# Multiplatformní aplikace pro komunikaci mezi uživateli v reálném čase

Daniel Gabzdyl

Bakalářská práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

| | |
|---|---|
| Jméno a příjmení: | **Daniel Gabzdyl** |
| Osobní číslo: | **A18039** |
| Studijní program: | **B3902 Inženýrská informatika** |
| Studijní obor: | **Softwarové inženýrství** |
| Forma studia: | **Prezenční** |
| Téma práce: | **Multiplatformní aplikace pro komunikaci mezi uživateli v reálném čase** |
| Téma práce anglicky: | **Multiplatform Application for Real-time Communication Between Users** |

## Zásady pro vypracování

1. Prostudujte síťový protokol MQTT a jeho možnosti využití.
2. Seznamte se s možnostmi vývoje aplikací pro platformy společnosti Apple.
3. Navrhněte klientskou část aplikace pro real-time komunikaci prostřednictvím protokolu MQTT. Zaměřte se i na návrh uživatelského rozhraní.
4. Implementujte klientskou část aplikace pomocí vybrané technologie.
5. Prezentujte dosažené výsledky a diskutujte možné pokračování práce.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:


1. SwiftUI. Apple Developer [online]. [cit. 2021-12-02]. Dostupné z: https://developer.apple.com/documentation/swiftui
2. Human Interface Guidelines. Apple Developer [online]. [cit. 2021-12-02]. Dostupné z: https://developer.apple.com/design/human-interface-guidelines/
3. MQTT Version 5.0: OASIS Standard. OASIS [online]. 2019-03-07 [cit. 2021-12-02]. Dostupné z: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html
4. KNIGHTLY, Ryan. MQTT In iOS: The why and the how. Medium: The Floating Point [online]. 2017-01-26 [cit. 2021-12-02]. Dostupné z: https://medium.com/thefloatingpoint/mqtt-in-ios-d8574b55e006
5. BALKAYA, Can. Building Cross-Platform Apps With SwiftUI: Let’s build a mobile and desktop app using a single codebase. BetterProgramming [online]. 2020-12-17 [cit. 2021-12-02]. Dostupné z: https://betterprogramming.pub/building-cross-platform-apps-with-swiftui-3fea88cdb0ae
6. GONZALEZ GARCIA, Cristian. Protocols and applications for the industrial internet of things[online]. Hershey, Pennsylvania [cit. 2021-12-02]. ISBN 9781522538066.


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

Datum zadání bakalářské práce: **3. prosince 2021**
Termín odevzdání bakalářské práce: **23. května 2022**

**doc. Mgr. Milan Adámek, Ph.D.** v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

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

Ve Zlíně dne 24. ledna 2022

**Prohlašuji, že**

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářské práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 20. 5. 2022 Daniel Gabzdyl, v. r.

podpis studenta

**ABSTRAKT**

Cílem teoretické části této bakalářské práce je studium a popis funkce protokolu MQTT a seznámení se s principem vývoje multiplatformních aplikací pro platformy společnosti Apple. V praktické části je to návrh a implementace aplikace pro komunikaci v reálném čase pomocí vybrané technologie a knihovny pro práci s protokolem MQTT, a to konkrétně pro platformy iPhone, iPad a Mac. Vedlejším cílem je také návrh uživatelského rozhraní a jeho implementace pomocí SwiftUI.

Klíčová slova: iOS, iPadOS, macOS, SwiftUI, Swift, Xcode, Adobe XD, iPhone, iPad, Mac, MQTT, broker

**ABSTRACT**

The goal of theoretical part of this bachelor thesis is to study and describe the functions of the MQTT protocol and to get acquainted with the principle of developing multiplatform applications for Apple platforms. In the practical part, it is to design and implement an application for real-time communication using selected technology and a library for working with the MQTT protocol, specifically for the iPhone, iPad and Mac platforms. A secondary goal is also the design of the user interface and its implementation SwiftUI.

Keywords: iOS, iPadOS, macOS, SwiftUI, Swift, Xcode, Adobe XD, iPhone, iPad, Mac, MQTT, broker

Rád bych poděkoval panu Ing. Radku Valovi Ph.D. za vedení této práce, cenné rady a vstřícný přístup při konzultacích.

## OBSAH

# ÚVOD

Cílem této práce je seznámit čtenáře s velmi populárním komunikačním protokolem, který nese název MQTT a je hojně využíván např. v IoT (internet věcí). Může sloužit jako komunikační protokol mezi zařízeními chytré domácnosti a mobilní aplikací nebo pro zobrazování dat v reálném čase ve webové aplikaci. Využití je spousta.

Čtenář bude dále seznámen s možnostmi vývoje multiplatformních aplikací pro platformy společnosti Apple, kterými jsou iOS, iPadOS, macOS, watchOS a tvOS. V této kapitole bude také praktická ukázka demonstrující rozdíl mezi imperativním frameworkem UIKit a deklarativním frameworkem SwiftUI.

V poslední kapitole teoretické části bude stručně popsán architektonický vzor MVVM, aby byl čtenář obeznámen, protože bude použit v implementační části praktické části této bakalářské práce.

V praktické části bude představen návrh uživatelského rozhraní klientské části aplikace, a to nejprve pomocí drátěných modelů a v pozdější části i pomocí programu Adobe XD. Bude zde detailně popsána implementace jednotlivých částí včetně rozjetí projektu pomocí Xcode a instalace balíčků pomocí CocoaPods. Popis bude implementace komponent bude vždy rozdělen do podkapitol ViewModel a View.

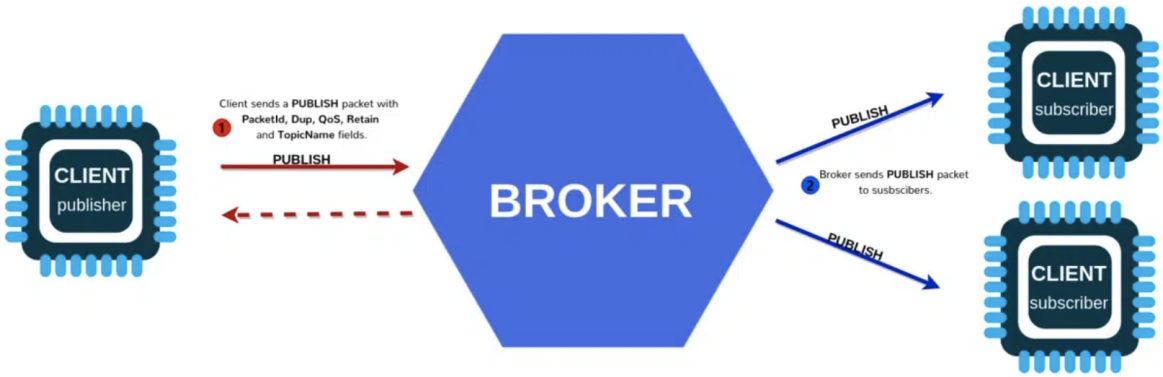
V poslední kapitole praktické části bude prezentována výsledná funkcionalita a podoba implementované aplikace a budou zmíněny její nedostatky a možnosti potenciálního rozvoje.

# I. TEORETICKÁ ČÁST

# 1 MQTT

Message Queue Telemetry Transport je jednoduchý a nenáročný komunikační protokol postavený nad TCP/IP navržený pro úzkou šířku pásma. Jedná se o tzv. publish/subscribe a je určený pro pro telemetrii s označením M2M, tj. zkratka pro Machine to Machine [1]. Pyšní se extrémně jednoduchou implementací v klientské části a exceluje v přenosu dat v porovnání s protokolem jako je HTTP. [2]

Funguje na bázi výměny zpráv, a to mezi klienty prostřednictvím centrálního bodu, nazývaného jako broker. Centrální bod přijímá zprávy od poskytovatelů (publisherů) a ty následně distribuuje příslušným příjemcům (subscriberům). Zprávy jsou poskytovány v rámci témat (topiců), které příjemci odebírají. Příjemci mohou odebírat zpráv z více témat a taktéž můžou být zároveň i poskytovatelé. Klientem může být například webová služba, čidlo nebo mobilní aplikace. Poskytovatel (publisher) tedy může být třeba senzor pokojové teploty, který vysílá naměřené hodnoty brokeru (centrální jednotce) a příjemce (subcriber), kterým může být mobilní aplikace, data přijímá a zobrazuje uživateli. [3]




Obrázek 1.1 Diagram toku dat pomocí MQTT [4]

## 1.1 Témata

Jednotlivé zprávy jsou tříděny do témat a každá zpráva patří právě do jednoho tématu. Témata si definuje samotný poskytovatel, a proto je nutné, aby příjemce předem znal název témat, které chce odebírat. Příjemce nemusí znát žádné informace ohledně poskytovatele, stačí pouze adresa brokeru. Názvy témat jsou hierarchicky odděleny lomítkem. Například „místnost pro zprávy s id 2“ může mít téma „b2a/messenger/chat-room/2“. Témata jsou reprezentována řetězcem s kódováním UTF-8, můžou tedy obsahovat i diakritiku. [5]

## 1.2 Zprávy

Obsah zprávy jsou binární data, která nejsou nijak specificky definována. Nejčastěji se setkáme s formátem JSON, BSON a textovým řetězcem. Ale protože broker data pouze přeposílá, můžeme využít libovolný formát, který klient bude umět zpracovat.

Aktuální verze protokolu MQTT omezuje velikost zprávy na necelých 256 MB. Zprávy zpravidla ale bývají mnohem menší. Minimalizuje se tak množství přenášených dat a hodí se na přenos občasných, méně náročných informací a hodnot, což je klíčové pro IoT (Internet of Things).

Protokol MQTT nedefinuje způsob přenosu. Využívá běžný protokol TCP/IP, má jednoduchou strukturu a využívá běžné ethernetové rozhraní. Tvoří tak pouze aplikační vrstvu referenčního modelu ISO/OSI. [5]

## 1.3 Kvalita služeb

Kvalita služeb je dohoda mezi poskytovatelem a příjemcem zprávy, která definuje garanci doručení pro konkrétní zprávu.

Existují tři úrovně:

- maximálně jednou (QoS 0),
- alespoň jednou (QoS 1),
- přesně jednou (QoS 2).

Ohledně kvality služeb MQTT musíme brát v úvahu dvě strany doručený zpráv:

1. doručení zprávy od poskytovatele služeb centrálnímu bodu (brokeru),
2. doručení zprávy od brokera příjemci.

Klient poskytující zprávu brokeru definuje úroveň kvality. Broker tuto zprávu vysílá s úrovní, na které se poskytovatel i příjemce „domluvili“. Pokud příjemce definuje nižší úroveň kvality než poskytovatel, broker vysílá zprávu s touto nižší úrovní kvality.

Kvalita služeb je pro MQTT protokol klíčová. Umožňuje klientu vybrat takovou úroveň, která odpovídá spolehlivosti internetového připojení a aplikační logice. MQTT řídí opětovný přenos zpráv a garantuje doručení, kvalita služeb dělá komunikaci v nespolehlivých sítích mnohem jednodušší. [6]

### 1.3.1 QoS 0 - maximálně jednou

Nejnižší úrovní je nula. Tato úroveň zaručuje dodání s maximálním úsilím. Není zde však žádná garance doručení. Příjemce nepotvrzuje přijetí zprávy a zpráva není uchována a opětovně vysílána. Úroveň nula poskytuje stejnou záruku jako základní protokol TCP. [6]

Využití:

- při kompletně, nebo alespoň obvykle stabilním připojení mezi odesílatelem a příjemcem (např. kabelové připojení mezi klientem a centrálním bodem) [7]

### 1.3.2 QoS 1 - alespoň jednou

Úroveň jedna zaručuje, že zpráva je příjemci doručena alespoň jednou. Poskytovatel zprávu uchovává do doby, než obdrží informaci o přijetí příjemcem (PUBACK paket). Může se stát, že zpráva je odeslána nebo doručena vícekrát. Poskytovatel využívá identifikátor v každém paketu, aby byl schopný spojit PUBLISH paket s odpovídajícím PUBACK paketem. Pokud je příjemcem centrální bod (broker), odešle zprávu všem poslouchajícím příjemcům a odpoví PUBACK paketem. [6]

Využití:

- je potřeba doručit každou zprávu a je možné pracovat s duplikáty [7]

### 1.3.3 QoS 2 - přesně jednou

Úroveň dva je nejvyšší možnou úrovní kvality služeb. Zaručuje, že každá zpráva je doručena jen jednou každému z plánovaných příjemců. Je nejbezpečnější, ale zároveň i nejpomalejší. Záruka je poskytována minimálně dvěma tzv. request/response toky mezi odesílatelem a příjemcem. Odesílatel i příjemce používají identifikátor paketů z původní zprávy ke koordinaci doručení zprávy. Jakmile příjemce obdrží PUBLISH paket od odesílatele, zprávu zpracuje odpovídajícím způsobem a odesílateli odpoví paketem PUBREC, která potvrzuje paket PUBLISH. Pokud odesílatel tento paket neobdrží, odešle znovu paket PUBLISH s příznakem duplikace, dokud neobdrží potvrzení.

Po přijetí paketu PUBREC může odesílatel bezpečně zahodit původní PUBLISH paket. Odesílatel uchová paket PUBREC a odpoví paketem PUBREL.

V momentě, kdy příjemce obdrží paket PUBREL, může zahodit všechen uchovaný stav a odpoví paketem PUBCOMP. Před odesláním PUBCOMP si příjemce uchovává referenci na identifikátor původního PUBLISH paketu, což zajistí to, aby stejnou zprávu nezpracoval znovu. Když odesílatel obdrží paket PUBCOMP, identifikátor se stává znovu dostupným.

Pokud se paket po cestě ztratí, odesílatel je odpovědný za znovu odeslání zprávy v co nejkratším možném čase. Příjemce je odpovědný příslušně odpovědět na každý konkrétní paket. [6]

Využití:

- je extrémně důležité, aby každá zpráva byla doručena přesně jednou (např. když duplikáty můžou způsobit problémy uživatelům aplikace nebo příjemcům) [7]

# 2 VÝVOJ APLIKACÍ PRO PLATFORMY SPOLEČNOSTI APPLE

Aplikace pro Apple platformy, kterými jsou iOS, macOS, tvOS a watchOS je možné vyvíjet vícero způsoby. Aplikace pro iPhone a iPad a Apple TV se vyvíjí v UIKitu. [8] Aplikace pro Mac se vyvíjí v AppKitu [9] a aplikace pro Apple Watch ve WatchKitu [10].

Pokud však chceme vyvíjet aplikaci, která bude spustitelná na více platformách a bude z velké většiny sdílet zdrojový kód, máme dvě možnosti.

## 2.1 Mac Catalyst

Než Apple představil Mac Catalyst, bylo převádění aplikací vyvíjených pro iPhone a iPad v UIKitu na Mac časově náročné. Macbooky a iMacy mají větší obrazovky a liší se od mobilních zařízení ve vstupních metodách. Bylo tedy nutné využít jinou sadu prvků. UIKit obsahuje třídy a kontrolery vyvinuty pro dotykové ovládání a pracuje s iOS SDK, zatímco AppKit obsahuje veškeré prvky pro počítačové rozhraní a pracuje s macOS SDK. [11]

Aplikace vyvíjené v Mac Catalyst můžou sdílet zdrojový kód s aplikací psanou pro iPad v UIKitu a nabízí možnost přidat charakteristické funkcionality určené pro Mac. V praxi to znamená, že můžeme využívat např. okna a dotykové ovládání je adaptováno na klávesnici a myš. Aplikace bude ze začátku sdílet rozměry aplikace pro iPad, ale máme možnost rozměry optimalizovat individuálně i pro Mac. [12]

## 2.2 SwiftUI

SwiftUI je novým frameworkem z dílny společnosti Apple, ve kterém se UI narozdíl od UIKitu píše deklarativně. To v praxi znamená, že v kódu popisujeme co chceme zobrazit a už nás nezajímá, jak toho na pozadí docílit. To za nás řeší právě SwiftUI. Toto řeší i např. problém se stavem aplikace, který vývojáře trápil zejména při vývoji v UIKitu, kde se UI vyvíjí imperativně. Vhodným příkladem by byla třeba přihlašovací obrazovka. V imperativním vývoji bychom museli přesně naprogramovat, že tlačítko po kliknutí zavolá konkrétní funkci (navíc s nedobrovolným využitím anotace @objc a zvolením typu eventu touchUpInside) a po změně stavu (přihlášení) ručně napsat, že se má zobrazit jiný ViewController. Deklarativní SwiftUI dokáže při změně stavu reagovat automaticky a vývojářům tak výrazně zjednodušit práci.

Navíc se jedná o multiplatformní framework, kde jedna aplikace s téměř stoprocentně sdíleným kódem vypadá dobře na všech Apple platformách a kódu je ve finále mnohem méně, je čitelnější a lépe udržitelný. [13]

SwiftUI také přichází s tzv. live preview, kdy konečně po každé změně v UI nemusíme nutně znovu a znovu kompilovat aplikaci a spouštět simulátor. Live preview je plně synchronizován s kódem a tudíž se po každé změně live preview přenačte a zobrazí změnu v UI téměř okamžitě. Zajímavou součástí je i drag and drop, kdy můžeme jednotlivé komponenty přetahovat přímo do canvasu a ty se automaticky v kódu promítnou. Navíc můžeme mít více previews najednou a zobrazit tak všechny koncové zařízení, a to i v landscape režimu nebo ve tmavém režimu. [14]

### 2.2.1 Layout systém

SwiftUI přináší úplně nový layout systém, díky kterému jsou komponenty dynamičtější a na každém zařízení se správně roztáhnou. To platí také např. pro paddingy, které se umí přizpůsobit pozici v hierarichii, kde jsou zobrazeny a nejsou tak fixně dané. Podobně fungují i barvy, kdy se konkrétní barva umí adaptovat na platformu, na které je aplikace spuštěna.

Musíme tedy dbát na to, že komponenty (View) by neměly být vázané na konkrétní prostředí, a to např. pevnou šířkou a výškou. Narozdíl podobně navrženému AutoLayoutu však nemusíme ručně psát constrainty (přichycení prvku k ostatním prvkům), hierarchie ve SwiftUI si to rozhodne sama dle kontextu. [15]

### 2.2.2 Deklarativní vs imperativní UI

V této podkapitole bude pro představu stručně demonstrován rozdíl mezi deklarativním a imperativním UI.

První demonstrace proběhne na vytvoření labelu. Nebude demonstrováno přidání labelu v UIKitu do view.

Imperativně (UIKit):

```
let label = UILabel()
label.text = "Toto je demonstrační label"
label.backgroundColor = .red
label.textColor = .white
```

Deklarativně (SwiftUI):

```
Text("Toto je demonstrační label")
    .background(.red)
    .foregroundColor(.white)
```

Druhá demonstrace proběhne na přidání tlačítka a obsluhu klepnutí, která zvýší hodnotu proměnné o jedna.

Imperativně (UIKit):

```
let button = UIButton()
button.setTitle("Zvyš o 1", for: .normal)
button.setTitleColor(.blue, for: .normal)
button.addTarget(self, action: #selector(didTapButton(_:)), for: .touchUpInside)
@objc func didTapButton(_ button: UIButton) {
    counter += 1
}
```

Deklarativně (SwiftUI):

```
Button {
    counter += 1
} label: {
    Text("Zvyš o 1")
         .foregroundColor(.blue)
}
```

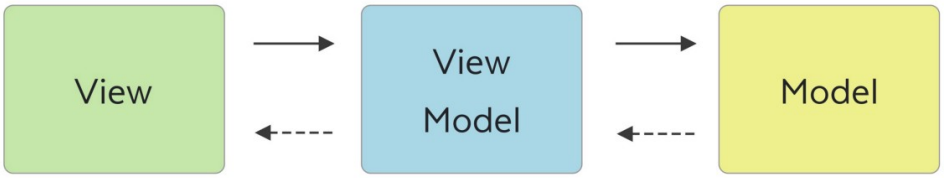
# 3 ARCHITEKTONICKÝ VZOR MVVM

V této kapitole bude stručně popsán architektonický vzor MVVM, protože bude využit v implementační kapitole praktické části této bakalářské práce.

MVVM, celým názvem Model-View-ViewModel, je architektonickým vzorem pro vývoj aplikací. Slouží především k oddělení dat, stavu aplikace a aplikační logiky od uživatelského rozhraní. [16]

Model reprezentuje data a veškerou logiku, se kterými aplikace pracuje. View je uživatelské rozhraní, typicky šablona. ViewModel obě vrstvy spojuje. Připravuje data z modelu k zobrazení v uživatelském rozhraní a taktéž na změny v uživatelském rozhraní reaguje. [17]

Mezi View a ViewModelem je binding, který zaručuje propsání změny v uživatelském rozhraní do ViewModelu a naopak. [18]




Obrázek 3.1 Jednoduchý MVVM diagram [19]

# II. PRAKTICKÁ ČÁST

# 4 NÁVRH KLIENTSKÉ ČÁSTI APLIKACE

Aplikace bude vyvinuta ve frameworku SwiftUI společnosti Apple a bude se věnovat problematice odesílání zpráv mezi uživateli v reálném čase.

## 4.1 Funkční požadavky

Funkčními požadavky rozumíme požadavky, které specifikují konkrétní funkcionality aplikace, které může uživatel případně ovlivnit.

Tabulka 4.1 Základní funkční požadavky

| ID | Požadavek |
|---|---|
| FP1 | Aplikace bude umožňovat přihlášení zaměstnaneckým číslem a pinem. |
| FP2 | Aplikace nebude po opětovném otevření aplikace vyžadovat přihlášení. |
| FP3 | Aplikace bude zobrazovat seznam místností. |
| FP4 | Aplikace bude umožňovat proklik na detail. |
| FP5 | Aplikace bude na detailní obrazovce v reálném čase přijímat zprávy a zobrazovat je uživateli. |
| FP6 | Aplikace umožní na detailní obrazovce uživateli zadat zprávu. |
| FP7 | Aplikace po odeslání zprávy odešle zprávu pomocí protokolu MQTT centrálnímu bodu (brokeru). |

## 4.2 Nefunkční požadavky

Nefunkčními požadavky rozumíme požadavky, které uživatel nemůže přímo ovlivnit a specifikují způsob vývoje aplikace a její vlastnosti.

Tabulka 4.2 Základní nefunkční požadavky

| ID | Požadavek |
|---|---|
| NP1 | Aplikace bude vyvíjena pro v nejnovější verze operačních systémů iOS 15 a macOS 12. |
| NP2 | Programovacím jazykem byl zvolen jazyk Swift. Objective-C již Apple nepropaguje a noví vývojáři už pracují převážně s programovacím jazykem Swift. |
| NP3 | Aplikace bude psána jako multiplatformní ve frameworku SwiftUI a jako cílové zařízení byl zvolen iPhone, iPad a Mac. |
| NP4 | Aplikace bude inspirována Human Interface Guidelines od společnosti Apple a bude využívat převážně nativní komponenty, které framework SwiftUI nabízí. |
| NP5 | Aplikace bude v tuto chvíli lokalizována pouze do českého jazyka. |
| NP6 | Aplikace bude jako zdroj dat využívat již existující serverové řešení. |
| NP7 | Aplikace bude využívat struktury, které již existují v serverovém řešení. |

## 4.3 Popis obrazovek

Aplikace se bude skládat ze tří obrazovek.

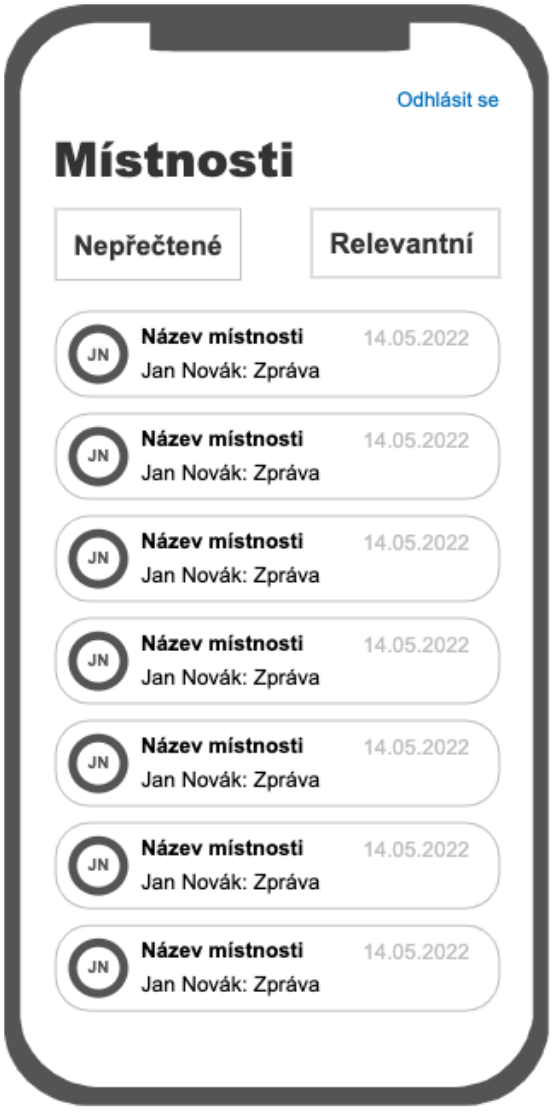
### 4.3.1 Obrazovka pro přihlášení do systému

Pokud uživatel po spuštění aplikace nebyl v minulosti přihlášen, zobrazí se přihlašovací obrazovka. Obrazovka obsahuje dvě pole, a to pole pro zadání osobního čísla a pole pro zadání pinu. Dále obrazovka obsahuje tlačítko, kde po klepnutí na něj dochází k dotazu na server a v případě korektních údajů zobrazí obrazovku se seznamem místností.


Obrázek 4.1 Drátěný model přihlašovací obrazovky

### 4.3.2 Obrazovka se seznamem místností

Obrazovka bude načítat data ze serveru a zobrazovat přívětivě vypadající seznam místností, kde každá položka bude obsahovat iniciály autora poslední zprávy jako avatar, název místnosti, část textu a datum poslední zprávy. V horní části se bude nacházet lišta s filtry, kde uživatel bude moci zaškrtnout, zda chce zobrazit pouze relevantní místnosti, nebo pouze nepřečtené místnosti. Tyto filtry lze kombinovat. Po klepnutí na položku seznamu je uživatel přesměrovaný na detailní obrazovku místnosti.




Obrázek 4.2 Drátěný model obrazovky se seznamem místností

### 4.3.3 Detailní obrazovka místnosti

Detailní obrazovka bude zobrazovat všechny zprávy, které kdy v místnosti byly poslány. Stránkování nebude implementováno. Po otevření se automaticky zobrazí poslední zpráva. Zprávy budou zobrazeny seskupeně dle dne odeslání, aby bylo možné zobrazit hlavičku pro každou sekci (den) a uživatel tak nebyl ztracen, kde se nachází. Vzhledově se zpráva bude podobat položce seznamu na předchozí obrazovce. Ve spodní části se bude nacházet pole pro zadání textové zprávy a tlačítko pro odeslání.


Obrázek 4.3 Drátěný model detailní obrazovky

### 4.4 Návrh UI

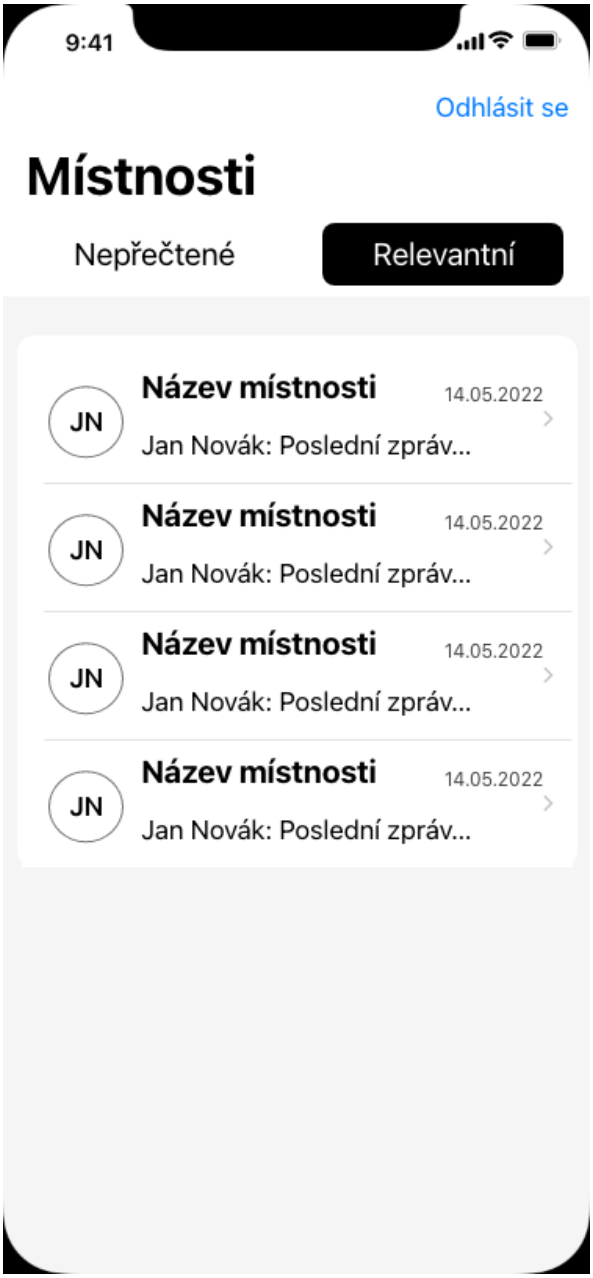
Vzhled aplikace byl navržen v programu XD od společnosti Adobe. Pro návrh byly využity komponenty přímo společnosti Apple pro iOS 15 a iPadOS 15 ze stránky Apple Design Resources - https://developer.apple.com/design/resources/. Některé z komponent navrhl sám autor.

Návrh UI byl vytvořen pro platformu iOS a částečně ipadOS. Většina navrženého UI bude sdílena mezi platformami. Rozhraní aplikace na macOS je velice podobné tomu na iPadOS v landscape režimu.

### 4.4.1 Obrazovka pro přihlášení do systému

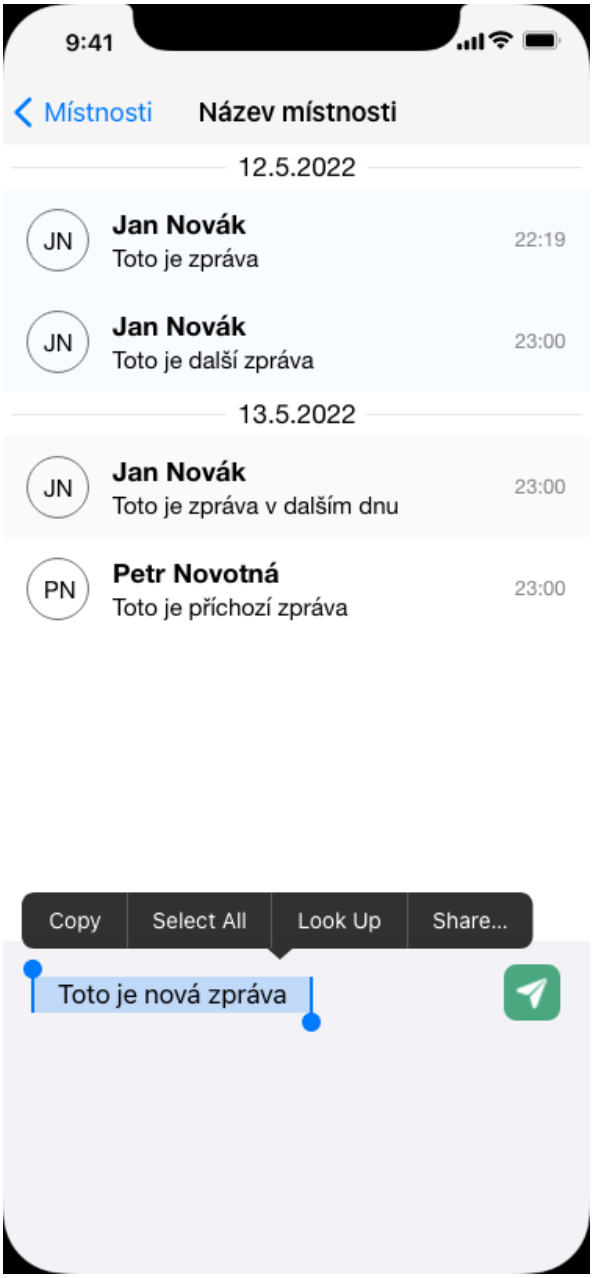

Obrázek 4.4 Návrh UI přihlašovací obrazovky

### 4.4.2 Obrazovka se seznamem místností

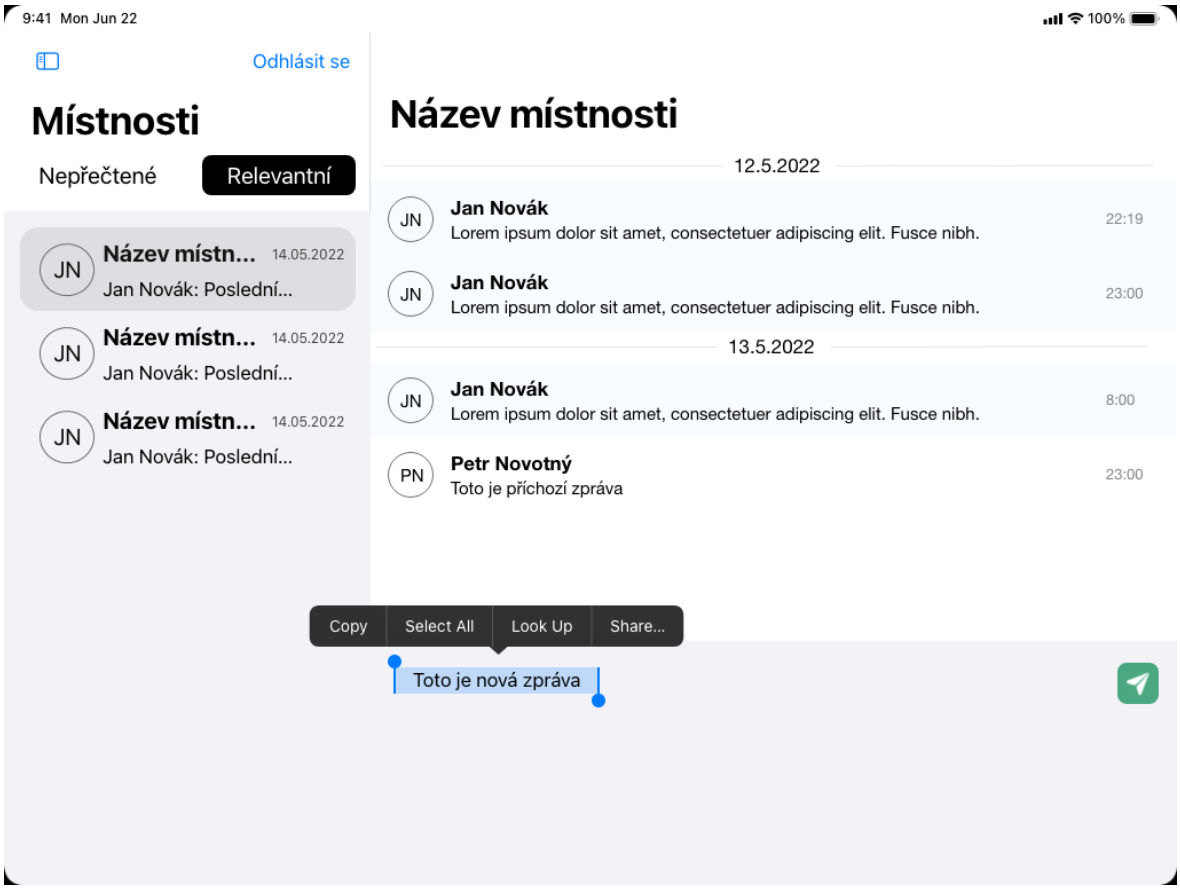



Obrázek 4.5 Návrh UI obrazovky se seznamem místností

### 4.4.3 Detailní obrazovka místnosti




Obrázek 4.6 Návrh UI detailní obrazovky místnosti




Obrázek 4.7 Návrh UI obrazovky se seznamem místností a detailem pro iPad (landscape režim)

# 5 IMPLEMENTACE APLIKACE

V této kapitole bude popsána implementace multiplatformní aplikace ve SwiftUI s knihovnou CocoaMQTT - pro práci s MQTT a Alamofire - pro práci s HTTP. Aplikace bude implementována dle architektonického vzoru MVVM (Model-View-ViewModel).

## 5.1 Vytvoření projektu

Vytvoření multiplatformní aplikace ve SwiftUI bylo díky rozhraní a možnostem Xcode opravdu snadné. Na obrazovce se šablonami byla zvolena kategorie Multiplatform a vybrána šablona App ve skupině Application. Jako název aplikace byl zvolen MultiplatformMessenger.

## 5.2 Instalace potřebných knihoven

Pro práci s MQTT byla vybrána jedna z nejpoužívanějších knihoven svého druhu - CocoaMQTT. Pro usnadnění práce s posíláním dotazů pomocí HTTP a následným mapováním vrácených dat byly vybrány knihovny Alamofire a ObjectMapper.

Instalace knihoven byla provedena pomocí správce balíčků CocoaPods, který bylo potřeba mít v počítači nainstalovaný.

Inicializace CocoaPods byla spuštěna příkazem `pod init` ve složce s projektem. V nově vzniklém souboru Podfile byly uvedeny vybrané knihovny pro všechny schémata - v případě této práce se jedná o iOS a macOS.

Spuštěním příkazu `pod install` byly knihovny nainstalovány a připraveny tak k použití v aplikaci.

### 5.3 API objekty

Aby bylo možné pracovat s daty, které server vrací, byly pomocí firemního nástroje vygenerovány API objekty, které data reprezentují. Dále byly přidány protokoly a extensions, které jsou pro práci s těmito objekty nezbytně nutné.

Každý API objekt implementuje protokol ApiObject, který definuje několik funkcí, které lze využít. V extension ApiObject je pak základní implementace těchto funkcí. Za zmínku stojí např. `toJSON(includeNilValues: Bool)`, která objekt převede do formátu JSON. Protokol ApiObject dále implementuje protokol Mappable z knihovny ObjectMapper. Ten nařizuje implementovat metodu `mapping(map: Map)`, kterou knihovna využívá při serializaci a deserializaci. V metodě mappable uvádíme konkrétně jaká JSON vlastnost se mapuje na kterou vlastnost a případně jaký TransformType pro mapování dané property použít. To je potřeba zpravidla v případě složitějších datových typů než jsou např. primitivní datové typy jako Int, String, Bool apod.

Důležitou vlastností každého API objektu vlastnost @type (mapováno na API_TYPE), která přesně definuje název objektu pro každou z platforem. Díky přítomnosti této vlastnosti a pomocných metod protokolu ApiObject je možné mapovat API objekty i bez nutnosti implementovat protokol Codable.

Ukázka API objektu:

```
open class B2aDicoMention: ApiObject {
    open var API_TYPE: [String: String] = [:]
    open var expandable: [String: Bool] = [:]
    open var id: Int?
    open var title: String!
    open var code: String!
    public required init?(map: Map) {}
    open func mapping(map: Map) {
        id <- map["id"]
        title <- map["title"]
        code <- map["code"]
        API_TYPE <- map["@type"]
        expandable <- map["@expandable"]
    }
    public init() {
        self.title = ""
        self.code = ""
        self.API_TYPE = [
            "php": "\\B2A\\DICO\\Mention\\Module\\Mention\\API\\Mention",
            "swift": "B2aDicoMention",
            "typescript": "B2aDicoMention"
        ]
        self.expandable = [:]
    }
}
```

### 5.4 Služba pro správu přihlášení uživatele

Pro práci s uživatelem byla implementována jednoduchá služba UserService, která umožňuje ukládání aktuálně přihlášeného uživatele do perzistentní paměti a také jeho získání a smazání. Služba byla implementována dle návrhového vzoru singleton, kterou sám Apple v jeho knihovnách hojně využívá a který nám zaručí jedinou instanci dané služby v aplikaci. Služba implementuje protokol ObservableObject, který umožňuje informovat ohledně změn @Published vlastností.

```
import SwiftUI
class UserService: ObservableObject {
    static let shared = UserService()
    private let userKey = "user"
    @Published public var loggedUser: LoggedUser? = nil
    public var isLogged: Bool {
        return loggedUser != nil
    }
    public func initUserFromDefaults() {
        guard let data = UserDefaults.standard.data(forKey: userKey) else {
            return
        }
        loggedUser = try? JSONDecoder().decode(LoggedUser.self, from: data)
    }
    public func saveToDefaults() {
        guard let user = loggedUser,
              let data = try? JSONEncoder().encode(user) else { return }
        UserDefaults.standard.set(data, forKey: userKey)
    }
    public func logout() {
        loggedUser = nil
        UserDefaults.standard.removeObject(forKey: userKey)
    }
}
```

Jakmile je aplikace spuštěna, aplikace musí zkusit načíst uživatele z perzistentní paměti. Kód, který metodu `initUserFromDefaults()` zavolá, byl umístěn do konstruktoru vstupního místa aplikace (struktura s anotací @main).

```
import SwiftUI
@main
struct AppMain: App {
    init() {
        UserService.shared.initUserFromDefaults()
    }
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

### 5.5 Služba pro dotazování API

Další nezbytnou částí je služba NetworkService, která implementuje metody pro posílání dotazů. Podobně jako UserService je implementována dle návrhového vzoru singleton. Obsahuje natvrdo definovanou adresu serveru. V našem případě, kdy nevyužíváme více schémat, nám to postačí. Dále obsahuje metody pro konstrukci HTTP hlavičky, do které vloží token právě přihlášeného uživatele a metodu pro konstrukci finální URL, aby bylo umožněno volat dotazovací metody bez nutnosti neustále přikládat adresu serveru.

Zjednodušená ukázka NetworkService:

```
import Foundation
import Alamofire
class NetworkService {
    static let shared = NetworkService()
    static let baseUrl = "https://backend-url.cz"
    open func prepareHeaders(_ headers: HTTPHeaders?) -> HTTPHeaders {
        var result = headers ?? [:]
        if let userApiKey = UserService.shared.loggedUser?.apiKey {
            result["Authorization"] = "Token token=\"\(userApiKey)\""
        }
        return result
    }
    open func prepareUrl(_ url: URLConvertible) -> URLConvertible {
        var result = url
        if let urlString = url as? String {
            result = "\(Self.baseUrl)\(urlString)"
        }
        return result
    }
    public func makeRequest(_ url: URLConvertible,
                            method: HTTPMethod = .get,
                            parameters: Parameters? = nil,
                            encoding: ParameterEncoding = URLEncoding.default,
                            headers: HTTPHeaders? = nil) -> DataRequest {
        return AF.request(prepareUrl(url),
                          method: method,
                          parameters: parameters,
                          headers: prepareHeaders(headers))
    }
}
```

### 5.6 Služba pro práci s MQTT

Pro práci s MQTT byla implementována MqttService dle návrhového vzoru singleton. Tato service obsahuje vlastnosti status, client a observers. Status ukládá stav připojení k centrálnímu bodu (brokeru). Client je typu CocoaMQTT, jenž je objektem z knihovny CocoaMQTT, který implementuje metody např. pro připojení, odpojení, odeslání zprávy atd. Observers je pole typu MqttObserver, což je protokol definující metodu `onMessageReceive(_ payloadString: String, topic: String)`. Tento protokol implementují ViewModely, aby bylo možné jim nově příchozí zprávy distribuovat.

Metoda `prepareMqttConnection()` se stará o vytvoření instance CocoaMQTT s parametry jako jsou clientId, host, port a konkrétními nastaveními.

```
func prepareMqttConnection() {
    let client = CocoaMQTT(clientID: "client-id",
                           host: "mqtt-host.cz",
                           port: 21317)
    client.keepAlive = 0
    client.delegate = self
    client.autoReconnect = true
    client.autoReconnectTimeInterval = 5
    client.enableSSL = true
    client.cleanSession = true
    self.client = client
}
```

Metoda `connect()` se pokusí navázat připojení, pokud již není připojeno a pokud není připojování v procesu.

```
func connect() {
    guard status != .connected, status != .connecting else { return }
    _ = client?.connect()
}
```

Metoda `registerObserver(_ observer: MqttObserver)` přidá do pole observers instanci implementující protokol MqttObserver.

```
func registerObserver(_ observer: MqttObserver) {
    observers.append(observer)
}
```

Metoda `subscribe(to topic: String)` začne přijímat zprávy na předaném tématu.

```
func subscribe(to topic: String) {
    client?.subscribe(topic)
}
```

Metoda `unsubscribe(from topic: String)` ukončí příjem zpráv na předaném tématu.

```
func unsubscribe(from topic: String) {
    client?.unsubscribe(topic)
}
```

Metoda `publish(string payload: String, topic: String, qos: CocoaMQTTQoS = .qos1, retain: Bool = false)` odešle zprávu centrálnímu bodu pro předané téma s možnosti určit parametry přenosu.

```
func publish(string payload: String,
             topic: String,
             qos: CocoaMQTTQoS = .qos1,
             retain: Bool = false) {
    client?.publish(topic, withString: payload, qos: qos, retained: retain)
}
```

MqttService implementuje protokol CocoaMQTTDelegate, který je nezbytné implementovat, aby bylo možné využít jeho metod životního cyklu MQTT připojení. Reálně v našem případě byly využity tři metody z tohoto protokolu.

Metoda `didConnectAck` nastaví vlastnost status na připojeno.

```
public func mqtt(_ mqtt: CocoaMQTT, didConnectAck ack: CocoaMQTTConnAck) {
    if ack == .accept {
        status = .connected
    }
}
```

Metoda `mqttDidDisconnect` nastaví vlastnost status na odpojeno.

```
public func mqttDidDisconnect(_ mqtt: CocoaMQTT, withError err: Error?) {
    status = .disconnected
}
```

Metoda `didReceiveMessage` přijímá zprávu a dále ji distribuuje všem instancím implementující protokol MqttObserver v poli observers.

```
public func mqtt(_ mqtt: CocoaMQTT,
                 didReceiveMessage message: CocoaMQTTMessage,
                 id: UInt16) {
    DispatchQueue.main.async {
        if let payload = message.string {
            for observer in self.observers {
                observer.onMessageReceive(payload, topic: message.topic)
            }
        }
    }
}
```

Obdobně jako u UserService bylo nezbytné zajistit, aby se inicializace a připojení provedlo v procesu spouštění aplikace.

```
import SwiftUI
@main
struct AppMain: App {
    init() {
        UserService.shared.initUserFromDefaults()
        MqttService.shared.prepareMqttConnection()
        MqttService.shared.connect()
    }
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

### 5.7 BaseView

Před implementací obrazovek samotných bylo implementováno jedno obecné View, které umí přijímat stav isLoading a dle hodnoty případně zobrazovat indikátor načítání. Pomocí ZStacku bylo možné docílit toho, aby obsah byl případně překryt a bylo tak zamezeno dalším akcím, než se např. provede dotaz.

```
import SwiftUI
struct BaseView<Content>: View where Content: View {
    var content: Content
    @Binding var isLoading: Bool
    public init(
        isLoading: Binding<Bool>,
        @ViewBuilder content: () -> Content
    ) {
        self._isLoading = isLoading
        self.content = content()
    }
    var body: some View {
        ZStack {
            content
            if isLoading {
                HStack {
                    Spacer()
                    LoadingView()
                    Spacer()
                }
            }
        }
    }
}
```

## 5.8 Přihlašovací obrazovka

V předchozích bodech byla předpřipravena většina prerekvizit, a proto bylo možné začít s vývojem první obrazovky.

### 5.8.1 ViewModel

Po vytvoření SwiftUI View s názvem LoginView byl vytvořen ViewModel, který má za úkol ukládat potřebný stav obrazovky a obsluhovat přihlašovací tlačítko.

V tomto projektu byly ViewModely jednotlivých obrazovek implementovány jako třídy spadající do daného View, tzn. všechny se nazývají ViewModel a je možné je instanciovat pouze v rámci daného View. ViewModel vždy implementuje protokol ObservableObject, který díky publisherovi informuje, že se nějaká z jeho @Published vlastností změnila.

ViewModel přihlašovací obrazovky obsahuje @Published vlastnosti personalNr, pin a isLoading, kde první dvě ukládají stav textových polí na obrazovce a isLoading ukládá stav obrazovky a je dále předáván do BaseView, které případně zobrazuje indikátor načítání.

Metoda `signIn()` provádí dotaz na API a při korektní odpovědi nastaví přihlášeného uživatele v UserService a zavolá metodu, která jej uloží do perzistentní paměti.

Ukázka metody `signIn()`:

```
func signIn() {
    isLoading = true
    NetworkService.shared
        .makeRequest("/auth/api/v1/employee/personal-nr-and-pin",
                     method: .post,
                     body: [
                         "personalNr": personalNr,
                         "pin": pin,
                         "includeMqttClient": true
                     ])
        .validate()
        .responseJSON { response in
            self.isLoading = false
            switch response.result {
                case .success(let json as [String: Any]):
                    guard let data = json["data"] as? [String: Any] else {
                        fallthrough
                    }
                    UserService.shared.loggedUser = LoggedUser(data: data)
                    UserService.shared.saveToDefaults()
                    break
               default:
                   break
            }
    }
}
```

### 5.8.2 View

View má vlastnost viewModel, ve které je uložena instance ViewModelu a byl použit property wrapper @StateObject. Ten je vhodný pro komplexnější datové typy, např. třídy a struktury, které implementují protokol ObservableObject. Použitím @StateObject dáváme SwiftUI jasně najevo, že má sledovat @Published vlastnosti objektu a případně přerenderovat View.

```
@StateObject private var viewModel = ViewModel()
```

Samotné View přihlašovací obrazovky se skládá z BaseView, které přebírá stav isLoading z ViewModelu. BaseView obaluje VStack, který slouží ke skládání Views nad sebe (vertikálně). Do VStacku byl přidán TextField jako pole pro osobní číslo, SecureField jako pole pro pin a Button jako tlačítko pro vyvolání přihlašovací akce.

Pole využívají vlastní modifikátor pro stylizaci jak vzhledovou, tak i pro zakázání autokorekce apod.

Tlačítko po klepnutí zavolá metodu `signIn()` ve ViewModelu.

```
var body: some View {
    BaseView(isLoading: $viewModel.isLoading) {
        VStack {
            TextField("Osobní číslo", text: $viewModel.personalNr)
                .textFieldStyle(LoginTextFieldStyle())
            SecureField("Pin", text: $viewModel.pin)
                .textFieldStyle(LoginTextFieldStyle())
            Button(action: {
                viewModel.signIn()
            }, label: {
                Text("Přihlásit se")
            })
        }
        .padding()
    }
}
```

Vlastní styl pro stylizaci textových polí:

```
struct LoginTextFieldStyle: TextFieldStyle {
    func _body(configuration: TextField<Self._Label>) -> some View {
        configuration
            .disableAutocorrection(true)
             #if os(iOS)
                .keyboardType(.numberPad)
                .textInputAutocapitalization(.never)
             #endif
            .padding()
            .background(Color.lightGray)
            .cornerRadius(5)
    }
}
```

## 5.9 Obrazovka se seznamem místností

Obrazovka obsahuje vlastní ViewModel, který stahuje data ze serveru a zobrazuje je v seznamu.

### 5.9.1 ViewModel

ViewModel obsahuje 4 @Published vlastnosti - chatrooms, isUnreadFilterOn, isRelevantFilterOn a isLoading. Dále obsahuje metodu `getChatrooms()`, která volá API s danými query parametry a výsledek (pole místností) následně ukládá do vlastnosti chatrooms.

K isUnreadFilterOn a isRelevantFilterOn je připojený tzv. property observer didSet, jenž pozoruje hodnotu vlastnosti a po její změně můžeme vykonat nějaký kód. Bylo implementováno, že po změně jakéhokoliv filtru je zavolána metoda `getChatrooms()` s aktualizovanými hodnotami.

Ukázka použití property observer didSet:

```
@Published var isUnreadFilterOn: Bool = false {
    didSet {
        getChatrooms()
    }
}
```

Toto je jedna z možností jak tohoto docílit. Dalo by se to vyřešit i reaktivně, ale toto řešení mi přišlo z hlediska jednoduchosti pro tuto práci vhodnější.

### 5.9.2 View

Obsahem je BaseView, které zaobaluje VStack, ve kterém se nachází lišta s filtry a List, kde každá položka (buňka) je zaobalena do NavigationLink. Prvním parametrem NavigationLink je odkaz, neboli View, na které bude navigace směřovat a druhým je label, tedy co bude zobrazeno.

Ukázka použití List a NavigationLink:

```
List {
    ForEach(viewModel.chatrooms, id: \.id) { chatroom in
        NavigationLink {
            ChatroomDetailView(chatroom: chatroom)
        } label: {
            ChatroomListCell(chatroom: chatroom,
                             lastMessage: chatroom.lastMessages.last)
        }
    }
}
```

Na stejné úrovni v hieararchii s BaseView je View Text s textem *Vyberte místnost*, které je zobrazeno v případě macOS a landscape režimu na iPadu v případě, že nebyla dosud vybrána žádná místnost (nebylo navigováno na detail).

Metoda `getChatrooms()` pro získání místností je volána v metodě `onAppear(perform:)` aplikované na BaseView, která se provádí po zobrazení View. Protože ViewModel implementuje ObservableObject a má @Published vlastnost chatrooms, SwiftUI automaticky přerenderuje a data se tak vykreslí.

```
.onAppear {
    viewModel.getChatrooms()
}
```

Na List byla aplikována metoda `refreshable(action:)`, díky které je možné spustit nativní funkcionalitu ke stažení listu směrem dolů (pull to refresh) a spustit obsluhující kód. Konkrétně se volá metoda `getChatrooms()` pro přenačtení seznamu místností.

```
.refreshable {
    viewModel.getChatrooms()
}
```

## 5.10 Obrazovka s detailem místnosti

Obrazovka obsahuje seznam zpráv rozdělených do skupin dle dne, kdy byly odeslány. Ve spodní části se nachází textové pole pro zadání zprávy a tlačítko pro odeslání. Obrazovka přijímá nově příchozí zprávy pro MQTT téma, které je pro danou místnost charakteristické a zprávy zobrazuje uživateli. Na nově příchozí zprávu je vždy sescrollováno, aby ji uživatel zaznamenal. Stránkování není implementováno a vždy jsou načteny všechny zprávy, které se v místnosti nacházejí.

### 5.10.1 ViewModel

ViewModel kromě ObservableObject implementuje také protokol MqttObserver a obsahuje 4 @Published vlastnosti - message, messages, groups a isLoading.

Messages je datového typu B2aDicoMessengerMessage, který dědí od základního typu ApiObject. Při uložení nové hodnoty do tohoto pole dojde k seskupení jednotlivých zpráv dle hodnoty timestamp a uložení seskupených zpráv do vlastnosti groups. K seskupení dojde při každé změně pole zpráv, protože se jedná o jednodušší řešení, než vyhodnocovat do které skupiny daná zpráva patří. Z hlediska potenciálního rozšíření je totiž možné, že zprávy budou moci být vloženy do různých skupin - v případě stránkování, znovu odeslání zprávy po selhání apod.

```
@Published var messages: [B2aDicoMessengerMessage] = [] {
    willSet {
        let groupedDict = Dictionary(grouping: newValue) {
            return Calendar.current.dateComponents(
                [.day, .year, .month, .calendar],
                from: Date(timeIntervalSince1970: $0.timestamp))
            }.sorted {
                $0.key.date?.compare($1.key.date ?? Date())
                    == .orderedAscending
            }
        groups = Array(groupedDict).compactMap { item in
            return GroupedMessages(date: item.key.toString(),
                                   messages: item.value)
        }
    }
}
```

Vlastnost topic je konstruována dle kontextu, tedy dle ID místnosti (vlastnost chatroomId).

```
var topic: String {
    return "/b2a/messenger/chat-room/\(chatroomId ?? 0)/message"
}
```

Metoda `registerObserver()` zaregistruje sebe sama do MqttService jako příjemce zpráv.

```
func registerObserver() {
    MqttService.shared.registerObserver(self)
}
```

Metoda `subscribe()` informuje MqttService, že má poslouchat zprávy na patřičném tématu.

```
func subscribe() {
    MqttService.shared.subscribe(to: topic)
}
```

Metoda `unsubscribe()` informuje MqttService, že nemá nadále poslouchat zprávy na patřičném tématu.

```
func unsubscribe() {
    MqttService.shared.unsubscribe(from: topic)
}
```

Metoda `sendMessage()` zkonstruuje zprávu ve formátu, který umí centrální bod přijmout a pomocí MqttService zprávu odešle k příslušnému tématu.

```
func sendMessage() {
    guard let chatroomIdUnwrapped = chatroomId,
          message.isEmpty == false else {
        return
    }
    let message = DicoMqttMessage()
    message.type = "new-message"
    let createObj = B2aDicoMessengerMessageCreate()
    createObj.uuid = .uuidString(length: 64)
    createObj.content = self.message
    message.data = createObj
    MqttService.shared.publish(string: message.toJSONString() ?? "",
      topic: "/b2a/messenger/chat-room/\(chatroomIdUnwrapped)/new-message")
    self.message = .empty
}
```

Metoda `onMessageReceive(_ payloadString: String, topic: String)` je metoda protokolu MqttObserver a je zavolána při příjmu zprávy. Díky parametru topic je možné vyhodnotit, zda je zpráva pro tohoto příjemce relevantní a pokud ano, je přidána do pole messages a tím pádem zobrazena uživateli.

```
func onMessageReceive(_ payloadString: String, topic: String) {
    guard let chatroomIdUnwrapped = chatroomId else {
        return
    }
    if topic == "/b2a/messenger/chat-room/\(chatroomIdUnwrapped)/message" {
        let message = DicoMqttMessage(JSONString: payloadString)
        guard let messageAO = message?.data as? B2aDicoMessengerMessage else {
            return
        }
        messages.append(messageAO)
    }
}
```

### 5.10.2 View

V hierarchii tohoto View bylo pro změnu použito ScrollView a ScrollViewReader. ScrollViewReader umožňuje sescrollovat na určitý element dle jeho id.

Místo List byl zde využit LazyVStack, protože nabízí možnost zobrazit přichycenou hlavičku sekce a nenačítá všechny položky najednou. Toho je možné v budoucnosti využít, protože se dá jednoduše zjistit, zda např. poslední zpráva byla již viděna.

Na LazyVStack byla aplikovaná metoda `onChange(of:perform:)`, která spustí obsluhující kód v případě, že byla předána @Published vlastnost změněna. Konkrétně, pokud byly načteny zprávy, nebo byla zpráva přidána do pole messages, sescrolluje se na poslední zprávu.

Ukázka použití ScrollView, ScrollViewReader a LazyVStack se sekcemi s hlavičkami:

```
...
ScrollView {
    ScrollViewReader { scrollView in
        LazyVStack(spacing: 0, pinnedViews: [.sectionHeaders]) {
            ForEach(viewModel.groups, id: \.date) { group in
                Section(header: DayDividerView(date: group.date)) {
                    ForEach(group.messages, id: \.timestamp) { message in
                        MessageView(message: message)
                            .id(message.timestamp)
                    }
                }
            }
        }
        .onChange(of: viewModel.messages) { messages in
            withAnimation {
                 guard let lastMessageTimestamp =
                   viewModel.messages.last?.timestamp else {
                     return
                 }
                 scrollView.scrollTo(lastMessageTimestamp)
            }
        }
    }
}...
```

Jako poslední View je zobrazena komponenta MessageAreaView, která přijímá stavovou proměnnou message z ViewModelu a akci, kterou je v tomto případě metoda sendMessage ve ViewModelu.

```
BaseView(isLoading: $viewModel.isLoading) {
    VStack {
        ScrollView {
            <...>
        }
        MessageAreaView(message: $viewModel.message,
                        action: viewModel.sendMessage)
    }
}
```

V metodě `onAppear(perform:)` aplikované na BaseView je zavolána metoda pro získání zpráv, registruje se příjemce MQTT zpráv a začne se odebírat téma dané pro tuto místnost.

```
.onAppear {
    guard let chatroomId = chatroom.id else { return }
    viewModel.getMessages(for: chatroomId)
    viewModel.registerObserver()
    viewModel.subscribe()
}
```

V metodě `onDisappear(perform:)` aplikované na BaseView vyvoláme ukončení odběru daného téma.

```
.onDisappear {
    viewModel.unsubscribe()
}
```

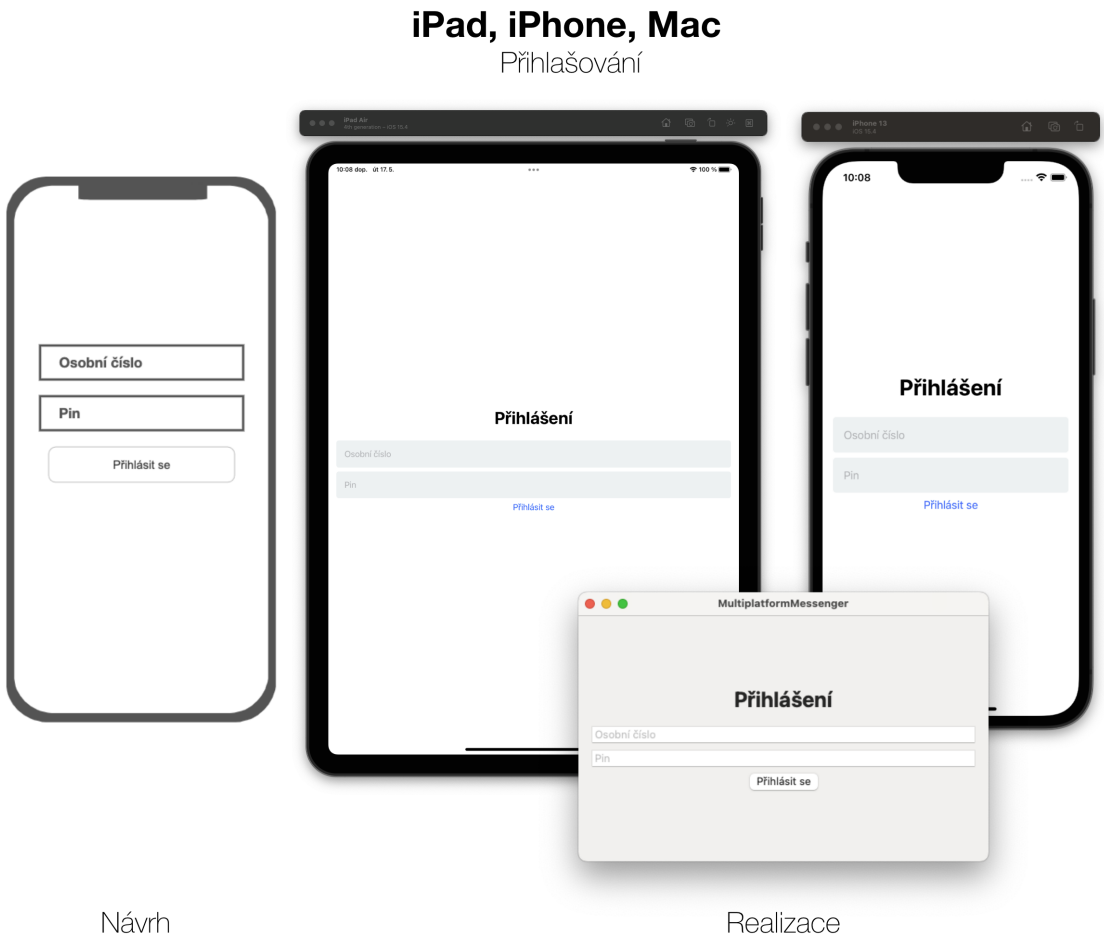
# 6 PREZENTACE DOSAŽENÝCH VÝSLEDKŮ

V této kapitole bude představena stávající funkcionalita a výsledná podoba aplikace na platformách iOS, iPadOS a macOS. Bude polemizováno nad případnými nedostatky a možnostmi potenciálního budoucího rozvoje.

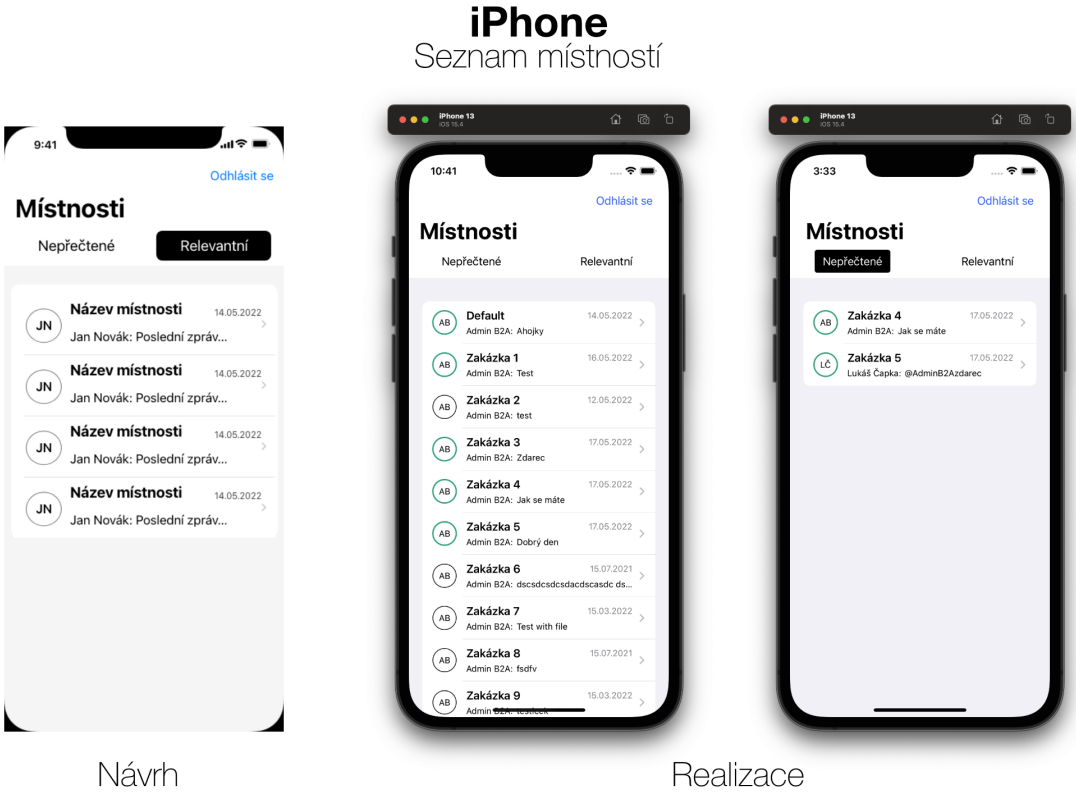
## 6.1 Stávající funkcionalita

Stávající funkcionalita a vzhled aplikace je v úplném souladu s návrhem sepsaným v kapitole 3. Ve stručnosti tedy aplikace umožňuje přihlášení uživatele a po znovu otevření aplikace přihlášení nevyžaduje, pokud je uživatel uložen v perzistentní paměti. Aplikace zobrazuje seznam místností s možností filtrovat a prokliknout se na detail. Na detailu se v reálném čase přijímají zprávy pro danou místnost a taktéž je možné zprávu v reálném čase odeslat. Zprávy jsou seskupeny dle dne, ve kterém byly poslány.

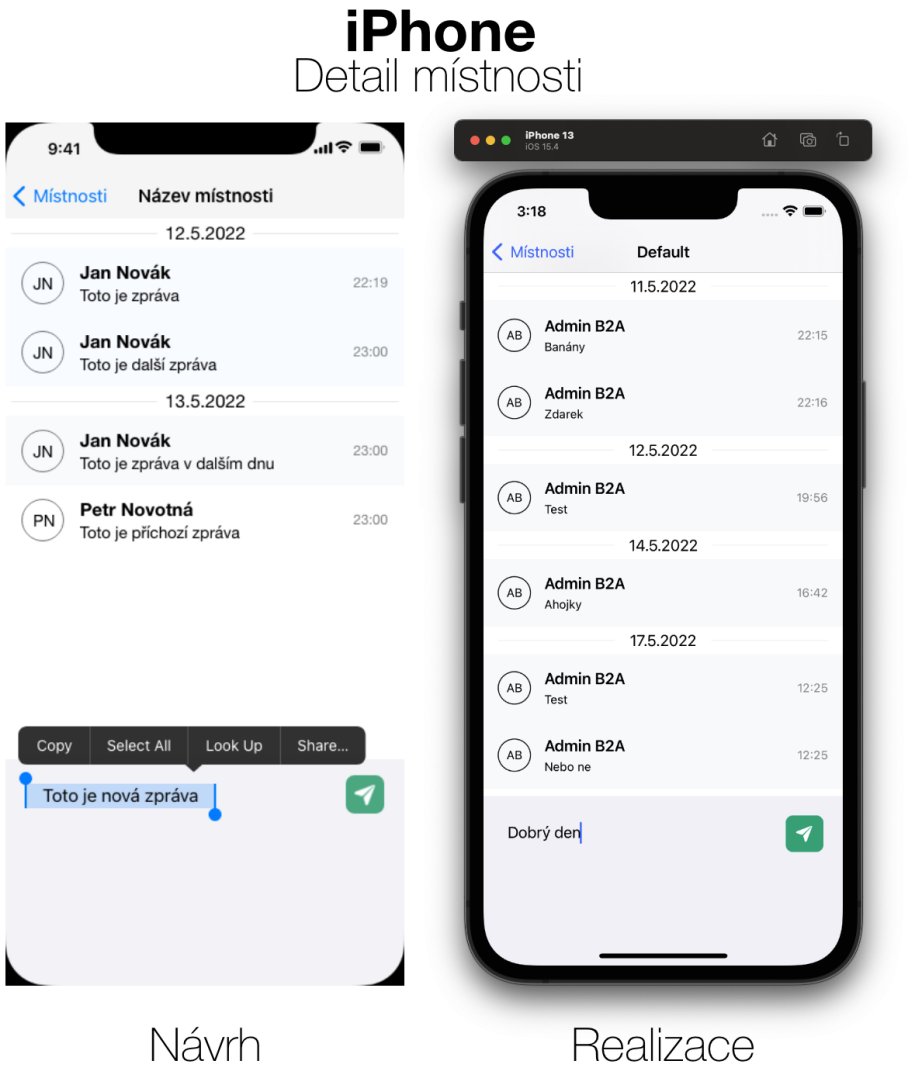
## 6.2 Výsledná podoba (porovnání s UI návrhem)



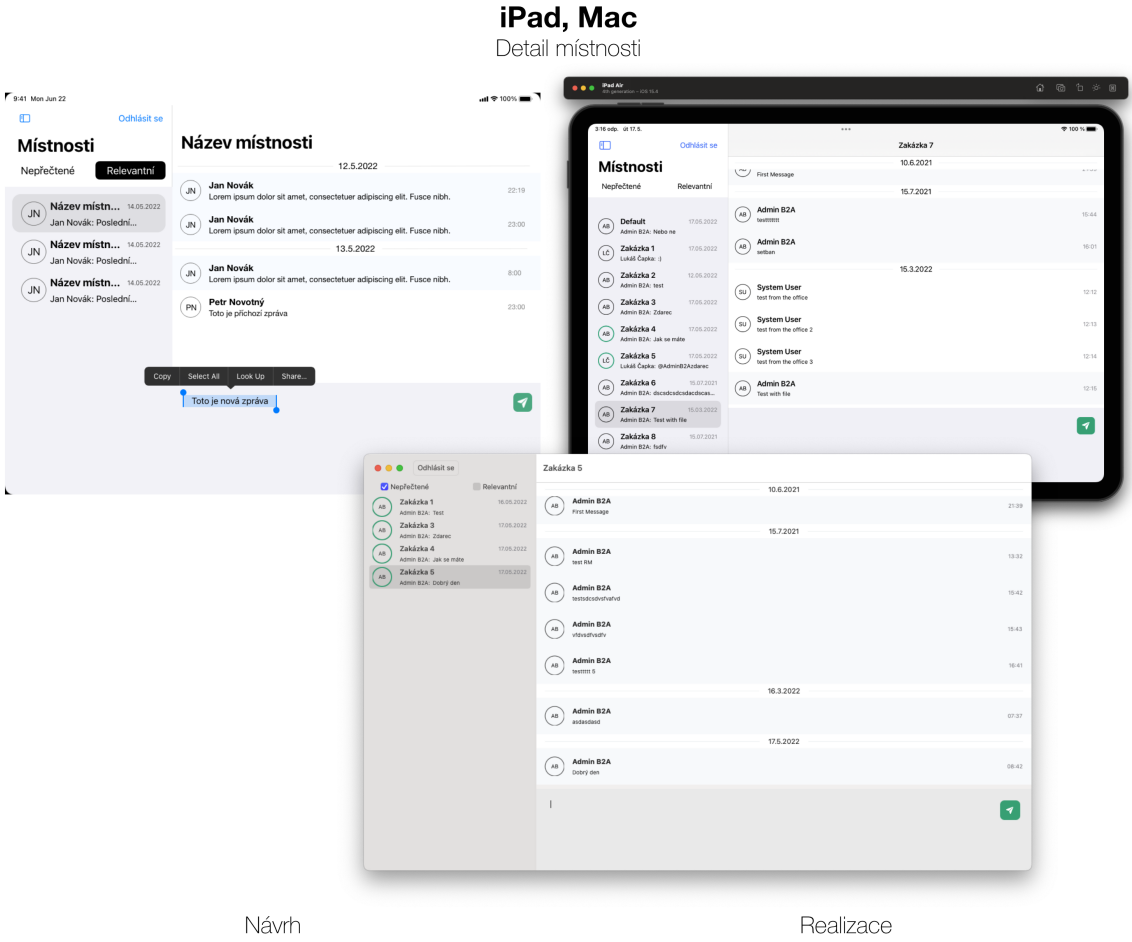

Obrázek 6.1 Porovnání UI návrhu s realizací - Přihlašování (všechny platformy)




Obrázek 6.2 Porovnání UI návrhu s realizací - Seznam místností (iPhone)




Obrázek 6.3 Porovnání UI návrhu s realizací - Detail místnosti (iPhone)




Obrázek 6.4 Porovnání UI návrhu s realizací - Seznam místností a detail místnosti (iPad, Mac)

## 6.3 Nedostatky

Výsledná aplikace má bezesporu hned několik nedostatků. Některé jsou ale neimplementovanou funkcionalitou, a proto budou zmíněny v následující podkapitole.

Nalezené nedostatky:

- Pozadí aktivního prvku v seznamu není konzistentní a není vhodné pro použitou buňku.
- Barvy některých prvků se na platformách liší a nejsou použité obecné barvy, aby aplikace plně podporovala tmavý režim.
- Aplikace po odeslání zprávy pomocí protokolu MQTT čeká na příjem téže zprávy, místo toho, aby ji rovnou přidala do pole a tím tak eliminovala prodlevu před zobrazením.
- Aplikace nezobrazuje v seznamu místností žádnou informaci, pokud žádná místnost neodpovídá zvolenému filtru.
- Aplikace na iPadu v portrait režimu neuchovává data v seznamu místností, a proto při znovuotevření sidebaru není označena právě otevřená místnost.
- Aplikace ve většině případů neinformuje uživatele o nevydařeném API dotazu.

## 6.4 Potenciální rozvoj aplikace

V této podkapitole budou zmíněny funkcionality, které ve výsledné aplikaci chybí a jednoznačně by uživateli zpříjemnili používání.

- Aplikace by mohla v reálném čase na obrazovce se seznamem místností reagovat na události a aktualizovat tak mísnosti, ve kterých je nově přichozí zpráva nebo ve kterém byla poslední zpráva přečtena.
- Aplikace by mohla v seznamu místností zobrazovat místo data text *Dnes*, pokud je poslední zpráva v místnosti odeslána v aktuálním dni.
- Aplikace by mohla v oddělovači dnů na detailu zobrazovat *Dnes*, pokud je datum dané skupiny v aktuálním dni.
- Aplikace by mohla ve zprávách podporovat zmínky, tedy označení konkrétního uživatele. Zmínkou uživatele vznikne relevantní zpráva a místnost lze tak vyfiltrovat jako relevantní.

- Aplikace by mohla podporovat push notifikace a informovat tak uživatele o nové zprávě a ihned jej přesměrovat do správné místnosti.
- Aplikace by mohla podporovat znovu odeslání zprávy v případě, že se zprávu nepodaří z jakéhokoliv důvodu odeslat.

# ZÁVĚR

Cílem této práce bylo nastudovat a popsat funkce protokolu MQTT a seznámit se s principem vývoje multiplatformních aplikací pro platformy společnosti Apple. V první kapitole teoretické části bylo popsáno, co to protokol MQTT je, jak funguje, co jsou to témata, zprávy a kvalita služeb. Druhá kapitola se věnuje technologii Mac Catalyst a frameworku SwiftUI, které slouží pro vývoj aplikací pro více platforem společnosti Apple. Dále byl stručně zmíněn i architektonický vzor MVVM.

Praktická část se zabývá návrhem a implementací klientské části aplikace, kde jako první byly sepsány funkční a nefunkční požadavky. Dále byly navrženy obrazovky pomocí jednoduchých drátěných modelů a jejich funkcionalita. V druhé kapitole praktické části již bylo reálněji navrženo uživatelské rozhraní pomocí nástroje Adobe XD a pro návrh bylo využito převážně nativních komponent společnosti Apple.

V praktické části byl také podrobně a velmi návodně zdokumentován popis implementace klientské části aplikace s využitím architektonického vzoru MVVM a frameworku SwiftUI a může tak být zajímavým přínosem pro čtenáře.

V poslední kapitole praktické části byly prezentovány jednotlivé obrazovky výsledné aplikace na všech cílených platformách s porovnáním původního návrhu uživatelského rozhraní. Byly taktéž vypíchnuty nedostatky a možnosti případného pokračování.

## SEZNAM POUŽITÉ LITERATURY

[1] Co je MQTT a k čemu slouží ve IIoT? Popis protokolu MQTT. *IPC2U* [online]. [cit. 2022-05-17]. Dostupné z: https://ipc2u.cz/blogs/news/mqtt-protokol

[2] The HiveMQ Team. Introducing the MQTT Protocol - MQTT Essentials: Part 1. *HiveMQ* [online]. 2015 [cit. 2022-05-17]. Dostupné z: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/

[3] OBDRŽÁLEK, David. Co je MQTT. *RoboDoupě* [online]. 2019 [cit. 2022-05-17]. Dostupné z: http://robodoupe.cz/2019/co-je-mqtt/

[4] MQTT: otevřený síťový protokol a jeho význam v IoT. In: *HW libre* [online]. [cit. 2022-05-20]. Dostupné z: https://www.hwlibre.com/cs/mqtt/

[5] VOJÁČEK, Antonín. IoT MQTT prakticky v automatizaci - 1.díl - úvod. *Automatizace.HW.cz* [online]. 2017 [cit. 2022-05-17]. Dostupné z: https://automatizace.hw.cz/iot-mqtt-prakticky-v-automatizaci-1dil-uvod.html

[6] MALÝ, Martin. Protokol MQTT: komunikační standard pro IoT. *Root.cz* [online]. 2016 [cit. 2022-05-17]. Dostupné z: https://www.root.cz/clanky/protokol-mqtt-komunikacni-standard-pro-iot/

[7] The HiveMQ Team. Quality of Service (QoS) 0,1, & 2 MQTT Essentials: Part 6. *HiveMQ* [online]. 2015 [cit. 2022-05-17]. Dostupné z: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/

[8] UIKit. *Apple Developer* [online]. [cit. 2022-05-17]. Dostupné z: https://developer.apple.com/documentation/uikit

[9] AppKit. *Apple Developer* [online]. [cit. 2022-05-17]. Dostupné z: https://developer.apple.com/documentation/appkit

[10] WatchKit. *Apple Developer* [online]. [cit. 2022-05-17]. Dostupné z: https://developer.apple.com/documentation/watchkit

[11] Mac Catalyst. *AppleInsider* [online]. [cit. 2022-05-17]. Dostupné z: https://appleinsider.com/inside/mac-catalyst

[12] Mac Catalyst. *Apple Developer* [online]. [cit. 2022-05-17]. Dostupné z: https://developer.apple.com/mac-catalyst/

[13] HADSON, Paul. What is SwiftUI?. *HACKING WITH SWIFT* [online]. 2021 [cit. 2022-05-17]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui

[14] SwiftUI. *Apple Developer* [online]. [cit. 2022-05-17]. Dostupné z: https://developer.apple.com/xcode/swiftui/

[15] Discover SwiftUI. *Swift by Sundell* [online]. [cit. 2022-05-17]. Dostupné z: https://www.swiftbysundell.com/discover/swiftui/

[16] DAJBYCH, Václav. Mvvm: model-view-viewmodel. *DotNetPortal* [online]. 2009 [cit. 2022-05-17]. Dostupné z: https://www.dotnetportal.cz/clanek/4994/MVVM-Model-View-ViewModel

[17] MIŠTOVÁ, Jaroslava. MVVM. *Ackee* [online]. 2018 [cit. 2022-05-17]. Dostupné z: https://www.ackee.cz/blog/glossary/mvvm

[18] ČÁPKA, David. Lekce 3 - Třívrstvá architektura a další vícevrstvé architektury. *ITNetwork* [online]. 2017 [cit. 2022-05-17]. Dostupné z: https://www.itnetwork.cz/navrh/architektury-a-dependency-injection/trivrstva-architektura-a-dalsi-vicevrstve-architektury

[19] MANILII, Alessandro. MVVM Pattern in SwiftUI: A Practical Example. In: *Devgenius* [online]. 2022 [cit. 2022-05-20]. Dostupné z: https://blog.devgenius.io/mvvm-pattern-in-swiftui-a-practical-example-c79c5cc44f74

# SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

| | |
|---|---|
| MQTT | Message Queuing Telemetry Transport |
| MVVM | Model-View-ViewModel |
| UI | User Interface |
| API | Application Programming Interface |

## SEZNAM OBRÁZKŮ

## SEZNAM TABULEK

## SEZNAM PŘÍLOH

# PŘÍLOHA P I. PŘILOŽENÉ CD

Přiložené CD obsahuje:

- bakalářskou práci ve formátu PDF,
- zdrojové kódy aplikace.