**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**
**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

# ДИПЛОМНА РАБОТА

**по професия код 481020 „Системен програмист“**
**специалност код 4810201 „Системно програмиране“**

Тема: Мобилно приложение за следене на резултати и други характеристики на състезания и включване в тях.

Дипломант:
*Йордан Николаев Марков*

Дипломен ръководител:
*маг. инж. Марин Ранчев*

СОФИЯ

2 0 2 3

# ОТЗИВ НА ДИПЛОМЕН РЪКОВОДИТЕЛ

# УВОД

Данните показват, че мобилните устройства се използват все по-често от настолните и преносимите компютри, защото предлагат богат избор от функции и интерфейси, които не се срещат никъде другаде. Например, потребителите имат достъп до мобилни платформи, като iOS и Android, които дават възможност за достъп до различни приложения и услуги.

Мобилните приложения оставят своя отпечатък върху съвременния живот, превръщайки го в по-динамичен и удобен. Те дават на хората възможност да използват мобилно устройство, за да видят и взаимодействат с различни съдържания, да получат информация и да си изпълняват дейностите в дигиталното пространство.

Често срещано е в днешно време по време на маратони да се използва Facebook или ръчно записване на участници. Следенето на резултати става трудно и неприятно за повечето любители на спорта. Вече е възможно чрез мобилните устройства, човек да се записва за такива мероприятия, но платформите са много индивидуални и затворени в себе си, като не предлагат глобална възможност за преизползване и за други маратони и така допринасят за трудностите при записване, следене на резултати и други.

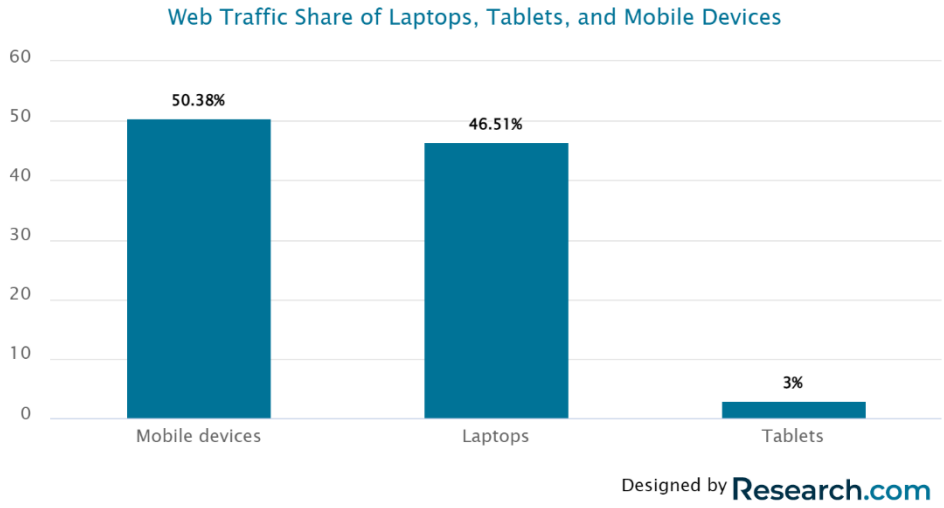
Този дипломен проект има за цел да предостави възможността на всеки един потребител да бъде организатор на свой собствен маратон или да бъде участник в такъв, предоставяйки му информация за неговите резултати.

# ПЪРВА ГЛАВА

# Методи и технологии за реализиране на мобилно приложение

## 1.1. Защо мобилно приложение?

В ежедневието на един модерен човек голям процент от времето е запълнено с използването на мобилни устройства *(Фиг. 1.1.) [1]*. Много по-практично е да се използва смартфон, за да се следят резултати, защото не се задължават потребителите да седят пред настолен компютър или лаптоп. Менажирането на данни за участници може да става директно и по време на маратона, като се изисква само интернет свързаност.




*Фиг. 1.1.: Употреба на мобилни устройства сравнено с лаптопи и таблети.*

## 1.2. Основни технологии за разработка на мобилни приложения

### 1.2.1. Класифициране на мобилните приложения

Мобилните приложения могат да бъдат класифицирани в две основни категории: многоплатформени и едноплатформени.

#### 1.2.1.1. Многоплатформени приложения

Многоплатформени приложения са тези, които могат да работят на различни платформи, като iOS, Android, или дори на десктопни операционни системи, като Windows или MacOS. Те се разработват чрез технологии, като React Native, Xamarin или Flutter, които позволяват използването на един код база за различните платформи.

- **React Native *[2]***

React Native е отворено-кодова софтуерна рамка, създадена от Facebook, която се използва за разработване на многоплатформени приложения за Android и iOS. Приложението има една кодова база, която работи на много операционни системи, което означава, че е нужно да се създаде само едно приложение. React Native продължава да нараства в популярност, като възползва най-новите JavaScript стандарти, за да направи разработването на мобилно приложение още по-лесно и ефективно.

- **Xamarin *[3]***

Microsoft създаде Xamarin през 2011 г. като технология за разработване на мобилни приложения, които могат да работят както на iOS, така и на Android. С Xamarin разработчиците могат да споделят до 90% от своя код между основните платформи. Xamarin използва един език, C#, за създаване на приложения за всички мобилни платформи. За разлика от други технологии, Xamarin се компилира естествено (в самата операционна система), което го прави подходящ вариант за изграждане на високопроизводителни приложения с естествен вид и усещане. Платформата има два основни продукта за компилиране: Xamarin.iOS и Xamarin.Android. В случая на iOS изходният код се компилира директно в ARM 8 код (предварителна компилация), докато за приложенията за Android Xamarin първо се компилира до междинен език и след това до собствен код по време на изпълнение. И в двата случая обаче процесът е автоматизиран и може да се справя с проблеми като разпределение на паметта, събиране на отпадъци и съвместимост на платформата по подразбиране. С други думи, Xamarin е технология, която позволява създаването на мобилни приложения, които работят както на iOS, така

и на Android. Технологията е нативно компилирана и позволява на разработчиците да използват C# като език за програмиране. Платформата има два основни продукта, Xamarin.iOS и Xamarin.Android, които са в състояние автоматично да управляват разпределението на паметта, събирането на боклука и съвместимостта на платформата.

- **Flutter *[4]***

Flutter е иновативна технология, която набира много популярност в настояще време и се развива много бързо. Програмирането се извършва чрез Dart, което позволява бърз и ефективен анализ, разработка на потребителски интерфейси и поправяне на грешки в кратък срок. Разработчиците могат да видят промените, които правят в реално време, което го прави много по-лесно за използване. Flutter е лесен за създаване на красив и функционален потребителски интерфейс, което е атракционно за потребителите. Без него, едно приложение е трудно да бъде успешно на пазара. Flutter е съвместим с различни IDE, включително Android Studio и VS Code и документацията е ясна, проста и лесно разбираема.

#### 1.2.1.2. Едноплатформени приложения

Едноплатформени приложения, от своя страна, са тези, които работят само на една платформа, като iOS или Android. Те се разработват специфично за една платформа и могат да използват устройствени функционалности, като камерата или GPS, по-лесно отколкото многоплатформените приложения.

- **Swift *[5]***

Swift се използва за разработка на приложения за iOS и е един от първите езици, които се придвижват при разработка на такъв софтуер. Създаден е през 2014 година и е оптимизиран за лесна и безопасна разработка. За създаване на потребителски интерфейси се използва UIKit и Storyboard, като се въвежда и SwiftUI, който предлага реално време наблюдение на промени, като Flutter.

- **Kotlin *[6]***

Kotlin е език, който се конкурира със Swift и се използва за разработка на приложения за Android. Създаден е малко по-рано от Swift през 2011 г. и работи

на Java Virtual Machine (JVM). Въпреки че неговият синтаксис не е съвместим с Java, Kotlin е проектиран да взаимодейства с Java код и разчита на съществуващата Java Class Library, като например рамката на колекциите. По същество Kotlin може да се използва за различни цели, но основната му употреба е в мобилната среда.

- **Objective-C *[7]***

Objective-C е език за програмиране, който се използва за разработка на софтуер за iOS и е предшественик на C. Той предлага обектно-ориентирани възможности, добавя синтаксис за дефиниране на класове и методи и поддържа обектни литерали. Objective-C се отличава с динамичното свързване и зареждане на време на изпълнение. Той беше създаден преди 38 години и все още се използва.

- **Java *[8]***

Java е един от най-разпространените обектно-ориентирани езици за разработка на софтуер за компютри и мобилни приложения, в това число и за Android. Синтаксисът му е подобен на C и C++. Java има преимуществото да може да пренася софтуер от компютър към мобилно приложение. Разликата с JavaScript е, че Java се компилира до битов код и след това се интерпретира и изпълнява чрез JVM. Kotlin е негов наследник.

## 1.3. Проучване на развойни среди за разработка на мобилни приложения

Развойните среди за разработка на мобилни приложения се използват от разработчици за създаване и интегриране на мобилни приложения. Те могат да включват уеб-базирани инструменти, конзоли, редактори и програмна рамка, които помагат на разработчиците да разработват, тестват и доведат до активно състояние мобилни приложения.

Ето няколко популярни развойни среди за разработка на мобилни приложения:

- Xcode - развойна среда за разработка на приложения за iOS, macOS, watchOS и tvOS.
- Android Studio - развойна среда за разработка на приложения за Android.

## 1.4. Проучване на облачни технологии, които предоставят услуги в реално време

Едни от примерите за облачни технологии, които предоставят услуги в реално време, са:

- Amazon Web Services (AWS)
- Microsoft Azure
- Firebase (Google)
- IBM Cloud

Те се използват от много организации за съдържане на приложения, данни и други технологии в облака, за да предоставят услуги в реално време на своите клиенти.

- **Amazon Web Services (AWS) *[9]***

Amazon Web Services (AWS) е облачния платформен доставчик от Amazon, който предлага множество инфраструктурни и платформени услуги в облака. AWS е една от най-големите и най-популярните платформи за облачна възпроизводимост и инфраструктура, която поддържа стабилна и мащабируема инфраструктура за приложения и услуги.

AWS предлага множество услуги, като виртуални сървъри, бази данни, аналитични услуги, комуникационни услуги и много други. Той позволява на компаниите да използват възможностите на облака без да се притесняват за инфраструктурните изисквания, необходими за поддържане на приложения. Още, AWS предлага множество инструменти и услуги за управление и администриране на облачните инфраструктури, които помагат за оптимизирането на ресурсите и

за улеснение на управлението на инфраструктурата. Това включва инструменти за мониторинг, одит, автоматизация и управление на резервно копие.

AWS е достъпен в много региони по целия свят и предлага гъвкавост и еластичност при разработването на приложения. Той също така предлага гъвкавост при закупуването на ресурси, като позволява на компаниите да се абонират за по месец или по година, в зависимост от нуждите им.

В общия случай, Amazon Web Services (AWS) предлага мощна и гъвкава облачна инфраструктура, която може да поддържа различни видове приложения и услуги. Той е отличен избор за компании, които търсят ефективност, гъвкавост и икономия при управлението на своя инфраструктура

- **Microsoft Azure *[10]***

Microsoft Azure е облачна платформа, разработена от Microsoft. Тя предлага множество услуги и инструменти за разработка, управление и възпроизвеждане на приложения. Azure е достъпен в много региони по целия свят и предлага гъвкавост и еластичност при разработването и менажирането на приложения.

Azure предлага множество възможности за управление на инфраструктурата, като инструменти за мониторинг, одит, автоматизация и управление на резервно копие. Azure позволява на разработчиците да работят с много програмни езици и технологии, като Azure Functions, Azure Container Instances, Azure Kubernetes Service и други.

Azure също така предлага услуги за анализа и хранене на данни, включително Azure SQL Database, Azure Cosmos DB и Azure Data Lake Storage. Той позволява на компаниите да анализират, хранят и използват своите данни в ефективен и безопасен начин. Това включва и възможност за обработка на големи обеми данни, аналитика в реално време и възможност за възстановяване на данни в случай на взрив.

Още една важна част на Azure е еластичността му да работи с други продукти и услуги от Microsoft, като Office 365, Dynamics 365 и Power Platform. Това позволява на компаниите да интегрират своите бизнес приложения и процеси в

една уеб-базирана екосистема, която подобрява ефективността и производителността.

- **Firebase (Google) *[11]***

Firebase е мобилна и уеб платформа, разработена от Google. Тя предлага множество инструменти и услуги, които поддържат разработчиците при създаването, публикуването и поддържането на мобилни и уеб приложения.

Firebase предлага функционалност за автентикация, уеб поддръжка, бази данни, аналитика, уведомления и много други. Тя позволява на разработчиците да създават и управляват приложения бързо и лесно, без да е необходимо да поддържат инфраструктурата или да се грижат за малки детайли.

Firebase е гъвкава и еластична платформа, която може да се интегрира с други услуги и продукти на Google, като Google Cloud Platform. Тя е популярна сред разработчиците заради нейната елегантност и простота при използване, както и за многото удобни инструменти, които предоставя. Тя е безплатна до определен обем от трафик и ресурси, а по-нататъшно може да се плати по месечен план, в зависимост от използването и нуждите на разработчика.

- **IBM Cloud *[12]***

IBM Cloud е облачна платформа, разработена от IBM. Тя предлага възможност за разработка, управление и разгръщане на приложения, инфраструктура и услуги. IBM Cloud поддържа множество операционни системи, включително Linux, Windows и мобилни платформи, като iOS и Android.

IBM Cloud предлага услуги за възможности за машинно обучение и анализи, като Watson, както и бази данни, интеграция с приложения, поддържане на приложения и много други. Тя позволява на разработчиците да разработват и разгръщат приложения бързо и лесно, като предлага много удобни инструменти за управление и мониторинг на приложенията.

IBM Cloud е гъвкава и еластична платформа, която може да се интегрира с други услуги и продукти на IBM. Тя е популярна сред големи компании и предприятия, които искат да имат достъп до висококачествени облачни услуги и инструменти за разработка на приложения. Тя също така е достъпна за малки и

средни предприятия, които търсят достъпна и надеждна платформа за управление на своите приложения.

## 1.5. Съществуващи решения

Съществуват много решения като някои от тях са:

- Wings for Life World Run
- BMW BERLIN-MARATHON
- TCS Amsterdam Marathon 2022
- Athens Marathon and Half
- TCS New York City Marathon
- RunCzech App
- И други.

- **Wings for Life World Run *[13]***

Wings for Life World Run е социално приложение за забавление, което се използва в мрежата на Wings for Life. То предоставя на потребителите възможност да участват в световно състезание, и да подкрепят инициативата на Wings for Life да помогне за набиране на средства за инвалидите. Приложението позволява на потребителите да се регистрират в състезанието и да изберат място и дата за започване. След това могат да съберат пари за дарение и да проследят своя резултат в реално време. Приложението също така предоставя и потребителски профил, където хората могат да направят снимки и да споделят историите си с други потребители в мрежата.

- **BMW BERLIN-MARATHON *[14]***

BMW Berlin-Marathon е приложение за iOS и Android, което предоставя предварителна информация и детайли за Берлинския Маратон. Приложението предоставя пълни подробности относно разписанието на състезанието, маршрута и данните за предстоящи развлечения. Може да се използва приложението, за да се създаде профил и да се следят личните прогреси. Приложението дава възможност да се присъединява към безброй други бегачи и да се проследява и да се сравняват личните резултати. В приложението има и карта, която показва маршрута на маратона, както и места на точките за хранене и напитки.

Приложението предлага и набор от функции, които помагат справянето с маратона.

- **TCS Amsterdam Marathon 2022 *[15]***

TCS Amsterdam Marathon 2022 е мобилно приложение, което предоставя информация и инструменти за подготовка за Амстердамския маратон през 2022 година. Приложението включва пътеводител, карти, дневник за тренировка, календар за записване на предстоящи събития и др. Приложението също предоставя информация за маршрутите, стартовите зони и много други. Освен това, приложението включва информация за протоколите, стартовите пакети, информация за присъствието на фенове и други полезни детайли. Приложението предоставя и връзка към официалния сайт на Амстердамския маратон и други медии, които са в подкрепа на маратона.

- **Athens Marathon and Half *[16]***

Athens Marathon and Half е мобилно приложение, създадено за да помогне на всички участници в Атинския маратон и полумаратон. Приложението предлага много информация за събитието, включително разписание на трасета, карти на маршрута, информация за регистрация, интерактивна карта на събитието и много други. Приложението осигурява участниците с много полезни инструменти за планиране на бъдещите си състезания и предоставя информация за избраните им състезания. Приложението също така предоставя възможност за свързване на участниците с други любители на бега и поддържа дебати и интерактивни дискусии. Приложението има и мобилен калкулатор за скорост, който помага на бегачите да измерват точното си време и да проследят своя прогрес.

- **TCS New York City Marathon *[17]***

TCS New York City Marathon е официално приложение за Маратона в Ню Йорк. То предоставя пълен обзор на маратона и предоставя възможност за следене на другите участници по време на маратона. Приложението предоставя подробна информация за маршрута на маратона, разписание на автобусите и предоставя информация за това как да се присъединява към маратона. Приложението предоставя и информация за предстоящите събития по време на маратона, както

и други полезни ресурси, като актуална карта и мултимедийни източници. Освен това, приложението дава възможност на регистрираните участници да изпращат и получават SMS-и и имейли по време на маратона. За да подобри изживяването на участниците, приложението предоставя лесна и бърза навигация, достъп до музикални плейлисти и живи потоци на радиото, както и достъп до социални медии.

- **RunCzech App *[18]***

RunCzech App е мобилно приложение, създадено от компанията RunCzech, за да подобри изпитанието на бегачите в централна Чехия. Приложението предлага пътеводител по маршрути в централна Чехия, както и маркирани и немаркирани трасета за бягане. Потребителите могат да планират своите маршрути, да използват маркираните трасета за бягане, да проследяват своя прогрес и дори да видят своите лични постижения. Приложението осигурява точна информация за всеки маршрут, като показва дължината на маршрута, превозно средство както и интересни точки на маршрута. Приложението може да се свърже с други приложения за спорт и да помогне да поддържането на профила за бягане. В приложението има и карта на беговите маршрути в Чехия, която показва всички маркирани пътища и актуалнато трасе за бягане. Край това, приложението предлага и информация за разни събития в централна Чехия, както и различни връзки към други бегови приложения.

# ВТОРА ГЛАВА

# Проектиране и избиране на технологии на iOS мобилно приложение „LapIt”

## 2.1. Основни функционални изисквания

Целите, които трябва да се постигнат, за да се изпълни дипломният проект, са следните:

- Да се определи вида на акаунта – потребителски или организаторски.
- Включване в състезание.
- Преглед на история на предишни състезания.
- Следене на резултати на състезания.
- Създаване на състезания.
- Задаване на вида на състезанието – по време или по дистанция.
- Въвеждане на данни за участниците.
- Запазване на състезанието като модел за преизползване.

## 2.2. Избор на технология

Изборът на технология е Swift – едноплатформена. Когато става въпрос за разработка на приложение, което е за iOS, от малкото езици за разработката му се избира Swift, като съществува и Objective-C, който е всъщност е основоположникът му. В Swift съществуват две основни програмни рамки, които могат да бъдат полезни в разработването на едно мобилно приложение. Това са UIKit & Swift.

- **UIKit *[19]***

UIKit е програмна рамка, разработена от Apple, и е използвана за създаването на графичен интерфейс и потребителски интерфейс за iOS и iPadOS приложения. UIKit съдържа много готови елементи за интерфейс, като бутони, таблици,

елементи за избор, и други, които може да се използват от разработчиците за бързо създаване на приложения с привлекателен интерфейс.

UIKit също предлага възможност за получаване и менажиране на събития, свързани с потребителския интерфейс, като например натискане на бутон или сваляне на плъзгач. Това позволява на разработчиците да направят приложението си интерактивно и да отговаря на действията на потребителя.

В UIKit също има възможност за настройка на дизайна и поддръжка на множество различни устройства и разрешения на екрана.

- **SwiftUI *[20]***

SwiftUI е програмна рамка, разработена от Apple, за създаване на приложения за iOS, iPadOS, macOS, watchOS и tvOS. SwiftUI е интегрирана с езика за програмиране Swift и предлага гъвкав и удобен начин за създаване на интерфейс за потребители.

SwiftUI използва декларативен подход към създаването на интерфейс, където разработчикът определя как интерфейсът трябва да изглежда, вместо да пише код, който контролира индивидуалните елементи на интерфейса. Това позволява на разработчиците да създават интерфейси с по-малко код и по-бързо, и да осигурят по-добро интегриране с другите части на приложението.

SwiftUI включва много готови компоненти, като бутони, плъзгачи, сегментирани контроли и други, които могат да бъдат използвани за създаване на приложения с интуитивен интерфейс. Те също така могат да бъдат модифицирани и персонализирани, за да отговарят на индивидуалните нужди на проекта.

SwiftUI включва и автоматично адаптиране на интерфейса, което означава, че интерфейсът може да се преобразува и да се оптимизира в зависимост от различните устройства, на които се използва приложението.

Като заключение, SwiftUI предлага по-бърз и ефективен начин за създаване на интерактивни приложения с декларативен интерфейс и автоматично адаптиране. Тя е идеална за разработчици, които търсят лесен и бърз начин за създаване на интерактивни приложения за Apple платформи.

- **Кое от двете?**

В този проект се използва SwiftUI. Предпочита се пред UIKit, защото SwiftUI предлага много предимства в сравнение с UIKit. Едно от главните предимства е декларативния подход, който позволява на разработчиците да описват как интерфейсът трябва да изглежда, вместо да пишат код, който да контролира всеки елемент на интерфейса. Това осигурява по-лесно и по-бързо създаване на интерфейси, както и по-добро интегриране с останалата част на приложението.

Друго предимство е автоматичното адаптиране на интерфейса към различните устройства и резолюции, което позволява на разработчиците да се фокусират върху функционалността на приложението, а не върху поддържането на много различни интерфейси.

## 2.3. Избор на облачна технология

Платформата Firebase е избрана за облачната технология, защото е лесна за използване с добре описана документация. За съхранението на данни Cloud Firestore е предпочитан пред Realtime Database, тъй като използва метод, базиран на документи и колекции, който е по-удобен от подхода на ключ и стойност (JSON).

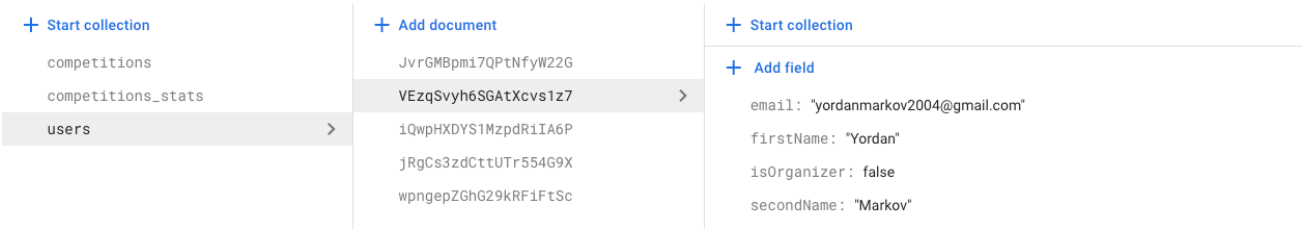
Cloud Firestore е употребен за създаването на следните колекции от документи:

- “users”
- “competitions”
- “competitions_stats”

- **Структура на документи в колекцията “users” *(Фиг. 2.1.)***

  Всеки потребител има:

- email: променлива от тип низ от символи, която служи за запазване на имейла на потребителя; това е уникалният идентификатор; също така служи за смяна на парола и влизане в акаунта.
- firstName: променлива от тип низ от символи, която служи за запазване на първото име на потребителя.
- secondName: променлива от тип низ от символи, която служи за запазване на второто име на потребителя.
- isOrganizer: променлива от булев тип, която служи за запазване на състоянието на акаунта – потребителски или организаторски.

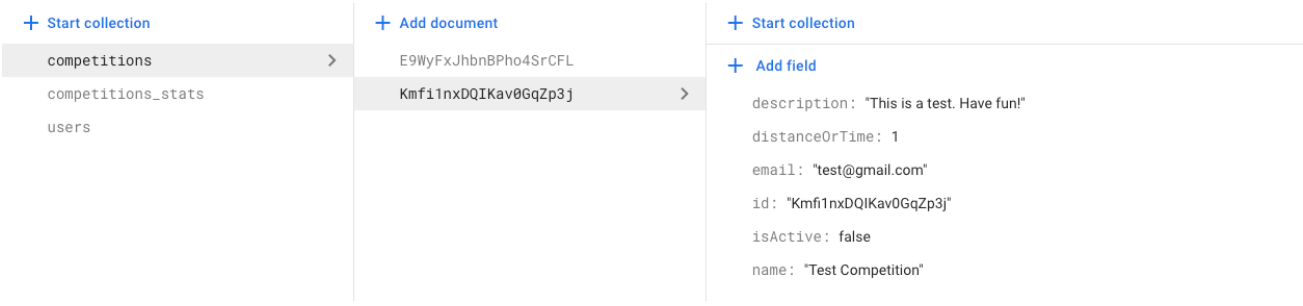



*Фиг. 2.1.: Структурата на документи в колекцията „users”.*

- **Структура на документи в колекцията “competitions” *(Фиг. 2.2.)***

  Всяко състезание има:

- id: променлива от тип низ от символи, която служи като уникален идентификатор на състезанието.
- email: променлива от тип низ от символи, която служи за запазване на имейла на организатора, който е създал състезанието.
  - name: променлива от тип низ от символи, която служи за запазване на името на състезанието.
- description: променлива от тип низ от символи, която служи за запазване на описанието на състезанието.
- distanceOrTime: променлива от тип целочислено число, която определя по какъв начин ще се определи ранглиста.
- isActive: променлива от булев тип, която определя дали състезанието е активно (дали е отворено за присъединяване на участници и за редактиране или не).

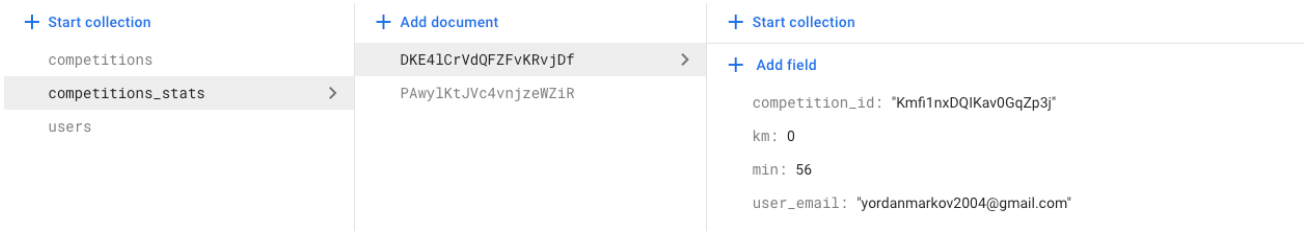



*Фиг. 2.2.: Структурата на документи в колекцията „competitions”.*

- **Структура на документи в колекцията “competitions_stats”** ***(Фиг. 2.3.)***

Всяка връзка между човек и състезание (резултати) има:

- competition_id: променлива от тип низ от символи, която служи за запазване на уникалния идентификатор на състезанието.
- user_email: променлива от тип низ от символи, която служи за запазване на имейла на потребителя, който участва в това състезание.
- km: променлива от тип целочислено число, която служи за запазване на километрите, които участникът е извървял. (При състезание от тип distance се използва тази променлива.)
- min: променлива от тип целочислено число, която служи за запазване на минутите, за които участникът е минал цялото състезание. (При състезание от тип time се използва тази променлива.)




*Фиг. 2.3.: Структурата на документи в колекцията „competitions_stats”.*

## 2.4. Избор на развойна среда

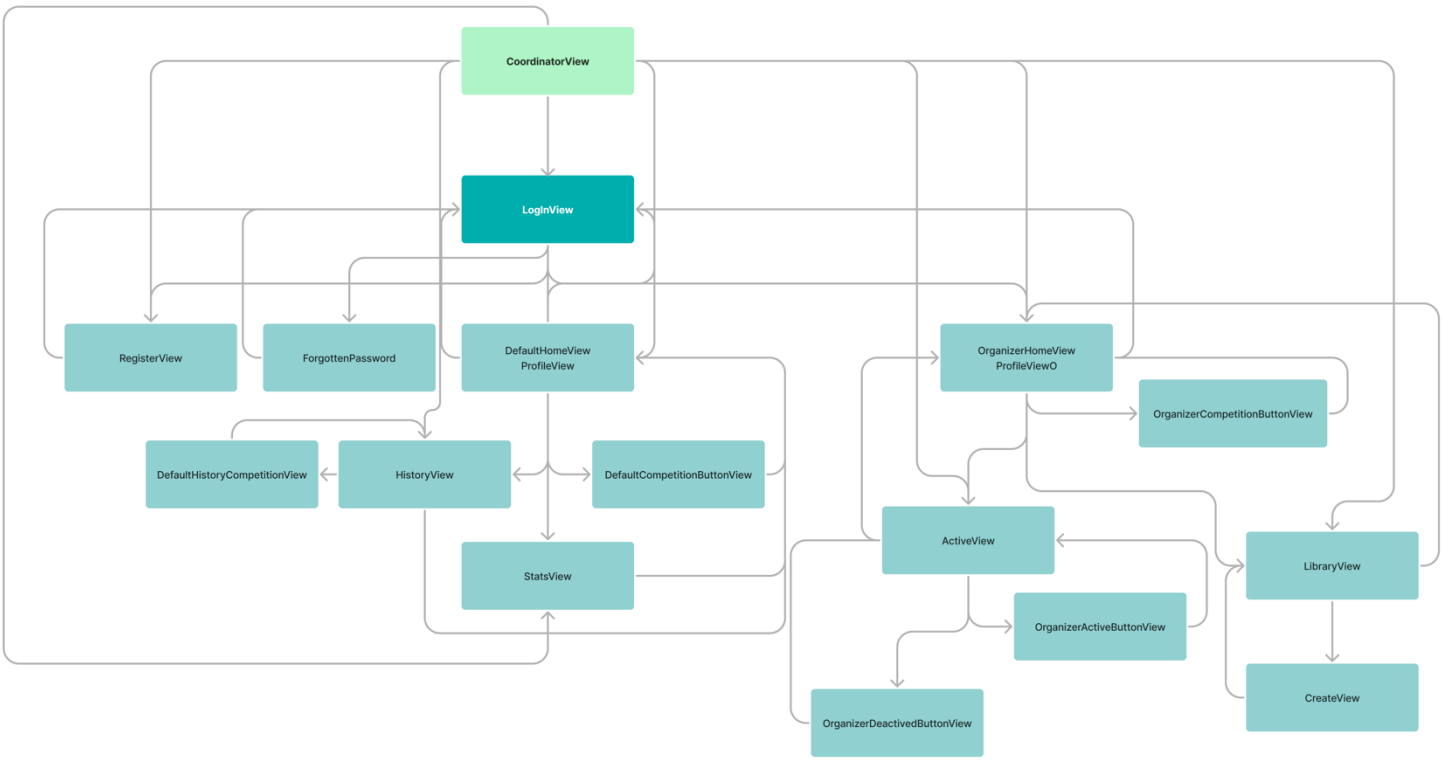
За разработка на мобилното приложение е избрана развойната среда Xcode пред другите, защото е напълно безплатна, предоставена от Apple и има големи възможности за разработка на iOS и macOS приложения. Тя е лесна за използване и предоставя мощни инструменти за отстраняване на грешки, профилиране и оптимизация, които помагат за улесненото разработване на качествено приложение.

## 2.5. Структура на приложението

### 2.5.1. Структура на екраните (Фиг. 2.4.) *[21]*

За разлика от UIKit, в SwiftUI екраните се описват и настройват чрез код, вместо Storyboard, като се използват модели на View. В SwiftUI, елементите на екрана се дефинират в модела на View, а презентацията им е декларирана и визуално описана на интерфейса, като автоматично се обновява при промяна на модела на View.

Използва се CoordinatorView (с ViewModel: Coordinator), който автоматично сменя изгледите на приложението според текущия раздел. Задава се в главния файл на приложението (LapItApp), което съдържа структурата за стартиране на приложението.




*Фиг. 2.4.: Структура на екраните на приложението.*

### 2.5.2. Структура на базите данни

На фигури *(Фиг. 2.1.)*, *(Фиг. 2.2.)*, *(Фиг. 2.3.)* може да се види къде се съхранява цялата информация на приложението. За да се съхраняват данните локално, информацията извлечена от Firestore се декодира в различни променливи и структури от данни (Състезание – Competition, Потребител - User) *(Фиг. 2.5.)* чрез функции/заявки, които се викат от ViewModel-ите на приложението.

| Competition | |
|---|---|
| id | String |
| name | String |
| description | String |
| distanceOrTime | Int |
| isActive | Bool |

| User | |
|---|---|
| user_email | String |
| firstName | String |
| secondName | String |
| km | Int |
| min | Int |

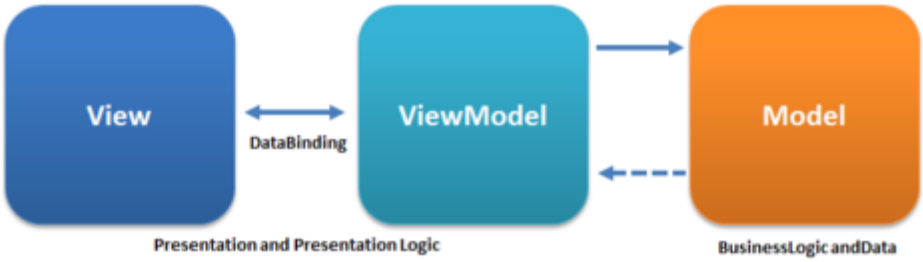
*Фиг. 2.5.: Две често срещани структури и тяхното съдържание.*

### 2.5.3. Структура на кода

При разработката на приложението с програмната рамка SwiftUI се избра да се следва архитектурата MVVM.

- **Какво е MVVM? *[22]***

MVVM (Model-View-ViewModel) *(Фиг. 2.6.)* е архитектурен шаблон за програмиране, който се използва по-често в приложения с уеб или мобилни платформи. Той се използва за разделяне на логиката за данни и представяне от интерфейса на потребителя в приложението. Основната идея е да се отделят моделите (Models) (данните), изгледите (Views) (външен вид на приложението) и ViewModel-ите (логиката за данни) в различни части, за да може да се поддържа по-лесно и да се използва по ефективен начин.




*Фиг. 2.6.: MVVM архитектура.*

# ТРЕТА ГЛАВА

# Програмна реализация на iOS мобилно приложение „LapIt”

## 3.1. Използване на координатор за смяна на екрани (изгледи).

Както вече бе споменато, използва се class Coordinator (Фиг. 3.1.), който сменя изгледите на приложението спрямо текущия раздел (Tab).

```
class Coordinator: ObservableObject {
    static var shared = Coordinator()
    enum Tab {
        case defaultHome
        case history
        case stats
        case organizerHome
        case library
        case active
        case login
        case register
    }

    @Published var currTab = Tab.login
    private  let network: Network

    lazy var registerViewModel: RegisterViewModel = {
        RegisterViewModel(network: self.network, coordinator: self)
    }()

    lazy var logInViewModel: LogInViewModel = {
        LogInViewModel(network: self.network, coordinator: self)
    }()

    lazy var defaultHomeViewModel: DefaultHomeViewModel = {
        DefaultHomeViewModel(network: self.network, coordinator: self)
    }()
```

```
    lazy var organizerHomeViewModel: OrganizerHomeViewModel = {
        OrganizerHomeViewModel(network: self.network, coordinator: self)
    }()

    lazy var historyViewModel: HistoryViewModel = {
        HistoryViewModel(network: self.network, coordinator: self)
    }()

    lazy var statsViewModel: StatsViewModel = {
        StatsViewModel(network: self.network, coordinator: self)
    }()

    lazy var libraryViewModel: LibraryViewModel = {
        LibraryViewModel(network: self.network, coordinator: self)
    }()

    lazy var activeViewModel: ActiveViewModel = {
        ActiveViewModel(network: self.network, coordinator: self)
    }()

    init() {
        self.network = Network()
    }

    func route(to newTab: Tab) {
        self.currTab = newTab
    }
}
```

*Фиг. 3.1.: Class Coordinator.*

Класът наследява протокола „ObservableObject“, който го прави способен да публикува промени в своите притежания и да известява своите наследници за тези промени.

В тялото на класа има дефинирани осем различни променливи от тип ViewModel, като всички са „lazy“ (т.е. създават се само при първоначалното им извикване). Тези променливи са отговорни за контролиране на бизнес логиката и състоянието на съответните екрани на приложението.

Класът има и една списъчна променлива с име „Tab“, която описва възможните раздели на приложението.

В конструктора на класа се инициализира променливата „network“ с нова инстанция на класа „Network“, който се използва за взаимодействие с облачната услуга Firebase.

Функцията „route“ се използва за промяна на текущо избрания раздел в приложението.

Общо взето, класът „Coordinator“ предоставя централизиран контрол върху логиката на приложението и координира неговото поведение, като осигурява свързаност между различните ViewModel променливи и техните съответни View (изгледи) екрани.

## 3.2. Интеграция на Firebase в проекта

Вече е ясно, че Firebase е избран като предпочитана облачна услуга в проекта. Той предлага много важни функционалности, като удостоверяване и достъп до бази данни. Удостоверяването на входа в приложението се осъществява чрез услугата Authentication. Всяка информация, необходима за функционирането на приложението, се съхранява във Firestore Database. За да се оптимизира работата с Firebase, е дефиниран class Network *(Фиг. 3.2.)*, който има за задача да управлява връзката с базите данни и услугата за удостоверяване на входа.

```
class Network {

    private lazy var firebaseAuth: Auth = {
        Auth.auth()
    }()

    private lazy var firestore: Firestore = {
        Firestore.firestore()
    }()
    // More code here…
}
```

*Фиг. 3.2.: Class Network.*

Класът Network има две частни свойства, firebaseAuth и firestore, които представляват инстанции на класовете Auth и Firestore от Firebase SDK съответно.

В този клас се съдържат и множество функции, които се повикват в последствие във ViewModel-ите.

- **signIn** ***(Фиг. 3.3.)***

```
func signIn(email: String, password: String) async throws {
    try await firebaseAuth.signIn(withEmail: email, password:
password)
}
```

*Фиг. 3.3.: Функция „signIn”.*

Тази функция служи за вписването на потребител в системата. Тя приема два параметъра - имейл и парола, и използва firebaseAuth, който е екземпляр на класа Auth от Firebase, за да извърши вход в системата. Ако има грешка при опит за вход в системата, функцията хвърля грешка чрез използване на ключовата дума „throws“.

- **signOut** ***(Фиг. 3.4.)***

```
func signOut() async throws {
    try firebaseAuth.signOut()
}
```

*Фиг. 3.4.: Функция „signOut”.*

Тази функция служи за изписването на потребител от системата. Тя извиква метода signOut() на инстанцията на класа firebaseAuth. Методът signOut() е предоставен от Firebase Authentication и се използва за излизане от сесията на потребителя в приложението.

- **sendPasswordReset** ***(Фиг. 3.5.)***

```
func sendPasswordReset(email: String) async throws {
    try await firebaseAuth.sendPasswordReset(withEmail: email)
}
```

*Фиг. 3.5.: Функция „sendPasswordReset”.*

Тази функция служи за смяна на парола. При забравена парола на потребителя – тя се вика и приема неговия имейл (email), за да му се изпрати имейл за смяна

на паролата. В тялото на функцията се извиква методът sendPasswordReset на обекта firebaseAuth.

- **register** ***(Фиг. 3.6.)***

```
func register(email: String, password: String, firstName: String,
secondName: String, isOrganizer: Bool) async throws {
    try await firebaseAuth.createUser(withEmail: email, password:
    password)
    firestore.collection("users").addDocument(data: ["email":
    email, "firstName": firstName, "secondName": secondName,
    "isOrganizer": isOrganizer])
}
```

*Фиг. 3.6.: Функция „register”.*

Тази функция приема параметри email, password, firstName, secondName и isOrganizer, които се използват за създаване на новия потребителски акаунт. Първо, функцията използва метода createUser на firebaseAuth, който създава и потвърждава акаунта на потребителя по зададените имейл и парола. След това, функцията записва информацията за новия потребител в колекция на базата данни на Firestore. Това става чрез използване на метода addDocument, който добавя нов документ към колекцията "users" с полетата "email", "firstName", "secondName" и "isOrganizer", съответно попълнени със стойностите от параметрите.

- **deleteAccount** ***(Фиг. 3.7.)***

```
func deleteAccount(email: String) async throws {
        try await firebaseAuth.currentUser?.delete()

        let usersCollection = firestore.collection("users")
        let query = usersCollection.whereField("email",
isEqualTo: email)
        let querySnapshot = try await query.getDocuments()
        let userData = querySnapshot.documents.first
        try await userData?.reference.delete()

        let competitionsCollection =
firestore.collection("competitions")
        let query2 =
competitionsCollection.whereField("email", isEqualTo: email)
        let querySnapshot2 = try await query2.getDocuments()
        let competitionsData = querySnapshot2.documents
        competitionsData.forEach { document in
            document.reference.delete()
        }

        let competitionsStatsCollection =
firestore.collection("competitions_stats")
        let query3 =
competitionsStatsCollection.whereField("user_email",
isEqualTo: email)
        let querySnapshot3 = try await query3.getDocuments()
        let competitionsStatsData = querySnapshot3.documents
        competitionsStatsData.forEach { document in
            document.reference.delete()
        }
    }
```

*Фиг. 3.7.: Функция „deleteAccount”.*

Тази функция има за цел да изтрие акаунт на потребител. Първоначално се опитва да изтрие потребителския акаунт от Firebase Authentication, използвайки метода delete() на firebaseAuth.currentUser. След това търси и изтрива всички документи в колекцията "users", които съдържат потребителското имейл на потребителя, за да изтрие информацията за потребителя от базата данни.

След това функцията търси всички състезания, в които е участвал потребителят, в колекцията "competitions" и ги изтрива. След това търси и изтрива всички статистически данни за тези състезания в колекцията "competitions_stats", които са свързани с потребителя.

Функцията приема имейл на потребителя като аргумент и може да хвърли грешка, ако не успее да изтрие акаунта или някоя от колекциите.

- **getCurrentUserEmail *(Фиг. 3.8.)***

```
func getCurrentUserEmail() async throws -> String {
        if let user = firebaseAuth.currentUser {
            return user.email ?? ""
        } else {
            throw NSError(domain: "UserNotFound", code: 0,
userInfo: nil)
        }
    }
```

*Фиг. 3.8.: Функция „getCurrentUserEmail”.*

Тази функция връща имейл адреса на текущият потребител, ако има влязла сесия. Използва се firebaseAuth.currentUser за да се получи информацията за влезлият потребител и ако има такъв - се връща имейл адреса му. Ако няма влязъл потребител, хвърля грешка от тип NSError, която може да сигнализира, че потребителят не е намерен.

- **getUserOrganizerStatus** ***(Фиг. 3.9.)***

```
func getUserOrganizerStatus(email: String) async throws ->
Bool {
        let userCollection = firestore.collection("users")
        let query = userCollection.whereField("email",
isEqualTo: email)
        let querySnapshot = try await query.getDocuments()
        let userData = querySnapshot.documents.first
        let isOrganizer = userData?["isOrganizer"] as? Bool ??
false
        return isOrganizer
    }
```

*Фиг. 3.9.: Функция „getUserOrganizerStatus”.*

Тази функция приема имейл като параметър и връща булева стойност, която определя дали потребителят с този имейл е организатор или не.

Първоначално се получава колекцията от потребители във firestore базата данни и се прави заявка, която филтрира потребителите по техния имейл, за да се намери потребителят с дадения имейл адрес. След това се извличат данните на потребителя от querySnapshot.documents.first. Имейл адресът на потребителя е уникален и не е възможно да има повече от един резултат в querySnapshot.documents.

Накрая се извлича стойността на isOrganizer от данните на потребителя и се връща като резултат. Ако стойността на isOrganizer не може да бъде извлечена от данните на потребителя, тогава се връща стойността по подразбиране, която е false.

- **getUserFirstName** ***(Фиг. 3.10.)***

```
func getUserFirstName(email: String) async throws -> String {
        let userCollection = firestore.collection("users")
        let query = userCollection.whereField("email", isEqualTo:
email)
        let querySnapshot = try await query.getDocuments()
        let userData = querySnapshot.documents.first
        let firstName = userData?["firstName"] as? String ?? ""
        return firstName
}
```

*Фиг. 3.10.: Функция „getUserFirstName”.*

Тази функция търси потребител в колекцията "users" във Firestore базата данни, който има имейл, съвпадащ с този, подаден като аргумент. След това връща първото име на този потребител, ако такъв е намерен, като използва данните, записани във Firestore. Ако потребителят не е намерен, функцията връща празен низ.

- **getUserSecondName** ***(Фиг. 3.11.)***

```
func getUserSecondName(email: String) async throws -> String {
        let userCollection = firestore.collection("users")
        let query = userCollection.whereField("email", isEqualTo:
email)
        let querySnapshot = try await query.getDocuments()
        let userData = querySnapshot.documents.first
        let secondName = userData?["secondName"] as? String ?? ""
        return secondName
}
```

*Фиг. 3.11.: Функция „getUserSecondName”.*

Тази функция приема същия аргумент (имейл) и прави почти същото като функцията „getUserFirstName”, но връща второто име на потребителя. Отново, ако такъв потребител не е намерен – функцията връща празен низ.

- **getUserKm** ***(Фиг. 3.12.)***

```
func getUserKm(email: String) async throws -> Int {
        let competitionsStatsCollection =
firestore.collection("competitions_stats")
        let query =
competitionsStatsCollection.whereField("user_email", isEqualTo:
email)
        let querySnapshot = try await query.getDocuments()
        var totalKm = 0
        for document in querySnapshot.documents {
            let data = document.data()
            if let km = data["km"] as? Int {
                totalKm += km
            }
        }
        return totalKm
}
```

*Фиг. 3.12.: Функция „getUserKm”.*

Тази функция се използва за изчисляване на общите километри направени от потребител с даден имейл. Тя използва колекцията "competitions_stats" от Firestore, която съдържа всички въведени потребителски данни за самото състезание. Функцията търси всички документи в колекцията, които имат дадения като аргумент имейл на потребителя. След това разглежда всеки документ и извлича полето „km“ от него и го добавя към общия брой километри. След това функцията връща общите километри на потребителя, ако такива са намерени.

- **getUserMin** ***(Фиг. 3.13.)***

```
func getUserMin(email: String) async throws -> Int {
        let competitionsStatsCollection =
firestore.collection("competitions_stats")
        let query =
competitionsStatsCollection.whereField("user_email", isEqualTo:
email)
        let querySnapshot = try await query.getDocuments()
        var totalMin = 0
        for document in querySnapshot.documents {
            let data = document.data()
            if let min = data["min"] as? Int {
                totalMin += min
            }
        }
        return totalMin
}
```

*Фиг. 3.13.: Функция „getUserMin”.*

Тази функция приема същия аргумент (имейл) и прави почти същото като функцията „getUserKm”, но с разликата, че чете информацията от полето „min”. Връща общата стойност на минутите на потребителя, ако такива са намерени.

- **createCompetition** ***(Фиг. 3.14.)***

```
func createCompetition(email: String, name: String, description:
String, distanceOrTime: Int, isActive: Bool) {
        let documentRef =
firestore.collection("competitions").addDocument(data: ["id": "",
"email": email, "name": name, "description": description,
"distanceOrTime": distanceOrTime, "isActive": isActive])
        let documentID = documentRef.documentID

firestore.collection("competitions").document(documentID).updateDat
a(["id": documentID])
}
```

*Фиг. 3.14.: Функция „createCompetition”.*

Тази функция приема няколко аргумента - имейл на потребителя, име на състезанието, описание, разстояние или време и дали състезанието е активно. При създаване на самото състезание, по подразбиране състезанието е първоначално активно.

След това функцията създава нов документ в колекцията "competitions" във Firestore, използвайки подадените аргументи. След това извлича уникалния идентификатор на документа и го добавя като ново поле в документа. Това е нужно, за да се предостави уникален идентификатор за всяко състезание.

- **joinCompetition** ***(Фиг. 3.15.)***

```
func joinCompetition(competition_id: String, user_email: String) {
        let data: [String: Any] = [
            "competition_id": competition_id,
            "user_email": user_email,
            "min": 0,
            "km": 0
        ]

firestore.collection("competitions_stats").addDocument(data: data)
}
```

*Фиг. 3.15.: Функция „joinCompetition”.*

Тази функция се използва за регистриране на потребител като участник в дадено състезание. Тя използва колекцията "competitions_stats" от Firestore. Функцията добавя документ в колекцията с данните, които са подадени като аргументи (ID (уникален идентификатор) на състезанието и имейл на потребителя). Този документ има полетата „competition_id“, „user_email“, „min“ и „km“, първите две от които представляват аргументите на функцията, а последните две представляват броя минути и километри, записани за потребителя.

- **leaveCompetition** ***(Фиг. 3.16.)***

```
func leaveCompetition(competition_id: String, user_email: String)
async throws {
        let competitionStatsCollection =
firestore.collection(„competitions_stats“)
        let query =
competitionStatsCollection.whereField(„competition_id“, isEqualTo:
competition_id).whereField(„user_email“, isEqualTo: user_email)
        let querySnapshot = try await query.getDocuments()
        let competitionStatsData = querySnapshot.documents
        competitionStatsData.forEach { document in
            document.reference.delete()
        }
}
```

*Фиг. 3.16.: Функция „leaveCompetition”.*

Тази функция се използва за изтриване на потребителски данни от колекцията „competitions_stats“ от Firestore. Тя приема два аргумента: идентификатор на състезанието и имейла на потребителя. След това функцията използва заявка, за да търси документи в колекцията с дадените аргументи. След това функцията цикли през всички документи, които бяха намерени, и изтрива всеки от тях.

- **isAlreadyJoined** ***(Фиг. 3.17.)***

```
func isAlreadyJoined(competition_id: String, user_email: String)
async throws -> Bool {
        let competitionsStatsCollection =
firestore.collection("competitions_stats")
        let query =
competitionsStatsCollection.whereField("competition_id", isEqualTo:
competition_id).whereField("user_email", isEqualTo: user_email)
        let querySnapshot = try await query.getDocuments()
        let competitionsStatsData = querySnapshot.documents
        if competitionsStatsData.count > 0 {
            return true
        } else {
            return false
        }
}
```

*Фиг. 3.17.: Функция „isAlreadyJoined".*

Тази функция има за цел да прави проверка дали даден потребител с даден имейл е вече включен в дадено състезание. Тя приема два аргумента: „competition_id" - идентификатор на състезанието и „user_email" - имейл на потребителя. Функцията използва „competitions_stats" колекцията от Firestore, която съдържа данни за потребителските статистики за състезанието.

Първо, функцията създава заявка в „competitions_stats" колекцията с whereField метода, за да избере всички документи, които имат стойност на „competition_id" полето, съвпадаща с подадения „competition_id" аргумент, и имат стойност на „user_email" полето, съвпадаща с подадения „user_email" аргумент.

След това функцията прави заявка за получаване на документите, съответстващи на тази заявка, чрез използване на getDocuments метода.

След като получи заявката, функцията разглежда върнатите документи в „competitions_stats" колекцията и проверява дали има документ, за който „competition_id" и „user_email" съвпадат с подадените като аргументи. Ако има

такъв документ, функцията връща true. Ако няма такъв документ, функцията връща false.

- **getActiveCompetitions** ***(Фиг. 3.18.)***

```
func getActiveCompetitions() async throws -> [String: Any] {
    var competitions = [String: Any]()
    let competitionsCollection =
    firestore.collection("competitions")
    let query = competitionsCollection.whereField("isActive",
    isEqualTo: true)
    let querySnapshot = try await query.getDocuments()
    let competitionsData = querySnapshot.documents
    competitionsData.forEach { document in
        let competition = document.data()
        competitions[document.documentID] = competition
    }
    return competitions
}
```

*Фиг. 3.18.: Функция „getActiveCompetitions”.*

Тази функция проверява дали даден потребител с даден имейл вече е включен в дадено състезание. Тя използва колекцията „competitions_stats“ от Firestore, която съдържа всички потребителски данни за самото състезание. Функцията създава заявка в тази колекция с дадения като аргументи идентификатор на състезанието и имейл на потребителя. След това тя прави заявка и получава снимка от документите в резултат от заявката. След това тя разглежда тези документи и проверява дали има документ, за който идентификаторът и имейлът съвпадат с подадените като аргументи. Ако има такъв документ, функцията връща true. Ако няма такъв документ, функцията връща false.

- **getInactiveCompetitions** ***(Фиг. 3.19.)***

```
func getInactiveCompetitions() async throws -> [String: Any] {
        var competitions = [String: Any]()
        let competitionsCollection =
firestore.collection("competitions")
        let query = competitionsCollection.whereField("isActive",
isEqualTo: false)
        let querySnapshot = try await query.getDocuments()
        let competitionsData = querySnapshot.documents
        competitionsData.forEach { document in
            let competition = document.data()
            competitions[document.documentID] = competition
        }
        return competitions
}
```

*Фиг. 3.19.: Функция „getInactiveCompetitions”.*

Тази функция предоставя масив от всички неактивни състезания във Firestore. Тя използва колекцията „competitions“, за да избере всички документи, които съдържат поле "isActive", което е равно на false. След това функцията обхожда всеки документ и запазва данните в масив. Накрая функцията връща масива с данните.

- **getActiveAndJoinedCompetitions** ***(Фиг. 3.20.)***

```
func getActiveAndJoinedCompetitions(user_email: String) async
throws -> [String: Any] {
        var competitions = [String: Any]()
        let activeCompetitions = try await getActiveCompetitions()
        for (key, value) in activeCompetitions {
            let isAlreadyJoined = try await
isAlreadyJoined(competition_id: key, user_email: user_email)
            if isAlreadyJoined {
                competitions[key] = value
            }
        }
        return competitions
}
```

*Фиг. 3.20.: Функция „getActiveAndJoinedCompetitions”.*

Тази функция извлича активните състезания, в които е вписан даден потребител. Тя използва две други функции за да достигне това - getActiveCompetitions() за извличане на активните състезания и другата за проверяване дали потребителя е вписан в дадено състезание *(Фиг. 3.17.)*. Тя връща колекция с ключ номера на състезанието и всичките му полета.

- **getInactiveAndJoinedCompetitions** ***(Фиг. 3.21.)***

```
func getInactiveAndJoinedCompetitions(user_email: String) async
throws -> [String: Any] {
        var competitions = [String: Any]()
        let activeCompetitions = try await
getInactiveCompetitions()
        for (key, value) in activeCompetitions {
            let isAlreadyJoined = try await
isAlreadyJoined(competition_id: key, user_email: user_email)
            if isAlreadyJoined {
                competitions[key] = value
            }
        }
        return competitions
}
```

*Фиг. 3.21.: Функция „getInactiveAndJoinedCompetitions”.*

Тази функция се използва за взимане на списък от неактивни и присъединени от потребител с даден имейл състезания. Тя извиква функцията getInactiveCompetitions(), за да вземе всички неактивни състезания и преброй през всички документи в тази колекция. За всеки документ проверява дали дадения потребител се е присъединил към него с функцията isAlreadyJoined(). Ако е правда, тогава добавя данните за състезанието в резултата и връща всички присъединени и неактивни състезания за дадения потребител.

- **deactivateCompetitionById** ***(Фиг. 3.22.)***

```
func deactivateCompetitionById(competition_id: String) async throws
{
        let competitionCollection =
firestore.collection("competitions")
        let query = competitionCollection.document(competition_id)
        try await query.updateData(["isActive": false])
}
```

*Фиг. 3.22.: Функция „deactivateCompetitionById”.*

Тази функция има за цел да деактивира определено състезание в колекцията „competitions“ във Firestore. Тя приема един аргумент – „competition_id“, който е идентификаторът на състезанието, което трябва да бъде деактивирано.

След като функцията получи идентификатора на състезанието, тя създава заявка за избиране на документа в колекцията „competitions“ с този идентификатор, използвайки document метода. След това функцията използва updateData метода, за да промени стойността на полето „isActive“ на false. Ако заявката бъде изпълнена успешно, тогава състезанието ще бъде деактивирано и ще има стойност false за полето „isActive“ във Firestore.

- **activateCompetitionById *(Фиг. 3.23.)***

```
func activateCompetitionById(competition_id: String) async throws {
        let competitionCollection =
firestore.collection("competitions")
        let query = competitionCollection.document(competition_id)
        try await query.updateData(["isActive": true])
}
```

*Фиг. 3.23.: Функция „activateCompetitionById”.*

Тази функция има за цел да активира определено състезание в колекцията „competitions“ във Firestore. Работи по същия начин като deactivateCompetitionById *(Фиг. 3.22.)*, с разликата, че стойността на полето „isActive” се променя на true.

- **deleteCompetition** ***(Фиг. 3.24.)***

```
func deleteCompetition(competition_id: String) async throws {
        let competitionCollection =
firestore.collection("competitions")
        let query = competitionCollection.document(competition_id)
        try await query.delete()
        let competitionStatsCollection =
firestore.collection("competitions_stats")
        let query2 =
competitionStatsCollection.whereField("competition_id", isEqualTo:
competition_id)
        let querySnapshot = try await query2.getDocuments()
        let competitionStatsData = querySnapshot.documents
        competitionStatsData.forEach { document in
            document.reference.delete()
        }
}
```

*Фиг. 3.24.: Функция „deleteCompetition”.*

Тази функция има за цел да изтрие състезание и всички свързани с него статистически данни. Първо, тя изтрива документа на състезанието от колекцията “competitions” чрез използване на метода delete на даден документ. След това функцията търси всички документи в колекцията “competitions_stats”, които имат стойност на полето “competition_id”, равна на “competition_id” аргумента на функцията. Тя получава тези документи чрез използване на whereField метода и след това използва getDocuments метода, за да получи QuerySnapshot обект. Накрая, функцията обхожда всеки документ от получения масив от документи и го изтрива от Firestore чрез използване на метода delete на референцията на документа.

- **getUserById** ***(Фиг. 3.25.)***

```
func getUsersById(competition_id: String) async throws -> [String: Any] {
        var users = [String: Any]()
        let competitionsStatsCollection = firestore.collection("competitions_stats")
        let query = competitionsStatsCollection.whereField("competition_id", isEqualTo: competition_id)
        let querySnapshot = try await query.getDocuments()
        let usersData = querySnapshot.documents
        usersData.forEach { document in
            let user = document.data()
            users[document.documentID] = user
        }
        return users
}
```

*Фиг. 3.25.: Функция „getUsersById”.*

Тази функция търси всички потребители, които са регистрирани за дадено състезание във Firestore, като използва колекцията „competitions_stats“. Функцията извършва заявка, за да намери всички документи в колекцията, които имат поле „competition_id“, равно на подадения аргумент „competition_id“. След това функцията обхожда всеки документ и добавя данните за потребителите в речник, като използва „documentID“ като ключ за речника, а данните на потребителя като стойност. Накрая функцията връща речника с данните на потребителите.

- **updateKm** ***(Фиг. 3.26.)***

```
func updateKm(competition_id: String, km: Int, user_email: String)
async throws {
        let competitionsStatsCollection =
firestore.collection("competitions_stats")
        let query =
competitionsStatsCollection.whereField("competition_id", isEqualTo:
competition_id).whereField("user_email", isEqualTo: user_email)
        let querySnapshot = try await query.getDocuments()
        for document in querySnapshot.documents {
            try await document.reference.updateData(["km": km])
        }
}
```

*Фиг. 3.26.: Функция „updateKm”.*

Тази функция извлича данни от колекцията "competitions_stats" във Firestore, като използва условията „competition_id“ и „user_email“ за да намери конкретния документ, който трябва да се обнови. След това функцията обхожда всеки документ, който отговаря на условията и обновява стойността на полето „min“ с предоставената стойност. Обновяването става с помощта на метода „updateData“ на референцията на документа.

- **updateMin** ***(Фиг. 3.27.)***

```
func updateMin(competition_id: String, min: Int, user_email:
String) async throws {
        let competitionsStatsCollection =
firestore.collection("competitions_stats")
        let query =
competitionsStatsCollection.whereField("competition_id", isEqualTo:
competition_id).whereField("user_email", isEqualTo: user_email)
        let querySnapshot = try await query.getDocuments()
        for document in querySnapshot.documents {
            try await document.reference.updateData(["min": min])
        }
}
```

*Фиг. 3.27.: Функция „updateMin”.*

Тази функция работи по същия начин като updateKm *(Фиг. 3.26.)*, с разликата, че се обновява полето „min”. Обновяването става с помощта на метода „updateData“ на референцията на документа.

- **getActiveCompetitionsByEmail** ***(Фиг. 3.28.)***

```
func getActiveCompetitionsByEmail(email: String) async throws ->
[String: Any] {
        var competitions = [String: Any]()
        let competitionsCollection =
firestore.collection("competitions")
        let query = competitionsCollection.whereField("isActive",
isEqualTo: true).whereField("email", isEqualTo: email)
        let querySnapshot = try await query.getDocuments()
        let competitionsData = querySnapshot.documents
        competitionsData.forEach { document in
            let competition = document.data()
            competitions[document.documentID] = competition
        }
        return competitions
}
```

*Фиг. 3.28.: Функция „getActiveCompetitionsByEmail”.*

Тази функция извлича данни от колекцията „competitions“ във Firestore, като използва условията „isActive“ и „email“, за да намери всички активни състезания, в които участва потребителят с подадения имейл. След това функцията обхожда всеки документ, отговарящ на условията, и запазва данните му в речник, като ключове са идентификационните му номера, а стойностите са всички данни на документа. Накрая функцията връща речника с данните на всички намерени активни състезания, в които участва потребителят с подадения имейл.

- **getDeactivatedCompetitionsByEmail** ***(Фиг. 3.29.)***

```
func getDeactivatedCompetitionsByEmail(email: String) async throws
-> [String: Any] {
        var competitions = [String: Any]()
        let competitionsCollection =
firestore.collection("competitions")
        let query = competitionsCollection.whereField("isActive",
isEqualTo: false).whereField("email", isEqualTo: email)
        let querySnapshot = try await query.getDocuments()
        let competitionsData = querySnapshot.documents
        competitionsData.forEach { document in
            let competition = document.data()
            competitions[document.documentID] = competition
        }
        return competitions
}
```

*Фиг. 3.29.: Функция „getDeactivatedCompetitionsByEmail”.*

Тази функция работи по същия начин като getActiveCompetitionsByEmail *(Фиг. 3.28.)*, но с разликата, че извлича всички състезания, които не са активни. Накрая функцията връща речника с данните на всички намерени активни състезания, в които участва потребителят с подадения имейл.

- **getCompetitionsByEmail** ***(Фиг. 3.30.)***

```
func getCompetitionsByEmail(email: String) async throws -> [String: Any] {
        var competitions = [String: Any]()
        let competitionsCollection = firestore.collection("competitions")
        let query = competitionsCollection.whereField("email", isEqualTo: email)
        let querySnapshot = try await query.getDocuments()
        let competitionsData = querySnapshot.documents
        competitionsData.forEach { document in
            let competition = document.data()
            competitions[document.documentID] = competition
        }
        return competitions
}
```

*Фиг. 3.30.: Функция „getCompetitionsByEmail".*

Тази функция извлича всички състезания от колекцията „competitions" във Firestore, които са свързани с дадения имейл адрес, като използва условието „email". След това функцията обхожда всеки документ, който отговаря на условието и добавя данните на състезанието в речник с ключ идентификатора на документа и стойността на данните на състезанието. Накрая функцията връща речника с данните на всички състезания, свързани с дадения имейл адрес.

## 3.3. ViewModel-и

ViewModel е основен компонент на архитектурния модел MVVM (Model-View-ViewModel) в Swift проекти. ViewModel-ът е отговорен за свързването на модела (Model) с изгледа (View) и позволява на потребителя да взаимодейства с данните в модела чрез изгледа.

ViewModel-ът съдържа логиката на приложението, която управлява данните, подавани на изгледа, като изпълнява операции като зареждане на данни, трансформиране на данни и обработка на входни данни от потребителя. Той също така управлява валидацията на данните и комуникацията с модела и други.

ViewModel-ът се свързва с изгледа чрез двустранна връзка. Това означава, че ViewModel получава данни от модела и ги обработва, след което ги предава на изгледа за показване. Също така, ViewModel-ът може да получава входни данни от потребителя, обработва ги и ги предава на модела за обработка.

Използването на ViewModel-ът в MVVM архитектурата *(Фиг. 2.6.)* помага за разделянето на логиката на приложението и подобрява преносимостта и тестването на приложението. Той позволява на разработчиците да разделят графичния интерфейс и логиката на приложението, като по този начин могат да се съсредоточат върху всеки аспект отделно.

Всеки един ViewModel в проекта съдържа следната информация:

- скрит атрибут „network" *(Фиг. 3.31.)* от тип class Network *(Фиг. 3.2.)*, който служи за достъп до функциите, които извличат информация от Firebase.

```
private let network: Network
```

*Фиг. 3.31.: Константен атрибут „network".*

- скрит атрибут "coordinator" *(Фиг. 3.32.)* от тип class Coordinator *(Фиг. 3.1.)*, който служи за смяна на изгледи. Също така той е „unowned", което означава, че се предполага, че референцията ще бъде винаги валидна и няма нужда от грижа за управлението на жизнения цикъл на обекта, към който се отнася.

```
private unowned let coordinator: Coordinator
```

*Фиг. 3.32.: Константен атрибут „coordinator".*

- метод „route" *(Фиг. 3.33.)*, който служи за отместването от един изглед към друг. Извиква функцията „route" от класа Coordinator *(Фиг. 3.1.)*.

```
func route(to newTab: Coordinator.Tab) {
     coordinator.route(to: newTab)
}
```

*Фиг. 3.33.: Метод „route".*

- инициализация на класа *(Фиг. 3.34.)*. Приема два аргумента – „network" *(Фиг. 3.31.)* и "coordinator" *(Фиг. 3.32.)*.

```
init(network: Network, coordinator: Coordinator) {
      self.network = network
      self.coordinator = coordinator
}
```

*Фиг. 3.34.: Инициализация на класа.*

Също така всеки един ViewModel е клас от тип „ObservableObject", което означава, че този клас може да бъде използван като модел във визуалните елементи на приложението, като те ще бъдат автоматично обновявани, когато се промени състоянието на модела.

## 3.4. View-та

View-то (изгледът) е част от UI-а, която представя на потребителя визуалната информация на приложението. В тази архитектура, View служи за показване на данни, които идват от ViewModel-ите и позволява на потребителя да взаимодейства с приложението.

View-то може да бъде декларирано по различни начини, включително с използване на структура, клас или функция. Всяко View може да има собствено състояние, което е описано с помощта на @State, @Binding или @ObservedObject. В MVVM архитектурата *(Фиг. 2.6.)*, View-то обикновено получава ViewModel като параметър в конструктора си, което му позволява да взаимодейства с ViewModel-а и да получава от него данни за показване и информация за извършване на действия.

Всяко едно View в проекта съдържа следната информация:

- скрит атрибут от тип „@ObservableObject", която е от тип ViewModel *(Фиг. 3.35).*

```
@ObservedObject private var viewModel: ViewModel
```

*Фиг. 3.35.: Атрибут „viewModel".*

- инициализация на структурата *(Фиг. 3.36)*. Приема един аргумент – ViewModel.

```
init(viewModel: ViewModel) {
    self.viewModel = viewModel
}
```

*Фиг. 3.36.: Инициализация.*

- свойство „body” *(Фиг. 3.37.)*. То е единственото свойство, което трябва да бъде реализирано във всяко View в SwiftUI. То връща описание на изгледа, изграден от елементи. Ключовата дума „some” допълва „body”, което означава, че компилаторът знае кой тип View да върне. Реално няма различни типове View, но се различават по условията или конкретната реализация на дадени компоненти.

```
var body: some View {
    // Code here...
}
```

*Фиг. 3.37.: Свойство „body”.*

# ЧЕТВЪРТА ГЛАВА

# Наръчник на потребителя

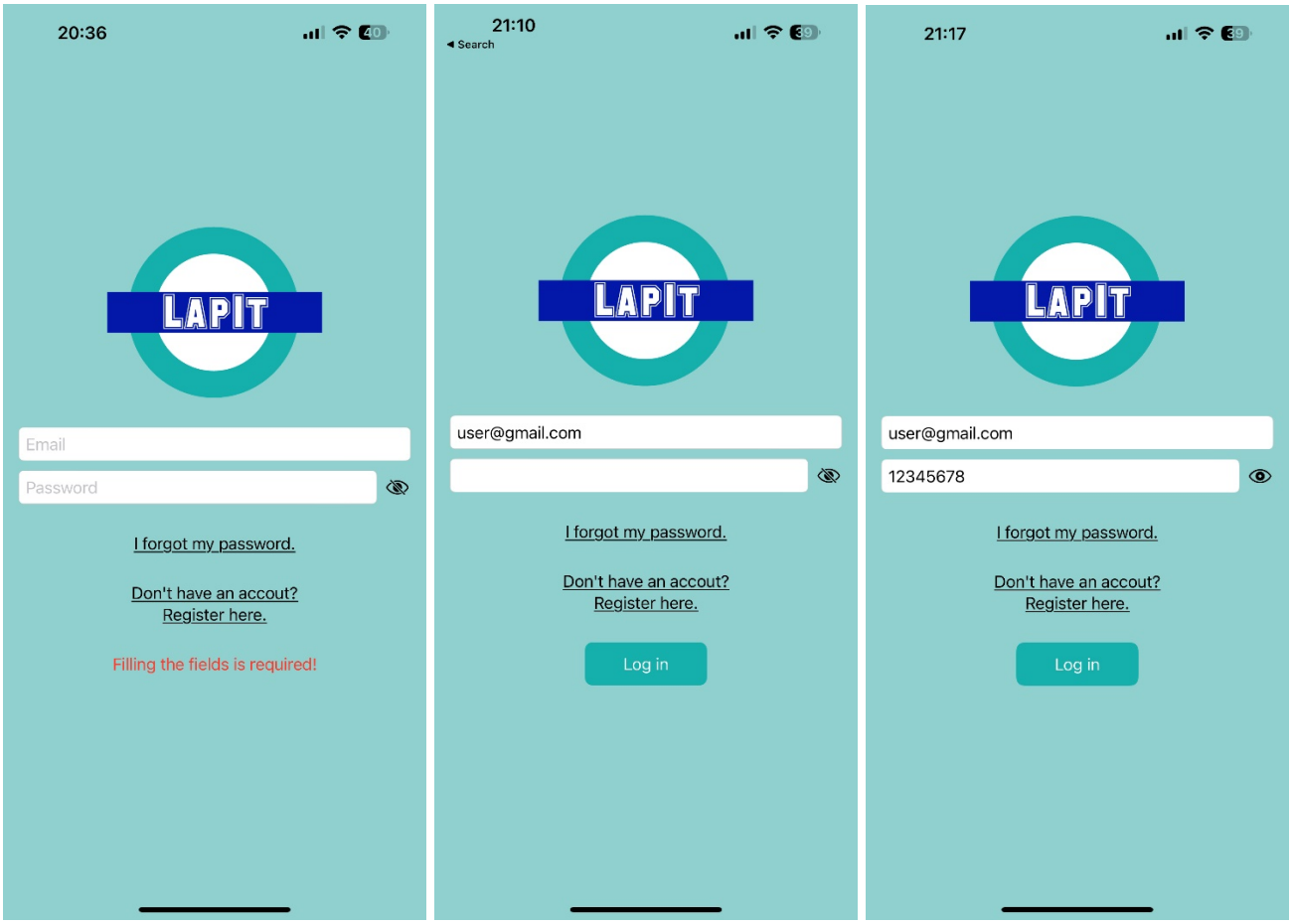
## 4.1. Инсталация

За да се изтегли iOS мобилното приложение „LapIt”, трябва да се следват следните стъпки:

- Да се клонира проекта от публичното хранилище: https://github.com/YordanMarkov/LapIt
- Да се отвори папката „LapIt” в програмата Xcode.
- Да се свърже телефон, iPhone, който поддържа минимум iOS 16 операционна система.
- От телефона да се пусне Developer mode от настройките.
- Да се build-не приложението и да се стартира след като се избере като устройството за симулатор на проекта.
- От телефона да се удостовери приложението от настройките за поверителност.

## 4.2. Въведение през екраните

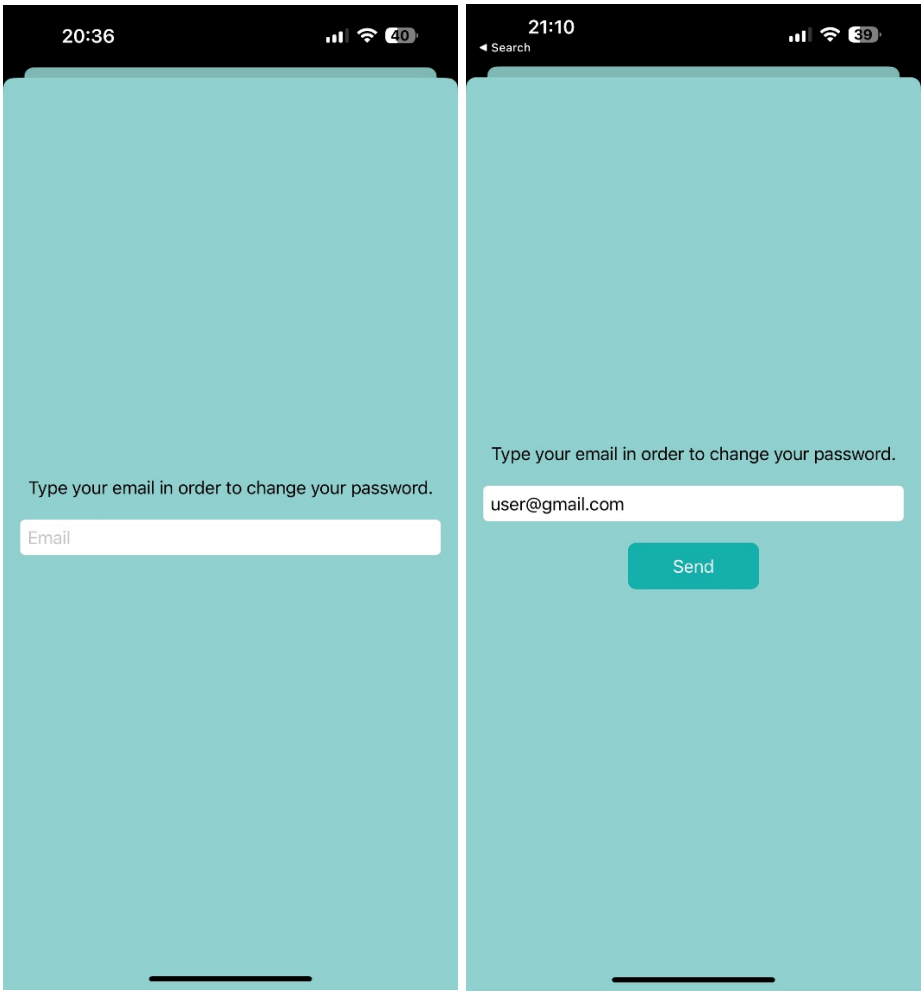
Първият екран, който се показва, когато потребителят стартира приложението, е за вход в профила *(Фиг. 4.1.)*.




*Фиг. 4.1.: Екран за вход в профила.*

*Забележка: Полето „Password" е защитено. При екранни снимки информацията в него не се показва, докато не се натисне бутона за разкриване.*

Този екран предоставя и възможността на потребителя да достигне до екраните за смяна на паролата си *(Фиг. 4.2.)*, когато я забрави, и до екрана за регистрация на акаунт *(Фиг. 4.3.)*.



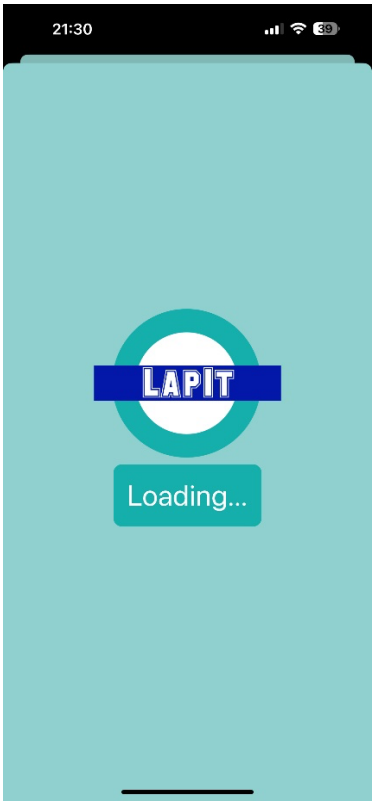

*Фиг. 4.2.: Екран за забравена парола.*


*Фиг. 4.3.: Екран за регистрация.*

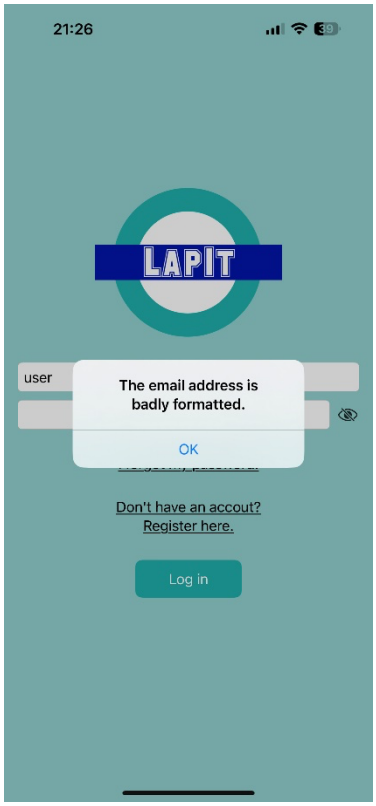
*Забележка: Полетата „Password” и „Repeat Password” са защитени. При екранни снимки информацията в тях не се показва, докато не се натисне бутона за разкриване.*

След въвеждането на имейл за смяна на паролата излиза бутона „Send”. При натискането му се появява зареждащ екран *(Фиг. 4.4.)*. При неуспешно

изпращане поради некоректен имейл, се появява грешка *(Фиг. 4.5.)*. При успешно изпращане на имейл за смяна на парола *(Фиг. 4.6.)* се връща на екрана за вход в профила *(Фиг. 4.1.)*.




*Фиг. 4.4.: Зареждащ екран.*




*Фиг. 4.5.: Грешка за невалиден имейл адрес.*


Hello,

Follow this link to reset your project-1035047860086 password for your user@gmail.com account.

https://lapit-7a6d9.firebaseapp.com/__/auth/action?mode=resetPassword&oobCode=BPoGKvFMs6aPvklvcgLxTU8AEB27Ic_61dhOmBY2zhQAAAGGW7T5qw&apiKey=AIzaSyB01HcnnO1iRZqZthVODQSt71FsRh6Z0Co&lang=en

If you didn't ask to reset your password, you can ignore this email.

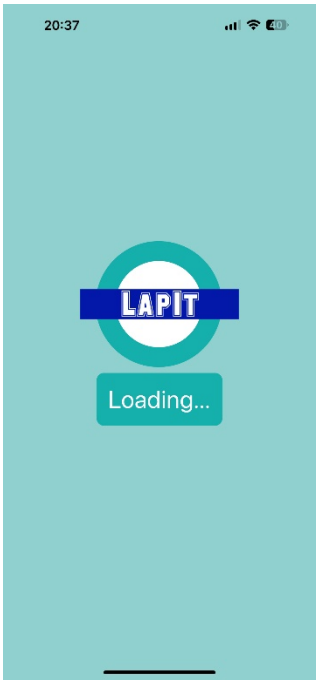
Thanks,

Your project-1035047860086 team

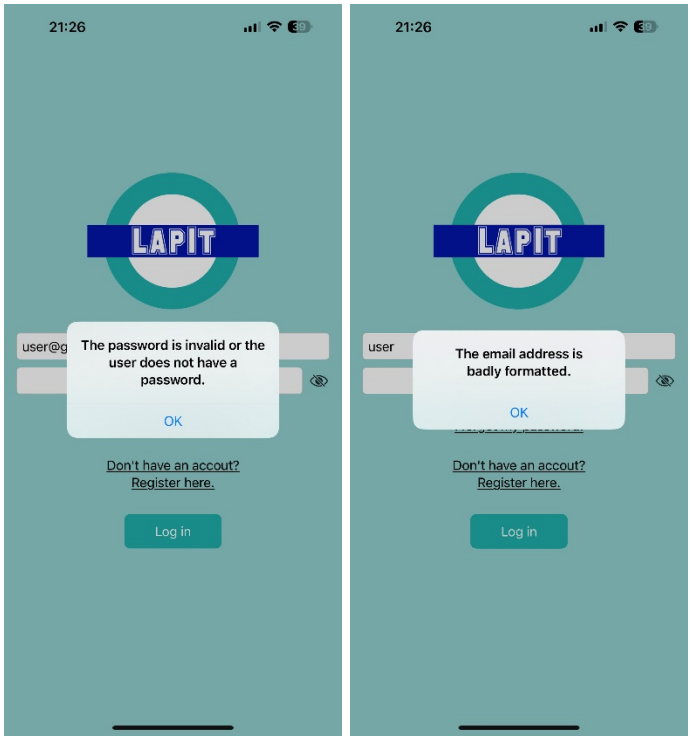
*Фиг. 4.6.: Имейл за смяна на паролата.*

След успешна регистрация, потребителят отново се връща на екрана за вход в профила *(Фиг. 4.1.)*.

След натискане на бутона „Log In” се появява зареждащ екран *(Фиг. 4.7.)* и след това е възможно да се появят грешки за грешна парола или невалиден имейл адрес *(Фиг. 4.8.)*.




*Фиг. 4.7.: Зареждащ екран.*

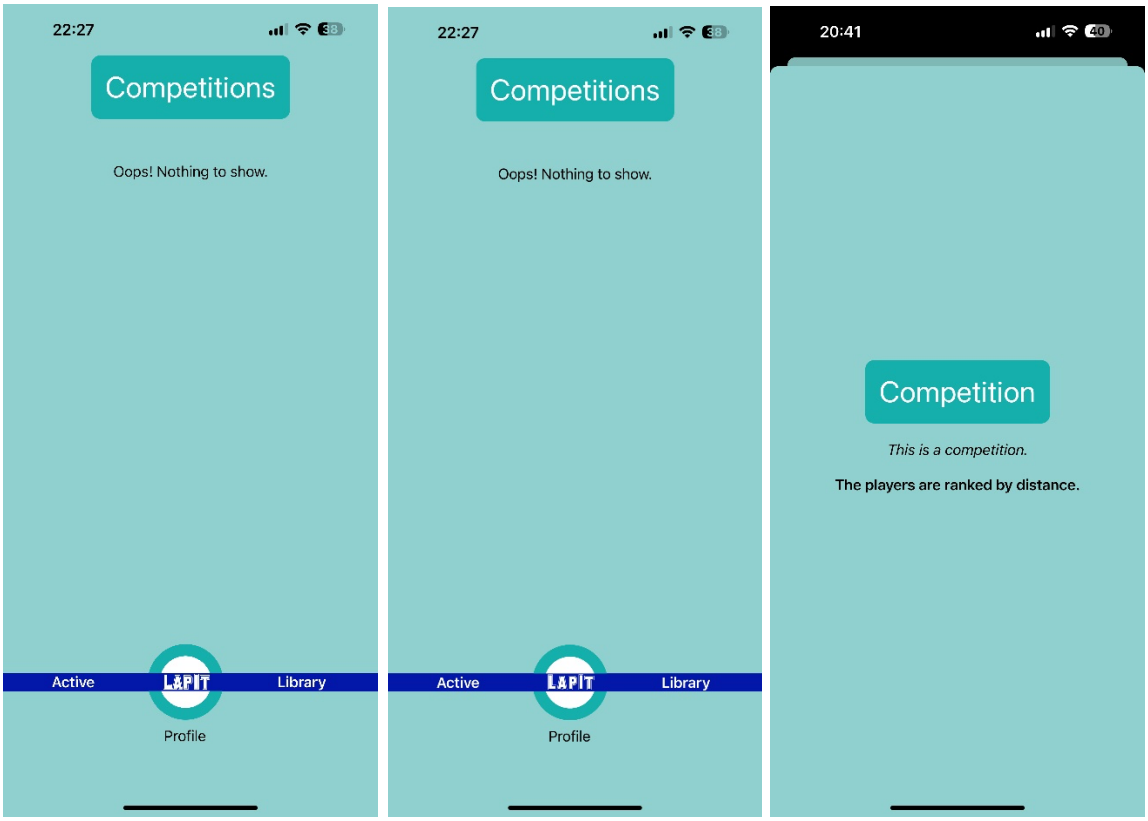



*Фиг. 4.8.: Грешки за неправилна парола и недобре форматиран имейл адрес.*

След успешно влизане в акаунта, потребителят се препраща в началното меню на приложението. В зависимост от това дали е организатор или не – екраните се различават.

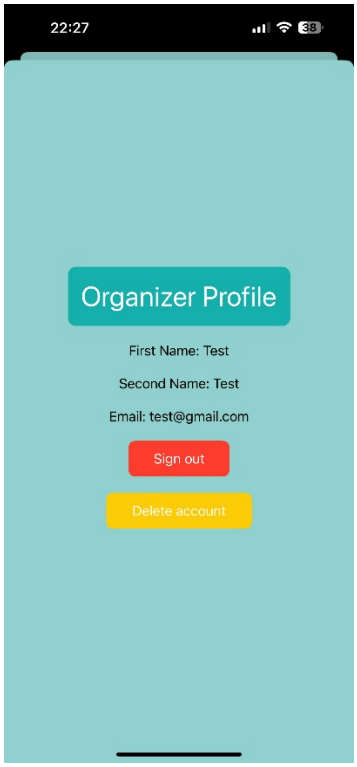
Когато се влезе в организаторски профил, потребителят достига до неговия начален екран *(Фиг. 4.9.)*. В зависимост от това дали има състезания, отворени за

записване (активни), началният екран може да бъде празен. Ако има състезания – те могат да се отварят като екрани и да се чете информацията за тях.

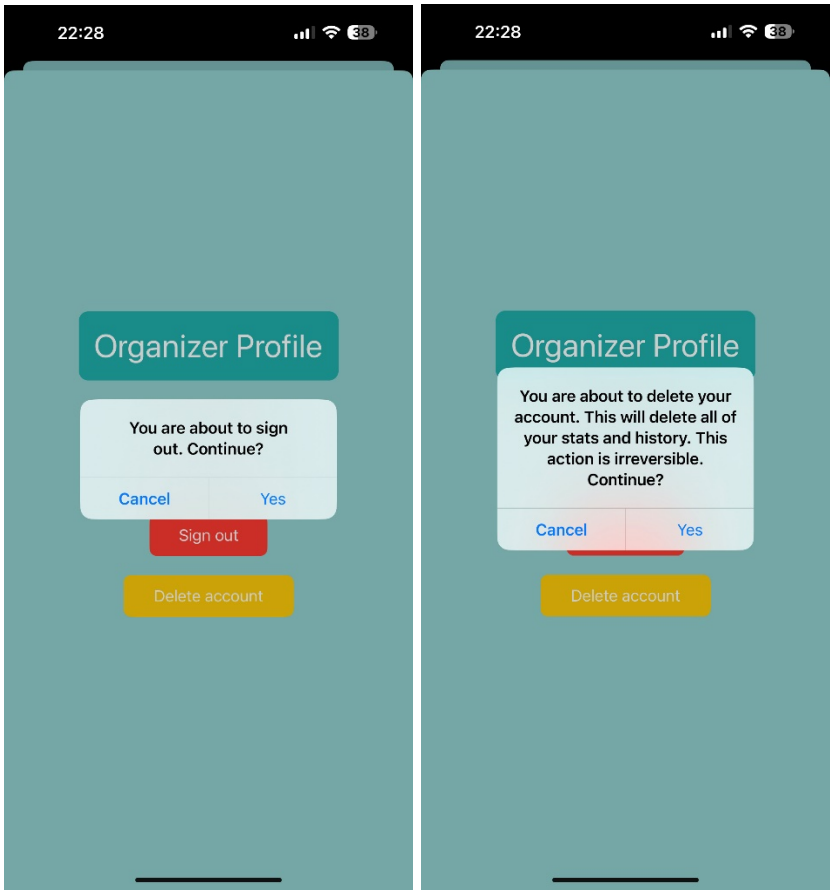



*Фиг. 4.9.: Начален екран - организаторски акаунт.*

Най-отдолу на екрана се намира лента, върху която се намира бутон „Profile”. Когато се натисне се отваря екран с данните за профила *(Фиг. 4.10.)*. Там се съдържат и два бутона – „Sign out“ и „Delete account”. След натискането на бутоните излизат предупреждения за излизане от акаунта и за изтриването му *(Фиг. 4.11.)*. След изпълняването им – потребителят се връща при входа.




*Фиг. 4.10.: Екран за профилна информация – организаторски акаунт.*

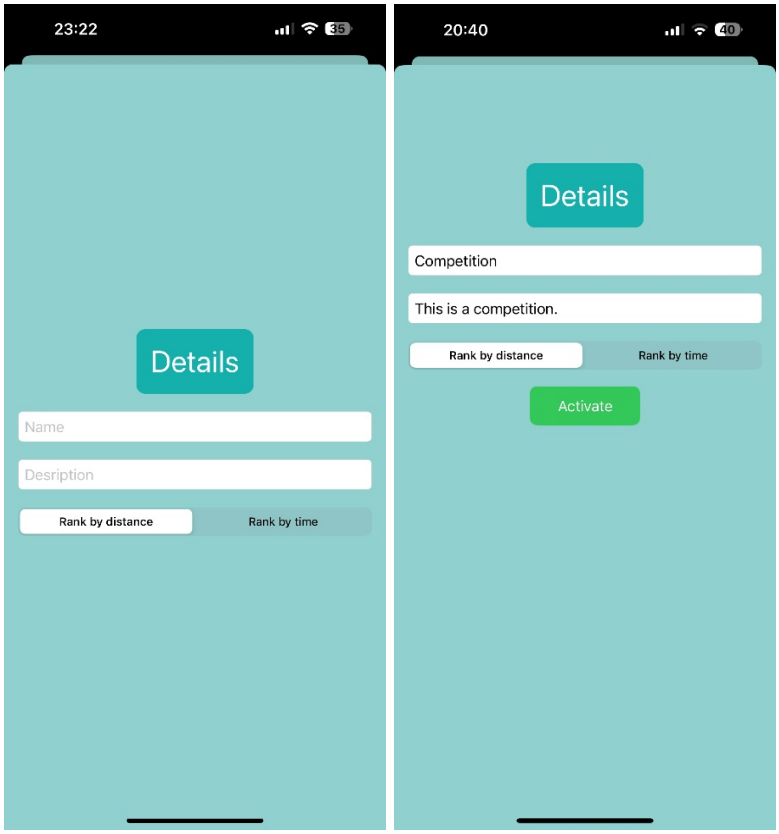



*Фиг. 4.11.: Предупреждения за излизане от акаунт и изтриването му – организаторски акаунт.*

На началния екран *(Фиг. 4.9.)* се намира и бутон „Library” долу вдясно на екрана. Той води до екрана за всички запазени състезания, създадени от потребителя *(Фиг. 4.12.)*. Ако има създадени състезания, те могат да се натискат и да излезе предупреждение за преизползване. След съгласие за преизползване, потребителят вижда екран с попълнени полета за създаване на състезание *(Фиг. 4.14., но с попълнена информация)*. Отстрани до самото състезание в списъка се намира и бутон за изтриване на състезанието. При натискане излиза отново предупреждение *(Фиг. 4.13.)*. В долния ляв ъгъл се намира бутон, който пренасочва към началния екран, а в десния – бутон за създаване на състезание „Create” *(Фиг. 4.14)*. След натискането на бутона „Activate” състезанието се създава, добавя се към запазените и се показва като активно за всеки.


*Фиг. 4.12.: Екран за всички запазени състезания, създадени от потребителя.*


*Фиг. 4.13.: Предупреждения за преизползване и изтриване на състезание.*



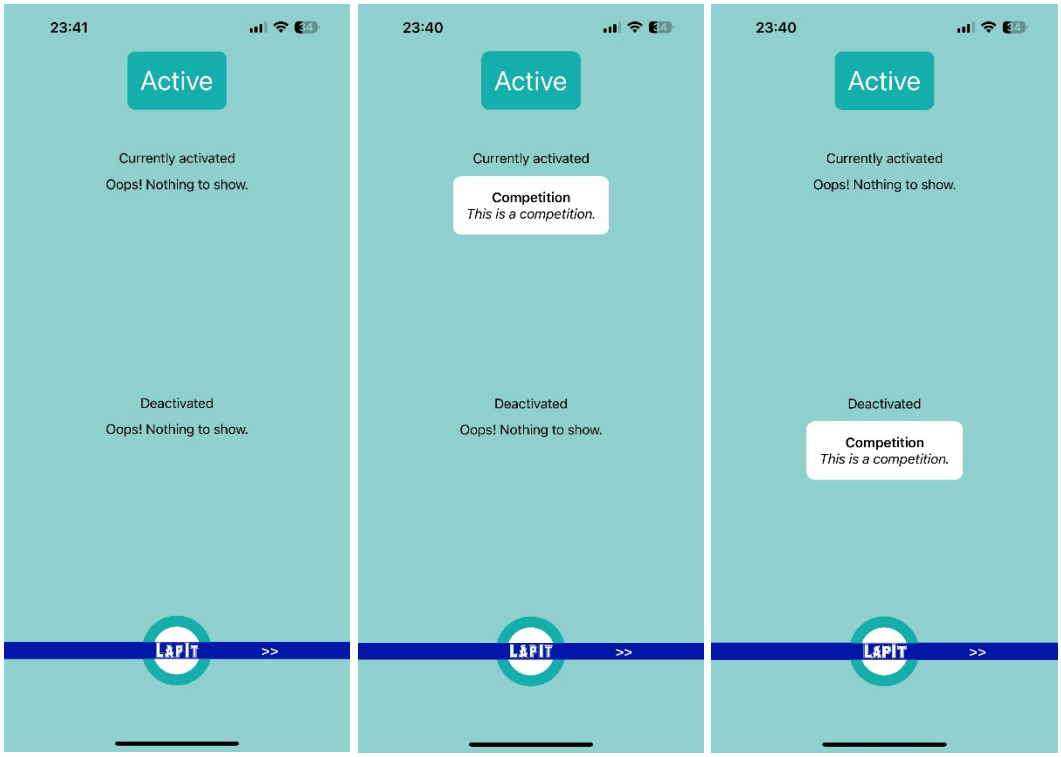

*Фиг. 4.14.: Екран за създаване и преизползване на състезание.*

На началния екран в долния му ляв ъгъл се намира бутон „Active”. След натискането му, потребителят се пренасочва към екран, който показва всичките

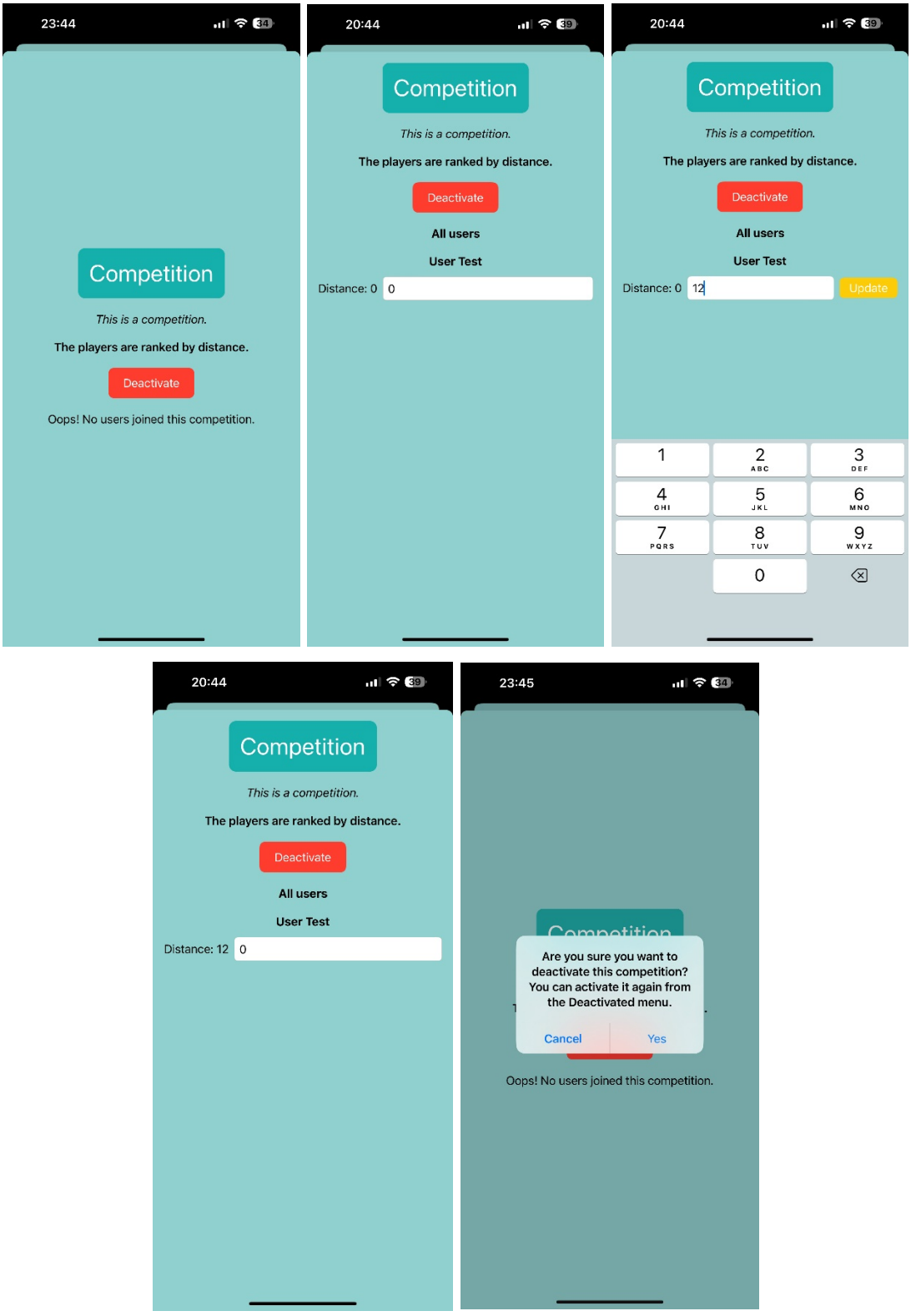
му активни в момента състезания и всички, които е дезактивирал *(Фиг. 4.15.)*. При налични активни или дезактивирани състезания – те могат да се отварят.

Всяко едно активно състезание се отваря в екран *(Фиг. 4.16.)*, който показва информацията за самото състезание и неговите участници (ако има такива, които са се включили). На всеки участник може да се задава време или дистанция (в зависимост от вида на състезание). След натискането на бутон „Update", данните за определения участник се актуализират. При натискането на бутон „Deactivate" – излиза предупреждение и при съгласие състезанието се затваря за записване и участниците повече няма да могат да се включват в него, но ще могат да му наблюдават класацията.

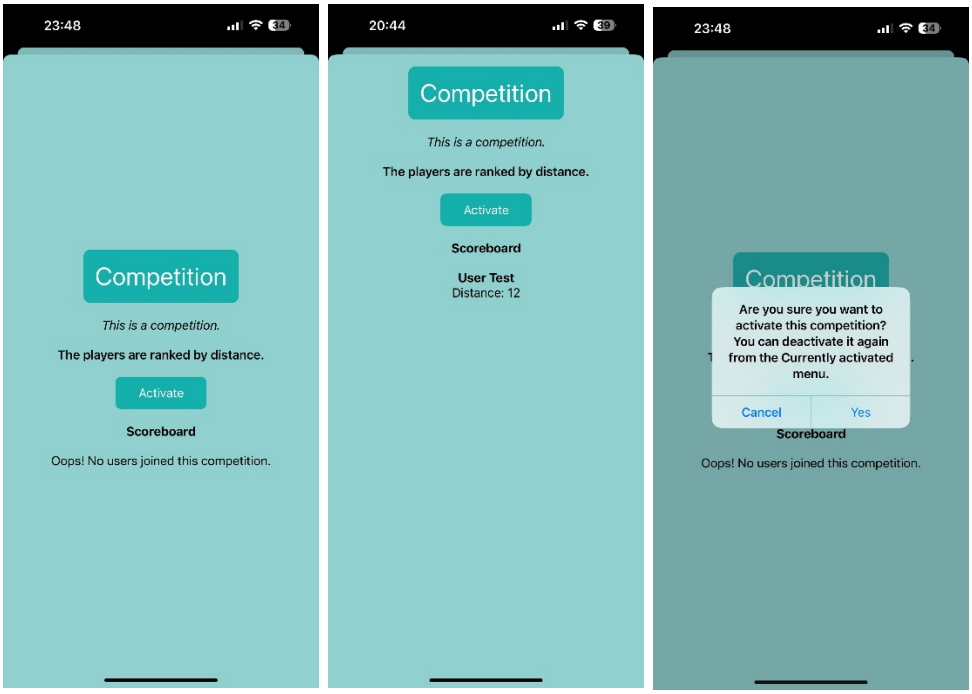
Всяко едно дезактивирано състезание се изобразява в екран *(Фиг. 4.17.)*, който съдържа информацията за самото състезание, неговите участници и последния вариант на неговата класация. То може да бъде активирано отново, като се съгласи потребителя с предупреждението, за да се редактират данните отново или да позволи да се записват в него допълнителни участници.




*Фиг. 4.15.: Екран за активни и дезактивирани състезания.*

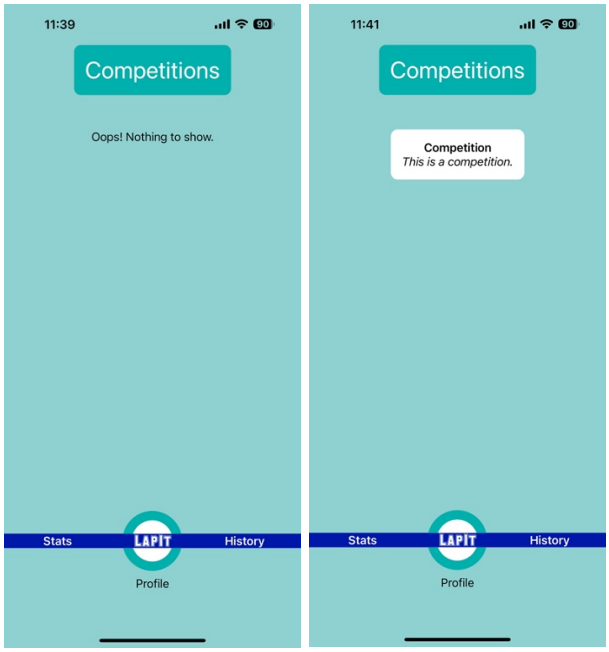



*Фиг. 4.16.: Екран за активно състезание.*

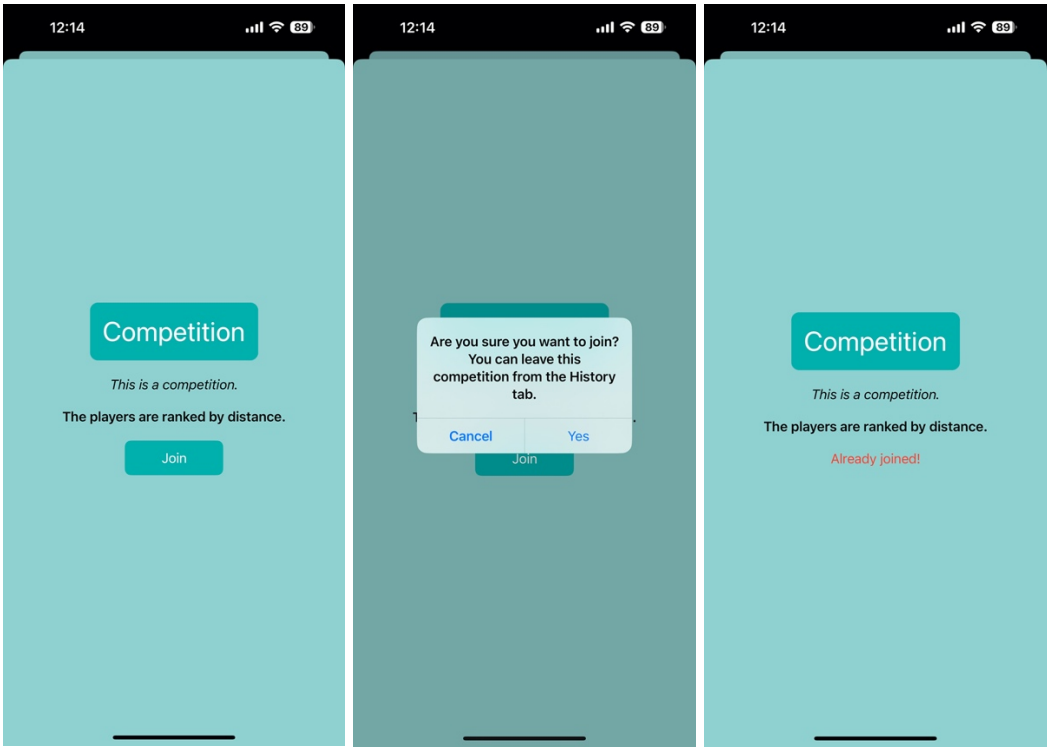



*Фиг. 4.17.: Екран за дезактивирано състезание.*

Когато се влезе в нормален профил за участници, потребителят достига до неговия начален екран *(Фиг. 4.18.)*. В зависимост от това дали има състезания, отворени за записване (активни), началният екран може да бъде празен. Ако има състезания – те могат да се отварят като екрани, да се чете информацията за тях и да се записват участниците в тях *(Фиг. 4.19.)*. Влизането в състезание се осъществява след съгласяване с предупреждение.

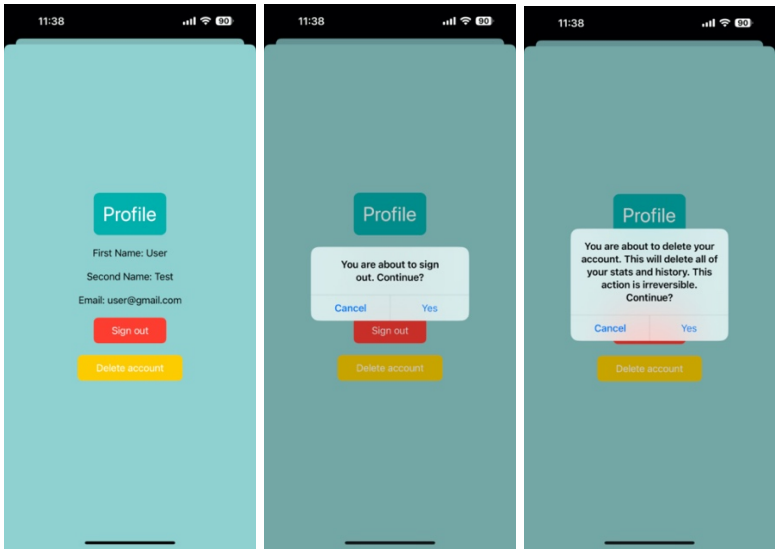



*Фиг. 4.18.: Начален екран – акаунт за участници.*



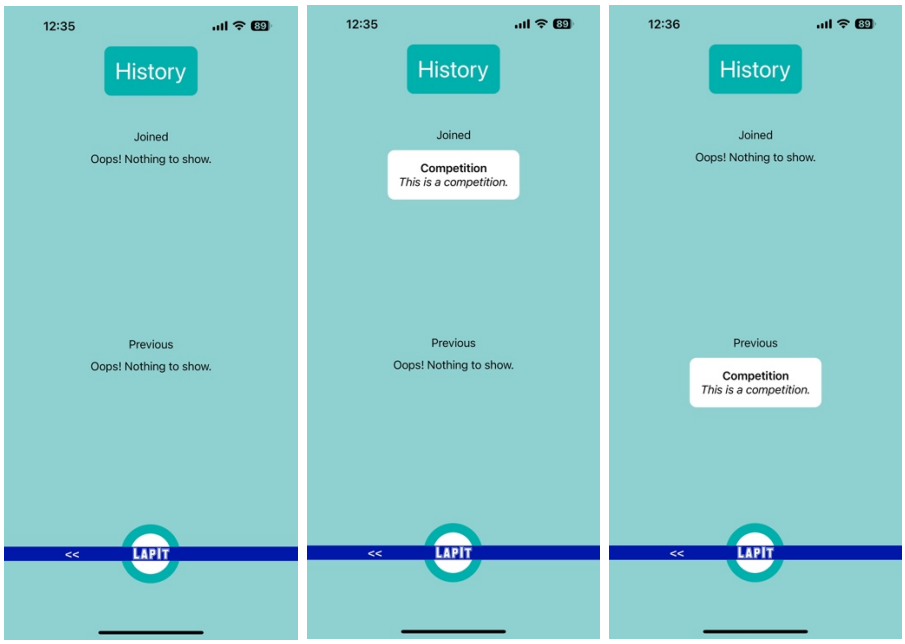

*Фиг. 4.19.: Екран за състезание с опция влизане.*

Най-отдолу на екрана, както и при организаторите, се намира лента, върху която се намира бутон „Profile”. Когато се натисне се отваря екран с данните за профила *(Фиг. 4.20.)*. Там се съдържат, отново, два бутона – „Sign out“ и „Delete account”. След натискането на бутоните излизат предупреждения за излизане от акаунта и за изтриването му. След изпълняването им – потребителят се връща при входа.

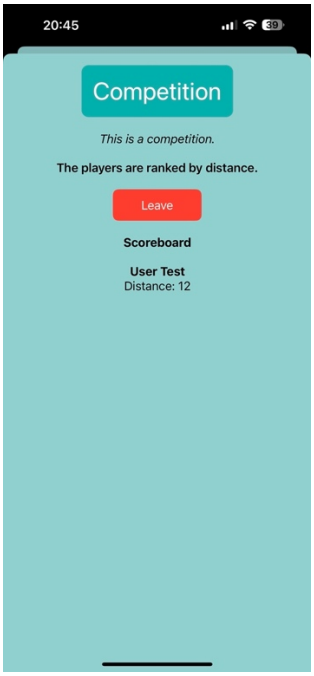



*Фиг. 4.20.: Екран за профилна информация и предупреждения – акаунт за участници.*

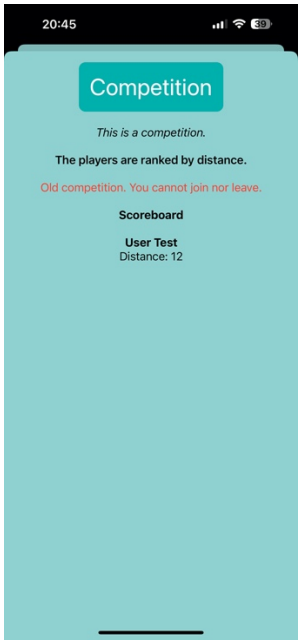
На началния екран има и два пренасочващи бутона. Единият от тях е “History” (История) *(Фиг. 4.21.)*. В него се съдържат всички състезания, в които потребителят е участвал или в момента участва. В списъка „Joined” се намират всички активни състезания в момента. При натискане на състезанието се отваря информация за състезанието, резултатите на участниците и бутон, който предоставя опция за напускане - „Leave” *(Фиг. 4.22.)*. След съгласяване с предупреждението – състезанието се изтрива от списъка. В списъка „Previous” се намират всички състезания, които са неактивни вече, но потребителят е участвал в тях. При натискане на състезанието се отваря информация за състезанието и резултатите на участниците *(Фиг. 4.23.)*.




*Фиг. 4.21.: Екран за история.*

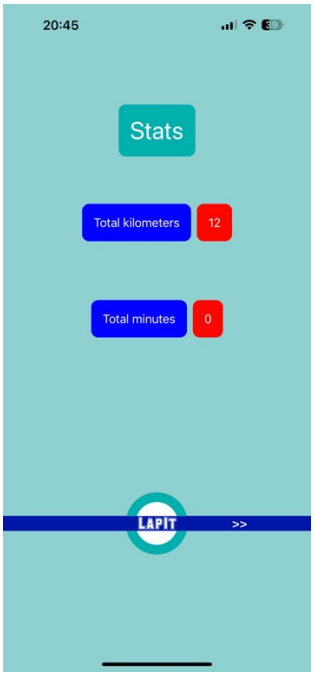



*Фиг. 4.22.: Екран за активно състезание с опция за напускане.*




*Фиг. 4.23.: Екран за неактивно състезание.*

На главния екран (Фиг. 4.18.) се намира бутон „Stats”, който води до последния екран от профила на участниците – сбор от всичките резултати на потребителя. *(Фиг. 4.24.)* .




*Фиг. 4.24: Екран за сбор от всичките резултати на потребителя.*

# ЗАКЛЮЧЕНИЕ

Всички функционални изисквания, нужни, за да се разработи проекта, са изпълнени. Тези изисквания бяха:

- Да се определи вида на акаунта – потребителски или организаторски. ✓
- Включване в състезание. ✓
- Преглед на история на предишни състезания. ✓
- Следене на резултати на състезания. ✓
- Създаване на състезания. ✓
- Задаване на вида на състезанието – по време или по дистанция. ✓
- Въвеждане на данни за участниците. ✓
- Запазване на състезанието като модел за преизползване. ✓

В бъдеще приложението може да бъде разширено и подобрено. Някои от идеите са:

- Като всеки един софтуер, приложението може да бъде оптимизирано.
- Откъм визуализация – UI-ът на приложението може да се подобри.
- Добавяне на опция за реално и автоматично следене на дистанция и време.
- Добавяне на опция „Добави приятел“.
- Връзка с организаторите и участниците – чат в реално време.
- Добавяне на карта, която показва маршрута на състезанието в реално време.
- Добавяне на възможността няколко организатори да менажират едно състезание през мобилните приложения с различни профили.

# ИЗПОЛЗВАНИ ТЕРМИНИ

1. UI – User Interface – Потребителски Интерфейс
2. Dart – Програмен език
2. View – Изглед
3. ViewModel – Модел на изгледа
4. Profile – Профил
5. History – История
6. Library – Библиотека
7. Organizer – Организатор
8. Create – Създайте
9. Stats – Статистика
10. Active – Активно
11. Activate – Активирайте
12. Previous – Предишни
13. Deactivate – Деактивирайте
14. Deactivated – Деактивиран
15. Total kilometers – Обща сума километри
16. Total minutes – Обща сума минути
17. Leave – Напуснете
18. Join – Влезте
19. Update – Актуализирайте
20. Build – Постройте
21. Sign in – Впишете се
22. Sign out – Отпишете се
23. Delete account – Изтрийте акаунт

# ИЗПОЛЗВАНА ЛИТЕРАТУРА


1. Mobile vs. Desktop: https://research.com/software/mobile-vs-desktop-usage
2. React Native: https://reactnative.dev/
3. Xamarin: https://dotnet.microsoft.com/en-us/apps/xamarin
4. Flutter: https://flutter.dev/
5. Swift: https://developer.apple.com/swift/
6. Kotlin: https://kotlinlang.org/
7. Objective-C: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html
8. Java: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html
9. Amazon Web Services: https://aws.amazon.com/
10. Microsoft Azure: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/
11. Firebase: https://firebase.google.com/
12. IBM Cloud: https://www.ibm.com/cloud
13. Wings for Life World Run: https://www.wingsforlifeworldrun.com/en
14. BMW BERLIN-MARATHON: https://www.bmw-berlin-marathon.com/en/
15. TCS Amsterdam Marathon 2022: https://www.tcsamsterdammarathon.eu/
16. Athens Marathon and Half: https://athinahalfmarathon.gr/index.php/en/home-en
17. TCS New York City Marathon: https://www.nyrr.org/
18. RunCzech App: https://www.runczech.com/en/for-you/inspiration/aplication
19. UIKit: https://developer.apple.com/documentation/uikit
20. SwiftUI: https://developer.apple.com/xcode/swiftui/
21. Figma: https://www.figma.com/
22. MVVM: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel
23. SwiftUI Documentation: https://developer.apple.com/documentation/swiftui/

24. The Swift Programming language: https://books.apple.com/bg/book/the-swift-programming-language-swift-5-7/id881256329
25. Stanford University Lectures: https://www.youtube.com/watch?v=bqu6BquVi2M&list=PLpGHT1n4-mAsxuRxVPv7kj4-dQYoC3VVu
26. Hide Password: https://stackoverflow.com/questions/63095851/show-hide-password-how-can-i-add-this-feature

# СЪДЪРЖАНИЕ