Vývoj běžecké aplikace Runny pro iOS a watchOS

Filip Vabroušek

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Filip Vabroušek**
Osobní číslo:	**A22609**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Vývoj běžecké aplikace Runny pro iOS a watchOS**
Téma práce anglicky:	**Development of Runny Running App for iOS and WatchOS**

Zásady pro vypracování

1. Stručně popište možnosti vývoje aplikace pro platformy iOS, watchOS a web.
2. Vyberte vhodné technologie pro vývoj na výše uvedených platformách, včetně technologie serverové strany a popište je.
3. Pro praktickou část navrhněte funkční a nefunkční požadavky na běžeckou aplikaci, která bude schopna kvantifikovat parametry běhu a zálohovat je na web.
4. Navrhněte grafický design obrazovek a uživatelské rozhraní reflektující výše uvedené požadavky.
5. Implementujte aplikaci pro iOS, watchOS a web.
6. Zhodnoťte přínos aplikace pro běžce používající výše uvedené platformy.

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

Seznam doporučené literatury:

1. NOAKES, Tim. Lore of Running. 4th edition. OXFORD UNIVERSITY PRESS, 2001. ISBN 0195780167
2. SLEAMAKER, Rob a Ray BROWNING. SERIOUS Training for Endurance Athletes. 2nd edition. Human Kinetics, 1996. ISBN 0873226445.
3. LACKO, Luboslav. Vývoj aplikaci pro iOS. Brno: Computer Press, 2018. ISBN 978-80-251-4942-3.
4. Apple Inc. SwiftUI | Apple Developer Documentation. [online dokumentace]. ©2023 [cit.08.11.2023] Dostupné z: https://developer.apple.com/documentation/swiftui/
5. Google Inc. Firebase | Google's Mobile and Web App Development Platform. [online dokumentace]. ©2023 [cit.08.11.2023] Dostupné z: https://firebase.google.com/
6. Apple Inc. Core Location | Apple Developer Documentation. [online dokumentace]. ©2023 [cit.08.11.2023] Dostupné z: https://developer.apple.com/documentation/corelocation
7. Apple Inc. MapKit | Apple Developer Documentation. [online dokumentace]. ©2023 [cit.08.11.2023] Dostupné z: https://developer.apple.com/documentation/mapkit/
8. Apple Inc. Core Motion | Apple Developer Documentation. [online dokumentace]. ©2023 [cit.08.11.2023] Dostupné z: https://developer.apple.com/documentation/coremotion

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

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

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

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

Ve Zlíně dne 5. ledna 2024

U

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové 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 diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/ 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 diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB 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 – diplomovou 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 diplomové 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 diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Filip Vabroušek, v. r.

ABSTRAKT

Tématem této diplomové práce je mobilní aplikace Runny pro operační systémy iOS a watchOS. Aplikace slouží k zaznamenávání běžeckých aktivit. Od ostatních aplikací se odlišuje tím, že kromě zaznamenání běžných běžeckých metrik umožňuje sledování efektivního tempa, efektivní kadence a efektivního srdečního tepu. Aktivity v aplikaci jsou zcela soukromé, pokud není rozhodnuto uživatelem o sdílení. Vyhodnocení aktivit obsahuje i statistiky umožňující sledování vývoje běžecké výkonosti.

Aplikace byla vytvořena pro tři platformy: iOS, watchOS a web. Při zpracování této aplikace byl použit framework SwiftUI a UIKit pro platformu iOS, a SwiftUI pro platformu watchOS. Pro tvorbu webové verze byla použita knihovna React a vlastní knihovna webových komponent. Jako technologie serverové strany byla použita backendová knihovna Firebase. Aplikace Runny je dostupná na App Store, její webová verze je zveřejněna na adrese https://runny-app.github.io/.

Klíčová slova: SwiftUI, UIKit, React, Firebase, iOS, watchOS, web, App Store

ABSTRACT

The topic of this thesis is the mobile application Runny for iOS and watchOS operating systems. The application is used to record running activities. It differs from other apps in that it allows tracking of effective pace, effective cadence, and effective heart rate in addition to recording common running metrics. Activities in the app are completely private unless a decision is made by the user to share. Activity evaluation also includes statistics to track the evolution of running performance.

The application was created for three platforms: iOS, watchOS and the web. The mobile app uses the SwiftUI and UIKit framework for the iOS platform, and SwiftUI for the watchOS platform. The React library and my own web component library were used to create the web version. Firebase backend library was used as the server side technology. The Runny app is available on the App Store, and the web version is published at https://runny-app.github.io/.

Keywords: SwiftUI, UIKit, React, Firebase, iOS, watchOS, web, App Store

Děkuji Ing. Radku Valovi, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích k této diplomové práci.
Také děkuji svým rodičům za všestrannou podporu při studiu.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Obsah

ÚVOD

Cílem této práce bylo vytvořit mobilní aplikaci Runny pro zaznamenávání a relevantní vyhodnocování parametrů běžeckých aktivit. Aplikace umožňuje znázorňovat a měřit trasu uživatele pomocí GPS a hrudního pásu.

Aplikace tato data (vzdálenost, čas, tempo, tepovou frekvenci, kadenci) ukládá a je možné si v ní přehledně zobrazit jejich historii a libovolně definované statistiky.

Jako jediná běžecká aplikace na trhu průměrné hodnoty vyhodnocuje také od začátku do konce aktivity a zohledňuje úvodní fázi rozběhání a závěrečné vyklusání. Srovnávací hodnota takto definovaných průměrů má výrazně vyšší vypovídací hodnotu o skutečném zaměření tréninkové jednotky.

Runny vyhodnocuje nejen tato celková data, ale také průměrné hodnoty hlavní fáze (od ukončení rozběhání do zahájení vyklusání).

Zdrojem GPS dat jsou běžecké hodinky Apple Watch (watch OS) nebo iPhone (iOS). Zdrojem dat tepové a krokové frekvence je běžecký hrudní pás Garmin. Při zpracování této mobilní aplikace pro platformu iOS byl použit framework SwiftUI a UIKit. Pro watchOS aplikaci byla využita knihovna SwiftUI. Pro komunikaci s webovým rozhraním byla použita knihovna Firebase Firestore.

Pro tvorbu webové verze pro zálohování běžeckých aktivit byla aplikována knihovna React s vlastní knihovnu webových komponent. Aplikace Runny je dostupná na App Store, její webová verze je zveřejněna na adrese https://runny-app.github.io/.

I. TEORETICKÁ ČÁST

1 Možnosti vývoje pro platformu iOS

V následujícím textu jsou uvedeny základní pojmy týkající se programovacích jazyků a požadavků na vývoj aplikací. Také jsou zmíněny využívané frameworky pro vývoj na platformě iOS.

1.1 Programovací jazyky

Počítače neumí provést žádný úkon bez programu, který by počítači poskytoval přesné instrukce. Počítače rozumí pouze jednoduchým příkazům ve formě nul a jedniček. Proto vznikly programovací jazyky, které umožňují zapsat programy formou čitelnou pro člověka.

Firma Apple původně využívala jazyk Objective-C, který vznikl v roce 1984. Tento jazyk ale kvůli své komplexnosti příliš nepřitáhl vývojáře. Proto Apple v roce 2014 představil nový programovací jazyk Swift. Tento jazyk má jednodušší syntaxi než Objective-C, a na rozdíl od něj automaticky maže nepotřebnou paměť. Stal se tedy jasnou volbou pro začínající vývojáře na Apple platformách. [1]

1.2 Vývoj aplikací

Vývoj mobilních aplikací je rychle rozvíjející se odvětví. Oficiální obchod s aplikacemi - App Store - vznikl v roce 2008, kdy obsahoval několik set aplikací. [2] V roce 2024, už se jedná přibližně o 4,97 milionu aplikací a her [3]. Mobilní aplikace ve spoustě případů nahradily mobilní stránky, a vytvořily vysoce zabezpečený systém. Na platformách Apple iOS, iPadOS a watchOS se totiž mimo EU defaultně nedají spustit aplikace z jiného zdroje, než právě z App Store. Nyní si projdeme seznam nástrojů, které budou k vývoji potřeba. [4]

1.3 Požadavky na vývoj aplikací

Chceme-li vyvíjet mobilní aplikace uvádím požadavky na potřebný hardware a software.

1.3.1 Mac

Xcode je možné používat pouze na Macu. I když je možné Mac používat v cloudu, je vhodné si na vývoj zakoupit jakýkoli Mac s procesorem Apple Silicon M3, nebo novějším, který podporuje nejnovější operační systém. V době psaní práce macOS 14 (Sonoma). [5]

1.3.2 Xcode

Xcode je IDE (Integrated Development Environment = Integrované vývojářské prostředí) pro vývoj aplikací na Macu. Xcode lze stáhnout zdarma z Mac App Store. [6]

1.3.3 Vývojářský účet

Xcode je možné používat i bez vývojářského účtu. Na vývoj na zařízení stačí běžný účet AppleID, na nahrání do App Storu a přístup k beta verzím software je potřeba mít vývojářský účet. [7]

1.3.4 Apple Developer členství

Pro zveřejnění aplikace na App Store je potřeba vývojářský účet. Apple si za tento účet účtuje 99$ na rok. Je možné ho vytvořit na adrese: https://developer.apple.com/programs/enroll/ [8]

1.3.5 Mobilní zařízení Apple

I když je možné aplikaci testovat na simulátoru, je před vydáním doporučováno otestovat aplikaci na reálném Apple zařízení. Toto zařízení by mělo podporovat iOS 17 nebo iPadOS 17 nebo novější. Pro vývoj aplikací na watchOS 10 na reálném zařízení jsou potřeba Apple Watch. [9]

1.3.6 Instalace Xcode

Xcode lze stáhnout zdarma z Mac App Store. Použijeme následující postup:

1) na Macu otevřeme aplikaci App Store

2) do vyhledávacího pole zadáme Xcode

3) nainstalujeme Xcode klepnutím na tlačítko “Get/Získat” [6]

1.4 Frameworky (API a knihovny)

Vývoj jednoduché aplikace s pomocí pouhého programovacího jazyku by trval dlouhé roky. Proto Apple vytvořil frameworky a knihovny, které obsahují předepsané bloky kódu usnadňující vývoj aplikací. Nejprve Apple uvedl framework Foundation obsahující nástroje pro práci s daty, textem atd., a framework UIKit pro tvorbu uživatelských rozhraní. V roce 2019 na konferenci WWDC19 Apple uvedl nový framework SwiftUI, který výrazně zjednodušil vývoj aplikací. Protože je SwiftUI nejnovější knihovna pro vývoj aplikací, věnoval jsem jí největší část teoretické části práce. [10]

1.4.1 Cocoa Touch

Cocoa Touch je framework, který poskytuje základní vrstvu abstrakce nad systémovým rozhraním. Umožňuje přístup ke službám zařízení Apple, jako GPS, akcelerometr a kamera. [11]

1.4.2 UIKit

Nad frameworkem Cocoa Touch se nachází framework UIKit. Tento framework je napsaný v jazyce Objective-C a byl vydán v roce 2008. Framework UIKit umožňuje jednoduše tvořit aplikace pomocí imperativních instrukcí. V roce 2019 ale Apple uvedl zcela nový framework napsaný v jazyce Swift, který výrazně zjednodušil tvorbu uživatelských rozhraní na platformách firmy Apple - SwiftUI. Některé views ve SwiftUI interně používají prvky ze frameworku UIKit. [12]

1.4.3 Foundation

Foundation je framework napsaný v jazyce Objective-C, který usnadňuje časté operace, které potřebujeme řešit v naší aplikaci, jako například:

- používání matematických funkcí

- převody jednotek
- převody časů mezi regiony
- převody jednotek měn

a mnohé další.

Pokud chceme framework použít v našem kódu, importujeme ho pomocí následující deklarace:

```
import Foundation [13]
```

1.4.3.1 Matematické funkce

Mezi základní matematické funkce patří zjištění minima a maxima ze 2 čísel. Ve Swiftu toho lze docílit pomocí funkcí min a max.

```
min(22, 10) // 10
max(22, 10) // 22
```

Mezi další funkce se řadí: `pow(a, b)` vrátí číslo a umocněné na mocninu b [14]

`sqrt(a)` vrátí odmocninu z čísla a

log(a) vrátí přirozený logaritmus z čísla a

```
pow(3,3) //9
sqrt(23) // 4.79
log(21.0) // 1.32 [15]
```

1.4.3.2 Práce s jednotkami

Framework Foundation nám jednoduše umožní formátovat údaje, např. měnu: Nejprve pomocí třídy `NSNumber` definujeme proměnnou `amount` a poté instanci třídy `NSFormatter()`, které přiřadíme formátovací styl na `.currency` (měna). Dále dosadíme proměnnou `amount` do funkce `string(from:_)` a získáme výsledek "$9.58" (9.58 amerického dolaru) [16]

```
var amount = NSNumber(value: 9.41)
let format = NumberFormatter()
format.numberStyle = .currency
let price = format.string(from: amount) //"$9.41"
```

Framework Foundation nám jednoduše umožňuje převádět jednotky. Na tomto příkladu je ukázán převod z kilometrů na míle.

```
let distance = Measurement(value: 100, unit: UnitLength.kilometers)
let miles = distance.converted(to: .miles)
// 62.13711922373339 mi [17]
```

1.4.3.3 Sčítání jednotek

Je také jednoduché sečíst jednotky různých délek. V tomto případě zadáváme 2 metry a 2 centimetry a pomocí třídy `Measurement` a sečítáme je v proměnné `height`. Výsledek je 2.38 m, tedy 2 metry a 38 centimetrů.

```
let meters = Measurement(value: 2, unit: UnitLength.meters)
let centimeters = Measurement(value: 38, unit: UnitLength.centimeters)
let height = meters + centimeters
print(height) // "2.38 m\n" [17]
```

1.4.3.4 Práce s daty a časem

Pro zobrazení aktuálního roku, hodiny, sekundy, nebo čísla aktuálního měsíce, lze využít třídy `Calendar`. Na prvním řádku si definujeme komponenty `.year, .month, .day` a do argumentu `from` napíšeme `Date()`, což nám vrátí aktuální datum. Na dalších řádcích si z data přečteme číslo měsíce, a napíšeme ho do konzole. Protože se jedná o `Optional`, musíme proměnnou rozbalit pomocí symbolu `!`.

```
let calendar = Calendar.current.dateComponents([.year, .month, .day],
from: Date())
let monthNumber = calendar.month
print("This month has a number \(monthNumber!).") [18]
```

Pokud například chceme zobrazit aktuální čas v americkém formátu, lze použít třídu `Locale` s identifikátorem `en_US_POSIX`. Čas 15:38 v ČR bude zapsán jako 3:38 PM.

```
let today = Date()
let hd = DateFormatter()
hd.flocafle = Locafle(fidentfifffier: "en_US_POSIX")
hd.dateFormat = "hh:mm a"
hd.amSymbol = "AM"
hd.pmSymbol = "PM"
let USATime = hd.string(from: today) // 3:38 PM [19][20]
```

1.4.3.5 Formátování data

Nejprve definujeme dnešní datum do proměnné today pomocí třídy `Date()`. Pomocí třídy `DateFormatter` můžeme naformátovat datum na formát, jaký si přejeme. V tomto případě nastavíme formát data "`dd/MM/yyyy`", (den/měsíc/rok). Poté použijeme metodu `formatter.string(from: today)`, která nám vrátí datum v požadovaném formátu.

```
import Foundation
let today = Date()
let formatter = DateFormatter()
formatter.dateFormat = "dd/MM/yyyy"
```

```
let formattedDate = formatter.string(from: today)[19][20]
```

1.4.3.6 Formátování data pro určitý region

Je také možné datum naformátovat pro jakýkoli region. V tomto případě nastavíme region našeho data na čínský pomocí třídy `Locale` a kódu `zh_CN`. Poté už jen stačí přiřadit instanci třídy `Locale` našemu formatteru a vypsat výsledek v konstantě `date`.

```
let today = Date()
let formatter = DateFormatter()
formatter.dateStyle = .full
flet chfinaLocafle = Locafle(fidentfifffier: “zh_CN”)
formatter.locale = chinaLocale
let date = formatter.string(from: today)[19][20]
```

1.4.3.7 Práce s časovými pásmy

Nejprve si definujeme proměnnou, která obsahuje pražskou časovou zónu. Poté si definujeme dnešní datum a `DateFormater`, ve kterém nastavíme styl data na `.long` a styl času na `.short`. Poté nastavíme jeho časovou zónu na pražskou a do proměnné vytvoříme datum formátované na pražský čas.

```
flet pragueTfimeZone = TfimeZone(fidentfifffier: “Europe/Prague”)
let today = Date()
let formatter = DateFormatter()
formatter.dateStyle = .long
formatter.timeStyle = .short
formatter.timeZone = pragueTimeZone
let pragueDate = formatter.string(from: today)[21][20]
// “March 19, 2024 at 10:19 PM”
```

1.4.3.8 Časovač

Například můžeme vytvořit časovač, který každé tři sekundy vypíše text “three seconds have passed” na obrazovku.

```
Timer.scheduledTimer(withTimeInterval: 3.0, repeats: true) { (timer) in
    print(“three seconds have passed”)
}
```

Na dalším příkladě si ukážeme, jak časovač zastavit, když potřebujeme. Každé 3 sekundy navýšíme hodnotu proměnné `value` o 1, a pokud hodnota přesáhne 3 zastavíme časovač.

```
var timer: Timer
var value = 0
```

```
timer = Timer.scheduledTimer(withTimeInterval: 2.0, repeats: true) {
(timer) in
    print("3 seconds have passed")
    value += 1
    if value > 3 {
        timer.invalidate()
    }
} [22][23]
```

1.4.3.9 Formátování řetězců

Pomocí metody `.localizedStringWithFormat` můžeme napsat lokalizovaný textový řetězec s číslem ve odpovídající regionu uživatele.

```
let amount = 12.341
let decimal = String.localizedStringWithFormat("Decimals: %.2f", amount)
// "Decimals: 12.34" [24]
```

1.4.3.10 Nahrazení části textového řetezce

Je také možné nahradit část textového řetězce jiným textem. Nejprve definujeme proměnné s naším textem text a řetězcem k vyhledání `search`. Poté ze `search` ořežeme prázdné místo a případná zalomení řádků pomocí `.trimmingCharacters(in: .whitespacesAndNewlines).`

```
var text = "Filip is 24"
var search = "24 "
search = search.trimmingCharacters(in: .whitespacesAndNewlines) [25]
```

Dále definujeme proměnnou `range` která nám vrátí rozsah hledného textu v původním řetezci. Pomocí `.caseInsensitive` zajistíme, že se bude vyhledávat textový řetězec převedený na malá písmena. Poté si pomocí `if let` ověříme, že řetězec obsahuje daný rozsah, a protože jsem zase o rok starší, nahradíme daný rozsah pomocí `replaceSubrange`() textem 25.

```
var range = text.range(of: search, options: .caseInsensitive)
if let rangeToReplace = range {
    text.replaceSubrange(rangeToReplace, with: "25") [26][27]
}
print(text) // "Filip is 25"
```

1.4.3.11 Zjišťování pořadí slov

Foundation nám umožňuje zjistit pořadí slov pomocí metody `compare`. Například: "Orange" a "Apple". Protože je v abecedě "A" před "O", výsledkem bude "Fruit follows search", protože `.orderAscending` seřazení položky dle abecedy ve vzrůstajícím pořadí. [28]

```
var fruit = "Orange
```

```
var search = "Apple"
var results = fruit.compare(search, options: .caseInsensitive)
switch results {
case .orderedSame:
    print("Fruit and search are equal.")

case .orderedAscending:
    print("Fruit follows search")

case .orderedDescending:
    print("Fruit precedes Search.")
}
```

Ve frameworku Foundation se dá provádět další množství operací, jako například získávání dat z webu, atd. Framework je v aplikaci Runny využit například na naformátování data.

1.4.4 SwiftUI

SwiftUI je deklarativní framework, který Apple představil na konferenci WWDC 2019. Framework razantně ulehčil tvorbu deklarativních uživatelských rozhraní na všech platformách firmy Apple a zjednodušil vstup do světa vývoje začínajícím vývojářům. [10]

1.4.4.1 Základy SwiftUI

Pro vytvoření aplikace byl otevřen software Xcode, a byla vybrána možnost File -> New -> Project. Zkontrolujeme, že je nahoře vybraná záložka iOS, a vybraná možnost App. Bylo klepnuto na Next, do pole Product name byl zadán název našeho projektu: Runny, a bylo stisknuto tlačítko Next. Nakonec bylo stisknuto tlačítko Create.

Otevřeme soubor Xcode_UIApp.swift a vymažeme jeho obsah. Pomocí import SwiftUI importujeme framework SwiftUI. Poté byla definována struktura `MyApp`, která bude dědit z protokolu `App`. Následně byla zapsána požadovaná proměnná `body`, která pomocí protokolu Scene a klíčového slova some byla vrácena scénu v aplikaci V tomto případě `WindowGroup`, který obsahuje `Text("Hello!")`, který bude zobrazen v našem uživatelském rozhraní. [29]

```
import SwiftUI
@main
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
           Text("Hello!")
        }
```

```
    }
}
```

1.4.4.2 SwiftUI View

SwiftUI nám umožní vytvořit náš vlastní view. Byl otevřen soubor `ContentView`.swift a byl smazán jeho obsah. Byla zapsána následující deklarace view. [30]

```
import SwiftUI
struct MyView: View {
    var body: some View {
        return Text("Hello!")
    }
}
```

Následně byl tento view přidán do aplikace tak, že byl vložen do těla `WindowGroup`.

```
import SwiftUI
@main
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
           MyView()
        }
    }
}
```

1.4.4.3 Previews

SwiftUI nám umožňuje používat tzv. Previews. Previews slouží k náhledu živé aplikace. Xcode vytvoří Previews za nás, můžeme si ale také vytvořit vlastní Previews. Při tvorbě Preview Musíme vyhovět protokolu PreviewProvider, který vyžaduje proměnnou preview, která vrací `some View`. V tomto případě bude zobrazen element `MyView`. Následující kód bude zapsán přímo pod view `MyView`. [31]

```
struct MyPreview: PreviewProvider {
    static var previews: some View {
        MyView()
    }
}
```

V pravé části Okna Xcode je zobrazena plocha `Preview` s náhledem naší aplikace. Jedná o naši už zkompilovanou aplikaci. Pravděpodobně se nám zobrazí zpráva Automatic preview updating paused. Pokud ano, stačí stisknout tlačítko Resume. Po eventuální změně kódu se

Preview automaticky překreslí. Previews jsou interaktivní, takže nám umožňují graficky editovat prvky našeho uživatelského rozhraní, a nebo přidávat nové pomocí Drag & Drop, o tomto bude diskutováno ke konci této sekce. [31]

Ve view můžeme definovat properties, jako v klasických strukturách. V tomto příkladu si definujeme proměnnou name, která bude mít hodnotu Filip. Poté si hodnotu této proměnné vypíšeme v elementu Text

```
struct GreetingView: View {
    var name: String = "Filip"
    var body: some View {
        Text("Hello \(name)!")
    }
}
```

Pokud chceme, aby bylo možné do view dosadit jakékoli jméno, nepřiřadíme proměnné `name` hodnotu.[32]

```
struct GreetingView: View {
    var name: String
    var body: some View {
        Text("Hello \(name)!")
    }
}
```

V Preview bude příklad vypadat následovně: [32]

```
struct MyPreview: PreviewProvider {
    static var previews: some View {
        GreetingView(name: "Filip")
    }
}
```

Od verze iOS 13 podporují všechny operační systémy Apple Dark Mode. Uvnitř Previews můžeme zobrazit náhled view v Dark Mode pomocí `.preferredColorScheme(.dark)`. [33]

```
struct MyPreview: PreviewProvider {
    static var previews: some View {
        MyView()
            .preferredColorScheme(.dark)
    }
}
```

Je možné si zobrazit více Previews nad sebou:

```
struct MyPreviews: PreviewProvider {
    static var previews: some View {
        Group {
            MyView()
            MyView()
        }
    }
}
```

Pomocí `PreviewDevice` si můžeme nastavit zařízení, na kterém naše aplikace poběží.

```
MyView()
      .previewDevice("iPhone 15 Pro")
```

Pokud chceme vytvořit `Preview`, které má přesně stejnou velikost jako náš view, použijeme modifier `PreviewLayout` s hodnotou `.sizeThatFits`. [34]

```
MyView()
      .previewLayout(.sizeThatFits)
```

Pokud známe přesné rozměry v obrazových bodech, které chceme aby naše aplikace podporovala, je možné použít volbu `.fffixed(wfidth: _, hefight: _)`, kde za `width` a `height` dosadíme hodnoty, které potřebujeme.

V tomto případě nám vznikne `view` o rozměrech `200 × 300` obrazových bodů [35]

```
MyView()
    .prevfiewLayout(.fffixed(wfidth: 200, hefight: 300))
```

Pomocí tzv. environmentální proměnné se můžeme například zeptat, zda naše aplikace běží na Dark Mode. Pokud ano, změníme text v aplikaci na "Dark Mode". [36]

```
struct EnvfironmentTest: Vfiew {
    @Envfironment(\.coflorScheme) var coflorScheme: CoflorScheme
    var body: some View {
       Text(colorScheme == .dark ? "Dark Mode" : "Light Mode")
    }
}
```

Můžeme také simulovat nastavení velikosti textu v systému uživatelem. [37]

```
MyView()
```

```
.envfironment(\.sfizeCategory, .extraExtraLarge)
```

Tyto volby jsou dostupné i v rozklikávacím menu, které se objeví po stisknutí prvního tlačítka napravo od textu “Preview” v Preview panelu.

1.4.4.5 Layout algoritmus

Algoritmus SwiftUI vykoná tři zásadní kroky při tvorbě layoutu.

1) Rodičovský view navrhne elementu `Text` celou svoji velikost.

2) View `Text` si vybere svoji velikost

3) Rodič musí element `Text` někam umístit, a tak jej umístí do středu obrazovky. [38]

1.4.4.6 Lifecycle

Pomocí environmentální proměnné `scenePhase` a modifikátoru `onChange`, můžeme reagovat na jednotlivé stavy naší aplikace.

`.active` - scéna je v popředí a není aktivní

`.background` - scéna právě není viditelná

`.inactive` - scéna je v popředí, ale měl by se zastavit spuštěný kód [39]

```
@main
struct MyApp: App {
    @Envfironment(\.scenePhase) prfivate var scenePhase
    var body: some Scene {
        WindowGroup {
          Text(“Hello”)
        }.onChange(of: scenePhase) { (phase) in
            if phase == .active {
                print(“Scene is in the foreground and inactive”)
            }
            if phase == .background {
                print(“Scene isn’t currently visible in the UI”)
            }
            if phase == .inactive {
                print(“Scene is in the foreground, but should pause its
work.”)
            }

        }
    }
```

```
}
```

1.4.4.6 Modifikátory

SwiftUI nám umožňuje změnit vzhled elementů UI pomocí tzv. modifiers (modifikátorů). V tomto příkladu použijeme modifikátor `foregroundColor`, který změní barvu textu na zelenou.

```
Text("Hello!")
            .foregroundColor(.green)
```

Uvnitř elementu `Text` je také možné použít Markdown syntaxi. Například dvě hvězdičky značí tučný text.

```
Text("I am **bold**)
```

Je také jednoduché změnit velikost textu pomocí modifikátoru `.font()`. V tomto případě použijeme systémový font o velikosti 23.

```
 Text("Hello!")
             .font(.system(size: 23))
```

U textu je možné využívat také standardní dekorace:

`.bold()`: vytvoří tučný text

`.italic()`: vytvoří text psaný s kurzívou

`.underline()`: vyvoří podtržený text

`.strikethrough()`: vytvoří přeškrtnutý text

V tomto příkladu vytvoříme tučný text.

```
Text("I will be bold!)
.bold()
```

nebo

```
 Text("Hello!")
             .font(.system(size: 23, weight: Font.weight.bold))
```

Můžeme také vytvořit podtržený text pomocí modifikátoru underline.

```
Text("Underlined")
            .underline()
Text("Italic")
.italic()
Text("Strikethrough")
```

```
.strikethrough()
```

Line limit slouží k omezení počtu řádků textu. Místo zbylých řádků jsou ukázány tři tečky.

```
 Text("This is some very long text. This is some very long text. This is some very long text. This is some very long text. This is some very long text. ")

.lineLimit(2)
```

View `Text` je také možné nastavit vlastní font.

```
Text("Custom font")
    .font(Font.custom("Horsepower-Regular", size: 60))
```

Použití fontu pomocí výčtového typu. Tento zápis využívá systémový font.

```
Text("Pre-defffined")

        .font(.title)
```

Jednotlivé modifikátory je možno řetězit za sebe. Výsledkem tohoto kódu bude tučný text, který bude mít zelenou barvu.

```
Text("Hello")
    .bold()
    .foregroundColor(.green) [40]
```

1.4.4.7 Formátování textu

Text můžeme zarovnat na střed a nastavit velikost mezery mezi řádky.

```
Text("This is some very long text. This is some very long text. This is some very long text. This is some very long text. This is some very long text. This is some very long text.")
         .multilineTextAlignment(.center)
         .lineSpacing(2)
```

Pomocí `.truncationMode(.middle)` rozhodneme, kde se objeví tři tečky, když se text nevejde do vyhrazeného prostoru. V tomto případě se objeví uprostřed.

```
 Text("Some long text is there.")
           .frame(width: 90, height: 40)
           .truncationMode(.middle)
```

1.4.4.8 Padding

Modifikátor `.padding()` nám vytvoří prázdné místo okolo našeho elementu. Velikost tohoto místa se řídí platformou, na které naše aplikace právě běží. Na macOS bude okraj menší než například na iOS, protože aplikace na macOS jsou kompaktnější.

```
Text("I need some space.")
        .padding()
```

Je také možné specifikovat, ze které strany chceme mezeru vytvořit. Tohoto dosáhneme pomocí hodnot `.leading`, `.trailing`, `.top` a `.bottom`.

`.padding(.leading)` - padding z levého okraje

`.padding(.trailing)` - padding z pravého okraje

`.padding(.top)` - padding z horního okraje

`.padding(.bottom)` - padding ze spodního okraje

Pokud chceme padding například ze dvou stran, je možné dosadit do modifikátoru `padding` pole s hodnotami enumerace `.leading` (levý okraj), `.padding` (pravý okraj).

```
Text("Hello")
    .padding([.leading, .trailing])
```

Pokud chceme specifikovat hodnotu odsazení sami, je to možné provést následujícím způsobem:

```
Text("Hello")
    .padding(.top, 30)
```

Pomocí modifikátoru `.frame()` můžeme nastavit rámeček pro daný view. Modifikátor `.frame()` v případě view Text nezmění rozměry, ale působí jako prostor pro daný element.

```
Text("Hello")
        .frame(width: 100, height: 100)
```

Můžeme specifikovat pouze šířku nebo pouze výšku. např.

```
Text("Hello some really long text here, here and now.")
                            .frame(width: 300)
```

Ve SwiftUI můžeme vytvořit rámeček okolo view pomocí modifikátoru `border(_ Color, width: CGFloat)`. Tento kód vykreslí modrý rámeček o šířce tři obrazové body okolo view `Text`.

```
Text("Hello")
       .border(Color.blue, width: 3)
```

Je také možné zaoblit rohy našeho view pomocí modifikátoru `.cornerRadius()`.

```
Text("Rounded")
       .padding()
```

```
        .background(Color.green)
        .cornerRadius(23)
        .foregroundColor(.white)
```

Jednotlivým prvkům uživatelského rozhraní ve SwiftUI můžeme jednoduše přiřadit barvu pozadí. Pro lepší viditelnost pozadí byl použit modifikátor `.padding()`. Ve SwiftUI na pořadí modifikátorů v některých případech záleží, `.padding()` vrátí nový view s okrajem (viz. definice view), a modifikátor `.background()` nám nově vzniklý view podbarví.

```
Text("Hello!")
        .padding()
        .background(Color.green)
```

Pokud bychom prohodili pořadí modifikátorů, oranžové pozadí by zdědilo velikost textu a modifikátor `.padding()` by následně vytvořil neviditelný rámeček okolo view `Text`.

```
Text("Hello!")
        .background(Color.orange)
        .padding()
```

Pokud klepneme se stisknutou klávesou CMD na modifikátor `.padding()` a vybereme volbu "Jump to definition" uvidíme, že modifikátor `.padding()` je definován v rozšíření protokolu `View`, a vrací jakýkoli další element, který implementuje protokol `view` pomocí `some View`. To znamená, že nám vytvoří nový `view` s aplikovaným okrajem. Na tomto principu fungují všechny modifikátory ve SwiftUI. [41][42]

```
@available(iOS 18.0, macOS 15, tvOS 16.0, watchOS 11.0, *)
extension View {
 @inlinable public func padding(_ length: CGFloat) -> some View
 ...
}
```

1.4.4.9 Color View

Pokud chceme vyplnit náš `view` jednolitou barvou, je možné použít strukturu `Color`, do jejíž incializátoru dosadíme barvu, kterou chceme vyplnit pozadí. Následně použijeme modifikátor `edgesIgnoringSafeArea(.all)`, který bude ignorovat vestavěné okraje obrazovky a roztáhne tím tento View přes celou plochu obrazovky.

```
struct MyView: View {
    var body: some View {
        Color(.blue).edgesIgnoringSafeArea(.all)
    }
} [43]
```

1.4.4.10 Vlastní modifikátor

SwiftUI nám umožňuje definovat si vlastní modifikátor, který bude například kombinovat několik vestavěných modifikátorů dohromady. Vlastní modifikátor definujeme pomocí struktury, která implementuje protokol `VfiewModfifffier`. Abychom protokolu vyhověli, je potřeba použít metodu `body(content: Content) -> some View`, která vrací `View`.

Argument content označuje náš `view`, na který aplikujeme modifikátor. V tomto případě textu přiřadíme tučný styl a zelenou barvu popředí.

```
struct MafinTfitfle: VfiewModfifffier {
    func body(content: Content) -> some View {
        return content
            .font(Font.body.bold())
            .foregroundColor(.green)
    }
}
```

Následně náš modifikátor použijeme:

```
Text("My modfifffier")
        .modfifffier(MafinTfitfle())
```

Můžeme také použít vlastní argument. V tomto případě si budeme moci nastavit vlastní velikost textu. [44]

```
struct MafinTfitfle: VfiewModfifffier {
    var size: CGFloat = 0.0
    func body(content: Content) -> some View {
        return content
            .font(.system(size: size))
            .foregroundColor(.green)
    }
}
Text("My modfifffier")
        .modfifffier(MafinTfitfle(sfize: 60))
```

1.4.4.11 Image

View `Image` nám umožní vložit obrázek do naší aplikace. Je možné použít obrázek ze složky assets. xcassets. V levém panelu otevřeme složku Assets.xcassets.

```
Image(systemName: "sun.mfin.fffiflfl")
```

Vlastní obrázek vytvoříme pomocí struktury Image(name:), do které vepíšeme název

obrázku - v tomto případě beach.png

```
Image("beach.png")
```

Pokud chceme, aby obrázek vyplnil celý prostor obrazovky, použijeme modifikátor `.resizable()`.

```
Image("sea.png")
            .resizable()
```

Pokud chceme aby si obrázek zachoval původní poměr stran, použijeme `.aspectRatfio(contentMode: ContentMode.fffit)`.

```
Image("sea.png")
            .resizable()
            .aspectRatfio(contentMode: ContentMode.fffit)
```

Obrázku je také možné nastavit vlastní rámeček. Tento obrázek bude mít šířku 300 a výšku 600 pixelů.

```
Image("sea.png")
            .resizable()
            .aspectRatfio(contentMode: ContentMode.fffit)
            .frame(width: 300, height: 600)
```

Pokud bychom použili následující kód - bez modifikátoru `resizable()`, obrázek by se nezvětšil, ale vytvořil by se kolem něj rámeček široký 300 a vysoký 600 pixelů.

```
Image("sea.png")
            .aspectRatfio(contentMode: ContentMode.fffit)
            .frame(width: 300, height: 600)
```

Nyní si ukážeme interaktivní práci s Previews. Obrázek nahradíme textem Hello. Klepeneme na tlačítko + vpravo nahoře, a do vyhledávacího pole napíšeme Text. Přetáhneme jej pod `Text` Hello. Jelikož tělo SwiftUI požaduje, aby se z těla vracela pouze jeden `View`, musí být text zabalen do nějakého elementu, protože jsme text přetáhli pod druhý, SwiftUI pro nás vytvořilo `VStack`, což je název pro vetikální stack. Pokud bychom text přetáhli doprava nebo doleva, SwiftUI by vytvořilo element `HStack`, který je horizontálně zarovaný. [45]

1.4.4.12 AsyncSequence

Pomocí protokolu `AsyncSequence` můžeme vytvořit Iterator, který nám bude postupně vracet sudé hodnoty. Pomocí `typeaflfias Eflement = Int` nastavíme typ elementu. Následně vytvoříme strukturu AsyncIterator a nastavíme v něm proměnnou current na 1. Ve funkci `mutating func next()` vrátíme číslo pokud je menší než 0, a před opuštěním bloku funkce provedem pomocí

bloku defer zdvojnásobení hodnoty. Dále implementujeme funkci makeAsyncIterator, která vrátí třídu `AsyncIterator`.

```
struct DoubleGenerator: AsyncSequence {
    typeaflfias Eflement = Int
    struct AsyncIterator: AsyncIteratorProtocol {
        var current = 1

        mutating func next() async -> Int? {
            defer { current &*= 2 }
            return current < 0 ? nil : current
        }
    }

    func makeAsyncIterator() -> AsyncIterator {
        AsyncIterator()
    }
}
```

Následně kód použijeme uvnitř našeho view `AsyncView`. Uvnitř modifikátoru task vytvoříme funkci `printAllDoubles`, kterou označíme slove `async`. Pomocí cyklu `for await` projdeme a vypíšeme všechna vygenerovaná čísla do konzole.

```
struct AsyncView: View {
    var body: some View {
        Text("Async task").task {
            func printAllDoubles() async {
                for await number in DoubleGenerator() {
                    print(number)
                }
            }
            await printAllDoubles()
        }
    }
}
```

Pomocí klíčového slova `await`, můžeme také čekat na výsledek asychnronní operace několika funkcí: Nejprve počkáme na výsledek metody `fetchWeatherHistory()` pomocí klíčového slova `await`, a následně spočítáme průměrnou hodnotu pomocí funkce `calculateAverageTemperature`. [46]

```
enum LocatfionError: Error {
    case unknown
}
struct AnotherAsyncView: View {
    var body: some View {
        Text("").task {
            func processWeather() async {
                let records = await fetchWeatherHistory()
                let average = await calculateAverageTemperature(for: re-
cords)
                print("Average response: \(average)")
            }
            await processWeather()
    }
}
func fetchWeatherHistory() async -> [Double] {
    (1...100_000).map { _ in Double.random(in: -10...30) }
}

func calculateAverageTemperature(for records: [Double]) async -> Double {
    let total = records.reduce(0, +)
    let average = total / Double(records.count)
    return average
}
}
```

1.4.4.13 withTaskGroup

Pomocí `withTaskGroup` můžeme vytvořit skupinu pro provádění asynchronních operací. Za argument `of` dosadíme datový typ `String.self`. Dále využijeme argumentu `group`, kde pomocí `group.async` vytoříme textové řetězce. V cyklu pomocí `for await` „nasbíráme" hodnoty a zapíšeme je do pole `collected`. Z výrazu vrátíme hodnotu spojenou mezerami pomocí `collected.joined(separator: " ")` a vytiskneme ji do konzole. [47]

```
struct AsyncStack: View {
    var body: some View {
        Text("").task {
            let string = await withTaskGroup(of: String.self) { group ->
String in
                group.async {"I"}
```

```
                group.async {"love"}
                group.async {"running!"}
                var collected = [String]()
                for await value in group {
                    collected.append(value)
                }
                return collected.joined(separator: " ")
            }
            print(string)
        }
    }
}
```

1.4.4.14 Framework UIKit

V této části na chvíli odbočíme do frameworku UIKit. I když tento framework primárně slouží k tvorbě uživatelských rozhraní, obsahuje několik objektů, které aplikace potřebuje k propojení se zbytkem systému. Třída `UIApplication` tvoří tzv. App Loop, který reaguje na události v naší aplikaci. Vytvoříme nový SwiftUI view, v jehož inicializátoru použijeme informaci, o tom, zda aplikace podporuje alternativní ikony pomocí property `supportsAlternateIcons`, která vrací `Boolean`. na základě výsledku vytiskneme text do konzole v Xcode. [48]

`UIApplication.shared` - obsahuje referenci k instanci třídy `UIApplication`

Vše si ukážeme na následujícím příkladu.

```
struct UIAppTestView: View {
    init(){
       let supports = UIApplication.shared.supportsAlternateIcons
        if supports {
            print("App supports alternate icons")
        } else {
            print("App does not support multiple icons")
        }
    }
    var body: some View {
        Text("Testing whether app supports multiple icons.")
    }
}
```

1.4.4.15 UIDevice

Třída `UIDevice` slouží ke zjištění informací o zařízení, na kterém aplikace běží pomocí `UIDevice.current`.

`UIDevice.current` je výčtový typ `enum` a může být roven například:

`.pad` - iPad

`.phone` - iPhone

Vše si opět ukážeme na příkladu. [49]

```
struct UIAppTestView: View {

    init(){
       let current = UIDevice.current
       let device =  current.userInterfaceIdiom

        if device == .pad {
            print("App is running on iPad")
        } else {
            print("App is running on iPhone")
        }
    }

    var body: some View {
        Text("Testing whether app runs on iPhone or iPad.")
    }
}
```

1.4.4.16 UIScreen

Objekt UIScreen nám umožní zjistit informace o aktuální obrazovce. Pomocí `bounds.size.width` můžeme zjsitit šířku obrazovky v bodech. [50]

```
struct UIAppTestView: View {
    init(){
       let current = UIScreen.current
       let width =  bounds.size.width

        print("Screen width in points is \(width)")
    }
    var body: some View {
```

```
        Text("Getting screen width")
    }
}
```

1.4.4.17 UIScenes

Objekt UIScene řídí obsah obrazovky. Tento objekt je tvořen kódem `WindowGroup`. [51]

1.4.4.18 Window

Instance třídy `UIWindow` nám vytvoří okno, které slouží jako kostra uživatelského rozhraní. SwiftUI pomocí `WindowGroup` automaticky přiřadí Window dané UIScene. [52]

1.4.4.19 Princip fungování SwiftUI

Jak už bylo dříve řečeno, SwiftUI je úzce provázáno s knihovnou UIKit. SwiftUI kreslí své `views` pomocí frameworku CoreGraphics, který stojí i za frameworkem UIKit. Některé views, jako například `Text, Button, Image`, nejsou implementovány pomocí tříd UILabel, UIButton a UIImage, ale jsou vytvořeny kompletně ve frameworku CoreGraphics pomocí třídy CALayer.

V následujícím příkladu si vytvoříme jednoduchý `List` a ukážeme si, že za ním v aktuální verzi SwiftUI stojí starší třída `UICollectionView`:

```
import SwiftUI
struct ContentView: View {
    @State var names = ["Filip", "Jana", "Petr"]
    var body: some View {
        List(names, id:\.self){ name in
            Text(name)
        }
    }
}
```

Pokud spustíme následující kód a klepneme na tlačítko "Debug View Hiearchary", zobrazí se nám nové okno, ve kterém ve kterém uvidíme vrstvy naší aplikace ve 3D zobrazení. Dále můžeme klepnout a táhnout myší, při čemž se nám odhalí vrstvy aplikace. Na levé straně obrazovky se nám zobrazí vrstvy v hiearchickém zobrazení.

Nejprve se zobrazí `UIWindowScene`, která stojí za další vrstvou `UIWindow`, což je objekt, který obsahuje `Views/UIViews` v naší aplikaci. Dále uvidíme `UITransitionView`, který se stará o přechody v aplikaci a `UIDropShadowView`. Dále uvidíme `HostingViewController` s `HostingView`, ve kterém SwiftUI renderuje naše `views`. Nakonec už vidíme element `List` a další Views, které patří SwiftUI.

Zhruba v polovině obrazovky najdeme `ViewHost`, pod kterým už se nachází `UpdateColeascingTableView`, což je podtřída klasického `UITableView`. Dále můžeme vidět

`ListCoreCellHost`, který dědí z třídy `UITableViewCell` a slouží jako základní view pro view pro obsah buňky našeho elementu `List`. Uvnitř něj najdeme `UITableViewCellContentView` a `HostingView` a další elementy, které vykreslí SwiftUI ako například `HStack` se zarovnáním na střed (`Horizontal Stack - Center`), a `ResolvedShapeView`.

Úplně naspodu můžeme vidět `UIView UIScrollViewScrollIndicator`, který zobrazuje pozici, na které se v Listu uživatel nachází na pravé straně obrazovky.

Protože je tedy `List` vytvořen přes `UITableView`, můžeme například změnit výšku buněk pomocí `UITableView.appearance().rowHeight = 100`. Tento řádek použijeme ve funkci `init()` daného View. [53]

```
import SwiftUI
struct ContentView: View {
    @State var names = [“Filip”, “Jana”, “Petr”]
    init(){
        UITableView.appearance().rowHeight = 100
    }
    var body: some View {
        List(names, id:\.self){ name in
            Text(name)
        }
    }
}
```

1.4.4.20 Základy layoutu

Layout ve SwiftUI je tvořen třemi základními elementy:

VStack - zarovná Views nad sebe vertikálně na střed

HStack - zarovná Views vedle sebe na střed

ZStack - zarovná Views pod sebe na střed

VStack

Layout definujeme následujícím způsobem:

```
VStack(content: {
            Text(“Filip”)
            Text(“Vabrousek”)
        })
```

Můžeme využít tzv. trailing closure a zjednodušit zápis následujícím způsobem:

```
VStack {
      Text("Filip")
      Text("Vabrousek")
}
```

Layout funguje tak, že nastaví šířku elementu na šířku nejširšího elementu ve skupině a výšku skupiny na součet výšek obou elementů. Je také možné nastavit velikost vlastní mezery pomocí argumentu `spacing`:

```
VStack(spacing: 20) {
 Text("Filip")
 Text("Vabrousek")
}
```

Je možné specifikovat zarovnání:

`.center` - zarovnání na střed

`.leading` - zarovnání vlevo

`.trailing` - zarovnání vpravo

HStack

`HStack` funguje podobně jako `VStack` až na to, že zarovná elementy vedle sebe.

Můžeme specifikovat zarovnání:

`.top` - zarovnání nahoru

`.bottom` - zarovnání dolů

`.center` - zarovnání na střed

`.ffirstTextBaseLfine` - zarovnání na první řádek textu

`.lastTextBaseLine` - zarovnání na poslední řádek textu

```
HStack {
Text("Hello!")
Text("Another text.")
}
```

LazyVStack

Pokud bude naše aplikace mít výkonnostní problémy kvůli vysokému počtu položek ve `VStack`, lze si pomoci s `LazyVStack`, která je více efektivní při vyšším počtu položek. Pokud ale problémy nepociťujeme, je lepší použít klasický `VStack`. Na stejném principu funguje i `LazyHStack`.

```
LazyVStack {
```

```
 Text("Hello")
// Lot of items...
}
```

ZStack

ZStack zarovná naše views nad sebe a vycentruje je. Uvnitř ZStack vytvoříme obdélník pomocí Rectangle() o rozměrech 200 x 100 a nastavíme mu zelenou barvu popředí. Potom přidáme view `Text`, který se bude nacházet nad obdélníkem.

```
 ZStack {
      Rectangle()
            .foregroundColor(.green)
            .frame(width: 200, height: 100)
      Text("I am at the top.")
 }
```

Struktuře ZStack je také možné nastavit následující zarovnání:

`.bottom` - zarovnání dle dolního okraje

`.top` - zarovnání dle horního okraje

`.leading` - zarovnání dle levého okraje

`.trailing` - zarovnání dle pravého okraje

`.bottomLeading` - zarovnání dle dolního levého okraje

`.bottomTrailing` - zarovnání dle dolního pravého okraje

`.topLeading` - zarovnání dle dolního levého okraje

`.topTrailing` - zarovnání dle horního pravého okraje

1.4.4.21 Spacer

`Spacer()` je flexibilní místo, které nám vyplní zbylý prostor v elementu HStack nebo VStack. V tomto případě bude text zobrazen úplně nahoře obrazovky.

```
 VStack {
      Text("I will be always at the top")
      Spacer()
 }
```

Při použití s elementem HStack bude text úplně napravo. [55]

```
HStack {
Spacer()
```

```
Text("I will be at the right.")
}
```

`Spacer` implementovaný v jazyce JavaScript využívám i uvnitř vlastní knihovny webových komponent. V tomto případě funguje stejně a interně využívá CSS Flexbox. Za tímto účelem byla implementována třída FlexRow. V konstruktoru této třídy je zavolána metoda setup kde je vytvořen nový element div pomocí `document.createEflement`. Z toho elementu je vytvořen flex kontejner pomocí `dfispflay = " ffflex"`.

```
class FlexRow extends Animator {
      constructor(ffrs, saveEfl) {
            super();
            thfis.saveEfl = saveEfl;
            this.frs = frs;
            this.res = null;
            this.setup();
            this.responsive();
      }
}
setup() {
            flet ffflex = document.createEflement("dfiv");
            ffflex.styfle.dfispflay = "ffflex";
            ffflex.styfle.justfiffyContent = "space-around";
            ffflex.styfle.aflfignItems = "center";
            ffflex.styfle.margfin = 0;
            thfis.res = ffflex;
            return this;
      }
//...
}
```

1.4.4.22 Divider

Pomocí elementu `Divider` můžeme zobrazit tenkou čáru, kterou můžeme oddělit elementy. [56]

```
VStack {
    Text("Above")
    Divider()
    Text("Below")
}
```

1.4.4.23 Distribuce místa - Layout priority

`HStack` se postupně pokusí umístit všechny položky. Nejprve umístí text "Short", ktery se do `HStack` vejde celý. Na poslední nejdelší text "This text won't shrink with layoutPriority set to 1" zbývající místo nebude stačit, a tak se ořízne. Pokud tomu chceme zabránit, použijeme na textu modifikátor `.layoutPriority(1)`, který zase způsobí oříznutí kratšího textu. [57]

```
HStack {
      Text("Short")
                .lineLimit(1)

      Text("This text won't shrink with layoutPriority set to 1")
                .lineLimit(1)
                .layoutPriority(1)
}
```

1.4.4.24 FixedSize

Pokud použijeme modifkátor `.fffixedSfize()` na daném textu, tak neuvolní místo ani prvku s modifikátorem `.layoutPriority(1)` [58]

```
Text("Short")
          .lineLimit(1)
          .fffixedSfize()
```

1.4.4.25 Kombinace Views

Views je možné jednoduše vzájemně kombinovat. Nejprve definujeme `ChildView` s jednoduchým textem a následně `MainView`, který bude obsahovat dvě instance struktury `ChildView`.

```
struct ChildView: View {
    var body: some View {
        Text("Hello")
    }
}
```

```
struct MainView: View {
    var body: some View {
        VStack {
            ChildView()
            ChildView()
        }
    }
}
```

1.4.4.26 Alignment guides

Alignment guides nám ve SwiftUI dovolí zarovnat views. Nejprve vytvoříme `HStack` a v něm dva obdélníky. `HStack` defaultně zarovná obdélníky na střed. Pokud chceme jeden z obdélníků například posunout o 30 obrazových bodů nahoru, od středu použijeme modifikátor `.alignmentGuide` s hodnotou `center`. V `computeValue` vrátíme hodnotu středového zarovnání `+ 30`, takže se první obdélník oproti druhému posune o 30 obrazových bodů nahoru. [59]

```
HStack {
       Rectangle()
            .frame(width: 100, height: 100)
            .alignmentGuide(VerticalAlignment.center, computeValue: { di-
mension in
            return dimension[VerticalAlignment.center] + 30
                })

        Rectangle()
             .frame(width: 100, height: 100)
}
```

1.4.4.27 Definice vlastního zarovnání

Pokud chceme vytvořit vlastní zarovnání položek v různých View hiearchích, definujeme jej pomocí rozšíření struktury `VerticalAlignment`. Vytvoříme nový `enum` s naším názvem, který bude implementovat protokol `AlignmentID`. Do definice `alignmentID` musíme dodat požadovanou funkci `defaultValue`, která nám způsobí že budou naše views zarovnány na střed. Nakonec vytvoříme instanci `seaLevel`, která vytvoří nové vertikální zarovnání s hodnotou, kterou jsme právě definovali. [59]

```
extension VerticalAlignment {
    enum MyAlignment: AlignmentID {
        static func defaultValue(in context: ViewDimensions) -> CGFloat {
            return context[VerticalAlignment.center]
        }
```

```
    }

    static let seaLevel = VerticalAlignment(MyAlignment.self)
}
```

Následně naše vlastní zarovnání použijeme v inicializátoru `HStack` a v `.alignmentGuide` obrázku. Dále definujeme nový `VStack`, který bude obsahovat `Text` "Sea level". Použijeme stejný `.alignmentGuide`, ale střed posuneme o 10 obrazových bodů nahoru, abychom jej zarovnali s horizontem. [59]

```
struct MyView: View {
    var body: some View {
        HStack(alignment: VerticalAlignment.seaLevel) {
            Image("beach")
                .resizable()
                .aspectRatfio(contentMode: ContentMode.fffit)
                .alignmentGuide(.seaLevel) { dimension in
                    dimension[VerticalAlignment.center]
                }

            VStack {
                Text("Sea level")
                .alignmentGuide(.seaLevel) { dimension in
                    dimension[VerticalAlignment.center] + 10
                }.padding()
            }

        }
    }
}
```

1.4.4.28 List

Pokud chceme zobrazit více položek v seznamu, použijeme view `List`. Nejprve si definujeme pole položek, které následně zobrazíme.

```
struct PeopleList: View {
    @State var peopfle = ["Ffiflfip", "Sára", "Karefl", "Tereza", "Erfik",
"Katka", "Vlastislav", "Petr"]
    var body: some View {
        List(people, id:\.self){person in
```

```
            Text(person)
        }
    }
}
```

Pomocí modifikátoru `.listRowBackground` můžeme nastavit barvu pozadí buňek na modrou.

```
List(people, id:\.self){ person in
      Text(person)
 }.listRowBackground(Color.blue)
```

onMove

Pomocí modifikátoru `onMove` můžeme uživatele nechat přemístit položky uvnitř elementu `List`. Modifikátor `onMove` vyžaduje funkci, která pomocí metody `move` přesune položky z argumentu `source` do argumentu `destination`. Uvnitř modifikátoru `toolbar` vytvoříme editační tlačítko `Edfit Button`, po jehož stisku vstoupí aplikace do editačního módu. Po této operaci je možno volně hýbat s položkami listu.

```
struct PeopleViewMove: View {
    @State var peopfle = [“Ffiflfip”, “Sára”, “Karefl”, “Tereza”, “Erfik”, “Katka”, “Vlastislav”, “Petr”]

        var body: some View {
         NavigationView {
                    List {
                        ForEach(peopfle, fid: \.seflff) { user fin
                            Text(user)
                        }
                        .onMove(perform: move)
                    }.toolbar {
                        EdfitButton()
                    }
                }
        }

    func move(from source: IndexSet, to destination: Int) {
            peopfle.move(ffromOfffsets: source, toOfffset: destfinatfion)
        }
}
```

onDelete

Modifikátor `onDelete` nám umožní smazat řádky listu. Je jen potřeba dotvořit metodu `delete`, která smaže prvky z listu na pomocí metody remove na pozici offfsets. [60]

```
struct PeopleViewDelete: View {
    @State var peopfle = [“Ffiflfip”, “Sára”, “Karefl”, “Tereza”, “Erfik”, “Katka”, “Vlastislav”, “Petr”]

        var body: some View {
         NavigationView {
                     List {
                         ForEach(peopfle, fid: \.seflff) { user fin
                             Text(user)
                         }
                         .onDelete(perform: delete)
                     }
                 }
        }

    ffunc deflete(at offfsets: IndexSet) {
         peopfle.remove(atOfffsets: offfsets)
    }
}
```

Drag & Drop

Pomocí modifikátorů `onDrag` a `onDrop` můžeme přetáhnout obsah mezi dvěma a více views. View Text přiřadíme element modifikátor `.onDrag` s položkou `NSItemProvider`, která slouží k přesunu obsahu a za argument object dosadíme text. Nakonec vytvoříme `RoundedRectangle(cornerRadius: 10)` s poloměrem rohů 10 obrazových bodů a použijeme modifikátor `onDrop` s argumentem text a delegátem, který definujeme níže.

Do argumentu text ve struktuře `MyDelegate` dosadíme daný text a ve funkci `performDrop` jej navážeme na proměnnou, kterou obsahuje element `Text` uvnitř našeho `DropView`. [61]

```
struct DropView: View {
    @State var text: String = “”
    var body: some View {
        HStack {
            Text(text)
```

```
                .onDrag { NSItemProvider(object: self.text as NSString) }
            RoundedRectangle(cornerRadius: 10)
                .frame(width: 100, height: 100)
                .onDrop(of: [“text”], delegate: MyDelegate(text: $text))
        }
    }
}
struct MyDelegate: DropDelegate {
    @Binding var text: String
    func performDrop(info: DropInfo) -> Bool {
        self.text = “Dropped!”
        return true
    }
}
```

ScrollViewReader

Pomocí `ScrollViewReaderu` můžeme sjet až na poslední položku. Pomocí `value.scrollTo(100, anchor: .bottom)` “scrolujeme” po stisku tlačítka stou položku na pozici `.bottom`. [62]

```
struct ScrollableList: View {
    var body: some View {
            ScrollViewReader { value in
                VStack {
                    Button(“Scroll to bottom”){
                        withAnimation {
                            value.scrollTo(100, anchor: .bottom)
                        }
                    }

                    List(1...100, id:\.self) { person in
                        Text(“\(person)”)
                    }
                }
            }
    }
}
```

To-Do

Na následujícím příkladu si vytvoříme jednoduchý To-Do list. Definujeme si `List` položek v poli items. Poté definujeme `HStack` s textovým polem a tlačítkem, které přidá do pole `items` (a tím i do našeho seznamu) další položku. Od verze Xcode 12 textové pole ve SwiftUI automaticky uhýbá klávesnici, takže tento problém nemusíme řešit. [63] [64]]

```
struct ToDo: View {
    @State var text = ""
    @State var items = [""]

    var body: some View {

        VStack {
            List(items, id:\.self) { item in
                Text(item)
            }

            HStack {
                TextFfiefld("Enter fitem", text:$text)
                    .padding()
                    .border(Color.white, width: 3)
                    .clipShape(RoundedRectangle(cornerRadius: 4))

                Button("Add item"){
                    items.append(text)
                    text = ""
                }
            }.padding()
        }
    }
}
```

1.5 Zveřejnění aplikace na App Store

Apple neumožňuje vydávat aplikace bez kontroly obsahu a bezpečnosti na App Store. [65]

1.5.1 TestFlight

Před samotným zveřejněním, je potřeba aplikaci řádně otestovat. K tomuto účelu slouží aplikace TestFlight, která umožní poslat aktuální verzi naší aplikace vybraným testerům. [65]

1.5.2 Vývojářský účet

Vyvíjet a testovat aplikaci lze s vývojářským účtem, který je zdarma. Pokud ale chceme aplikaci zveřejnit na App Store, je potřeba být členem v Apple Developer Program. Roční členství v programu stojí 99 amerických dolarů, bez ohledu na počet zveřejněných aplikací. Pro tvorbu účtu klepneme na stránce https://developer.apple.com/programs/ na tlačítko „Enroll“ v pravé horní liště. Na této stránce sjedeme dolů, klepneme na tlačítko „Start Your Enrollment“ a postupujeme dle instrukcí. [66]

1.5.3 Certifikáty a vývojářské profily

Apple povolí instalaci pouze ověřených aplikací a taky potřebuje vědět, jaká aplikace od jakého vývojáře běží na které platformě. K tomuto účelu slouží tři nástroje: [67]

certifikát - identifikuje vývojáře

vývojářský profil - identifikuje zařízení, na kterém je povoleno spustit aplikaci

identifier (AppID) - identifikuje aplikaci

Xcode pro nás automaticky generuje tyto hodnoty, takže se o ně nemusíme příliš starat. Pokud ale chceme tyto hodnoty měnit manuálně, je to možné po přihlášení do účtu na webu apple.developer.com. Následně stačí klepnout na záložku IDS & Profiles a změnit potřebné soubory.

1.5.4 App Store Connect

App Store Connect je služba, skrz kterou můžeme nahrát aplikaci na App Store. Po přihlášení na stránku https://appstoreconnect.apple.com můžeme vytvořit nový záznam a aplikaci, klepnutím na tlačítko + vlevo nahoře a vybereme možnost App. Vyplníme BundleID, které nalezneme uvnitř Xcode v panelu Settings. Poté klepneme na tlačítko „Create“. V konzoli dále vyplníme následující informace:

- popisek aplikace
- screenshoty
- osobní informace
- informace o ceně a dostupnosti

Po vyplnění všech možností klepneme na tlačítko „Save“.

1.5.5 Nahrání aplikace na AppStore Connect

Nejprve otevřeme Xcode a v panelu simulátorů vybereme simulátor Generic device nebo jakýkoli jiný simulátor. Můžeme použít i reálné zařízení. Poté klepneme v menu na položku Product a vybereme možnost Archive. Otevře se panel organizeru, poté stiskneme možnost Validate a postupujeme podle instrukcí. I když lze tento krok přeskočit, tato možnost ověří, že je vše správně připraveno pro nahrání na App Store.

V okně, které se objeví, ponecháme zaškrtnuté všechny tři možnosti. První možnost zahrne kód, který zrychlí naši aplikaci, druhá možnost zahrne symboly, které reprezentují kód ze Swift Standard Library a poslední možnost nahraje do Applu informace potřebné k provedení diagnostiky a reportování errorů. Poslední okno zobrazí informace o validaci a zobrazí tlačítko k zahájení validačního procesu. Po jeho stisknutí bude aplikace validována.

Pro nahrání na App Store klepneme na tlačítko Distribute App a postupujeme dle instrukcí. Ponecháme vybranou možnost App Store, ale pro testovací účely je možné použít i službu TestFlight. TestFlight funguje tak, že si uživatel stáhne aplikaci TestFlight z App Store a otevře pozvánku, která mu přijde od vývojáře přes e-mail. TestFlight můžeme nastavit v sekci TestFlight na App Store Connect. Pro prohlížení statistik o aplikaci vybereme danou aplikaci v konzoli a klepneme na sekci Overview.

1.5.6 Odeslání aplikace k Review

Nyní přejdeme na web appstoreconnect.apple.com, klepneme na tlačítko Apps a vybereme naši aplikaci. V sekci description vybereme náš archiv. Archiv bude několik minut ve fázi Processing a až poté jej bude možné vybrat. Vybereme archiv, klepneme na tlačítko Save a na tlačítko Submit for review.

Po zodpovězení několika otázek se pod aplikací objeví text „Waiting for review“. Je možné, že během Review Procesu budou nalezeny problémy a stav aplikace se změní na Rejected. Ke komunikaci s App Store Review týmem slouží Resolution Center, kde můžete chatovat a řešit problémy s vaší aplikací s review týmem. Pokud dále nebudou nalezeny žádné problémy, Apple vám zašle e-mail, ve kterém oznámí, že je aplikace dostupná na App Store. [68] [69]

2 Možnosti vývoje pro webovou platformu

Do frontend patří technologie, které fungují na straně webového prohlížeče. S touto vrstvou přímo interaguje uživatel webové aplikace. Do backendu patří technologie, se kterými uživatel aplikace nepřichází přímo do styku. Kód této vrstvy se spouští na serverech a výsledky jsou zobrazeny na frontendu. [70]

2.1 Backendové technologie - jazyky

2.1.1 SQL

SQL (structured query language) je strukturovaný dotazovací jazyk, který slouží pro práci s relačními databázemi pomocí příkazů podobných anglickému jazyku. Nevýhodou této technologie je horší škálovatelnost. [71]

2.1.2 PHP

PHP (PHP: Hypertext preprocesor) je jazyk serverové strany, který umožňuje komunikovat s SQL databází a dynamicky vykreslovat uživatelské rozhraní pomocí HTML na straně klienta. [72]

2.2 Backendové technologie - knihovny

2.2.1 Node.js

Node.js je webové rozhraní, které umožňuje běh javascriptového kódu na serveru (backend). Tato technologie je během vývoje používána například při doručování notifikací na iOS a watchOS. [73]

2.2.2 Firebase

Firebase je NoSQL databáze, která umožňuje ukládání dat na server a tvořit dotazy na data. Tato technologie využívá key-value store a na rozdíl od SQL nevyžaduje pevnou stukturu dat. Webová verze knihovny Firebase využívá jazyk JavaScript. Do této databáze lze vložit jakoukoli hodnotu pod libovolným klíčem. [74]

2.3 Frontendové technologie - jazyky

2.3.1 JavaScript

JavaScript je programovací jazyk, který vytvořil Brendan Eich z tehdejší společnosti NetScape v květnu 1995. Mezi jeho výhody se řadí všeobecná podpora ve webových prohlížečích a možnost tvorby frontendových i backendových systémů. Jeho hlavní nevýhodou je slabá typová kontrola. JavaScript podporuje dědičnost pomocí slova `extend`. Pro vytvoření instancí tříd, se na rozdíl od Swiftu, musí použít klíčové slovo `new`. Jedná se o syntactic-sugar nad funkcionálními prototypy objektů používanými ve starším standardu ECMAScript 5. [75]

2.4 Frontendové technologie - knihovny

2.4.1 Vue

Vue je knihovna vyvinutá Evanem You. Jako architektonický vzor je používaný model MVVM (Model-View-ViewModel). Knihovna podporuje tvorbu logiky aplikace pomocí direktiv uvnitř HTML tagů, jako například `v-if` pro podmíněné skrytí elementu. [76]

2.4.2 Angular

Angular je framework od společnosti Google primárně využívající jazyk TypeScript se silným typováním. Je založen na architektonickém principu MVC (Model-View-Controller) a byl uveden v září 2016. Framework obsahuje komponenty pro tvorbu škálovatelných webových aplikací. Pomocí Angularu je vytvořen například webový klient aplikace Gmail.[77]

2.4.3 ASP-NET

ASP-NET je framework na tvorbu webových aplikací vyvinutý společností Microsoft. Využívá zápis HTML a C# kódu do jednoho souboru pomocí technologie Blazor. Je schopna úzké spolupráce s frameworkem Entity Framework Core pro mapování C# tříd na databázové tabulky. [78]

II. PRAKTICKÁ ČÁST

3 Požadavky na aplikaci

Před programováním aplikace je nutno definovat funkcionální a nefunkcionální požadavky. Funkcionální požadavky zajišťují plynulou a bezchybnou funkci aplikace. Nefunkcionální požadavky určují požadavky na bezpečnost a kompatibilitu s příslušnými verzemi operačních systémů na mobilních zařízeních a webových prohlížečů.

3.1 Funkcionální požadavky

ID požadavku	Popis požadavku
FUN-001	Aplikace musí být schopná zaregistrovat uživatele.
FUN-002	Aplikace musí být schopná přihlásit uživatele.
FUN-003	Aplikace musí být uložit běžeckou aktivitu.
FUN-004	Aplikace musí být schopná zaznamenat efektivní tempo.
FUN-005	Aplikace musí být schopná zaznamenat efektivní vzdálenost.
FUN-006	Aplikace musí být schopná zaznamenat efektivní čas.
FUN-007	Aplikace musí být schopná zaznamenat efektivní kadenci.
FUN-008	Aplikace musí být schopná zobrazit aktivity v měsíci a týdnu.
FUN-009	Aplikace musí umožnit zobrazit detail běžecké aktivity.
FUN-010	Aplikace musí být schopna změnit jednotku vzdálenosti z mílí na kilometry.
FUN-011	Aplikace musí být schopna zobrazit grafy srdečního tepu a kadence.

Tabulka č. 1: Funkční požadavky

3.2 Nefunkcionální požadavky

ID požadavku	Popis požadavku
NFUN-001	Popisují chování aplikace, které nemají tak velký vliv na aplikaci.
NFUN-002	Aplikaci musí být kompatibilní s verzí iOS n-1. Toto znamená, že aplikace bude dostupná na aktuální verzi iOS a na té předchozí.
NFUN-003	Ve webové verzi je důležitá podpora všech hlavních webových prohlížečů (Chrome, Safari, Firefox) a zabezpečení SSL certifikátem.
NFUN-004	Aktuální verze knihoven React a Firebase.

Tabulka č. 2: Nefunkční požadavky

3.3 Efektivní veličiny

S efektivními veličinami aplikace pracuje zejména proto, že díky jejich vzájemnému porovnávání získá běžec lepší přehled o skutečné hodnotě v průběhu optimální tréninkové jednotky. Ta vždy zahrunuje i fázi rozběhání (warm-up) a vyklusání (cool-down). Prakticky všechny dostupné běžecké aplikace vyhodnocují průměrnou hodnotu těchto veličin od startu aktivity do jejího ukončení. Průměr (tempa, srdeční frekvence a kadence) je tak značně zkreslen pomalejším tempem ve fázi rozběhání a vyklusání a současně tomu odpovídající nižší kadenci a nižší hodnotě tepové frekvence. Tento vliv je o to větší, o co je celková tréninková jednotka kratší. Při porovnávání těchto veličin mezi jednotlivými typy tréninku tak dochází k chybným závěrům. Pokud ale uvažujeme pouze

hodnoty dosažené v hlavní tréninkové fázi, tj. mezi ukončením rozběhání a započetím vyklusání, dostaneme nezkreslená data. Ta pak můžeme efektivně porovnávat mezi jednotlivými typy tréninku i u stejného motivu v tréninkové historii běžce. S využitím čistě efektivních dat získáme cenná data pro korelaci všech tří veličin mezi sebou - srdeční frekvence při různých tempech, vliv kadence na tempo při stejné srdeční frekvenci.

V dnešní době s vysokým vlivem sociálních sítí se velká část běžců dopouští zásadní chyby. Tréninkové grafiky zobrazují ve většině běžeckých aplikací průměrné tempo za celý trénink. Běžci tak ve snaze o co nejlepší průměrné tempo zcela vynechávají pozvolné rozběhání a závěrečné vyklusání. Vybíhají ihned v maximálním udržitelném tempu a v něm také aktivitu končí. Ve výsledku tak výrazně zvyšují riziko běžeckého zranění (absence rozběhání) a také výrazně zhoršují následnou regeneraci (absence vyklusání). Proto aplikace Runny zobrazuje výslednou grafiku obsahující efektivní tempo. Umožňuje tak běžcům vést tréninkovou jednotku optimálním způsobem.

3.4 Zóny srdečního tepu

Zóna 1: Zdravé srdce - 50-60% maximální tepové frekvence. Trénink v této zóně v dostatečné délce (30 minut, 4 a vícekrát týdně) přináší široké zdravotní benefity.

Zóna 2: The Temperate zone 60-70% maximální tepové frekvence. Trénink v této fázi produkuje podobné zdravotní benefity, ale ve větší míře.

Zóna 3: The Aerobic Zone: 70-80% maximální tepové frekvence. Trénink okolo 70% - 80% maximální tepové frekvence je lehce těžký.

Zóna 4: Treshold Zone - 80-90% maximální tepové frekvence. Trénink v této zóně podporuje rychlý růst výkonnosti a je vhodný před závodem.

Zóna 5: 90 - 100% maximální tepové frekvence. Trénink v této zóně je vhodný na úseky do 400 metrů. [79]

3.5 Grafický návrh obrazovek

Celý projekt jsem vytvářel samostatně, a tak jsem místo wireframů začal rovnou grafickým návrhem obrazovek. K tomuto účelu jsem použil aktuální verzi aplikace Adobe Photoshop. Pomocí tohoto software jsem vytvořil návrhy hlavních obrazovek:

1) Obrazovka s aktuální aktivitou

2) Obrazovka se seznamem historie běhů

3) Obrazovka s detailem běhu

4) Obrazovka se statistikou běhu

4 Tvorba aplikace

Aplikace je rozčleněna do tříd, které řídí jednotlivé obrazovky. Tyto třídy dědí ze třídy UIViewController. Další třídy jsou implementovány pro zajištění funkcionality aplikace.

4.1 Obecná struktura aplikace

Aplikace je členěna do následujících tříd pro řízení jednotlivých obrazovek:

ViewController - zobrazuje mapu a hlavní uživatelské rozhraní

HistoryController - zobrazuje uložené aktivity

ActivityController - zobrazuje detail aktivity

ShareController - umožňuje sdílení aktivity

BluetoothController - zajišťuje komunikace s hrudním pásem

PlayerController - umožňuje přehrávání hudby z knihovny uživatele

EditController - umožňuje nastavit běžecký cíl a kontrolovat progres k jeho splnění

ManualController - umožňuje manuální přidání aktivity

SettingsController - umožňuje změnit nastavení aplikace

SumarryController - zobrazuje přehled aktivit v čase

StatsController - zobrazuje statistiku běhů

ChartController - zobrazení grafu tepové frekvence a běžecké kadence

PlaylistController - zobrazení skladeb dle běžecké kadence

Modelová vrstva aplikace zahrnuje následující třídy:

CadWorker - pro obsluhu kadence

LMO - pro zjištění vzdálenosti

Pace - pro zjištění tempa

Fetcher - pro načtení běhů z Core Data

Eraser - pro vymazání dat z Core Data

RunEraser - pro vymazání běhů z Core Data

NetworkVM - načítání skladeb dle kadence

SplitKeeper - obsluha okruhů

ColorVM - obarvení grafu srdečního tepu a kadence

RangeGetter - vytvoření popisků os grafů

FirebasePoster - zálohování dat na web

GoalChanger - obsluha přidání cíle

4.1.1 Knihovna Pin.swift

Původní uživatelské rozhraní aplikace bylo vytvořeno ve frameworku UIKit. Kvůli modernizaci a čitelnosti kódu jsem se rozhodl vytvořit knihovnu, která umožní používat SwiftUI view v již existující aplikaci. Důležitou metodou v této knihovně je metoda `func addSwiftUI<Content>(_ newView: Content, top: CGFloat, right: CGFloat, w: CGFloat, h: CGFloat, clear: Bool = false)`. Tato metoda vytvoří obalující element pomocí vytvoření instance třídy `UIHostingController`. Za argument rootView je dosazen SwiftUI view, který chceme přidat do uživatelského rozhraní.

Další užitečnou metodou je metoda `stretchSwiftUI<Content>(_ newView: Content, insets: [CGFloat], clear: Bool = false)`. Pomocí této metody je možné roztáhnout SwiftUI přes celou šířku rodičovského view. Argument `insets` dovoluje nastavit okraje okolo vloženého SwiftUI view.

Knihovnu lze použít také s view implementovanými v jazyce UIKit. Příkladem použití této metody může být meroda `pinTo(_ view: UIView?, position: Q, h: CGFloat, margin: CGFloat)`. Tato metoda umožňuje připnout UIKit view k jinému view. Systém autolayout umožňuje pozicovat prvky v uživatelském rozhraní. Pro aktivaci systému autolayout byla nastavena hodnota `translatesAutoresizingMaskIntoConstraints` na hodnotu `false`. Objekt je umístěn na obrazovku pomocí kotev. Dostupné kotvy v systému autolayout:

`bottomAnchor` - spodní kotva

`topAnchor` - horní kotva

`rightAnchor` - pravá kotva

`leftAnchor` - levá kotva

Datový typ `Q` byl vytvořen pomocí výčtového typu `enum`. Jednotlivé typy jsou pojmenovány `top`, `bottom`, `left` a `right`. Tímto bude vybrána odpovídající strana pro umisťování view. Dále byla implementována funkce `paddingPin(_ view: UIView?, position: Q, w: CGFloat, h: CGFloat, margin: CGFloat)`, která připne view k jinému view a uživateli knihovny dovolí si nastavit offset.

4.2 ViewController

Tato třída se stará o řízení hlavní obrazovky aplikace. Obsahuje informace o vzdálenosti, době trvání aktivity, aktuálním tempu a srdečním tepu. Pozice uživatele je během aktivity zobrazena na mapě. Je možné si přepnout typ mapy na jiný. Na této obrazovce je také možné spustit nebo zastavit aktivitu nebo efektivní tempo.

Obrázek 1: View Controller

4.2.1 CadWorker

Aplikace Runny se od konkurenčních aplikací odlišuje měřením efektivní běžecké kadence. K měření běžecké kadence byl použit framework CoreMotion a třída `CMPedometerData`. Následně byla vytvořena metoda `callFillCads()`, která z aktuálních dat zjistí kadenci a zavolá metodu `getFillCads()`, které vyplní pole kadencemi uvnitř třídy `ViewController`. Tato kadence se měří pouze v efektivním úseku a je z ní později tvořen graf kadence spolu s průměrnou hodnotu, která se zobrazuje uživateli v průběhu aktivity.

4.2.2 LMO

Pro sledování lokace uživatele byla vytvořena třída LMO. Tato třída implementuje protokoly `NSObject` a `CLLocationManagerDelegate`. Uvnitř této třídy jsem použil třídu `CLLocationManager`, která se stará o zjištění aktuální pozice uživatele. Ve třídě bylo povoleno sledování polohy uživatele na pozadí pomocí `allowsBackgroundLocationUpdates = true`.

Při každé aktualizaci polohy uživatele je zavolána metoda `locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation])`. V této metodě si můžeme zjistit polohu uživatele. Vzdálenost, kterou urazil uživatel, byla vypočtena jako vzdálenost aktuální pozice uživatele a poslední uložené lokace. Tohoto bylo dosaženo pomocí

zápisu `travelled += round(location.distance(from: AL))`.

4.2.3 Kreslení trasy uživatele

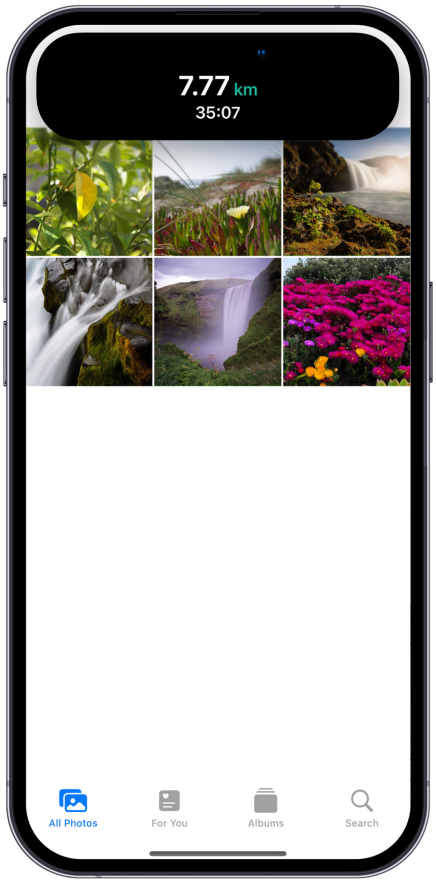
Aplikace podporuje kreslení trasy uživatele za běhu. Za tímto účelem byla implementována funkce drawPolyline. Do třídy PolylinePoints byly dosazeny hodnoty o typu `CLLocation`. Tato třída hodnoty převedla na `CLLocationCoordinate2D`, který mohl být použit uvnitř konstruktoru MKPolyline. Křivka znázorňující trasu běhu byla mapě přidána pomocí metody `map.addOverlay(polyline)`.

4.2.4 SplitKeeper

Tato třída slouží k vytvoření pole obsahující časy jednotlivých kilometrů nebo mil. Pokud je aktuální vzdálenost větší než aktuální limit (při prvním běhu funkce 1000) zapíše se tato hodnota do pole časů a limit se navýší o 1000. Další zápis času tedy proběhne na hodnotě 2000. Ke každému kilometru je zapsán celkový čas běhu. Následně jsou mezi těmito hodnotami zjištěny rozdíly, čímž získáme časy jednotlivých kilometrů.

4.2.5 Dynamic Island

Společně s iPhone 14 Pro představil Apple Dynamic Island. Jedná se o dynamickou oblast, okolo výřezu pro FaceID a přední kameru. Tento výřez zobrazuje data v tzv. Live Activites, které ve výřezu běží, i když aplikace není v popředí. Live Activities mohou běžet i na uzamčené obrazovce telefonu. V aplikaci Runny jsem se rozhodl použít Dynamic Island pro živé zobrazování běžecké



Obrázek 2: Dynamic Island

vzdálenosti v průběhu aktivity. Aplikace zobrazuje vzdálenost a po rozkliknutí Dynamic Islandu také délku aktuálního tréninku.

Dynamic Island byl v Xcode vytvořen jako Widget Extension. Nejprve byla vytvořena struktura `RunnyDynamicIslandLiveActivity` implementující protokol `Widget`. V těle widgetu byla vytvořena třída `ActfivfityConfffiguratfion`, uvnitř které byl vytvořen `LockscreenView` pro zobrazení Live Activity na uzamčené obrazovce telefonu. V uzávěru struktury `ActfivfityConfffiguratfion`, který se nazývá `dynamicIsland`, byl vytvořen view, který se zobrazí v Dynamic Islandu. Jako argumenty do tohoto view byly dosazeny vzdálenost a aktuální jednotka vzdálenosti, kterou má uživatel nastavenou - buď míle nebo kilometry.

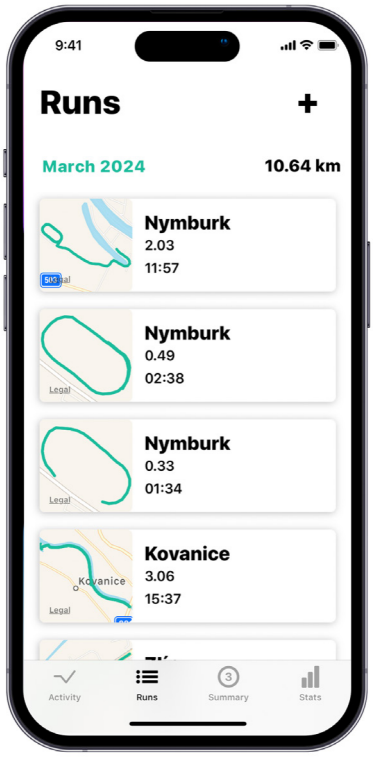
Do uzávěru funkce `compactLeading` ve view `DynamicIsland` byl uveden název aplikace - Runny a do argumentu `compactTrailing` byla uvedena uběhnutá vzdálenost získaná z `context.state.travelled`. Na převod kilometrů bylo použito rozšíření `toKm()`, které převede aktuální vzdálenost na míle nebo kilometry dle preference uživatele.

K tomuto převodu bylo použito rozšíření datového typu `String`, ve kterém byla nejprve zjištěna preferovaná jednotka uživatele, a poté byl proveden převod. Buď byla vzdálenost vydělena 1609 při převodu na míle, nebo 1000 při převodu na kilometry. Pro získání konečného textového řetězce bylo použito inicializátoru třídy `String - init(format:_:)` a za argument `format` byla dosazena hodnota `%.2f`, což značí formátování na dvě desetinná místa.

Aktuální jednotka vzdálenosti byla přečtena jako hodnota klíče `unit` z třídy `UserDefaults`. Pomocí `context.state.time.getFormattedTime()` byl získán naformátovaný čas. Nejprve byly převedeny sekundy na hodiny, minuty a sekundy a poté byla přidána nula k jednociferným hodnotám, aby bylo zachováno formátování.

4.3. HistoryController

Pro načítání uložených běžeckých aktivit je vytvořena třída `HistoryController`. Pro vytvoření seznamu běžeckých aktivit byla použit scrolovací list vytvořený pomocí třídy `UITableView`. Pro obsluhu tohoto seznamu byla implementována třída `HistorySource`.Tato třída implementuje protokol `UITableViewDataSource` a jeho metody. Pro naplnění této třídy daty byly vytvořeny třídy `Fetcher` a `MonthGen`.



Obrázek 3: HistoryController

4.3.1 Třída Fetcher

Třída byla vytvořena k načtení dat o jednotlivých bězích. Za tímto účelem byla vytvořena metoda `fetchR`. Nejprve byla získána reference aplikace pro zajištění přístupu ke kontejneru dat pomocí `UIApplication.shared.delegate as! AppDelegate`. Potřebný kontejner byl z aplikace získán pomocí `let context = delegate.persistentContainer.viewContext`. Tento kontejner obsahuje perzistentní data aplikace. Dále byl vytvořen request, kterým si můžu načíst samotná data aplikace. Pro načtení dat o bězích byla do konstruktoru této třídy dosazena hodnota `Runs`. Zápis vypadal následovně: `let request = NSFetchRequest<NSFetchRequestResult>(entityName: "Runs")`. Po zavolání metody fetch na objektu `request`, byla získána data o bězích. Každý běh byl dosazen do pole `runs`, které bylo z funkce navráceno. Toto pole poslouží k vyplnění obsahu tabulky ve třídě `HistoryController` pro zobrazení historie běhů.

4.3.2 Třída MonthGen

Třída `MonthGen` byla vytvořena pro roztřídění běhů dle měsíce a roku. Při zobrazení historie běhů probíhá filtrování pomocí rozšíření typu Array - `zmGroup`. Toto rozšíření sdruží běhy podle měsíce a data a vrátí dvojrozměrné pole běhů.

4.3.3 Třída Eraser

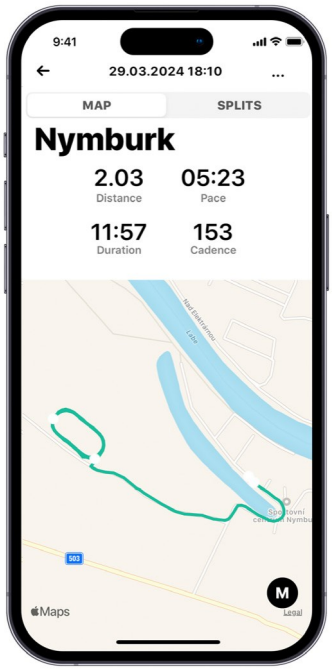
Tato třída byla implementována pro mazání dat z aplikace. Pro získání kontejneru s daty, které chceme smazat, byl opět použit zápis `let context = delegate.persistentContainer.viewContext`. Poté byl vytvořen request ke smazání položek pomocí `NSBatchDeleteRequest(fetchRequest: _)`. Za argument `fetchRequest` byl dosazen objekt o typu `NSFetchRequest`. Pro samotné smazání data byla na objektu `NSBatchRequest` zavolána metoda `context.execute(deleteRequest)`. Pro ukončení operace byla použita operace `context.save()`.

4.3.4 Třída RunEraser

Tato třída funguje podobně jako třída `Eraser` a obsahuje metodu `erase(date: String)`. Za argument date je dosazeno přesné datum s časem běžecké aktivity. Tímto je zajištěno jednoznačné určení běžecké aktivity ke smazání. Poté je pomocí volání `context.fetch(request)` získáno pole běhů o typu `NSManagedObject`. Dále je cyklem iterováno pole běhů a pomocí zápisu `myrun.value(forKey: "runs") as! Run` přetypováno na typ `Run`. Pokud se datum běhů rovná datumu v argumentu, pomocí volání `context.delete()` je potřebný objekt smazán a pomocí `context.save()` uloženy změny. Celá operace je prováděna v konstrukci pro zachycení výjimek - `try catch`.

4.4 ActivityController

Tato třída byla vytvořena pro obrazovku, na které je zobrazován detail běhu. Pro tvorbu mapy byla vytvořena třída Map, která dědila ze třídy `MKMapView`. Pomocí `self.showUserLocation` bylo nastaveno zobrazování polohy uživatele na mapě. Dále bylo potřeba vytvořit křivku znázorňující trasu uživatele. Za tímto účelem byla vytvořena třída `MapDelegate` implementující protokol `MKMapViewDelegate`. Uvnitř této třídy je implementována metoda `mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) -> MKOverlay`. Uvnitř této metody byla vytvořena křivka pomocí třídy `MKPolylineRenderer`. Křivce byla byla přiřazena barva tahu pomocí `renderer.strokeColor = hex("#1abc9c")`. Tloušťka křivky byla definována pomocí `renderer.lineWidth = 4`.



Obrázek 4: ActivityController

Pro přepínání mezi obrazovkou s mapou a časy kilometrů byla vytvořena instance třídy `UISegmentedControl`. Pomocí deklarování proměnné o typu `[NSAttributedString.Key: Any]` byla segmentu přiřazena tučná váha fontu a černá barva.

4.4.1 Pace

V této třídě probíhá výpočet tempa uživatele. Třída ve svém konstruktoru přijímá čas v sekundách a vzdálenost v metrech. Následně je v metodě `getPace` vypočteno aktuální tempo uživatele podle jím preferované jednotky. Následně je tempo naformátováno a z funkce navráceno.

4.4.2 Změna typu mapy

Uživatel si v aplikaci může zvolit mezi třemi typy map:

1) normální - mapa z pohledu satelitu

2) hybridní - satelitní mapa se znázorněním liniových staveb

3) fly-over - mapa z perspektivního pohledu tam, kde je funkce podporována

Typ mapy lze změnit před zahájením aktivity nebo v `ActivityViewController` při jejím prohlížení.

Pro vytvoření action sheetu byla implementována třída `Sheet` s metodou `show`. Tato metoda si bere do argumentu referenci na `ViewController`, titulek sheetu a zprávu, kterou zobrazí uživateli a pole o typu `[UIAlertAction]`, které slouží pro přidání jednotlivých akcí. Po vytvoření instance třídy `UIAlertController` byl za argument `preferredStyle` dosazen typ `.actionSheet`, čímž bylo zajištěno, že action sheet bude typu rozklikávací rolety, která vyjede zespodu obrazovky. Následně cyklus procházel pole akcí a každá akce byla třídě `AlertController` přiřazena.

`AlertController` byl asynchronně prezentován pomocí volání funkce `DispatchQueue.main.async`. Uvnitř této funkce byla zavolána funkce `on.present(alert, animated: true, completion: nil)`, která action sheet zobrazí uživateli.

4.4.3 Metoda addPolyline

Tato metoda přidá křivku znázorňující trasu uživatele do mapy. Tato křivka je vytvořena pomocí třídy `MKPolyline`, do které jsou dosazeny souřadnice běžecké aktivity o typu `CLLocationCoordinate2D`. Následně je zavolána metoda `seflff.fffit(poflyflfine: poflyflfine, padding: 38)`, která přizpůsobí velikost mapy křivce.

4.4.4 Metoda fit

Tato metoda umožňuje zmenšit mapu tak, aby se její velikost přizpůsobila křivce zobrazující trasu uživatele. Metoda ~~fffit~~ byla definována jako rozšíření třídy `MKMapView`. Do této metody byla dosazena čára zobrazující trasu uživatele o typu `MKPolyline` a vnější okraj (`padding`) . Uvnitř této metody byla definována konstanta insets o typu `UIEdgeInsets`. Velikost tohoto odsazení bude přečtena z hodnoty argumentu `padding` z této metody. Poté byla použita metoda `setVisibleMapRect`, do které byl dosazen rámeček o velikosti polyline (`polyline.boundingMapRect`). Za argument `edgePadding` byla dosazena hodnota `insets`.

4.4.5 GPX generátor

GPX je formát, který akceptuje většina běžeckých platforem pro navigaci. V aplikaci Runny byla implementována možnost generování gpx souboru pomocí třídy `GPXGen`. V této třídě byla vytvořena metoda `getGPX()`, která vytvořila textový řetězec se souřadnicemi. Formát gpx vyžaduje uvnitř tagu `<gpx>` tag `<wpt>`, do kterého dosadíme za argumenty `lat` a `lon` aktuální zeměpisnou šířku a délku běhu. Nakonec vrátíme z funkce pole obsahující tyto textové řetězce. Uživatel si může zvolit název souboru a soubor je uložen do adresáře uživatele.

4.5 ShareController

Třída `ShareController` slouží pro ovládání obrazovky s generováním obrázku pro přidání na sociální sítě. Pro výběr obrázku byla použita třída `UIImagePickerController` pro přepínání mezi mapou a obrázkem třída `UISegmentedControl`. Na obrázek bylo potřeba vepsat aktuální vzdálenost, efektivní tempo a délku trvání aktivity. Za tímto účelem byla implementována třída `TransformVM`, do které byl v jejím inicializátoru předán obrázek s trasou běhu nebo obrázek z galerie uživatele. Pomocí třídy `UILabel` byly vytvořeny popisky, které jsou nadepsány nad obrázkem. Tyto popisky byly poté přidány do `UIStackView` pro zobrazení v řádku vedle sebe. Pomocí třídy `UIGraphicsGetCurrentGraphicContext` bylo spuštěno kreslení do aktuálního kontextu. Jednotlivé položky byly vyrenderovány pomocí volání metody `element.layer.render(in: ctx)` a nakresleny do výsledného obrázku. Obrázek poskytuje důležitá data o běhu, včetně názvu aplikace.

4.5.1 Tvorba snímku z aktivity

Pro tvorbu snímku z aktivity byla použita třída `SnapShooter`. V metodě `getSnap` byla vytvořena instance třídy `MKMapSnapshooter.Options()` a velikost byla nastavena na `300 x 300` obrazových bodů. Uvnitř této metody byla použita třída `MKMapSnapshooter`, do které byla dosazena data o požadované velikosti obrázku a její metodu `start`, ve které byla nakreslena křivka znázorňující trasu, po které uživatel běžel. Tato křivka byla vytvořena pomocí Core Graphics operací. Kreslení bylo zahájeno pomocí `UIGraphicsGetCurrentContext()`. Křivce byla nastavena tloušťku tahu na hodnotu 3.0 a barva na #1abc9c, což je také hlavní barva uživatelského prostředí aplikace. Kreslení bylo zahájeno pomocí metody `context.move()`, kde byla nastavena počáteční pozice pro kreslení na první souřadnici z uložené aktivity. Poté byla kombinací metod `addLine(to:)` a `move(to:)` nakreslena křivka znáznorňující trasu běhu. Konečný tvar trasy byl vytvořen voláním `context.strokePath()`. Výsledný obrázek byl nakreslen pomocí volání metody `UIGraphicsGetImageFromCurrentImageContext()` a ukončen pomocí `UIGraphficsEndImageContext()`.

Metoda `shooter.genSnap` byla použita uvnitř Action Sheetu, který se zobrazuje při rozkliknutí detailu dané aktivity. Výsledný obrázek je dosazen do třídy `ShareController`, ve které si uživatel může vygenerovat obrázek pro vytvoření příspěvku na sociální sítě.

4.6 BluetoothController

Aplikace Runny podporuje sledování srdečního tepu pomocí hrudního pásu. Hrudní pásy podporované aplikací využívají technologii Bluetooth pro odesílání dat do telefonu. Pro vytvoření seznamu nalezených Bluetooth zařízení byla implementována třída `BluetoothController`.

Tato třída nejprve proskenuje okolí na kompatibilní Bluetooth běžecké pásy. Aplikace skenuje Bluetooth zařízení s identifikátorem `0x180D`, což označuje pásy pro měření tepové frekvence. Skenování je zahájeno pomocí metody `scanForPeripherals(withServices: [hr], options: nil)`. Po nalezení vhodných Bluetooth zařízení jsou tato zařízení zapsána do

seznamu a tabulka je aktualizována pomocí volání metody `tableView.reloadData()`.

Po klepnutí na buňku je nejprve ověřeno, že zařízení uživatele přijímá Bluetooth připojení pomocí `bmanager.state == . poweredOn` a dojde k připojení běžeckého pásu pomocí metody `func connect(peripheral: CBPeripheral,options: [String : Any]? = nil)`. Dále bylo potřeba přečíst hodnotu srdečního tepu. V metodě `didUpdateCharacteristics` je přečtena hodnota srdečního tepu a je aktualizováno uživatelské rozhraní. Aplikace zaznamenává data za celou aktivitu a uživatel si po uložení aktivity může zobrazit graf srdečního tepu.

4.7 PlayerController

Třída `PlayerController` byla vytvořena pro zobrazení přehrávače hudby. Hudba byla načtena z knihovny uživatele. Instance třídy `PlayerStack` byla použita pro vytvoření tlačítek previous, next a forward. Jednotlivé metody byly tlačítkům přiřazeny pomocí metod `pl.addTarget(self, action: #selector(lefta(sender:)), for: .touchUpInside)`. Funkcionalita přehrávání hudby byla vytvořena pomocí instance třídy `AVQueuePlayer`. Pro přehrání nebo zastavení skladby byla implementována metoda `handle`, uvnitř které byly volány metody `play` a `pause`.

Metoda pro přehrání další nebo předchozí skladby byla implementována jako rozšíření třídy `AVQueuePlayer`. Uvnitř metod pro jednotlivá tlačítka byla pomocí metody playera.next přehrána následující skladba. Přehrání předchozí nebo další skladby bylo řízeno pomocí argumentu „dir", který mohl nabývat hodnoty „right" nebo „left".

Pokud je směr přehrávání nastaven na „right" bude uvnitř metody `next` byly zavolány metody `self.advanceToNextItem()` a `self.play()`. Pomocí zápisu `button.setImage(UIImage(named: "pause-30-30.png"), for: [])` bylo během změny skladby nastaveno tlačítko pro přehrání skaldby na symbol pause.

Pokud je směr přehrávání nastaven na "left", byla zavolána metoda `previous(index: Int, items: [AVPlayerItem])`. Z argumentu `index` byl získán index skladby pro následující přehrání. Pomocí `self.canInsert(obj, after: nil)` si zjistíme, zda můžeme do seznamu vložit další skaldbu a pokud ano, přetočíme skaldbu na začátek pomocí `obj.seek(to: .zero, completionHandler: nil)` a pomocí `self.insert(obj, after: nil)` ji přehrajeme.

4.7.1 PlayerStack

Pro zobrazení hlavního uživatelského rozhraní byla implementována třída `PlayerStack`. Tato třída vznikla jako potomek třídy `UIStackView`. Uvnitř jejího inicializátoru byla vytvořena instance mé třídy `Flabel`, která vytváří popisky s názvem alba a umělce. Pro přepínání další nebo předchozí skladby byla implementována tlačítka `leftf` a `rightf`. Obě tlačítka byla vytvořena pomocí třídy `UIButton`. Tlačítku byly přiřazeny obrázky forward.png a previous.png pomocí zápisu `b.setImage(UIImage(named: "previous.png"), for: [])`. Stejný postup byl použit pro zobrazení tlačítka "Play". Do UI byla tlačítka přidána pomocí metody `addArrangedSubviews`.

4.7.2 AssetsVM

Pomocí třídy `AssetsVM` byly načteny položky z knihovny uživatele. Uvnitř metody `fetch` byly položky načteny pomocí volání `MPMediaQuery.songs().items`. Následně byla vytvořena instance třídy `AVURLAsset`, do které byla dosazena adresa `URL` písničky a instance třídy `AVPlayerItem`, do které byla dosazena instance třídy `Asset`. Tyto položky byly nakonec dosazeny do třídy `AVQueuePlayer`, která slouží jako samotný přehrávač.

4.7.3 MusicLoader

Pro načtení položek o jednotlivých interpretech byla vytvořena třída `MusicLoader`, jednotlivé položky byly načtena pomocí volání `MPMediaQuery.songs().items`. Z této třídy byly načteny následující údaje:

- název alba - `val.title`
- skladba - `val.assetURL?.absoluteString`
- název skaldby - `val.title!`
- jméno interpreta - `val.artist`
- obrázek skladby - `val.artwork?.image(at: .init(width: 50, height: 50))`

Pokud skladba neexistuje, byl použit generický obrázek music.png. Jednotlivé hodnoty byly z pole přiřazeny příslušným labels v uživatelském rozhraní.

4.8 EditController

Třída `EdfitControflfler` implementuje protokol `UIViewController` a umožňuje uživateli změnit cíl aktivity. Uživatel si může nastavit nový cíl pomocí slideru a během nastavování se mu zobrazují užitečné běžecké tipy.

4.8.1 GoalChanger

Aplikace podporuje nastavení cíle běžecké vzdálenosti. Uživatelské rozhraní je tvořeno pomocí slideru a indikátoru, který ukazuje progres splnění cíle. Aktuální progres vůči spnění cíle je načten pomocí vytvoření instance třídy `Fetcher` - `Fetcher(ename: "Users", key: "users")`. Tato třída je využita i pro načtení dat o běžeckých aktivitách. Po zavolání metody `fetchU()` je načten aktuální cíl uživatele. Pokud uživatel nastaví nový cíl a obrazovku se pokusí opustit stisknutím tlačítka zpět, tak je mu ukázána roleta s možností potvrzení nového cíle. Pokud stiskne tlačítko "Save", `EdfitControflfler` zmizí a cíl je uložen.

4.8.2 Třída NetworkVM

Třída `NetworkVM` měla za cíl načítání běžeckých tipů. Tato třída byla vytvořena pomocí třídy `URLSession.shared.dataTask`. Tato data jsou uložena v textovém souboru na doméně

runnyapp.github.io. K načtení dat byla použita metoda `fetch`. Obsah adresy byl načten do proměnné `ctx` uvnitř bloku try catch pro zachycení případných výjimek. Díky tomuto zápisu nedojde v případě výjimky k pádu aplikace. Po načtení dat je z nich vytvořeno pole pomocí metody `ctx.components(separatedBy: .newlines)`. Po zavolání funkce `completion(arr)` bylo pole z funkce vráceno.

4.8.3 Tvorba tlačítek

AddB

Třída AddB byla vytvořena pro zajištění konzistentního stylu tlačítek v aplikaci. Tato třída implememtuje metody `setup()` a `layoutSubviews()`. Ve metodě `setup` je pomocí `self.layer.cornerRadius = self.frame.height / 2` nastaven poloměr rohů tlačítka na polovinu výšky rámečku. Tímto bylo dosaženo zaoblených rohů. Aby zůstaly rohy tlačítka zaobleny i po použití auto layoutu, byl tento krok zopakován i v metodě `layoutSubviews()`.Tato třída byla použita pro základ tlačítka. Později byla tomuto tlačítku nastavena barva popředí a pozadí. Toto tlačítko bylo použito pro vytoření tlačítka "Save" na obrazovce. Titulek tlačítka byl nastaven pomocí volání funkce `setTitle("Save", for: [])`. Voláním metody `b.setTitleColor(.white, for: [])` byla nastavena barva textu tlačítka na bílou a pomocí `b.backgroundColor = hex("#1abc9c")` barva pozadí na světle zelenou. Text tlačítka byl zarovnán na střed pomocí `b.titleLabel?.textAlignment = .center`. Pomocí volání funkce `b.addTarget` byla tlačítku přiřazena akce po stisknutí. V tomto případě pro uložení cíle.

ShareButton

Pro vytvoření tlačítka na přepínání typů map byla vytvořena třída, která dědila ze třídy `UIButton`. Uvnitř inicializátoru byla zavolána metoda `setup`. Uvnitř této metody byl nastaven průměr rohů tlačítka na hodnotu 20 pomocí `self.layer.cornerRadius = 20`. Toto mělo za následek vytvoření kulatého tlačítka.

4.9 ManualController

Třída `ManualController` slouží pro manuální vytvoření a uložení aktivity. Uvnitř této třídy byla vytvořena metoda `save(gdate: Date, location: String, date: String, dist: String, time: String, dur: Int)`, která uloží danou aktivitu. Nejprve je zjištěno aktuální datum. Datum bylo sestaveno následujícím způsobem:

týden: `cal.component(.weekOfYear, from: gdate)`

měsíc: `cal.component(.month, from: gdate)`

rok: `cal.component(.year, from: gdate)`

Z argumentů funkce sestavím objekt pro uložení do databáze. Pokud uživatel nevyplní všechny povinné údaje, objeví se mu okno s upozorněním na všechny povinné údaje. Pro uložení aktivity je použita instance třídy `Saver` a zavoláním metody `save()` je aktivita uložena.

4.9.1 Třída RField

Třída `RField` byla vytvořena pro vytvoření jednotného vzhledu textového pole pro zadání dat. Třída dědí ze tříd `UITextField` a `UITextFieldDelegate` a uvnitř svého inicializátoru volá metodu `setUI()`. Tato metoda metoda nastavuje tloušťku rámečku textového pole pomocí `self.layer.borderWidth`, poloměr rohů pomocí `self.layer.cornerRadius = 6.0` a barvu pozadí na hodnotu .clear. Textovému poli je nastaven tučný systémový font o velikosti 21 pomocí `self.font = UIFont.boldSystemFont(ofSize: 21)`.

4.9.2 hex

Pro zápis barev v aplikaci Runny byla implementována globální funkce `hex(_ hex: String)`, která přijme v argumentu barvu ve formátu HEX a pomocí bitových operací z ní vytvoří instanci třídy `UIColor`. Tato funkce může být později použita pro definici barev pro uživatelské rozhraní aplikace.

4.10 SummaryController

Pro zobrazení historie běhů byla vytvořena třída `SummaryController`. Jejím důležitým komponentem je roleta pro výběr roku - `PickerPasser`. Tato roleta implementuje tři důležité metody:

`pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int` - pro načtení počtu položek rolety

`pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?` - tato metoda vrátí názvy položek v roletě

`pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int)` - tato metoda zavolá metoda v `SumarryControlleru` přes delegáta a vyfiltruje potřebné informace.

Filtrování informací probíhá ve třídě Report. Tato třída implementuje metody jako `getYear` pro navrácení celkového počtu kilometrů a `getDuration` pro zobrazení celkové délky aktivit za daný rok. Dále obsahuje metodu jako `getMonths` pro vrácení počtu uběhnutých kilometrů za daný měsíc a `getWeeks` pro vrácení počtu uběhnutých kilometrů za daný týden.

Uvnitř metody `getWeeks` se pomocí volání metody ~~fffilter~~ profiltrují běhy tak, aby se načetly jen běhy z daného týdne a roku. Poté byla na výsledek zavolána metoda `map` pro vytvoření pole vzdáleností. Tyto vzdálenosti byly následně sečteny pomocí metody `reduce`.

Po vybrání měsíce nebo týdne v roletě automaticky sjede scrolovací seznam běhů na potřebný řádek pomocí volání metody `scrollTo`.

Celá roleta je implementována uvnitř rolety o typu `.actionSheet`. Uvnitř této rolety lze také měnit mezi týdnem a měsícem. Roleta se uživateli zobrazí po kliknutí na tlačítko “Filter“.

Sloupce ve vyfiltrovaném seznamu jsou následující:

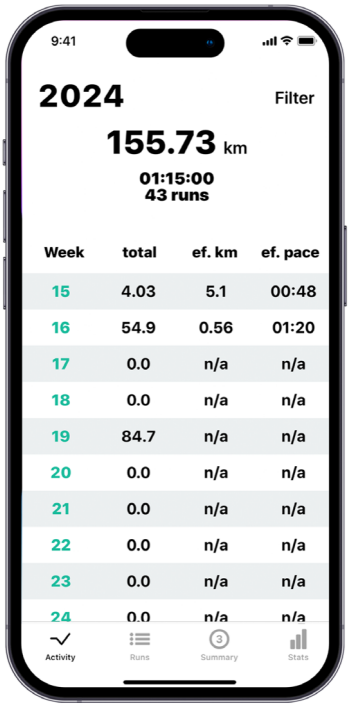
Week/Month

total = celkový počet naběhaných kilometrů

ef. km = vzdálenost naběhaná v efektivním tempu

ef. pace = efektivní tempo

Některé prvky této obrazovky byly implementovány s použitím SwiftUI a mé vlastní knihovny, která podporuje přidávání SwiftUI view do aplikací implementovaných ve UIKit. Rozborem této knihovny se budu zabývat v pozdějších částech této práce.



Obrázek 5: SummaryController

4.10.1 NavBar

Pro vytvoření navigační lišty byla implementována třída `NavBar`. Jednotlivá tlačítka jsou tvořena pomocí instance třídy `UINavigationItem`. Tlačítkům byl nastaven tučný text pomocí metody `setTitleTextAttributes`. Samotný tučný text byl nastaven pomocí `NSAttributedString.Key.font: UIFont.systemFont(ofSize: 18, weight: UIFont.Weight.bold)`.

Obdobným způsobem byl implementována třída `ShareBar`. Tato třída je použita na obrazovce se sdílením aktivit - `ShareController`. Dále byla vytvořena třída `dismissBar`, která je použita na více obrazovkách aplikace. Pro zajištění jednotného vzhledu byla jako symbol pro tlačítko zpět byla zvolena šipka doleva.

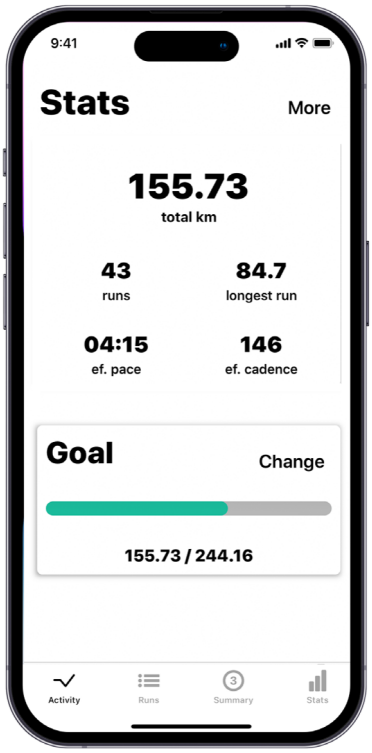
Do konstruktoru této byla dosazena funkce typu `Selector`. Tato funkce byla zavolána po stisknutí tlačítka v navigační liště. Do instance této třídy byla za argument `selector` dosazena hodnota `#selector(název funkce)`. Tento zápis umožní po stisknutí tohoto tlačítka zavolat určitou funkci.

Rozšíření `transparence()` bylo definováno na typu `UINavigationBar` a zajistilo tvorbu průhledné navigace pomocí nastavení vlastnosti `self.isTransluescent` na `true`.

4.11 StatsController

Třída StatsController slouží pro zobrazení statistik běhu jako:

- počet běhů
- nejdelší běh
- průměrné efektivní tempo
- průměrná efektivní kadence



Obrázek 6: StatsController

Tyto statistiky byly vypočteny pomocí třídy `StatsVM`. Tato tříde bere v potaz preferovanou jednotku vzdálenosti uživatele. Původní uživatelské rozhraní této třídy ve frameworku UIKit bylo v pozdější fázi vývoje přepsáno ve SwiftUI pomocí views `StatsView` a `StatsLabel`, které byly do aplikace přidány pomocí `addSwiftUI`.

4.12 SettingController

Třída `SettingsController` slouží pro celkové nastavení aplikace. Uživatel si může vybrat jednotku vzdálenosti (míle nebo kilometry) pomocí view `Picker`. Dále je možné aktivovat zvukovou zpětnou vazbu pomocí view `Toggle` a zobrazit podmínky používání aplikace (EULA). Také možné smazat veškeré aktivity z aplikace.

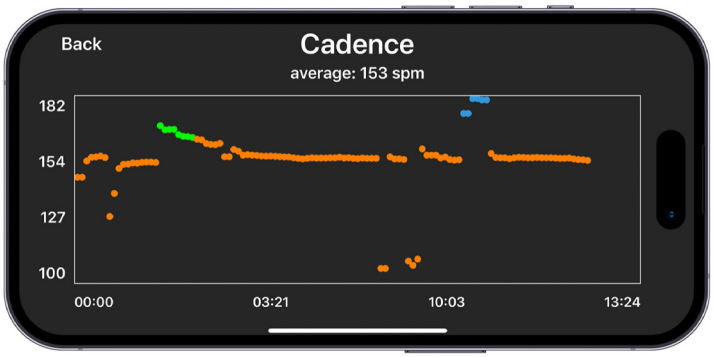
4.12.1 UnitGetter

Třída `UnitGetter` slouží k převodu jednotek vzdálenosti. Pro získání jednotky vzdálenosti byla deklarována property unit o typu `Unit`. Typ `Unit` byl definován jako výčtový typ a nabývá hodnot kilometr nebo míle.

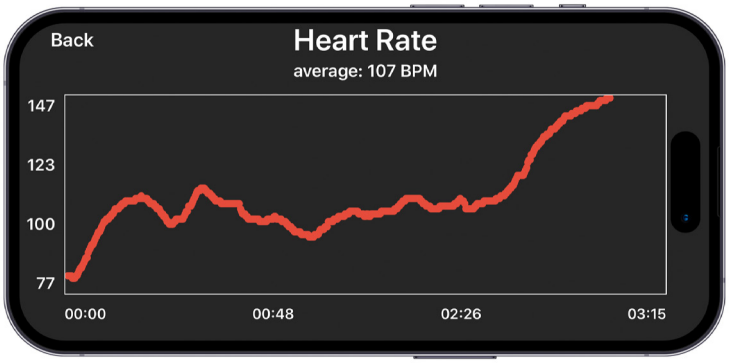
Pro zjištění aktuální jednotky uživatele byl použit zápis `UserDefaults.standard.value(forKey: "unit")`. Ve třídě `UserDefaults` jsou zapsána trvalá data aplikace. Pokud má uložený textový řetězec hodnotu "miles", z funkce byla navrácena hodnota `.miles`, v opačném případě nebo pokud hodnota ještě nebyla uložena, bude navrácena jednotka `.miles`. Tato metoda je použita například ve třídě `ManualVM`, kde je po zadání vzdálenosti v kilometrech vzdálenost převedena na míle, pokud uživatel preferuje míle.

4.13 ChartController

V aplikaci jsem se rozhodl implementovat grafy tepové frekvence a běžecké kadence pro zhodnocení tréninkové aktivity.



Obrázek 7: Graf běžecké kadence



Obrázek 8: Graf tepové frekvence

4.13.1 Graf tepové frekvence (iOS)

Pro graf tepové frekvence a běžecké kadence na systému iOS byla implementována třída `Chart`, která dědila ze třídy `UIView`. Pro samotné kreslení grafu byla implementována metoda override func `draw(_ rect: CGRect)`. Kreslení byl zahájeno voláním `UIGraphicsGetCurrentContext()`. Ploše byla nastavena barva pozadí na pomocí ctx. `setFillColor` na barvu #262626. Poté bylo cyklem iterováno pole jednotlivých bodů a tyto body byly přidány do grafu. Pro vytvoření kolečka, které bude zapsáno do grafu, byla vytvořena instance třídy `Point`. Třída point uvnitř své metody override `func draw(_ rect: CGRect)` kreslí kolečko pomocí metody `addArc`. Kolečku je nastavena barva výplně pomocí setFillColor. Kolečko je nakresleno pomocí volání `ctx.setfffiflflPath()`. Pro rovnoměrné rozmístění bodů v grafu byla

implementována metoda `grandFather`. Rozdíl mezi hodnotami je vypočítán jako max - min a konečná hodnota jako *(curr - min) * (h / diff)*, kde *h* představuje výšku elementu.

4.13.2 ColorVM

Tato třída slouží k obarvení bodů kadence dle hodnoty. Kadence, která má větší než hodnotu 183 kroků/min, je znázorněna fialovou barvou. Hodnota, která je větší než 163 kroků/min a zároveň menší než 173 kroků/min, je znázorněna oranžovou barvou.

4.13.3 RangeGetter

Třída RangeGetter byla implementována pro vytvoření vodorovné osy pro graf kadence. Nejprve je zjištěn rozsah kadencí pomocí odečtení nejvyšší hodnoty kadence od nejnižší. Maximální a minimální hodnota byla zjištěna pomocí vestavěných metod min a max. První bod odpovídá maximální hodnotě tepové frekvence. Druhý bod odpovídá rozdílu (`max - min) * 0.66 + min` a třetí bod odpovídá (`max - min) * 0.33 + min`. Poslední bod odpovídá maximální hodnotě. Všechny tyto hodnoty byly zapsány do pole a toto pole bylo z funkce navráceno.

4.14 SignController

Pro možnost zálohování běhu byla implementována třída `SignController`. Tato třída slouží k zálohování běžeckých aktivit do webové apliakace. Pro vytvoření navigační lišty byla implementována třída `DismissBar`. Běžecká aktivita ke sdílení byla přidána do třídy `SignController` skrze konstruktor. Následně byl použit SwiftUI view `DecideView`, který ve svém argumentu přebíral běh o typu `RunVM`. Tento SwiftUI view byl do aplikace předán voláním globální funkce `addSwiftUI`. Tato funkce pochází z mé vlastní knihovny Pin.swift.

4.14.1 DecideView

`DecideView` je SwiftUI view, který slouží pro řízení toku zálohování běhu na web. Pokud je uživatel přihlášen, zobrazí se mu view `BackupView`, ve kterém může uložit data o běhu. V opačném případě se mu zobrazí registrační a přihlašovací obrazovka.

4.14.2 BackupView

Uvnitř těla `BackupView` se nachází view s se zprávou “The run has been backed up to www.runny-app.github.io“. Tento view v metodě `onAppear` zavolá metodu `FirebasePoster.backup(run: run)` a aktivita je zálohována.

4.14.3 FirebasePoster

Třída `FirebasePoster` slouží pro zálohování běžeckých aktivit na web. Uvnitř této metody byla data připravena na zálohování na web. Nejprve bylo cyklem iterováno pole souřadnic a z každé souřadnice byl vytvořen textový řetězec pomocí třídy `Coord`. Následně byly uloženy ostatní podstatné hodnoty jako: tepová frekvence, kadence, časy kilometrů, časy mil, vzdálenost a souřadnice.

Tato data byla zapsána do pole o typu `["String":"Any"]`. Reference k databázi byla vytvořena zápisem `Ffirestore.fffirestore().Vofláním db.coflflectfion("Runs").document(uid!).collection("MyRuns").addDocument(data: dict)` byl dokument přidán do kolekce uvnitř Firestore.

4.15 PlaylistController

PlaylistController slouží ke zobrazení seznamu skladeb. Pro vytvoření tohoto seznamu byl implementován scrolovací seznam `PlaylistTable`. Tento seznam implementuje protokol `UITableView`. V inicializátoru třídy se volá metoda setup. Pomocí metody `self.regfister(PflayflfistCeflfl.seflff, fforCeflflReuseIdentfifffier: "PCCeflfl")` je zaregistrována buňka o typu `PlaylistCell`. Jako barva oddělovačů je použita průhledná barva pomocí zápisu `.clear`.

Třída `PlaylistController` implementuje metody `tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int` pro navrácení počtu položek a `func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell` pro vrácení buňky na danou pozici a její osazení textem. Po klepnutí na buňku je prezentován view controller se seznamem písniček. Klepnutí na buňku zachytíme pomocí metody `func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)`. Seznam písniček je implementován pomocí SwiftUI `List` a do aplikace přidán pomocí volání `stretchSwiftUI(DetailControllerView(heading: self.heading, songsvm: self.songsvm), insets: [40, 0, 0, 0])` z mojí knihovny Pin.swift.

Možnost navrácení se zpět přejetím prstu po obrazovce byla přidána pomocí metody `addGesture()`. Uvnitř této metody byla použita instance třídy `UISwipeGestureRecognizer` a směr přejíždění prstu byl nastaven pomocí `.right` na směr zleva doprava. Pomocí volání `view.addGestureRecognizer(rec)` je tento recognizer přiřazen view controlleru. Po přejetí prstu je zavolána funkce `getBack`, která zavolá metodu `dismiss(animated: true, completion: nil)`. Toto volání skryje aktuální view controller.

Pomocí přepsání property `preferredStatusBarStyle` pomocí `override var preferredStatusBarStyle: UIStatusBarStyle` je vždy vrácena podoba status baru v light módu pomocí `.lightContent`. Aby se podoba status baru zachovala při otevření obrazovky, je uvnitř metody `viewWillAppear(_ animated: Bool)` zavolána metoda `self.setNeedsStatusBarAppearanceUpdate()`.

4.16 Třída RunnyAlert

Pomocí třídy `RunnyAlert` bylo usnadněno vytvoření dialogových oken v aplikaci. Uvnitř této třídy byla implementována metoda `show(on: UIViewController, title:String, messsage:String, actions: [UIAlertAction])`, která měla za úkol zobrazit dialogové

okno. Tohoto bylo dosaženo pomocí instance třídy `UIAlertController`. Do konstruktoru této třídy byla za argument `preferredStyle` dosazena hodnota `.alert`. Následně bylo iterováno pole akcí a jednotlivé akce byly přidány třídě `UIAlertController` pomocí metody `alert.addAction`. Dialogové okno bylo uživateli zobrazeno pomocí volání metody `on.present(alert, animated: true, completion: nil)`. Tato metoda byla zavolána asnychronně uvnitř statické property `DispatchQueue.main.async`.

4.17 Důležitá rozšíření

- Pro zpřehlednění kódu v aplikaci bylo vytvořeno množství rozšíření datových typů pomocí klíčového slova extension. Zde je přehled těch nejdůležitějších:
  `setImage` - rozšíření obrázku `UIImage` pro zobrazení obrázku želvy nebo zajíce u nejrychlejšího nebo nejpomalejšího kilometru
- `inc` - navýšení hodnoty o 1
- `dec` - snížení hodnoty o 1
- `toMeters` - vynásobení hodnoty 1000

4.18 Aplikace pro watchOS

Protože velká část uživatelů iPhone využívá watchOS, rozhodl jsem se také vytvořit aplikaci pro tento operační systém. Obě aplikace jsou vytvořeny ve stejném projektu a sdílí některý kód. Aplikace pro watchOS byla vytvořena jako WatchKit Extension.

Obrázek 9: Základní obrazovka watchOS

4.18.1 Struktura RunnyWatcha

Jako hlavní entrypoint do aplikace slouží struktura, která implementuje protokol `App`. Uvnitř computed property o typu `some Scene` byla navrácena struktura o typu `WindowGroup`. Uvnitř této struktury je implementován vstupní view aplikace `EnvfironmentWrapper`. Uvnitř tohoto view jsou vytvořeny instance `Observable` objektů pro řízení aplikace. Struktura je anotována typem `@main`, který zajistí, že toto view bude aplikací přečteno jako první.

4.18.2 SwiftUILocation

Pro sledování vzdálenosti na systému watchOS byla implementována třída `SwiftUILocation`. Třída `SwiftUILocation` funguje podobně jako třída `LMO` v aplikaci Runny pro iOS, s tím rozdílem, že implementuje protokol `ObservableObject`. Properties, do kterých jsou zapisovány hodnoty, jsou typu `@Published`. Znamená to, že při změně hodnoty je automaticky aktualizováno uživatelské rozhraní, kde se proměnná vyskytuje. Při vývoji pro watchOS aplikace občas přestala po určité době sledovat vzdálenost uživatele. Řešením tohoto problému bylo vytvořit novou aktivitu pomocí `HKWorkoutConfffiguratfion`. Typ konfigurace byl nastaven na `.running` a `locationType` na `.outdoor`. Poté je tato konfigurace zapsána jako argument do třídy `HKWorkoutSession`. Aktivita je následně spuštěna pomocí volání `self.workoutSession?.startActivity(with: Date())` a `self.builder?.beginCollection(withStart: Date())`.

Další důležitou metodou uvnitř této třídy je metoda `getCandence`, která pomocí `pm.startUpdates` zapíše aktuální kadenci do proměnné `currentCadence` a přidá kadenci do pole `cadences` pro tvorbu grafu.

4.18.3 Stopwatch

Pro tvorbu stopek ve watchOS byl vytvořen View `ListAddView`. Tento view zobrazuje jednotlivé okruhy a tlačítko pro spuštění nebo zastavení stopek. Po spuštění běžecké aktivity se rozběhnou stopky. Stopky jsou vytvořeny pomocí volání `Timer.scheduledTimer(withTimeInterval:`

`0.01, repeats: true)`. To má za následek, že se stopky volají každou setinu sekundy. Do uzávěru funkce je dosazen zápis `seflff.secondsEflapsed += 0.01`, což znamená, že je k dané hodnotě přičtena setina sekundy. Do pole časových úseků je při stisknutí tlačítka Lap uložen aktuální čas. Následně jsou vypočteny rozdíly mezi jednotlivými časovými záznamy. Toto tvoří jednotlivé úseky, které jsou zobrazeny uživateli. Uvnitř metody `tfimeDfiffff` jsou vždy porovnány sousední časové úseky, čas je převeden na vteřiny a následně je zjištěn absolutní rozdíl hodnot pomocí globální funkce `abs`. Čas je naformátován v metodě `formatTime`. Pro zobrazení jednotlivých okruhů uživateli byl implementován view `List`.

4.18.4 WatchLaps

Pro zobrazení časů kilometrových a mílových okruhů v aplikaci pro watchOS byla implementována třída `WatchLaps`. Do inicializátoru této třídy byla dosazena instance běhu o typu `RunVM`. Poté byla pomocí třídy `UnitGetter` zjištěna používaná jednotka uživatele pomocí metody `getUnit`. Z této metody si můžeme zobrazit jak mílové okruhy, tak kilometrové. Aplikace během aktivity ukládá časy kilometrových okruhů pro obě jednotky (míle a kilometry). Pro zjištění nejrychlejšího a nejpomalejšího kilometru aplikace převede časy na sekundy pomocí metody `toSec` a zjistí minimální a maximální hodnotu. Tohoto bylo dosaženo pomocí následujícího zápisu: `filtered.map {$0.toSec()}.min()`. Následně je zjištěn index okruhu, který je nejrychlejší a nejpomalejší pomocí metody `fffirstIndexOff`. Konečný zápis vypadal následovně: `let indexOfMin = fffiltered.map {$0.toSec()}.ffirstIndex(off: mfin)!`.

Vzdálenost a čas je poté zapsán do pole `laps`. Pokud je aktuální index záznamu totožný s indexem nejryhlejšího nebo nejpomalejšího okruhu, je property `isFastest` nebo `isSlowest` nastavena na `true`.

4.18.5 WTime

Pro naformátování času, který uběhl od spuštění aktivity, byla implementována třída `WTime`. Třída do svého inicializátoru přijímá čas ve vteřinách. V computed property `createTime` je čas v sekundách převeden na hodiny minuty a vteřiny pomocí zápisu `let time = (seconds / 3600, (seconds % 3600) / 60, seconds % 60)`. Z této hodnoty byly přečteny minuty a vteřiny pomocí zápisu `let (_, m, s) = time`. Následně byla připsána před jednociferné hodnoty `0` a výsledný řetězec navrácen a následně zapsán do UI aplikace.

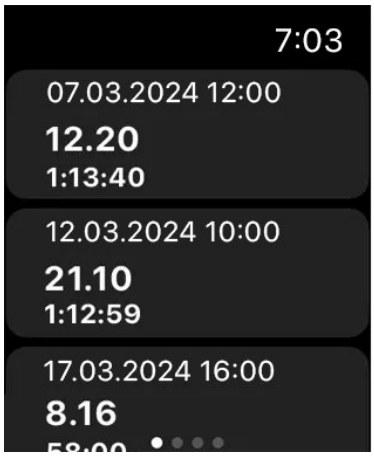
4.18.6 ActivitiesView

View `ActivitiesView` byl vytvořen pro zobrazení uložených běžeckých aktivit. Uvnitř tohoto view byl použit view `NavigationLink`, který se stará o otevření obrazovky s detailem běhu. Tento view je skrytý a objeví se až poté, co uživatel vybere aktivitu, jejíž detail chce zobrazit. Po výběru aktivity je do property `selectedModel` přiřazen běh, jehož detail chceme zobrazit. Tetnto model je použit v argumentu view `NavigationView`. Pomocí nastavení property `self.navigationViewIsActive` na `true` se otevře obrazovka s detailem. Tato hodnota je dosazena do argumentu `isActive` ve view `NavigationLink`.

4.18.7 RunCell

View `RunCell` byl použit pro vytvoření buňky s detailem běhu. Pro tvorbu tohoto view byla vytvořena struktura `ZStack` a její obsah byl zarovnán doleva pomocí zápisu `.leading`. Data o běhu byla vytvořena pomocí struktur `Text`. Tyto views byly zarovnány pod sebe pomocí struktury `VStack`. Tato struktura byla také zarovnána doleva pomocí zápisu `.leading`.

Pro zápis času byla použita třída `Time` a property `cleverTime`. Třída `Time` byla sdílena mezi aplikací pro iOS a aplikací pro watchOS.

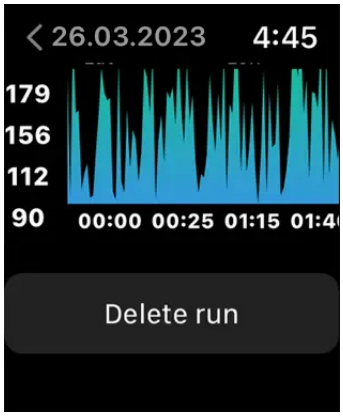


Obrázek 10: Historie aktivit watchOS

4.18.8 Tvorba grafu

Pro vytvoření grafu běžecké kadence a srdečního tepu na watchOS byla vytvořena struktura `MyShape`. Uvnitř metody `path(in rect: CGRect)` byla z dat nakreslena samotná křivka pomocí volání metod `path.move`, `path.addLine` a `path.closeSubpath()`. Jako pozadí grafu byl použit přechod vytvořený pomocí struktury `LinearGradient`. Tento byl grafu byl přiřazen pomocí modifikátoru `.gradient`.

Pro vytvoření popisku grafu byla vytvořena třída `AxisVM`. Do inicializátoru této třídy byla dosazena délka trvání běhu, minimální tepová frekvence a maximální tepová frekvence. Následně bylo potřeba vytvořit popisky os. Pro získání popisků os byly vytvořeny čtyři body. Nejprve byl definován počáteční bod jako bod 0, druhý body jako 25% délky trvání aktivity, tedy `dur * 0.25` a třetí body jako `dur * 0.75`. Jako poslední bod byla dosazena hodnota `dur`, což označuje celkovou délku trvání aktivity. Hodnoty pro svislou osu, na které se bude zobrazovat tepová frekvence, byly vytvořeny obdobným způsobem. Za tímto účelem byly implementovány metody `getTimes() a getHRS()`.



Obrázek 11: Graf běžecké kadence watchOS

4.18.9 SettingsView

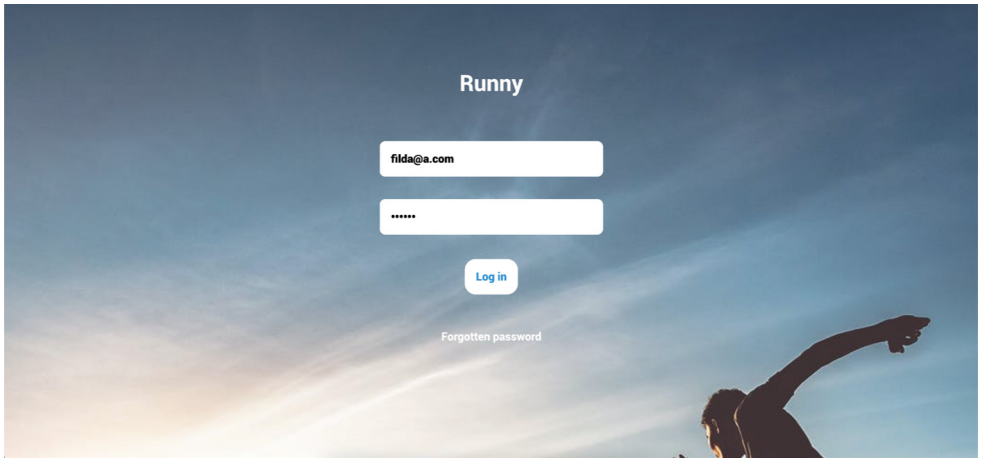
Pro tvorbu obrazovky s nastavením byl vytvořen view `SettingsView`. Pro změnu jednotky vzdálenosti bylo použito tlačítko pomocí struktury `Button`. Po stisknutí daného tlačítka je vedle jeho popisku přidán obrázek checkmarku pomocí struktury `Image(systemName: "checkmark")`. Symbol "checkmark" pochází z knihovny SF Symbols, která obsahuje stovky užitečných symbolů. Na této obrazovce je také možné aktivovat "Race Mode". Tento mód umožňuje uživateli vstoupit rovnou do efektivní fáze tréninku a přeskočit rozběhání a výklus.

5 Webová verze

Pro zálohování aktivit z běžecké aplikace byla vytvořena webová aplikace. Pro implementaci byla využita knihovna React a vlastní knihovna webových komponent. Tato knihovna byl vytvořena pro usnadnění tvorby často používaných webových komponent. Všechny její elementy jsou tvořeny pomocí volání `document.createEflement`. Knihovna je inspirována SwiftUI a tyto elementy lze upravovat pomocí podobných modifikátorů. Uvnitř knihovny jsou obsaženy elementy jako Image, `Text`, nebo `Column`, nebo `Row`. Tyto elementy jsou implementovány jako třídy v jazyce ES6. Pro propojení apliakce vytvořené v Reactu s touto knihovnou byla vytvořena komponent `LayoutHost`. Aktivitu je z aplikace možné zálohovat pomocí volby "Backup" v `ActivityControlleru`. Uživateli je poté zobrazena obrazovka s možností registrace nebo přihlášení. Po přihlášení nebo registraci je aktivita ihned zálohována. Pro zobrazení mapy a kreslení trasy uživatele je využíváno Google Maps API. Webová aplikace je dostupná na adrese www.runny-app.github.io.

5.1 Přihlašovací obrazovka

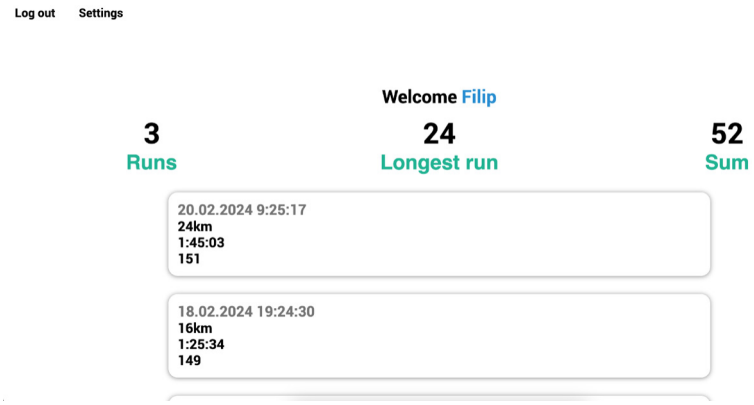
Přihlašovací obrazovka byla vytvořen pomocí komponentu Login. Uživatelské rozhraní obsahuje pole na e-mail a heslo. Dále byla vytvořeny tlačítka na přihlášení a zaslání zapomenutého hesla. Uživatel byl přihlášen pomocí volání metody `fffirebase.auth().sfignInWfithEmafiflAndPassword(emafifl, password)`. V případě chyby je tato chyba vypsána do uživatelského rozhraní. Pro zajištění funkcionality pro reset hesla byla implementován kompoent `MailReset` pomocí hooks. Properties uvnitř tohoto komponentu byly zapsány do hooku `React.useState`. Uživatel může zadat svůj e-mail a po stisknutí tlačítka "Send" je uživateli zaslán odkaz pro reset hesla a uživateli se zobrazí potvrzení s textem: "The reset e-mail has been sent. Check your inbox.".



Obrázek 12: Přihlašovací obrazovka web

5.2 Třída Wrapper

Tato třída slouží jako vstupní bod do webové aplikace. Pro vytvoření hlavní obrazovky byly implementovány komponenty `<HooksDataView>` a `<DataView>`. Pro tvorbu obrazovky s detailem aktivity a nastavením byly vytvořeny komponenty `<SettingsView>` a `<HooksDetailView>`. Navigace v aplikaci je řízena pomocí property `this.state.currentDetail`, která může nabýt hodnot “detail“, “settings“ a “home“. Podle této hodnoty je zobrazen odpovídající view. Každý komponent je vytvořen jako rozšíření třídy `React.Component` pomocí klíčového slova `extends`.



Obrázek 13: Hlavní obrazovka web

5.3 Třída RunFetcher

Třída `RunFetcher` slouží k načítání běžeckých aktivit uživatele z databáze Firestore. Za tímto účelem byla implementována statická metoda `fetch(uid)`. Z metody je navrácena instance třídy `Promise`.Tato třída umožňuje navrátit data z asynchronní operace pomocí metod `resolve` a reject. Metoda `resolve` je použita pro kladnou odpověď, metoda `reject` pro odpověď, u které nastala chyba. Uvnitř této třídy objektu je vytvořen přístup k databázi pomocí instance `fffirebase.fffirestore()`. Data byla načtena pomocí volání await `db.collection(“Runs”).doc(uid).collection(“MyRuns”).get()`. Konkrétní údaje o dané aktivitě si načteme pomocí metody data(). Například vzdálenost je načtena pomocí `d.data().distance`. Běh je zapsán do pole běhů `runs` pomocí metody `this.runs.push(obj)`. Pro navrácení běhů z této metody byla zavolána funkce `resolve()`.

5.4 Třída HooksDataView

Uvnitř této třídy byla vytvořena tlačítka pro odhlášení a otevření obrazovky s nastavením. Do elementu `h1` je zapsáno aktuální uživatelské jméno uživatele. Pro nastavení uživatelského jméno byl použit hook `const [nickname, setNickname] = React.useState(“”)`. Uvnitř metody `onAuthStateChanged` je zavolána metoda `setNickname` s aktuální přezdívkou uživatele v databázi. Po zavolání této metody je díky hooku automaticky přiřazena přezdívka uživatele do proměnné `nickname`. Toto uživatelské jméno je následně zobrazeno v uživatelském prostředí pomocí tagu `h1`.

5.5 Třída DataView

Třída `DataView` slouží k zobrazení statistik o aktivitách uživatele. Tato třída byla opět vytvořena jako rozšíření třídy `React.Component`. V této třídě bylo zkontrolováno, jestli je počet aktivit větší než 0 a byla vrácena instance třídy `GridView`. Uvnitř této třídy byly načteny běhy pomocí třídy `RunFetcher` do pole data. Toto pole bylo iterováno pomocí metody `map`, a pro každý běh byl vytvořen element `div` s daty o běhu. Pro datum aktivity byl vytořen nadpis o typu `h3` a pro vzdálenost a délku trvání titulek o typu `h2`. Do metody `onClick` byla pomocí `this props.switch(a)` zavolána metoda pro zobrazení child view. Za argument `a` byl dosazen běh z pole `runs`.

5.6 Třída GridView

Tato třída slouží pro zobrazení statistik běhu. Pro tvorbu této obrazovky byla použita má vlastní knihovna webových komponent. Nejprve byla uvnitř metody `getCard` vytvořena instance třídy `Card` a pomocí metody items byly do této třídy přidány položky. Hodnoty a popisky kategorií byly implementovány jako instance třídy `Text`. Responzivní velikosti písma bylo dosaženo pomocí modifikátoru “S4“ a “S5“. Tyto modifikátory automaticky přiřadí elementu fluid typografii. Do modifikátoru font byl dosazen font, který chceme použít - v tomto případě Helvetica. Pomocí modifkátoru `color("#1abc9c")` byla nastavena barva popisku na světle zelenou.

Pole volání metody `getCard` bylo dosazeno do instance třídy `Grid`, která také pochází z mé vlastní knihovny. Položky byly dosazeny do modifikátoru `items` a do konstruktoru třídy byla dosazena hodnota tři, což označuje rozložení do třech sloupců.

5.7 Detail aktivity

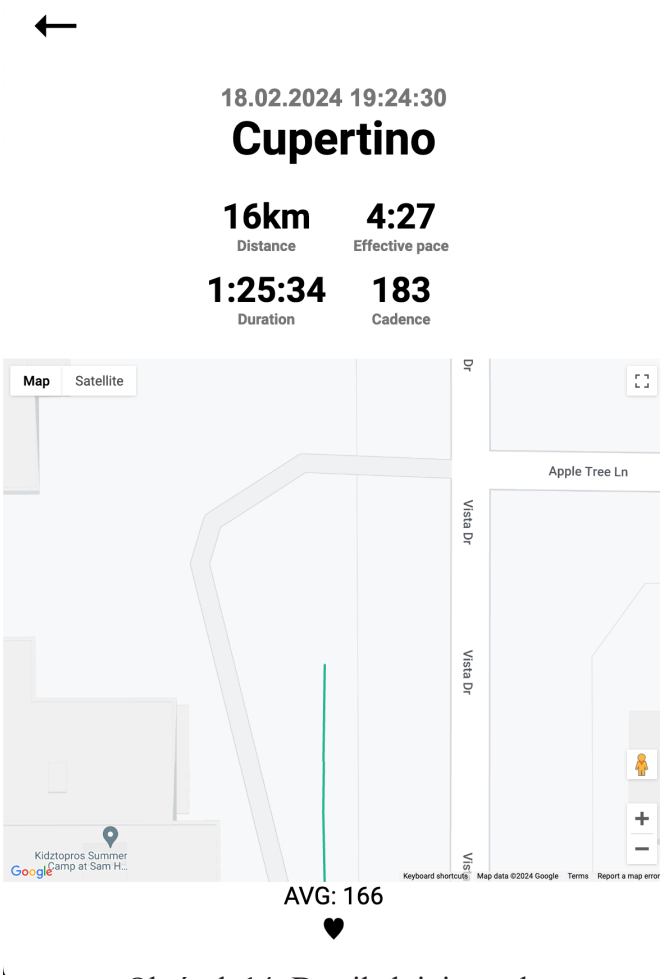
Pro zobrazení detailu aktivity byl vytvořen komponent `HooksDetailView`. V elementech `h3` a `h4` byly zobrazeny následující údaje:

- vzdálenost
- efektivní tempo
- uběhnutá vzdálenost
- doba trvání aktivity

Dále byla pomocí komponentu `TableComponent` zobrazeny okruhy jednotlivých kilometrů. Uvnitř této komponenty byla použita instance třídy `Table` z mé vlastní knihovny. Vnitřní okraj byl nastaven pomocí metody `cellPadding` a zarovnání na střed pomocí metody `cellAlign`. Záhlaví a sudé řádky tabulky byly obarveny pomocí metod `rowStyle` a `headStyle`.

5.8 Mapa

Pro tvorbu mapy byla zobrazena vytvořena třída `MapHost`. Mapa byla načtena uvnitř metody `componentDidMount` pomocí volání `new google.maps.Map`. Z `this props.coordinates` byly načteny souřadnice běhu. Pomocí volání `bounds.extend` byla velikost mapy přizpůsobena čáře znázorňující trasu běhu. Tato čára byla vytvořena pomocí metody `new google.maps.Polyline` a byla mapě přiřazena v metodě `setMap`.



Obrázek 14: Detail aktivity web

5.9 Grafy srdeční frekvence a kadence

Stejně jako v aplikacích pro iOS a watchOS podporuje webová aplikace zobrazení grafů kadence a tepové frekvence. Za tímto účelem byl vytvořen komponent `ChartComponent`. Do jeho properties byl dosazen objekt s hodnotami určujícími typ grafu, barvu grafu, popisek a samotná data.

Uvnitř tohoto komponentu byla implementována metoda `distributeDistance`, ve které byla vzdálenost rozdělena na rovnoměrné dílky pro popisek osy x. Pro samotný graf byla použita knihovna Chart.js. Graf byl vytvořen pomocí instance třídy `Chart`. Do konstruktoru této instance dosadíme objekt `data`. Uvnitř tohoto objektu použijeme v klíči `labels` metodu pro rovnoměrné rozmístění popisků osy x - `distributeDistance`. Do objektu `datasets` dosadíme barvu grafu, barvu okraje grafu, popisek a samotná data pro tvorbu grafu. Data se nachází v objektu `this.props.rates.hrArr`.

Pro tvorbu grafu běžecké kadence je použit stejný komponent `ChartComponent`. Za argument `hrArr` je v tomto případě pole s kadencemi `props.run.cadencesArray`. V tomto případě je nastavena barva grafu na zelenou a popisek na “Cadence“.

Obrázek 15: Grafy aktivity web

5.10 Komponenta Settings

Tato komponenta byla vytvořena pro umožnění přepínání jednotek mezi mílemi a kilometry. Uživatelské rozhraní obsahuje přepínač pomocí dvou tlačítek. Po stisknutí tlačítka je příslušná jednotka přiřazena property `unit` pomocí volání metody `setState` a příslušné tlačítko je zvýrazněno. Tato hodnota je zapsána do trvalého uložiště prohlížeče pomocí zápisu `window.localStorage.setItem(“unit”, unit)`. Po navrácení se zpět na hlavní obrazovku jsou všechny hodnoty v aplikaci přepočítány na správné jednotky.

ZÁVĚR

Cílem diplomové práce byl vývoj běžecké aplikace Runny pro platformy iOS, watchOS a web. Aplikace Runny měří běžecké metriky včetně efektivní kadence a srdečního tepu. Aplikace pro iOS je implementována pomocí SwiftUI a UIKit, aplikace pro watchOS je implementována pouze ve SwiftUI. Pro tvorbu webové verze byla použita knihovna React a vlastní knihovna webových komponent.

Práce se skládá z části teoretické a praktické. V teoretické části jsou popsány možnosti vývoje pro platformy iOS, watchOS a webovou platformu. Jsou popsány výhody a nevýhody programovací jazyků, knihoven pro tvorbu uživatelských rozhraní a backendových jazyků. Pro vývoj byla zvolena knihovna SwiftUI a UIKit pro aplikaci na iOS a watchOS. Jako technologii serverové strany byla zvolena Firebase. Pro tvorbu uživatelského rozhraní webové aplikace byla zvolena mnou vytvořená knihovna responzivních webových komponent a knihovna React.

V praktické části byly identifikovány a popsány funkční a nefunkční požadavky pro aplikaci Runny. Dále byl navržen vzhled obrazovek a uživatelského rozhraní aplikace odpovídající funkcionálním a nefunkcionálním požadavkům. Aplikace pro iOS byla implementována pomocí UIKit a SwitUI. Nejprve byl vytvořen `ViewController` pro tvorbu hlavní obrazovky, následně `HistoryController` pro zobrazení historie běhů a `ActivityController` pro zobrazení detailu běhu. Pro zobrazení statistiky běhů byly vytvořeny třídy `StatsController` a `SummaryController`. Pomocí SwiftUI byla implementována aplikace pro systém watchOS. Aplikace umožňuje zálohování aktivit pomocí Firestore. iOS aplikace využívá knihovnu Firebase a aplikace pro watchOS POST request. Pro zálohování aktivit byla implementována webová aplikace. Její frontendová část byla vytvořena pomocí Reactu a vlastní knihovny webových komponent. Jako backend byla použita dokumentová databáze Firebase Firestore.

Obě aplikace jsou zveřejněny na App Store. Webová aplikace je zveřejněna na adrese www.runny-app.github.io. Aplikace umožňuje kromě zaznamenávání běžných běžeckých metrik zaznamenávání efektivního tempa, efektivní kadence a efektivního srdečního tepu. Tímto aplikace umožňuje vyhodnotit statistické hodnoty uvedených veličin. Toto je velice důležité při tvorbě lepší tréninkové struktury. Výsledkem je dosažení lepších výkonů a předcházení zranění. Aplikace má v současnosti více než 400 stažení od uživatelů z celého světa a má mezi běžci kladný ohlas.

SEZNAM POUŽITÉ LITERATURY

[1] Objective-C - In The Pocket [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://tech.inthepocket.com/technology/objective-c

[2] Apple Confirms iPhone SDK Coming Next Year | Digital Trends [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.digitaltrends.com/mobile/apple-confirms-iphone-sdk-coming-next-year/

[3] Number of apps from the Apple App Store 2023 | Statista [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.statista.com/statistics/268251/number-of-apps-in-the-itunes-app-store-since-2008/

[4] How to help ensure you only install apps from the App Store in the European Union – Apple Support (UK) [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://support.apple.com/en-gb/118128#:~:text=the%20App%20Store.-,Open%20the%20Settings%20app%2C%20then%20tap%20App%20Installation.,set%20to%20the%20App%20Store.

[5] macOS Sonoma - Apple [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.apple.com/macos/sonoma/

[6] Xcode on the Mac App Store [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://apps.apple.com/us/app/xcode/id497799835?mt=12/

[7] Account Management - Support - Apple Developer [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/support/account/

[8] Choosing a Membership - Support - Apple Developer [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/support/compare-memberships/

[9] Testing in Simulator versus testing on hardware devices | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/xcode/testing-in-simulator-versus-testing-on-hardware-devices

[10] Apple unveils groundbreaking new technologies for app development - Apple (CZ) [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.apple.com/cz/newsroom/2019/06/apple-unveils-groundbreaking-new-technologies-for-app-development/

[11] Cocoa (Touch) [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html

[12] iOS 17.1 Runtime Headers - SwiftUI - SwiftUI.UpdateCoalescingCollectionView.h [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.limneos.net/?ios=17.1&framework=SwiftUI.framework&header=SwiftUI.UpdateCoalescingCollectionView.h

[13] Chapter 4. Foundation: The Objective-C Standard Library - Cocoa® Programming Developer's Handbook, Second Edition [Book] [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.oreilly.com/library/view/cocoa-programming-developers/9780321647986/ch04.html

[14] max() | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/data/1780052-max

[15] min() | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/data/1779816-min

[16] NumberFormatter | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/numberformatter

[17] Measurement | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/measurement

[18] Calendar | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/calendar

[19] Locale | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/locale

[20] DateFormatter | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/dateformatter

[21] TimeZone | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/timezone

[22] Timer | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/timer

[23] invalidate() | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/timer/1415405-invalidate

[24] localizedStringWithFormat(_:_:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swift/string/localizedstringwithformat(_:_:)

[25] trimmingCharacters(in:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swift/stringprotocol/trimmingcharacters(in:)/

[26] Range | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swift/range

[27] replaceSubrange(_:with:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swift/array/replacesubrange(_:with:)-6a2ai

[28] compare(_:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/foundation/nsstring/1414082-compare

[29] WindowGroup | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/windowgroup

[30] View | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z:

https://developer.apple.com/documentation/swiftui/view/

[31] PreviewProvider | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/previewprovider/

[32] Just a moment... [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/how-to-create-and-compose-custom-views

[33] preferredColorScheme(_:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/view/preferredcolorscheme(_:)

[34] previewDevice(_:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/view/previewdevice(_:)

[35] previewLayout(_:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/view/previewlayout(_:)

[36] colorScheme | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/environmentvalues/colorscheme

[37] sizeCategory | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/environmentvalues/sizecategory

[38] Just a moment... [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.hackingwithswift.com/books/ios-swiftui/how-layout-works-in-swiftui

[39] How to Use scenePhase in SwiftUI | DeveloperMemos [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developermemos.com/posts/scene-phase-swiftui

[40] SwiftUI Cookbook, Chapter 5: Format Text in SwiftUI | Kodeco [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.kodeco.com/books/swiftui-cookbook/v1.0/chapters/5-format-text-in-swiftui

[41] Padding in SwiftUI: A Comprehensive Guide [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://blog.schurigeln.com/padding-in-swiftui/

[42] SwiftUI | ViewModifier | .border() | Codecademy [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.codecademy.com/resources/docs/swiftui/viewmodifier/border

[43] Color | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/color

[44] How to create custom view modifiers in SwiftUI | Sarunw [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://sarunw.com/posts/how-to-create-custom-view-modifier-in-swiftui/

[45] Mastering SwiftUI Image View [online] Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.swiftyplace.com/blog/mastering-swiftui-image-view

[46] AsyncSequence explained with Code Examples - SwiftLee [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.avanderlee.com/concurrency/asyncsequence/

[47] withTaskGroup(of:returning:body:) | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swift/withtaskgroup(of:returning:body:)

[48] UIKit | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/uikit

[49] UIDevice | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/uikit/uidevice/

[50] UIScreen | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/uikit/uiscreen/

[51] UIScene | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/uikit/uiscene

[52] Window | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/window

[53] Utilizing the Debug View Hierarchy to better understand your apps UI | by Lene Whiteley | Stacc | Medium [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://medium.com/stacc/utilizing-the-debug-view-hierarchy-to-better-understand-your-apps-ui-c8655f6d34e9

[54] Mastering Layouts in SwiftUI: Understanding Views, Stacks, and Modifiers. | by Parpods | Bootcamp [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://bootcamp.uxdesign.cc/mastering-layouts-in-swiftui-understanding-views-stacks-and-modifiers-32cd8747d0f1

[55] Mastering Layouts in SwiftUI: Understanding Views, Stacks, and Modifiers. | by Parpods | Bootcamp [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://bootcamp.uxdesign.cc/mastering-layouts-in-swiftui-understanding-views-stacks-and-modifiers-32cd8747d0f1

[56] Divider in SwiftUI - Everything you need to know | Sarunw [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://sarunw.com/posts/swiftui-divider/

[57] Layout priorities in SwiftUI | Swift with Majid [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://swiftwithmajid.com/2020/04/15/layout-priorities-in-swiftui/

[58] What is the fixedSize modifier in SwiftUI | Sarunw [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://sarunw.com/posts/swiftui-fixedsize/

[59] Just a moment... [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.hackingwithswift.com/books/ios-swiftui/alignment-and-alignment-guides

[60] SwiftUI List Tutorial: Grow from Beginner to Advanced | by Chase | Medium [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://medium.com/@jpmtech/swiftui-list-from-beginner-to-merlin-5308261b78b6

[61] Drag and Drop List In SwiftUI. In this article, We will explore how to… | by Mobile Apps Academy | Medium [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://mobileappsacademy.medium.com/drag-and-drop-list-in-swiftui-30b53682d447

[62] Just a moment... [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/how-to-make-a-scroll-view-move-to-a-location-using-scrollviewreader

[63] Managing user interface state | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/managing-user-interface-state

[64] TextField | Apple Developer Documentation [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/documentation/swiftui/textfield

[65] Submit for review - Manage submissions to App Review - App Store Connect - Help - Apple Developer [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/help/app-store-connect/manage-submissions-to-app-review/submit-for-review/

[66] Choosing a Membership - Support - Apple Developer [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/support/compare-memberships/

[67] The ‘what’ and ‘why’ of iOS signing certificates and provisioning profiles — and how to manage them as your team grows | Runway [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.runway.team/blog/ios-certificates-provisioning-profiles-large-teams

[68] App Store Connect - Apple Developer [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/app-store-connect/

[69] App Review Guidelines - Apple Developer [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.apple.com/app-store/review/guidelines/

[70] Front End vs Back End - Difference Between Application Development - AWS [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://aws.amazon.com/compare/the-difference-between-frontend-and-backend/#:~:text=Frontend%20and%20backend%20are%20two,that%20make%20your%20application%20work.

[71] SQL CREATE DATABASE Statement [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.w3schools.com/sql/sql_create_db.asp

[72] PHP Introduction [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://www.w3schools.com/php/php_intro.asp

[73] Node.js — Run JavaScript Everywhere [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://nodejs.org/en

[74] Firebase | Google’s Mobile and Web App Development Platform [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://firebase.google.com/

[75] JavaScript Guide - JavaScript | MDN [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

[76] Conditional Rendering | Vue.js [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://vuejs.org/guide/essentials/conditional.html

[77] Angular [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://angular.io/guide/component-overview

[78] ASP.NET Core | Open-source web framework for .NET [online]. Copyright ©2024 [cit.8.4.2024]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet

[79] NOAKES, Tim. Lore of running. 4. OXFORD UNIVERSITY PRESS, 2001. ISBN 0195780167.

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH