykazovatelného času a sdílení pokroku na projektech s klienty. [14]

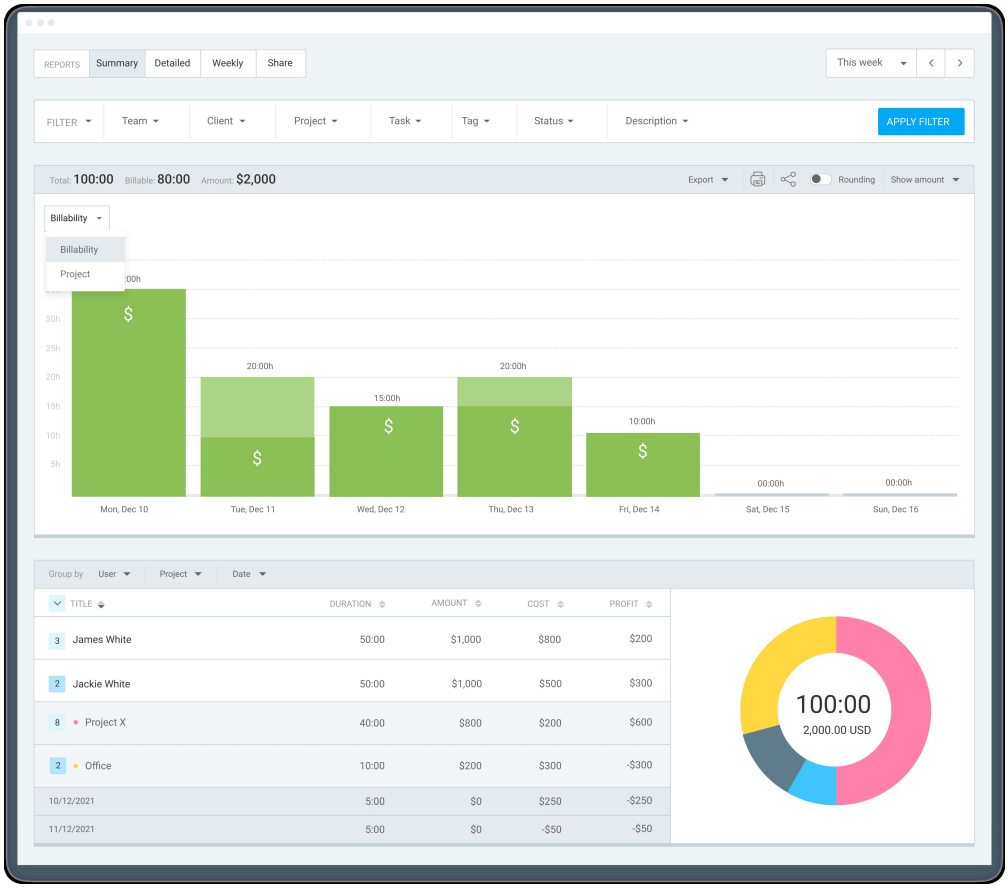
Clockify nabízí aplikace pro mnoho platforem: Desktopovou aplikaci pro Mac, Windows a Linux, doplněk webového prohlížeče pro Chrome, Firefox a Edge, mobilní aplikaci pro iOS a Android a nakonec sdílené Kioskové řešení, na kterém si uživatelé mohou zapisovat příchody, přestávky a další. [18]

V základní variantě je používání aplikace zdarma. Clockify ale nabízí i placené plány, které nabízí přidané funkce, jako audity naměřených časů, používání přestávek a další. Tyto placené plány mohou stát od $3.99 měsíčně za uživatele až po $11.99 měsíčně za uživatele. [19]

Clockify nabízí dobře zdokumentované API pro své služby [21]. Napojení na funkce systému lze tedy jednoduše udělat i z vlastního řešení. Do Clockify lze také data importovat z CSV tabulek, jsou-li data vhodně naformátována [22].



Obrázek 2.3 Časovač na měření času v aplikaci Clockify [20]



Obrázek 2.4 Reporty v aplikaci Clockify [20]

Mezi nevýhody systému Clockify Forbes uvádí, že projekty nejdou označit jako dokončené, a že shrnující reporty mohou být zpočátku matoucí. Celkově Clockify ale vnímá jako nejlepší řešení pro pokrytí celé škály funkcionalit.

Na obrázku 2.3 je vidět hlavní obrazovka ve webové verzi aplikace – časovač. Uživatel si zde může časovač ručně zapnout a vypnout, přidělit projekt, přidat značky, označit vykazovatelnost a přidat popisek. Také má možnost přidat odpracovaný čas ručně, tedy ne pomocí časovače. K tomu stačí jen napsat čas začátku a konce. Pod ovládacím panelem časovače potom uživatel vidí seznam svých již zadaných časových záznamů, které zde může libovolně upravovat.

Dále je na obrázku 2.4 vidět obrazovka určená pro reporty. Uživatel si zde může vybrat časový úsek, pro který chce report vypracovat, a může si nastavit filtry, podle kterých chce časové vstupy filtrovat (tým, klient, projekt, úkol, značka, stav a popisek). Po aplikování filtru uživatel následně uvidí report své práce vyhovující zadaným parametrům, který si může exportovat ve formátech PDF, CSV a Excel.

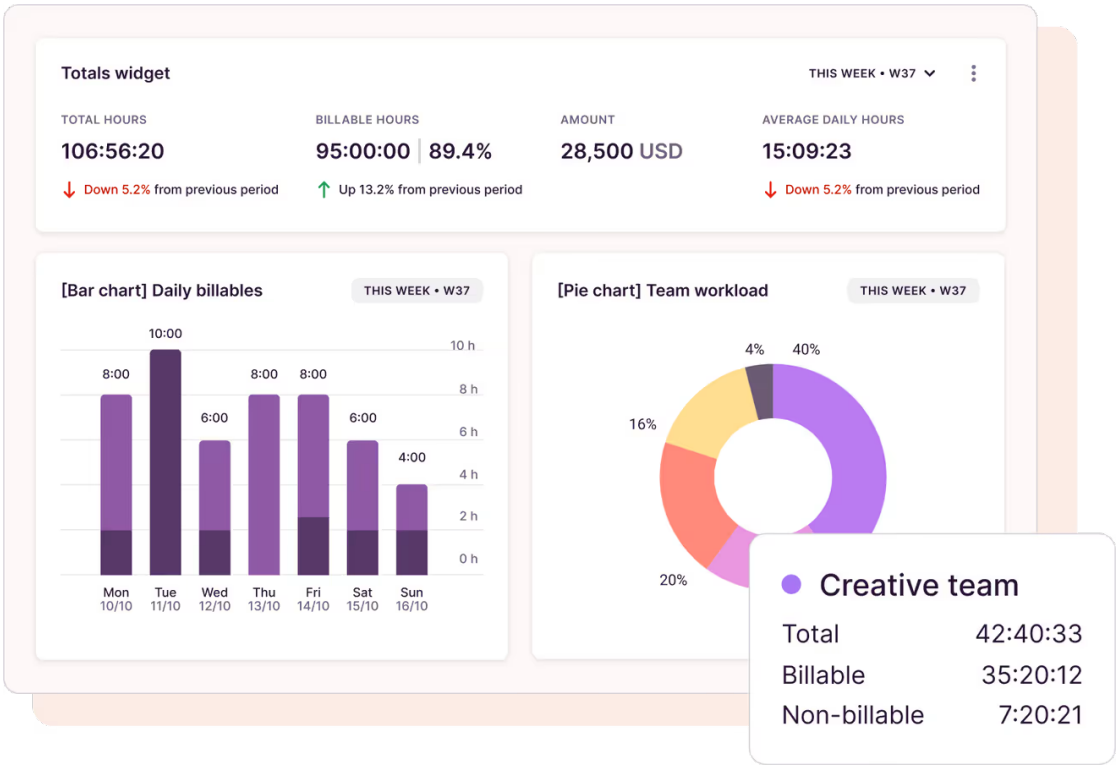
2.3.2 Toggl Track

Toggl track je další poměrně rozšířená aplikace pro měření a správu času. Na platformě iOS má více než 1 milion stažení [23].

Forbes tuto aplikaci považuje za nejlepší pro malé týmy, protože nabízí plán zdarma pro týmy do 5 uživatelů a nabízí neomezený počet klientů a projektů. [14]

Aplikace nabízí určitou formu automatické detekce toho, že uživatel nemá zapnutý časovač, i když pracuje, nebo automatickou detekci naopak toho, že uživatel už nepracuje, ale časovač má zapnutý. Aplikace nabízí detailní denní, týdenní a měsíční reporty.

Toggl Track také pokrývá více platforem: Webovou aplikací, mobilní aplikací pro iOS a Android a desktopovou aplikací pro Windows a Mac. [24]



Obrázek 2.5 Aplikace Toggl Track [24]

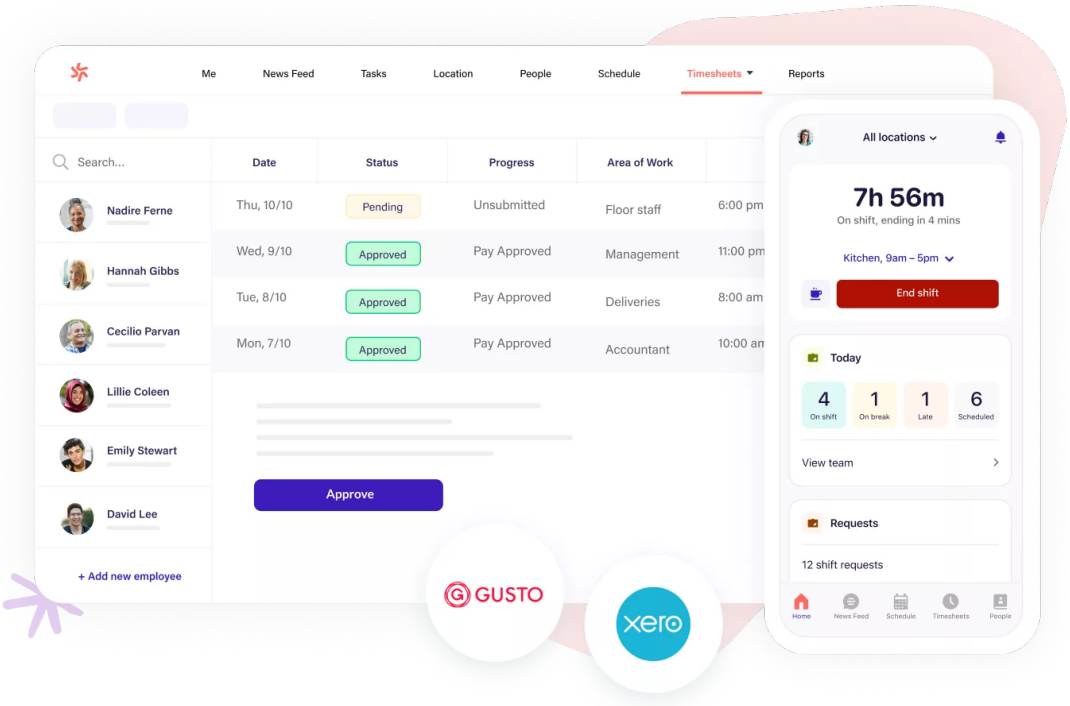
Toggl Track také nabízí možnost importu dat z CSV tabulky, ve velmi podobném formátu jako Clockify [25]. A stejně jako Clockify nabízí vlastní otevřené API [26].

2.3.3 Deputy

Deputy je další aplikací, která umožňuje měření a správu odpracovaného času. Jejím hlavním účelem je však plánování směn zaměstnanců. Je uvedena jako jeden z příkladů proto, aby šlo nahlédnout na problém měření času i z jiného úhlu, než striktně měření času primárně za účelem vykazování, fakturování nebo analýzy efektivity. Je také velice rozšířenou aplikací – počty stažení na mobilních telefonech překračují 5 milionů [27].

Právě plánování směn zaměstnanců je hlavním důvodem, proč Forbes tuto aplikaci ve svém seznamu zmiňuje. Aplikace totiž nabízí plánování neomezeného počtu směn za měsíc. Uživatelé se mohou přihlásit k aplikacím pro plánování a pro prezenci zvlášť. I neplacený plán umožňuje automatické plánování směn, což šetří čas manažerům. Toto plánování umožňuje i započítání obědových pauz, přestávek a dalšího, přímo do plánu směn, podle příslušných zákonů. [14]

Deputy nabízí aplikaci pro měření odpracovaného času pro mobilní platformy iOS a Android.



Obrázek 2.6 Aplikace Deputy [28]

Stejně jako předchozí systémy, Deputy nabízí možnost importu dat z CSV tabulky [29]. Vzhledem k trochu jinému byznysovému modelu této aplikace ale tyto data musí na rozdíl od předchozích systémů obsahovat dodatečná data, jako místo, pauzy na oběd, a další. A ani u této aplikace nechybí otevřené API [30].

2.4 Vývoj mobilních aplikací pro systém iOS

Tato sekce se věnuje několika klíčovým tématům souvisejícím s vývojem aplikací pro zařízení s operačním systémem iOS. Tato témata shrnuje a poskytuje tím základ pro diskuzi o vývoji

aplikací pro tento systém.

2.4.1 Historie a vývoj iOS platformy

Historie operačního systému iOS začala v roce 2007, kdy byl představen a začal se prodávat první mobilní telefon od společnosti Apple, kterým byl iPhone. V tomto telefonu byl od výroby nainstalován operační systém, který ještě v tu dobu nenesl název *iOS*, ale *iPhone OS*. Název *iOS* se začal používat až později od verze systému 4. Přestože z dnešního pohledu chybělo v systému mnoho funkcí, se kterými si dnes mobilní telefony spojujeme (například možnost instalace aplikací třetích stran), tak v tu dobu to byl velký posun. Věci jako *Multitouch screen*, vizuální znázornění hlasové schránky, nebo integrace s *iTunes*, byly ohromnou výhodou. Mezi předinstalované aplikace patřil kalendář, fotky, fotoaparát, poznámky, prohlížeč webu *Safari*, e-mailový klient, telefon a *iPod* (který se později rozdělil na aplikace pro hudbu a aplikace pro videa).

Obrázek 2.7 První model mobilního telefonu iPhone [31]

O rok později, v roce 2008, byla představena nová generace mobilního telefonu s názvem iPhone 3G, se kterým také vznikla nová verze operačního systému iOS (iPhone OS) 2.0. Hlavní novinkou této nové verze byl nový obchod s aplikacemi třetích stran, *App Store*. Až 500 aplikací bylo v obchodě k dispozici v době, kdy byl zpřístupněn uživatelům.

V roce 2009 poté s novým iPhonem 3GS přišla verze iOS (iPhone OS) 3, která přinesla hlavně vylepšení, jako možnost používat kopírování a vkládání, vyhledávání přes *Spotlight*, podporu MMS a možnost natáčet videa (do té doby iPhone uměl pouze pořizovat fotky). Také to byla první verze operačního systému, která fungovala i na nových tabletech od společnosti Apple – na iPadech. První iPad byl představen v roce 2010.

V roce 2010 byl představen nový iPhone 4 a s ním verze operačního systému 4. Od této verze se začal používat název *iOS*, který nahradil do té doby používaný *iPhone OS*. Toto nové pojmenování mělo sjednocovat názvosloví pro více zařízení – iPhone, iPad a iPod. iPhone 4 byl poměrně velkou změnou, protože disponoval novým hranatým designem, a systém iOS 4 přinesl novinky, které tento operační systém pomalu začaly rýsovat do dnešní podoby. Mezi novinky patřil *FaceTime*, *multitasking*, *iBooks*, organizování aplikací do složek, osobní hotspot nebo sdílení dat pomocí *AirPlay* a *AirPrint*. Také to byla první verze iOS, která nepodporovala všechny dosud vydané iPhony, protože nebyla kompatibilní s prvním iPhonem.

O rok později, s iPhonem 4S, byl představen iOS verze 5. Apple v ní reagoval na rostoucí trend bezdrátovosti a *cloud computing* představením několika nových funkcí a platforem. Mezi

ně patřil třeba iCloud nebo synchronizace s iTunes přes Wi-Fi. Také vznikla platforma *iMessage* a začalo se používat oznamovací centrum.

Dalším modelem byl iPhone 5, který poprvé změnil velikost displeje a zvětšil ji z 3,5 palce na 4. S tímto modelem přišla verze iOS 6, kterou doprovázelo mnoho problémů. V této verzi Apple představil hlasového asistenta *Siri*, který i přes to, že ho později předčila konkurence, byl poměrně zásadní novinkou. Dále Apple v nové verzi představil nové aplikace pro mapy, na které do té doby používal řešení od firmy Google. Tyto mapy od začátku obsahovaly mnoho chyb a celkově byly považovány za dost nedokončené, což způsobilo ve firmě mnoho problémů.

V roce 2013, s novým modelem iPhone 5S, přišla verze iOS 7. Tato verze zásadním způsobem změnila vzhled uživatelského rozhraní, které se na začátku od uživatelů netěšilo příliš dobrému přijetí, ale po několika vylepšeních a po tom, co si uživatelé na nový vzhled zvykli, problémy ustaly. Mezi nové funkce této verze se řadí sdílení dat s dalšími uživateli přes *AirDrop*, používání operačního systému v displejích automobilů přes *CarPlay*, ovládací centrum, nebo nový způsob odemykání zařízení pomocí otisku prstu – *Touch ID*.

Obrázek 2.8 Nový vzhled iOS verze 7 [32]

S dalším rokem a novým modelem iPhone 6 přišla verze iOS 8. iPhone 6 opět zásadně změnil svůj vzhled, ale v operačním systému se moc zásadních změn nedělo. Apple se v této verzi soustředil hlavně na nové funkce. Mezi ně se řadí *Apple Music*, placení přes *Apple Pay*, cloudové úložiště *iCloud Drive*, sdílení práce mezi Apple zařízeními přes *Handoff*, rodinné sdílení a další.

Rok 2015 přinesl iPhone 6S a iOS 9. V této verzi se Apple soustředil hlavně na optimalizaci a stabilitu, příliš nových funkcí představeno nebylo – pouze nasvícení pro noční používání *Night shift*, režim nízké spotřeby a možnost používat veřejný testovací beta program.

Dalším modelem byl v roce 2016 iPhone 7 s iOS 10. Opět nepřinesl moc nových funkcí, mimo například možnosti mazat původní nainstalované aplikace.

O rok později poté s iPhonem 8 přišel iOS 11. V této verzi se Apple soustředil převážně na nové funkcionality pro iPad, které se jeho použití snažily přiblížit k plnohodnotnému počítači. Nově přibyly funkce jako podpora tužky *Apple Pencil*, podpora více otevřených oken a pracovních ploch nebo prohlížeč souborů. Apple ale na závěr představil ještě další nový iPhone – iPhone X. Jednalo se o další velký redesign, kdy se výrazně zvětšila plocha displeje, zmizelo přední tlačítko a vzniklo ověření uživatele pomocí obličeje – *Face ID*.

V roce 2018 Apple představil iPhone XS a levnější variantu XR, se kterými přišel iOS 12, kde se jednalo opět jen o nepatrná vylepšení.

S dalším rokem přišel iPhone 11 a iOS 13. Tento systém už se rozdělil – pro zařízení iPad nyní vznikl oddělený *iPadOS*. V iOS 13 Apple představil možnost tmavého režimu pro celý systém, nové možnosti bezpečnosti a soukromí a s tím související možnost přihlášení přes Apple.

V roce 2020 s iPhonem 12 a iOS 14 Apple přidal několik menších změn a vylepšení, jako *Widgety* na domovské obrazovce.

Podobně tomu bylo i v roce 2021, kdy Apple s novým iPhonem 13 a iOS 15 přidával velké množství menších vylepšení, primárně do vlastních aplikací.

S rokem 2022 přišel iPhone 14 a iOS 16, který přinesl sadu vylepšení a nových funkcí pro zamčenou obrazovku, spolu s dalšími vylepšeními.

Poslední znatelný update iOS přišel v roce 2023 s iPhonem 15 a iOS 17. a jak už bylo v posledních letech zvykem, tak tato aktualizace přinášela větší množství menších vylepšení a aktualizací, které ale už nijak zásadně neměnily systém a interakci s ním. [33]

Za těchto 17 let vývoje se operační systém iOS dostal do stavu, kdy ho v současnou chvíli používá 1,46 miliard aktivních uživatelů. Mobilních telefonů iPhone se od roku 2007 prodalo 2,3 miliard. [34]

2.4.2 Vývojové nástroje a prostředí

Jak již bylo zmíněno v předchozí sekci, obchod s aplikacemi *App Store* byl uživatelům dostupný od roku 2008. Ve stejnou dobu také Apple zpřístupnil vývojářům iPhone SDK ve svém vývojovém prostředí *Xcode*. Toto vývojové prostředí od Applu již existovalo od roku 2003, kdy vzniklo jako rebranding původního vývojového prostředí *Project Builder*. S příchodem App Storu Apple umožnil všem vývojářům v Xcodu vyvíjet aplikace pro iPhone OS od verze 2.0 pomocí zmíněného iPhone SDK.

Vývojové prostředí Xcode je jednotné integrované vývojové prostředí, které slouží pro vývoj aplikací pro všechny platformy společnosti Apple. Lze stáhnout zdarma z obchodu *Mac App Store* nebo ze stránek společnosti Apple. Instalace vývojového prostředí obsahuje i instalaci dalších pomocných nástrojů, jako je simulátor zařízení (iPhone, iPad, ...), nástroje příkazové řádky, a další. [35]

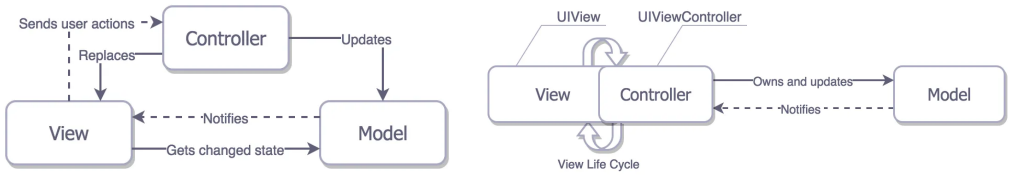
Pro nativní vývoj iOS aplikací se dříve používal programovací jazyk *Objective-C* [36] a pro tvorbu uživatelského rozhraní knihovna *UIKit*. V roce 2014 Apple přidal podporu svého nového programovacího jazyku *Swift* [37] a v roce 2019 přidal podporu nové deklarativní knihovny pro tvorbu uživatelského rozhraní *SwiftUI* [38].

2.4.3 Architektura aplikací

Jako u každého softwarového projektu, architektura je důležitou součástí jeho návrhu. K architekturám mobilních aplikací existuje mnoho přístupů. Například u Android aplikací sám Google doporučuje, jaké architektonické vzory by měly být používány a doporučuje architektury pro tyto aplikace [39]. Apple přímo žádné architektury pro aplikace cílící na iOS platformu nedoporučuje.

Dokumentace pro iOS platformu ale obsahuje mnoho ukázek kódu věnujících se konkrétním tématům (například [40]), kde je vždy nějaký náznak architektury připraven. V těchto ukázkách se vyskytuje architektura *Model-View-Controller* (zkráceně MVC). Na rozdíl od tradiční MVC architektury se ta používaná Applem mírně liší, jelikož tradiční MVC architektura není moc dobře aplikovatelná na moderní iOS vývoj. Jak lze vidět na obrázcích 2.9, v tradičním MVC by mělo *View* být beze stavu a měl by ho pouze překreslovat *View Controller*. Toto je sice možné v iOS aplikaci implementovat, ale nedává to moc smysl, protože všechny tyto 3 entity jsou hluboce provázané, což dramaticky snižuje jejich opětovné použití. Při použití v iOS aplikaci je *Controller* mediátorem mezi *View* a *Model*, kteří o sobě navzájem nic nevědí. Jelikož je ale *View Controller* zpravidla velice provázán s *View*, tak vzniká potřeba psát masivní *View Controller*. Proto se v iOS vývoji často referuje na MVC jako na *Massive View Controller*.

Architektura MVC je ale v ukázkách kódu od Applu používána pravděpodobně převážně proto, že se hodí pro výstižné a krátké ukázky kódu – obsahuje totiž minimální *overhead*. Pro potřeby větších projektů začíná být nedostačující. Nelze nijak testovat logiku prezentování,

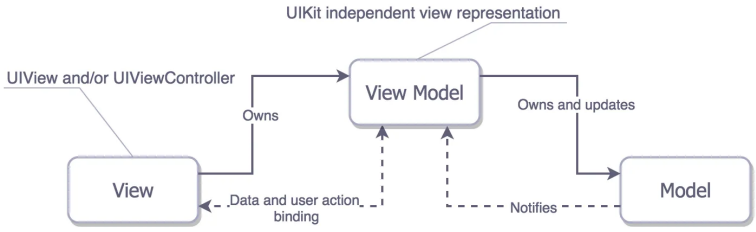


(a) Tradiční MVC architektura **(b)** MVC architektura v iOS aplikacích

Obrázek 2.9 Architektura Model-View-Controller [41]

a *View* také nelze pomocí jednotkových testů otestovat. Jediné, co lze testovat, je *Model*. a velkou nevýhodou je právě zmíněný masivní *View Controller*, který pro složitější obrazovky ztrácí přehlednost.

Velmi rozšířenou architekturou v iOS vývoji je *Model-View-ViewModel* (zkráceně MVVM). *View* a *Model* zastávají stejnou funkci jako v MVC, a mediátorem je zde *View Model*. *View* je v tomto případě konkrétní obrazovka a/nebo *View Controller*. *View Model* je nezávislý na knihovně uživatelského rozhraní, což umožňuje jeho lepší testovatelnost. Drží stav obrazovky, vyvolává změny pro *Model* a aktualizuje se podle něj. Vizuální reprezentace architektury MVVM lze nahlédnout v obrázku 2.10.



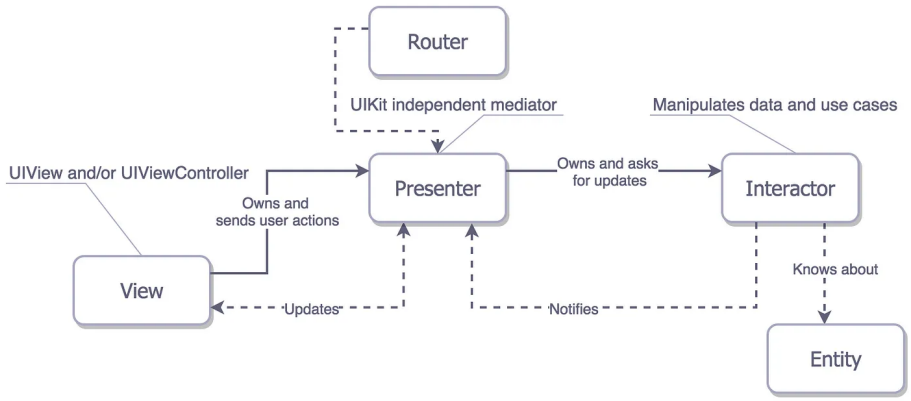
Obrázek 2.10 Architektura MVVM [41]

Další rozšířenou architekturou v iOS aplikacích je *Viper*. Tato architektura nepochází ze skupiny MV(X). *Viper* se snaží vzít rozdělovaní odpovědností o krok dále – obsahuje 5 vrstev:

- **View:** Zastává stejnou funkcionalitu jako v MV(X), tedy *View* a/nebo *View Controller*.
- **Interactor:** Obsahuje byznysovou logiku související s daty, jako tvorba instancí nebo načítání dat ze serveru.
- **Presenter:** Obsahuje byznysovou logiku UI, ale je nezávislý na UI knihovně. Volá metody *Interactoru*.
- **Entities:** Čisté objekty dat.
- **Router:** Odpovědný za přechody mezi *Viper* moduly.

Tato architektura je poměrně volná v tom, co bude konkrétní *Viper* modul reprezentovat. Může to být jedna samotná obrazovka, ale může to být celá část aplikace. Lze si všimnout několika rozdílů od architektur ze skupiny MV(X):

- Model (interakce s daty) je přesunut do *Interactoru* pomocí *Entities*.
- Pouze povinnosti UI reprezentace *Controlleru*/*View Modelu* je přesunuta do *Presenteru*, ale ne byznysová logika s daty.
- *Viper* explicitně adresuje odpovědnost navigace, kterou řeší *Router*.



Obrázek 2.11 Architektura *Viper* [41]

Vizualizace architektury *Viper* lze nahlédnout v obrázku 2.11. [41]

Existuje také rozšíření architektury MVVM o navigační logiku, takzvaná MVVM-C, kde písmeno C představuje *Flow Coordinator*. Tato vrstva je odpovědná za navigační tok, vytváří instance *View* a/nebo *View Controller* a prezentuje je, předává data mezi těmito instancemi a obsluhuje akce uživatele. Struktura *Flow Coordinatoru* je vhodně navržena tak, aby se dala také dobře testovat. [42]

Jednotlivé architektury mají také své aktualizované varianty pro nově používanou deklarativní UI knihovnu *SwiftUI*. Všechny ze zmíněných architektur se dají vhodně použít i s touto knihovnou. Ve *SwiftUI* už se jednotlivé obrazovky nerozdělují na *View* a *View Controller*, ale zůstává zde pouze *View*. *SwiftUI* ale ze své podstaty jako deklarativní knihovna přináší nové možnosti, jak k architektuře aplikace přistupovat.

Jednou z novějších architektur, které využívají výhody deklarativních UI knihoven, jako je *SwiftUI*, je *Model-View-Intent* (zkráceně MVI). *View* a *Model* reprezentují stejné vrstvy jako doposud a *Intent* reprezentuje nějaký úmysl vyvolat určitou akci. Tím může být například kliknutí uživatele. Konkrétní *Intent* poté pomocí nějaké byznysové logiky aktualizuje stav, podle kterého se následně aktualizuje *View*. [43]

Všechny výše popsané architektury pracují s daty pomocí nějakého *Modelu* (*Viper* pomocí *Interactoru*). Práce s daty ale obvykle není jednoduchá logika, která by si zasloužila pouze zmínku o tom, že se o ní stará nějaký *Model*. Může se totiž jednat o poměrně komplikovanou doménovou a byznysovou logiku, stahování dat ze serveru, cache dat, práce s databází, a další. V tomto ohledu je namístě diskutovat o nějaké obecnější architektuře aplikace, ne pouze o prezentační logice, tedy primárně o logice uživatelského rozhraní. Obecným zvykem je aplikace rozdělovat do třívrstvé architektury. Tyto vrstvy se mohou nazývat různě, ale obvykle jde o následující:

- **Prezentační vrstva:** Logika uživatelského rozhraní, interakce s uživatelem, obrazovky, ...
- **Doménová/byznysová vrstva:** Byznysová a doménová logika.
- **Datová vrstva:** Práce s daty (komunikace se serverem, databáze, ...)

Rozšířená architektura, vhodná pro iOS aplikace, která definuje strukturu aplikace, je *Clean Architecture*. Tato architektura je navržena pro dobrou testovatelnost, rozdělení odpovědnosti, a vyhovění dalším zašlým návrhovým vzorům týkajících se návrhu softwaru. V této architektuře se jednotlivé vrstvy skládají z následujících částí:

- **Prezentační vrstva:** *View*, *View Controllery*, *View Modely*, a další, podle dané architektury (MVC, MVVM, Viper, ...). Tato vrstva má závislost na doménové vrstvě a volá její *Use Cases*.

- **Doménová vrstva:** Rozhraní *Use Cases* a jejich implementace, rozhraní *Repositories*, které *Use Cases* používají a doménové objekty. *Use Cases* obsahují byznysovou logiku aplikace a datovou logiku nechávají na *Repositories*.
- **Datová vrstva:** Implementace *Repositories*, rozhraní *Providers* a implementace *Providers*. Implementace repozitářů jsou nezávislé na konkrétních knihovnách třetích stran, obsahují logiku s daty. Implementace *Providers* už jsou závislé na konkrétních knihovnách. Jednotlivé *Providers* se mohou starat například o práci s konkrétní databází nebo s konkrétní knihovnou pro serverovou komunikaci.

V *Clean Architecture* se také pracuje s *Dependency Injection*, což řeší logiku toho, které konkrétní implementace budou dosazeny jednotlivým rozhraním (*Use Cases, Repositories, Providers*). [44]

2.5 Cross-platformní a multi-platformní možnosti

V současném prostředí mobilního vývoje se rozhodnutí ohledně vhodné platformy pro tvorbu aplikací stává klíčovým faktorem pro úspěch projektu. Existuje široká škála přístupů, které vývojáři mohou zvolit, od tradičních nativních řešení až po moderní multi-platformní a cross-platformní frameworky. Tato sekce se zaměřuje na analýzu těchto možností s ohledem na tvorbu mobilní aplikace pro zaznamenávání odpracovaného času. Zhodnocuje jejich výhody, nevýhody a vhodnost pro konkrétní aplikaci. Tento přístup umožní lépe pochopit, jakým směrem se vydat při návrhu a implementaci projektu.

Pro rozdíl mezi cross-platformním a multi-platformním vývojem neexistuje přesná definice a v různých zdrojích lze nalézt různé interpretace (např. [45] a [46]). V kontextu vývoje pro mobilní aplikace si budeme tyto pojmy vykládat následovně:

- Cross-platformní vývoj představuje řešení, ve kterých sdílený kód běží na koncových platformách přímo přes nějakou formu abstrakce nebo interpretace.
- Multi-platformní vývoj představuje řešení, ve kterých se sdílený kód přímo kompiluje do nativního kódu specifického pro každou platformu.

2.5.1 Nativní vývoj

Nativní vývoj pro platformu iOS znamená vytváření mobilních aplikací přímo v jazyce *Swift* nebo *Objective-C* s využitím oficiálních nástrojů poskytovaných společností Apple, jako je Xcode IDE a iOS SDK. Tento přístup umožňuje vytvořit aplikaci, která je optimalizovaná pro konkrétní operační systém a využívá všech funkcí a výhod, které iOS nabízí.

Výhody nativního vývoje pro iOS spočívají především v plné integraci s ekosystémem Apple, což zajišťuje vysokou kvalitu, rychlost a stabilitu aplikací. Vývojáři mají přístup ke kompletní sadě nástrojů, dokumentace a podpory přímo od výrobce platformy, což usnadňuje vývoj a řešení potíží. Díky nativnímu přístupu je možné vytvářet aplikace s vysokou výkonností a možnostmi, které jsou na míru prostředí iOS.

Nevýhody nativního vývoje spočívají v tom, že vyžaduje znalost specifických jazyků a nástrojů pro každou platformu (*Swift*/*Objective-C* pro iOS, *Kotlin*/*Java* pro Android), což může zvýšit náklady na vývoj a časovou náročnost. Navíc nativní přístup vyžaduje oddělený vývoj pro každou platformu, což může být neefektivní pro projekty s omezeným rozpočtem nebo krátkým časovým rámcem.

Nativní vývoj je ideální pro projekty, které se zaměřují na plné využití možností iOS platformy, jako jsou výkonné aplikace nebo aplikace s náročnějším uživatelským rozhraním. Také je vhodný pro aplikace, které potřebují maximální stabilitu a bezpečnost, například bankovní aplikace nebo aplikace pro zpracování citlivých údajů. Pro vývojáře, kteří chtějí mít plnou kontrolu nad každým aspektem aplikace a využít všech funkcí, které iOS nabízí, je nativní vývoj nejlepší volbou.

2.5.2 Cross-platformní vývoj

Cross-platformní vývoj se zaměřuje na tvorbu mobilních aplikací, které mohou běžet na více než jedné platformě (např. iOS a Android) s využitím jediného kódu a jednoho vývojového prostředí. Tento přístup umožňuje vývojářům sdílet co nejvíce kódu mezi různými platformami, čímž snižuje náklady a zjednodušuje správu aplikací pro různé zařízení.

Mezi hlavní výhody cross-platformního vývoje patří efektivita a rychlost vývoje díky sdílení kódu mezi platformami. Vývojáři mohou využít frameworky jako *Flutter* [47], *React Native* [48] nebo *Xamarin* [49], které umožňují psát kód v populárních jazycích (např. *JavaScript*, *TypeScript*, *Dart*, *C#*) a následně ho spouštět na různých platformách. Tento přístup také usnadňuje aktualizace a údržbu aplikací, protože změny se projeví na všech podporovaných platformách současně.

Nevýhody cross-platformního vývoje se mohou projevit ve snížené flexibilitě a omezení přístupu k některým pokročilým funkcím a knihovnám specifickým pro danou platformu. Rovněž může docházet k omezení rychlosti nebo výkonu aplikace v porovnání s nativními aplikacemi. Další nevýhodou může být závislost na externích frameworcích a jejich aktualizacích.

Cross-platformní vývoj je ideální pro projekty, které vyžadují rychlou dostupnost na více platformách s omezenými zdroji. Hodí se pro aplikace s jednodušším uživatelským rozhraním, obsahově orientované aplikace (např. novinky, e-commerce), nebo pro firemní aplikace, které nevyžadují specifické funkce jednotlivých platforem. Tento přístup je také vhodný pro malé a střední projekty, kde je důležitá rychlá a efektivní tvorba aplikace pro více platforem.

2.5.3 Multi-platformní vývoj

Multi-platformní vývoj se liší od cross-platformního vývoje tím, že přímo kompiluje zdrojový kód do nativního kódu specifického pro každou platformu, místo aby spoléhal na vrstvu abstrakce nebo interpretaci. To znamená, že výsledná aplikace běží jako nativní aplikace bez vrstvy prostřednictvím frameworku. Typickým příkladem multi-platformního vývoje je použití jazyka jako *Kotlin* pro Android a *Kotlin/Native* pro iOS, které se kompilují do nativního kódu pro obě platformy. Tato technologie se nazývá *Kotlin Multiplatform* [50].

Výhody multi-platformního vývoje zahrnují možnost sdílet větší část kódu mezi různými platformami a zároveň dosahovat výkonnosti a funkčnosti nativních aplikací. To znamená, že vývojáři mohou využívat specifické funkce a knihovny pro každou platformu, aniž by se museli spoléhat na obecné abstraktní vrstvy. Tento přístup také umožňuje lepší optimalizaci výkonu a přístup k pokročilým funkcím operačních systémů.

Nevýhody multi-platformního vývoje mohou zahrnovat větší složitost a náročnost vývoje oproti čistě cross-platformním frameworkům. Každá platforma může vyžadovat jiné postupy a úpravy, ačkoli základní kód je sdílen. Některé specifické funkce nebo optimalizace pro konkrétní platformu mohou být obtížnější dosáhnout pomocí multi-platformního přístupu než s nativním vývojem.

Multi-platformní vývoj je vhodný pro projekty, které vyžadují vysokou výkonnost a přístup k nativním funkcím a knihovnám, ale zároveň potřebují sdílet co nejvíce kódu mezi platformami. Ideální je pro rozsáhlejší projekty nebo aplikace, které potřebují plnou integraci s operačním systémem, ale zároveň chtějí minimalizovat duplicitu práce a zjednodušit správu a údržbu kódu. Multi-platformní vývoj je také vhodný pro situace, kdy je důležitá konzistence a shoda funkcí mezi různými verzemi aplikace na různých platformách.

2.6 Backendová řešení pro mobilní aplikace

Backendová část mobilních aplikací hraje klíčovou roli v poskytování dat, zpracování požadavků a správě uživatelských účtů a obsahu. Tato sekce se zaměřuje na analýzu různých backendových

řešení a technologií, které jsou vhodné pro podporu mobilních aplikací. Při výběru správného backendového řešení je důležité pochopit úlohu, kterou backend hraje v kontextu mobilního prostředí a identifikovat faktory, které ovlivňují výběr a implementaci.

Backendová část mobilní aplikace zajišťuje komunikaci mezi klientem (mobilní aplikací) a serverem, zpracování dat, autentizaci uživatelů, a další potřebné funkce. Tento centrální prvek infrastruktury zabezpečuje efektivní a spolehlivé fungování mobilních aplikací, přičemž umožňuje sdílení dat mezi různými zařízeními a poskytuje uživatelům personalizovaný a interaktivní zážitek.

Při výběru backendového řešení pro mobilní aplikaci je důležité zvážit několik faktorů. Patří mezi ně škálovatelnost a výkon serveru, podpora pro bezpečnostní standardy a autentizaci, možnosti správy uživatelských dat, a také kompatibilita s konkrétními požadavky a technologiemi použitými ve frontendové části aplikace. Dále je důležité zvážit náklady na provoz, údržbu a rozvoj backendového systému v průběhu životního cyklu aplikace.

Existuje široká škála backendových technologií a frameworků, které lze použít při vývoji mobilních aplikací. Od tradičních serverových platforem a frameworků až po moderní cloudová řešení a služby. Každá možnost má své vlastní výhody a nevýhody. Důkladná analýza těchto možností je zásadní k správnému výběru backendové architektury pro konkrétní mobilní aplikaci.

2.6.1 Backend as a Service (BaaS) a jiná delegovaná řešení

Tato podsekce se zaměřuje na možnosti, kdy jsou části backendové infrastruktury mobilní aplikace outsourcovány na externí poskytovatele služeb, jako je *Firebase*, *AWS Amplify* nebo *Parse*. Tento přístup umožňuje vývojářům rychle nasadit backendovou část aplikace bez nutnosti spravovat a udržovat vlastní serverovou infrastrukturu.

Jednou z hlavních výhod BaaS je urychlení vývoje aplikace a snížení nákladů a komplexity provozování backendu. Poskytovatelé BaaS nabízejí hotová řešení pro autentizaci uživatelů, ukládání dat, notifikace, analýzu chování uživatelů a řadu dalších funkcí, což umožňuje vývojářům zaměřit se více na samotnou funkcionalitu aplikace a méně na infrastrukturální detaily.

Další výhodou je škálovatelnost a výkon poskytovaných služeb, který může být optimalizován poskytovatelem a automaticky přizpůsobován podle potřeb aplikace. Tento přístup je obzvláště užitečný pro malé a střední projekty s omezenými zdroji nebo pro projekty, které potřebují rychle nasadit MVP (Minimum Viable Product) bez investice do vlastní infrastruktury.

Nevýhodou použití BaaS může být omezení v možnostech škálování a přizpůsobení, zejména u složitějších aplikací nebo projektů se specifickými požadavky na infrastrukturu. Dále může být problémem závislost na externím poskytovateli služeb a jejich možná změna podmínek nebo dostupnosti.

Jako příklad BaaS lze uvést *Firebase* od společnosti Google [51]. Firebase poskytuje širokou škálu služeb, včetně realtime databáze, autentizace, cloudového úložiště, notifikací, analýzy a mnoha dalších. Tato platforma je oblíbená mezi vývojáři pro svou jednoduchost a širokou nabídku poskytovaných funkcí, což umožňuje rychlý vývoj a nasazení mobilních aplikací s minimálním úsilím na správu backendové infrastruktury.

Dalšími příklady BaaS služeb jsou *AWS Amplify* od společnosti Amazon [52] a *Parse* [53], který je open-source frameworkem pro tvorbu aplikací. Tyto služby nabízejí podobné funkce jako *Firebase* a umožňují vývojářům využít hotová řešení pro backendovou část svých mobilních aplikací bez nutnosti psát a spravovat vlastní kód pro serverovou stranu.

2.6.2 Vývoj vlastního backendu

Tato sekce se zaměřuje na možnosti vytvoření a implementace vlastního backendového řešení pro podporu mobilních aplikací. Tento přístup umožňuje vývojářům plnou kontrolu nad backendovou infrastrukturou a její přizpůsobení specifickým požadavkům aplikace.

Jednou z hlavních výhod vývoje vlastního backendu je možnost plného přizpůsobení infrastruktury a funkcí podle konkrétních potřeb mobilní aplikace. Vývojáři mají kontrolu nad

škálovatelností, bezpečností a výkonem backendu, což je zvláště důležité pro aplikace s vyššími nároky na bezpečnost, správu dat nebo specifické obchodní požadavky.

Další výhodou je snížená závislost na externích poskytovatelích služeb a jejich změnách v podmínkách či dostupnosti. Vlastní backend umožňuje také integraci s existujícími systémy a infrastrukturou v organizaci, což může být podstatné pro firemní aplikace nebo projekty s komplexními integračními požadavky.

Nevýhodou vývoje vlastního backendu může být zvýšená náročnost a časová zátěž vývoje a údržby. Vývojáři si musí sami implementovat všechny potřebné funkce backendu, včetně autentizace, ukládání dat, správy uživatelů a dalších. To může vést ke zvýšeným nákladům a časovému zpoždění při nasazení aplikace na trh.

Mezi nejpoužívanější řešení pro vývoj vlastního backendu patří frameworky jako *Node.js* [54] s frameworky *Express* [55] nebo *NestJS* [56] pro *JavaScript*/*TypeScript*, *Ruby on Rails* [57] pro *Ruby*, *Django* [58] pro *Python* nebo *Spring Boot* [59] pro *Javu*. Tyto frameworky nabízejí komplexní sadu nástrojů pro rychlý vývoj a nasazení backendové aplikace s podporou různých funkcí, včetně routování, databázového přístupu, autentizace a dalších.

Každý z těchto frameworků má své výhody a nevýhody. Například *Node.js* s *Express* je velmi populární pro svou rychlost a flexibilitu, zatímco *Spring Boot* je oblíbený pro svou robustnost a škálovatelnost v prostředí *Java* [60]. Výběr správného frameworku závisí na preferencích vývojářů, technologických požadavcích a cílech aplikace.

2.6.3 Databáze

Implementace databáze je dalším důležitým prvkem vývoje vlastního backendu pro mobilní aplikace, neboť poskytuje úložiště pro data, která aplikace zpracovává a uchovává. Existuje několik možností pro implementaci databází v rámci backendového prostředí, které se liší podle typu databáze a potřeb aplikace.

Mezi nejpoužívanější řešení patří relační databáze, jako je *Oracle Database* [61] nebo *PostgreSQL* [62], a také *noSQL* databáze, jako je *MongoDB* [63] nebo *Redis* [64]. Relační databáze jsou založené na modelu relačního datového skladu s použitím *SQL* (Structured Query Language) pro manipulaci s daty. Tyto databáze jsou vhodné pro aplikace, které vyžadují komplexní transakční operace a silné zajištění integrity dat. Na druhou stranu *noSQL* databáze jsou navrženy tak, aby byly flexibilnější a lépe škálovatelné pro různé typy dat. Jsou ideální pro aplikace, které mají různorodá data a vyžadují rychlý a flexibilní přístup k nim.

Výhody relačních databází zahrnují silnou konzistenci dat, vysokou integritu a možnost provádět komplexní dotazy pomocí SQL. Na druhou stranu *noSQL* databáze nabízejí vyšší škálovatelnost, flexibilitu datového modelu a lepší výkon pro určité aplikace s velkým objemem dat. [65]

Výběr správného typu databáze závisí na specifických požadavcích a charakteristikách projektu. Pro aplikace s potřebou silné konzistence a transakcí jsou vhodné relační databáze, zatímco pro aplikace s velkým objemem různorodých dat a vyššími požadavky na škálovatelnost jsou vhodnější *noSQL* databáze.

Výběr implementace databázového řešení lze také outsourcovat na externí poskytovatele služeb BaaS. Jak bylo zmíněno v sekci 2.6.1, poskytovatelé jako *Firebase* [51] nebo *AWS Amplify* [52] již implementují databázové řešení, které lze v rámci těchto platforem využít. Existují také řešení, která poskytují pouze databázi jako externí řešení. Tato řešení jsou někdy nazývána DBaaS (Database as a Service) [66]. Příkladem tohoto řešení je například *MongoDB Atlas* [67], právě pro databázi *MongoDB*.

V implementaci vlastního backendu pro mobilní aplikace je důležité pečlivě zvážit výběr databázového řešení, aby byly splněny požadavky na výkon, škálovatelnost a bezpečnost aplikace. Zvážení výhod a nevýhod jednotlivých databázových systémů pomůže zajistit optimální implementaci a správu dat pro mobilní aplikaci pro zaznamenávání odpracovaného času.

2.7 Závěr analýzy

V této kapitole byly probrány všechny informace potřebné pro navazující návrh mobilní aplikace pro zaznamenávání odpracovaného času.

Co se týče domény problému, byla provedena rešerše spouštěčů měření času, ze které mohou být vhodně navrženy požadavky pro aplikaci na propojení s nimi. Dále byla také provedena rešerše existujících systémů pro zaznamenávání odpracovaného času, ze které lze také vhodným způsobem navrhnout požadavky na integraci s těmito systémy.

Dále se kapitola věnovala vývoji mobilních aplikací pro systém iOS, různým možnostem cross-platformního a multi-platformního vývoje a možnostem, jak přistupovat k backendovým řešením podporující mobilní aplikace. Tyto informace by měly být plně postačující pro vhodný návrh funkcionalit aplikace, jejího uživatelského rozhraní, vnitřní architektury a nástrojů k tomu potřebným.

Kapitola 3
Návrh

Tato kapitola se věnuje návrhu aplikace, tedy technologické architektury celé platformy a jednotlivých implementací, návrhu funkcionalit a uživatelskému rozhraní aplikace. V následujících sekcích budou rozebírány jednotlivé navržené funkcionality a jejich rozhraní. Nejprve je ale potřeba si stanovit vzhledový styl aplikace.

Pro návrh uživatelského rozhraní aplikace byl použit nástroj *Figma* [68], dále byly použity zdroje z šablony *Apple Design Resources* [69]. Zdroj všech návrhů lze nahlédnout v [70].

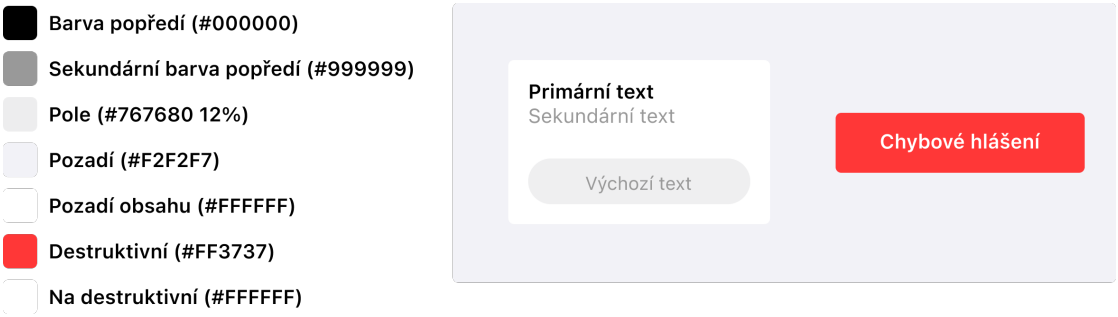
3.1 Vzhledový styl aplikace

Aplikace cílí na platformu iOS, což bude důležitou součástí jejího návrhu. Apple definuje rozsáhlou příručku pro návrh uživatelského rozhraní pro platformu iOS [71] a návrh aplikace se touto příručkou bude v mnoha ohledech řídit.

Každá aplikace má nějaký svůj vzhledový styl, který definuje základní barvy, které bude aplikace používat, vzhledy tlačítek, textových polí, fontů a dalšího. Následující definice těchto prvků vychází převážně z osobní preference, která se soustředí spíše na jednoduchost a nepříliš velkou výraznost rozhraní. Cílem tedy bude se přiblížit systémovému vzhledu platformy iOS a přidat vlastní mírný vzhledový jazyk.

3.1.1 Barvy

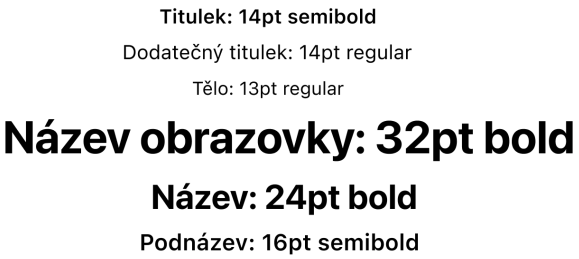
Základní návrh barev aplikace lze nahlédnout v obrázku 3.1. Barvy jsou navrženy tak, aby vždy vznikl dostatečný kontrast mezi barvou pozadí a barvou popředí.



Obrázek 3.1 Vzhledový styl aplikace – Barvy

3.1.2 Fonty

Základní návrh fontů lze nahlédnout v obrázku 3.2. Daný font bude vždy používat systémovou rodinu fontů, tedy obvykle *San Francisco* (SF). Jednotlivé velikosti jsou pouze referenční, protože aplikace by měla podporovat dynamické fonty a reflektovat tak škálování uživatele. Daná velikost je tedy velikost pro výchozí nastavení škálování textu.



Obrázek 3.2 Vzhledový styl aplikace – Fonty

3.1.3 Prvky

Návrh prvků rozhraní vychází z již definovaných barev a fontů, lze jej nahlédnout v obrázku 3.3.

Obrázek 3.3 Vzhledový styl aplikace – Prvky

Na všechny ostatní interakční prvky, jako prvky navigace, seznamy, alerty, a další, bude využito systémových prvků. Tím bude nejlépe vyhověno vzhledové příručce firmy Apple, pouze budou upravené některé barvy těchto prvků, aby ladily k vzhledu aplikace.

3.1.4 Název a ikona

Návrh chytlavého názvu bývá obvykle složitá věc. Pro tuto aplikaci byl zvolen název *Trackee* (anglicky [tra · ki]), který je odvozen z anglického pojmu *Time tracking*, což představuje měření odpracovaného času. Koncovka *-ee* je také poslední dobou častou volbou pro názvy různorodých aplikací, jako *Spendee* [72], *Fondee* [73] a další. Pod tímto názvem není registrována žádná ochranná známka [74], ani není veden žádný záznam u správce české domény [75].

Ikona aplikace také neprocházela nijak složitým procesem návrhu, byl pouze použit systémový symbol časovače na pozadí s barvami aplikace popředí a pole. Ikonku lze nahlédnout v obrázku 3.4.

Obrázek 3.4 Ikona aplikace

3.2 Funkcionality aplikace a jejich uživatelské rozhraní

Aplikace bude primárně sloužit pro zaznamenávání odpracovaného času. Je tedy potřeba, aby každý uživatel měl možnost si vytvářet vlastní záznamy a další data, která budou propojena pouze s ním, a ke kterým bude mít přístup pouze on sám. Toto je obvyklý případ užití mobilní aplikace, který ze své podstaty vyžaduje nějakou formu vytvoření uživatelského účtu, se kterým budou data propojena, a jeho autentizace. Nejobvyklejším způsobem autentizace je autentizace pomocí e-mailu a hesla. Tento způsob je i poměrně jednoduchý z hlediska implementace a spousta poskytovatelů BaaS (Backend as a Service, vizte 2.6.1) tento způsob autentizace implementuje.

3.2.1 Přihlášení a registrace

Tato funkcionalita bude sloužit pro následující případy užití:

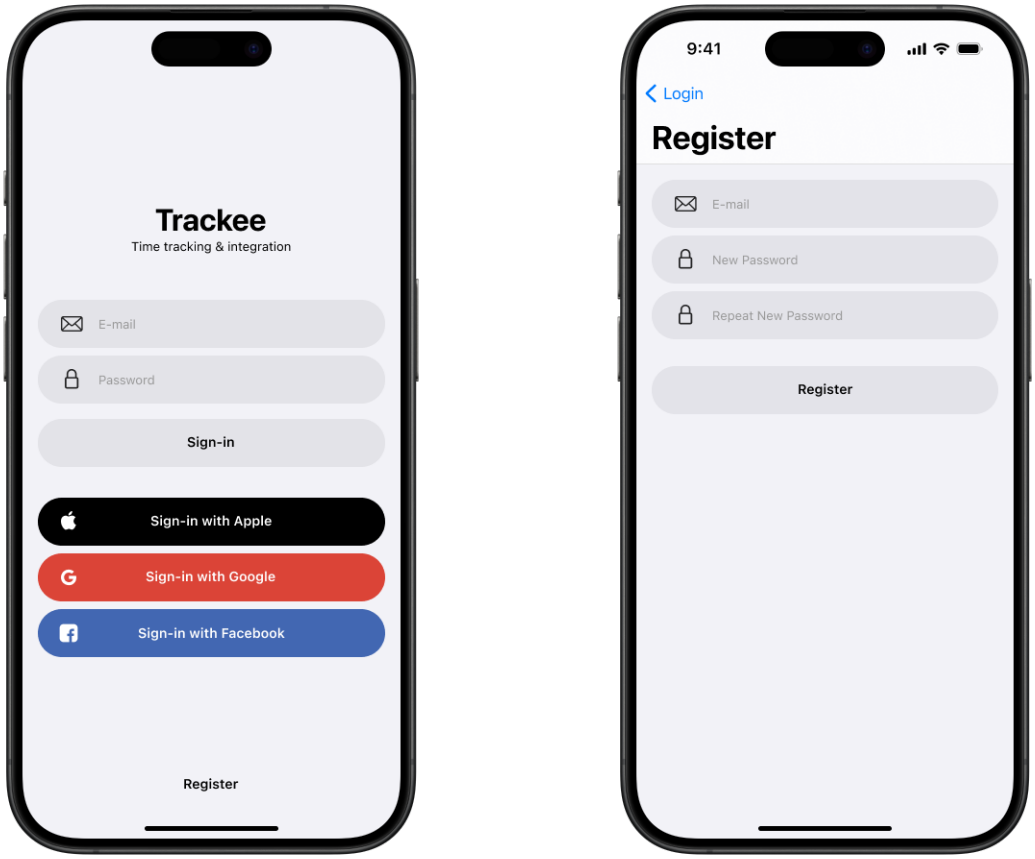
- **Přihlášení** – Uživatel se přihlásí pomocí e-mailu a hesla.
- **Registrace** – Uživatel si vytvoří účet pomocí e-mailu a hesla.

Přihlašovací obrazovka bude obsahovat pouze nadpis, pole pro vyplnění e-mailové adresy, hesla, primární tlačítko pro přihlášení a sekundární tlačítko pro registraci. Obrazovka pro registraci, která se otevře po kliku na tlačítko pro registraci, poté bude od uživatele potřebovat také jen e-mail a heslo, které je ale zvykem napsat dvakrát, aby se snížila šance, že se v něm vyskytl překlep. Obrazovky přihlášení a registrace lze nahlédnout v obrázku 3.5. Úspěšné přihlášení a registrace uživatele přesměruje na hlavní obrazovku aplikace.

3.2.2 Lišta karet a časovač

Funkcionalita časovače bude sloužit pro následující případy užití:

- **Zapnutí časovače** – Uživatel spustí časovač a začne tak měřit odpracovaný čas.
- **Výběr projektu** – Uživatel přiřadí projekt k záznamu, který chce měřit nebo vytvořit.
- **Zadání popisu** – Uživatel zadá popis k záznamu, který chce měřit nebo vytvořit.



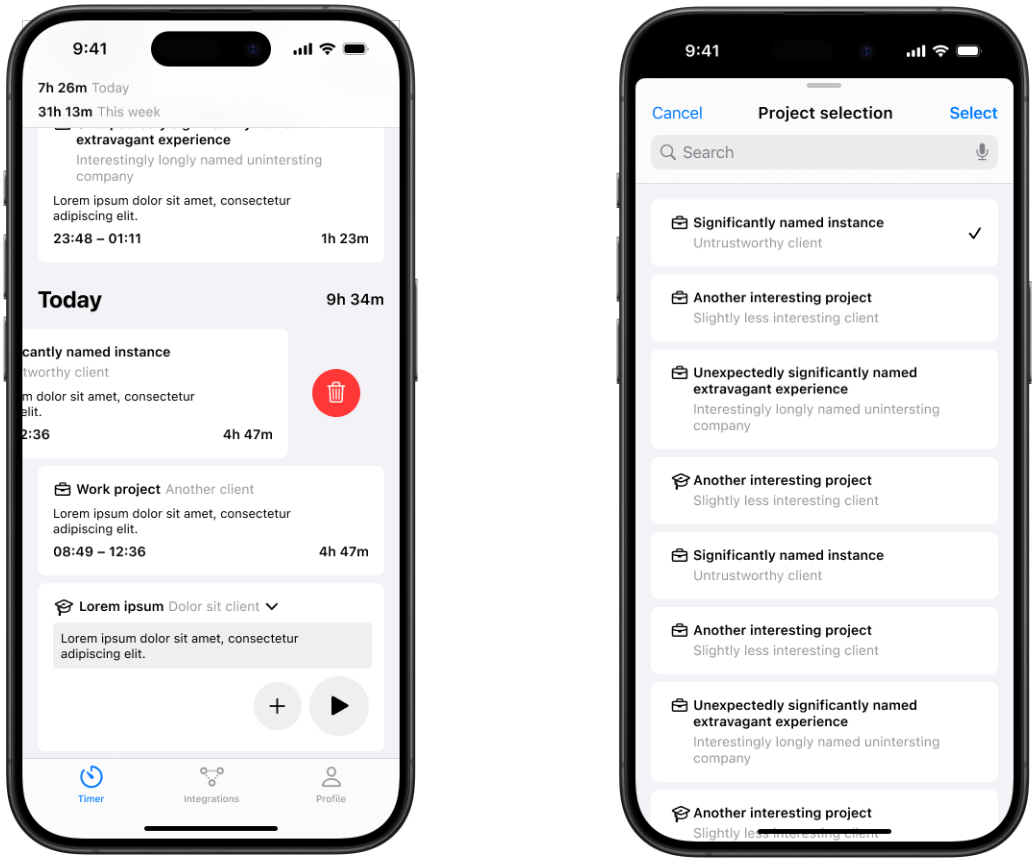
(a) Přihlášení **(b)** Registrace

Obrázek 3.5 Onboarding

- **Zastavení časovače a uložení záznamu** – Uživatel zastaví časovač a uloží tím záznam, který do zastavení měřil.
- **Manuální přidání záznamu** – Uživatel vytvoří nový záznam podle zadání času začátku a konce.
- **Odstranění záznamu** – Uživatel odstraní časový záznam z historie.
- **Zobrazení historie záznamů** – Uživatel si prohlédne libovolné záznamy z historie.
- **Zobrazení shrnutí odpracovaných hodin** – Uživatel si zobrazí souhrn odpracovaných hodin daného dne nebo současného týdne.

Navigace mezi hlavními obrazovkami aplikace bude řešena pomocí lišty karet, jelikož se jedná o častý a doporučený způsob, jak navigovat mezi vzájemně exkluzivními částmi obsahu [76]. Hlavní obrazovkou bude přehled, na kterém bude uživatel moct ovládat časovač, a kde uvidí historii svých časových záznamů, seřazenou od nejnovějších po nejstarší. Jelikož pro uživatele je nejjednodušší dosáhnout na ovládací prvky, které jsou ve spodní části displeje, bude ovládání časovače umístěno ve spodní části obrazovky, a časové záznamy se budou řadit nad ním. Návrh této obrazovky lze nahlédnout na obrázku 3.6a.

Ovládací panel pro časovač může mít různé varianty, jak bude vypadat, podle toho, v jakém je stavu. Návrh počítá se dvěma možnostmi, jak půjde přidávat časové záznamy do historie:



(a) Časovač **(b)** Výběr projektu

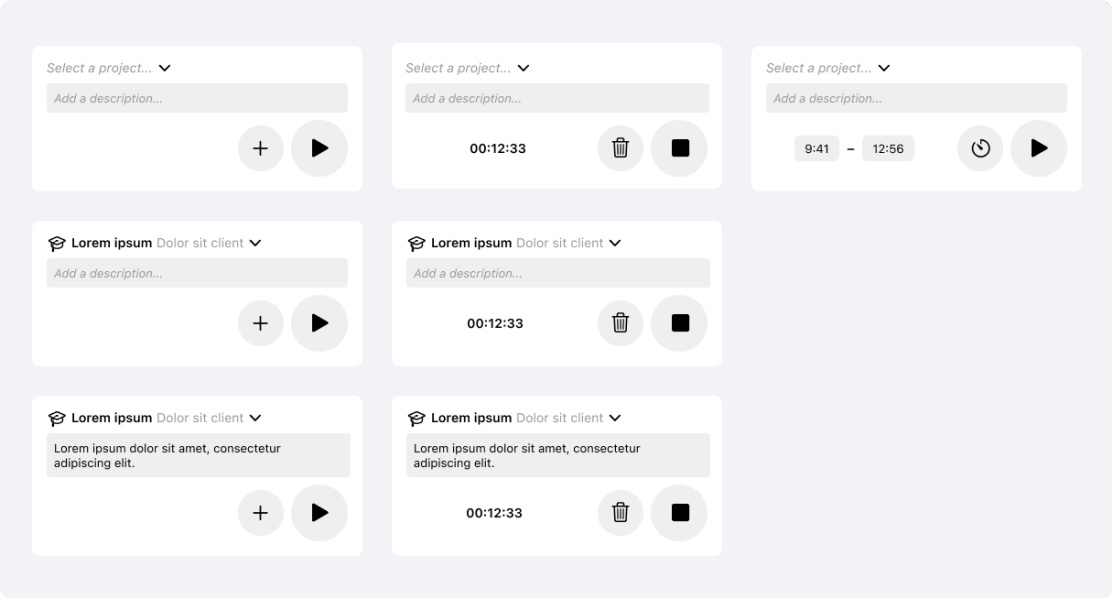
Obrázek 3.6 Hlavní obrazovka

- Pomocí časovače – uživatel zapne časovač, když bude chtít začít měření, a poté ho vypne, když bude měření chtít ukončit, čímž se automaticky uloží záznam se zadanými vlastnostmi.
- Ručně – uživatel ručně zadá začátek a konec záznamu a poté časový záznam uloží.

Ovládací panel půjde přepínat mezi těmito dvěma stavy pomocí vedlejšího tlačítka. Hlavní ovládací tlačítko bude vypínat/zapínat časovač, pokud bude přepnut do stavu časovače, nebo bude přidávat ruční záznam, pokud bude v ručním stavu. V ovládacím panelu časovače bude také možné vybrat projekt a popis, které budou k danému záznamu přiděleny. Klik na volbu projektu otevře novou obrazovku, která umožní vyhledávání v projektech a výběr projektu, jak lze vidět na obrázku 3.6b. Různé stavy časovače (časovač/manuální, zapnutý/vypnutý, vyplněný/nevyplněný, atd.) lze nahlédnout v obrázku 3.7.

Na obrázku 3.6a lze dále v horní části obrazovky vidět souhrn časových záznamů za aktuální den a týden. Uživatel tak uvidí, kolik času již odpracoval v daný den i týden. Časové záznamy v historii budou také seskupeny podle dnů – každý den bude mít nadpis s datem a součtem odpracovaného času za daný den. Jednotlivé záznamy také půjde mazat pomocí posuvného gesta.

Vzhledem k tomu, že v historii se může časem nacházet mnoho záznamů, měla by tato obrazovka podporovat stránkování, tedy funkci, že nebude ze zdroje načítat všechny záznamy najednou, ale pouze nějaký kus (stránku), a postupně může načítat další, pokud si to uživatel bude přát. Pokud se během načítání objeví chyba, tak se na této obrazovce ovládací panel časovače



Obrázek 3.7 Různé stavy ovladače pro časovač

ani historie záznamů vůbec nezobrazí – zobrazí se pouze popis chyby a tlačítko pro opakování pokusu o načtení. Pokud uživatel zatím žádné záznamy mít nebude, tak nebude potřeba zobrazovat nějakou explicitní formu prázdné obrazovky – pouze bude ve spodní části obrazovky ovládací panel a nad tím prázdno.

U obrazovky pro výběr projektu bude také explicitní chybový stav, který ukáže popis chyby včetně tlačítka pro opakování. Na této obrazovce už ale bude potřeba definovat i prázdný stav, aby se zobrazila instrukce, že uživatel nemá vytvořené žádné projekty a musí si je vytvořit na místě k tomu určeném (bude navrženo dále). Prázdná data lze ale ještě rozdělit do dvou kategorií – prázdná data kvůli tomu, že uživatel žádné projekty nemá, nebo prázdná data kvůli tomu, že jeho vyhledávání neodpovídá žádný projekt. Pro tyto dva stavy je potřeba použít rozdílné texty pro uživatele.

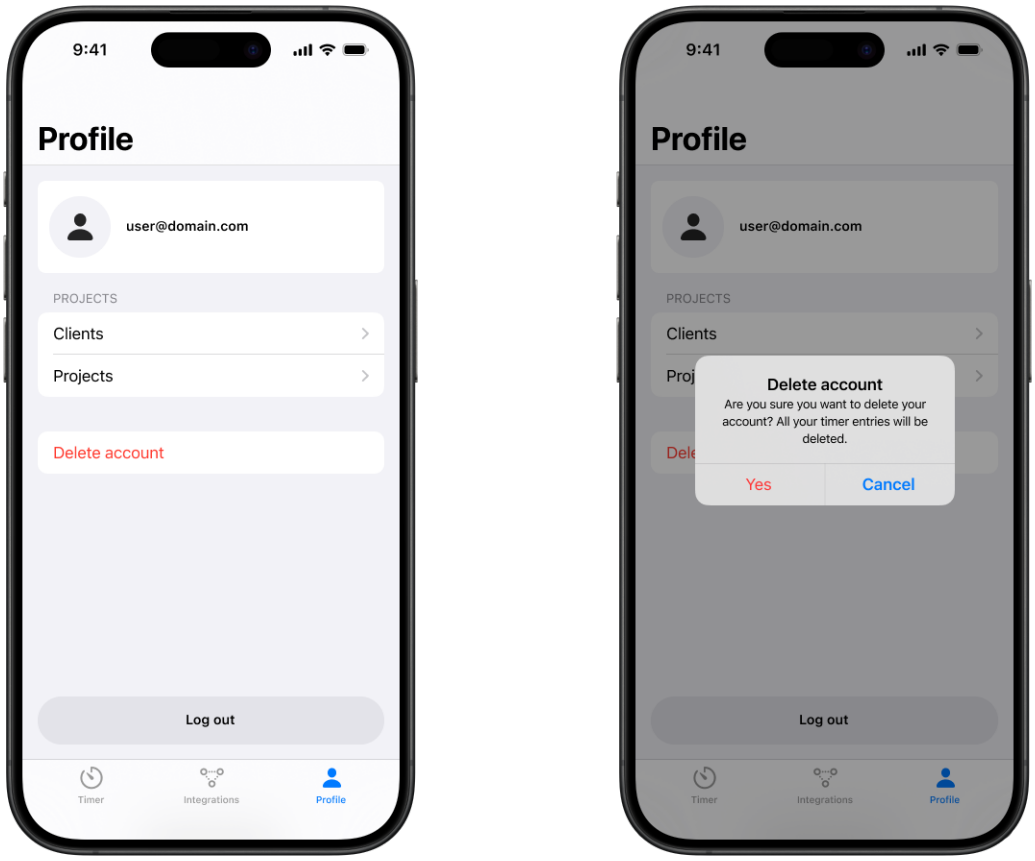
3.2.3 Profil uživatele

Profil uživatele bude sloužit pro následující případy užití:

- **Zobrazení e-mailové adresy přihlášeného uživatele** – Uživatel si zobrazí e-mailovou adresu účtu, na kterém je přihlášen.
- **Odhlášení** – Uživatel se odhlásí ze svého účtu.
- **Odstranění účtu** – Uživatel smaže svůj účet.
- **Zobrazení projektů** – Uživatel si zobrazí projekty, které má přiřazené.
- **Zobrazení klientů** – Uživatel si zobrazí klienty, které má přiřazené.
- **Tvorba nebo úprava projektu** – Uživatel si vytvoří projekt nebo upraví existující přiřazený projekt.
- **Tvorba nebo úprava klienta** – Uživatel si vytvoří klienta nebo stejně jako u projektu upraví existujícího přiřazeného klienta.

- **Odstranění projektu** – Uživatel odstraní projekt, s ním i všechny časové záznamy, které pod projekt spadají.
- **Odstranění klienta** – Uživatel odstraní klienta, s ním i všechny projekty a časové záznamy, které pod klienta spadají.

Poslední kartou v navigační liště aplikace bude karta s Profilem. Uživatel zde bude mít základní přehled a akce týkající se jeho účtu, jak lze nahlédnout na obrázku 3.8a. Uživatel se odsud dostane do seznamu klientů a seznamu profilů, dále si může pomocí tlačítka svůj účet smazat, nebo se odhlásit, což ho vrátí zpět na přihlašovací obrazovku 3.5a. Mazání účtu je nevratná akce, která vymaže spoustu dat spojených s uživatelem, je tedy potřeba alespoň ukázat ověřující dialog, který lze nahlédnout na obrázku 3.8b.



(a) Přehled **(b)** Smazání účtu

Obrázek 3.8 Profil

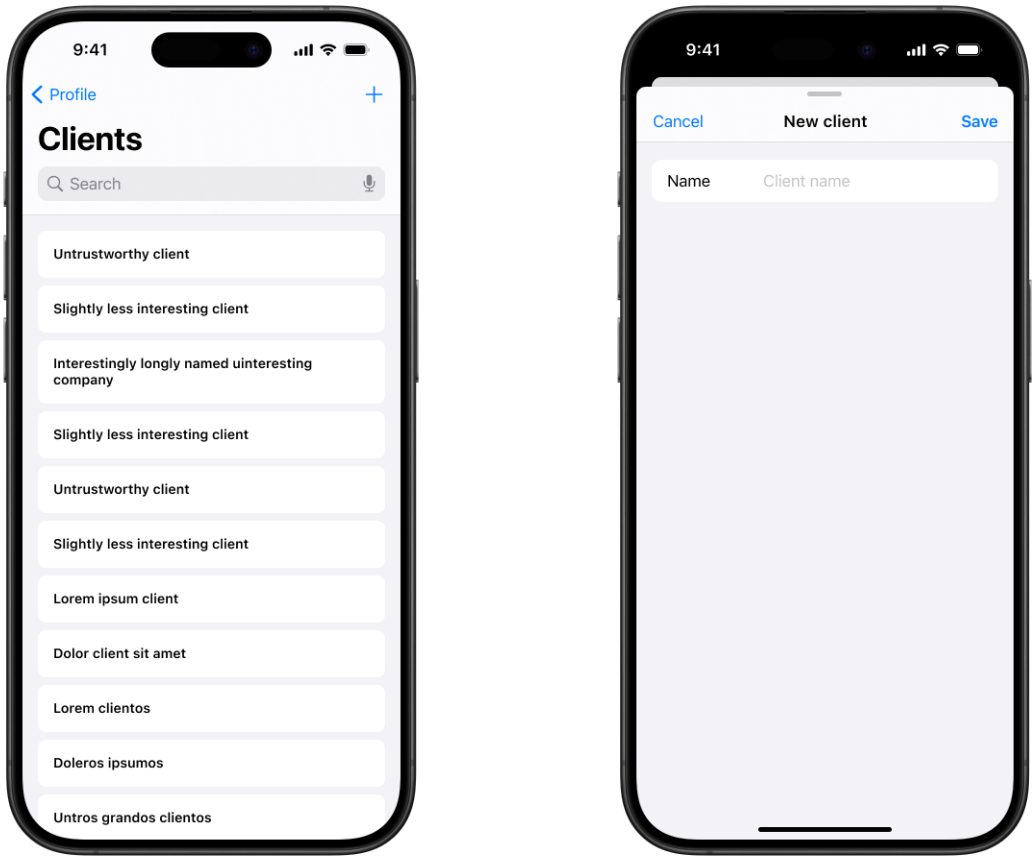
V horní části obrazovky se také nachází uživatelův e-mail, jako indikace toho, na kterém účtě je uživatel přihlášen.

Pokud uživatel klikne na tlačítko klientů, zobrazí se mu seznam jeho klientů, jak lze vidět na obrázku 3.9a. V tomto seznamu může v klientech vyhledávat, otevřít detail klienta, nebo vytvořit nového, pomocí tlačítka vpravo nahoře v navigační liště. Při kliknutí na konkrétního klienta, s cílem zobrazit jeho detail, i při kliknutí na volbu tvorby nového klienta, se zobrazí stejná obrazovka detailu, kterou lze nahlédnout na obrázku 3.9b. V případě zobrazení detailu již existujícího klienta se akorát změní název v navigační liště, předvyplní se hodnoty klienta a zobrazí se navíc tlačítko pro smazání klienta.

Při tvorbě nebo úpravě klienta uživatel může zvolit jeho název. V budoucnu je možné přidat další parametry, které by mohly být ke klientovi přiděleny. Kliknutím na tlačítko *Uložit* v navigační liště se upravený klient uloží a uživatel bude odnavigován zpět na seznam klientů. V případě, že uživatel klikne na tlačítko zrušit, bude také odnavigován zpět na seznam, ale všechny změny budou zahozeny.

Seznam projektů by měl mít také definované stavy pro chybu a prázdná data. V případě chyby se zobrazí popis chyby a tlačítko pro opakování, v případě, že uživatel nemá žádné klienty, se zobrazí tato informace a instrukce k tomu, aby si nějakého klienta vytvořil. Opět je také potřeba rozlišit mezi tím, zda se žádní klienti nezobrazují proto, že žádní nejsou, nebo že žádní nevyhovují vyhledávánému výrazu.

Detail klienta bude mít také explicitní chybový stav, ale prázdný stav zde potřeba není, jelikož existující klient musí mít data vždycky, a nový klient určitě žádná nemá, tudíž budou pole prázdná.



(a) Seznam **(b)** Nový klient

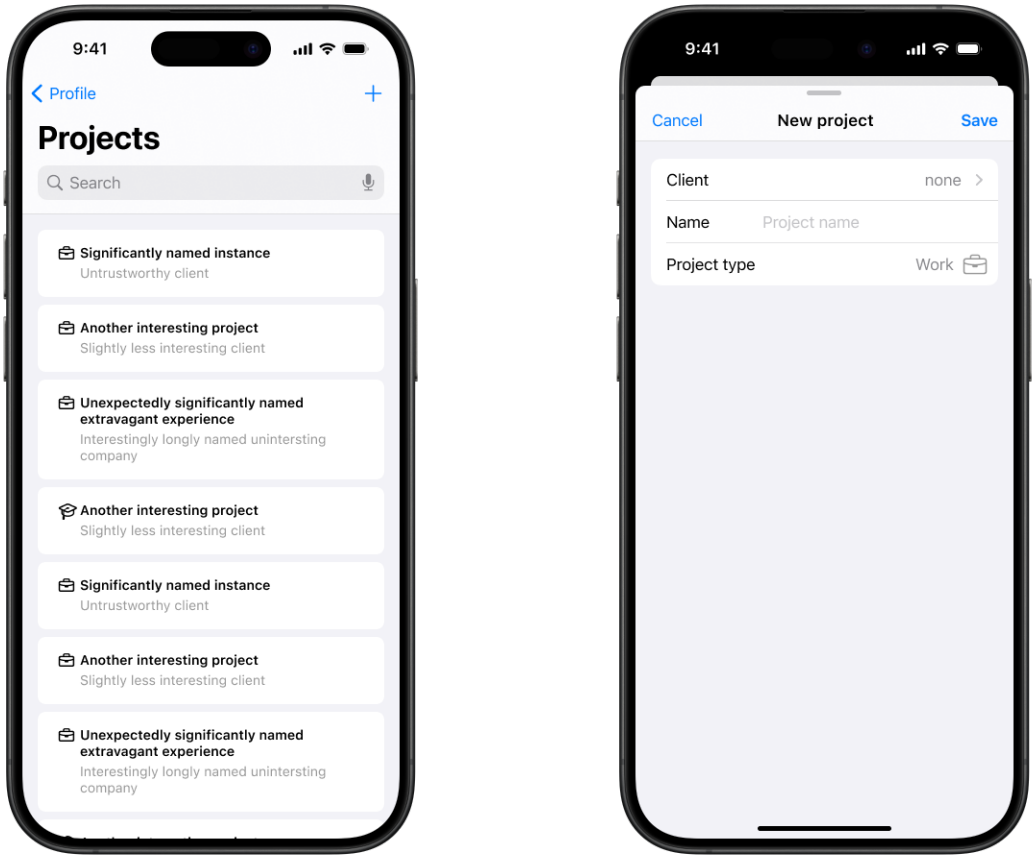
Obrázek 3.9 Klienti

Seznam projektů a detail projektu funguje stejným způsobem, jako u klientů. Kliknutí na tlačítko projektů v profilu otevře seznam, odkud lze otevřít detail/tvorbu nového projektu. Seznam projektů lze nahlédnout na obrázku 3.10a a detail projektu na obrázku 3.10b.

Detail projektu na rozdíl od klienta obsahuje další informace. Každý projekt musí patřit k nějakému klientovi, tudíž je potřeba tohoto klienta zvolit, k čemuž bude sloužit další obrazovka pro výběr klienta, která bude vypadat stejně, jako seznam klientů 3.9a, ale funkčně bude stejná,

jako výběr projektu v časovači 3.6b. Dále je možnost nastavit jméno projektu, a poté nepovinný údaj o typu projektu, což bude definovaný seznam hodnot, ze kterého půjde volit (*Práce*, *Škola* a další).

Chybové a prázdné stavy budou fungovat stejným způsobem, jako u klientů.



(a) Seznam **(b)** Nový projekt

Obrázek 3.10 Projekty

3.2.4 Integrace

Jedním z cílů práce je stanovit požadavky pro integraci spouštěčů měření času, dále požadavky na integraci aplikace s existujícími systémy pro měření času, a tyto požadavky v aplikaci implementovat. Tyto dva typy integrací lze rozdělit do skupin *import* (integrace se spouštěči) a *export* (integrace s existujícími systémy).

3.2.4.1 Import

V sekci 2.2 byly popsány teoretické možnosti, s jakými spouštěči by aplikace šla propojit.

Prvním typem spouštěčů měření času byly fyzické spouštěče, tedy nějaké fyzické formy ovladače. Z uvedených příkladů v analýze byl pouze jeden, který by uměl uskutečnit napojení na aplikaci napřímo, a to *TIMEFLIP* [2], který poskytuje protokol pro BLE komunikaci. Implementace BLE komunikace s hardwarovým produktem je ale poměrně komplexní záležitost,

která byla začleněna nad rámec rozsahu této práce, která se už takto věnuje návrhu a implementaci celé platformy pro měření odpracovaného času. Implementace tohoto typu komunikace tedy může být podnětem pro budoucí rozšíření aplikace.

Důležité ale je, aby na takové rozšíření byla aplikace dobře připravená, a aby minimálně poskytovala veřejné API umožňující se z jakéhokoli budoucího konfigurovatelného hardwarového řešení na aplikaci napojit.

Dalším typem spouštěčů byly softwarové spouštěče, tedy nějaké formy automatizace. V sekci 2.2.2 zabývající se tímto tématem bylo rozebráno několik typů automatizace (podle polohy, času, režimu soustředění, a podobně). Také bylo zmíněno systémové řešení pomocí aplikace *Zkratky* [11], které dokáže automatizaci všech těchto typů vytvořit. *Zkratky* poskytují společné systémové API, které je velmi rozšířené a Apple se snaží vývojáře přesvědčit, aby nějakou část funkcionality aplikace na toto API napojili. Potenciál této platformy je velký, protože její API umožňuje propojování různých procesů navzájem, aplikace si můžou přeposílat výsledky jednotlivých procesů, navazovat na ně a pracovat s nimi.

Aplikace se mohou na toto API napojit pomocí definic takzvaných *App Intents*. Apple v jejich dokumentaci [77] poskytuje podrobné informace k tomu, jak tyto *App Intents* definovat, jak pro ně vytvářet parametry, a další. Tyto struktury reprezentují určitou akci aplikace, kterou uživatel může spustit. Nejpoužívanější (podle očekávání) automatizovatelné funkce aplikace budou zapínání časovače, vypínaní časovače a případně rušení časovače. Bylo by tedy vhodné pro tyto tři funkce vytvořit *App Intents*, které uživatelům umožní vytvářet zkratky a automatizace pro jejich spuštění. V případě zapínání časovače se také naskytuje možnost použití dvou nepovinných parametrů, a to projekt a popis, který bude uživatel chtít k záznamu přiřadit.

Implementace integrace se systémovými zkratkami umožní rozsáhlou a potenciálně velmi rozšiřovatelnou míru integrace, protože se jedná o společné systémové API, na které lze napojit jakoukoli aplikaci, a spousta aplikací nějakou formu napojení na zkratky implementuje. U systémových aplikací to dokonce platí pro všechny – pokrytí automatizovatelných funkcí je zde velmi široké.

Integrace pro import tedy budou sloužit pro následující případy užití:

- **Tvorba, úprava nebo odstranění zkratky** – Uživatel si vytvoří, upraví nebo odstraní existující zkratku, která může obsahovat následující akce:
  - **Spuštění časovače** se zadanými parametry.
  - **Zastavení časovače** a uložení měřeného záznamu.
  - **Zrušení časovače** a zahození měřeného záznamu.
- **Tvorba, úprava nebo odstranění automatizace** – Uživatel si vytvoří, upraví nebo odstraní existující automatizaci, která může spouštět vytvořené zkratky.
- **Manuální spuštění zkratky** – Uživatel manuálně spustí zkratku, například pomocí hlasového asistenta *Siri*, pomocí *Spotlight* vyhledávání, nebo přímo v aplikaci *Zkratky*.

3.2.4.2 Export

V sekci 2.3 byly popsány populární systémy *Clockify*, *Toggl Track* a *Deputy*, které poskytují funkce pro zaznamenávání odpracovaného času. Všechny tyto systémy poskytují možnost importu dat z CSV souborů, a u *Clockify* a *Toggl Track* šlo o velmi podobné formáty.

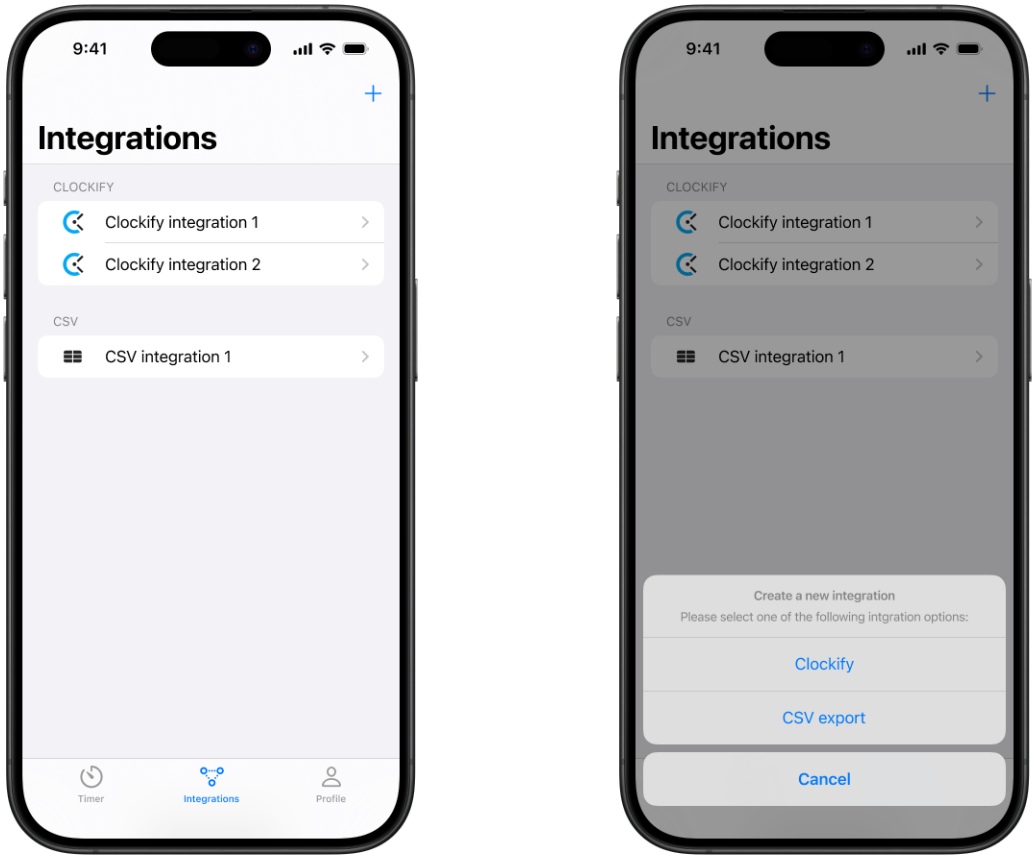
Možnost exportu z aplikace do CSV souboru by tedy byla silným nástrojem, pomocí kterého by se záznamy z aplikace mohly nejen importovat do těchto systémů, ale byla by uživatelům poskytnuta volná možnost, co s exportovanými daty dělat. V různých programech by si je mohli dle svého uvážení analyzovat, vizualizovat, a další. Možnost exportu dat do CSV souboru by tedy v aplikaci neměla chybět. Ideálně by mělo jít o formát, který půjde importovat jak do systému *Clockify*, tak do systému *Toggl Track*. Systém *Deputy* cílí na trochu jinou cílovou skupinu, napojení na něj by proto nebylo tolik relevantní, jak již bylo popsáno v sekci 2.3.3.

Všechny zmíněné systémy také poskytovaly veřejné API pro napojení na jejich struktury. Zde už se nepůjde spolehnout na nějaké obecné řešení, které bude pasovat na více systémů, ale bude se potřeba na každý systém napojit zvlášť. V rámci implementace by mohla být vytvořena integrace alespoň s jedním systémem, a zároveň by implementace mohla být připravena pro rozšíření o napojení na další systémy.

Integrace pro export tedy budou sloužit pro následující případy užití:

- **Zobrazení existujících integrací** – Uživatel si zobrazí vytvořené integrace.
- **Tvorba, úprava nebo odstranění integrace** – Uživatel si vytvoří, upraví nebo odstraní integraci.
- **Export dat pomocí integrace** – Uživatel pomocí konkrétní integrace exportuje data v zadaném rozsahu.

Rozhraní pro integraci s dalšími systémy bylo navrženo následovně. V pořadí druhá, tedy prostřední, karta v navigační liště bude sloužit pro vytváření integrací. Aby aplikace mohla v budoucnu podporovat různé typy integrací a zároveň aby si uživatel mohl svá nastavení integrací ukládat, bude hlavní obrazovku integrací představovat seznam již vytvořených integrací, který lze nahlédnout na obrázku 3.11a.



(a) Seznam **(b)** Nová integrace

Obrázek 3.11 Integrace

V tomto seznamu budou prvky jednotlivých vytvořených integrací, které budou mít přidělenou

ikonku podle toho, o jaký typ integrace se jedná (CSV, *Clockify*, a další). Vpravo nahoře v navigační liště bude tlačítko pro přidání nové integrace, které otevře dialog s možnostmi, jaký typ integrace chce uživatel vytvořit, jak lze nahlédnout na obrázku 3.11b. Tento seznam bude mít definovaný chybový stav, kde se místo seznamu zobrazí popis chyby a tlačítko pro opakování, a prázdný stav, který zobrazí jen informaci o tom, že si uživatel žádné integrace zatím nevytvořil.

Vytvoření nové integrace, nebo otevření detailu existující integrace, otevře obrazovku pro detail integrace, jak lze vidět na obrázku 3.12a. Obrázek 3.12a reprezentuje detail *Clockify* integrace, u které je potřeba nastavit název integrace, API klíč pro napojení na *Clockify* účet a případně nějaké další parametry pro exportování dat. Tento detail se bude lišit pro různé typy integrací podle toho, jaké parametry budou pro exportování dat potřeba. Například u exportu do CSV dat bude potřeba jen název a nic dalšího. Tato obrazovka bude potřebovat definic chybového stavu, který bude opět zobrazovat popis chyby a tlačítko pro opakování. Prázdný stav zde nemá smysl. Obrazovky pro vytvoření nové integrace, nebo pro úpravu existující integrace, se budou lišit akorát v možnosti odstranění integrace – u existující integrace se přidá tlačítko, které to umožní.

Kliknutí na tlačítko exportu dat otevře další obrazovku, kterou lze vidět na obrázku 3.12b. Tato obrazovka bude sloužit pro manuální export dat v zadaném období. U některých typů integrací totiž bude dávat smysl nabídnout automatický export dat, například u *Clockify* integrace, kde je možné každý nově vytvořený záznam rovnou automaticky exportovat do *Clockify* účtu. Ale například u exportu do CSV souboru taková automatizace nedává smysl, tento typ integrace bude tedy sloužit pouze pro manuální export. Tato obrazovka je poměrně jednoduchá, vyžaduje po uživateli pouze zadání od kdy a do kdy chce časové záznamy exportovat. Pokud zadá validní interval a klikne na tlačítko pro export, aplikace data exportuje. V případě *Clockify* integrace se pokusí data odeslat na *Clockify* API, v případě CSV integrace nabídne uživateli výsledný soubor, se kterým si poté uživatel může dělat co chce – uložit do souborů, odeslat e-mailem, otevřít v jiné aplikaci, atd. U této obrazovky není potřeba definovat žádné chybové nebo prázdné stavy, protože samotná obrazovka žádná data nepotřebuje.

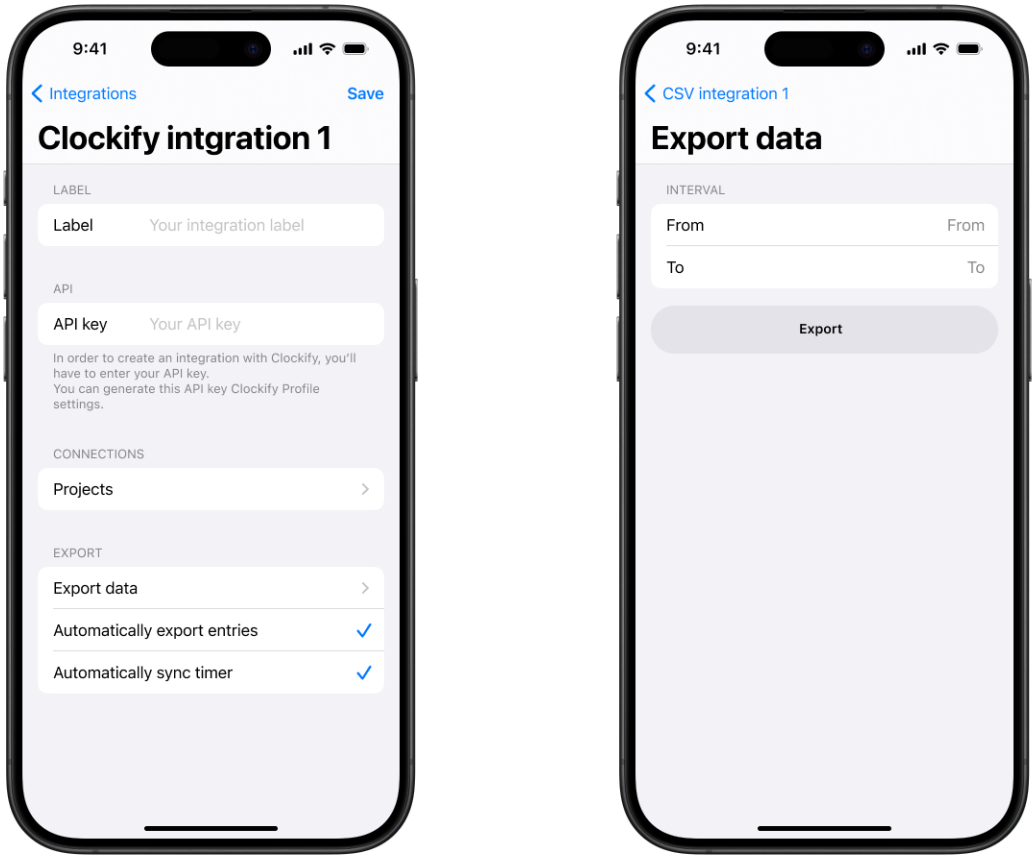
3.3 Architektura

V analýze v sekcích 2.4.3, 2.5 a 2.6 byly popsány různé architektury pro mobilní aplikace, backendová řešení a databáze. Tato sekce se bude věnovat návrhu architektury celé platformy a nástrojů či jiných produktů, které budou potřeba pro realizaci aplikace.

3.3.1 Architektura platformy

V první řadě je potřeba se rozhodnout, jakou architekturu bude mít celá platforma, tedy jaký přístup bude zvolen pro vývoj mobilní aplikace, jaké řešení bude zvoleno pro implementaci backendu a jaká bude zvolena implementace databáze. Kombinace těchto řešení lze seřadit na stupnici, která má dva konce:

- Outsourcování co největší části platformy na externí poskytovale (BaaS, DBaaS, atd.). Tento přístup by byl nejjednodušší z hlediska komplexní a časové náročnosti implementace, ale ze své podstaty by byl nejvíce omezující ve flexibilitě a možnosti budoucích rozšíření. V případě, že by bylo v budoucnu rozhodnuto, že se například změní poskytovatel databáze, tak by bylo značně náročnější implementaci pro takovou změnu upravit. Zejména v případě, kdyby na jednom externím poskytovateli záviselo více částí infrastruktury.
- Vlastní implementace všech částí platformy. Toto by vyžadovalo vlastní implementaci nativní aplikace, backendu i databáze. V případě, že by bylo v budoucnu rozhodnuto, že se aplikace rozšíří například o Android aplikaci, musela by být implementována tato aplikace celá, ale zase by se jen napojila na existující backend. V tomto ohledu je toto řešení nejvíce flexibilní.



(a) Clockify integrace **(b)** Export dat

Obrázek 3.12 Detail integrace

Předchozí příklad eventuální změny implementace databáze by byl mnohem jednodušší, pouze by se změnil zdroj dat v implementaci backendu. Tento přístup by byl ale zásadně náročnější na komplexnost a časovou náročnost implementace.

Vhodné řešení pro implementaci této aplikace bude pravděpodobně ležet někdy mezi těmito dvěma extrémy. Vzhledem k tomu, že zadání aplikace vyžaduje implementaci aplikace pouze pro platformu iOS, nabízí se lákavé řešení implementovat pouze tuto nativní aplikaci a veškerý zbytek infrastruktury opravdu delegovat na nějaký BaaS. Z osobních preferencí a kvůli solidní konkurenceschopnosti řešení a jeho obstání jako práce z dílny oboru softwarového inženýrství se ale toto řešení zdá jako poněkud méně šťastné. Jako jedním z vedlejších cílů této práce byla zvolena široká možnost budoucích rozšíření této aplikace. Potenciál, kam dále by se implementace dala rozšířit, je široký – aplikace by mohla být rozšířena o již zmíněnou Android aplikaci, ale třeba také o webovou aplikaci, nebo o nějakou administrátorskou variantu. Zmíněné řešení s delegováním celé infrastruktury mimo nativní aplikaci na BaaS by jednoduché řešení těchto rozšíření moc neumožňovala, protože každé další napojení by muselo být implementováno úplně odděleně a muselo by se nějakým způsobem na BaaS napojit, čímž by navíc vznikla silná závislost na poskytovateli BaaS služby.

Zajímavou technologií, která byla zmíněna v sekci 2.5 je *Kotlin Multiplatform* [50]. Z osobního pohledu tato technologie činí ideální kompromis mezi maximálním využití funkcí jednotlivých platforem a snahy sdílení co největší části kódu. Tato technologie poskytuje nástroje pro sdílení

kódu mezi aplikacemi pro různé platformy – iOS, Android, Web a další. Využití této technologie, přestože se realizace bude věnovat pouze implementaci pro platformu iOS, poskytne mnohem jednodušší možnosti pro budoucí rozšíření o další platformy. Ve sdíleném kódu bude potřeba jen přidat nové moduly podle cílových platforem a bude možnost využít co největší část sdíleného kódu, který je společný pro všechny platformy. Použití této technologie je také osobní preferencí kvůli sympatiím s jejím používáním, lehkou znalostí práce s ní a možnost se s touto technologií více seznámit napřímo.

Rozhodnutí využít technologie *Kotlin Multiplatform* ovlivní i rozhodnutí při volbě technologie pro implementaci backendu. Pro jazyk *Kotlin* totiž existuje knihovna *Ktor* [78], která poskytuje silné nástroje pro serverovou komunikaci. Poskytuje nástroje jak pro implementaci backendu, tak pro implementaci klienta a jejich vzájemnou komunikaci. Přestože tato knihovna není tak rozšířená a známá jako například *Spring Boot* nebo *Django*, které byly zmíněny v sekci 2.6, tak se ale hodí pro přehlednost a jednotu zdrojového kódu platformy, kde by celý backend i multiplatformní část mohla být napsána v jednom jazyce, v jednom projektu, v jednom společném repozitáři.

Rozhodnutí využít nástroje spojené s multi-platformním vývojem v *Kotlinu* by se daly využít i dále. Ve světě vývoje aplikací pro systém Android je poslední dobou častou volbou implementace uživatelského rozhraní pomocí knihovny *Jetpack Compose* [79]. Společnost *JetBrains*, tvůrce technologie *Kotlin Multiplatform*, vyvíjí technologii *Compose Multiplatform* [80], která dále umožňuje dokonce implementaci uživatelského rozhraní ve sdíleném kódu, které si na rozdíl od cross-platformních řešení stále drží výhody multi-platformního přístupu, tedy lepšího využití možností dané platformy. Tato technologie je ale ve fázi *Alpha* a není tolik pokročilá. Z toho důvodu zůstala volba pro implementaci uživatelského rozhraní na nativním moderním přístupu s pomocí knihovny *SwiftUI* [38].

Zatím tedy padly následující volby – nativní uživatelské rozhraní s pomocí jazyka *Swift* a knihovny *SwiftUI*, multi-platformní implementace sdíleného kódu s pomocí technologie *Kotlin Multiplatform*, backend s pomocí jazyka *Kotlin* a knihovny *Ktor*. Nyní ještě zbývá volba pro implementaci databáze.

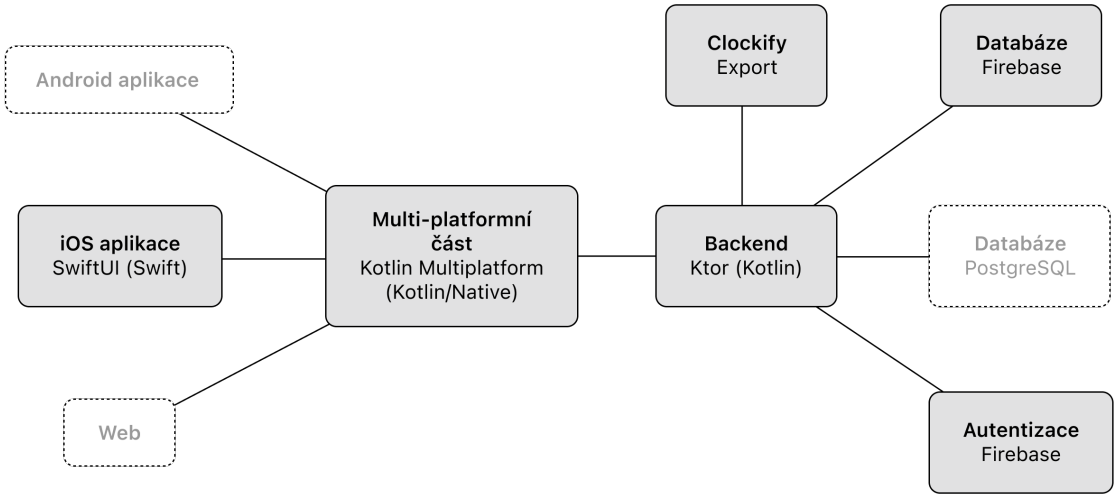
V tomto směru už bylo rozhodnuto, že využití externího DBaaS poskytovatele bude na místě. Toto řešení zjednoduší implementaci aplikace o technologii vlastní databáze, ale stále si udrží možnost eventuální volby, kdy by se poskytovatel databáze mohl změnit. Jelikož bude mít platforma vlastní backend, nebude problém technologii databáze změnit z externího poskytovatele například na technologii *PostgreSQL*, protože už to bude jen otázka změny zdroje a migrace dat, zatímco implementace byznysové logiky a dalšího, která bude na backendu, zůstane stejná.

Jako dobrá volba externího poskytovatele BaaS se jeví *Firebase* [51] od firmy *Google*. Jedná se o rozšířenou technologii používanou mnoha aplikacemi. Tato technologie zároveň umožní obsluhu jiných funkcionalit aplikace, jako například autentizace. Není potřeba mezi implementací databáze a autentizace tvořit nějakou závislost, přijde-li tedy v budoucnu požadavek poskytovatele autentizace nebo databáze změnit, může tak být učiněno nezávisle na sobě, tedy klidně pouze jedno z toho, nebo obojí.

Tato volba architektury celé platformy dodržuje stanovený požadavek na velkou míru flexibility a rozšiřovatelnosti, zároveň se snaží zjednodušit náročnost implementace a eliminovat duplikace kódu, bude-li aplikace rozšířena o další platformy. Schéma této architektury lze nahlédnout na obrázku 3.13. V tomto obrázku jsou také znázorněny možnosti případného rozšíření aplikace. Z této celkové architektury platformy lze poté navrhovat architekturu jednotlivých částí.

3.3.2 Architektura nativní aplikace

Architektura nativní aplikace bude vycházet z veřejné šablony pro projekt mobilní aplikace pro platformy iOS a Android s použitím technologie *Kotlin Multiplatform*, nazvané *Matee KMP DevStack* [81]. Důvody k použití této platformy jsou osobní preference a znalost tohoto projektu. Alternativou by bylo například použití generátoru šablon projektů *Kotlin Multiplatform Wizard*



Obrázek 3.13 Architektura platformy

[82], ale *DevStack* společnosti *Matee* obsahuje velkou řadu užitečných nástrojů pro uživatelské rozhraní a další užitečné komponenty.

Matee DevStack používá architekturu *Clean Architecture* popsanou v sekci 2.4.3. Technologie *Kotlin Multiplatform* do této architektury zapadá tak, že nativní aplikace implementuje celou prezentační vrstvu, a multi-platformní část implementuje co největší možnou část doménové a datové vrstvy. Multi-platformní část je do nativní aplikace přidána jako knihovna, která je závislostí doménové vrstvy. Nativní aplikace má tedy přístup hlavně k doménovým modelům a *Use Cases*.

Nativní aplikace využívá *modularizaci*, která představuje další vlastnost architektury, ve které je aplikace rozdělená do několika modulů, v tomto případě podle funkcionalit. Prezentační vrstva aplikace tedy bude disponovat moduly podle funkcí jako *Onboarding* (přihlášení/registrace), *Timer* (časovač, historie záznamů), *Integrations* (nastavení integrací), *Profile* (profil uživatele) a poté například modulem *UIToolkit*, který definuje společné komponenty pro prezentační vrstvu.

Doménová vrstva nativní aplikace bude obsahovat moduly *SharedDomain*, což je modul, který definuje společné atributy domény aplikace, ale zde bude mít hlavní funkci jako poskytovatel sdílené knihovny, která bude výstupem multi-platformní části. Dále bude vrstva obsahovat modul *Utilities*, který definuje užitečné nástroje pro práci s doménovými či sdílenými funkcemi a objekty.

Datová vrstva obvykle obsahuje implementace repozitářů, v modulech podle funkcionalit, a *Providers*, také v modulech podle funkcionalit. Jelikož většinu této vrstvy bude implementovat sdílený kód, bude tato vrstva obsahovat jen ty části, které sdílený kód implementovat nemůže. Což jsou obvykle platformní záležitosti, jako obsluha notifikací, GPS polohy, a další. Mohou to být ale i další implementace, pro které zkrátka multi-platformní část nemá podporu, jako třeba různé funkce poskytovatele *Firebase*. Tyto části kódu budou fungovat tak, že v multi-platformní části kódu se budou nacházet rozhraní pro funkcionality, které musí implementovat každá platforma sama. V multi-platformní části se bude pracovat s těmito rozhraními a nativní aplikace bude zodpovědná za to, že rozhraním poskytne implementace. Nativní aplikace tedy bude implementovat pouze tu část, kterou sdílená část implementovat nemůže, a výstupy těchto implementací bude vracet do sdíleného kódu.

Zmíněné moduly jsou v nativní aplikaci ve skutečnosti balíčky typu *Swift Package*, které jsou spravovány nástrojem *Swift Package Manager* [83]. Nativní aplikace dále obsahuje aplikační

vrstvu, která obsahuje základní části aplikace, jako *AppDelegate*, *Info.plist*, ikonu aplikace, nebo například rodičovský *Flow Controller* aplikace, obvykle *AppFlowController*, ze kterého se vytváří všechny další navigační toky. Architekturu nativní aplikace lze nahlédnout na obrázku 3.14.

3.3.3 Architektura multi-platformní části

Multi-platformní část je rozdělena do dvou modulů – *commonMain* a *iosMain*. *commonMain* obsahuje sdílený kód všech platforem, a *iosMain* obsahuje kód, který se kompiluje pouze, pokud je cílovou platformou iOS. Toho se využívá speciálními klíčovými slovy v *Kotlinu* – `expect` a `actual`. Funguje to podobně, jako rozhraní – *commonMain* definuje rozhraní funkce/třídy pomocí `expect`, a platformní modul definuje implementaci pomocí `actual`. Modul *iosMain* má totiž přístup k některým platformním API, ke kterým sdílený modul přístup nemá.

Modul *commonMain* poté dodržuje stejnou architekturu, jako nativní aplikace, tedy *Clean Architecture* s *modularizací* podle funkcionalit aplikace. Jediný rozdíl oproti architektuře používané v nativní aplikaci a popsané v sekci 2.4.3 je ten, že *Providers* se ve světě *Kotlinu* nazývají *Sources*, ale jejich význam je identický.

Pokud bude v budoucnu přidána podpora pro další platformy (Android, Web, ...), bude potřeba v multi-platformní části přidat konkrétní modul, například *androidMain*. Tyto platformní moduly musí implementovat jen ty části, které jsou specifické pro platformu a *Kotlin/Native* API k nim má přístup. Dalším specifikem `expect` a `actual` je, že pokud je v *commonMain* v nějakém balíčku použito klíčové slovo *expect*, tak v platformním modulu potom musí být jeho `actual` implementace ve stejném balíčku. Tedy pokud je například `expect someFunction(param: String): String` v balíčku `com.some.path`, tak implementace potom musí být také v tomto balíčku. Tím pádem potom musí automaticky platformní moduly reflektovat strukturu aplikace, která je v *commonMain*.

3.3.4 Architektura backendu

Architektura backendu byla zvolena tak, aby byla v souladu s *Clean Architecture*, která je používána v nativní aplikaci i v multi-platformní části. Datové a doménové vrstvy jsou tedy identické, s jednou výjimkou, že v doménové vrstvě nejsou *Use Cases*, ale ve vyšších vrstvách se budou používat *Repositories* napřímo. To zejména proto, že na backendu mají *Use Cases* menší smysl, než v aplikaci. Ale nic by nebránilo tomu je používat také. Prezentační vrstva zde bude obsahovat *Routes*, tedy *Endpoints*, se kterými bude moct aplikace komunikovat. Jednotlivé *Endpoints* budou přímo volat *Repositories*.

3.3.5 Architektura dat

S ohledem na typ databáze, která byla pro platformu zvolena, je potřeba stanovit nějaké schéma dat. *Firebase* nabízí dva typy databází – *Cloud Firestore* a *Realtime Database*. Oba typy jsou *NoSQL* databáze, *Cloud Firestore* je dokumentová databáze, která ukládá data jako kolekce dokumentů. *Realtime Database* ukládá data v jednom velkém JSON stromu. Vzhledem k tomu, že *Cloud Firestore* disponuje oproti *Realtime Database* řadou výhod a je doporučenou variantou, bude tato databáze zvolena pro potřeby aplikace Trackee. [84]

Jak již bylo řečeno, bude se jednat o dokumentovou databázi. Pro potřeby aplikace bude nutné definovat následující objekty:

- **Klient** – Bude obsahovat informace o klientovi, v základní variantě jméno. V budoucnu může být přidáno více atributů.
- **Projekt** – Bude obsahovat informace o projektu. Projekt musí patřit k nějakému klientovi, takže bude muset obsahovat identifikátor klienta, ke kterému patří. Dále bude obsahovat jméno a nepovinně typ projektu. V budoucnu také může být přidáno více atributů.

- **Typ projektu** – Předdefinovaný seznam hodnot, například *Práce* nebo *Škola*.
- **Uživatel** – Bude obsahovat informace o uživateli. Identifikátorem uživatele bude `uid` (User ID), které by mělo být sdíleno s identifikací v autentizaci. Dále bude uživatel obsahovat informace o svém časovači. Poté bude potřeba držet informace o tom, jaké klienty a projekty má uživatel přiřazené. To by šlo ukládat například pomocí vnořené kolekce, kde bude jednotlivý dokument odpovídat klientovi, a tento dokument bude obsahovat pole hodnot, kterými budou identifikátory projektů, které má od daného klienta přiřazené. A nakonec bude ještě uživatel muset obsahovat data o svých vytvořených integracích. Ty mohou být uloženy v další kolekci, kde jednotlivý dokument bude odpovídat integraci.
- **Data časovače** – Informace o tom, jaký má vybraný projekt, popis, zda časovač běží a případně od kdy běží.
- **Integrace** – Bude obsahovat název, typ a případně další atributy relevantní k typu integrace.
- **Typ integrace** – Prozatím *CSV* nebo *Clockify*.
- **Časový záznam** – Prvek historie časových záznamů, který bude obsahovat informace zadané v časovači. Tím je tedy vybraný projekt, popis, čas začátku a čas konce.

Klient, Projekt, Integrace a Časový záznam budou mít implicitně své identifikátory. U Uživatele bylo již zmíněno, že jeho identifikátorem bude `uid`.

Dále je tedy potřeba navrhnout schéma dat. Jelikož se jedná o dokumentovou databázi, je potřeba ho navrhnout tak, aby nevznikaly zbytečné duplicity a aby byla potřebná data lehce přístupná. Navržené schéma lze nahlédnout na obrázku 3.15.

Schéma je navrženo tak, aby základní objekty byly ukládány ve vlastních top-level kolekcích (*users*, *clients*, *projects* a *entries*). Kolekce *users*, respektive *clients*, obsahují přímo dokumenty jednotlivých uživatelů, respektive klientů, podle jejich identifikátoru. Ale protože *projects* vždy patří k nějakému klientovi, respektive *entries* k nějakému uživateli, mají ještě jejich top-level kolekce vnořené kolekce v dokumentech podle identifikátoru jejich klienta, respektive jejich uživatele. Alternativním přístupem by bylo *projects*, respektive *entries*, ukládat přímo do dokumentů jednotlivých *clients*, respektive *users*, ale tímto způsobem budou dokumenty odlehčeny od obsahování tolika dat a všechny nejpoužívanější objekty domény budou mít své top-level kolekce.

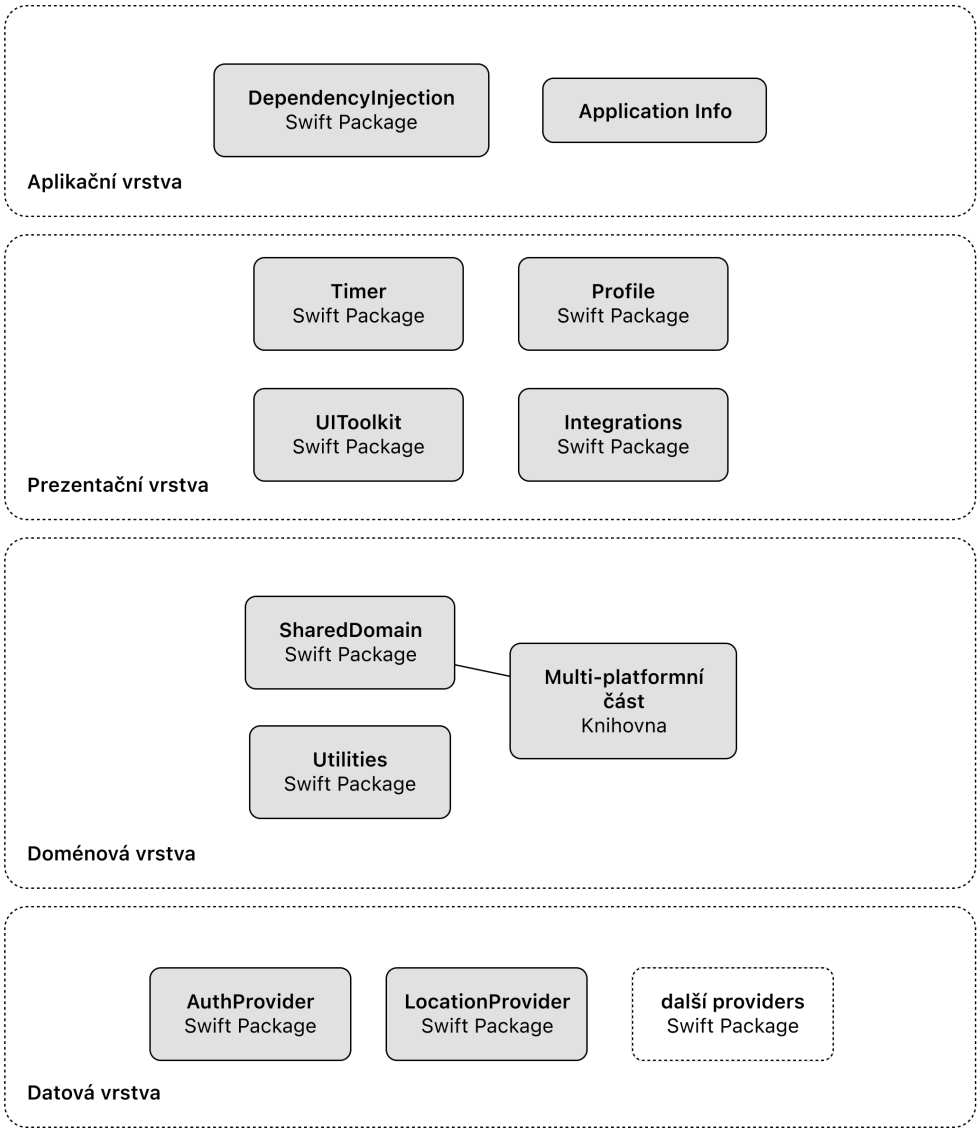
3.4 Nástroje potřebné pro realizaci

Před začátkem implementace je také potřeba navrhnout nástroje, které budou pro realizaci potřeba.

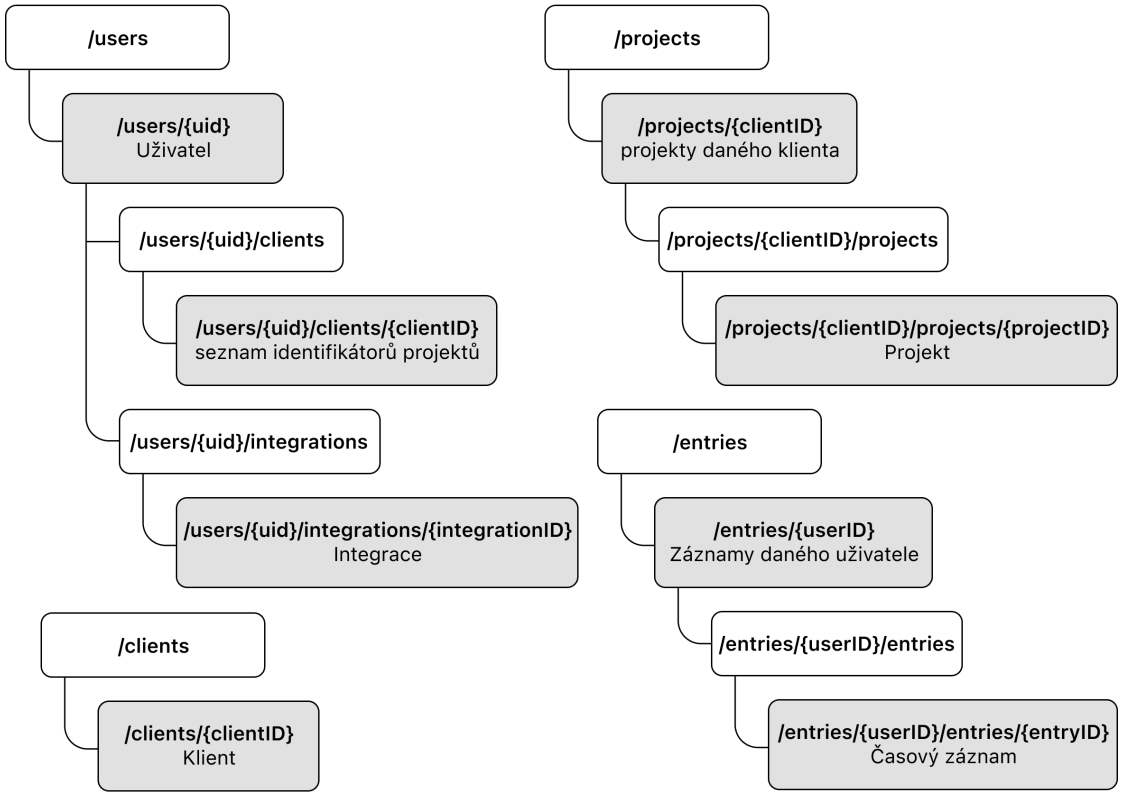
U vývoje nativní aplikace není moc široký výběr vývojových nástrojů, jelikož jediné oficiální vývojové prostředí pro platformu iOS je *Xcode*, popsané v sekci 2.4.2, které podporuje pouze operační systém *macOS*. Toto vývojové prostředí obsahuje software pro podporu všech potřebných nástrojů, které jsou pro vývoj pro platformu iOS potřeba – kompilace, archivace, simulátory mobilních zařízení, debugger a další.

Pro vývoj multi-platformní části a backendu bude použito populární vývojové prostředí *IntelliJ IDEA* společnosti *JetBrains* [85]. Toto IDE bylo zvoleno z osobních preferencí a jeho dobré podpoře technologie *Kotlin Multiplatform* i jazyka *Kotlin*.

Dalším užitečným nástrojem během vývoje bude verzovací systém *Git* [86], který bude spravovat a verzovat zdrojový kód implementace.



Obrázek 3.14 Architektura nativní aplikace



Obrázek 3.15 Schéma dat

Kapitola 4

Realizace

Tato kapitola se zabývá samotnou realizací platformy pro měření odpracovaného času. Popisuje nástroje použité během vývoje, strukturu zdrojového kódu a nakonec implementaci jednotlivých navržených funkcionalit aplikace.

4.1 Nástroje pro vývoj

Jak bylo navrženo v sekci 3.4, pro nativní aplikaci bylo použito vývojové prostředí *Xcode*. Verze tohoto IDE, na kterém byla implementace vyvíjena, byla 15.2. S touto verzí IDE se pojí verze programovacího jazyku *Swift* 5.9.2. Aplikace byla vyvíjena na operačním systému *macOS Sonoma* (14.4). Nativní aplikace bude definovat minimální iOS verzi cílové platformy na verzi 17.0, jelikož tato verze umožňuje použití nejnovějších nástrojů platformy iOS. Podle dat z února 2024 už tuto nejnovější verzi operačního systému používalo 66% ze všech aktivních uživatelů mobilních telefonů iPhone. Mezi zařízeními představených v předchozích čtyřech letech je to dokonce 76% [87].

Vývojové prostředí pro multi-platformní část a backend také dodržuje návrh ze sekce 3.4, bylo tedy použito *IntelliJ IDEA*, verze 2024.1. Pro jazyk *Kotlin* byla použita verze 1.9.10. a *Git* repozitář byl zálohován nástrojem *GitHub* [88] v repozitáři [89]. Během vývoje nebyly použity žádné větve ani další nástroje pro týmovou spolupráci v nástroji *Git*, jelikož nikdo další na projektu nespolupracoval. Veškerý vývoj tedy probíhal přímo v hlavní větvi *main*.

4.1.1 Firebase

V nástroji *Firebase*, který byl vybrán v návrhu v sekci 3.3.1, byl vytvořen projekt pro aplikaci Trackee. Byly vytvořeny tři prostředí – Alpha, Beta a Produkce. V projektu zatím byly nastaveny funkce pro autentizaci a *Firestore* databázi.

Firebase autentizace podporuje různé formy autentizace, jako e-mail s heslem, ale třeba také přihlášení přes Google, přes Apple, přes Facebook a podobně. Rozšíření možností přihlášení by mohlo být podnětem pro budoucí vývoj aplikace, v této fázi je zatím implementováno pouze přihlášení přeš e-mail a heslo.

Při tvorbě *Firestore* databáze *Firebase* nabízí výběr, kde se má instance databáze nacházet (západní Evropa nebo Severní Amerika). Byla vybrána nejbližší možnost, tedy západní Evropa.

4.1.2 Nasazení backendu

Během vývoje byla aplikace backendu spouštěna a testována na lokálním stroji. V místních sítích se tedy stačilo připojovat pomocí IP adresy počítače, na kterém aplikace backendu běžela, přes protokol HTTP, například: `http://192.168.88.70:8080`

Pro umožnění používání aplikace kdekoli, kdykoli a bez potřeby na lokálním stroji neustále ručně spouštět aplikaci backendu, bylo vhodné vybrat nějaké řešení, které by umožnilo nasazení aplikace backendu. K tomu existuje řada externích poskytovatelů. Pro potřeby tohoto projektu byl zvolen nástroj *Railway* [90], jelikož zdarma nabízí počáteční kredit v hodnotě pěti amerických dolarů, což by mělo na nějakou dobu vystačit pro potřeby vývoje aplikace. Tomuto nástroji pouze stačí propojení s *GitHub* repozitářem, nastavení skriptu pro spuštění a je hotovo. Nástroj poté při každé aktualizaci repozitáře sestaví projekt, spustí backend a nasadí ho na URL adrese `https://trackee-app-production.up.railway.app`.

Railway také podporuje automatické uspávání aplikace, které může snížit využívání kreditu v době, kdy aplikace není využívána [91]. Pokud se tedy aplikace nějakou dobu nepoužívá, je potřeba počítat s několika sekundovou prodlevou při prvním požadavku.

Jednou nevýhodou nástroje *Railway* ale je, že v bezplatné variantě nabízí umístění stroje, na kterém běží instance, pouze ve státě Oregon ve Spojených státech amerických. Jelikož je to v podstatě na druhé straně Země, vzniká tím poměrně znatelná prodleva mezi klientem a serverem. A v kombinaci s tím, že při nastavení nástroje *Firebase* bylo zvoleno umístění instance v Evropě (a toto umístění není možné po nastavení nástroje změnit), vznikají tím hlavně prodlevy při komunikaci s databází, protože nyní musí backend každý požadavek na čtení nebo zápis dat posílat zhruba 8 tisíc kilometrů daleko. Rozdíl mezi používáním instance backendu běžící pomocí nástroje *Railway* a používáním lokální instance je znatelný – například přihlášení a načtení hlavní obrazovky včetně plné počáteční stránky historie záznamů (10 záznamů) trvá 5-7 sekund, zatímco při použití lokální instance toto trvá 2-3 sekundy.

4.1.3 Nasazení aplikace a Testflight

Vývojové prostředí *Xcode* umožňuje nahrání vyvíjené aplikace do připojeného mobilního zařízení. Dané zařízení ale musí být fyzicky připojeno k počítači, na kterém IDE běží, musí být ve vývojářském režimu a musí být napojené na vývojářský účet, pod který aplikace spadá. Jedná se tedy o poměrně komplikované řešení, je-li účelem instalace aplikace mezi další uživatele, například mezi testery.

Nástrojem, jak řešit nasazení aplikace mezi širší počet uživatelů, ale nikoli zatím jako finální aplikaci mířenou na reálné uživatele, je *Testflight* [92]. Tento nástroj je určen pro beta testování vyvíjených aplikací. Umožňuje nahrávání archivovaných (sestavených) aplikací pomocí nástroje *App Store Connect* [93] a jejich následnou distribuci mezi vybrané uživatele, kteří si tuto aplikaci nainstalují. Jsou dva způsoby, jak lze aplikace přes *Testflight* distribuovat:

- **Interní testování** – vývojář pozve jednotlivé uživatele pomocí jejich e-mailové adresy k testování aplikace a ti si ji mohou stáhnout. Tito vývojáři musí mít účet *Apple ID* a musí být přidání mezi interní vývojáře aplikace.
- **Veřejné testování** – vývojář může do testování přizvat kohokoli, dokonce stačí jenom takzvaný *public link*, přes který si uživatelé mohou aplikaci nainstalovat. Aby aplikace mohla být testována pomocí veřejného testování, musí být schválena firmou *Apple* v rámci procesu *App Review*, který aplikace podstupují i tehdy, pokud chtějí zamířit mezi reálné uživatele do obchodu *App Store*.

Aplikace Trackee umožňuje oba typy testování, již prošla schvalovacím procesem *App Review* a může si ji tak nainstalovat kdokoli, kdo bude pozvaný přes e-mailovou adresu, nebo pomocí *public linku*: `https://testflight.apple.com/join/cTRdRkBc`

4.2 Architektura a struktura projektu

Jako šablona celého projektu sloužil *DevStack* společnosti *Matee* [81], ze kterého byla odstraněna většina implementovaných funkcionalit a jejich balíčků, kromě těch, které se hodilo znovu využít (onboarding, profil, ...). Ze šablony byly také ponechány nástroje na síťovou komunikaci (síťový klient pomocí knihovny *Ktor* [78]), interoperabilitu a další. V šabloně také byly ponechány některé nástroje, které zatím projektem nejsou využívány, ale v budoucnu při dalším rozvoji aplikace by se mohly hodit, jako nástroje pro práci s lokálním úložištěm *SQLDelight*, *UserDefaults* nebo *Keychain*, nebo *Providers* pro obsluhu GPS, obsluhu notifikací, a další. Jelikož Aplikace Trackee neimplementuje Android klienta, je celý jeho modul ze šablony odstraněn.

Aplikace implementuje tři prostředí – Alpha, Beta a Produkce. Zvykem bývá, že Alpha má vlastní instanci databáze a serveru, která slouží pouze pro testování a ladění, například při nasazování nových verzí backendu, a podobně. Beta na tom je pak obvykle podobně, ale buď duplikuje data z produkčního prostředí (ale přímo ho nemůže ovlivňovat), nebo je na produkční prostředí přímo napojena, ale nabízí přidané ladící možnosti. Produkční prostředí pak poskytuje databázi a server, které jsou využívány reálnými uživateli, kde je priorita orientována spíše na rychlost, než na možnosti ladění. Během vývoje aplikace Trackee nebylo potřeba využívat více instancí databáze nebo serveru, protože na projektu nepracuje více lidí, nevznikají konflikty a není zatím využíván reálnými uživateli. Prostředí se tedy liší jenom v drobnostech, jako například v tom, že Alpha a Beta nabízí rozšířené možnosti zachycování a přepisování požadavků na backend, nebo poskytuje detailnější popisy chybových hlášek, které mohou obsahovat více technických informací.

Multi-platformní část a backend používají automatizační sestavovací nástroj *Gradle* [94], což je populární nástroj pro flexibilní konfiguraci a automatizaci sestavování softwarových projektů. Nativní iOS aplikace používá pro sestavení nástroj *xcodebuild*, který je součástí *Xcode* IDE. Multiplatformní knihovna je do nativní iOS aplikace propagována jako knihovna typu *xcframework*, což je *multiplatformní binární framework* [95]. *Gradle* zkompiluje multi-platformní kód do této *xcframework* reprezentace a zkopíruje ji do iOS projektu.

Implementace API backendové části aplikace dodržuje návrhové vzory *REST* [96], jedná se tedy o *RESTful API*. Pro možnosti rozšiřovatelnosti aplikace nabízí *Swagger OpenAPI* dokumentaci [97] na URL adrese `<backend-host-url>/openapi`, v případě nasazení v *Railway* aplikaci tedy na URL adrese `https://trackee-app-production.up.railway.app/openapi`. Zdrojem této dokumentace je soubor `openapi/documentation.yaml` ve složce `resources` ve zdrojovém kódu aplikace backendu.

Všechny části implementace (nativní aplikace, multi-platformní část a backend) jsou součástí jednoho repozitáře (*monorepo*), který má následující strukturu:

- `backend` – modul obsahující projekt backendu
- `build-logic` – společný modul ostatních modulů, obsahující pluginy a nástroje pro sestavení multi-platformní knihovny
- `gradle` - složka pro soubory nástroje *Gradle*
- `ios` – složka obsahující projekt nativní iOS aplikace
- `other` – nástroje projektu
- `shared` – modul obsahující sdílený kód pro technologii *Kotlin Multiplatform*
- `twine` – složka obsahující lokalizační soubor

Jednotlivé části implementace poté dodržují architekturu navrženou v sekci 3.3.1.

4.2.1 Lokalizace

Aplikace byla implementována a lokalizována pro tři jazyky – čeština, slovenština a angličtina. Zdroje pro lokalizace všech textů (tlačítka, popisky, instrukce, ...) jsou uloženy v souboru `strings.txt` ve složce `twine`, který je připraven ve formátu pro nástroj *Twine* [98], který tento soubor zpracovává a tvoří z něj lokalizační soubory pro cílové platformy (pro iOS jsou to soubory `Localizable.strings`). Nastavení tohoto nástroje pro tyto jazyky je již v šabloně *DevStack* připraveno.

4.2.2 Automatické generování kódu

Šablona také používá několik pomocných nástrojů pro automatické generování kódu. Jedním takovým nástrojem je *SwiftGen* [99], který zde slouží k několika účelům:

- Generování struktur pro lokalizaci: *SwiftGen* podle šablon generuje statické struktury z lokalizačních souborů `Localizable.strings` pro jednodušší použití v kódu, které zamezí použití špatných klíčů lokalizací.
- Generování struktur pro obrázky: *SwiftGen* je v šabloně nastaven takovým způsobem, aby ze zdrojů `Images.xcassets` generoval statické struktury, které umožní jednodušší referování obrázků ze zdrojového kódu.
- Generování struktur pro barvy: Funguje stejným způsobem, jako obrázky.

Nástroj *SwiftGen* je v šabloně přidán jako *plugin* modulu *UIToolkit* v prezentační vrstvě.

Nativní iOS aplikace také v šabloně využívá makra *swift-spyable* [100] pro generování *Use Case Mocks*, tedy statických náhrad pro *Use Cases* používaných pro testování. Toto je ale používáno jen pro nativní *Use Cases*, které aplikace Trackee neobsahuje. Pro sdílené *Use Cases* je využíváno vlastních rodičovských tříd `UseCaseResultMock`, `UseCaseResultNoParamsMock`, atd.

4.2.3 Dependency Injection

Pro *Dependency Injection* v nativní aplikaci je využíváno knihovny *Factory* [101], v multi-platformní části a na backendu knihovny *Koin* [102]. V aplikaci Trackee jsou sice všechny *Use Cases* a *Repositories* v multi-platformní části, ale *Dependency Injection* v prezentační vrstvě nativní aplikace získává implementace sdílených *Use Cases* knihovnou *Factory*, aby bylo dosazování sjednocené s případnými nativními *Use Cases*. *Dependency Injection* v nativní aplikaci tedy funguje tak, že v modulu *DependencyInjection* v aplikační vrstvě se registrují implementace všech závislostí, včetně sdílených *Use Cases*, které ale poskytuje knihovna *Koin*.

4.3 Implementace jednotlivých funkcionalit

Tato sekce se věnuje implementaci jednotlivých funkcionalit navržených v sekci 3.2. Implementaci každé funkcionality rozebírá z hlediska její realizace v backendové části, v multi-platformní části a v nativní aplikaci.

U jednotlivých funkcionalit mohou být uvedeny ukázky výpisů kódu, většinou tak, aby pro každou funkcionalitu byl ukázán příklad zdrojového kódu z jiné části aplikace. Cílem je poskytnout pár příkladů pro představu toho, jakým stylem je zdrojový kód implementován, a ne zbytečně ukazovat výpisy kódů pro věci, které jsou velmi podobné implementaci, která již byla ukázána v jiné části. Pro důkladnější studium kódu samozřejmě slouží zdrojový kód sám, který je součástí přiloženého média této práce. Text této práce slouží mimo jiné jako jeho dokumentace, která v něm má usnadnit orientaci.

4.3.1 Přihlášení a registrace

Přihlášení a registrace je základní funkcionalitou, kterou aplikace potřebuje, aby mohla držet informace o konkrétním uživateli. V návrhu platformy (obrázek 3.13) byla autentizace navržena tak, že bude probíhat přes backend. Ale vzhledem k tomu, že *Firebase* autentizace nabízí více možností autentizace, jako přihlášení přes Apple, Google, a podobně, tak byl v implementaci zvolen přístup, že autentizace bude probíhat na straně klienta, a ne backendu, protože jí celou obstará *Firebase* autentizace. Bylo by sice možné provádět autentizace přes e-mail a heslo tak, že by klient poslal danou kombinaci e-mailu a hesla na backend, který by uživatele přes *Firebase* přihlásil, klientovi vrátil jeho *access token*[1], který by ho poté dále používal při následujících požadavcích. V případě rozšíření o další formy přihlášení, jako je přihlášení přes Apple, by v tomto přístupu musel *Firebase* provést autentizaci u klienta, získat mezistavový *token* pro daného poskytovatele přihlášení, poslat ho na backend, který by ho poté ověřil s autentizací *Firebase*, získal *access token*, a poslal ho klientovi. Vzhledem k tomu, že takto by se autentizační API muselo volat jak u klienta, tak na backendu, byl zvolen přístup, že celý tento proces se bude dít u klienta pomocí knihovny pro *Firebase* autentizaci, a na backend se bude ve všech možnostech přihlášení posílat jen *access token*. V případě, že by v budoucnu padlo rozhodnutí tuto metodiku změnit a implementovat autentizační procesy pouze na backendu, musela by se tedy implementovat metodika popsaná výše.

4.3.1.1 Backend

Jak již bylo popsáno, backendová část nebude implementovat celý autentizační proces, ale jen bude ověřovat platnost *access tokenu*. To bylo naimplementováno pomocí kofigurace bezpečnosti knihovny *Ktor*. Pro všechny *Routes*, tedy *Endpoints*, které jsou definovány vnořeně ve funkci `authenticate`, se bude ověřovat, zda požadavek ve hlavičce obsahuje *Bearer token* (*access token*), ověří se přes *Firebase* autentizaci pomocí funkce `verifyIdTokenAsync`, poté se získá uživatel pomocí `uid` a dále budou moct *Routes* pracovat přímo s informacemi o přihlášeném uživateli. Průběh verifikace *access tokenu* lze vidět v ukázce kódu 1. V ukázce kódu 2 lze poté vidět, jak bude backend získávat objekt úspěšně ověřeného uživatele. Pokud je uživatel úspěšně ověřen, ale nemá záznam v databázi, bude mu tento záznam automaticky vytvořen – toto se bude dít při registraci uživatele.

Žádné *Routes* pro funkcionalitu přihlášení a registrace být definovány nemusí, jelikož tento proces bude probíhat přes *Firebase* autentizaci. V ukázce kódu 2 je ale vidět, že po úspěšném ověření *access tokenu* se aplikace pokusí získat objekt uživatele z databáze, případně nového uživatele vytvořit. Tyto funkce poskytuje `UserRepository`, která bude dále definovat řadu dalších funkcích souvisejících v práci s daty, které se týkají uživatele. Definice rozhraní tohoto repozitáře je součástí doménové vrstvy, implementaci poté definuje datová vrstva. Implementace repozitáře poté bude potřebovat pro přístup k datům *Source*, v tomto případě `UserSource`. Implementace tohoto *Source* už poté přímo poskytuje komunikaci s *Firestore* databází.

V ukázce kódu 3 je vidět implementace toho, jak se v databázi vytváří nový uživatel. Lze nahlédnout, že tato metoda vrací objekt úspěšně vytvořeného uživatele typu `FirestoreUser`. Jako parametr funkce ale bere typ `User`. `User` je totiž doménový objekt reprezentující uživatele, v databázi se používá databázový model nazvaný `FirestoreUser`, a mezi těmito objekty existují převáděcí funkce (`toFirestore()`, `toDomain()`). Další backendovou reprezentací uživatele je pak ještě `UserDto`, která se zase používá při komunikaci s klientskou aplikací. Pro tuto reprezentaci také existují převáděcí funkce (`toDto()`, `toDomain()`). V backendové části je potřeba používat všechny tyto tři typy reprezentací pro jeden typ objektu, protože pro interní práci s objekty slouží doménový objekt (tedy například `User`) a pro externí práci s objekty slouží DTO objekty pro jednotlivé části infrastruktury (tedy například `FirestoreUser` a `UserDto`).

[1] *Access token* je v autentizační terminologii řetězec znaků, se kterým klient posílá požadavky na server, aby prokázal svou identitu. [103]

```
class FirebaseAuthProvider(config: Config) : AuthenticationProvider(config) {

    // ...

    override suspend fun onAuthenticate(context: AuthenticationContext) {
        val authHeader = context.call.request.parseAuthorizationHeader()
            as? HttpAuthHeader.Single ?: throw AuthException.Unauthorized

        val token = try {
            firebase.verifyIdTokenAsync(authHeader.blob).await()
        } catch (e: FirebaseAuthException) {
            throw AuthException.InvalidToken(e)
        }

        val user = firebase.getUserAsync(token.uid).await()

        context.principal(FirebasePrincipal(user))

        val principal = context.mapPrincipal(user)
        if (principal != null) context.principal(principal)
    }

    // ...
}

fun AuthenticationConfig.firebase(
    name: String? = null,
    configure: FirebaseAuthProvider.Config.() -> Unit
) {
    val provider = FirebaseAuthProvider.Config(name).apply(configure).build()
    register(provider)
}
```

Výpis kódu 1 Průběh verifikace *access tokenu*

```
fun Application.configureSecurity() {

    // ...

    authentication {
        firebase {
            firebase = firebaseAuth
            mapPrincipal { userRecord ->
                try {
                    val user = userRepository.readUserByUid(userRecord.uid)
                    UserPrincipal(user)
                } catch(e: UserException.UserNotFound) {
                    val user = userRepository.createUser(userRecord.uid)
                    UserPrincipal(user)
                }
            }
        }
    }
}
```

Výpis kódu 2 Konfigurace bezpečnosti knihovny *Ktor*

```
internal class UserSourceImpl : UserSource {

    private val db = GoogleFirestoreClient.getFirestore()

    // ...

    override suspend fun createUser(user: User): FirestoreUser {
        db
            .collection(SourceConstants.Firestore.Collection.USERS)
            .document(user.uid)
            .set(user.toFirestore())
            .await()

        return db
            .collection(SourceConstants.Firestore.Collection.USERS)
            .document(user.uid)
            .get()
            .await()
            .toObject(FirestoreUser::class.java)
                ?: throw UserException.UserNotFound(user.uid)
    }

    // ...
}
```

Výpis kódu 3 Vytvoření nového uživatele v `UserSourceImpl`

4.3.1.2 Multi-platformní část

Pro potřeby přihlášení a registrace poskytuje multi-platformní knihovna následující *Use Cases*:

- `LoginWithCredentialsUseCase` – Poskytuje přihlášení pomocí e-mailu a hesla. Tuto funkcionalitu ale implementuje *Firebase* autentizace na straně nativní aplikace, multi-platformní kód tedy musí provolat sdílený `AuthProvider`, který implementuje nativní aplikace pomocí *Firebase* knihovny.
- `RegisterUseCase` – Registruje uživatele přes *Firebase* autentizaci, stejným způsobem jako přihlášení.
- `LogoutUseCase` – Odhlásí uživatele pomocí *Firebase* autentizace. Tento *Use Case* sice bude potřeba až v profilu uživatele, ale je součástí *auth* modulu multi-platformní části, je zde tedy zmíněn.
- `IsLoggedInUseCase` – Zjistí, zda je uživatel do aplikace přihlášený, nebo ne. Tento *Use Case* se zavolá při čistém spuštění aplikace, aby zjistila, zda má ukázat přihlašovací obrazovku, nebo uživateli rovnou ukázat časovač. Funguje tak, že se zeptá *Firebase* autentizace, zda má uložený *access token*.

API *Firebase* knihovny je potřeba provolávat pomocí sdíleného `AuthProvider` zpět do nativní aplikace, protože *Firebase* oficiálně nenabízí SDK pro *Kotlin Multiplatform*, ale pouze SDK pro nativní aplikace (*firebase-ios-sdk* [104] a *firebase-android-sdk* [105]). Existuje sice *firebase-kotlin-sdk* [106], která podporuje *Kotlin Multiplatform*, ale nejedná se o oficiální SDK a nemá 100% pokrytí celého *Firebase* API.

Žádné z těchto *Use Cases* neprovolávají *Endpoints* na backendu, ovlivňují ale to, jak budou ostatní požadavky vypadat, konkrétně obsah parametru pro autorizaci v hlavičce požadavků.

4.3.1.3 Nativní aplikace

Implementace přihlášení a registrace dodržuje vzhledový návrh ze sekce 3.2.1. Realizace viditelná na skutečném zařízení lze vidět na obrázku 4.1, kde lze i vidět jak se rozhraní přizpůsobí vysunuté klávesnici. Na obrázku 4.1a je také vidět, jakým způsobem ze zobrazí *Toast*[2] s chybovou hláškou.

4.3.2 Časovač

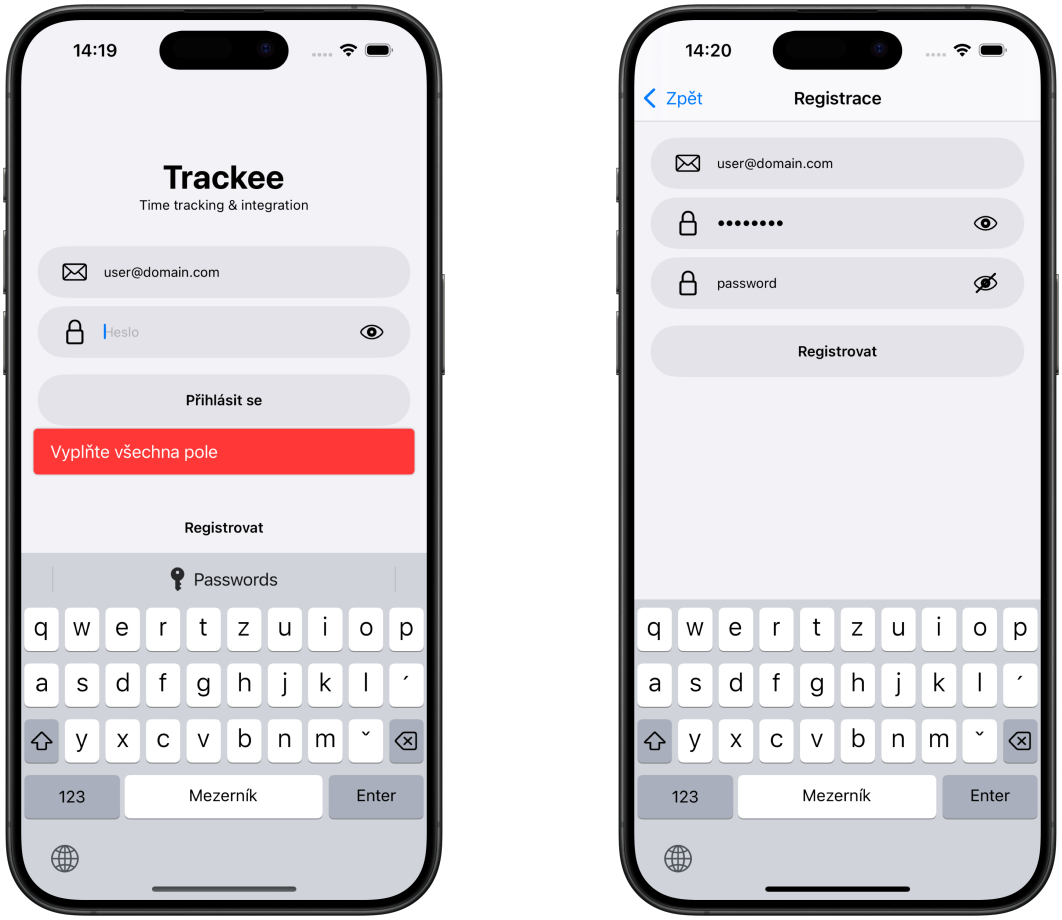
Tato funkcionalita pokrývá veškeré operace s ovládáním časovače, získáváním historie časových záznamů a vytváření nových záznamů. Jedná se o hlavní funkci aplikace.

4.3.2.1 Backend

Backendová část platformy bude muset spravovat veškerá data, která se týkají časovače a časových záznamů. Backend je ale rozdělený podle funkcionalit jiným způsobem, než nativní aplikace a multi-platformní část, které jsou rozděleny podle funkcionalit z hlediska uživatele. Moduly backendu jsou rozděleny podle toho, čeho se data v modulu týkají, tedy například klienti, projekty, integrace a uživatel. Jelikož pro data časovače budou potřeba data uživatele, u kterého se ukládá aktuální nastavení časovače a historie časových záznamů, ale také data projektů a klientů, bude implementace funkcionality časovače zasahovat do více modulů. Největší část funkcionality časovače bude obsluhovat modul uživatele.

Obrazovka časovače bude potřebovat následující data – souhrn odpracovaných hodin v aktuální den a týden, historii časových záznamů, aktuální nastavení časovače a přehled projektů, bude-li si uživatel chtít vybrat projekt, který časovači přidělí.

[2] *Toast* je UI komponentou z *DevStack* šablony sloužící pro dočasné zobrazovaní informací nebo chybových hlášek na obrazovce.



(a) Chyba při přihlášení **(b)** Registrace

Obrázek 4.1 Realizace přihlášení a registrace

V první řadě je tedy potřeba implementovat získávání historie časových záznamů, protože to bude potřeba i pro výpočet odpracovaných hodin v aktuálním dnu a týdnu. V návrhu v sekci 3.2.2 bylo popsáno, že načítání historie časových záznamů by mělo podporovat stránkování, jelikož může teoreticky obsahovat velké množství záznamů, a pokud by se mělo velké množství načítat celé najednou, mohlo by to trvat dlouho a mohlo by se jednat o velké množství dat, která by ale uživatel nejspíš ani všechna vůbec nepotřeboval. `UserRepository` tedy definuje mimo jiné funkce pro získávání záznamů, které lze nahlédnout v ukázce kódu 4. Tyto funkce potřebují parametr `uid` identifikující uživatele, pro kterého mají být záznamy čteny, a dále přijímají nepovinné parametry pro specifikaci toho, od kdy a do kdy mají být záznamy čteny, a kolik maximálně záznamů se ve stránce má nacházet. Pokud bude mít některý z těchto nepovinných parametrů hodnotu `null`, nebude žádné omezení na záznamy klást. Rozdíl mezi funkcemi, které vrací stránky objektu `TimerEntry` a stránky objektu `TimerEntryPreview` je ten, že objekt `TimerEntryPreview` obsahuje navíc celé zdrojové objekty klienta a projektu, který k záznamu patří, zatímco `TimerEntry` obsahuje pouze jejich identifikátory. V obrazovce pro zobrazení historie záznamů budou tyto zdrojové objekty potřeba, protože při jejich vizualizaci (při jejich *preview*) bude potřeba zobrazit jméno klienta a projektu.

V ukázce kódu 5 lze poté vidět, jakým způsobem aplikace získává záznamy přímo z *Firestore* databáze. Omezení ve smyslu od kdy do kdy záznamy číst a kolik maximálně jich přečíst se implementuje pomocí funkcí `startAfter(start)`, `endAt(end)` a `limit(limit)`, které přímo

```
interface UserRepository {

    // ...

    suspend fun readEntries(
        uid: String,
        startAfter: Instant?,
        limit: Int?,
        endAt: Instant?
    ): Page<TimerEntry>

    // ...

    suspend fun readEntryPreviews(
        uid: String,
        startAfter: Instant?,
        limit: Int?,
        endAt: Instant?
    ): Page<TimerEntryPreview>

    // ...
}
```

Výpis kódu 4 Funkce pro získávání časových záznamů v `UserRepository`

nabízí *Firestore* API. `UserRepositoryImpl` používá tuto *Source* funkci i pro načtení *preview* objektů, objekty klienta a projektu si pak pomocí identifikátoru načte sama a připojí je.

Při načítání historie záznamů na obrazovce časovače pak bude potřeba použít parametry `startAfter`, abychom definovali datum a čas, od kterého další záznamy načítat, pokud načítáme další stránky, a `limit`, který určí velikost stránky. Při načítání všech záznamů v daném období se pak využije parametrů `startAfter` a `endAt`, například když bude potřeba zjistit souhrn odpracovaných hodin v aktuální den nebo týden. Při výpočtu tohoto souhrnu bude tedy `UserRepositoryImpl` sčítat doby trvání všech záznamů v daném období.

Pro obsluhu časovače pak `UserRepository` poskytuje funkce pro načtení dat časovače a pro jejich aktualizaci. Při výběru projektu bude uživatel potřebovat vidět všechny své projekty a názvy jejich klientů, pro což bude sloužit opět *preview* objekt `ProjectPreview`. `ClientRepository` tedy nabízí funkce pro získání klienta podle identifikátoru, a `UserRepository` nabízí funkce pro získání projektu podle identifikátoru klienta a projektu, ale také funkci pro získání všech projektů daného uživatele.

Pro komunikaci s klientem se také využívá vlastních struktur pro reprezentaci chyb. Například modul `user` používá výjimky ze skupiny `UserExceptions`, která dědí z třídy `BaseException`. V případě, že některé volání vrátí výjimku s tímto rodičem, tak ji komunikace umí zakódovat do DTO reprezentace, která je známá pro klienta. Ten může potom rozlišovat mezi různými známými chybami. V případě, že se nejedná o známou chybu, zakóduje ji komunikace jako obecnou chybu s HTTP status kódem 500. Implementace tohoto chování lze nahlédnout ve výpisu kódu 6.

4.3.2.2 Multi-platformní část

Multi-platformní část aplikace už rozděluje své moduly podle funkcionalit z hlediska uživatele, modul `timer` bude tedy poskytovat všechny *Use Cases* pro potřeby časovače:

```
internal class UserSourceImpl : UserSource {

    // ...

    override suspend fun readEntries(
        uid: String,
        startAfter: Instant?,
        limit: Int?,
        endAt: Instant?
    ): Page<FirestoreTimerEntry> {
        val entriesCollection = db
            .collection(SourceConstants.Firestore.Collection.ENTRIES)
            .document(uid)
            .collection(SourceConstants.Firestore.Collection.ENTRIES)
            .orderBy(
                SourceConstants.Firestore.FieldName.STARTED_AT,
                Query.Direction.DESCENDING
            )

        val snapshot = entriesCollection
            .startAfter(startAfter?.toTimestamp() ?: Timestamp.now())
            .endAt(endAt?.toTimestamp() ?: Timestamp.MIN_VALUE)
            .limit(limit ?: Int.MAX_VALUE)
            .get()
            .await()

        val data = snapshot.documents.map {
            it.toObject(FirestoreTimerEntry::class.java)
        }

        val remainingCount = entriesCollection
            .startAfter(data.lastOrNull()?.startedAt ?: Timestamp.MIN_VALUE)
            .count()
            .get()
            .await()
            .count

        return Page(
            data = data,
            isLast = remainingCount == 0.toLong()
        )
    }

    // ...
}
```

Výpis kódu 5 Funkce pro získávání časových záznamů v `UserSourceImpl`

```
fun Application.configureRouting(isDebug: Boolean) {
    install(StatusPages) {
        exception<BaseException> { call, baseException ->
            call.respond(
                status = baseException.code,
                baseException.toDto(isDebug)
            )
        }

        exception<Throwable> { call, cause ->
            call.respond(
                status = HttpStatusCode.InternalServerError,
                message = ErrorDto(
                    type = "InternalError",
                    message = "Internal Server Error",
                    debugMessage = if (isDebug) cause.message else null
                )
            )
        }
    }

    // ...
}
```

Výpis kódu 6 Obsluha chyb na backendu

- `AddTimerEntryUseCase` – Vytvoří nový časový záznam podle parametrů.
- `DeleteTimerEntryUseCase` – Smaže časový záznam podle identifikátoru.
- `GetProjectsUseCase` – Získá *preview* objekty všech projektů, které má uživatel přiřazeny.
- `GetTimerDataPreviewUseCase` – Získá *preview* objekt pro aktuální nastavení časovače. Čistá data o aktuálním stavu časovače totiž opět obsahují jen identifikátory klienta a projektu, ale při vizualizaci dat jsou potřeba jejich názvy.
- `GetTimerEntriesUseCase` – Získává stránky *preview* objektů časových záznamů, umí tedy pracovat se všemi parametry pro omezení stránky. Zároveň časové záznamy seskupuje do seznamu objektů typu `TimerEntryGroup`, což je skupina, která obsahuje datum, seznam všech záznamů patřící k tomuto datu a součet odpracovaných hodin všech těchto záznamů. Toho se využije při vizualizaci v nativní aplikaci, kde se nad každou skupinou ukáže datum a časový souhrn pro daný den.
- `GetTimerSummariesUseCase` – Získá souhrny časových záznamů pro aktuální den a týden.
- `UpdateTimerDataUseCase` – Aktualizuje aktuální nastavení časovače.

Všechny tyto *Use Cases* automaticky počítají s tím, že pracují s daty aktuálně přihlášeného uživatele, žádné parametry pro jeho identifikaci tedy nepotřebují.

Jednotlivé *Use Cases* už komunikují s backendem, a to v nejnižší *infrastructure* vrstvě, která požadavky posílá pomocí knihovny *Ktor*. Příklad toho, jak probíhá komunikace, lze nahlédnout v ukázce kódu 7.

```
internal class RemoteTimerSource(
    private val client: HttpClient
) : TimerSource {
    override suspend fun readEntries(
        startAfter: String?,
        limit: Int?,
        endAt: String?
    ): Result<PageDto<TimerEntryDto>> =
        runCatchingCommonNetworkExceptions {
            val res = client.get("user/entries") {
                url {
                    startAfter?.let { parameters.append("startAfter", it) }
                    limit?.let { parameters.append("limit", it.toString()) }
                    endAt?.let { parameters.append("endAt", it) }
                }
            }
            res.body<PageDto<TimerEntryDto>>()
        }

    // ...
}
```

Výpis kódu 7 Funkce pro získávání časových záznamů v `RemoteTimerSource`

Multi-platformní část pracuje s objekty v DTO reprezentaci, kterou definuje backend. Poté si je také převádí pomocí vlastních funkcí do vlastních doménových reprezentací.

Také lze ve výpisu kódu 7 nahlédnout, že celé API volání je obaleno do pomocné funkce `runCatchingCommonNetworkExceptions`, jejíž implementace lze nahlédnout ve výpisu 8. Tato funkce odchytává všechny výjimky, které při komunikaci s backendem mohou vzniknout, a převádí je do vlastních reprezentací, se kterými poté umí pracovat nativní aplikace.

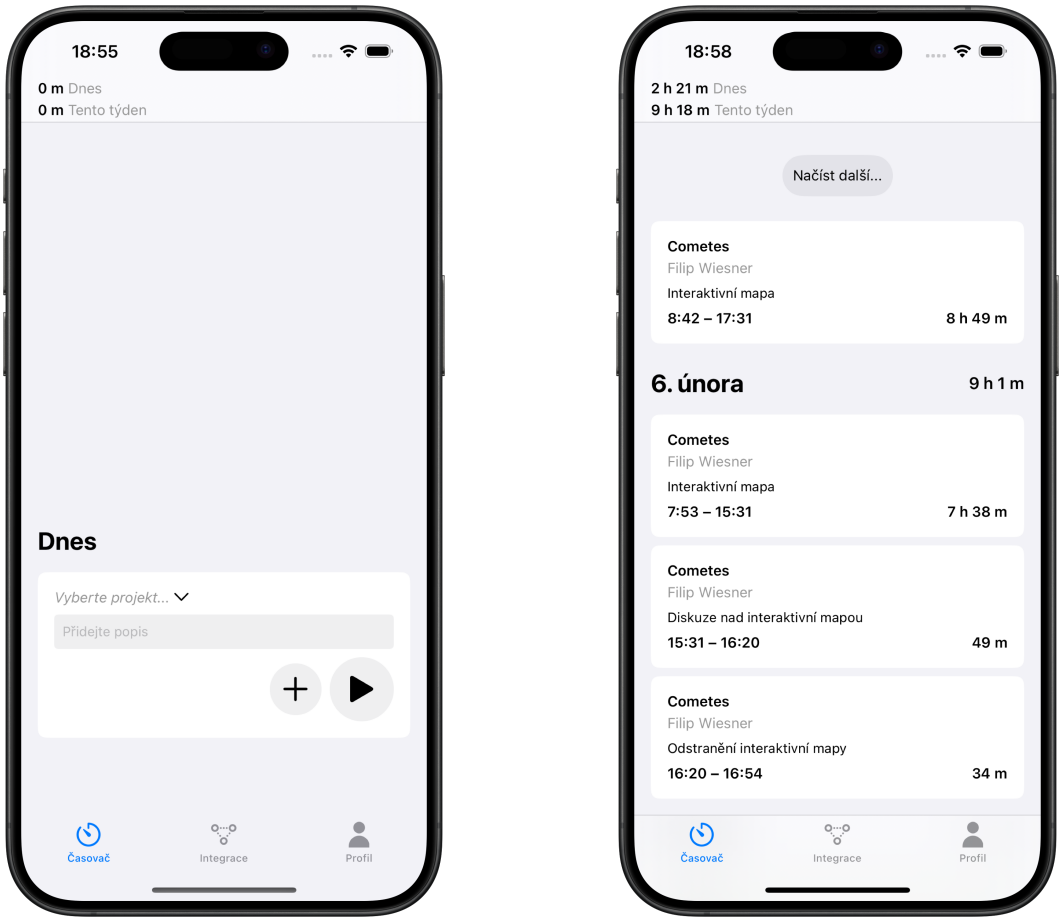
4.3.2.3 Nativní aplikace

Realizace časovače dodržuje vzhled navržený v sekci 3.2.2. Na obrázku 4.2a lze nahlédnout obrazovka časovače, jak bude vypadat, pokud uživatel zatím nemá žádná data, jako například nově registrovaný uživatel. Na obrázku 4.2b lze zase nahlédnout, jak vypadá tlačítko pro načtení dalších záznamů, pokud se uživatel posune v načtených datech úplně nahoru. Na obrázku 4.3a lze nahlédnout stav, kdy si uživatel chce změnit popisek časovače, a na obrázku 4.3b zase obrazovka pro ruční vybrání času, pokud chce uživatel přidat záznam manuálně. Na obrázku 4.4 lze poté vidět výběr z projektů a možnost v nich vyhledávat.

Jeden problém, který musí nativní aplikace řešit, je situace, když má zobrazit nově načtenou navazující stránku časových záznamů. Můžou nastat dvě situace – buď nová stránka bude obsahovat data skupiny, která má prázdný průnik s poslední skupinou již načtených dat (tedy již načtená data neobsahují data nějakého dne, který by neměl plně načtené všechny záznamy), nebo bude tento průnik neprázdný (tedy nová data obsahují data, která se musí spojit s daty nějakého dne, který je již z části načtený). Nově načtenou stránku tedy nelze vždy obyčejně připojit za již načtená data, ale musí probíhat kontrola, zda se nějaká skupina z již načtených dat nemá spojit s nějakou skupinou nových dat. Multi-platformní část také musí poskytovat data o tom, zda je jednotlivá skupina již plně načtená. Jediný případ, kdy o tomto faktu může výstup multi-platformního *Use Case* lhát, je ten, pokud se jedná o poslední skupinu v načtených datech, jejíž konec ale přesně končí v místě, kde končí i záznamy dalšího dne, což ale *Use Case* nemá jak

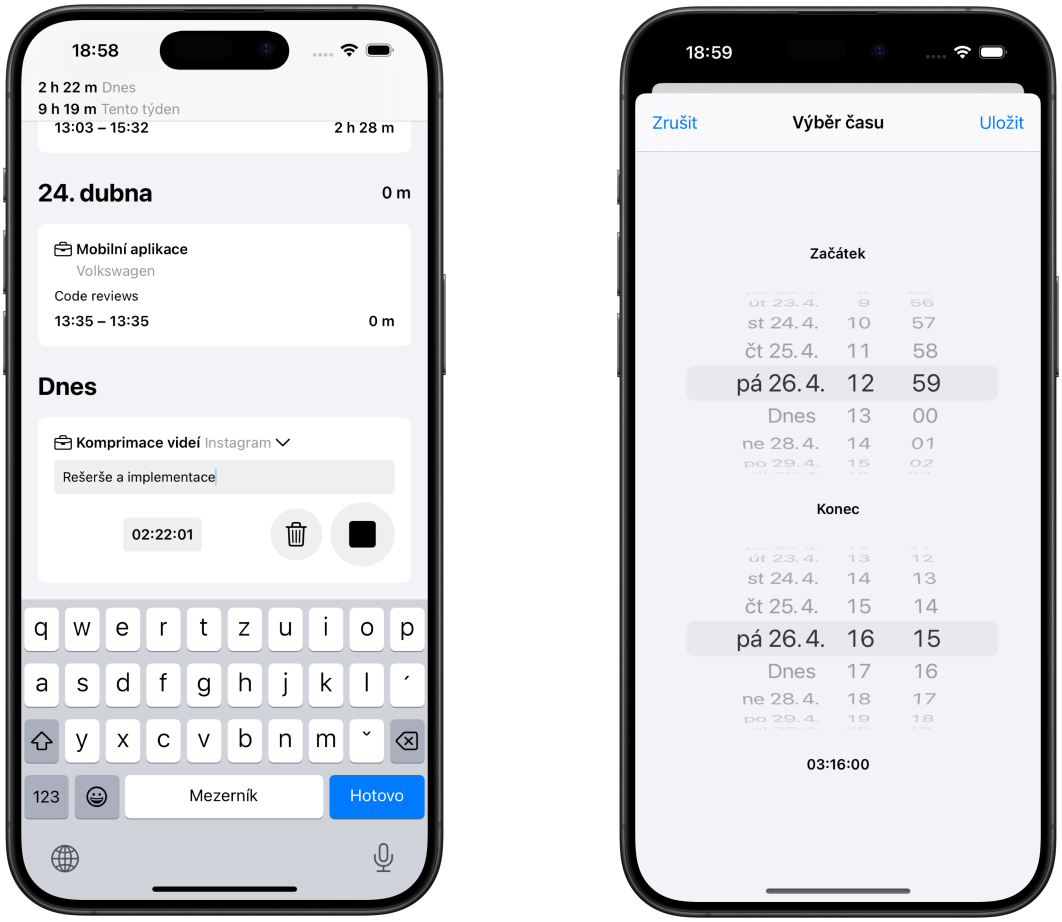
```
internal suspend inline fun <R : Any> runCatchingCommonNetworkExceptions(
    block: () -> R
): Result<R> =
    try {
        Result.Success(block())
    } catch (e: ResponseException) {
        val body = e.response.body<ErrorDto>()

        val error = when (body.type) {
            "Unauthorized" -> BackendError.NotAuthorized(e.response.toString(), e)
            "ProjectNotAssignedToUser" -> BackendError.ProjectNotAssignedToUser(
                body.message,
                e
            )
            "MissingProject" -> BackendError.MissingProject(body.message, e)
            "ProjectNotFound" -> BackendError.ProjectNotFound(body.message, e)
            "ClientNotFound" -> BackendError.ClientNotFound(body.message, e)
            "ClockifyProjectNotFound" -> BackendError.ClockifyProjectNotFound(
                body.message,
                e
            )
            "ClockifyInvalidApiKey" -> BackendError.ClockifyInvalidApiKey(e)
            "ClockifyUnknownError" -> BackendError.ClockifyUnknownError(e)
            "ClockifyWorkspaceNotFound" -> BackendError.ClockifyWorkspaceNotFound(
                body.message,
                e
            )
            else -> ErrorResult(message = body.message, throwable = e)
        }

        Result.Error(error)
    } catch (e: Throwable) {
        val error = when (e::class.simpleName) {
            "UnknownHostException" -> CommonError.NoNetworkConnection(e)
            "HttpRequestTimeoutException", "ConnectTimeoutException",
            "SocketTimeoutException" -> CommonError.Timeout(e)
            "CancellationException" -> CommonError.Cancelled(e)
            else -> handlePlatformError(e)
        }
        Result.Error(error)
    }
```

Výpis kódu 8 Odchytávání výjimek při komunikaci s backendem



(a) Prázdná data **(b)** Načtení další stránky

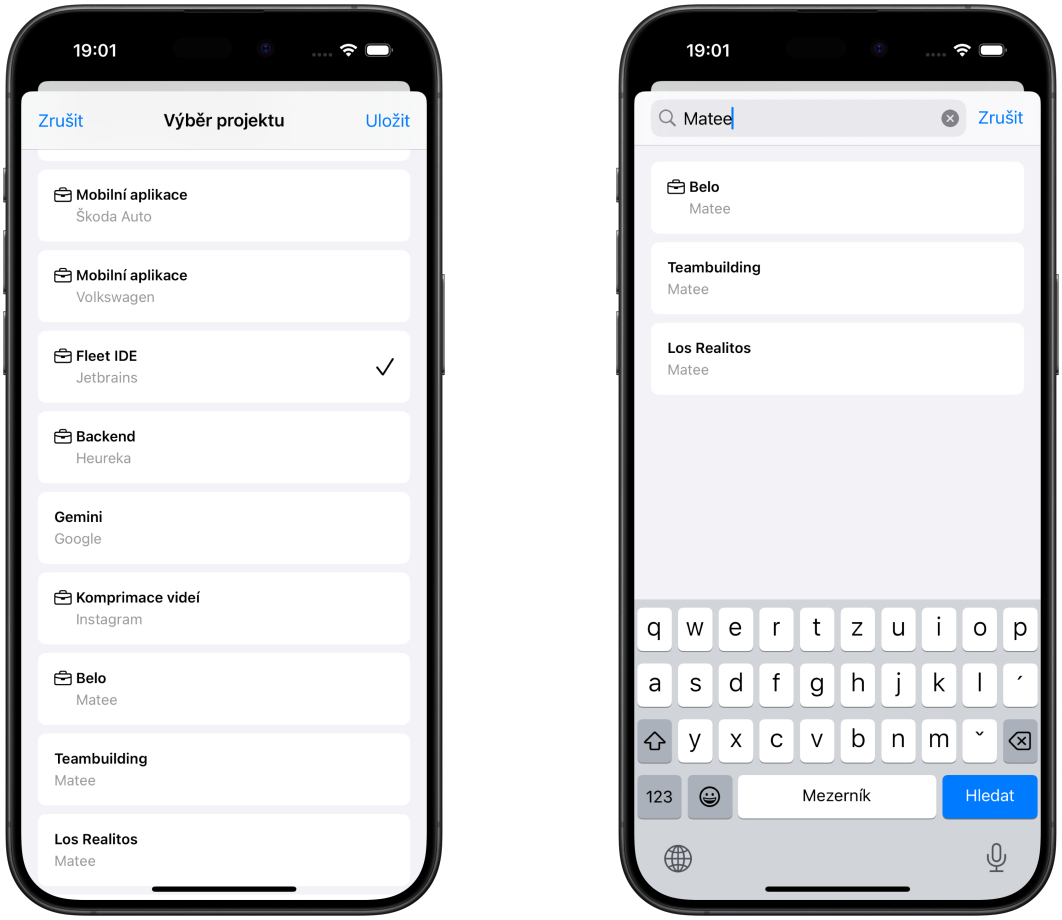
Obrázek 4.2 Realizace časovače



(a) Úprava popisu

(b) Manuální výběr času

Obrázek 4.3 Realizace ovládání časovače



(a) Vybraný projekt **(b)** Vyhledávání

Obrázek 4.4 Realizace výběru projektu

zjistit. Tento případ musí ručně detekovat nativní aplikace při načtení nové stránky a opravit skupinu, u které to mohlo nastat, tak, že jí označí za plně načtenou. Implementace této logiky lze nahlédnout ve výpisu kódu 9.

Veškerá ostatní logika spočívá v přímočarém používání *Use Cases* podle toho, co je úmyslem uživatele. *View Model* této obrazovky si mimo to musí pamatovat ručně zadaný konec záznamu, jelikož ten není součástí aktuálního nastavení časovače, nebo také musí každou sekundu aktualizovat čas, který se ukazuje na běžících stopkách, ale také tento čas přičítat k souhrnům aktuálního dne a týdne.

4.3.3 Profil uživatele

Tato funkcionalita pokrývá operace, které může uživatel dělat se svým profilem, se svými klienty a se svými projekty.

4.3.3.1 Backend

Obsluhu požadavků týkajících se projektů bude řešit modul pro projekty, požadavky týkající se klientů bude řešit modul pro klienty a požadavky týkající se profilu bude řešit modul pro uživatele.

Co se týče funkcí, co může uživatel dělat se svým profilem, tak se na obrazovce profilu může buď odhlásit, nebo si svůj účet smazat.

Odhlášení je čistě autentizační záležitost, probíhá tedy pouze na straně klienta, kde je součástí modulu `auth`, jak bylo popsáno v sekci 4.3.1.

Smazání účtu už je ale z hlediska dat zajímavější záležitost. Během mazání účtu bude potřeba smazat všechna data, která jsou s uživatelem propojená, tedy všechny jeho záznamy a integrace. V první řadě bude tedy potřeba smazat každý dokument v kolekci `/entries/{uid}/entries`, protože pro smazání kolekce je potřeba smazat každý dokument, nelze smazat celou kolekci najednou. Poté se bude moct smazat dokument `/entries/{uid}`. Poté se budou muset smazat všechny integrace, tedy všechny dokumenty v kolekci `/users/{uid}/integrations`, poté všechny informace o tom, jaké klienty a projekty má uživatel přiřazené, tedy dokumenty v kolekci `/users/{uid}/clients` a ve finále se může smazat dokument uživatele, tedy `/users/{uid}`. Tím budou smazána všechna data související s uživatelem ve *Firestore* databázi. Poté je ještě potřeba smazat uživatele z *Firebase* autentizace, aby se na něj už nešlo přihlásit. Celou tuto logiku implementuje funkce `deleteUser(uid: String)` v `UserSourceImpl`.

Další, co bude backend muset implementovat, jsou všechny CRUD (Create, Read, Update, Delete) operace pro klienty a projekty. Za to mají odpovědnost příslušné `ProjectRepository`, `ClientRepository` a jejich *Sources*. Specifikem projektů je, že pro jejich jednoznačnou identifikaci nestačí pouze identifikátor projektu, ale je potřeba i identifikátor klienta, ke kterému patří. Také je potřeba implementovat funkce pro přečtení všech klientů pro konkrétního uživatele a přečtení všech projektů daného uživatele. Poté je také potřeba poskytnout API pro přiřazení klienta nebo projektu k uživateli, samotné vytvoření je totiž pouze zařadí do kolekcí, ale nepřiřadí je k žádnému uživateli. Implementace *Route*, která obsluhuje přiřazení projektu k uživateli, lze nahlédnout ve výpisu kódu 10. Toto přiřazení řeší speciální *PUT endpoint*, který nepřijímá žádné tělo požadavku.

Úprava a mazání projektů a klientů je také poměrně komplexní záležitost, jelikož je spolu s nimi potřeba upravit nebo smazat řadu dat, které s nimi souvisí. Například, pokud se u upravovaného projektu změní klient, ke kterému patří, bude potřeba změnit všechny časové záznamy, které tento projekt obsahují, aby měly identifikátor nového klienta, dále je potřeba u všech uživatelů, kteří měli tento projekt přiřazený, toto přiřazení přesunout z dokumentu původního klienta do dokumentu nového klienta, je potřeba samotný projekt přesunout do kolekce nového klienta, a ověřit aktuální data časovače u všech uživatelů, protože tam tento projekt se starým klientem může být. Podobně složité může být například mazání klienta. Spolu s ním je totiž

```
// Fetch more groups
let fetchedGroups: [TimerEntryGroup] = try await getTimerEntriesUseCase.execute(
    params: params
)

var newGroups: [TimerEntryGroup] = []

// If there is an overlapping group
if let firstOfCurrent = currentGroups.first,
   let lastOfNew = fetchedGroups.last,
   firstOfCurrent.date == lastOfNew.date {
    // Append already fetched groups without the last one
    newGroups.append(contentsOf: fetchedGroups.dropLast())

    // Calculate total interval of the overlapping group
    var interval: KotlinLong? {
        if lastOfNew.interval == nil
            && firstOfCurrent.interval == nil
        { return nil }

        let lastOfNewInterval = lastOfNew.interval?.int ?? 0
        let firstOfCurrentInterval = firstOfCurrent.interval?.int ?? 0

        return KotlinLong(
            value: (lastOfNewInterval + firstOfCurrentInterval).int64
        )
    }

    // Append union of the overlapping group
    newGroups.append(TimerEntryGroup(
        date: firstOfCurrent.date,
        interval: interval,
        entries: lastOfNew.entries + firstOfCurrent.entries,
        isFullyLoaded: lastOfNew.isFullyLoaded
    ))

    // Append the fetched groups without the first one
    newGroups.append(contentsOf: currentGroups.dropFirst())
} else {
    // No overlapping groups, just append
    newGroups.append(contentsOf: fetchedGroups)

    // Fix the `isFullyLoaded` flag if necessary
    if let firstOfCurrent = currentGroups.first, !firstOfCurrent.isFullyLoaded {
        firstOfCurrent.isFullyLoaded = true
    }
    newGroups.append(contentsOf: currentGroups)
}

// Display the updated data
state.listData = .data(newGroups)
```

Výpis kódu 9 Obsluha nově načtené navazující stránky záznamů

```
fun Routing.userRoute() {

    // ...

    authenticate {

        // ...

        route("/user") {

            // ...

            route("/projects") {

                // ...

                put("/add") {
                    val user = call.requireUserPrincipal().user
                    val clientId: String by call.request.queryParameters
                    val projectId: String by call.request.queryParameters

                    userRepository.assignProjectToUser(user.uid, clientId, projectId)

                    call.respond(HttpStatusCode.OK)
                }
            }

            // ...

        }
    }
}
```

Výpis kódu 10 *Route* pro přiřazení klienta k uživateli

potřeba smazat u všech uživatelů případná přiřazení tohoto klienta a jeho projektů, smazat všechny projekty, které ke klientovi patří, smazání všech časových záznamů všech uživatelů, kteří mají tohoto klienta přiřazeného, a až poté smazat klienta. Všechnu tuto logiku definují implementace jednotlivých *Sources*, tedy `ProjectSourceImpl` a `ClientSourceImpl`.

4.3.3.2 Multi-platformní část

Multi-platformní část aplikace poskytuje *Use Cases* pro všechny funkce popsané výše, tedy:

- `AddAndAssignClientUseCase` – Přiřadí klienta k přihlášenému uživateli.
- `AddAndAssignProjectUseCase` – Přiřadí projekt k přihlášenému uživateli.
- `DeleteUserUseCase` – Smaže uživatele podle identifikátoru.
- `GetClientsUseCase` – Získá všechny klienty přiřazené k přihlášenému uživateli.
- `GetClientUseCase` – Získá klienta podle jeho identifikátoru.
- `GetProjectPreviewUseCase` – Získá *preview* objekt pro projekt podle jeho identifikátoru.
- `GetUserEmailUseCase` – Získá e-mail přihlášeného uživatele, který se pak zobrazuje v obrazovce profilu. Tuto informaci nezískává z backendu, ale pomocí *Firebase* autentizace. E-mail uživatele není ve *Firestore* databázi vůbec uložen, jedná se pouze o autentizační nástroj.
- `RemoveClientUseCase` – Smaže klienta podle identifikátoru.
- `RemoveProjectUseCase` – Smaže projekt podle identifikátoru.
- `UpdateClientUseCase` – Aktualizuje klienta.
- `UpdateProjectUseCase` – Aktualizuje projekt.

4.3.3.3 Nativní aplikace

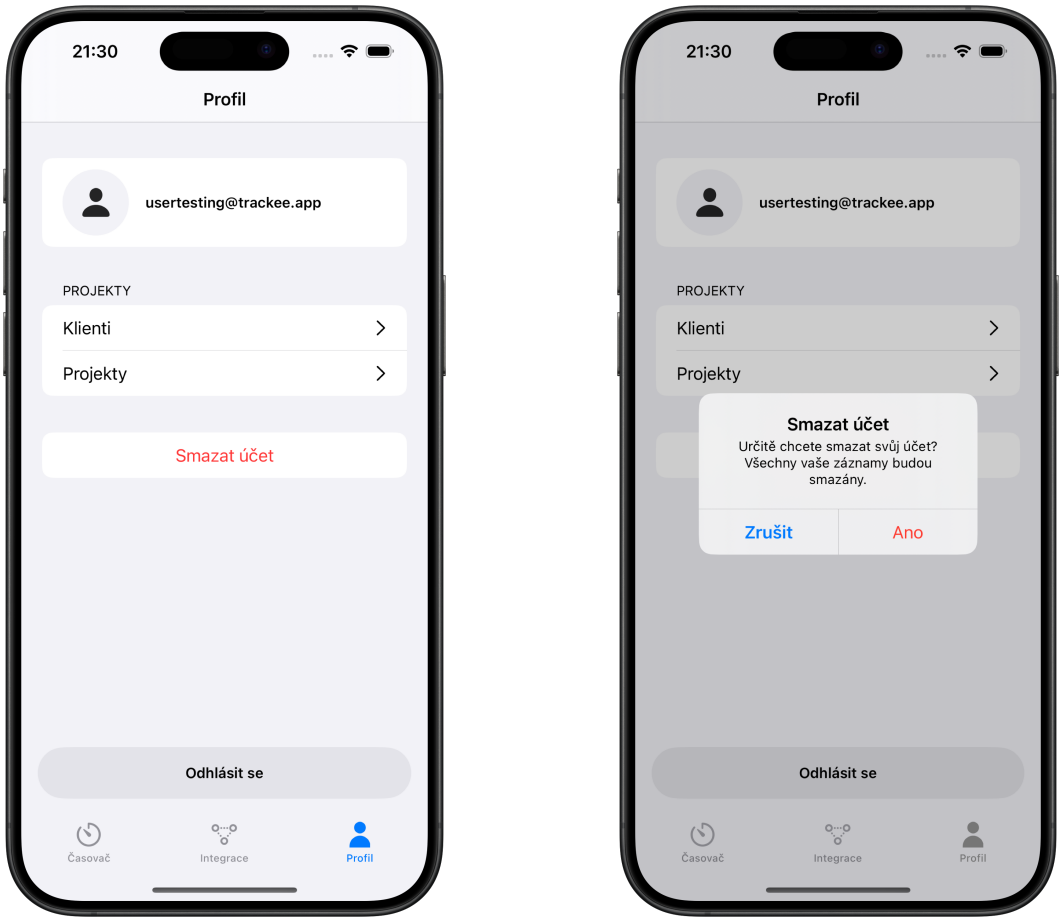
Realizace profilu v iOS aplikaci dodržuje návrh ze sekce 3.2.3. Na obrázku 4.5 lze nahlédnout přehled profilu a dialog, který se zobrazí při pokusu o smazání účtu. Na obrázku 4.6 lze poté vidět seznam klientů, které má uživatel přiřazené, a možnost v nich vyhledávat. Dále lze na obrázku 4.7 vidět detaily klienta a projektu, obrázek 4.7b navíc ukazuje stav detailu během ukládání změn, během kterého s prvky nelze interagovat. Poslední obrázek 4.8 poté znázorňuje seznam projektů a výběr klienta k projektu.

4.3.4 Integrace

Tato funkcionalita pokrývá možnosti, jak může uživatel napojovat aplikaci na navržené spouštěče měření času (import) a jak propojit aplikaci s existujícími systémy (export), jak bylo navrženo v sekci 3.2.4.

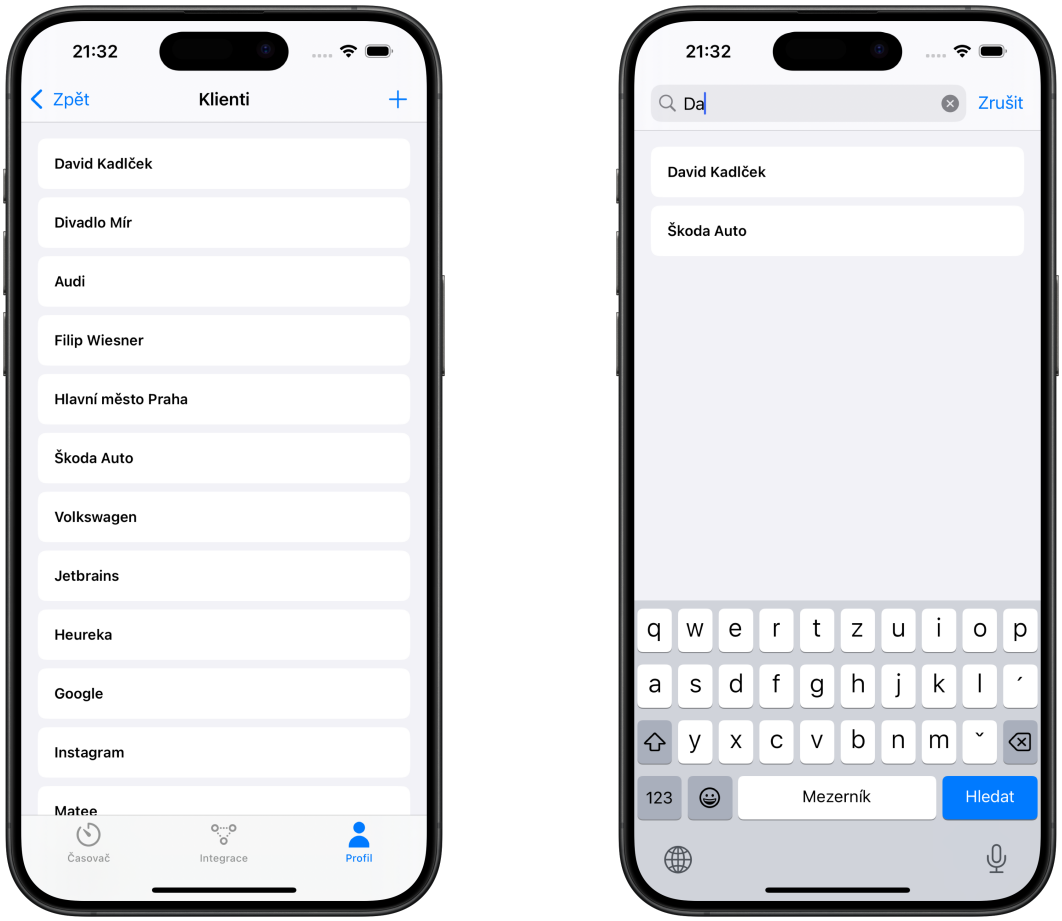
4.3.4.1 Backend

Aplikace uživateli umožňuje si nastavené integrace vytvářet, ukládat, upravovat a mazat. Backend bude tedy muset opět implementovat všechny CRUD operace a získání všech integrací daného uživatele, tentokrát pro objekty integrací. Toto je odpovědností `IntegrationRepository` a jejího *Source*. Na rozdíl od uživatelů, klientů nebo projektů, nejsou integrace propojeny s jinými typy objektů napříč celou databází, jejich úprava je tedy poměrně jednoduchá a týká se jen vlastních objektů.



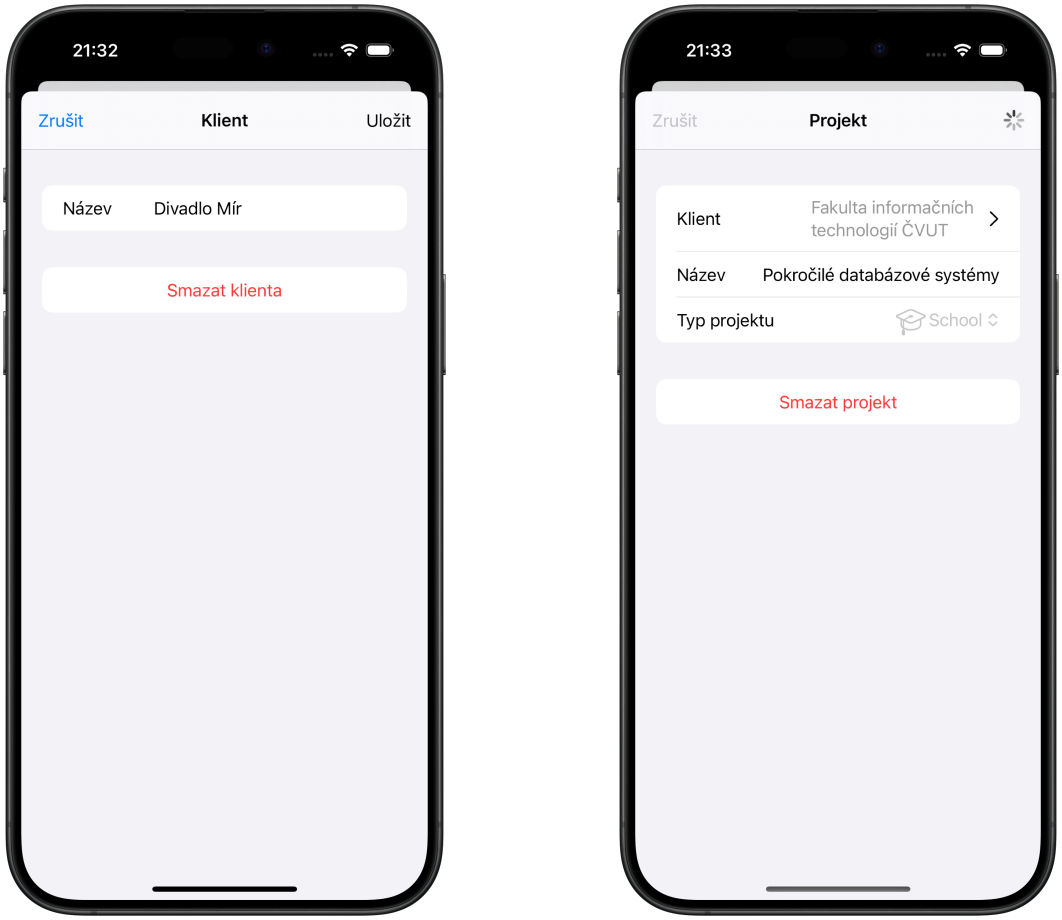
(a) Přehled **(b)** Dialog pro smazání účtu

Obrázek 4.5 Realizace profilu



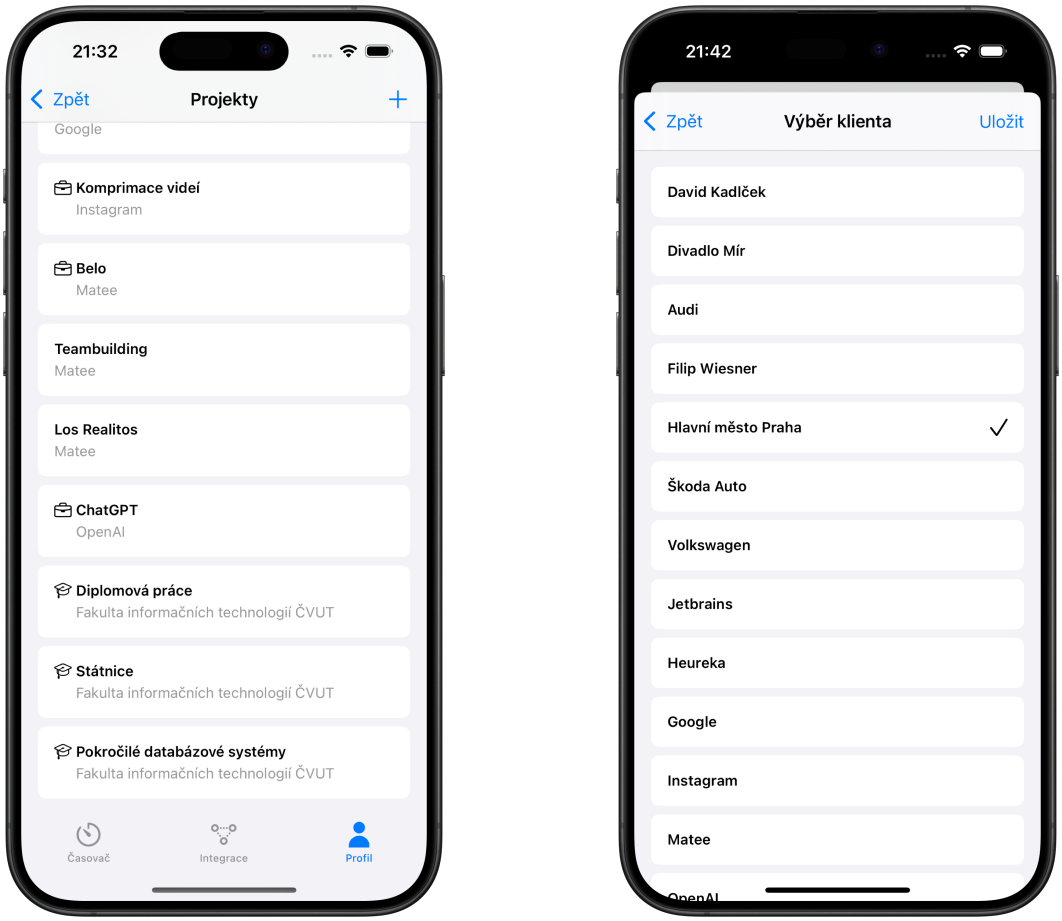
(a) Seznam **(b)** Vyhledávání

Obrázek 4.6 Realizace klientů



(a) Detail klienta **(b)** Detail projektu během ukládání

Obrázek 4.7 Detail klienta a projektu



(a) Seznam projektů **(b)** Výběr klienta k projektu

Obrázek 4.8 Seznam projektů a výběr klienta k projektu

```
fun Routing.integrationRoute() {

    // ...

    authenticate {
        route("/integrations") {

            // ...

            route("/csv") {
                get {
                    val user = call.requireUserPrincipal().user
                    val from = call.request.queryParameters["from"]
                    val to = call.request.queryParameters["to"]

                    val entries = userRepository.readEntryPreviews(
                        uid = user.uid,
                        startAfter = to?.toInstant(),
                        limit = null,
                        endAt = from?.toInstant()
                    ).data
                    val csv = repository.readCsv(entries.reversed())

                    call.respondFile(csv)

                    repository.deleteTempCsvFile(csv.name)
                }
            }

            // ...
        }
    }
}
```

Výpis kódu 11 *Route* pro export do CSV souboru

Co se týče exportu do CSV souboru, tak celou tuto logiku implementuje backend, který CSV soubor vytvoří a klientovi ho pošle hotový. Ve výpisu kódu 11 lze nahlédnout implementace *Route* pro export do CSV souboru, kde lze vidět, že nejprve získá všechny záznamy v zadaném období, ty uloží do dočasného lokálního souboru, který pošle klientovi, a poté dočasný soubor smaže.

Napojení aplikace na spouštěče měření (import), které byly pro aplikaci Trackee navrženy, je záležitostí klienta, jelikož se jedná o systémovou aplikaci *Zkratky*. Bude pouze potřeba poskytnout dodatečné API, které umožní samotné zapnutí časovače, jeho zrušení a jeho vypnutí spolu s vytvořením nového časového záznamu. Tyto funkce totiž pro potřeby funkcionality časovače implementuje klientská aplikace ručně, která aktualizuje aktuální nastavení časovače, čímž reálně ovlivňuje to, zda časovač běží, od kdy běží, a podobně. Při tvorbě nového záznamu také volá pouze API pro ruční vytvoření nového záznamu, protože všechna potřebná data zná. Při použití zkratek se jedná pouze o jednoduchý požadavek, který ale nemá žádné informace o aktuálním nastavení časovače, takže pro tyto potřeby bude tato logika na backendu. Alternativou by bylo,

aby vyvolané *Zkratky* přímo interagovaly s rozhraním aplikace, ale to by nedávalo moc smysl, protože v tomto případě je tím rozhraním právě aplikace *Zkratky*.

4.3.4.2 Multi-platformní část

Multi-platformní část pokrývá funkcionalitu integrací ve dvou modulech – `integration` a `intent`. Modul `integration` pokrývá všechny funkce, které aplikace nabízí v kartě *Integrace* (návrh z obrázku 3.11a), a modul `intent` nabízí funkce pro *Intents*, což jsou obsluhy jednotlivých zkratek aplikace *Zkratky*, jak bylo popsáno v sekci 3.2.4.1.

Modul `integration` poskytuje tyto *Use Cases*:

- `AddIntegrationUseCase` – Přidá novou integraci.
- `DeleteIntegrationUseCase` – Smaže integraci podle jejího identifikátoru.
- `ExportToCsvUseCase` – Exportuje data ve zvoleném období do CSV souboru.
- `GetIntegrationsUseCase` - Získá všechny integrace přihlášeného uživatele.
- `GetIntegrationUseCase` – Získá integraci podle jejího identifikátoru.
- `UpdateIntegrationUseCase` – Aktualizuje integraci.

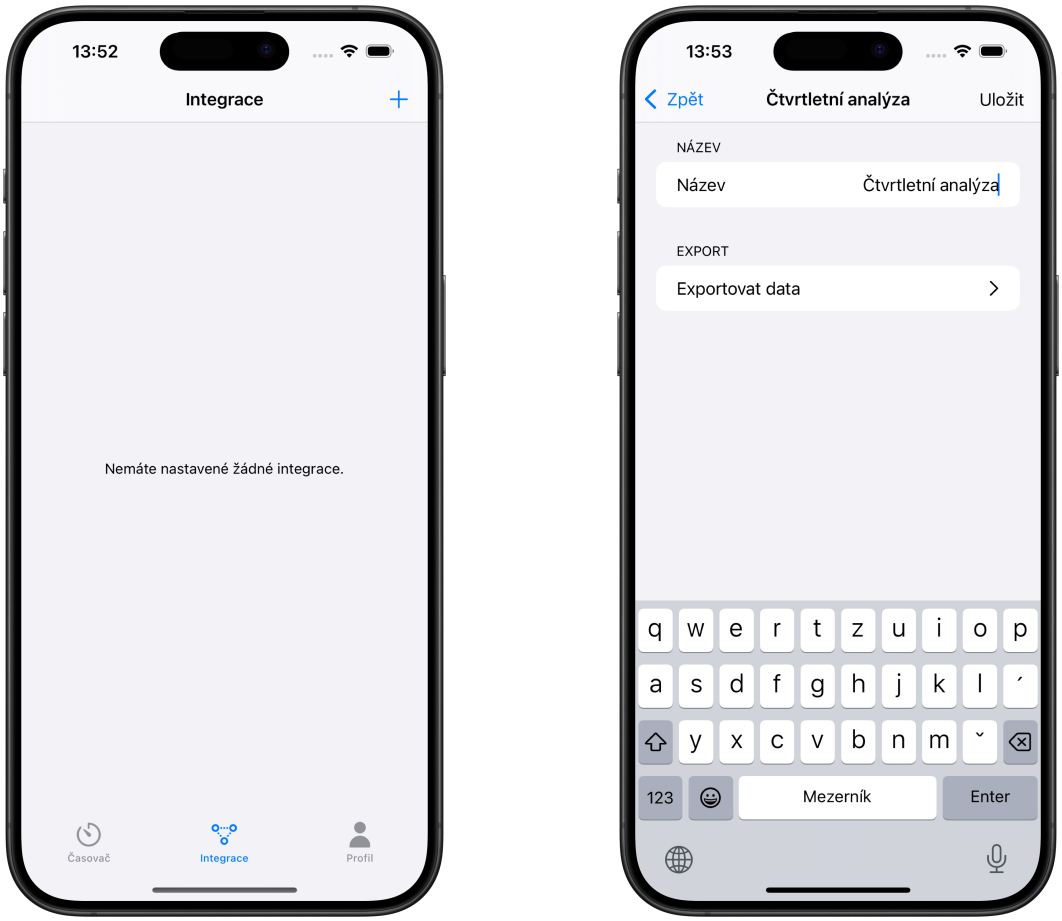
Modul `intent` poté poskytuje následující *Use Cases*:

- `CancelTimerUseCase` – Zruší běžící časovač, zahodí tedy jeho data. Pokud časovač neběží, neudělá nic.
- `StartTimerUseCase` – Spustí časovač, tedy aktualizuje jeho data tak, že bude ve stavu `active` a bude měřit od momentu, kdy byl *Use Case* zavolán. Pokud časovač už běží, neudělá nic (tedy počátek měření zůstane nezměněn).
- `StopTimerUseCase` – Zastaví časovač a vytvoří z jeho dat nový časový záznam. Pokud například chybí zadání vybraného projektu, tak *Use Case* vrátí chybu. Pokud časovač neběží, neudělá nic.

U těchto *Use Cases* je důležité, aby v případě, že se daný *Intent* snaží dostat časovač do stavu, ve kterém již je, opravdu neudělal nic. Tedy například v případě opakovaného zapnutí nebyl měněn čas zapnutí podle nového volání. Je tak potřeba proto, protože operace zkratek jsou navrženy tak, aby byly idempotentní[3]. Tyto zkratky totiž pravděpodobně budou součástí nějakých automatizací, které můžou například zapínat časovač podle polohy. Pokud by opakované spuštění zkratky přepisovalo data podlé nových informací, byla by původní data smazána a uživatel by o ně přišel. Podnětem pro budoucí vylepšení aplikace může být například to, aby se v případě, že uživatel vyvolá zkratku pro spuštění časovače s novými daty, když už časovač běží, časovač automaticky zastavil, uložil nový časový záznam z dosud běžících dat, a poté se znovu zapnul pro nová data.

4.3.4.3 Nativní aplikace

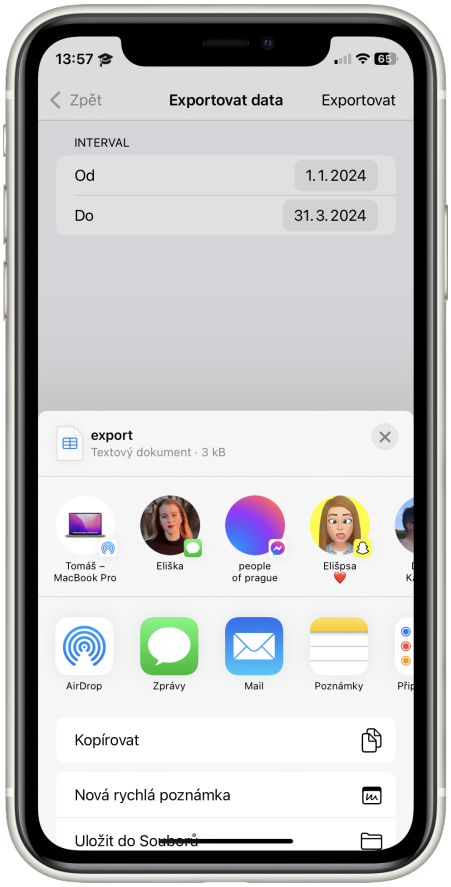
Uživatelské rozhraní iOS aplikace dodržuje návrh ze sekce 3.2.4. Na obrázku 4.9a lze vidět kartu integrací ve stavu, když uživatel nemá vytvořené žádné integrace. Na obrázku 4.9b lze poté vidět rozhraní pro tvorbu nové CSV integrace. Export CSV dat a jejich vizualizace v aplikaci *Numbers* lze nahlédnout na obrázku 4.10. Pro tvorbu zkratek bude sloužit systémová aplikace *Zkratky*, jejíž hlavní stránka lze vidět na obrázku 4.11a, rozhraní pro tvorbu nové zkratky lze pak vidět na obrázku 4.11b. Na obrázku 4.12 lze poté vidět nabídka možností zkratek pro aplikaace Trackee a možnost nastavení parametrů zkratky. A nakonec je na obrázku 4.13 vidět možnost tvorby automatizací pro operace s časovačem.



(a) Prázdný seznam

(b) Nová CSV integrace

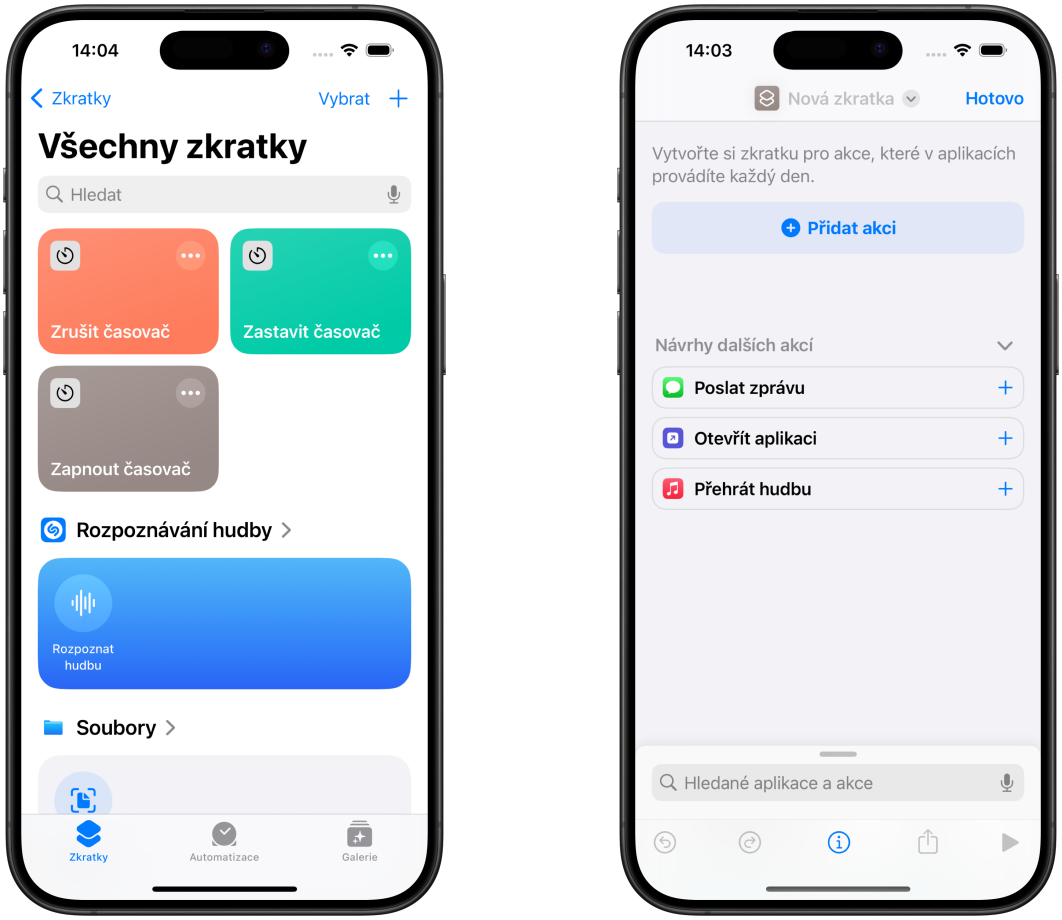
Obrázek 4.9 Realizace integrací



(a) Dialog pro exportovaný CSV soubor

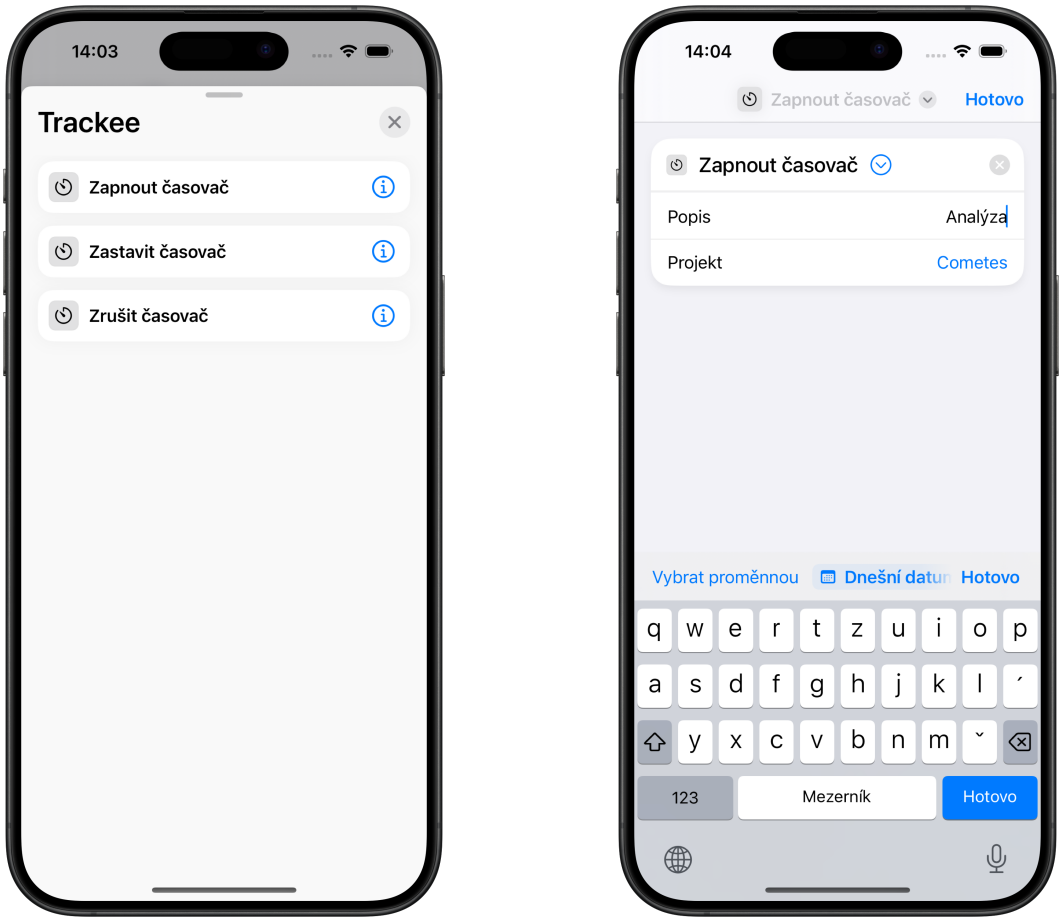
(b) Exportovaný CSV soubor v aplikaci *Numbers*

Obrázek 4.10 Realizace exportu do CSV



(a) Přehled zkratek **(b)** Tvorba nové zkratky

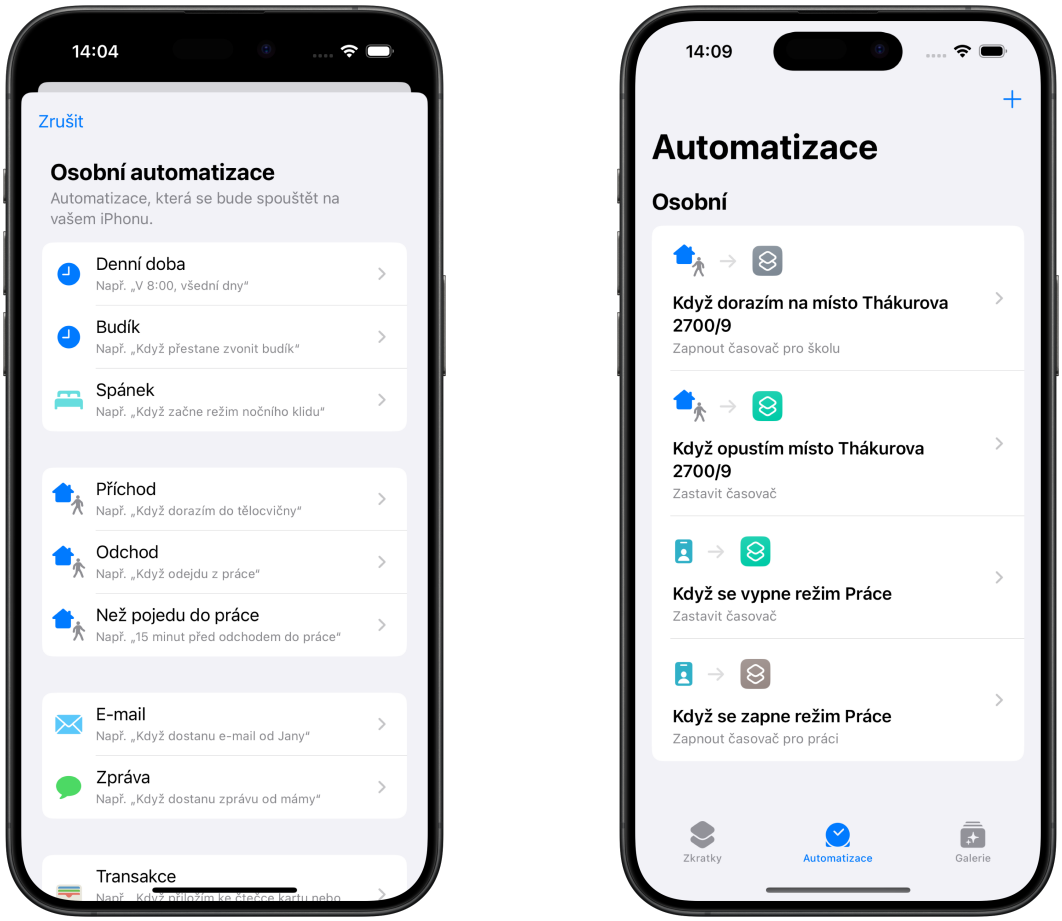
Obrázek 4.11 Systémová aplikace *Zkratky*



(a) Možnosti zkratek aplikace Trackee

(b) Parametry zkratky

Obrázek 4.12 Tvorba nové zkratky pro Trackee



(a) Tvorba nové automatizace **(b)** Seznam automatizací

Obrázek 4.13 Automatizace

V sekci 3.2.4.1 bylo také zmíněno, že pro podporu interakce s aplikací pomocí systémových zkratek je potřeba implementovat takzvané *App Intents*. Tyto *intents* představují nějaký úmysl uživatele, který může pomocí zkratky spustit. [77]

Struktury pro reprezentaci *App Intents* je potřeba definovat v hlavním *targetu* aplikace, nelze je definovat v žádném balíčku, protože by pak nebyly detekovány systémem, aby se zobrazily v aplikaci *Zkratky*. Jednotlivé *intents* musí implementovat protokol (rozhraní) `AppIntent`, který vyžaduje, aby *intent* obsahoval minimálně název, popis a funkci `perform()`, která se zavolá při pokusu o provedení *intentu*. `AppIntent` také vyžaduje, aby byly texty definovány jako `LocalizedStringResource`, nelze tedy použít lokalizaci definovanou v balíčku *UIToolkit*. Pro potřeby *App Intents* slouží ve složce iOS projektu složka `Intents`, kde jsou definovány *intenty* a jejich entity podle funkcionalit aplikace, a také *Resources*, tedy `Localizable.strings` soubor, který bude obsahovat texty a jejich lokalizace pro všechny *App Intents*.

Ve výpisu kódu 12 lze nahlédnout implementace *intentu* pro zapnutí časovače. Implementace obsahuje referenci na typ `StartTimerUseCase`, který poskytuje *Dependency Injection* stejným způsobem, jako ve *View Models*. Dále iplementace obsahuje definici názvu, popisu a parametrů, které daný *intent* může přijímat. A nakonec implementace obsahuje definici funkce `perform()`, která se spustí při spuštění *intentu*.

Aby mohl nějaký objekt být parametrem *App Intentu*, musí splňovat požadavky protokolu `AppEntity`. Tento protokol vyžaduje po objektech, které ho implementují, aby obsahovaly proměnnou `displayRepresentation` typu `DisplayRepresentation`, která definuje, jak se daný parametr zobrazí uživateli, tedy nějaký název, případně podnázev a obrázek. Také po objektech vyžaduje, aby implementovaly statickou proměnnou `defaultQuery`, což je další objekt, který musí implementovat protokol `EntityQuery`. Tento protokol slouží pro objekty, které definují, odkud se mají brát všechny možnosti pro výběr konkrétní instance parametru, k čemuž slouží funkce `suggestedEntities()` a `entities(for identifiers: [String])`.

Pro potřeby *intentu* `StartTimerIntent` je potřeba entita `ProjectPreviewEntity`, která představuje objekt projektu, který si uživatel může nastavit jako parametr *intentu*. Tento objekt v podstatě reflektuje `ProjectPreview`, který se do této reprezentace umí převést. Obsahuje tedy název, název klienta a případně typ projektu pro určení obrázku. Identifikátor tohoto objektu má strukturu `<clientID>-<projectID>`, protože je potřeba, aby měl pouze jeden identifikátor. Implementace `ProjectPreviewEntityQuery` poté získává objekty projektů přímo přes `GetProjectsUseCase` a `GetProjectPreviewUseCase`. Ostatní *intenty* žádné parametry nepotřebují.

Propagace chyb do aplikace *Zkratky* funguje automaticky, protože když nějaký *Use Case* vrátí chybu, tak vrátí strukturu `KMPError`, která sama umí poskytnout lokalizovanou chybovou hlášku, která se zobrazuje jak v aplikaci, tak v aplikaci *Zkratky*.

[3]Operace je idempotentní, pokud jejím opakovaným použitím na nějaký vstup vznikne stejný výstup, jako vznikne jediným použitím dané operace. [107]

```
struct StartTimerIntent: AppIntent {

    // MARK: - Dependencies

    @Injected(\.startTimerUseCase) private var startTimerUseCase

    // MARK: - Required

    static var title = LocalizedStringResource("start_timer_intent_title")

    static var description = IntentDescription("start_timer_intent_description")

    // MARK: - Parameters

    @Parameter(title: "start_timer_intent_parameter_description")
    var description: String?

    @Parameter(title: "start_timer_intent_parameter_project")
    var project: ProjectPreviewEntity?

    // MARK: - Perform

    func perform() async throws -> some IntentResult {
        let params = StartTimerUseCaseParams(
            body: StartTimerBody(
                clientId: project?.clientId,
                projectId: project?.projectId,
                description: description
            )
        )
        try await startTimerUseCase.execute(params: params)
        return .result()
    }
}
```

Výpis kódu 12 *App Intent* pro zapnutí časovače

Kapitola 5
Testování

Výslednou aplikaci a její uživatelské rozhraní je potřeba důkladně otestovat. Tato kapitola popisuje implementaci automatických testů a scénáře testování s reálnými uživateli. Výsledky z testování jsou poté zhodnoceny.

5.1 Automatické testování

Pro účely automatizovaného testování se u Apple platforem používá knihovna *XCTest* [108]. Jak již bylo zmíněno, *Clean architecture* ja navržená tak, aby se dala dobře otestovat jednotkovými testy, které jsou popsány v následující sekci.

Dalšími možnostmi testování jsou UI testy a testy výkonu. Aplikace Trackee implementaci těchto typů testů neobsahuje. Jejich implementace může být podnětem pro budoucí vylepšení aplikace.

5.1.1 Jednotkové testování

V aplikaci jsou jednotkovými testy pokryty všechny *View Modely*. Pro každý z nich je v modulu, ke kterému patří, soubor, který daný *View Model* testuje. Např. pro `LoginViewModel.swift` bude testovací soubor `LoginViewModelTests.swift`.

View Model testy jsou navržené tak, aby testovaly každý jednotlivý *Intent*, pokud existuje vhodná možnost, jak test provést. Některé intenty vhodně otestovat nelze – například takové, které otevírají nějaké systémové dialogy, které nemají žádný výstup pro *View Model* aplikace. Daný test tedy vždy ověřuje, zda se stav *View Modelu* po aplikování *Intentu* změnil podle očekávání.

Jednotkové testy *View Modelů* využívají *Mocky* pro *Use Cases*, tedy určité zjednodušené náhrady těchto *Use Cases*, u kterých si každý test může definovat, co bude daný *Use Case Mock* vracet za hodnotu. Toto je klasický přístup k jednotkovému testování – testujeme zde *View Modely*, nikoli *Use Cases*.

Předpokládejme tedy, že máme *View Model* se strukturou ve výpisu 13. Struktura testu poté bude vypadat jako ve výpisu 14. Struktura obsahuje následující:

- `flowController` – *Mock* pro *FlowController*, tedy náhrada reálného controlleru v aplikaci. Tento *Mock* slouží k tomu, abychom mohli otestovat, jaká hodnota byla vložena do posledního volání funkce `handleFlow(_:)` a kolikrát byla tato funkce zavolána. Toto slouží k otestování toho, zda se bude apikace snažit uživatele navigovat do předpokládané destinace.

```
final class SomeViewModel: BaseViewModel, ViewModel, ObservableObject {

    // ...

    @Injected(\.someUseCase) private var someUseCase

    // ...

    enum Intent {
        case doThis
        case doThat
    }

    // ...
}
```

Výpis kódu 13 Struktura View Modelu pro testování

- *Mocky* pro *Use Cases* – zde vytváříme instance *Mocků* pro *Use Cases*, které jsou v daném *View Modelu* používány. Je potřeba to dělat takto ručně, protože kdybychom použili registrování pomocí `Container.shared.registerUseCaseMocks()`, tak bychom nemohli v testech měnit hodnoty, které dané *Use Cases* budou vracet.
- `createViewModel()` – tato funkce se bude volat na začátku každého testu, aby získal instanci pro otestování. Pokud konstruktor *View Modelu* vyžaduje nějaké parametry, můžou být předány jako parametr této funkce, nebo zde můžou mít nějakou definovanou hodnotu. V této funkci se také registrují *Mocky* pro *Use Cases* definované výše.
- Testovací funkce pro jednotlivé intenty – funkce, které mají v názvu prefix `test`, tak se automaticky považují ze testovací funkce a spouštějí se během testu. Každý vhodně otestovatelný intent zde tedy bude mít vlastní funkci, případně více funkcí, pokud lze otestovat úspěch, neúspěch, různé formy úspěchu, a podobně. Tyto funkce dodržují strukturu given-when-then, tedy rozdělení do tří částí (předdefinované hodnoty a konstanty, interakce, ověření).

Jednotkové testy *View Modelů* testují byznysovou prezentační vrstvy. Pro otestování dalších vrstev je možné v multi-platformním modulu testovat *Use Cases*, *Repositories* a *Sources*. V backendové části je také možné implementovat jednotkové testy pro *Repositories*, *Sources*, ale i pro *Routes*. Implementace testů pro tyto vrstvy může být podnětem pro budoucí vylepšení aplikace.

5.2 Uživatelské testování

Pro účely uživatelského testování byly navrženy scénáře, podle kterých se testeři mají řídit. Tyto scénáře vycházejí z navržených případů užití v sekci 3.2 a měly by pokrývat většinu funkcionalit aplikace, kterých uživatel může využít. Povinnými účastníky během testování budou moderátor a uživatel. Moderátor bude uživateli dávat instrukce a dokumentovat, jak uživatel s mobilním telefonem interaguje. Uživatel se bude snažit instrukce plnit a bude při tom nahlas popisovat své akce s telefonem (co vidí, na co se chystá kliknout, co očekává, že se stane, atd.), aby usnadnil následnou analýzu celé situace.

Uživatelského testování se zúčastnilo 5 testerů. Záznamy z testování lze nahlédnout v [109]. Přehled testerů lze nahlédnout v tabulce 5.1. Instrukce, podle kterých se testeři během testu řídili, lze nahlédnout v příloze A. Protokol z testování lze nahlédnout v příloze B.

```
@MainActor
final class SomeViewModelTests: XCTestCase {

    private let flowController = FlowControllerMock<SomeFlow>(
        navigationController: UINavigationController()
    )

    private let someUseCaseMock = SomeUseCaseMock(
        executeReturnValue: /* Some default return value */
    )

    private func createViewModel(
        someOptionalParam: SomeType
    ) -> SomeViewModel {
        Container.shared.someUseCase.register { self.someUseCaseMock }

        return SomeViewModel(
            someParam: someOptionalParam,
            flowController: flowController
        )
    }

    // MARK: - Tests

    func testDoThis() async {
        // given
        let vm = createViewModel()
        // ... some constants and given values
        let someConstant = ...

        // when
        vm.onIntent(.doThis)
        // ... some interactions
        await vm.awaitAllTasks()

        // then
        // ... assertions
        XCTAssertEqual(vm.state.someValue, someConstant)
    }

    func testDoThat() async {
        // ...
    }
}
```

Výpis kódu 14 Struktura View Model testu

Tester	Pohlaví	Používaný OS	Používá aplikace pro měření času
F. W.	Muž	Android	ANO (Clockify)
D. K.	Muž	iOS	ANO (Clockify)
D. Ž.	Muž	Android	ANO (Clockify)
E. Č.	Žena	iOS	NE
T. S.	Muž	Android	ANO (Clockify, Toggl Track)

Tabulka 5.1 Přehled testerů aplikace

Uživatelské testování probíhalo na produkčním prostředí aplikace Trackee, tedy na prostředí, které by mělo cílit na reálné uživatele a které poskytuje omezené možnosti ladění, jako jsou například omezené popisy neznámých chyb. Aplikace, kterou testeři testovali, také komunikovala s instancí backendu na lokální síti, a nikoli s instancí nasazenou v aplikaci *Railway* (více informací v sekci 4.1.2). Lokální instance byla zvolena proto, aby byli testeři během testování odstíněni od prodlev, které jsou způsobeny velkou vzdáleností od instance v aplikaci *Railawy*.

5.2.1 Scénář

Tvorba uživatelského účtu: Každý nový uživatel mobilní aplikace Trackee si bude muset nejprve vytvořit svůj uživatelský účet. V tomto scénáři bude uživatel instruován k tomu, aby aplikaci poprvé spustil, vytvořil si nový uživatelský účet pomocí předepsaného e-mailu a hesla a následně se do aplikace pod tímto účtem přihlásil.

Vytvoření nového klienta: Po úspěšné registraci bude uživatel instruován, aby si vytvořil 2 nové klienty s předepsanými názvy.

Vytvoření nového projektu: Po úspěšném vytvoření klientů bude uživatel instruován, aby vytvořil 2 nové projekty s předepsanými vlastnostmi.

Spuštění časovače: Po úspěšném vytvoření projektů dostane uživatel instrukci, aby spustil časovač pro měření odpracovaného času a přiřadil mu předepsaný projekt a popis.

Změna začátku časovače: Uživatel bude instruován k tomu, aby u spuštěného časovače změnil začátek na předepsaný čas.

Zastavení časovače: Uživatel bude instruován, aby ukončil časovač a uložil časový záznam, který do teď měřil.

Manuální přidání časového záznamu: Uživatel bude instruován, aby ručně přidal časový záznam s předepsanými parametry a časem.

Úprava projektu: Uživatel bude instruován, aby aktualizoval předepsaný projekt s novými předepsanými vlastnostmi.

Odhlášení a přihlášení na testovací účet: Uživatel ve svém nově vytvořeném účtě nebude mít dostatek klientů, projektů a časových záznamů, které by odpovídaly dlouhodobějšímu používání aplikace. Pro lepší otestování orientace v aplikaci bude uživatel instruován, aby se přihlásil na předepsaný testovací účet, který obsahuje větší množství záznamů.

Odstranění časového záznamu: Uživatel bude instruován, aby odstranil předepsaný časový záznam z historie.

Export historie do CSV souboru: Uživatel bude instruován, aby vytvořil integraci pro exportování do CSV souboru a exportoval data do tabulky pro předepsané časové období. Dále bude instruován, aby tuto tabulku otevřel ve vhodné aplikaci, kde si ji může prohlédnout (např. *Numbers*).

Odstranění klienta: Budeme předpokládat, že si uživatel v exportované tabulce všimne, že tam má záznamy patřící předepsanému klientovi, které tam mít nechce. Bude tedy instruován, aby klienta smazal a znovu vyexportoval CSV soubor, ve kterém zkontroluje, že žádné záznamy patřící k tomuto klientovi nejsou.

Tvorba automatizace pro spuštění časovače: Uživatel dostane instrukci, aby pomocí aplikace *Zkratky* vytvořil novou automatizaci, která zapne časovač s předepsanými parametry, pokud se uživatel objeví na předepsané poloze. Bude také instruován, aby tuto automatizaci zkusil ručně spustit a v aplikaci zkontroloval, že časovač opravu běží.

Tvorba automatizace pro vypnutí časovače (odstraněno): Uživatel bude požádán, aby vytvořil automatizaci, která se ho zeptá, zda nechce zastavit běžící časovač, pokud opustí předepsanou polohu. Zde bude také instruován, aby automatizaci zkusil ručně spustit. Tento scénář byl po několika testech ze seznamu odstraněn, protože bylo zhodnoceno, že je příliš podobný předchozímu scénáři a zároveň přímo netestuje rozhraní samotné aplikace.

5.2.2 Výsledky

V protokolu z průběhu testování (příloha B) byly označeny takzvané *kritické* a *důležité poznatky*. Kritické poznatky jsou takové, u kterých bylo zhodnoceno, že se jedná o závažnou chybu rozhraní aplikace, která může uživateli závažným způsobem zhoršit zkušenost s jejím používáním. Důležité poznatky jsou potom takové, které nemusí být nutně nějakou chybou v rozhraní, ale potenciálním podnětem pro zlepšení rozhraní. Tato sekce rozebírá a analyzuje tyto dva druhy poznatků. Následující sekce postupně rozebírají kritické a důležité poznatky, seřazené podle toho, jaká jim byla přidělena priorita (první má největší prioritu).

5.2.2.1 Kritické poznatky

Kritické poznatky byly v průběhu testování zaznamenány hned u prvního testera (tester F. W.). Vzhledem k jejich závažnosti byly chyby objevené těmito poznatky opraveny okamžitě. Z tohoto důvodu se už u dalších testerů tyto stejné chyby objevit nemohly.

- **Opakované ukazování neznámé chyby během registrace.** Testerovi F. W. se po zadání údajů pro registraci nedařilo registraci dokončit, protože po klikání na tlačítko *Registrovat* se neustále ukazovala *Neznámá chyba*. Tato chybová hláška neposkytovala žádný popis toho, co konkrétně by mohlo být špatně, ani návrh na to, jak chybu opravit. Tester musel opakovaně zkoušet vyplňovat e-mail v jiných formátech. Nakonec se mu přihlášení podařilo a hádal, že chyby asi byly kvůli špatnému formátu e-mailu nebo slabému heslu.

5.2.2.2 Důležité poznatky

- **Obtížná klikatelnost prvků v navigační liště.** Většina testerů musela opakovaně klikat na tlačítka *Uložit* nebo *Exportovat* v navigační liště, protože se jim do nich nedařilo trefit.
- **Redundantní potvrzení po výběru klienta nebo projektu.** Někteří testeři zmínili, že se jim zdá redundantní klikat na tlačítko *Uložit* po výběru projektu pro ovladač časovače, nebo po výběru klienta pro projekt. Očekávali, že po zvolení projektu nebo klienta se volba uloží automaticky a aplikace se vrátí o obrazovku zpět.

- **Absence možnosti úpravy časového záznamu.** Několik testerů se při různých fázích scénáře snažilo klikat nebo podržet na časový záznam v historii, pravděpodobně s očekáváním, že to bude mít nějakou odezvu, jako otevření detailu. Aplikace ale nic takového neimplementuje, pouze swipe-to-delete.

- **Počáteční zmatení nad přepínáním ovladače mezi časovač a manuální zadávání.** Někteří testeři byli zpočátku zmateni nad rozhraním ovladače pro přepínání mezi klasickým měřením pomocí časovače a mezi manuálním zadáváním času nového záznamu. Jeden tester zmínil, že by pro manuální zadání času preferoval nějaký dialog místo in-place změny rozhraní časovače, ale poté zmínil, že přepínání nakonec pochopil a že jde možná o zvyk.

- **Absence swipe-to-delete u některých seznamů – nekonzistence v rámci aplikace.** Jeden tester zmínil, že byl zmatený z toho, že při pokusu o smazání klienta nefungovalo gesto swipe-to-delete, které fungovalo při mazání časových záznamů, a používání tohoto gesta tak bylo v rámci aplikace nekonzistentní.

- **Klávesnice při zadávání e-mailové adresy nenabízí v základním rozmístění kláves tečku.** Při zadávání e-mailové adresy se zobrazí klávesnice, která ve svém výchozím rozmístění neobsahuje klávesu pro tečku, ale ani například klávesu pro zavináč. Uživatel tak musí dodatečně klikat na klávesu pro zobrazení speciálních znaků.

- **Malá klikatelná plocha tlačítka pro smazání.** Jeden tester si všiml, že pro kliknutí na tlačítko pro mazání je potřeba kliknout na text tlačítka, nefunguje kliknutí pouze na pozadí tlačítka.

- **Neviditelný kurzor při zobrazení zadaného hesla.** Při zadávání hesla aplikace nabízí možnost zobrazit přímo text hesla, které uživatel zadal, místo bezpečnostního zakrytí tečkami. Při tomto zobrazení ale z pole zmizí kurzor, přestože do pole lze normálně psát i nadále.

- **Jednotlivé karty aplikace si pamatují zanoření.** Někteří testeři byli zmatení nad tím, že při přepnutí do karty profilu byli zanořeni v obrazovce, do které se proklikli při jednom z předchozích scénářů. Očekávali, že kliknutí na karty profilu je dostane do přehledu profilu.

- **Absence onboardingu nebo možnosti vytvoření počátečních klientů a projektů.** Pokud účet uživatele neobsahuje žádná data, zpravidla po registraci, tak sice zobrazuje informaci, že uživatel například nemá žádné projekty a že si je může vytvořit v profilu, ale nenabízí nějaký jednoduchý proklik, který by toto počáteční zadávání dat usnadnil. Někteří testeři zmínili, že by aplikace mohla obsahovat nějakou formu onboardingu, což je obvykle nějaká úvodní nápověda, v rámci které si uživatel může například právě vytvořit nějaká počáteční data.

- **Nedostatečná vizuální indikace běžícího časovače.** Jeden tester zmínil, že by při běžícím časovači přidal tlačítku pro zastavení červené pozadí, jelikož se jedná o zvyk u existujících řešení pro měření a správu času.

- **Ne příliš výstižný popis tlačítka Uložit.** Jeden tester zmínil, že při vytváření nového klienta nebo projektu by upřednostnil textaci tlačítka jako *Vytvořit* místo *Uložit*.

- **Ne příliš výstižný popis tlačítka Smazat.** Jeden tester zmínil, že v potvrzovacím dialogu pro smazání klienta (tedy i projektu) by očekával text Smazat místo Ano.

- **Přednost popisu záznamu před názvem klienta nebo projektu.** Dva testeři zmínili, že sem jim zdá, že popis časového záznamu je důležitější, než název projektu nebo klienta.

- **Absence potvrzení před smazáním časového záznamu.** Dva testeři zmínili, že je překvapilo, že pro mazání časového záznamu nebyl žádný potvrzovací dialog.

- **Údiv nad zařazením exportu do CSV mezi integrace.** Dva testeři zmínili, že se jim zdá zvláštní, že je export do CSV souboru zařazen mezi integrace.
- **Aplikace se po exportu dat nevrací o obrazovku zpět.** Jeden tester zmínil, že by po úspěšném exportu očekával, že se aplikace vrátí o obrazovku zpět.
- **Absence indikace spuštěné zkratky v aplikaci.** Dva testeři zmínili, že by po spuštění zkratky očekávali, že se v aplikaci objeví nějaká indikace, že byla zkratka provedena.
- **Při snaze o úpravu zadaného hesla se maže celé pole.** Někteří testeři byli podrážděni tím, že když potřebovali upravit zadané heslo, tak se při pokusu o smazání jednoho znaku smazalo celé pole.
- **Zapnutí časovače v aplikaci Zkratky nemá automaticky otevřený dialog pro parametry.** Jeden tester zmínil, že při tvorbě zkratky by čekal, že nabídka parametrů bude otevřená automaticky.

5.2.3 Zhodnocení

Tato sekce rozebírá poznatky z uživatelského testování, diskutuje nad tím, čím byly způsobeny a jakým způsobem by případně mělo na jejich základě být upraveno rozhraní aplikace.

Co se týče kritických poznatků, který byl jen jeden, tak jak již bylo zmíněno, byly opraveny ihned po provedení uživatelského testování s prvním testerem. Jednalo se o chybu rozhraní, které nezobrazovalo dostatečný popis chyby během registrace. Pokud uživatel zadal nevalidní formát e-mailové adresy nebo příliš slabé heslo, tak se tato chyba uživateli zobrazila pouze jako *Neznámá chyba*, z čehož by uživatel měl jen velmi nízkou šanci pochopit, v čem udělal chybu. Tato chyba byla způsobena tím, že aplikace Trackee v produkčním prostředí neposkytuje technické popisy chyb pro neznámé druhy chyb, tedy takové, které nejsou explicitně detekovány a není jim přidělena vlastní lokalizovaná textace. To přesně se dělo u těchto chyb, kdy z knihovny *Firebase* autentizace přišla nějaká chyba, která sice obsahovala nějaký popis chyby z knihovny, ale ten aplikace v produkčním prostředí neukazovala a z chyby se tak stala *Neznámá chyba*. Byla tedy přidána explicitní detekce chyb pro špatný formát e-mailu, pro příliš slabé heslo a pro pokus o registraci s již existujícím e-mailem.

Důležité poznatky byly zhodnoceny následovně:

- **Obtížná klikatelnost prvků v navigační liště:** U této chyby se lze do jisté míry odvolat na systémové rozhraní iOS, protože způsob, kterým jsou v aplikaci přidávány tlačítka do navigační lišty, přesně dodržuje způsob navržený Applem a jedná se o způsob, který sám Apple ve svých aplikacích používá. Do navigační lišty je zkrátka nutné kliknout přesně, jinak aplikace dotek zaregistruje mimo tlačítko. Ale vzhledem k tomu, že se snad žádnému testerovi nepodařilo kliknout na všechny navigační tlačítka na první pokus, tak se jedná o tak závažný problém, že by se mu budoucí vývoj měl určitě věnovat i tak. Mohlo by být vyvinuto nějaké úsilí na pokus rozšíření klikatelné plochy v navigační liště, a pokud by tento postup byl při použití systémové navigační lišty obtížný, mohly by být potvrzovací tlačítka z navigační lišty zcela odstraněny a nahrazeny primárním tlačítkem přímo vespod obsahu dané obrazovky.
- **Redundantní potvrzení po výběru klienta nebo projektu:** Někteří testeři přímo zmínili, že jim toto přijde zbytečné. A vzhledem k tomu, že se nejedná o žádnou nevratnou akci, kterou by uživatel nemohl například při špatné volbě ihned změnit, tak není důvod odmítat změnu rozhraní, ve kterém se při volbě klienta nebo projektu aplikace opravdu automaticky vrátí o obrazovku zpět a volbu sama uloží. Ve všech případech, kde se tento přístup používá, se lze na výběr ihned vrátit.

- **Absence možnosti úpravy časového záznamu:** Během různých úmyslů se testeři snažili na záznam kliknout, ale nic se nestalo. Možnost vytvoření nějakého rozhraní pro úpravu záznamu je tedy určitě podstatným podnětem pro další vývoj aplikace.

- **Počáteční zmatení nad přepínáním ovladače mezi časovač a manuální zadávání:** Rozhraní ovladače, konkrétně přepínaní mezi stavem časovače a stavem manuálního zadávání, bylo inspirováno ovladačem časovače v aplikaci *Clockify* [1]. Uživatelům umožňuje ovládat časovač různými způsoby na jednom místě. Uživatelé zmínili, že jakmile způsob přepínání pochopili, tak s rozhraním problém neměli. Je tedy na pováženou, zda se smířit s tím, že ovladač časovače vyžaduje nějaké pochopení, které není zcela intuitivní, ale jakmile k pochopení dojde, umožní snadnější tvorbu záznamů. Protože například použití dialogu pro manuální tvorbu záznamu, jak navrhl jeden tester, by mohlo způsobit, že takový dialog překryje část viditelného rozhraní historie a tím uživateli znemožní se inspirovat předchozími záznamy, ověřit si, že na sebe časy navazují, a podobně.

- **Absence swipe-to-delete u některých seznamů – nekonzistence v rámci aplikace:** Poznámka testera, že to způsobuje nekonzistenci v rámci aplikace, byla zcela namístě. Podnětem pro další vývoj aplikace by mělo být sjednocení intearkčních prvků ve všech seznamech, aby všude byl implementován detail i swipe-to-delete, a všechny se tak chovaly stejným způsobem.

- **Klávesnice při zadávání e-mailové adresy nenabízí v základním rozmístění kláves tečku:** Systém iOS umožňuje různým textovým polím přidělovat různé typy klávesnic, u pole pro e-mailovou adresu by tohoto tedy mělo být využito, aby se zobrazila klávesnice se specializovaným rozmístěním kláves pro zadávání e-mailu.

- **Malá klikatelná plocha tlačítka pro smazání:** U obrazovek, kde vznikl tento problém, se používá nativní UI komponenta *List*, která pracuje s tlačítky v řádcích takovým způsobem, že detekuje kliknutí na ně pouze tehdy, pokud se klikne na text tlačítka, alespoň v základním použití. Podnětem dalšího vývoje by proto mělo být zkoumání, jak tento problém vyřešit. Buď snahou o rozšíření klikatelné plochy, nebo nahrazení vlastním primárním tlačítkem mimo nativní seznam.

- **Neviditelný kurzor při zobrazení zadaného hesla:** Aplikace používá vlastní řešení pro přepínání mezi viditelným a neviditelným stavem pole pro hesla, které odstraňuje chybu nativního řešení, které při přepnutí viditelnosti zruší *focus* na pole, což způsobí zmizení klávesnice a uživatel na pole musí kliknout znovu. Vedlejším produktem tohoto řešení je ale zřejmě tento problém, je tedy potřeba použité řešení zrevidovat a pokusit se tento problém odstranit.

- **Jednotlivé karty aplikace si pamatují zanoření:** U tohoto poznatku bylo překvapující, že vznikl i u uživatelů systému iOS, ve kterém je toto chování při používání několika navigačních karet ve spodní liště zcela standardní. Je opět na pováženou, jestli se snažit toto chování měnit a určovat tento poznatek za určující, když se jedná o standardní chování.

- **Absence onboardingu nebo možnosti vytvoření počátečních klientů a projektů:** Prázdné stavy jednotlivých obrazovek by mohly být mimo samotný informující text rozšířeny o tlačítko pro přidání prvku, které je jinak pouze v navigační liště. Implementace úvodní nápovědy poté může být podnětem pro další vývoj aplikace.

- **Nedostatečná vizuální indikace běžícího časovače:** Tester, který tento poznatek vznesl, zmínil, že je z ostatních řešení zvyklý, že při běžícím časovači má ovládací tlačítko červené pozadí, aby dostatečně vizuálně odlišilo stav běžícího časovače. Jestliže se jedná o běžnou praxi u měřičů odpracovaného času, není důvod se této praxi vyhýbat a červené pozadí nepřidat také.

- **Ne příliš výstižný popis tlačítka Uložit:** Navigační tlačítko *Uložit* může být v případě vytváření nového klienta nebo projektu změněno na *Vytvořit*. Tento text je výstižnější a není důvod ho nepoužít.

- **Ne příliš výstižný popis tlačítka Smazat:** Tlačítko *Ano* může být nahrazeno textací *Smazat*, která bude pravděpodobně více intuitivní.

- **Přednost popisu záznamu před názvem klienta nebo projektu:** První tester, který tento podnět vznesl, nazval popis jako *title* – jeho význam tedy asi pochopil jako název či nadpis záznamu. Popis ale v záznamech poskytuje až sekundární roli, je nepovinný. Projekt je naopak u záznamu povinný. Tento názor ale později vyjádřil i další tester. Mělo by být tedy zváženo, zda v rozhraní neupřednostnit popis záznamu před projektem, přestože vůbec nemusí být přítomen.

- **Absence potvrzení před smazáním časového záznamu:** Testeři, kteří tento podnět vznesli, měli pravdu v tom, že se jedná o destruktivní akci a pokud ji uživatel udělá omylem, přijde o data a nemá možnost je získat zpět. Přidání explicitního dialogu se ale zdá být zase příliš, protože pokud by uživatel chtěl smazat více záznamů, musel by u každého záznamu potvrzovat dialog, a zas tolik dat se pod jedním záznamem neskrývá. Ideálním řešením by tedy mohla být přidaná možnost vrátit akci mazání. Po smazání záznamu, které by bylo učiněno stejným způsobem jako do teď, by se mohl zobrazit *Toast* s informací, že záznam byl smazán, a s tlačítkem *Vrátit*, které by akci vrátilo a smazaný záznam vrátilo zpět.

- **Údiv nad zařazením exportu do CSV mezi integrace:** Dva testeři vyjádřili údiv nad tím, že export do CSV je zařazen mezi integrace. Je pravda, že samotný soubor CSV sám o sobě integraci netvoří. CSV soubor je ale integračním prvkem, který umožňuje integraci v podstatě s čímkoli – s řadou dalších systémů, s nějakými vizualizačními nástroji, a podobně. A vzhledem k tomu, že ho uživatel právě za účelem integrace bude pravděpodobně využívat, tak má určitě své místo na kartě integrací. Ostatně, oddělit ho od této karty a hledat mu jiné místo by pravděpodobně nedávalo smysl.

- **Aplikace se po exportu dat nevrací o obrazovku zpět:** Tento podnět vznesl jeden tester. Bylo zhodnoceno, že toto chování by bylo očekávatelné v případě, kdyby místo obrazovky sloužil pro export dat nějaký typ dialogu. Ale protože se jedná o standardní obrazovku, jejíž přístup uživatelům umožní například exportovat více časových období za sebou, nebo opětovné exportování se stejnými parametry (které bylo dokonce předmětem jednoho scénáře), bylo rozhodnuto, že automatické vrácení zpět, které by způsobilo ztrátu dat na této obrazovce, by být implementováno nemělo.

- **Absence indikace spuštěné zkratky v aplikaci:** Tento podnět zmínili dva testeři, kteří při návratu do aplikace po spuštění zkratky neviděl žádnou explicitní indikaci toho, že byla zkratka provedena. Zkratky implementované v aplikaci ale nemají záměr uživatele po jejich provedení přesměrovávat do aplikace, k tomu slouží jiné, *otevírací* typy zkratek. Zkratky aplikace Trackee slouží primárně pro automatizaci, uživatel nebo automatizace tedy zkratku spustí, aplikace vykoná úkony a uživatel vůbec nemá potřebu aplikaci otevírat. Tento podnět tedy byl pravděpodobně vznesen kvůli tomu, že instrukce scénáře testerovi přímo říkala, aby po spuštění zkratky zkontroloval, že se akce v aplikaci provedla, ale toto uživatelé ve skutečnosti vůbec dělat nemusí. Jedná se tedy o lehce zavádějící instrukci testovacího scénáře, která testery instruuje k tomu, aby udělali něco, co není klasickým případem užití. Úmyslem tohoto scénáře bylo spíše to, aby testeři pochopili, co vlastně zkratka udělala.

- **Při snaze o úpravu zadaného hesla se maže celé pole:** Několik testerů toto chování podrážděno, ale jedná se o standardní chování textových polí pro citlivé údaje. Pokud uživatel přestane s úpravou takového pole, později s úpravou zase začne a pokusí se smazat jeden

znak, smaže se celý text pole. Jedná se o bezpečností prvek, není tedy vhodné se toto chování snažit měnit.

- **Zapnutí časovače v aplikaci Zkratky nemá automaticky otevřený dialog pro parametry:** V dokumentaci pro *App Intents* [77] nebyla nalezena zmínka o tom, že by existovala nějaká možnost, jak aplikaci *Zkratky* donutit, aby ve výchozím stavu otevřela dialog pro parametry. Toto chování tedy aplikace nemůže ovlivnit.

Kapitola 6

Prostředí pro budoucí podporu a provoz aplikace

Během tvorby této práce vzniklo mnoho podnětů pro budoucí vývoj a vylepšení aplikace. Už v samotném návrhu nebo realizaci byla rozebírána řada možností, které v této práci realizovány nejsou, ale bylo by je vhodné v budoucnu implementovat. Během uživatelského testování pak vznikla řada doporučení pro úpravu rozhraní aplikace.

6.1 Zrychlení komunikace a lokální data

Prvním tématem, kterému by se budoucí provoz aplikace měl zabývat, je celkové zrychlení manipulace s aplikací. Jsou dvě zásadní věci, které zpomalují interakci s aplikací a které vyžadují vylepšení.

6.1.1 Komunikace s databází

Pomalá komunikace s databází je způsobena hlavně tím, že zvolená řešení pro nasazení backendu a databáze provozují jejich instance navzájem vzdálené 8 000 km od sebe. Důvody, proč tomu tak je, byly popsány v sekci 4.1. Nejlepším řešením tohoto problému by bylo používat takové nástroje, které minimalizují vzdálenost mezi backendem a databází, která ale budou pravděpodobně vyžadovat placené plány. Mezi backendem a databází probíhá největší počet požadavků na čtení či zápis, jejich vzájemná komunikace je proto ještě důležitější, než komunikace mezi backendem a samotným klientem.

6.1.2 Lokální data (cache)

Moderní aplikace obvykle využívají nějakou formu lokální *cache*, která může mít mnoho využití, jako například dočasné zastoupení vzdálených dat z databáze. Vhodné navržení a implementace takové *cache* je ale poměrně komplexní záležitost a proto nemohla být realizována v rámci této práce. Lokální *cache* by šlo využít mimo jiné takto:

- *UseCases*, které vrací různé seznamy dat (záznamy, klienty, projekty, ...), by místo prostého vrácení jedné hodnoty mohly vracet nějaký *stream* dat (například *KotlinX Flow* [110]), které by nejprve vrátily data z *cache* lokální databáze, a hned jak by přišla data ze vzdálené databáze, tak by se data nahradila těmito daty a také by byla aktualizována *cache*. Tímto způsobem by v podstatě zmizely prodlevy, ve kterých musí uživatel čekat, než se mu zobrazí

data a může interagovat s aplikací. Je však potřeba v jednotlivých případech počítat s tím, kdy aplikace nějaká data aktualizuje – tehdy je potřeba na vzdálená data počkat.

- Funkcionality, které nějakým způsobem aktualizují data (ovládání časovače, uložení záznamu, tvorba/úprava klienta nebo projektu, ...), by nemusely po aplikování změn čekat na to, až aktualizovaná data přijdou, ale mohly by rovnou zobrazit data v takovém formátu, v jakém by aplikace předpokládala, že aktualizovaná data budou vypadat (například při vytvoření klienta se klient rovnou zobrazí v seznamu klientů a až po chvíli jeho přidání potvrdí vzdálená data). Bylo by však třeba v případě, že aktualizace dat selže, vrátit data do původního stavu, před očekávanou aktualizací.

Implementace aplikace je pro využití lokální *cache* dobře připravena. V multi-platformní části již obsahuje nástroje pro manipulaci s lokální databází pomocí knihovny *SQLDelight* [111]. Lze také v implementaci nahlédnout, že v multi-platformní části se jednotlivé *sources* jmenují například `RemoteIntegrationSource`, což představuje zdroj vzdálených dat (z backendu) pro integraci. Při použití lokální databáze by se definoval `LocalIntegrationSource`, který by implementoval obdobné funkcionality, a logika toho, kdy a jak se má který typ *source* použít, by byla obsažena v konkrétní *repository*, v uvedeném příkladě `IntegrationRepository`.

6.2 Úpravy uživatelského rozhraní

Z průběhu uživatelského testování vzešla řada poznatků, ať už velmi důležitých nebo méně důležitých, které poukázaly na nějaké nedostatky či možnosti vylepšení v uživatelském rozhraní aplikace. Důkladný popis, analýza a doporučení pro další vývoj aplikace, vycházejících z těchto poznatků, je v sekci 5.2.2, která by měla sloužit jako podnět budoucího vývoje.

6.3 Rozšíření funkcionalit

Funkcionality, které aplikace implementuje, mají také široký potenciál, jak rozšířit jejich možnosti a přinést tak uživatelům další funkce při používání aplikace. Mezi tyto možnosti může patřit:

- **Možnost úpravy časového záznamu.** Tato funkce by uživatelům ulehčila práci se záznamy, pokud například špatně zadají nějaké parametry záznamu. Také se jednalo o jeden z poznatků uživatelského testování – zájem ze strany uživatelů o tuto funkci tedy určitě je.
- **Více parametrů pro klienty.** U klientů lze v realizované aplikaci zadávat pouze název, pro různé budoucí účely se ale mohou hodit další parametry, jako třeba fakturační adresa klienta, poznámky, nebo barevné rozlišení.
- **Více parametrů pro projekty.** U projektů se také v budoucnu mohou hodit další parametry, mimo ty, které aplikace v tuto chvíli nabízí. Mezi ty může například patřit cena práce za hodinu u konkrétního projektu, kterou lze využít během fakturace, nebo možnost vytvořit si vlastní typy projektů.
- **Reporty a shrnutí.** Mnoho existujících řešení pro měření a správu odpracovaného času umožňuje vytvářet nějaké formy reportů nebo vizualizací odpracovaného času. Uživatelé si mohou prohlížet, kolik práce vykonali v určitých obdobích u různých klientů a projektů, a podobně. Aplikace tuto funkci v současné chvíli nemá a tvorbu takových shrnutí umožňuje pouze pomocí jiných nástrojů, například z CSV dat, která aplikace poskytne.
- **Integrace s dalšími existujícími systémy.** V analýze v sekci 2.3 byly popsány různé existující systémy pro měření a správu odpracovaného času, spolu s možnostmi integrace s nimi. Realizace aplikace implementuje základ pro tyto integrace (CSV, *Clockify*), ale potenciál těchto integrací je mnohem větší. Předmětem dalšího vývoje by tedy mohla být nějaká

analýza toho, jaké další integrace by bylo vhodné implementovat a tyto integrace poté realizovat.

- **Integrace s hardwarovými spouštěči.** V analýze byly také popsány možnosti hardwarových spouštěčů (fyzické ovladače apod.), které by šly využít při integraci s aplikací. Pro propojení s jedním z nich by bylo potřeba implementovat BLE komunikaci. Zvážení této funkce také může být předmětem dalšího vývoje.
- **Přihlášení přes externí poskytovatele.** *Firebase* autentizace, které aplikace využívá, nabízí možnosti přihlášení přes *Apple*, *Facebook* a *Google*. V aplikaci je dokonce připraven *provider* pro přihlášení přes *Apple* – `AppleSignInProvider`.
- **Více parametrů pro integrace.** Integrace by mohly umožňovat větší míru flexibility, například tím, že by umožnily, aby jednotlivé integrace exportovaly pouze vybrané projekty. Toto by umožnilo efektivní propojení s více systémy.

6.4 Nasazení mezi reálné uživatele

V realizaci v sekci 4.1.3 bylo popsáno, jakým způsobem byla aplikace nasazována pro účely testování během vývoje. Podobným způsobem se poté realizuje i nasazení mezi reálné uživatele v obchodě *App Store*. Pro nahrání aplikace do tohoto obchodu také slouží nástroj *App Store Connect*.

Jak již bylo také zmíněno, aplikace musí před schválením pro nasazení do obchodu *App Store* projít procesem *App Review*, ve kterém *Apple* kontroluje splnění všech pravidel pro aplikace, které do obchodu míří. Aplikace už ale tímto procesem úspěšně prošla už v rámci veřejného testování. *App Review* pro *App Store* je sice o něco pečlivější než pro veřejné testování, ale zásadní porušení pravidel pro šíření aplikací v *App Store* by mělo být odhaleno již v tomto procesu.

Ostatní nástroje, které jsou aplikací využívány, pak žádnou další konfiguraci pro nasazení mezi reálné uživatele nepotřebují. Backend i databáze jsou nasazeny pod veřejně přístupnými doménami a aplikace je na ně napojena. Bude pouze potřeba monitorovat využití backendu nebo databáze, které jsou sice prozatím pod neplacenými plány, ale v budoucnu budou pravděpodobně zapotřebí placené plány. Také bude podle využití potřeba zhodnotit, zda se stále vyplatí využívat funkce pro uspávání backendu, kterou *Railway* nabízí.

6.5 Podpora pro další platformy

Návrh architektury platformy (sekce 3.3.1) počítal s možnostmi budoucího rozšíření podpory pro další platformy, jako je Android aplikace, webová aplikace a další. Toto rozšíření může být implementováno s pomocí technologie *Kotlin Multiplatform*, kterou klientská aplikace používá. Stačí přidat moduly pro nové platformy a jejich aplikace implementovat podobným stylem, jako nativní iOS aplikaci. Detailnější popis struktury celého projektu v rámci *monorepa* je v realizaci v sekci 4.2.

6.6 Analytika

U různých softwarových aplikací může být užitečné nějakým způsobem zaznamenávat a analyzovat chování uživatelů v aplikaci. Lze zaznamenávat, na co jak často klikají, jak používají různé interakční prvky, a podobně. Tato data poté mohou být vhodným způsobem analyzována pro zhodnocení účinností jednotlivých prvků rozhraní, testování, co funguje lépe, a další. Je však potřeba si dávat pozor na to, aby takové sbírání dat podléhalo všem zákonům a regulacím o sbírání uživatelských dat.

Jelikož je v aplikaci již používána technologie *Firebase*, lze pro tyto účely použít nástroj *Firebase Analytics* [112], který slouží pro zaznamenávání a analýzu různých akcí (klikání na některé prvky, interakce, ...), a nástroj *Firebase Crashlytics* [113], který slouží pro analýzu chyb a pádů aplikace.

Závěr

Hlavním cílem práce bylo analyzovat, navrhnout a implementovat mobilní aplikaci pro systém iOS (frontend i backend) pro zaznamenávání odpracovaného času, která měla umožnit integraci s různými spouštěči akcí a propojení s dalšími systémy pro zaznamenávání a spravování odpracovaného času. V rámci návrhu bylo cílem navrhnout vhodné technické řešení pro implementaci aplikace, funkcionality aplikace a jejich uživatelské rozhraní a tohoto návrhu se poté při realizaci aplikace držet. Následně bylo cílem práce realizované řešení vhodně otestovat a provést uživatelské testování.

V rámci analýzy se podařilo poskytnout přehled o tom, co je problematika měření a správy odpracovaného času, jaké jsou možnosti propojení s různými spouštěči akcí, jaké jsou možnosti integrace s existujícími systémy, které řeší stejnou problematiku, a o tom, jaká jsou specifika vývoje pro mobilní platformu iOS. Návrh implementace poté navrhl technická řešení a přístupy pro realizaci, funkcionality aplikace a jejich uživatelské rozhraní, čehož se poté držela realizace aplikace. Výsledkem poté byla plně funkční mobilní aplikace, která umožňuje některé formy integrace se spouštěči a s dalšími systémy.

Realizovaná aplikace implementuje funkcionality aplikace pro měření a správu odpracovaného času – umožňuje měření času pomocí časovače nebo pomocí manuálního zadání a udržuje historii časových záznamů, ke kterým může přidělovat projekt a popis. Z hlediska integrace aplikace implementuje základní funkce, jako propojení se systémovými zkratkami, které mají teoreticky neomezené možnosti pro tvorbu automatizací, nebo možnost exportu historie záznamů do CSV souboru, což umožňuje import historie do mnoha existujících systémů.

Vedlejším cílem práce byla flexibilita a rozšiřovatelnost technické implementace aplikace. Během návrhu a realizace celé platformy byl kladen důraz na její přehlednost a rozšiřovatelnost, která je díky použití technologií jako *Kotlin Multiplatform*, nebo díky implementaci vlastního backendu, dobře připravena na rozšíření nejen o další cílové platformy (Android, Web a další), ale i na rozšíření o další funkcionality, jako možnosti integrace s dalšími systémy.

Velkým přínosem pro budoucí vývoj aplikace jsou také výsledky uživatelského testování, kterého se zúčastnilo 5 testerů. Většina testerů měla větší zkušenost s používáním mobilních aplikací pro měření a správu odpracovaného času, a poskytli tak relevantní zkušenost cílového uživatele s používáním aplikace. Z testování vzešla řada poznatků, které byly popsány, analyzovány a ze kterých vznikla doporučení, jak by měla být aplikace upravena.

Na závěr aplikace navrhuje prostředí pro budoucí provoz a podporu aplikace, které navrhuje řadu možností pro budoucí vylepšení a rozšíření, úpravy rozhraní a další. Dále také popisuje, jakým způsobem by bylo možné aplikaci finálně nasadit mezi reálné uživatele, nebo jakým způsobem aplikaci rozšířit o podporu pro další platformy.

Práce také může poskytnout určitou formu inspirace nebo seznámení pro čtenáře, kteří se zajímají o různé přístupy k možnostem vývoje mobilních aplikací, nebo o konkrétní technologie, kterých tato práce v rámci realizace využila.

Příloha A

Pokyny k uživatelskému testování

Úvod

Během testování budete dostávat psané instrukce, které se budete snažit splnit. Snažte se prosím komentovat své postupy, myšlenky a interakce s aplikací. Jakákoli intuice nebo očekávání, jak byste danou instrukci řešili, kde byste očekávali obrazovku nebo prvek aplikace, cokoli Vás napadne.

Pamatuje, že jakýkoli průběh, ať už splní nebo nesplní cíl instrukce, nikdy není Vaší chybou, ale chybou rozhraní aplikace. Vaše dojmy, jakožto uživatele, který s aplikací interaguje poprvé, jsou v tomto ohledu zásadní pro zhodnocení uživatelského rozhraní aplikace.

Během testu bude nahrávána Vaše interakce s aplikací a bude s Vámi moderátor testu. Moderátor by Vám neměl radit v interakci s aplikací, pouze by Vám měl pomoct s průběhem testu, či s vyjasněním zadání.

Kdykoli Vám bude něco v rozhraní připadat zvláštní, neintuitivní, budete něčím překvapeni nebo nebudete vědět, jak dané zadání splnit, zkuste vždy tuto situaci popsat.

Jakmile budete cíl daného scénáře testu považovat za splněný, informujte o tom moderátora.

Dotazník před testem

- Jaké je Vaše jméno?
- Kolik Vám je let?
- Používáte často mobilní aplikace? A s jakou platformou máte nejvíce zkušeností (iOS/Android)?
- Používáte nějaké aplikace pro měření odpracovaného času? Pokud ano, jaké?

Scénáře testu

1. Tvorba uživatelského účtu

Na mobilním telefonu spusťte aplikaci *Trackee* a pokuste se pro sebe vytvořit nový uživatelský účet. E-mail a heslo můžete zvolit jaké chcete.

2. Vytvoření nového klienta

Vytvořte v aplikaci 2 nové klienty, pro které budete dále moci vytvořit projekty. Pro prvního klienta použijte název *Matee Devs* a pro druhého klienta použije název *FIT ČVUT*.

3. Vytvoření nového projektu

Vytvořte v aplikaci 2 nové projekty s následujícími vlastnostmi:

- První projekt
  - **Klient**: Matee Devs
  - **Název**: Unikátní půllitr
  - **Typ projektu**: Work
- Druhý projekt
  - **Klient**: FIT ČVUT
  - **Název**: Diplomová práce
  - **Typ projektu**: School

4. Spuštění časovače

Spusťte časovač pro měření odpracovaného času. Zvolte, aby časovač měřil práci pro projekt *Unikátní půllitr* (klient *Matee Devs*) a přidejte popis *Code reviews*.

5. Změna začátku časovače

Předpokládejte, že jste časovač spustili až později, než jste na této práci skutečně začali pracovat. Pokuste se proto posunout začátek, od kdy časovač měří, o 2 hodiny dříve, než je nyní.

6. Zastavení časovače

Zastavte měření časovače, aby se Vám uložila do teď měřená práce do historie záznamů.

7. Manuální přidání časového záznamu

Předpokládejte, že jste si vzpomněl(a), že jste včera od *11:03* do *13:49* pracovali na projektu *Diplomová práce* (klient *FIT ČVUT*), ale tuto skutečnost jste zapomněli zaevidovat do aplikace. Vytvořte pro to manuálně nový časový záznam podle těchto údajů a přidejte mu popis *Analýza*, aby nyní tento záznam byl součástí historie záznamů.

8. Úprava projektu

Upravte projekt *Diplomová práce* (klient *FIT ČVUT*) tak, aby nový název projektu byl *Diplomová práce a obhajoba*. Zkontrolujte, že historie časových záznamů reflektuje tuto změnu.

9. Odhlášení a přihlášení na testovací účet

Odhlašte se ze svého účtu a přihlašte se na testovací účet, který již obsahuje řadu klientů, projektů a časových záznamů. Jako e-mail použijte *usertesting@trackee.app* a jako heslo použijte *nejlepsiappka*.

10. Odstranění časového záznamu

Smažte z historie záznamů záznam z *8. února* s popisem *Workshop na dojení zákazníků*.

11. Export historie do CSV souboru

Vytvořte integraci pro export historie do CSV souboru. Název integrace použijte jaký chcete. Exportuje poté historii v intervalu od *1.1.2024* do *30.4.2024*. Výsledný soubor zkontrolujte v aplikaci *Numbers*.

12. Odstranění klienta

Předpokládejte, že jste si v exportované tabulce všimli záznamů projektu *Vyšetřování podezřelých aktivit kolegy Hanžlíka* a že tyto záznamy v tabulce nechcete, protože vaše spolupráce s klientem *Fitify* je tajná. Smažte proto radši celého klienta *Fitify* úplně a znovu vyexportujte data do CSV souboru od *1.1.2024* do *30.4.2024* a v aplikaci *Numbers* ověřte, že se zde zmíněné záznamy již nenacházejí.

13. Tvorba automatizace pro spuštění časovače

Pomocí aplikace *Zkratky* vytvořte novou automatizaci, která automaticky spustí časovač pro projekt *Diplomová práce* (klient *FIT ČVUT*) s popisem *Obhajoba*, pokud dorazíte na místo *Thákurova 9, Praha 6*. Zkuste zkratku spustit a vyzkoušet, že se časovač opravdu spustil.

Dotazník po testu

- Jaký máte celkový dojem z používání aplikace? Co se Vám líbilo a co se Vám nelíbilo?
- Stalo se Vám u nějakého bodu, že jste si nevěděl(a) rady s tím, jak ho splnit? Čím to podle Vás bylo a co by tomu pomohlo?
- Používal(a) byste tuto aplikaci?

Příloha B

Protokol z uživatelského testování

iOS	Uživatel platformy iOS
Android	Uživatel platformy Android
Clockify	Uživatel aplikace Clockify
Zvýrazněný text	Důležitý poznatek testu
(!)	Kritický poznatek testu

Tester F. W. (27 let, Android, Clockify)

Tvorba uživatelského účtu: Klikl na tlačítko Registrovat a pokusil se do pole e-mailu zadat nevalidní e-mail ve zvědavosti, zda to aplikace správně detekuje. Poté se také zkusil zadat různá hesla, aby zjistil, zda to aplikace verifikuje, a zjistil, že ano. Chtěl druhé heslo upravit a **vyjádřil zmatení nad tím, že ve stavu, kdy má heslo zobrazené, nevidí kurzor**. Heslo opravil, ale aplikace po snaze se registrovat **ukázala neznámou chybu** (!). Uživatel předpokládal, že je to kvůli nevalidnímu formátu e-mailu, tak ho upravil, ale **aplikace stále ukazovala neznámou chybu** (!). Také zmínil, že mu vadí, že **klávesnice při zadávání e-mailu nenabízí v primárním rozmístění kláves tečku**. Tak zkusil celý proces znovu s jiným heslem a registrace už se podařila, a uživatel předpokládal, že **měl asi příliš krátké heslo**. Poté si začal prohlížet aplikaci a zmínil, že by mu **zde dával smysl nějaký onboarding, aby věděl, že si musí vytvořit projekt**. Při prohlížení karty integrací zmínil, že **CSV mu nepřijde jako typ integrace**, ale že tomu rozumí.

Vytvoření nového klienta: Během prohlížení aplikace už se dostal na kartu profilu, takže poté rovnou intuitivně klikl na tlačítko Klienti. Ihned poté si všiml tlačítka + a klikl na něj, poté zadal název klienta. Poté klikl na tlačítko Uložit a zmínil, že by mu **dávalo větší smysl například Vytvořit**. Poté stejným způsobem vytvořil druhého klienta. Kliknout na tlačítko Uložit se mu ale **podařilo až na 3. pokus**.

Vytvoření nového projektu: Rovnou se vrátil na profil a klikl na tlačítko Projekty a poté na tlačítko +. Zadal předepsané parametry a při výběru klienta zmínil, že by očekával, že **při kliknutí na klienta se aplikace rovnou vrátí o obrazovku zpět**, že to nebude muset dělat dalším kliknutím na Uložit. Při vybírání typu projektu zmínil, že neví, kde se tento seznam typů vzal, že ho nevytvářel, takže je to asi výchozí. Opět se mu **nedařilo kliknout na tlačítko + a musel to dělat vícekrát**. Druhý projekt už vytvořil bez problémů, pouze zmínil údiv nad autokorekcí systému iOS.

Spuštění časovače: Přesunul se na kartu časovače a intuitivně klikl na tlačítko pro spuštění. Ihned poté klikl na výběr projektu, vybral projekt a zmínil, že **je otravné, že opět musí klikat na tlačítko Uložit**. Popis zadal bez problému.

Změna začátku časovače: Intuitivně rovnou klikl na stopky časovače a správně posunul čas začátku. Poté si ještě vyzkoušel, co se stane, když nastaví čas začátku v budoucnu, a aplikace mu správně vrátila chybu, že to nejde.

Zastavení časovače: Intuitivně klikl na tlačítko pro stopnutí a viděl, že se záznam uložil. Také zmínil, že mu nastavení z minulého měření zůstalo v časovači.

Manuální přidání časového záznamu: Intuitivně klikl na tlačítko +, ale **zmátlo ho, jak na to rozhraní ovladače reagovalo**, že se mu tam objevil nějaký náhodný čas. Chvíli to zkoumal a zmínil, že mu to přijde **neintuitivní** a že by **spíše čekal, že se mu objeví nějaký dialog**. Po chvíli pochopil, že jde o nějakou formu přepínání, a dodal, že je to asi o zvyku. Poté ale už správně zadal čas začátku a konce a zmínil, že informace o tom, jak dlouhý bude výsledný interval, je poměrně schovaná. Poté přemýšlel, jak záznam uloží, a klikl na tlačítko + a viděl, že se záznam uložil správně. Poté si všiml, že zapomněl změnit projekt a popis podle zadání, a **snažil se záznam upravit**, ale všiml si, že to asi nepůjde. Moderátor ho ujistil, že znova vytvářet záznam kvůli tomu nemusí. Zopakoval, že dialog by mu dával větší smysl, protože mu přesně nedošlo, co se děje.

Úprava projektu: Klikl na kartu profilu a byl chvíli zmaten, že zůstala zanořena v projektech. Klikl na správný projekt, změnil jeho název, přešel na časovač, ale tam se nic nezměnilo, protože u záznamu zapomněl přiřadit správný projekt.

Odhlášení a přihlášení na testovací účet: Šel na kartu profilu, odhlásil se a zadal údaje testovacího účtu. Po přihlášení si všiml, že záznamů už je více.

Odstranění časového záznamu: Nejprve záznam nemohl najít, protože **čekal, že popis (doslovně ale řekl title) bude nahoře, ne pod projektem a klientem**. Chvíli zkoumal, jak záznam smazat, několikrát se na něj pokusil kliknout, a poté intuitivně zkusil swipe-to-delete gesto, u kterého ho **překvapilo, že není žádný potvrzující dialog**.

Export historie do CSV souboru: Zmínil, že mu přijde **zvláštní mít export v integracích**, ale přešel to. Novou integraci vytvořil a data exportoval bez problémů. Moderátor poté poradil, jak soubor otevřít v Numbers, jelikož tento dialog už není součástí aplikace.

Odstranění klienta: Při návratu do aplikace zmínil, že **by čekal, že se aplikace po exportu vrátí o obrazovku zpět**. Smazání provedl bez problémů, pouze zmínil, že **pro kliknutí na tlačítko musí kliknout na jeho text**. Druhý export byl také bez problémů.

Tvorba automatizace pro spuštění časovače: Moderátor testerovi radil, jak zkratky a automatizace tvořit, jelikož se nejedná o rozhraní aplikace. Nejprve zapomněl na nastavení parametrů časovače, moderátor mu to musel připomenout.

Tvorba automatizace pro vypnutí časovače: Upravil existující automatizaci a změnil zapnutí časovače na vypnutí.

Shrnutí: Zopakoval přednost popisů před názvem projektu a klienta, a také přepínání ovladače časovače. Také zopakoval, že by se mu hodil onboarding, ale jinak že mu používání aplikace dává smysl. Poté ještě zopakoval absenci potvrzení při mazání záznamu a poukázal na chybnou animaci při mazání. Zmínil, že mu přijde, že souhrn odpracovaných hodin mu přijde schovaný, a že by dnešek čekal vedle nadpisu skupiny, jako je to u ostatních dnů. Pak ho ještě napadlo, že integrace by možná nečekal v navigační liště, že to uživatelé asi nebudou používat příliš často, že to jen nastaví a pak to bude fungovat. Při zhodnocení, zda by aplikaci používal, tak zmínil, že asi ne, že mu aplikace přijde pomalá, nemůže upravovat záznamy, že v této formě by jí asi nepoužíval, protože je zkrátka zvyklý na jinou aplikaci.

Tester D. K. (24 let, iOS, Clockify)

Tvorba uživatelského účtu: Proběhlo bez problému, vše našel intuitivně.

Vytvoření nového klienta: **Nejprve klikl na + na ovladači časovače**, kde zjistil, že to dělá něco jiného. Poté **klikl na výběr projektu**, kde přečetl instrukci, že si musí vytvořit projekt. Moderátor ho upozornil, že přeskočil instrukci pro vytvoření klienta, poté omylem klikl na odhlášení a musel se znovu přihlásit. Vytvoření klienta poté proběhlo bez problémů.

Vytvoření nového projektu: Proběhlo bez problémů, vše našel intuitivně.

Spuštění časovače: Vybral projekt, poté spustil časovač a poté přidal popis.

Změna začátku časovače: Proběhlo bez problémů, vše našel intuitivně.

Zastavení časovače: Proběhlo bez problémů.

Manuální přidání časového záznamu: Proběhlo bez problémů, vše našel intuitivně.

Úprava projektu: **Nejprve zkusil klikat na záznam** a když zjistil, že to nic nedělá, upravil projekt standardně v profilu.

Odhlášení a přihlášení na testovací účet: Při přihlášení začal zadávat vlastní údaje, moderátor ho upozornil, aby zadal předepsané údaje. Zbytek proběhl bez problémů.

Odstranění časového záznamu: V testovacích datech chyběl předepsaný záznam pro smazání, což bylo chybou moderátora, že data špatně připravil. Instruoval proto testera, aby smazal jiný záznam. **Nejprve zkusil na záznam kliknout** – nic se nestalo. **Poté na něm zkusil podržet prst** – nic se nestalo. Hned poté zkusil swipe-to-delete a záznam smazal. Měl u toho pocit, že se animace trochu zasekává, ale přešel to a pokračoval.

Export historie do CSV souboru: Proběhlo bez problémů, vše našel intuitivně.

Odstranění klienta: Proběhlo bez problémů, vše našel intuitivně.

Tvorba automatizace pro spuštění časovače: Moderátor testerovi radil, jak aplikaci Zkratky používat. Po spuštění zkratky zmínil, že **by čekal, že se aplikace automaticky scrollne dolů, nebo nějakým jiným způsobem naznačí, že se něco stalo**.

Shrnutí: Zopakoval, že by přidal nějakou indikaci, že se po spuštění zkratky něco stalo. Také zmínil, že by na hlavní obrazovku přidal tlačítko pro přidání profilu nebo klienta, pokud žádné nemá, nebo rovnou celý onboarding.

Tester D. Ž. (31 let, Android, Clockify)

Tvorba uživatelského účtu: Během ověřování zadaného hesla napsal druhé heslo chybně, a když ho chtěl upravit a smazat poslední znak, tak ho znepokojilo, že se smazal celý obsah pole. Znovu heslo zadal a úspěšně se registroval.

Vytvoření nového klienta: Proběhlo bez problému, vše našel intuitivně. **Klikání na tlačítko + nebo Uložit se mu podařilo až na více pokusů**.

Vytvoření nového projektu: Vše našel intuitivně. **Pozastavil se nad tím, že po vybrání klienta k projektu musí klikat na Uložit**.

Spuštění časovače: Proběhlo bez problému, vše našel intuitivně.

Změna začátku časovače: Nejdříve zastavil časovač a poté přemýšlel, co má dělat, snažil se nejprve upravit uložený záznam. Poté přepnul ovladač do manuálního zadávání času a zadal čas. Poté přemýšlel, co dělat dál – nevěděl, jak záznam uložit. Ptal se, že by se záznam po uložení měl asi zobrazit v historii. Moderátor poté upřesnil instrukci, že začátek času měl být upraven u již běžícího časovače. Poté časovač znovu zapl a začátek upravil správně.

Zastavení časovače: Zmínil, že už to udělal v minulém kroku, a test přeskočil.

Manuální přidání časového záznamu: Zadání času našel intuitivně. **Musel opět několikrát kliknout na tlačítko Uložit, aby se dotyk zaregistroval**. Poté tlačítkem + záznam uložil.

Úprava projektu: Proběhlo bez problému, vše našel intuitivně.

Odhlášení a přihlášení na testovací účet: **V kartě profilu byl zmaten, že není v profilu, ale ve vnořené obrazovce projektů**, ale pak se vrátil o obrazovku zpátky a zbytek proběhl bez problému.

Odstranění časového záznamu: **Nejprve se snažil kliknout** a poté vyzkoušel swipe-to-delete gesto a záznam smazal.

Export historie do CSV souboru: Proběhlo bez problému, vše našel intuitivně.

Odstranění klienta: Proběhlo bez problému, vše našel intuitivně.

Tvorba automatizace pro spuštění časovače: Moderátor radil, jak aplikaci Zkratky používat. Zbytek proběhl bez problému.

Shrnutí: Zmínil, že protože je zvyklý na platformu Android, tak občas nevěděl, co se kde nachází, ale jinak neměl problém. Líbilo se mu, že aplikace je intuitivní. Zmínil, že byl akorát zmatený z editace času.

Tester E. Č. (24 let, iOS)

Tvorba uživatelského účtu: Nejprve se snažil zadat údaje v obrazovce pro přihlášení, ne pro registraci. Poté pochopil, že se musí nejdřív registrovat. Zbytek proběhl bez problému.

Vytvoření nového klienta: Proběhlo bez problému, vše našel intuitivně.

Vytvoření nového projektu: Proběhlo bez problému, vše našel intuitivně.

Spuštění časovače: Zapnul časovač a poté vybral projekt. Zeptal se moderátora, jestli časovač běží od doby, co byl zapnut, nebo od doby, co byl vybraný projekt. Moderátor poradil, ať se řídí podle toho, co ukazují stopy časovače.

Změna začátku časovače: Zeptal se moderátora, jestli má časovač stopovat. Ten poté poradil, ať se jen pokusí změnit čas začátku. **Nejdřív se pokusil kliknout na shrnutí odpracovaných hodin za den a týden**, nic se nestalo, tak poté klikl správně na stopky.

Zastavení časovače: Již vyzkoušel na konci předchozího scénáře.

Manuální přidání časového záznamu: Potřeboval pomoct s vysvětlením, co má vlastně udělat, že se jedná o časový záznam a ne o projekt. Poté si nebyl jistý, jak má záznam uložit, protože nikde nevidí tlačítko Uložit. Vyzkoušel tlačítko +, které záznam úspěšně uložilo. Poté si uvědomil, že zapomněl změnit popis, a **snažil se vytvořený záznam upravit**, což nešlo. Rovnou ho tedy zkusil smazat intuitivně pomocí swipe-to-delete gesta. Poté korektně vytvořil nový záznam.

Úprava projektu: **Nejdříve se snažil kliknout na časový záznam, aby ho upravil**. Když se nic nestalo, tak šel na profil do projektů a projekt upravil.

Odhlášení a přihlášení na testovací účet: Byl zmatený, že po kliknutí na kartu profilu se nenacházel na profilu, ale ve vnořené obrazovce projektů, a nevěděl, jak se má dostat do profilu. Moderátor poradil, že je potřeba se vrátit ze zanoření. Při přihlášení omylem zadal v hesle tečku navíc, a **když ho chtěl poté upravit, tak ho podrážděno, že se smazalo celé heslo a ne jen tečka**.

Odstranění časového záznamu: Proběhlo bez problému, vše našel intuitivně.

Export historie do CSV souboru: Potřeboval poradit, co je integrace a co je CSV soubor. Moderátor podal instrukci, ať se to pokusí v aplikaci najít, a zbytek proběhl bez problémů.

Odstranění klienta: **Nejdříve se klienta pokusil smazat pomocí swipe-to-delete, které ale u klientů nefunguje** – tuto skutečnost také popsal, že je to matoucí. Poté rozklikl detail a smazal klienta tam. Poté také zmínil, že mazání trvá dlouho.

Tvorba automatizace pro spuštění časovače: Moderátor testerovi radil, jak aplikaci Zkratky používat. Tester upozornil na to, že v zadání testu je špatný název projektu, který má být vybrán.

Shrnutí: Zmínil, že mobilní aplikace moc nepoužívá, takže byl občas zmaten, jak se co ovládá. S rozhraním aplikace byl spokojený, ale zmínil, že aplikace pro měření času nepoužívá, takže mu doména není moc blízká.

Tester T. S. (32 let, Android, Clockify)

Tvorba uživatelského účtu: **Divil se, že při zadávání e-mailové adresy není rozmístění kláves přizpůsobeno pro zadávání e-mailové adresy. Při úpravě hesla měl pocit, že nemůže do textového pole psát, protože nebyl při viditelném stavu vidět kurzor, a opakovaně na pole klikal.** Tuto skutečnost také zmínil.

Vytvoření nového klienta: Proběhlo bez problému, vše našel intuitivně.

Vytvoření nového projektu: Vyzkoušel, co se stane, když nevybere klienta. Aplikace zareagovala příslušnou chybovou hláškou. Zadal jiné názvy, ale moderátor podotkl, že to nevadí. Zbytek proběhl bez problému.

Spuštění časovače: **Zmínil, že po výběru projektu by čekal, že se dialog zavře**, že klikání na Uložit je otravné. Zbytek proběhl bez problému.

Změna začátku časovače: Proběhlo bez problému, vše našel intuitivně.

Zastavení časovače: Proběhlo bez problému, vše našel intuitivně. Po uložení záznamu zmínil, že až teď pochopil, proč byla obrazovka nad ovladačem časovače prázdná.

Manuální přidání časového záznamu: **Zmínil, že klikatelná plocha popisu je poměrně malá.** Zbytek proběhl bez problému.

Úprava projektu: Proběhlo bez problému, vše našel intuitivně.

Odhlášení a přihlášení na testovací účet: Proběhlo bez problému, vše našel intuitivně.

Odstranění časového záznamu: **Nejprve si daného záznamu nevšiml, protože popis očekával na prvním řádku.** Zmínil, že má dojem, že větší důraz je na popis, než projekt.

Export historie do CSV souboru: Nejprve hledal, kde integraci vytvořit, **byl zmaten z toho, že export do CSV je mezi integracemi**. Zbytek proběhl bez problému.

Odstranění klienta: **Nejprve zkoušel podržet na klientovi, poté swipe-to-delete**, nic se ale nedělo. Poté až se dostal na detail, kde vidět tlačítko pro smazání. U potvrzovacího dialogu zmínil, že **by čekal, že potvrzovací tlačítko bude obsahovat text Smazat**. Zbytek proběhl bez problému.

Tvorba automatizace pro spuštění časovače: **Zmínil, že by čekal, že nabídka parametrů bude otevřená rovnou. Po spuštění zkratky zmínil, že by čekal, že se obrazovka automaticky posune dolů.** Dále zmínil, že by **přidal červenou barvu jako pozadí ovládacího tlačítka časovače, pokud běží, jelikož to je zvykem u existujících řešení**.

Shrnutí: Přišlo mu, že aplikace se chová standardně. Líbilo se mu, že se nahoře v časovači vidí shrnutí odpracovaného času a že seznam historie je řazen zespodu. Zopakoval, že si není jistý, jestli by projekt měl být před popisem, že by dal popisu větší prioritu. Také zmínil, že by přidal potvrzovací dialog při mazání záznamu. Také zmínil překvapení nad tím, že export do CSV je řazen mezi integrace, když se jedná o export – přemýšlel nad tím, že by to oddělil, ale tím si taky nebyl jistý.

Bibliografie

1. *Clockify iOS app* [online]. [cit. 2024-02-20]. Dostupné z: https://clockify.me/iphone-time-tracking.
2. *TIMEFLIP2* [online]. [cit. 2024-02-20]. Dostupné z: https://timeflip.io.
3. *TIMEULAR Tracker* [online]. [cit. 2024-02-20]. Dostupné z: https://timeular.com/tracker/.
4. *timeBuzzer* [online]. [cit. 2024-02-20]. Dostupné z: https://timebuzzer.com/buzzer/.
5. *TIMEFLIP API* [online]. [cit. 2024-02-20]. Dostupné z: https://newapi.timeflip.io/swagger-ui.html?_gl=1*cc57tk*_ga*ODA0NjI0MjkuMTcwODQyMDU4Mw..*_ga_Z1JPGVWR86*MTcwODQzNzA3My40LjEuMTcwODQzODAyMi4wLjAuMA..#/.
6. *TIMEULAR API* [online]. [cit. 2024-02-20]. Dostupné z: https://developers.timeular.com.
7. *timeBuzzer API* [online]. [cit. 2024-02-20]. Dostupné z: https://github.com/timeBuzzer/timebuzzer-open-api-doc.
8. *TIMEFLIP BLE protocol* [online]. [cit. 2024-02-20]. Dostupné z: https://github.com/DI-GROUP/TimeFlip.Docs/blob/master/Hardware/TimeFlip%20BLE%20protocol%20ver4_02.06.2020.md.
9. *iOS režimy soustředění* [online]. [cit. 2024-02-21]. Dostupné z: https://support.apple.com/cs-cz/guide/iphone/iphd6288a67f/ios.
10. *Focus – Adjust your app's behavior and filter incoming notifications when the current Focus changes.* [online]. [cit. 2024-02-21]. Dostupné z: https://developer.apple.com/documentation/appintents/focus.
11. *Zkratky – App Store* [online]. [cit. 2024-02-21]. Dostupné z: https://apps.apple.com/cz/app/shortcuts/id1462947752?l=cs.
12. *Uživatelská příručka pro Zkratky* [online]. [cit. 2024-02-21]. Dostupné z: https://support.apple.com/cs-cz/guide/shortcuts/welcome/ios.
13. *Shortcuts for developers* [online]. [cit. 2024-02-21]. Dostupné z: https://developer.apple.com/shortcuts/.
14. MAIN, Kelly. *7 Free Time Tracking Apps (2024)* [online]. Forbes, 2024 [cit. 2024-02-21]. Dostupné z: https://www.forbes.com/advisor/business/software/free-time-tracking-apps/.
15. REPLOGLE, Nicole. *The 6 best time tracking apps in 2024* [online]. Zapier, 2023 [cit. 2024-02-21]. Dostupné z: https://zapier.com/blog/best-time-tracking-apps/.

16. *Clockify on App Magic* [online]. [cit. 2024-02-21]. Dostupné z: https://appmagic.rocks/iphone/clockify-legacy-/1304431926.
17. *Join millions who use Clockify* [online]. [cit. 2024-02-21]. Dostupné z: https://clockify.me/customers.
18. *Clockify – Time tracking apps* [online]. [cit. 2024-02-21]. Dostupné z: https://clockify.me/apps.
19. *Clockify – Choose your pricing plan* [online]. [cit. 2024-02-21]. Dostupné z: https://clockify.me/pricing#compare-table.
20. *Clockify – Features* [online]. [cit. 2024-02-21]. Dostupné z: https://clockify.me/features/.
21. *Clockify API documentation* [online]. [cit. 2024-02-21]. Dostupné z: https://docs.clockify.me/.
22. *Clockify – Import timesheets* [online]. [cit. 2024-02-21]. Dostupné z: https://clockify.me/help/extra-features/import-timesheets.
23. *Toggl Track on App Magic* [online]. [cit. 2024-02-21]. Dostupné z: https://appmagic.rocks/iphone/toggl-track-hours-and-time-log/1291898086.
24. *Toggl Track* [online]. [cit. 2024-02-21]. Dostupné z: https://toggl.com.
25. *Toggl track – CSV import guide* [online]. [cit. 2024-04-22]. Dostupné z: https://support.toggl.com/en/articles/3928821-toggl-track-csv-import-guide.
26. *Toggl track – Do you have an API available?* [online]. [cit. 2024-04-22]. Dostupné z: https://support.toggl.com/en/articles/2559637-do-you-have-an-api-available.
27. *Deputy on App Magic* [online]. [cit. 2024-02-21]. Dostupné z: https://appmagic.rocks/iphone/deputy-employee-scheduling/477070330.
28. *Accurate employee timesheet software & app, pass to payroll in a click* [online]. [cit. 2024-02-21]. Dostupné z: https://www.deputy.com/features/time-and-attendance.
29. *Deputy – Bulk import schedules and timesheets* [online]. [cit. 2024-04-22]. Dostupné z: https://help.deputy.com/hc/en-au/articles/6325924665871-Bulk-import-schedules-and-timesheets.
30. KOTHAWADE, Niraj. *Deputy – Getting Started with the Deputy API* [online]. 2023. [cit. 2024-04-22]. Dostupné z: https://developer.deputy.com/deputy-docs/docs/getting-started-with-the-deputy-api.
31. SNELL, Jason. *iPhone (2007) review: A game-changer years in the making* [online]. Macworld, 2022 [cit. 2024-02-22]. Dostupné z: https://www.macworld.com/article/186335/original-iphone-review-2.html.
32. ELGAN, Mike. *Why iOS 7 Is A Masterpiece of Design* [online]. Cult of Mac, 2013 [cit. 2024-02-22]. Dostupné z: https://www.cultofmac.com/232040/why-ios-7-is-a-masterpiece-of-design/?utm_content=cmp-true.
33. COSTELLO, Sam. *The History of iOS, from Version 1.0 to 17.0* [online]. Lifewire, 2023 [cit. 2024-02-22]. Dostupné z: https://www.lifewire.com/ios-versions-4147730#toc-ios-1.
34. SHEWALE, Rohit. *32 iPhone User Statistics: Sales, Usage & Revenue (2024)* [online]. Demandsage, 2024 [cit. 2024-02-22]. Dostupné z: https://www.demandsage.com/iphone-user-statistics/#.
35. BOHON, Cory. *Xcode Through the Years* [online]. Martiancraft, 2022 [cit. 2024-02-22]. Dostupné z: https://martiancraft.com/blog/2022/01/xcode-through-the-years/.

36. *About Objective-C* [online]. Apple [cit. 2024-02-22]. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.
37. *Programovací jazyk Swift* [online]. Apple [cit. 2024-02-22]. Dostupné z: https://developer.apple.com/swift/.
38. *Knihovna SwiftUI* [online]. Apple [cit. 2024-02-22]. Dostupné z: https://developer.apple.com/xcode/swiftui/.
39. *Guide to app architecture* [online]. Google [cit. 2024-02-23]. Dostupné z: https://developer.android.com/topic/architecture.
40. *SwiftUI tutorials – App design and layout – Composing complex interfaces* [online]. Apple [cit. 2024-02-23]. Dostupné z: https://developer.apple.com/tutorials/swiftui/composing-complex-interfaces.
41. ORLOV, Bohdan. *iOS Architecture Patterns* [online]. Medium, 2015 [cit. 2024-02-23]. Dostupné z: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52.
42. ILLIOPOULOS, Orfeas. *Building Scalable iOS Apps with the MVVM-C Design Pattern* [online]. Medium, 2023 [cit. 2024-02-23]. Dostupné z: https://blog.devgenius.io/building-scalable-ios-apps-with-the-mvvm-c-design-pattern-a6756e3611d1.
43. ASHRAFI, Siamak (Ash). *SwiftUI — Model View Intent (MVI)* [online]. Medium, 2019 [cit. 2024-02-23]. Dostupné z: https://zoewave.medium.com/swiftui-is-model-view-intent-mvi-fd142b12fc81.
44. KUDINOV, Oleh. *Clean Architecture and MVVM on iOS* [online]. Medium, 2019 [cit. 2024-02-23]. Dostupné z: https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3.
45. *Alternativní interpretace pojmů cross-platform a multi-platform* [online]. [cit. 2024-04-22]. Dostupné z: https://topdigital.agency/cross-platform-and-multi-platform-differences-to-know/.
46. SZILVÁSI, Péter. *Další alternativní interpretace pojmů cross-platform a multi-platform* [online]. [cit. 2024-04-22]. Dostupné z: https://stackoverflow.com/a/77254178/16712624.
47. *Flutter cross-platform* [online]. [cit. 2024-04-22]. Dostupné z: https://flutter.dev/multi-platform.
48. *React Native cross-platform* [online]. [cit. 2024-04-22]. Dostupné z: https://reactnative.dev/architecture/xplat-implementation.
49. *Xamarin cross-platform* [online]. [cit. 2024-04-22]. Dostupné z: https://learn.microsoft.com/en-us/xamarin/cross-platform/.
50. *Kotlin Multiplatform* [online]. [cit. 2024-04-22]. Dostupné z: https://kotlinlang.org/docs/multiplatform.html.
51. *Firebase* [online]. [cit. 2024-04-22]. Dostupné z: https://firebase.google.com.
52. *AWS Amplify* [online]. [cit. 2024-04-22]. Dostupné z: https://aws.amazon.com/amplify.
53. *Parse* [online]. [cit. 2024-04-22]. Dostupné z: https://parseplatform.org.
54. *Introduction to Node.js* [online]. [cit. 2024-04-22]. Dostupné z: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs.
55. *Express – Node.js framework* [online]. [cit. 2024-04-22]. Dostupné z: https://expressjs.com.
56. *NestJS* [online]. [cit. 2024-04-22]. Dostupné z: https://nestjs.com.

57. *Ruby on Rails* [online]. [cit. 2024-04-22]. Dostupné z: https://rubyonrails.org.
58. *Django* [online]. [cit. 2024-04-22]. Dostupné z: https://www.djangoproject.com.
59. *Spring Boot* [online]. [cit. 2024-04-22]. Dostupné z: https://spring.io/projects/spring-boot.
60. *Spring Boot vs Node JS: Choosing the Right Framework for Your Business Project!* [online]. Matellio, 2023 [cit. 2024-04-22]. Dostupné z: https://www.matellio.com/blog/spring-boot-vs-node-js.
61. *Oracle Database* [online]. [cit. 2024-04-24]. Dostupné z: https://www.oracle.com/database/.
62. *PostgreSQL* [online]. [cit. 2024-04-24]. Dostupné z: https://www.postgresql.org.
63. *MongoDB* [online]. [cit. 2024-04-24]. Dostupné z: https://www.mongodb.com.
64. *Redis* [online]. [cit. 2024-04-24]. Dostupné z: https://redis.io.
65. SVOBODA, Martin. *Lecture 1: Introduction* [online]. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023 [cit. 2024-04-24]. Dostupné z: https://www.ksi.mff.cuni.cz/~svoboda/courses/231-NIE-PDB/lectures/NIEPDB-Lecture-01-Introduction.pdf.
66. *MongoDB – Database as a Service* [online]. [cit. 2024-04-24]. Dostupné z: https://www.mongodb.com/database-as-a-service.
67. *MongoDB Atlas* [online]. [cit. 2024-04-24]. Dostupné z: https://www.mongodb.com/atlas/database.
68. *Figma* [online]. [cit. 2024-04-24]. Dostupné z: https://www.figma.com.
69. *Figma – Apple Design Resources – iOS 17* [online]. [cit. 2024-04-24]. Dostupné z: https://www.figma.com/file/L21Z0e7HiE3Cw5ftGTqn7v/Apple-Design-Resources-%E2%80%93-iOS-17-and-iPadOS-17-(Community).
70. *Figma – TrackeeApp* [online]. [cit. 2024-05-02]. Dostupné z: https://www.figma.com/file/qfU8hog2U6S4tPf5z6GOiZ/TrackeeApp?type=design&node-id=0%3A1&mode=design&t=HbKXcN4BTcpqEPMl-1.
71. *Apple Developer – Human Interface Guidelines – Designing for iOS* [online]. [cit. 2024-04-24]. Dostupné z: https://developer.apple.com/design/human-interface-guidelines/designing-for-ios.
72. *Spendee* [online]. [cit. 2024-04-24]. Dostupné z: https://www.spendee.com.
73. *Fondee* [online]. [cit. 2024-04-24]. Dostupné z: https://www.fondee.cz.
74. *Úřad průmyslového vlastnictví – Rešeršní databáze* [online]. [cit. 2024-04-24]. Dostupné z: https://isdv.upv.gov.cz/webapp/!resdb.oza.frm.
75. *CZ.NIC – Záznamy pro Trackee* [online]. [cit. 2024-04-24]. Dostupné z: https://www.nic.cz/whois/object/trackee/.
76. *Apple Developer – Human Interface Guidelines – Tab Bars* [online]. [cit. 2024-04-24]. Dostupné z: https://developer.apple.com/design/human-interface-guidelines/tab-bars.
77. *Apple Developer – App Intents* [online]. [cit. 2024-04-25]. Dostupné z: https://developer.apple.com/documentation/appintents.
78. *Ktor* [online]. [cit. 2024-04-25]. Dostupné z: https://ktor.io.
79. *Android Developers – Build better apps faster with Jetpack Compose* [online]. [cit. 2024-04-25]. Dostupné z: https://developer.android.com/develop/ui/compose.

80. *Compose Multiplatform* [online]. [cit. 2024-04-25]. Dostupné z: https://www.jetbrains.com/lp/compose-multiplatform/.
81. *Matee Devs – Devstack Native App Template* [online]. [cit. 2024-04-25]. Dostupné z: https://github.com/MateeDevs/devstack-native-app.
82. *Kotlin Multiplatform Wizard* [online]. [cit. 2024-04-25]. Dostupné z: https://kmp.jetbrains.com.
83. *Swift Package Manager* [online]. [cit. 2024-04-25]. Dostupné z: https://www.swift.org/documentation/package-manager/.
84. *Choose a Database: Cloud Firestore or Realtime Database* [online]. [cit. 2024-04-25]. Dostupné z: https://firebase.google.com/docs/database/rtdb-vs-firestore.
85. *IntelliJ IDEA: The Leading Java and Kotlin IDE* [online]. [cit. 2024-04-26]. Dostupné z: https://www.jetbrains.com/idea/.
86. *Git source code management* [online]. [cit. 2024-04-26]. Dostupné z: https://git-scm.com.
87. DESK, TOI Tech. *iOS 17 is being used by less number of iPhone users than iOS 16* [online]. TIMESOFINDIA.COM, 2024 [cit. 2024-04-26]. Dostupné z: https://timesofindia.indiatimes.com/gadgets-news/ios-17-adoption-rate-how-it-compares-to-ios-16/articleshow/107448477.cms#.
88. *GitHub* [online]. [cit. 2024-04-26]. Dostupné z: https://github.com.
89. *GitHub – Trackee App* [online]. [cit. 2024-05-07]. Dostupné z: https://github.com/tomas-bat/trackee-app.
90. *Railway – Instant Deployments, Effortless Scale* [online]. [cit. 2024-04-26]. Dostupné z: https://railway.app.
91. *Railway Docs – App Sleeping* [online]. [cit. 2024-04-26]. Dostupné z: https://docs.railway.app/reference/app-sleeping.
92. *Testflight* [online]. [cit. 2024-05-02]. Dostupné z: https://developer.apple.com/testflight/.
93. *App Store Connect* [online]. [cit. 2024-05-02]. Dostupné z: https://developer.apple.com/app-store-connect/.
94. *Gradle Build Tool* [online]. [cit. 2024-04-26]. Dostupné z: https://gradle.org.
95. *Apple Developer – Creating a multiplatform binary framework bundle* [online]. [cit. 2024-04-26]. Dostupné z: https://developer.apple.com/documentation/xcode/creating-a-multi-platform-binary-framework-bundle.
96. *What is a REST API?* [online]. IBM [cit. 2024-04-29]. Dostupné z: https://www.ibm.com/topics/rest-apis.
97. *Swagger – OpenAPI Specification* [online]. [cit. 2024-04-29]. Dostupné z: https://swagger.io/resources/open-api/.
98. *Twine* [online]. [cit. 2024-04-26]. Dostupné z: https://github.com/scelis/twine.
99. *SwiftGen* [online]. [cit. 2024-04-26]. Dostupné z: https://github.com/SwiftGen/SwiftGen.
100. *Swift Spyable* [online]. [cit. 2024-04-26]. Dostupné z: https://github.com/Matejkob/swift-spyable.
101. *Factory* [online]. [cit. 2024-04-26]. Dostupné z: https://github.com/hmlongco/Factory.
102. *Koin – The pragmatic Kotlin & Kotlin Multiplatform Dependency Injection framework* [online]. [cit. 2024-04-26]. Dostupné z: https://insert-koin.io.

103. *OAuth Access Tokens* [online]. [cit. 2024-04-26]. Dostupné z: https://oauth.net/2/access-tokens/.
104. *GitHub – Firebase iOS SDK* [online]. [cit. 2024-04-27]. Dostupné z: https://github.com/firebase/firebase-ios-sdk.
105. *GitHub – Firebase Android SDK* [online]. [cit. 2024-04-27]. Dostupné z: https://github.com/firebase/firebase-android-sdk.
106. *GitHub – Firebase Kotlin SDK* [online]. [cit. 2024-04-27]. Dostupné z: https://github.com/GitLiveApp/firebase-kotlin-sdk.
107. *Idempotence* [online]. [cit. 2024-04-28]. Dostupné z: https://cs.wikipedia.org/wiki/Idempotence.
108. *XCTest* [online]. [cit. 2024-04-22]. Dostupné z: https://developer.apple.com/documentation/xctest.
109. BATĚK, Tomáš. *Trackee user testing* [online]. Youtube [cit. 2024-04-28]. Dostupné z: https://www.youtube.com/playlist?list=PLqLRMhjBigswz_vDduZD6tdLs9jkb2C6n.
110. *KotlinX – Coroutines – Flow* [online]. [cit. 2024-05-02]. Dostupné z: https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/.
111. *SQLDelight* [online]. [cit. 2024-05-02]. Dostupné z: https://cashapp.github.io/sqldelight/2.0.2/.
112. *Google Analytics for Firebase* [online]. [cit. 2024-05-03]. Dostupné z: https://firebase.google.com/docs/analytics.
113. *Firebase Crashlytics* [online]. [cit. 2024-05-03]. Dostupné z: https://firebase.google.com/docs/crashlytics.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `app` ..... adresář se zkompilovanou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF