



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ
СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

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

Тема: Платформа за създаване и участие в различни приключенски
събития

Дипломант:
Ивайла Иванова Панайотова

Дипломен ръководител:
маг. инж. Станислав Милев

СОФИЯ

2024



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 15.11.2023 г.

Утвърждавам:.....

Дата на предаване: 15.02.2024 г.

/проф. д-р инж. П/ Якимов/

ЗАДАНИЕ за дипломна работа

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ
по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

на ученика Ивайла Иванова Панайотова от 12 в клас

1. Тема: Платформа за създаване и участие в различни приключенски събития
2. Изисквания:
 1. Създаване и валидиране на потребителски акаунти
 2. Създаване на събития за пътешествия
 3. Възможност за свързване между различни потребители на платформата, което ще позволи съвместното им участие в събития.
 4. Възможност на търсене на събития по име и описание.
 5. Възможност за създаване на списък с любими събития.
3. Съдържание
 - 3.1 Теоретична част
 - 3.2 Практическа част
 - 3.3 Приложение

Дипломант:.....

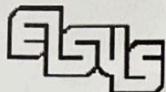
/ Ивайла Панайотова /

Ръководител:.....

/ Станислав Милев /

ВРИД Директор:.....

/ ст. пр. д-р Веселка Христова /



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ
СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

СТАНОВИЩЕ КЪМ ДИПЛОМНА РАБОТА

Тема: Платформа за създаване и участие в различни приключенски събития

Дипломантът се е справил успешно с дипломната работа, покрити са всички условия описани в изискванията към дипломната работа.

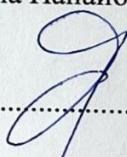
Ивайла е добила добри познания по език за програмиране swift, както и javascript. Също така е научила основите на разработката на мобилни приложения за айос платформа. Научена е архитектура MVVM, която помага за бъдещото развитие на проекта. Придобити са знания по HTTP протокол, както и бази данни. Разработката на собствен бекенд е помогнало за развитието на нови умения, които да помогнат за бъдещето ѝ.

Предлагам за рецензент:

Марин Ранчев от Стам Софт ООД (marin.ranchev@stamsoft.com)

Дипломант: 

/Ивайла Панайотова/

Ръководител: 

/маг. инж. Станислав Милев/

Дата: 20.02.2024

Увод

В днешно време, в свят, изпълнен с технологии, мобилните устройства заемат огромна част от нашето ежедневие. При всяка нужда от информация или услуга, най-често се обръщаме към нашето мобилно устройство. За да са достъпни тези неща, от които имаме нужда се появяват и мобилните приложения. Мобилното приложение е софтуерно приложение, предназначено за работа на мобилни устройства, като смартфони и таблети. Тези приложения се разработват, за да предоставят специфични функционалности, услуги или информация на потребителите. Те предоставят възможност за улеснение на множество действия като пазаруване, учене, общуване, организация, следене на финансите и много други. Както за извършване на обичайните ни действия обаче, различни платформи ни позволяват намирането на нови хобита, интереси и запознанства. Развитието на мобилните технологии продължава да променя начина, по който си взаимодействаме с околната среда и хората. В бъдеще можем да очакваме още иновации в областта на мобилните приложения, които ще продължат да подобряват нашия живот.

Съдържание

Първа глава	4
Технологии и методи за реализация на мобилни приложения	4
1.1. Основни принципи на мобилните приложения	4
1.2. Основни технологии за разработка на мобилни приложения	4
1.2.1. Native	4
1.2.2. Cross-platform	7
1.3. Среди за реализиране на мобилни приложения	9
1.3.1. Xcode	9
Xcode е интегрирана среда за разработка на приложения за iOS и macOS. Тя включва набор от инструменти за разработка, включително редактор на код, компилатор, дебъгер и интерфейсен дизайнер. Xcode също така предлага интеграция със системата за управление на версии Git, автоматично управление на зависимостите и мощни инструменти за профилиране и оптимизация на приложението.	9
1.3.2. Android Studio	9
Android Studio е интегрирана среда за разработка на приложения за Android. Базиран е на IntelliJ IDEA, Java интегрирана среда за разработка на софтуер. За да подпомогне разработката на приложения за операционната система Android, Android Studio използва Gradle базирана система за изграждане, емулатор на Android, кодови шаблони и интеграция с GitHub.	9
1.3.3. IntelliJ IDEA Community Edition	9
IntelliJ IDEA Community Edition е безплатна версия на популярната Java Integrated Development Environment (IDE) на JetBrains. IntelliJ IDEA предоставя значителен набор от инструменти за разработка на софтуер, включващи функционалности като довършване на код и маркиране на грешки, инструменти за рефакторинг и генериране на код, вграден Git, интегрирано отстраняване на грешки и тестване и вграден терминал.	9
1.4. Вече съществуващи решения	10
1.4.1. Explorers Connect	10
1.4.2. Adventure IO	10
1.4.3. Meetup	10
1.4.4. Adrenaline	10
Втора глава	12
Проектиране и структура на мобилното приложение	12
2.1. Функционални изисквания към мобилното приложение	12
2.2. Избор на технология за реализация на приложението	12
2.3. Избор на бекенд технология и обектно релационно картографиране	12
2.4. Избор на облачна технология	13
2.5. Избор на програмна среда	13
2.6. Структура на экраните	14
2.5.2 Структура на базите данни	16
Image	18
2.5.3. Структура на кода	20
Трета глава	22
Програмна реализация на платформа за създаване и участие в различни приключенски събития	22

3.1. Основни компоненти и конфигурация с базата данни PostgreSQL посредством TypeORM	22
3.1.1 Модел на приключение	23
3.1.2 Модел на приключенец	26
3.1.3 Модел на създател	29
3.2 Автентикация и създаване на потребителски профили	31
3.2.1. Автентикация	31
3.2.3 Валидация потребителски профил и вход	40
3.2 Създаване на приключенско събитие	50
3.4 Възможност за търсене на събития по име и описание	55
3.5 Възможност за добавяне на събития към списъка с любими събития	58
3.6 Изобразяване на компонентите	61
3.6.1 Функция за извлечане на данни за приключения от сървъра	61
3.6.2.Функция за извлечане на данни за списъка с любими приключенски събития от сървъра	62
3.6.3.Функция за извлечане на данни за приключенците	64
3.6.4.Функция за извлечане на данни за приключенците, с които текущият потребител е свързан.	65
3.6.5.Функция за извлечане на данни за приключенията в които даден приключенец е участвал.	66
3.7 Навигационна лента	66
3.8. Models	71
3.8.1. Adventurer model	72
3.9. ViewModels	80
3.10.3 AdventureView	84
Четвърта глава	87
Ръководство на потребителя	87
4.1. Инсталация	87
4.2. Въведение през экраните	88
4.2.1. Welcome screen	88
4.2.2. Choosing Role Screen	89
4.2.3. Sign up screen	90
4.2.5. Explore screen	92
4.2.7. Detailed Adventure screen	94
4.2.8. Wishlist screen	95
4.2.9. Meet screen	96
4.2.10. Connect to adventurers screen	97
4.2.11. Profile adventurer screen	98
4.2.12. Connected adventurers screen	99
4.2.13. Add new adventure screen	100
4.2.14. Search creator screen	101

Първа глава

Технологии и методи за реализация на мобилни приложения

1.1. Основни принципи на мобилните приложения

Основните принципи на мобилните приложения са:

- Интуитивен дизайн - Включва създаването на потребителски интерфейс, който е лесен за разбиране и използване без необходимост от дълги обяснения. Визуалните елементи и навигацията са организирани логично, като се отчита потребителската логика.
- Бърза реакция - Приложенията трябва да реагират бързо на потребителските действия, осигурявайки плавно и безпроблемно изживяване. Оптимизацията на кода и използването на подходящи технологии също са от голямо значение.
- Оптимизиране за различни устройства и платформи - Мобилните приложения трябва да бъдат съвместими с различни устройства (телефони и таблети) и операционни системи (iOS, Android и др.).
- Сигурност на данните - Защитата на потребителската информация е от голямо значение. Използване на шифроване, безопасни протоколи за комуникация и правилно управление на достъпа са важни аспекти на сигурността.
- Потребителското преживяване - Създаване на приложения, които предлагат приятни и полезни преживявания за потребителите. Стремеж към постоянно подобряние чрез обратна връзка и актуализации.
- Лесната навигация - Навигационните елементи трябва да бъдат ясни и достъпни, за да улеснят потребителите в ориентирането в приложението.

1.2. Основни технологии за разработка на мобилни приложения

1.2.1. Native

Нативните технологии за разработка на мобилни приложения се фокусират над използването на платформено - специфични езици и инструменти за създаване на приложения. Нативните приложения се оптимизират за конкретните платформи, осигурявайки висока производителност и директен достъп до характеристиките на устройството. Разработката на този тип приложения обаче е скъпа, тъй като е свързана с един тип операционна система, което принуждава компанията, която създава приложението, да прави дублирани версии, които работят на други платформи.

- **Objective-C** е език за програмиране, който се използва главно за разработка на приложения за платформите на Apple, като iOS и macOS. Той дълго време е основният език за разработка на приложения за Apple, преди да бъде заменен постепенно от Swift. Като обектно-ориентиран език, Objective-C използва обекти и класове за организация на кода. Позволява динамично свързване на методи по време на изпълнение, което предоставя гъвкавост и динамичност. Може да се използва съвместно с езика C, като предоставя възможност за интегриране със съществуващ код написан на C. Мнозина преустановяват да използват Objective-C в полза на Swift, поради по-модерните и удобни за програмиране възможности, които Swift предлага. Въпреки това, Objective-C все още се поддържа и използва в стари проекти или при работа със съществуващ код.
- **Swift** е програмен език, създаден от Apple за разработка на софтуер. Въведен е през 2014 г. и бързо става основният език за разработка на приложения за iOS, macOS, watchOS и tvOS. Swift е проектиран да бъде сигурен, бърз и лесен за употреба, предлагайки модерни функционалности и синтаксис. Swift е проектиран с фокус върху предотвратяването на грешки по време на компилация и изпълнение, което подобрява безопасността на приложението. Предлага висока производителност и оптимизации, като същевременно запазва лесната четливост на кода. Има изчистен и изразителен синтаксис, който улеснява

четенето и писането на код. Може да се използва съвместно със съществуващ код написан на Objective-C, което прави постепенното преминаване към Swift по-лесно. Swift продължава да се развива, като включва нови функционалности и подобрения през различни версии.

- **Java** е програмен език от високо ниво. В областта на разработката на мобилни технологии, той често се използва за създаване на Android приложения. С появата на платформата Android, Java става основният език за програмиране за тази мобилна операционна система. Java е официалният език за разработка на Android приложения. Android SDK (Software Development Kit) предоставя богата среда за разработка, където Java кодът се компилира до Android байткод. Езикът разполага с обширна общност от разработчици и богата система от библиотеки, които могат да се използват за ускоряване на процеса на разработка. Java е известен с платформената си независимост, което означава, че един и същ код може да се използва на различни операционни системи. Той е обектно-ориентиран, което спомага за структурирането на кода и улеснява преизползването на компоненти. Предоставя вградена поддръжка за многонишковост, което е полезно при разработката на многозадачни и многонишкови приложения, особено в средата на мобилните устройства.
- **Kotlin** е съвременен програмен език, разработен от компанията JetBrains. Този език е проектиран с фокус върху простота, изразителност и безопасност на типовете. Kotlin се компилира до Java байткод, което означава, че може да се използва в съществуващи Java проекти и е съвместим с Java библиотеки. Той предоставя чист и компактен синтаксис, който улеснява разработката и поддържането на кода. Интегрира модерни техники за безопасност на типовете, което помага за предотвратяването на често срещани грешки по време на компилация.

Kotlin е официален език за разработка на Android приложения, като предлага предимства пред Java, като по-кратко и по-четимо кодиране. Той е станал популярен избор сред програмистите, особено в областта на мобилната и уеб разработка, благодарение на своите съвременни възможности и лесната интеграция със съществуващ код.

1.2.2. Cross-platform

Cross-platform технологиите за разработка на мобилни приложения позволяват създаването на приложения, които могат да работят на различни мобилни платформи, като се използва общ кодова база. Те предоставят възможността за разработка на приложение с общ код, който може да се използва на различни операционни системи. По този начин се реализира създаването на приложения, които могат да бъдат изпълнявани на iOS и Android без значителни промени в кода.

Предоставят възможност за интеграция с нативни функционалности чрез специфични API. Въпреки че предлагат интеграция с нативни функционалности, някои cross-platform решения може да имат ограничения при достъпа.

Използването на един и същ код за различни платформи може да доведе до компромиси в дизайна и изживяването на потребителите, което не винаги е желателно.

- **Flutter** е framework отворен код разработен от Google, предназначен за създаване на крос-платформени мобилни приложения. Той позволява на разработчиците да изграждат приложения с еднократно написан код, който може да бъде използван както за iOS, така и за Android. Важен аспект на Flutter е използването на собствения си програмен език Dart. Flutter предоставя богати и гъвкави възможности за дизайн, позволявайки създаването на сложни и интересни потребителски интерфейси. Специфичната характеристика на "горещо презареждане" позволява на разработчиците да визуализират промените в реално време, което ускорява процеса на разработка. Framework-ът позволява лесно

взаимодействие с нативни функционалности, което улеснява интеграцията с платформено-специфични API. Всичко във Flutter е widget, включително структурни елементи и стилове, което улеснява създаването на компоненти и преизползването им.

- **React Native** - React Native е JavaScript framework за писане на приложения за iOS и Android. Базиран е на React, JavaScript библиотеката на Facebook за изграждане на потребителски интерфейси. Приложенията написани на React Native са с една кодова база, което улеснява едновременното разработване на приложения както за Android, така и за iOS
- **Sencha** - Sencha е софтуерна компания, специализирана в разработването на платформи за изграждане на уеб и мобилни приложения. Водещият продукт на компанията е Sencha Ext JS, JavaScript framework за изграждане на кросплатформени уеб приложения с интензивно използване на данни. Framework-ът е предназначен да помогне на разработчиците да създават високопроизводителни и съвместими с различни браузъри приложения. Той предоставя широк набор от вградени компоненти, като мрежи, диаграми и формуляри, както и мощен пакет от данни за работа с данни от различни източници.
- **Xamarin** - Xamarin е платформа за разработка на приложения, която позволява на програмистите да създават собствени мобилни приложения за iOS, Android и Windows, използвайки C# и .NET framework. Xamarin използва една кодова база, която може да се споделя между множество платформи, което позволява на програмистите да пишат един код и той да работи на множество платформи.

1.3. Среди за реализиране на мобилни приложения

1.3.1. Xcode

Xcode е интегрирана среда за разработка на приложения за iOS и macOS. Тя включва набор от инструменти за разработка, включително редактор на код, компилатор, дебъгер и интерфейсен дизайнер. Xcode също така предлага интеграция със системата за управление на версии Git, автоматично управление на зависимостите и мощни инструменти за профилиране и оптимизация на приложенията.

1.3.2. Android Studio

Android Studio е интегрирана среда за разработка на приложения за Android. Базиран е на IntelliJ IDEA, Java интегрирана среда за разработка на софтуер. За да подпомогне разработката на приложения за операционната система Android, Android Studio използва Gradle базирана система за изграждане, емулятор на Android, кодови шаблони и интеграция с GitHub.

1.3.3. IntelliJ IDEA Community Edition

IntelliJ IDEA Community Edition е бесплатна версия на популярната Java Integrated Development Environment (IDE) на JetBrains. IntelliJ IDEA предоставя значителен набор от инструменти за разработка на софтуер, включващи функционалности като довършване на код и маркиране на грешки, инструменти за рефакторинг и генериране на код, вграден Git, интегрирано отстраняване на грешки и тестване и вграден терминал.

1.4. Вече съществуващи решения

1.4.1. Explorers Connect

Explorers Connect е неправителствена организация, основана от изследователката Белинда Кърк. Целта е да помогне на хората да живеят по-приключенски чрез платформа за отбори, обучение и пионерска серия събития, изследваща връзката между приключението и благосъстоянието. Тя известява членовете на своята общност от над 30 000 человека чрез бюлетин, пълен със съвети, вдъхновение и уникални възможности.

1.4.2. Adventure IO

Adventure IO е платформа, която предлага най-високо оценени преживявания, водени от професионални спортисти, бранд амбасадори и местни експерти. Те осигуряват разнообразни дейности както в близост, така и по време на пътуване, като съчетават възможността за приключение с участието на опитни и вдъхновяващи ръководители.

1.4.3. Meetup

Meetup е платформа, където хора с разнообразни интереси се срещат и споделят общи страсти. Независимо дали са любители на планински преходи, книги, мрежови събрания или споделяне на умения, в Meetup има нещо за всекиго. Платформата обединява и помага на хората да се свързват.

1.4.4. Adrenaline

Adrenaline е платформа, където можем да открием и закупим ваучери за различни екстремни преживявания. Независимо дали търсим вълнуващо

изживяване за себе си или специален незабравим подарък за любим човек, Adrenaline предлага множество възможности за екстремни приключения.

В заключение платформата за създаване и участие в различни приключенски събития комбинира голяма част от гореописаните платформи и дава на потребителите си възможност да откриват или създават нови събития, да се свързват с хора, притежаващи подобни интереси и съвместното им участие в нови приключения.

Втора глава

Проектиране и структура на мобилното приложение

2.1. Функционални изисквания към мобилното приложение

2.2. Избор на технология за реализация на приложението

Swift UI е иновативна технология, разработена от Apple, която предоставя мощни инструменти за създаване на модерни и красими приложения за iOS, macOS, watchOS и tvOS. Тя предлага декларативен подход към създаването на потребителски интерфейси, като позволява на разработчиците да дефинират визуални елементи и техните взаимодействия с помощта на Swift код. Swift UI автоматично обновява интерфейса, когато данните се променят, като осигурява бързо и ефективно изграждане на приложения. Технологията включва широка гама от вградени компоненти и елементи на дизайн, които могат да се комбинират и персонализират за съответните нужди на приложението. С Swift UI разработчиците имат възможността да създадат приложения с висока производителност, красив дизайн и потребителски интерфейси, които се адаптират към различните устройства и размери на экрана.

2.3. Избор на бекенд технология и обектно релационно картографиране

Nest JS е технология за разработка на уеб приложения и сървърни приложения, базирана на Node.js. Тя предоставя елегантна и мощна архитектура, вдъхновена от Angular, което я прави лесна за употреба и поддържане. Nest предлага модулна структура, инжектиране на зависимости, обработка на HTTP заявки и много други функции, които правят разработката на приложения бърза и ефективна.

TypeORM е библиотека за обектно релационно картографиране за TypeScript и JavaScript. Тя предоставя инструменти за лесно и удобно управление на бази данни, като автоматично създаване на таблиците в базата данни, мапиране на обекти към редове в тези таблиците и изпълнение на заявки. TypeORM позволява на разработчиците да работят с бази данни, използвайки обектно-ориентиран подход, което улеснява и ускорява разработката на приложения, които изискват връзка с база данни.

TypeORM има удобна интеграция с NestJS, като предоставя мощни възможности за работа с бази данни в контекста на Nest приложенията.

2.4. Избор на облачна технология

За проекта ще използваме облачната услуга Firestore. Amazon S3 е уеб услуга за съхранение на данни в облака, предоставена от Firebase. Тя е част от Google Cloud Platform и предлага реално време синхронизиране и синхронизация на данни между приложения и облак. Firestore е проектирана да бъде изключително лесна за използване, като осигурява възможности за съхранение на данни, както и богат набор от възможности за търсене и филтриране на данни. Тя се използва широко за създаване на уеб и мобилни приложения, които изискват реално време обновяване на данни и мащабируемост.

Използваме я за съхранение на всички снимки в приложението, като след качването им запазваме имената им в базата данни, след което чрез тези имена ги достъпваме, когато са ни нужни

2.5. Избор на програмна среда

XCode е най-популярният избор за разработка на приложения на Swift UI поради няколко причини:

1. Xcode предоставя интелигентно допълване на кода, което улеснява писането и навигацията в Swift кода, като предлага автоматично допълване на синтаксиса и предложения за методи и свойства.

2. Xcode разполага със мощен дебъгер, който позволява на разработчиците да проследяват и отстраняват грешки в кода си. Това включва възможността за поставяне на точки на прекъсване, проверка на стойностите на променливите и анализ на стека на извикванията.
3. Xcode има интегрирана поддръжка за Swift UI, което улеснява разработката на потребителски интерфейси с помощта на декларативния подход на Swift UI. Това включва автоматично допълване на кода, подчертаване на синтаксиса и интегрирана документация.
4. Xcode предоставя вградени симулатори за различните устройства на Apple, които позволяват на разработчиците да тестват и отлагат своите приложения директно на своите компютри.
5. Xcode разполага с богата библиотека от готови компоненти и шаблони, които могат да бъдат използвани за създаване на различни елементи на потребителския интерфейс.

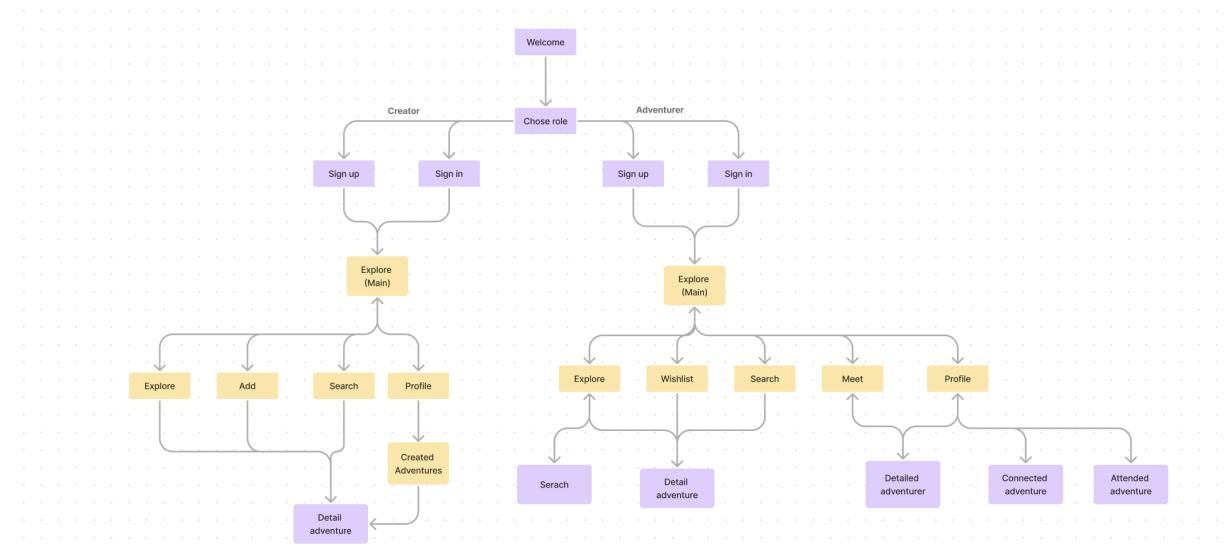
2.6. Структура на экраните

Приложението е съставено от 15 экрана. “Welcome” е първият экран, който се зарежда при стартиране на приложението за първи път. От него могат да се достъпи экраните за “Choose role”, където се избира тип потребителски профил, който ще бъде използван. “Choose role” пренасочва към страниците за регистрация или вход, в зависимост от избрания тип. След регистрация или вход в приложението, потребителят се отвежда към началния экран. Началният экран е различен за двета типа потребителски профили.

При регистрация или влизане като приключенец заедно с началният экран ще бъдат достъпни и экраните “Explore adventures”, “Wishlist”, “Meet” и “Profile”, които могат да бъдат достъпени от навигационната лента. От “Explore adventures”, при избиране на конкретно приключение потребителят е препратен на “Detailed Adventure”, в което е детайлната информация за избраното приключение. От бутона в горният десен ъгъл на экрана може да бъде отворен экрана за търсене на приложение по ключови думи - “Search”, При еcran

“Wishlist”, избирайки някое от запазените приключения , отново сме препратени към “Detailed Adventure”. В еcran “Meet” където са профилите на приключенците, може да се достъпи “Profile” на конкретен приключенец. В “Profile” , достъпен от навигационната лента имаме възможност да достигнем еcran “Connected adventurers”, където се показват потребителите, с които сегашният ни потребител е свързан както и чрез показаните приключения, в които е взел участие да достигне до “Detailed Adventure” за всяко едно.

При регистрация или влизане като създател заедно с началният еcran ще бъдат достъпни и екраните “Explore adventures”, “Add” , “Search”и “Profile”, които могат да бъдат достъпни от навигационната лента. При достъпване на “Explore adventures”, потребителят може да бъде препратен към “Detailed Adventure” за избрано от него приключение. При екрана “Add”, след създаване на ново събитие, потребителят е отведен на “Detailed Adventure” за новосъздаденото събитие. Секция “Search” предоставя достъп на създателя до екрана за търсене на приключения по ключови думи. В “Profile”, достъпен от навигационната лента, имаме възможност да достигнем до “Detailed Adventure” за избрано приключение от създадените.



Фиг. 2.1. Структура на екраните

2.5.2 Структура на базите данни

Използваме PostgreSQL за съхранение на данни и TypeORM за свързване на базата данни с Nest.js.

PostgreSQL, често наричан "Postgres", е мощна релационна база данни с отворен код, която е известна със своята надеждност, гъвкавост и производителност. Тя е сред най-разпространените релационни бази данни в света и се използва широко във всякакви видове приложения, от малки уеб сайтове до големи корпоративни системи.

Избираме да работим с PostgreSQL поради нейните възможности. PostgreSQL следва традиционния релационен модел на данни, който включва таблици, редове и връзки между тях. Това го прави подходящ избор за проекти, които изискват структурирано съхранение на данни. Предлага различни техники за оптимизация на заявките и индексиране на данни, което позволява на базата данни да работи ефективно дори с големи обеми данни и натоварване. Благодарение на своя отворен код лиценз и активната общност от разработчици, PostgreSQL продължава да се развива и да добавя нови функционалности и подобрения на редовни интервали. Тя предлага множество разширения и възможности за конфигурация, които позволяват на разработчиците да го приспособят към конкретните изисквания и нужди на техните проекти.

TypeORM поддържа множество функции и възможности, включително автоматично създаване на миграции за базата данни, валидация на данните, транзакции и много други. То е съвместимо с различни релационни бази данни, като PostgreSQL, MySQL, SQLite и други, което дава свобода на разработчиците да изберат подходящата база данни за техните нужди.

Комбинацията от PostgreSQL и TypeORM предоставя надеждна и ефективна среда за съхранение и управление на данни. Тя осигурява гъвкавост, мащабируемост и удобство при разработката, като същевременно гарантира висока производителност и сигурност на данните.

Базата данни се състои от три таблици, представящи обекти:

Adventurer е таблицата представяща обекта приключенец. Тя съдържа :

- *id* е основният ключ на таблицата Adventurer. То се генерира автоматично и идентифицира по уникален начин всеки приключенец.
- *username* съхранява потребителското име на приключенца. То е задължително и не допуска празни стойности.
- *email* записва имейл адресът на приключенца. То е задължително и трябва да е валиден имейл адрес.
- *password* записва паролата на приключенца. Тя е задължителна и трябва да има минимална дължина от 6 символа
- *profilePhoto* записва URL на изображението, което служи за профилна снимка
- *attendedAdventureIds* това поле е масив, в който се съхраняват идентификаторите на приключенията, в които е участвал приключенецът. Стойността по подразбиране е празен масив и е от тип `int[]`.
- *wishlistAdventureIds* е масив, в който се съхраняват идентификаторите на приключенията, които приключенецът е добавил в списъка си с желания. То е от тип `int[]`.
- *connectedAdventurers* представлява масив, в който се съхраняват идентификаторите на други прилюченци, с които е свързан настоящият приключенец. По подразбиране стойността му е празен масив и е от тип `int[]`

Adventurer	
id ↩	integer
username	string
email	string
password	string
profilephoto	string
attendedAdventuresIds	int[]
wishlistAdventuresIds	int[]
conectedAdventuresIds	int[]

Фиг. 2.2. Структура на таблицата приключенец

Creator е таблицата представляща обекта създател. Тя съдържа :

- *id* е основният ключ на таблицата Creator. То се генерира автоматично и идентифицира по уникален начин всеки създател.
- *username* съхранява потребителското име на създателя. То е задължително и не допуска празни стойности.
- *email* записва имейл адресът на създателя. То е задължително и трябва да е валиден имейл адрес.
- *password* записва паролата на създателя. Тя е задължителна и трябва да има минимална дължина от 6 символа.
- *profilePhoto* записва URL на изображението, което служи за профилна снимка
- *createdAdventurers* представлява масив, в който се съхраняват идентификаторите на приключенията, създадени от дадения създател. По подразбиране стойността му е празен масив и е от тип int[]

Creator	
id 	integer
username	string
gmail	string
password	string
profilephoto	string
createdAdventurers	Adventure[]

Фиг. 2.3. Структура на таблицата създател

Adventure е таблицата представляща обекта приключение. Тя съдържа :

- *id* е основният ключ на таблицата. То се генерира автоматично и идентифицира по уникален начин всяко приключение.
- *name* е името на даденото приключение.
- *description* записва описанието на приключението.
- *photoURL* записва URL на изображението на приключението
- *creator* представлява създателя на приключението. То има връзка много към едно между приключенията и създателите. Всяко приключение е свързано с един създател.

Adventure	
id ↕	integer
name	string
description	string
photoURLs	string[]
creator	Creator

Фиг. 2.4. Структура на таблицата приключение

2.5.3. Структура на кода

MVVM (Model-View-ViewModel). MVVM е архитектурен модел, разработен от Microsoft. Той разделя бизнес логиката на приложението от потребителския интерфейс. Главната цел на MVVM архитектурата е да направи интерфейса напълно независим от логиката на приложението. Начинът на работа на архитектурата е представен нагледно на *Фиг. 2.5.*

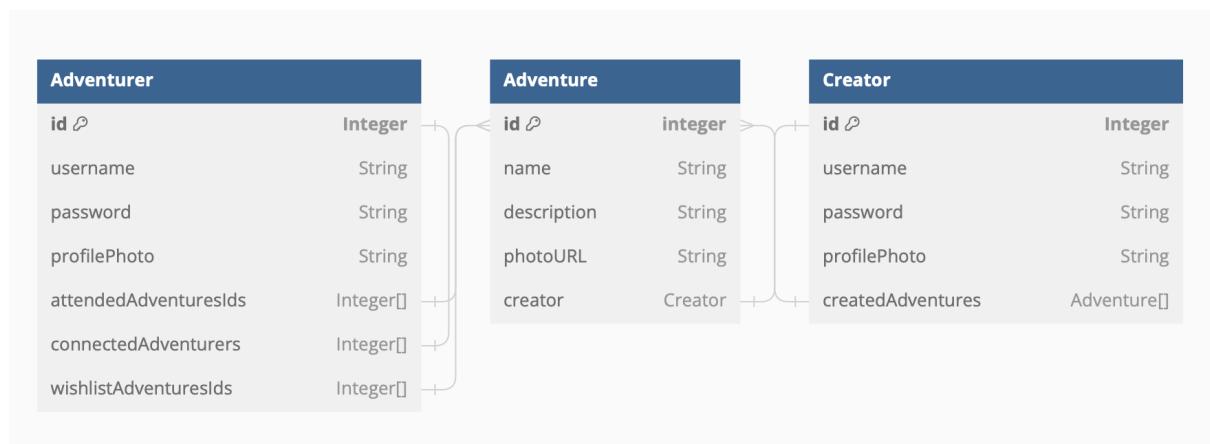
Model - Моделът представлява модела на домейна на приложението, който може да включва модел на данни, бизнес логика и логика за валидиране. Моделът съдържа извлечените данни от локална база данни или от уеб услуга. Той комуникира с ViewModel и не си взаимодейства с потребителския интерфейс.

View - View представлява потребителския интерфейс на приложението и съдържа ограничена, единствено презентационна логика. Изгледът не съдържа данни, нито ги манипулира директно. Той комуникира само с ViewModel чрез обвързване на данни и не си взаимодейства с Model.

ViewModel - ViewModel е връзката между View и Model, неговата задача е да предоставя данни за компонентите на потребителския интерфейс. Той също така включва и притежател на данни, наречен LiveData, който позволява на ViewModel да информира или актуализира View, когато данните се актуализират.



Фиг. 2.5. MVVM архитектура



Фиг. 2.6. Архитектура на базата данни

Трета глава

Програмна реализация на платформа за създаване и участие в различни приключенски събития

3.1. Основни компоненти и конфигурация с базата данни PostgreSQL посредством TypeORM

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  await app.listen(3001);
}

bootstrap();
```

Фиг. 3.1. Основният файл - начало на програмата main.ts

На (Фиг. 3.1) е показан основният файл, който създава и стартира уеб приложение с помощта на Nest.js. Той импортира основния клас "NestFactory" за създаване на приложения и главния модул на приложението, след което го стартира на порта, който сме избрали. В главния модул на приложението (Фиг. 3.2) в Nest.js се дефинират и конфигурират различни компоненти и модули, които са необходими за функционирането на приложението. Това включва конфигуриране на базата данни чрез TypeORM, настройка на JWT за автентикация, импортиране на други модули за различни части на приложението, като се включват съответните контролери и сервиси.

```

@Module({
  imports: [
    ConfigModule.forRoot(),
    AdventurerModule,
    CreatorModule,
    AdventureModule,
    TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: async (configService: ConfigService) => ({
        type: configService.get<string>('DB_TYPE'),
        host: configService.get<string>('DB_HOST'),
        port: configService.get<number>('DB_PORT'),
        username: configService.get<string>('DB_USERNAME'),
        password: configService.get<string>('DB_PASSWORD'),
        database: configService.get<string>('DB_DATABASE'),
        entities: [__dirname + '/*.*.entity{.ts,.js}'],
        synchronize: true,
      } as TypeOrmModuleOptions),
    }),
    JwtModule.registerAsync({
      imports: [ConfigModule],
      useFactory: async (configService: ConfigService) => ({
        secret: configService.get<string>('JWT_SECRET'),
        signOptions: { expiresIn: configService.get<string>('JWT_EXPIRES_IN') },
      }),
      inject: [ConfigService],
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}

```

Фиг. 3.2. app.module.ts

В секцията imports на класа AppModule се конфигурират и добавят всички импортирани модули и техните опции, които ще се използват в приложението.

В секцията controllers се посочват всички контролери, които са част от този модул (в случаят само AppController).

В секцията providers се посочват всички сервиси (провайдъри), които ще бъдат използвани в модула (в случаят само AppService)

3.1.1 Модел на приключение

Клас Adventure представлява модел за съхранение на информация за приключения в базата данни, който използва TypeORM за работа с PostgreSQL база данни.

```
    ...
    @Entity()
export class Adventure {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    name: string;

    @Column()
    description: string;

    @Column("int", { array: true, default: [] })
    attendedAdventurerIds: number[];

    @Column("string", { array: true, default: [] })
    photoURLs: string[];

    @ManyToOne(() => Creator, { cascade: true })
    @JoinTable()
    creator: Creator;
}
```

Фиг. 3.3. Модел на приключенец - adventure.entity.ts

TypeORM разполага с декоратори с чиято помощ успява да интегрира моделите от данни в базата, която сме избрали.

Декораторът **@Entity()** маркира класа Adventure като съответствие на таблица в базата.

@PrimaryGeneratedColumn() задава полето id като основен ключ на таблицата и автоматично генерира уникални стойности за всеки нов запис.

@Column() маркира полетата name и description като колони в таблицата със съответните имена.

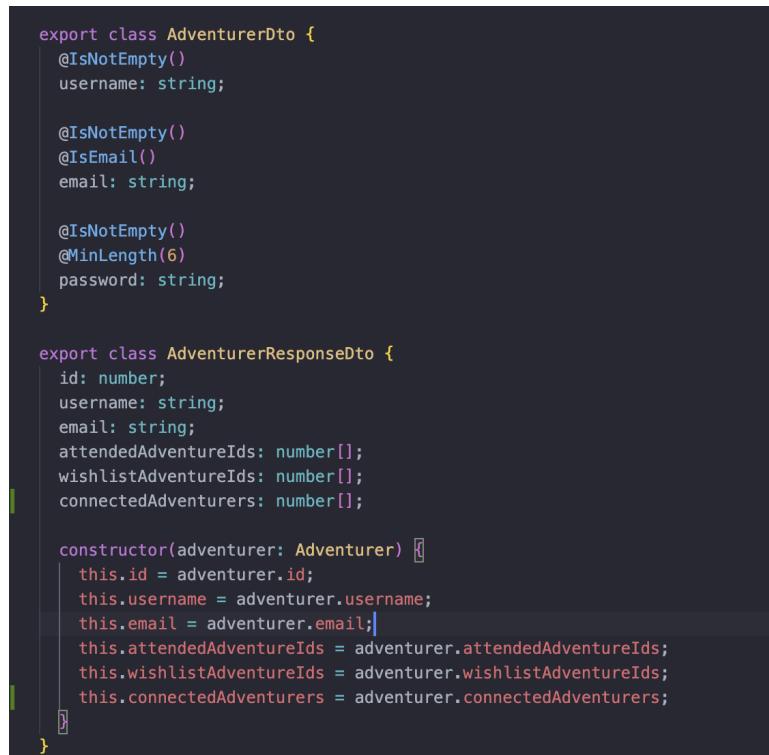
@Column("int", { array: true, default: [] }): маркира

attendedAdventurerIds като масив от цели числа в базата данни и му задава стойност по подразбиране празен масив.

@Column("string", { array: true, default: [] }): маркира photoURLs като масив от низове в базата данни и му задава стойност по подразбиране празен масив.

@ManyToOne(() => Creator, { cascade: true }): установява връзка "много към един" между приключението и създателя му (Creator). Той посочва, че всяко приключение има само един създал и се използва **cascade: true**, за да позволи автоматична промяна ако бъде променено приключението.

AdventureDto и AdventureResponseDto (*Фиг.3.4.*) са обекти, които се използват за обработка на данни при входящи и изходящи заявки. AdventureResponseDto съдържа данни за приключението, които се изпращат към клиента, включително име на създателя. Той се изгражда чрез конструктор, който взема обект от тип Adventure и извлича необходимите данни от него.



```
export class AdventurerDto {
  @IsNotEmpty()
  username: string;

  @IsNotEmpty()
  @IsEmail()
  email: string;

  @IsNotEmpty()
  @MinLength(6)
  password: string;
}

export class AdventureResponseDto {
  id: number;
  username: string;
  email: string;
  attendedAdventureIds: number[];
  wishlistAdventureIds: number[];
  connectedAdventurers: number[];

  constructor(adventurer: Adventurer) {
    this.id = adventurer.id;
    this.username = adventurer.username;
    this.email = adventurer.email;
    this.attendedAdventureIds = adventurer.attendedAdventureIds;
    this.wishlistAdventureIds = adventurer.wishlistAdventureIds;
    this.connectedAdventurers = adventurer.connectedAdventurers;
  }
}
```

Фиг. 3.4. AdventureDTO & AdventureDTOResponse

3.1.2 Модел на приключенец

Клас Adventurer представлява модел за съхранение на информация за приключенията на участниците в базата данни.

```
@Entity()
export class Adventurer extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    @IsNotEmpty()
    username: string;

    @Column({unique: true})
    @IsNotEmpty()
    @IsEmail()
    email: string;

    @Column()
    @IsNotEmpty()
    @MinLength(6)
    password: string;

    @Column()
    profilephoto: string;

    @Column("int", { array: true, default: [] })
    attendedAdventureIds: number[];

    @Column("int", { array: true })
    wishlistAdventureIds: number[];

    @Column("int", { array: true, default: [] })
    connectedAdventurers: number[];
}
```

Фиг. 3.5. adventurer.entity.ts

@Entity() маркира класа Adventurer като съответствие на таблица в базата данни. Класът Adventurer наследява BaseEntity от TypeORM, който предоставя допълнителни методи и функционалности за работа с базата данни.

@PrimaryGeneratedColumn() задава полето id като основен ключ на таблицата и автоматично генерира уникални стойности за всеки нов запис.

@Column() маркира полетата username, email, password и profilephoto като колони в таблицата със съответните имена.

Полетата attendedAdventureIds, wishlistAdventureIds и connectedAdventurers в базата данни са масиви от цели числа, които могат да съдържат множество стойности. Опцията "{ array: true }" указва, че тези полета трябва да бъдат интерпретирани като масиви от стойности в базата данни. Опцията "{ default: [] }" задава празен масив като стойност по подразбиране, ако не е зададена друга стойност. Това означава, че при създаване на нов запис в таблицата, съответните полета ще имат празен масив като стойност по подразбиране.

attendedAdventureIds съдържа идентификаторите на приключенията, в които приключенецът е взел участие, wishlistAdventureIds съдържа идентификаторите на приключенията, които участникът желае да посети, а connectedAdventurers се използва за съхранение на идентификаторите на други прилюченци, с които даден приключенец е свързан

Декораторът **@BeforeInsert()** определя метод, който се изпълнява преди вмъкване на нов запис в базата данни. В този случай, методът "hashPassword()" криптира паролата преди записване, използвайки библиотеката bcrypt.

```
51  @BeforeInsert()
52  async hashPassword() {
53      const errors: ValidationErrors[] = await validate(this, { skipMissingProperties: true });
54
55      if (errors.length > 0) {
56          throw new Error(errors.toString());
57      }
58
59      this.password = await bcrypt.hash(this.password, 10);
60  }
```

Фиг. 3.6. hashPassword()

AdventurerDto представлява обект за създаване на нови участници. Той съдържа полета за потребителско име, имейл и парола. За всеки от тези атрибути се прилага валидация, която изисква те да не са празни. За имейла се изисква и да е валиден имейл адрес, а за паролата - минимална дължина от 6 символа.

AdventurerResponseDto представлява обект за връщане на данни за участниците. Той съдържа полета като идентификационен номер, потребителско име, имейл и масиви с идентификатори на приключения. Конструкторът му приема обект от тип Adventurer и извлича необходимите данни от него, за да ги включи в обекта на DTO.

```
export class AdventureDto {
    name: string;
    description: string;
    attendedAdventurerIds: number[];
}

export class AdventureResponseDto {
    id: number;
    name: string;
    description: string;
    creatorName: string;

    constructor(adventure: Adventure) {
        this.id = adventure.id;
        this.name = adventure.name;
        this.description = adventure.description;
        this.creatorName = adventure.creator.username
    }
}
```

Фиг. 3.7. AdventurerDto & AdventureResponseDto

3.1.3 Модел на създател

Клас Creator също е дефиниран като Entity - съответствие на таблица в базата данни.

```
src/entities/creator/creator.entity.ts > ↗ Creator > ↗ password
@Entity()
export class Creator extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  username: string;

  @Column()
  @IsEmail({}, { message: 'Invalid email format' })
  email: string;

  @Column()
  @IsNotEmpty()
  @MinLength(6)
  password: string;

  @Column()
  profilephoto: string;

  @ManyToMany(() => Adventure, { cascade: true, })
  @JoinTable()
  createdAdventures: Adventure[];

  @BeforeInsert()
  async hashPassword() {
    const errors: ValidationErrors[] = await validate(this, { skipMissingProperties: true });

    if (errors.length > 0) {
      throw new Error(errors.toString());
    }

    this.password = await bcrypt.hash(this.password, 10);
  }
}
```

Φυλ. 3.8. creator.entity.ts

Декораторът **@Entity()** маркира класа Creator като съответствие на таблица в базата данни.

Класът Creator също наследява BaseEntity от TypeORM, което предоставя допълнителни методи и функционалности за работа с базата данни.

@PrimaryGeneratedColumn() задава полето id като основен ключ на таблицата и автоматично генерира уникални стойности за всеки нов запис.

@Column() маркира полетата username, email, password и profilephoto като колони в таблицата със съответните имена. За email се прилага валидация за валиден имейл формат.

Декораторът **@ManyToMany()** указва, че създателят може да има много приключения, и че тази връзка е много към много. { **cascade: true** } позволява автоматично да се добавят или премахват свързани приключения, когато се променя създателят.

@BeforeInsert() определя метод, който се изпълнява преди вмъкване на нов запис в базата данни. И тук използваме bcrypt за да криптиране паролата.

CreatorDto и CreatorResponseDto са класове за обработка на данни при входящи и изходящи заявки. CreatorResponseDto съдържа данни за създателя, които се изпращат към клиента, включително идентификационен номер, потребителско име и имейл.

Конструкторът на CreatorResponseDto взема обект от тип Creator и извлича необходимите данни от него, за да ги включи в обекта на DTO.

```
1
2  export class CreatorDto {
3      username: string;
4      email: string;
5      password: string;
6
7  }
8
9  export class CreatorResponseDto {
10     id: number;
11     username: string;
12     email: string;
13     createdAdventures: Adventure[];
14
15     constructor(creator: Creator) {
16         this.id = creator.id;
17         this.username = creator.username;
18         this.email = creator.email;
19         this.createdAdventures = creator.createdAdventures;
20     }
21 }
22
```

Фиг. 3.9. *CreatorDto & CreatorResponseDto*

3.2 Автентикация и създаване на потребителски профили

3.2.1. Автентикация

В приложението, аутентикацията се осъществява чрез използването на JSON Web Tokens (JWT) и специален guard, наречен "AuthenticationGuard". JWT е акроним за "JSON Web Token". Това е стандартизиран формат за представяне на обекти на база JSON структура. JWT се използва предимно за сигурно предаване на информация между две страни по интернет, особено между уеб приложения и уеб услуги. Състои се от три части:

Заглавка - съдържа типа на токена и алгоритъма за криптиране, използван при подписването на токена.

Съдържание - самата информация, която се предава между две страни.

Това може да бъде потребителска идентификация, разрешения, временни маркери и други.

Подпись - част, която се създава като подписва заглавката и съдържанието с помощта на секретен ключ (или чрез публичен/частен ключ при използване на асиметрични алгоритми). Този подпись се използва за проверка на токена и гарантира, че данните не са били променяни по време на транспортирането. JWT-тата са често използвани за аутентикация и авторизация. Например, след успешна аутентикация на потребител в уеб приложение, сървърът може да генерира JWT и да го изпрати към клиента. Клиентът може да съхранява този токен и да го включва във всички последващи заявки към сървъра. Сървърът може да провери токена за валидност и да извлече информацията, която е била добавена към него при генерирането му, за да реши дали да разреши заявката или не.

В апликацията първо, при създаване на нов потребител чрез `createAdventurer()` или `createCreator()`, потребителското име, имейлът и паролата се подават към съответния метод. Паролата се хешира чрез библиотеката "bcrypt", преди да бъде записана в базата данни.

След това, когато потребител желае да извърши операции, които изискват аутентикация, "AuthenticationGuard" е прикачен към съответните методи чрез декоратора "@UseGuards(AuthenticationGuard)".

В NestJS, "гардовете" са механизъм за изпълнение на логика, преди или след обработването на HTTP заявките във входящия HTTP пайплайн на приложението. Те позволяват на приложението да извършва аутентикация, авторизация, валидация и други действия в зависимост от нуждите на приложението. Гардовете в NestJS се използват чрез класовете, които имплементират определен интерфейс. Тези интерфейси определят методи, които трябва да бъдат реализирани и се използват за управление на потока на заявките. Най-често използваният интерфейс е `CanActivate`, който се използва за управление на достъпа до ресурси. Гардовете могат да проверяват дали

потребителят е автентициран преди да бъде разрешен достъп до определени ресурси или операции.

"AuthenticationGuard" проверява наличието на JWT токен в заглавките на HTTP заявката. Токенът се извлича и декодира, използвайки "JwtService.verify()". Ако токенът е валиден, декодираният обект се добавя към обекта на заявката като "request.user", позволявайки по-нататъчни операции да имат достъп до информацията за потребителя.

В случай, че токенът е невалиден или липсва, се генерира грешка от тип "UnauthorizedException", като съобщението на грешката указва проблема (например "Token not provided" или "Invalid token").

```
@Injectable()
export class AuthenticationGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = request.headers.authorization?.replace('Bearer ', '');

    if (!token) {
      throw new UnauthorizedException('Token not provided');
    }

    try {
      const decoded = this.jwtService.verify(token);
      if (decoded) {
        request.user = decoded;
        return true;
      }
    } catch (error) {
      throw new UnauthorizedException('Invalid token');
    }

    throw new UnauthorizedException('Invalid token');
  }
}
```

Фиг. 3.10. AuthenticationGuard

По този начин, чрез комбинация от JWT и "AuthenticationGuard", приложението осигурява сигурен механизъм за аутентикация, който позволява само на

упълномощени потребители да достъпват определени ресурси и функционалности.

3.2.2 Създаване на потребителски профил

Създаването на потребителски профил е в зависимост от типа на потребителя.

Приключенец:

Когато се изпраща POST заявка към тази входна точка, тя се предава към функцията `createAdventurer` на `AdventurerController`. Тази функция очаква данните за новия потребител да бъдат предоставени в тялото на заявката и ги извлича чрез параметъра `adventurerDto`. След като са получи данните за новия потребител, функцията извика метода `create` на `adventurerService`, който е отговорен за създаването на новия потребител в системата. След като потребителят е успешно създаден, обработката на заявката завършва, и входната точка връща отговор, който съдържа създадения потребител, като потвърждение, че операцията е изпълнена успешно.

```
@Post()  
async createAdventurer(@Body() adventurerDto: AdventurerDto): Promise<Adventurer> {  
    return await this.adventurerService.create(adventurerDto);  
}
```

Фиг. 3.11. `createAdventurer()` в controller

Функцията “`create()`” създава нов потребител в базата данни, като използва предоставените данни за потребителския профил и ги записва в съответната таблица. Преди записването, функцията инициализира празни списъци за различните типове приключвания на потребителя, за да гарантира, че няма предварително дефинирани стойности за тези списъци. Накрая, функцията връща обекта на създадения потребител, като потвърждение, че операцията е изпълнена успешно.

```

    async create(adventurer: AdventurerDto): Promise<Adventurer> {
      const newAdventurer = this.adventurersRepository.create(adventurer);
      newAdventurer.wishlistAdventureIds = [];
      newAdventurer.connectedAdventurers = [];
      newAdventurer.attendedAdventureIds = [];
      return await this.adventurersRepository.save(newAdventurer);
    }

```

Фиг. 3.12. create() в adventurer.service.ts

В модела имаме функция “createUserAdventurer()”, която създава нов потребител в системата, като изпраща POST заявка към сървъра.

```

mutating func createUserAdventurer(username: String, email: String, password: String, validateUser: @escaping (String, String, @escaping (Result<String, Error>) -> Void) -> Void, completion: @escaping (Result<Adventurer, Error>) -> Void) {
  let adventurerDto = AdventurerDto(username: username, email: email, password: password)
  guard let url = URL(string: "http://localhost:3001/adventurer") else {
    let error = NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid URL"])
    completion(.failure(error))
    return
  }
  var request = URLRequest(url: url)
  request.httpMethod = "POST"
  request.setValue("application/json", forHTTPHeaderField: "Content-Type")
  do {
    let jsonData = try JSONEncoder().encode(adventurerDto)
    request.httpBody = jsonData
    URLSession.shared.dataTask(with: request) { data, response, error in
      if let error = error {
        completion(.failure(error))
        return
      }
      let jsonDecoder = JSONDecoder()
      let adventurer = try jsonDecoder.decode(Adventurer.self, from: data!)
      completion(.success(adventurer))
    }
  }
}

```

Фиг. 3.13. createUserAdventurer()

Тя приема параметри, необходими за създаване на нов приключенец – потребителско име, имейл и парола. Освен това, функцията приема затворен блок (validateUser), който се използва за валидация на потребител, както и блок за завършване, който се извиква, когато операцията приключи. Създаден е обект от тип AdventurerDto, представляващ данните за новия приключенец. Проверява се валидността на URL адреса, където ще се изпрати заявката за създаване на нов приключенец. Ако URL адресът не е валиден, се връща грешка чрез блока за завършване и се прекратява изпълнението на функцията. Създава се HTTP заявка за POST операция към указанния URL адрес. Също така се задават хедъри за заявката, в този случай се задава Content-Type като application/json.

```

var request = URLRequest(url: url)
request.httpMethod = "POST"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")
do {
    let jsonData = try JSONEncoder().encode(adventurerDto)
    request.httpBody = jsonData
}
URLSession.shared.dataTask(with: request) { data, response, error in

```

Фиг. 3.14. Създаване на заявка

Създава се HTTP заявка (*Фиг. 3.13*) за POST операция към указанния URL адрес. Също така се задават хедъри за заявката, в този случай се задава Content-Type като application/json. Обектът adventurerDto се свежда до JSON формат и се задава като тяло на HTTP заявката

```

URLSession.shared.dataTask(with: request) { data, response, error in
    if let error = error {
        completion(.failure(error))
        return
    }
    if let data = data {
        do {
            let adventurer = try JSONDecoder().decode(Adventurer.self, from: data)
            validateUser(email, password) { result in
                switch result {
                case .success(let token):
                    jwtToken = token
                    currentAdventurer.id = adventurer.id
                    currentAdventurer.username = adventurer.username
                    currentAdventurer.email = adventurer.email
                    currentAdventurer.password = adventurer.password

                    completion(.success(adventurer))
                case .failure(let error):
                    completion(.failure(error))
                }
            }
        } catch {
            completion(.failure(error))
        }
    }
}.resume()

```

Фиг. 3.15. Изпращане на заявка и обработка на отговора

Изпраща се заявката към сървъра и се изчаква отговор. Данните от отговора се декодират от JSON формат до обект от тип Adventurer. При създаване и валидиране сървърът връща JWT токена, който запазваме, за да може успешно регистрирал се потребител да има достъп до действията. Запазват се също атрибутите на току-що създаденият потребител в променливата, използвана за

следене на текущият потребител, който използва платформата. Ако възникне грешка при декодирането на отговора от сървъра или при изпращането на заявката, се извиква блока за завършване с грешка.

```
        .resume()
    } catch {
        completion(.failure(error))
    }
}
```

Фиг. 3.16. Обработка на грешка

Създател:

“createCreator()” е функция, която обработва POST заявки за създаване на нов “creator”. Той извиква съответната услуга (“creatorService.create()”) за създаване на “creator” и връща обещание за резултата от операцията.

```
@Post()
async createCreator(@Body() creatorDto: CreatorDto): Promise<Creator> {
    return await this.creatorService.create(creatorDto);
}
```

Фиг. 3.17. createCreator() в controller.creator.ts

Този метод с име “create” създава нов създател в базата данни. Методът е асинхронен, като приема обект от тип “CreatorDto”, който съдържа информацията за създаването на създател. В тялото на метода, се създава нов създател като се извиква методът “create()” с предоставения обект. След това, резултатът от създаването на новия създател се запазва в променлива “newCreator”. Следващият ред използва “await”, за да изчака завършването на асинхронната операция. Това се прави с цел да се уверим, че новосъздаденият създател е успешно записан в базата данни. Когато операцията приключи, се връща резултатът от записването на създател в базата данни

```

async create(creator: CreatorDto): Promise<Creator> {
    const newCreator = this.creatorsRepository.create(creator);
    return await this.creatorsRepository.save(newCreator);
}

```

Фиг. 3.18. create в service.creator.ts

В модела имаме функцията createCreator(), която приема потребителско име, парола и затворен блок за валидация.

```

mutating func createCreator(username: String, email: String, password: String, validateUser: @escaping (String, String, @escaping (Result<String, Error>) -> Void) -> Void, completion: @escaping (Result<Creator, Error>) -> Void) {
    let creatorDto = CreatorDto(username: username, email: email, password: password)
    guard let url = URL(string: "http://localhost:3001/creator") else {
        let error = NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedStringKey: "Invalid URL"])
        completion(.failure(error))
        return
    }
}

```

Фиг. 3.19. createCreator() в CreatorModel

Също така приема блок за завършване, който се извиква, когато операцията приключи. Създава се обект от тип CreatorDto, който съдържа информацията за новия създалел.

Проверява се валидността на URL адреса, където ще се изпрати заявката за създаване на създалел. Ако URL адресът не е валиден, се връща грешка чрез блока за завършване и се прекратява изпълнението на функцията.

```

var request = URLRequest(url: url)
request.httpMethod = "POST"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")
do {
    let jsonData = try JSONEncoder().encode(creatorDto)
    request.httpBody = jsonData
}

```

Фиг. 3.20. Създаване на заявка

Създава се HTTP заявка за POST операция към указания URL адрес. Също така се задават хедъри за заявката, в този случай се задава Content-Type като application/json. Обектът creatorDto се сериализира до JSON формат и се задава като тяло на HTTP заявката

```

URLSession.shared.dataTask(with: request) { data, response, error in
    if let error = error {
        completion(.failure(error))
        return
    }
    if let data = data {
        do {
            let creator = try JSONDecoder().decode(Creator.self, from: data)

```

Фиг. 3.21. Изпращане на заявка към сървъра

Изпраща се заявката към сървъра и се изчаква отговор. Проверява се дали има данни в отговора от сървъра. Данните от отговора се декодират от JSON формат до обект от тип Creator.

Извиква се затвореният блок с функцията validateUser(), която се използва за валидация на потребителя. Обработва се резултатът от валидацията. Ако потребителят е валиден, токенът се запазва, информацията за текущия "creator" се актуализира с информацията от отговора на сървъра, и се извиква блока за завършване с успех, предавайки информацията за създадения "creator".

```

validateUser(email, password) { result in
    switch result {
        case .success(let token):
            jwtToken = token
            currentCreator.id = creator.id
            currentCreator.username = creator.username
            currentCreator.email = creator.email
            currentCreator.password = creator.password

            completion(.success(creator))
        case .failure(let error):
            completion(.failure(error))
    }
}

```

Фиг. 3.22. validateUser() и актуализация на текущия потребител

Ако възникне грешка при декодирането на отговора от сървъра или при изпращането на заявката, се извиква блока за завършване (completion) с грешка.

```
    } catch {
        completion(.failure(error))
    }
}
```

Фиг. 3.23. Обработка на грешка

3.2.3 Валидация потребителски профил и вход

Валидацията на профилите зависи от типа на потребителите.

Потребителска валидация:

POST заявки се обработват посредством функцията `AdventurerValidation()`. Тя приема данни за потребител, които включват имейл и парола. В метода се извличат имейлът и паролата от подадените данни. След това се извиква методът `'validate'` на услугата `'adventurerService'` със зададените имейл и парола. Този метод проверява валидността на потребителските данни.

```
@Post('validate')

async AdventurerValidation(@Body() userData: { email: string; password: string }) {
    const { email, password } = userData;
    const isValid = await this.adventurerService.validate(email, password);
    if (isValid) {
        const payload = { email };
        const token = this.jwtService.sign(payload);
        console.log(token);
        return { token };
    }
    throw new UnauthorizedException('Invalid email or password');
}
```

Фиг. 3.24. `AdventureValidation()` в `controller.creator.ts`

След като се извърши валидацията, ако потребителските данни са валидни, се генерира токен за достъп към приложението чрез 'jwtService'. Този токен съдържа информация за имейла на потребителя. Токенът се връща като отговор на заявката. Ако потребителските данни не са валидни, се хвърля изключение със съобщение "Невалиден имайл или парола".

Функцията за валидация в 'adventurerService' е асинхронна, приема два параметъра - имайл и парола. Целта на този метод е да провери дали съществува приключенец със зададения имайл и дали подадената парола съвпада с паролата на този приключенецът.

```
async validate(email: string, password: string): Promise<boolean> {
  const adventurer = await this.findByEmail(email);
  if (adventurer && password === adventurer.password) {
    return true;
  }
  return false;
}
```

Фиг. 3.25. validate() в service.creator.ts

В тялото на метода се извиква методът "findByEmail()" на същата услуга, който търси приключенец по подадения имайл. След това се проверява дали е намерен приключенец и дали паролата, подадена като аргумент, съвпада с паролата на намерения приключенец. Ако и двете условия са изпълнени, методът връща `true`, което означава, че потребителските данни са валидни. В противен случай, методът връща `false`, което означава, че потребителските данни не са валидни.

Метода "findByEmail()" е асинхронен и приема един параметър - имайл. Целта на този метод е да търси приключенец по подадения имайл в базата данни.

```

async findByEmail(email: string): Promise<Adventurer> {
  try {
    const adventurer = await this.adventurersRepository.findOneOrFail({ where: { email } });
    return adventurer;
  } catch (error) {
    if (error.name === 'EntityNotFoundError') {
      throw new NotFoundException('Adventurer not found');
    }
    throw error;
  }
}

```

Фиг. 3.26. findByEmail() в service.creator.ts

В тялото на метода се извиква методът “findOneOrFail()” на хранилището за приключенците. Този метод търси приключенец в базата данни, като филтрира резултатите по подадения имейл. Ако е намерен приключенец със съответния имейл, той се връща като резултат от метода. Ако не е намерен, методът хваща грешката, която може да възникне при опит за намиране на такъв потребител, и я обработва. Ако грешката е типа 'EntityNotFoundError', този тип грешка означава, че не е намерен по съответния имейл. Тогава методът хвърля изключение от тип 'NotFoundException' със съобщение "Приключенецът не е намерен". Ако възникне друг тип грешка, различен от 'EntityNotFoundError', методът я хваща и я хвърля нататък, за да бъде обработена от по-горните слоеве на приложението. В модела имаме метода, правещ заявка за валидиранието на приключенците.

```

func validateUser(email: String, password: String, completion: @escaping (Result<String, Error>) -> Void) {
  let parameters = ["email": email, "password": password]
  guard let url = URL(string: "http://localhost:3001/adventurer/validate") else {
    let error = NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid URL"])
    completion(.failure(error))
    return
  }

  var request = URLRequest(url: url)
  request.httpMethod = "POST"
  request.setValue("application/json", forHTTPHeaderField: "Content-Type")

  do {
    let jsonData = try JSONSerialization.data(withJSONObject: parameters, options: [])
    request.httpBody = jsonData
  }
}

```

Фиг. 3.27. validateUser() в AdventurerModel

В началото на функцията се създава речник, който съдържа имейла и паролата на потребителя, които ще се изпратят към сървъра за валидация. Създава се и URL адрес, към който ще се изпрати заявката. След това се създава HTTP заявка от тип POST към определения URL адрес. В тялото на заявката се включват потребителските данни като JSON обект.

```

URLSession.shared.dataTask(with: request) { data, response, error in
    if let error = error {
        completion(.failure(error))
        return
    }

    if let data = data {
        do {
            guard let token = try JSONSerialization.jsonObject(with: data, options: []) as? [String: Any],
                  let jwtToken = token["token"] as? String else {
                let error = NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid token response"])
                completion(.failure(error))
                return
            }
            completion(.success(jwtToken))
        } catch {
            completion(.failure(error))
        }
    }
}

```

Фиг. 3.28. Изпращане на заявка и обработка на отговора

Следва изпращането на заявката чрез функцията URLSession.shared.dataTask, която асинхронно изпраща заявката към сървъра. Когато се получи отговор от сървъра, се проверява за наличие на грешка. Ако има грешка при изпращането на заявката или при получаването на отговор от сървъра, се извиква затвореният блок completion с резултат от тип failure, който съдържа информация за грешката. Ако няма грешка, се проверява отговорът от сървъра за валиден JSON обект, който съдържа токен за достъп. Ако отговорът е валиден и съдържа токен, този токен се извлича и се връща като резултат от функцията чрез затвореният блок completion с резултат от тип success, който съдържа токена за достъп. Ако отговорът от сървъра не е валиден JSON обект или не съдържа токен, се връща грешка чрез затвореният блок completion с резултат от тип failure, който съдържа информация за грешката.

```

}.resume()
} catch {
    completion(.failure(error))
}

```

Фиг.3.29. Обработка на грешки

В модела има функция, която изпраща заявка към сървъра, за да получи информация за приключенец по зададен имейл адрес.

```

func getAdventurerByEmail(email: String, token: String, completion: @escaping (Result<Adventurer, Error>) -> Void) {
    guard let url = URL(string: "http://localhost:3001/adventurer/email/\\" + email + "\"") else {
        completion(.failure(NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid URL"])))
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.addValue("Bearer " + token, forHTTPHeaderField: "Authorization")
}

```

Фиг. 3.30. getAdventurerByEmail() в AdventurerModel

Проверява се валидността на URL адреса, към който ще се изпрати заявката. Създава се GET заявка към указанния адрес. Към хедъра "Authorization" се добавя токенът, който е предоставен като аргумент на функцията. Този токен се използва за удостоверяване на идентичността на потребителя.

```

URLSession.shared.dataTask(with: request) { data, response, error in
    guard let data = data, error == nil else {
        completion(.failure(error ?? NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Unknown error"])))
        return
    }

    do {
        let adventurer = try JSONDecoder().decode(Adventurer.self, from: data)
        currentAdventurer = adventurer
        print(currentAdventurer.username)
        completion(.success(adventurer))
    } catch {
        completion(.failure(error))
    }
}.resume()
}

```

Фиг.3.31. Изпращане на заявка към сървъра и обработка на отговора

Заявката се изпраща към сървъра чрез функцията "URLSession.shared.dataTask()"

Когато се получи отговор от сървъра, се проверява за наличие на грешка.

Ако има грешка при изпращането на заявката или при получаването на отговор от сървъра, се извиква затвореният блок completion с резултат от тип failure, който съдържа информация за грешката.

Ако няма грешка, се извършва опит за декодиране на получените данни от отговора като обект от тип приключениец. Ако декодирането е успешно, полученият приключениец се запазва чрез currentAdventurer, за да се установи потребителят, който ще използва платформата и се извиква затвореният блок

completion с резултат от успешен тип който съдържа информацията за приключенца.

Ако декодирането не е успешно, се извиква затвореният блок completion с резултат от тип failure, който съдържа информация за грешката, която възниква при декодирането на данните от отговора.

```
5 func signIn(email: String, password: String, completion: @escaping (Result<Void, Error>) -> Void) {
6     validateUser(email: email, password: password) { result in
7         switch result {
8             case .success(let token):
9                 jwtToken = token
10                print("Token: \(token)")
11
12                getAdventurerByEmail(email: email, token: token) { result in
13                    switch result {
14                        case .success(let adventurer):
15                            currentAdventurer = adventurer
16                            print("Current Adventurer: \(adventurer.username)")
17                            completion(.success(()))
18                        case .failure(let error):
19                            print("Error fetching adventurer: \(error)")
20                            completion(.failure(error))
21                    }
22                }
23            case .failure(let error):
24                print("Error validating user: \(error)")
25                completion(.failure(error))
26        }
27    }
28 }
```

Фиг. 3.32. signIn() в AdventurerModel

Функцията signIn() използва метода за валидация на потребителските данни и метода за търсене на приключенец в системата за да предостави вход на съществуващите потребители. Извиква се функцията “validateUser()” с предоставения имейл и парола. Тя има задачата да провери валидността на потребителските данни. Ако валидацията е успешна, токенът, получен от валидацията, се запазва чрез jwtToken. Този токен представлява потвърждение за успешна валидация на потребителските данни. Ако валидацията не е успешна, се извежда съобщение за грешка и се извиква затвореният блок completion с резултат от тип failure, който съдържа информация за грешката. Ако валидацията на потребителските данни е успешна, функцията “getAdventurerByEmail()” се извиква за да търси приключенец със съответния

имейл. Ако търсенето е успешно, информацията за приключението се запазва в глобалната променлива currentAdventurer. След това се извежда съобщение за успешно влизане в системата и се извиква затвореният блок completion с резултат от тип success. Ако търсенето не е успешно (резултатът е от тип failure), се извежда съобщение за грешка и се извиква затвореният блок completion с резултат от тип failure, който съдържа информация за грешката.

Създателска валидация:

Входната точка за валидация на потребителски данни е функция, очакваща данни за създателя - имейл и парола, които се предават чрез POST заявка. В тялото на функцията, чрез декоратора @Body(), се извличат данните за потребител от заявката и се записват в променливата userData.

```
@Post('validate')
async CreatorUserValidation(@Body() userData: { email: string; password: string }) {
  const { email, password } = userData;
  const isValid = await this.creatorService.validate(email, password);
  if (isValid) {
    const payload = { email };
    const token = this.jwtService.sign(payload);
    console.log(token);
    return { token };
  }
  throw new UnauthorizedException('Invalid email or password');
}
```

Фиг. 3.33. CreatorUserValidation() в controller.creator.ts

Извършва се валидация на потребителските данни, като се използва методът validate на услугата creatorService, който проверява дали имейлът и паролата са валидни. Ако потребителските данни са валидни, се генерира JWT токен, който съдържа имейла на потребителя. Този токен се подписва чрез метода sign на услугата jwtService. Токенът се връща като отговор на заявката. Ако потребителските данни не са валидни, се хвърля изключение от тип “UnauthorizedException” със съобщение “Невалиден имейл или парола”.

Асинхронна функция за валидация приема два параметъра: имейл и парола, и връща булева стойност

```
async validate(email: string, password: string): Promise<boolean> {
  const creator = await this.findByEmail(email);
  if (creator && password === creator.password) {
    return true;
  }
  return false;
}
```

Фиг. 3.34. validate() в service.creator.ts

Функцията извиква метода “findByEmail()” с цел да търси създалел по зададения имейл. След това функцията проверява дали е намерен създалел с такъв имейл.

```
func validateCreator(email: String, password: String, completion: @escaping (Result<String, Error>) -> Void) {
  let urlString = "http://localhost:3001/creator/validate"

  guard let url = URL(string: urlString) else {
    completion(.failureNSError(domain: "Invalid URL", code: 0, userInfo: nil))
    return
  }

  let parameters = ["email": email, "password": password]

  var request = URLRequest(url: url)
  request.httpMethod = "POST"
  request.addValue("application/json", forHTTPHeaderField: "Content-Type")
```

Фиг. 3.35 validateCreator() в .CreatorModel

Дали паролата, подадена като аргумент, съвпада с паролата на намерения създалел също бива проверено. Ако създалел е намерен и паролата съвпада с паролата на създалеля, функцията връща true, което означава, че входът на потребителя е валиден.

```

do {
    request.httpBody = try JSONSerialization.data(withJSONObject: parameters, options: [])
} catch {
    completion(.failure(error))
    return
}

let task = URLSession.shared.dataTask(with: request) { data, response, error in
    guard let data = data, error == nil else {
        completion(.failure(error ?? NSError(domain: "Unknown error", code: 0, userInfo: nil)))
        return
    }

    do {
        if let json = try JSONSerialization.jsonObject(with: data, options: []) as? [String: Any],
            let token = json["token"] as? String {
            completion(.success(token))
        } else {
            completion(.failure(NSError(domain: "Invalid response", code: 0, userInfo: nil)))
        }
    } catch {
        completion(.failure(error))
    }
}

task.resume()

```

Фиг. 3.36. Изпращане на заявка и обработка на отговора ѝ

В модела също имаме функция, която валидира създател по предоставен имейл и парола.

В тялото на функцията се създава URL адрес, към който се изпраща заявката за валидация. След това се създава заявка от тип POST, която включва имейла и паролата на създателя като JSON данни в тялото на заявката.

Заявката се изпраща чрез URLSession.shared.dataTask, който асинхронно изпълнява заявката и получава отговор от сървъра. Ако операцията е успешна и се получи отговор от сървъра, данните се декодират от JSON формат и се извиква затвореният блок completion с резултат от тип **success**, който съдържа токен за валидацията. Ако възникне грешка при изпращането на заявката или при получаването на отговор от сървъра, се извиква затвореният блок completion с резултат от тип **failure**, който съдържа информация за грешката.

Функцията, която извлича информация за създател по предоставен имейл адрес е” fetchCreatorByEmail()”.

```
func fetchCreatorByEmail(email: String, completion: @escaping (Result<CreatorDTORes, Error>) -> Void) {
    guard let url = URL(string: "http://localhost:3001/creator/email/\" + email)") else {
        completion(.failure(NSError(domain: "", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid URL"])))
        return
    }

    URLSession.shared.dataTask(with: url) { data, response, error in
        DispatchQueue.main.async {
            if let error = error {
                completion(.failure(error))
                return
            }

            guard let data = data else {
                completion(.failure(NSError(domain: "", code: 0, userInfo: [NSLocalizedDescriptionKey: "No data"])))
                return
            }
        }
    }
}
```

Фиг. 3.37. *fetchCreatorByEmail()* в *CreatorModel*

В тялото на функцията се проверява валидността на URL адреса, към който ще се изпрати заявката за извлечане на данни за създателя. След това се създава заявка от тип GET към указанния URL адрес чрез URLSession.shared.dataTask. Тази заявка се изпраща асинхронно и се очаква отговор от сървъра. Когато се получи отговор от сървъра или ако възникне грешка при изпращането на заявката, се извиква затвореният блок completion, който съдържа резултата от операцията. Ако операцията е успешна, данните се декодират от JSON формат и се извиква затвореният блок completion с резултат от тип **success**, който съдържа информация за създателя. Ако възникне грешка при декодирането на данните или при извършването на заявката, се извиква затвореният блок completion с резултат от тип **failure**, който съдържа информация за грешката.

3.2 Създаване на приключенско събитие

```
  @Post(':id/adventures')
  @UseGuards(AuthenticationGuard)
async createAdventure(
  @Param('id') id: number,
  @Body() adventureDto: AdventureDto,
): Promise<Adventure> {
  return this.creatorService.createAdventure(id, adventureDto);
}
```

Фиг. 3.38. *createAdventure()* в controller.adventure.ts

Събития могат да бъдат създавани само, чрез притежание на профил от тип създател. Заявката трябва да бъде от тип POST и трябва да включва идентификационен номер на създателя. Също така, се използва AuthenticationGuard за да се уверим, че потребителят е аутентициран преди да бъде създадено приключение.

Функцията “createAdventure()“ приема идентификационен номер на създателя и обект от тип AdventureDto, който съдържа информация за новото приключение. Тя извиква метода “createAdventure()” на услугата creatorService, предавайки му идентификационния номер на създателя и обекта AdventureDto. Резултатът от този метод е обещание, което съдържа информация за създаденото приключение.

Функцията връща това обещание. Той е асинхронна функция, която създава ново приключение за даден създател.

```

async createAdventure(creatorId: number, adventureDto: AdventureDto): Promise<Adventure> {
  const creator = await this.getSingleCreator(creatorId);

  if (!creator) {
    throw new NotFoundException('Creator not found');
  }

  const newAdventure = this.adventuresRepository.create({
    ...adventureDto,
    creator: creator,
  });

  return await this.adventuresRepository.save(newAdventure);
}

```

Фиг. 3.39. createAdventure() в service.adventure.ts

За да се намери, кой е сегашният създател, който да бъде записан и като създател на събитието използваме “getSinglecreator()”, която намира и връща създател по подаденият й идентификатор от същия сървиз. След това се създава нов обект adventure чрез метода create на adventuresRepository. Параметрите на новото приключение са съдържанието на adventureDto и създателят, който е намерен предварително. Накрая, новото приключение се записва в базата данни чрез метода save на adventuresRepository, след което функцията връща обещание (Promise), което съдържа информация за създаденото приключение.

В модела, функцията createAdventure приема име, описание и URL на снимката на приключение като параметри. Създава обект от тип "AdventureDto", който съдържа тези данни и ги кодира в JSON формат. След това функцията изпраща POST заявка към сървъра, като включва JSON данните за новото приключение в тялото на заявката. Заявката е отправена към URL адреса на сървъра, специфициран в кода. Функцията включва проверка с guard, която се уверява, че JSON кодирането на "adventureDto" е успешно. В противен случай, тя отпечатва съобщение за грешка и излиза от функцията.

След изпращане на заявката, функцията очаква отговор от сървъра. Ако отговорът е успешен (с HTTP статус код между 200 и 299), функцията отпечатва съобщение, че приложението е създадено успешно. Ако има грешка в отговора, функцията отпечатва съответния HTTP статус код на грешката.

```

func createAdventure(name: String, description: String, url: String) {
    let adventureDto = AdventureDto(name: name, description: description, url: url)
    guard let jsonData = try? JSONEncoder().encode(adventureDto) else {
        print("Failed to encode adventureDto")
        return
    }

    let url = URL(string: "http://localhost:3001/creator/\\(currentCreator.id)/adventures")!
    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    request.setValue("Bearer \\(jwtToken)", forHTTPHeaderField: "Authorization")
    request.httpBody = jsonData

    URLSession.shared.dataTask(with: request) { data, response, error in
        guard let data = data, error == nil else {
            print("Error: \(error?.localizedDescription ?? "Unknown error")")
            return
        }

        if let httpResponse = response as? HTTPURLResponse {
            if (200...299).contains(httpResponse.statusCode) {
                print("Adventure created successfully")
            } else {
                print("Error response: \(httpResponse.statusCode)")
            }
        }
    }.resume()
}

```

Фиг. 3.40. createAdventure() в AdventureFetcher

3.3 Възможност за свързване между потребителите на платформата

На *Фиг.3.40.* е методът за свързване на два приключенци в системата. Заявката трябва да бъде от тип POST и да включва идентификационни номера на двама приключенци в URL адреса. Функцията “connectAdventurers()” приема два параметъра: adventurerId1 и adventurerId2, които са идентификационни номера на приключенците, които трябва да бъдат свързани. Преди това обаче се използва AuthenticationGuard, за да се провери дали потребителят е аутентициран. След това се извиква методът connectAdventurers на услугата adventurerService, като се предават идентификационните номера на двамата приключенци. Накрая, функцията връща обещание (Promise), което не връща стойност (void), само се изчаква за завършването на операцията.

```
@Post('/connect/:adventurerId1/with/:adventurerId2')
@UseGuards(AuthenticationGuard)

async connectAdventurers(
  @Param('adventurerId1') adventurerId1: number,
  @Param('adventurerId2') adventurerId2: number,
): Promise<void> {
  await this.adventurerService.connectAdventurers(adventurerId1, adventurerId2);
  return;
}
```

Фиг. 3.41. connectAdventurers() в AdventurerModel

Чрез connectAdventurers() първо се търси приключенец с идентификационен номер adventurerId1, като извиква метода getSingleAdventurer(). Подобно, се търси и приключенец с идентификационен номер adventurerId2. След това се проверява дали свързването е възможно, като се проверява дали adventurer1 вече е свързан с adventurer2. Ако adventurer1 не е свързан с adventurer2, те се свързват, като се добавя adventurerId2 в списъка със свързаните приключенци на adventurer1 и adventurerId1 в списъка със свързаните приключенци на adventurer2. След това обектите се записват в базата данни чрез метода save на adventurersRepository. Ако adventurer1 вече е свързан с adventurer2, операцията не се извършва повторно. Ако adventurer1 няма други свързани приключенци, adventurerId1 се добавя като свързан приключенец към adventurer1 и обектът се записва в базата данни. Същото се прави и за adventurer2.

```

    ✓  async connectAdventurers(adventurerId1: number, adventurerId2: number): Promise<void> {
        const adventurer1 = await this.getSingleAdventurer(adventurerId1);
        const adventurer2 = await this.getSingleAdventurer(adventurerId2);
        if(adventurer1.connectedAdventurers.length > 0) {
            if (!adventurer1.connectedAdventurers.includes(adventurerId2)) {
                adventurer1.connectedAdventurers.push(adventurerId2);
                await this.adventurersRepository.save(adventurer1);
            }
        }
        if (!adventurer2.connectedAdventurers.includes(adventurerId1)) {
            adventurer2.connectedAdventurers.push(adventurerId1);
            await this.adventurersRepository.save(adventurer2);
        }
    }
}

```

Фиг. 3.42. connectAdventurers() в service.adventurer.ts

В модела метода, чрез който свързваме приключенците приема два идентификатора и ги свързва посредством изпращане на POST заявка. Параметърът token представлява JWT токен, който се използва за удостоверяване на потребителя. Първо се създава URL адреса за изпращане на заявката, като се включват идентификационните номера на приключенците в URL адреса. След това се създава заявка от тип POST към този URL адрес и се добавя токенът като Authorization заглавка. След това се изпраща заявката чрез метода dataTask на URLSession.shared, като се подава затворен блок, който се изпълнява, когато заявката бъде изпратена и се получи отговор от сървъра. Ако възникне грешка при изпращането на заявката, се извежда съобщение за грешката. Ако отговорът от сървъра е със статус код 200, това означава, че свързването на приключенците е успешно. В противен случай, се извежда съобщение, че свързването е неуспешно, заедно със статус кода на отговора.

```

func connectAdventurers(adventurerId1: Int, adventurerId2: Int, token: String) {
    let urlString = "http://localhost:3001/adventurer/connect/\\(adventurerId1)/with/\\(adventurerId2)"
    guard let url = URL(string: urlString) else {
        print("Invalid URL")
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("Bearer \(token)", forHTTPHeaderField: "Authorization")

    let task = URLSession.shared.dataTask(with: request) { data, response, error in
        if let error = error {
            print("Error: \(error)")
            return
        }

        guard let httpResponse = response as? HTTPURLResponse else {
            print("Invalid response")
            return
        }

        if httpResponse.statusCode == 200 {
            print("Adventurers connected successfully")
        } else {
            print("Failed to connect adventurers. Status code: \(httpResponse.statusCode)")
        }
    }

    task.resume()
}

```

Фиг. 3.43. connectAdventurers() в AdventurerModel

3.4 Възможност за търсене на събития по име и описание

```

@Get('search/:name')
@UseGuards(AuthenticationGuard)
async searchAdventureByName(@Param('name') name: string): Promise<Adventure[]> {
    return this.adventureService.searchByName(name);
}

@Get('search/description/:keyword')
@UseGuards(AuthenticationGuard)
async searchAdventureByDescription(@Param('keyword') keyword: string): Promise<Adventure[]> {
    return this.adventureService.searchByDescription(keyword);
}

```

Фиг. 3.44. searchAdventureByName() и searchByDescription() в service.adventure.ts

Тези две функции са входните точки за търсене на приключения по име и описание. Заявките трябва да бъдат GET заявки към съответните URL адреси.

Функцията "searchAdventureByName" приема име на приключение като параметър и връща обещание (Promise), което съдържа масив от приключения,

намерени по това име. Тази функция използва услугата "adventureService" за извлечане на приключения по име.

Функцията "searchAdventureByDescription" приема ключова дума за описание на приключение и връща обещание (Promise), което съдържа масив от приключения, съдържащи тази ключова дума в описанието си. Тази функция също използва услугата "adventureService" за извлечане на приключения по описание.

```
async searchByName(name: string): Promise<Adventure[]> {
  const adventures = await this.adventuresRepository.find({
    where: {
      name: ILike(`%${name}%`)
    }
  });

  if (!adventures || adventures.length === 0) {
    throw new NotFoundException(`No adventures found with the name containing "${name}"`);
  }

  return adventures;
}

async searchByDescription(keyword: string): Promise<Adventure[]> {
  const adventures = await this.adventuresRepository.find({
    where: {
      description: ILike(`%${keyword}%`)
    }
  });

  if (!adventures || adventures.length === 0) {
    throw new NotFoundException(`No adventures found with descriptions containing "${keyword}"`);
  }

  return adventures;
}
```

Фиг. 3.45. `searchAdventureByName()` и `searchByDescription()` в `service.adventure.ts`

И двата маршрута изискват, че потребителят да е аутентициран, което се контролира чрез "AuthenticationGuard".

В модела, "searchAdventureByName" търси приключения по част от име. Създава се URL към сървъра, който предоставя API за търсене на приключения по име. Създава се HTTP заявка към този URL с метод GET. Тази заявка ще изпрати заявка към сървъра, която иска да получи данни.

Добавят се необходимите заглавия към заявката, включително този за авторизация.

Изпраща се заявката към сървъра чрез "URLSession.shared.dataTask". Това е асинхронна операция, която изчаква отговор от сървъра. При получаване на отговор, функцията проверява за грешки, след което проверява дали отговорът е успешен. При успешен отговор и наличие на данни, функцията декодира JSON данните в масив от обекти от тип "AdventureResponseDto". Декодираният приключението се присвояват на полето "adventures", вероятно в контекста на някакъв обект или модел.

```
func searchAdventureByName(name: String) {
    guard let url = URL(string: "http://localhost:3001/adventure/search/\(name)")
        else {
            print("Invalid URL")
            return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.addValue("Bearer \(jwtToken)", forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { data, response, error in
        if let error = error {
            print("Error: \(error.localizedDescription)")
            return
        }

        guard let httpResponse = response as? HTTPURLResponse,
              (200...299).contains(httpResponse.statusCode) else {
            return
        }

        if let data = data {
            do {
                let decodedAdventures = try
                    JSONDecoder().decode([AdventureResponseDto].self, from: data)
                DispatchQueue.main.async {
                    self.adventures = decodedAdventures
                }
            } catch {
                print("Error decoding JSON: \(error.localizedDescription)")
            }
        }
    }.resume()
}
```

Фиг. 3.46. `searchAdventureByName()` в `AdventureFetcher`

Функцията за търсене по ключови думи в описанието е идентична, но с друг URL адрес.

```

func searchAdventureByDescription(desc: String) {
    guard let url = URL(string:
        "http://localhost:3001/adventure/search/description/\\" + desc + "\"") else {
        print("Invalid URL")
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.addValue("Bearer " + jwtToken, forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { data, response, error in
        if let error = error {
            print("Error: \(error.localizedDescription)")
            return
        }

        guard let httpResponse = response as? HTTPURLResponse,
              (200...299).contains(httpResponse.statusCode) else {
            return
        }

        if let data = data {
            do {
                let decodedAdventures = try
                    JSONDecoder().decode([AdventureResponseDto].self, from: data)
                DispatchQueue.main.async {
                    self.adventures = decodedAdventures
                }
            } catch {
                print("Error decoding JSON: \(error.localizedDescription)")
            }
        }
    }.resume()
}

```

Фиг. 3.47. searchByDescription() в AdventureFetcher

3.5 Възможност за добавяне на събития към списъка с любими събития

Функцията addToWishlist() извършва заявка към сървъра за добавяне на приключение към списъка с желания на приключението.

```

@Post(':adventurerId/add-to-wishlist/:adventureId')
@UseGuards(AuthenticationGuard)

async addToWishlist(
  @Param('adventurerId') adventurerId: number,
  @Param('adventureId') adventureId: number,
): Promise<Adventurer> {
  const adventurer = await this.adventurerService.addToWishlist(adventurerId, adventureId);

  if (!adventurer) {
    throw new NotFoundException(`Adventurer with ID ${adventurerId} not found`);
  }

  return adventurer;
}

```

Фиг. 3.48. addToWishlist() в controller/adventurer.ts

След като се извлекат идентификационните номера, функцията извика 'adventurerService.addToWishlist()', за да извърши добавянето на приключението към списъка с желания на приключенца. Съответното приключение се добавя към списъка с желания на приключенца, като този процес се извършва във услугата adventurerService. Тази функция приема два параметъра - идентификационни номера на приключенца и на приключението, което трябва да бъде добавено към списъка с желания. След това функцията извлича информация за приключенца с помощта на "getSingleAdventurer(adventurerId)" и за приключението с "getSingleAdventure(adventureId)", като използва съответните услуги. След това функцията проверява дали приключението е намерено. Ако приключението не е намерено, се хвърля грешка със статус код 404, указваща, че приключението не е намерено. След това функцията проверява дали приключенецът вече има добавено това приключение в списъка с желания. Ако не е добавено, тогава приключението се добавя към списъка с желания на приключенца. След добавянето на приключението към списъка с желания на приключенца, функцията извика "addAdventurerToAdventure(adventurer.id, adventure.id)" от услугата за приключения, като това ще осигури свързването на приключението с приключенца. Накрая, обновеният приключенец със своя актуализиран списък

с желания се записва обратно в базата данни чрез "adventurersRepository.save(adventurer)" и се връща като резултат от функцията.

```
async addToWishlist(adventurerId: number, adventureId: number): Promise<Adventurer> {
  const adventurer = await this.getSingleAdventurer(adventurerId);
  const adventure = await this.adventureService.getSingleAdventure(adventureId);

  if (!adventure) {
    throw new NotFoundException('Adventure not found');
  }
  adventurer.wishlistAdventureIds = adventurer.wishlistAdventureIds || [];
  if (!adventurer.wishlistAdventureIds.includes(adventureId)) {
    adventurer.wishlistAdventureIds.push(adventure.id);
    this.adventureService.addAdventurerToAdventure(adventurer.id, adventure.id)
  }

  return await this.adventurersRepository.save(adventurer);
}
```

Фиг. 3.49. *addToWishlist()* в *service.adventurer.ts*

В модела тази функция (Фиг. 3.50) служи за изпращане на заявка към сървъра с цел добавяне на конкретно приключение към списъка с желания на даден приключенец. Заявката се изпраща чрез POST метод, който включва идентификационните номера на приключенца и на приключението. Първоначално функцията генерира URL адреса, към който ще се изпрати заявката, като включва идентификационните номера на приключенца и приключението в пътя на URL адреса. След това се създава "URLRequest" обект с посочения URL адрес и се задава методът на заявката - POST. Към заявката се добавя заглавка за авторизация, което включва JWT токена на приключенца. След подготовката на заявката, тя се изпраща към сървъра чрез URLSession.shared.dataTask, като се очаква отговор от сървъра. При получаване на отговор, функцията проверява дали има получени данни и дали няма грешки по време на изпълнението на заявката. Ако има грешка, тя се извежда в конзолата. Ако заявката е успешно изпратена и има успешен отговор от сървъра, функцията извежда JSON отговора в конзолата, което може да включва съобщение за успешно добавяне на приключението към списъка с желания.

```

func addToWishlist(adventurerId: Int, adventureId: Int) {
    guard let url = URL(string: "http://localhost:3001/adventurer/\\(adventurerId)/add-to-wishlist/\\(adventureId)") else {
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"

    request.addValue("Bearer \(jwtToken)", forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { (data, response, error) in
        guard let data = data, error == nil else {
            print("Error: \(error?.localizedDescription ?? "Unknown error")")
            return
        }

        if let jsonString = String(data: data, encoding: .utf8) {
            print("JSON response: \(jsonString)")
            print("Successful")
        }
    }.resume()
}

```

Фиг. 3.50. addToWishlist() в AdventurerModel

3.6 Изобразяване на компонентите

3.6.1 Функция за извлечане на данни за приключения от сървъра

Тази функция има за цел да извлече данни за приключения от сървъра, като използва GET заявка към специфичен URL адрес. В началото на функцията се създава обект от тип URLRequest, който представлява заявката, като се задават методът, Content-Type заглавието за посочване на формата на данните, както и Authorization заглавката, която съдържа JWT токена за авторизация на потребителя. След това се изпраща заявката чрез URLSession.shared.dataTask, която при получен отговор обработва резултата. Ако отговорът е успешен и съдържа статус код 200, това означава, че заявката е изпълнена успешно. В такъв случай се опитва декодирането на получените данни като масив от обекти от тип Adventure и след това тези данни се присвояват на свойството adventures. В случай, че декодирането не успее, се извежда съобщение за грешка. Ако отговорът не е успешен, т.е. статус кодът не е 200, се извежда съобщение със съответния HTTP код.

```

func fetchData() {
    guard let url = URL(string: "http://localhost:3001/adventure") else {
        print("Invalid URL")
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    request.setValue("Bearer \(jwtToken)", forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { data, response, error in
        guard let data = data else {
            print("No data received: \(error?.localizedDescription ?? "Unknown error")")
            return
        }

        if let httpResponse = response as? HTTPURLResponse {
            if httpResponse.statusCode == 200 {
                if let decodedData = try? JSONDecoder().decode([Adventure].self, from: data) {
                    DispatchQueue.main.async {
                        self.adventures = decodedData
                    }
                } else {
                    print("Failed to decode data")
                }
            } else {
                print("HTTP status code: \(httpResponse.statusCode)")
            }
        }
    }.resume()
}

```

Фиг. 3.51. fetchData() в AdventureFetcher

3.6.2. Функция за извличане на данни за списъка с любими приключенски събития от сървъра

Създава се URL адрес чрез конкатенация на базовия URL с ID на текущия потребител. Това е URL адресът, към който ще бъде изпратена заявката за получаване на списъка с желания. След това се създава обект от тип URLRequest с указания URL адрес. Задава се HTTP методът на заявката да бъде GET. Добавят се заглавия в HTTP заявката, които посочват, че се използва JWT токен за авторизация. Този токен се взима от променливата "jwtToken", която е предварително настроена за автентикация на потребителя. Създава се и изпраща заявка към сървъра с помощта на метода "dataTask(with:)" на обекта на

URLSession. Тази заявка се изпълнява асинхронно. В заявката се използва затварящата фигурна скоба с параметрите "(data, response, error)", които се връщат от сървъра. Ако има грешка или неуспешен отговор, се извежда съобщение за грешка в конзолата. Ако получените данни са валидни, се опитва да се декодират от JSON формат към масив от обекти от тип "WishlistItem". Ако декодирането е успешно, списъкът с желания се присвоява на свойството "wishlistAdventures" след като се осъществи промяна на главния (UI) поток чрез "DispatchQueue.main.async". Ако декодирането не успее, се извежда съобщение за грешка в конзолата.

```

52     private func fetchWishlistData(){
53         guard let url = URL(string: "http://localhost:3001/adventurer/\" + currentAdventurer.id + "/wishlist")
54             else {
55                 return
56             }
57
58         var request = URLRequest(url: url)
59         request.httpMethod = "GET"
60
61         request.addValue("Bearer " + jwtToken, forHTTPHeaderField: "Authorization")
62
63         URLSession.shared.dataTask(with: request) { (data, response, error) in
64             guard let data = data, error == nil else {
65                 print("Error: \(error?.localizedDescription ?? "Unknown error")")
66                 return
67             }
68
69             if let jsonString = String(data: data, encoding: .utf8) {
70                 print("JSON response: \(jsonString)")
71             }
72
73             do {
74                 let result = try JSONDecoder().decode([WishlistItem].self, from: data)
75                 DispatchQueue.main.async {
76                     self.wishlistAdventures = result
77                 }
78             } catch {
79                 print("Error decoding JSON: \(error)")
80             }
81         }.resume()
82     }

```

Фиг. 3.52. `fetchWishlistData()` в `AdventurerModel`

3.6.3.Функция за извличане на данни за приключениците

Създава се URL адрес към сървъра, където се намират данните за приключениците. Създава се обект от тип URLRequest, който съдържа URL адреса и представлява заявката към сървъра. В заявката се добавя заглавка "Authorization", което съдържа JWT токена за удостоверяване на идентичността на потребителя. Създава се и се изпраща заявка към сървъра чрез метода "dataTask(with:)" на обекта на URLSession. Тази заявка е асинхронна, т.е. изпълнява се във фонов режим. В заявката се обработва отговорът от сървъра. Ако има грешка при извличането на данните или отговорът е невалиден, се извежда съобщение за грешка. Ако данните са валидни, те се декодират от JSON формат към масив от обекти от тип "Adventurer". След това тези данни се присвояват на свойството adventurers и се осъществява промяна на главния (UI) поток, за да се актуализира интерфейсът с новите данни. Ако декодирането не е успешно, се извежда съобщение за грешка.

```
func fetchAdventurerData() {
    guard let url = URL(string: "http://localhost:3001/adventurer") else {
        return
    }

    var request = URLRequest(url: url)
    request.addValue("Bearer \(jwtToken)", forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { (data, response, error) in
        guard let data = data, error == nil else {
            print("Error: \(error?.localizedDescription ?? "Unknown error")")
            return
        }

        if let jsonString = String(data: data, encoding: .utf8) {
            print("JSON response: \(jsonString)")
        }

        do {
            let result = try JSONDecoder().decode([Adventurer].self, from: data)
            DispatchQueue.main.async {
                self.adventurers = result
            }
        } catch {
            print("Error decoding JSON: \(error)")
        }
    }.resume()
}
```

Фиг. 3.53. *fetchAdventurerData()* в *AdventurerViewModel*

3.6.4.Функция за извличане на данни за приключениците, с които текущият потребител е свързан.

Този код извършва заявка към сървъра за извличане на данни за свързаните приключеници на даден приключенец, идентифициран по неговия идентификационен номер. Първо създава се URL адрес, който включва идентификационния номер на приключенец в сървъра, за когото ще бъдат извлечени свързаните приключеници. След това се създава обект от тип URLRequest със съответния URL адрес и определя се HTTP методът на заявката. В заявката се добавя заглавка "Authorization", която съдържа JWT токена за удостоверяване на идентичността на потребителя. Изпраща се заявка към сървъра чрез метода "dataTask(with:)" на обекта на URLSession. Тази заявка е асинхронна и ще изчака отговора на сървъра. В заявката се обработва отговорът от сървъра. Ако има грешка при извличането на данните или отговорът е

```
func fetchConnectedAdventurers(id: Int) {
    let urlString = "http://localhost:3001/adventurer/\(id)/connected-adventurers"
    guard let url = URL(string: urlString) else {
        print("Invalid URL")
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.setValue("Bearer \(jwtToken)", forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { data, response, error in
        if let error = error {
            print("Error: \(error)")
            return
        }

        guard let httpResponse = response as? HTTPURLResponse,
              (200...299).contains(httpResponse.statusCode) else {
            print("Invalid response")
            return
        }

        if let data = data {
            do {
                adventurers = try JSONDecoder().decode([AdventurerDtoRes].self, from: data)
            } catch {
                print("Error decoding JSON: \(error)")
            }
        }
    }.resume()
}
```

Фиг. 3.54. `fetchConnectedAdventurers()` в `AdventurerViewModel`

невалиден HTTP статус код, се извежда съобщение за грешка. Ако данните са валидни, те се декодират от JSON формат към масив от обекти от тип "AdventurerDtoRes". Ако декодирането не е успешно, се извежда съобщение за грешка.

3.6.5.Функция за извлечане на данни за приключенията в които даден приключенец е участвал.

Функцията е почти идентична с предходните за извлечане, различава се по URL адреса, към който се прави заявката.

```
func getAttendedAdventures(adventurerId: Int) {
    let urlString = "http://localhost:3001/adventurer/\(adventurerId)/attended-adventures"

    guard let url = URL(string: urlString) else {
        print("Invalid URL")
        return
    }
```

Фиг. 3.55. Начална част на getAttendedAdventures() в AdventurerViewModel

3.7 Навигационна лента

Обектът, показан на *Фиг. 3.55.* е перечисление в Swift, което съдържа определени стойности за различните раздели в приложението. В този случай те са "Explore", "WishList", "Add" и "Profile". Всеки от тези раздели също така е свързан с иконка. Enum има свойство tabName, което връща името на раздела според стойността на елемента. Например, ако стойността на елемента е "Add", tabName ще върне "Meet".

```
enum Tab: String, CaseIterable{
    case Explore = "house"
    case WishList = "heart"
    case Add = "figure.2"
    case Profile = "person"

    var tabName: String{
        switch self {
        case .Add:
            return "Meet"
        case .WishList:
            return "WishList"
        case .Explore:
            return "Explore"
        case .Profile:
            return "Profile"
        }
    }
}
```

Фиг. 3.56. Tab

Това е разширение на типа View, което добавя метод getSafeArea(). Този метод се използва за извлечане на информация за безопасната зона на устройството, където са разположени компоненти като статус лентата, през която може да се обработи излизането на информация. Методът връща информацията за безопасната зона като UIEdgeInsets.

Това е структура на Фиг.3.57. , която използва визуални ефекти от UIKit като част от представянето. style параметърът задава стила на размиване, който се използва. makeUIView(context:) методът създава и връща UIVisualEffectView, който прилага размиване с подадения стил. updateUIView(_:context:) методът обновява представянето на визуалния ефект при промяна в контекста на изгледа. В този случай, тъй като няма промени да се направят, този метод е празен.

```
extension View {
    func getSafeArea() -> UIEdgeInsets{
        guard let screen = UIApplication.shared.connectedScenes.first as?
            UIWindowScene else {
            return .zero
        }

        guard let SafeArea = screen.windows.first?.safeAreaInsets else {
            return .zero
        }

        return SafeArea
    }
}
```

Фиг. 3.57. Метод за правилно позициониране на навигационата лента

Функцията TabButton на *Фиг.3.58.* генерира бутон за навигационната лента, който може да бъде натиснат от потребителя, за да се превключи между различните раздели на приложението.

GeometryReader - контейнер, който се използва за измерване на геометрията на родителския изглед, за да може да се извършат операции върху него в зависимост от тази геометрия.

Button - SwiftUI компонентът за създаване на бутони, който реагира на натискане от потребителя. Когато бутона се натисне, текущият раздел се променя към този, който е асоцииран с бутона.

VStack - вертикален контейнер за подреждане на визуални елементи един след друг.

Image - изображението, което служи като икона на бутона. Иконата може да бъде пълна или празна, в зависимост от това дали текущият раздел съответства на този на бутона.

```

func TabButton(tab: Tab) -> some View {
    GeometryReader { proxy in
        Button(action: {
            withAnimation(.spring()) {
                currentTab = tab
            }
        }, label: {
            VStack(spacing: 0) {
                Image(systemName: currentTab == tab ? tab.rawValue + ".fill" : tab.rawValue)

                .resizable()
                .aspectRatio(contentMode: .fit)
                .frame(width: iconFrame, height: iconFrame)
                .frame(maxWidth: .infinity)
                .foregroundColor(currentTab == tab ? .primary : .secondary)
                .padding(currentTab == tab ? 15 : 0)
                .background(
                    ZStack {
                        if currentTab == tab {
                            MaterialEffect(style: .light)
                            .clipShape(Circle())
                            .matchedGeometryEffect(id: "TAB", in: animations)
                        }
                        Text(tab.tabName).foregroundColor(.primary).font(.footnote).padding(
                            .top, textPadding)
                    }
                )
                .contentShape(Rectangle())
                .offset(y: currentTab == tab ? -35 : 0)
            }
        })
    }
    .frame(height: 25)
}

```

Фиг. 3.58. Генериране бутона за навигация

MaterialEffect - SwiftUI view, която представлява размиващ ефект на материал, който се прилага на фона на бутона, когато той е активен (т.е. когато текущият раздел е съответстващият на бутона).

```

struct MaterialEffect: UIViewRepresentable{
    var style: UIBlurEffect.Style

    func makeUIView(context: Context) ->UIVisualEffectView {
        let view = UIVisualEffectView(effect: UIBlurEffect(style: style))

        return view
    }

    func updateUIView(_ uiView: UIVisualEffectView, context: Context) {}
}

```

Фиг. 3.59. Ефект за бутона()

Text:- текстът, който съдържа името на раздела, който бутона представлява.

contentShape - формата на бутона, която се използва за определяне на областта, която може да бъде натисната за активиране на бутона.

Frame- съдържащи се вътре View са настроени с определени размери и позиции, за да бъдат правилно показани на екрана.

Този код представлява SwiftUI view, която представя таблица за навигация. В една табличка за навигация има различни раздели на приложението.

TabView е контейнер за показване на различни раздели на приложението (или раздели). Това позволява на потребителя да превключва между различните изгледи, като използва табулация.

AdventureDisplayView показва раздела с приключения.

WishlistAdventuresDisplayView показва раздела с любими приключения.

AdventurerDisplayView показва раздела с пътешественици.

ProfileView показва профилния раздел на потребителя.

HStack е хоризонтален контейнер, в който са поставени различните бутони за раздели.

TabButton представлява бутона за навигация, който позволява на потребителя да превключва между различните раздели на приложението.

padding добавя вътрешни отстъпи на разделите.

background задава цвета на фона на разделите.

overlay предоставя възможност за добавяне на слой отгоре на съществуващото съдържание. В този случай се използва за поставяне на бутона за раздели в долната част на екрана.

ignoresSafeArea игнорира безопасната зона, която може да бъде присъща в горната или долната част на екрана, така че съдържанието да може да бъде поставено над или под тази зона.

navigationBarBackButtonHidden скрива бутона за навигация "назад" в горната част на екрана.

```

var body: some View {
    TabView(selection: $currentTab) {
        AdventureDisplayView(viewModel: viewModel)
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .background(Color("primaryColor").ignoresSafeArea())
            .tag(Tab.Explore)

        WishlistAdventuresDisplayView()
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .background(Color("primaryColor").ignoresSafeArea())
            .tag(Tab.WishList)

        AdventurerDisplayView(viewModel: AdventurerViewModel(), viewModelCon: viewModel)
        AdventurerDisplayView(viewModel: AdventurerViewModel(), viewModelCon: viewModel)
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .background(Color("primaryColor").ignoresSafeArea())
            .tag(Tab.Add)

        ProfileView(adventurer: currentAdventurer)
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .background(Color("primaryColor").ignoresSafeArea())
            .tag(Tab.Profile)
    }
    .overlay(
        HStack(spacing: 0) {
            ForEach(Tab.allCases, id: \.rawValue) { tab in
                TabButton(tab: tab)
            }
            .padding(.vertical)
            .padding(.bottom, getSafeArea().bottom == 0 ? 5 : (getSafeArea().bottom - 15))
            .background(Color("nav_bar_bg"))
        },
        alignment: .bottom
    )
    .ignoresSafeArea(.all, edges: .bottom)
    .navigationBarBackButtonHidden(true)
}

```

Фиг. 3.60. Навигация

3.8. Models

Моделите съдържат извлечените данните от сървър или локална база данни и методите за извлечането. В проекта са реализирани няколко модела - Adventurer model, Adventure model, AttendedAdventures model, Wishlist model, Creator model. Моделите за приключенец и създател, и за приключение и списък със желани събития и приключения в които приключенецът е участвал са идентични, единствената разлика е в структурата на конкретния тип, които съдържат и минимални разлики в методите. Като примери са разгледани модела за приключенец и модела за списъка от желани събития.

3.8.1. Adventurer model

Този модел се състои от две структури и шест метода. Методите се използват за конвертиране на обект Adventurer в и от JSON формат, добавяне на свързани приключенци, валидиране на потребителски данни и добавяне на нови участия в приключения. Тези структури представят данни за приключенците в приложението. Структурата съдържа информация за приключенец, като уникален идентификатор, потребителско име, имейл, парола, URL с профилна снимка, списък с идентификатори на посетените приключения, списък с идентификатори на приключения в техния списък с желания и списък с идентификатори на свързаните приключенци. Структурата Adventurer е кодируема и идентифицируема, което позволява кодирането и декодирането ѝ от и до външни представления като JSON и я прави уникално идентифицируема. AdventurerDtoRes. Структурата се използва за декодиране на отговор от сървъра, представлящ данните за приключенец. Отново включва основните атрибути, освен свързаните приключенци. Структурата е декодируема, което означава, че може да се декодира от външно представяне като JSON в Swift обекти.

```
struct Adventurer: Codable, Identifiable, Hashable {
    var id: Int
    var username: String
    var email: String
    var password: String
    var profilephoto: String
    var attendedAdventureIds: [Int]
    var wishlistAdventureIds: [Int]
    var connectedAdventurers: [Int]

}

struct AdventurerDtoRes: Decodable {
    let id: Int
    let username: String
    let email: String
    let attendedAdventureIds: [Int]
    let wishlistAdventureIds: [Int]
}
```

Фиг. 3.61. Структури за съхранение на извлечените данни

Тази функция създава нов авантюрист в системата, използвайки предоставеното потребителско име, имейл и парола. След създаването на авантюриста във външната база данни през API заявка, функцията извиква друга функция за валидиране на потребителя със същите данни.

След успешно създаване на потребителя и валидация, авантюристът се връща като резултат на завършването на операцията.

Функцията е разгледана по-подробно във втора глава, точка ...

```
mutating func createUserAdventurer(username: String, email: String, password: String, validateUser: @escaping (String, String, @escaping (Result<String, Error>) -> Void) -> Void, completion: @escaping (Result<Adventurer, Error>) -> Void) {
    let adventurerDto = AdventurerDto(username: username, email: email, password: password)
    guard let url = URL(string: "http://localhost:3001/adventurer") else {
        let error = NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid URL"])
        completion(.failure(error))
        return
    }
    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    do {
        let jsonData = try JSONEncoder().encode(adventurerDto)
        request.httpBody = jsonData
        URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                completion(.failure(error))
                return
            }
            if let data = data {
                do {
                    let adventurer = try JSONDecoder().decode(Adventurer.self, from: data)
                    validateUser(email, password) { result in
                        switch result {
                            case .success(let token):
                                jwtToken = token
                                currentAdventurer.id = adventurer.id
                                currentAdventurer.username = adventurer.username
                                currentAdventurer.email = adventurer.email
                                currentAdventurer.password = adventurer.password

                                completion(.success(adventurer))
                            case .failure(let error):
                                completion(.failure(error))
                        }
                    }
                } catch {
                    completion(.failure(error))
                }
            }
        }.resume()
    } catch {
        completion(.failure(error))
    }
}
```

Фиг. 3.62. Функция за създаване на приключенец

Тази функция валидира потребителски идентификационни данни - имейл и парола, чрез изпращане на POST заявка към външен ресурс. След успешна валидация на идентификационните данни, функцията връща JWT токен като резултат от операцията. Ако валидацията не успее или се получи невалиден токен, се предава съответната грешка чрез параметъра за завършване на заявката. Функцията е разгледана по-подробно във втора глава, точка.

```

func validateUser(email: String, password: String, completion: @escaping (Result<String, Error>) -> Void) {
    let parameters = ["email": email, "password": password]
    guard let url = URL(string: "http://localhost:3001/adventurer/validate") else {
        let error = NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid URL"])
        completion(.failure(error))
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")

    do {
        let jsonData = try JSONSerialization.data(withJSONObject: parameters, options: [])
        request.httpBody = jsonData

        URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                completion(.failure(error))
                return
            }

            if let data = data {
                do {
                    guard let token = try JSONSerialization.jsonObject(with: data, options: []) as? [String: Any],
                          let jwtToken = token["token"] as? String else {
                        let error = NSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid token response"])
                        completion(.failure(error))
                        return
                    }
                    completion(.success(jwtToken))
                } catch {
                    completion(.failure(error))
                }
            }
        }.resume()
    } catch {
        completion(.failure(error))
    }
}

```

Фиг. 3.63. validateUser() в AdventurerModel

Функция от (Фиг.3.64) извлича информация за приключенеца по зададен имейл и JWT токен, като използва HTTP GET заявка към външен ресурс. При получаване на отговор, функцията декодира получените данни за приключенеца и ги предава чрез параметъра за завършване на заявката. Ако операцията не успее, функцията предава съответната грешка. Функцията е разгледана по-подробно във втора глава, точка ...

```

func getAdventurerByEmail(email: String, token: String, completion: @escaping (Result<Adventurer, Error>) -> Void) {
    guard let url = URL(string: "http://localhost:3001/adventurer/email/\" + email)" else {
        completion(.failureNSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Invalid URL"]))
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.addValue("Bearer " + token, forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { data, response, error in
        guard let data = data, error == nil else {
            completion(.failureNSError(domain: "com.example", code: 0, userInfo: [NSLocalizedDescriptionKey: "Unknown error"]))
            return
        }

        do {
            let adventurer = try JSONDecoder().decode(Adventurer.self, from: data)
            currentAdventurer = adventurer
            print(currentAdventurer.username)
            completion(.success(adventurer))
        } catch {
            completion(.failure(error))
        }
    }.resume()
}

```

Фиг. 3.64. getAdventurerByEmail() в AdventurerModel

```

func signIn(email: String, password: String, completion: @escaping (Result<Void, Error>) -> Void) {
    validateUser(email: email, password: password) { result in
        switch result {
        case .success(let token):
            jwtToken = token
            print("Token: " + token)

            getAdventurerByEmail(email: email, token: token) { result in
                switch result {
                case .success(let adventurer):
                    currentAdventurer = adventurer
                    print("Current Adventurer: " + adventurer.username)
                    completion(.success(()))
                case .failure(let error):
                    print("Error fetching adventurer: " + error.localizedDescription)
                    completion(.failure(error))
                }
            }
        case .failure(let error):
            print("Error validating user: " + error.localizedDescription)
            completion(.failure(error))
        }
    }
}

```

Фиг. 3.65. signIn() в AdventurerModel

Тази функция от (Фиг.3.65) представлява процеса на влизане в системата чрез подаване на имейл и парола. След валидиране на потребителските данни се извиква функцията за валидиране на потребителя, която връща JWT токен. След това се извиква функцията за извлечане на информация за приключението чрез получения токен и имейл. В случай на успех се запазват токена и данните за приключението, а в противен случай се предава съответната грешка чрез параметъра за завършване на заявката.

Функцията е разгледана по-подробно във втора глава, точка ...

```

func connectAdventurers(adventurerId1: Int, adventurerId2: Int, token: String) {
    let urlString = "http://localhost:3001/adventurer/connect/\\(adventurerId1)/with/\\(adventurerId2)"
    guard let url = URL(string: urlString) else {
        print("Invalid URL")
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("Bearer \(token)", forHTTPHeaderField: "Authorization")

    let task = URLSession.shared.dataTask(with: request) { data, response, error in
        if let error = error {
            print("Error: \(error)")
            return
        }

        guard let httpResponse = response as? HTTPURLResponse else {
            print("Invalid response")
            return
        }

        if httpResponse.statusCode == 200 {
            print("Adventurers connected successfully")
        } else {
            print("Failed to connect adventurers. Status code: \(httpResponse.statusCode)")
        }
    }

    task.resume()
}

```

Фиг. 3.66. connectAdventurers() в AdventurerModel

Тази функция осъществява връзка между двама приключенци чрез изпращане на заявка към сървъра с техните идентификатори и токен за удостоверяване. Ако заявката бъде успешно обработена от сървъра и връзката бъде създадена, се отпечатва съобщение за успешно свързване. В противен случай се отпечатва съобщение за неуспешно свързване, заедно със статусния код на HTTP отговора. Функцията е разгледана по-подробно във втора глава, точка ...

Функцията на *Фиг. 3.67.* позволява на приключенец да се регистрира за участие в приключение, като изпраща заявка към сървъра с идентификаторите на приключенца и приключението. Сървърът обработва заявката и, ако е успешна, връща подходящ HTTP отговор, указващ, че регистрацията е успешна. Ако отговорът не е в подходящия диапазон на статус кодове, се извежда съобщение за невалиден отговор.

Функцията е разгледана по-подробно във втора глава, точка ...

```

func attendAdventure(adventurerId: Int, adventureId: Int) {
    let urlString = "http://localhost:3001/adventurer/(\(adventurerId))/attend/(\(adventureId))"
    guard let url = URL(string: urlString) else {
        print("Invalid URL")
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("Bearer \(jwtToken)", forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { data, response, error in
        if let error = error {
            print("Error: \(error)")
            return
        }

        guard let httpResponse = response as? HTTPURLResponse, (200...299).contains(httpResponse.statusCode) else {
            print("Invalid response")
            return
        }

        print("Adventure attended successfully!")
    }.resume()
}

```

Фиг. 3.67. attendedAdventurers() в AdventurerModel

3.8.2. WishlistAdventure model

Този модел се състои от структура и метод. На (Фиг.3.68.) е показана Структурата. Методите се използват заднавяне на ново събитие към списъка и за извлечане на събитията от базата.

Структурата WishlistItem е модел за декодиране на данни от JSON формат към обекти от тип WishlistItem. Всяка инстанция на WishlistItem има уникален идентификатор, който позволява на SwiftUI да определи елементите в списъци и да ги разпознае. Типът може да бъде декодиран от външен JSON формат. Това ни позволява да използваме типа в контекст на декодиране на JSON данни.

```

struct WishlistItem: Identifiable, Decodable {
    var id: Int
    let name: String
    let description: String

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.name = try container.decode(String.self, forKey: .name)
        self.description = try container.decode(String.self, forKey: .description)
        self.id = try container.decode(Int.self, forKey: .id)
    }

    enum CodingKeys: String, CodingKey {
        case id
        case name
        case description
    }
}

```

Фиг. 3.68. Структура на елементите в WishlistAdventureModel

Атрибутите ѝ са:

id - Уникален идентификатор на елемента.

name - Име на приключението.

description - Описание на приключението.

`init(from decoder: Decoder)` е инициализатор, който се извика при декодиране на данни от JSON формат. Той извлича стойностите на свойствата от контейнера на декодера, използвайки ключовете, дефинирани в `CodingKeys`. Това е вложен енумератор, който представлява ключовете, които съответстват на свойствата на структурата при декодиране на JSON данни. Този енумератор се използва, за да се осигури правилното съпоставяне между ключовете от JSON формата и свойствата на структурата. В този случай ключовете са `""id""`, `""name""` и `""description""`.

Методът `“addToWishlist(adventurerId: Int, adventureId: Int)”` се използва за добавяне на приключение към списъка с пожелания на даден авантюрист.

```

func addToWishlist(adventurerId: Int, adventureId: Int) {
    guard let url = URL(string: "http://localhost:3001/adventurer/\\(adventurerId)/add-to-wishlist/\\(adventureId)") else {
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"

    request.addValue("Bearer \(jwtToken)", forHTTPHeaderField: "Authorization")

    URLSession.shared.dataTask(with: request) { (data, response, error) in
        guard let data = data, error == nil else {
            print("Error: \(error?.localizedDescription ?? "Unknown error")")
            return
        }

        if let jsonString = String(data: data, encoding: .utf8) {
            print("JSON response: \(jsonString)")
            print("Successful")
        }
    }.resume()
}

```

Фиг. 3.69. Метод за добавяне на приключение към списък с любими събития в WishlistAdventureModel

Първо URL адресът се формира чрез конкатенация на базовия URL с идентификаторите на авантюриста и приключението, което ще се добави към списъка с пожеланията. След това се създава обект от тип "URLRequest", който се използва за изпращане на HTTP заявка към указанния URL адрес. Методът на заявката се задава като ""POST"", тъй като искаме да изпратим данни към сървъра. Добавя се заглавка. В заглавката на заявката се добавя информация за авторизация, по-специално токенът на JWT, който се използва за идентифициране на потребителя. Функцията "URLSession.shared.dataTask(with: request)" се извиква с "request" като параметър. Тази функция изпраща заявката към сървъра и получава отговор.

Изпълнява се код за обработка на отговора от сървъра. Ако заявката е успешна и има данни, те се прочитат и се извежда JSON отговорът. Ако има грешка при изпращането на заявката или получаването на отговор от сървъра, се извежда съобщение за грешка.

3.9. ViewModels

ViewModel-ът действа като посредник между View и Model. Той предоставя данните от Model на потребителския интерфейс. ViewModel-ът може да се разглежда като източник, към когото потребителските интерфейси се обръщат както за данни, така и за действия. Всички ViewModel-и в проекта са структурирани по сходен начин. Като пример за ViewModel е разгledан класът AdventureFetcher.

```
class AdventureFetcher: ObservableObject {
    @Published var adventures: [Adventure] = []

    func fetchData() {
        guard let url = URL(string: "http://localhost:3001/adventure") else {
            print("Invalid URL")
            return
        }

        var request = URLRequest(url: url)
        request.httpMethod = "GET"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.setValue("Bearer \jwtToken", forHTTPHeaderField: "Authorization")

        URLSession.shared.dataTask(with: request) { data, response, error in
            guard let data = data else {
                print("No data received: \(error?.localizedDescription ?? "Unknown error")")
                return
            }

            if let httpResponse = response as? HTTPURLResponse {
                if httpResponse.statusCode == 200 {
                    if let decodedData = try? JSONDecoder().decode([Adventure].self, from: data) {
                        DispatchQueue.main.async {
                            self.adventures = decodedData
                        }
                    } else {
                        print("Failed to decode data")
                    }
                } else {
                    print("HTTP status code: \(httpResponse.statusCode)")
                }
            }
        }.resume()
    }
}
```

Фиг. 3.70. Adventure Fetcher View Model

Целта на този ViewModel клас е да извлича и показва информация за създадените приключения. Методът fetchData() в AdventureFetcher клас служи за извличане на данни за приключения от сървъра.

Функцията е разгледана по-подробно във втора глава, точка ...

3.10 Views

В архитектурата MVVM (Model-View-ViewModel) View (изглед) представлява потребителския интерфейс (UI) на приложението. Той отговаря за показването на данните на потребителя и обработката на въведените от него данни. Изгледът трябва да бъде напълно отделен от основните данни и бизнес логиката на приложението. Вместо директен достъп и манипулиране на данните, изгледът взаимодейства с ViewModel-а, който действа като посредник между изгледа и данните. Подобно на ViewModel-ите, изгледите в проекта за изобразяване на събития и приключения са структурирани по подобен начин. Пример за изглед е структурата AdventureDisplayView, за подробен изглед е DetailedAdventureView, а за компонент AdventureView.

3.10.1. AdventureDisplayView

Екранът “Adventures“ позволява на потребителите да разгледат приключенията, които са активни в платформата и ако пожелаят да вземат участие. Екранът показва информация за приключенията като име, снимка, описание и създател.

```
3 struct AdventureDisplayView: View {
4
5     @ObservedObject var adventureFetcher = AdventureFetcher()
6     @ObservedObject var viewModel: Connection
7     var columns: [GridItem] = Array(repeating: .init(.flexible()), count: 1)
8
9     var body: some View {
10         NavigationView {
11             ScrollView {
12                 LazyVGrid(columns: columns, spacing: 20) {
13                     ForEach(adventureFetcher.adventures) { adventure in
14                         NavigationLink(destination: DetailedAdventureView(adventure: adventure,
15                             viewModel: viewModel, viewModelAdv: AttendedAdventuresVModel())) {
16                             AdventureView(title: adventure.name, adventurePhoto: adventure.urlPhoto,
17                                 profilePhoto: adventure.creatorName.url)
18                         }
19                     }
20                 }
21             }
22         }
23     }
24     .padding()
25     .onAppear {
26         adventureFetcher.fetchData()
27     }
28     .navigationBarTitle("Adventures")
29 }
30 }
```

Фиг.3.71. Изглед на приключенията

Горепоказаната *Фиг.3.71* е код, който дефинира SwiftUI изглед, наречен "AdventureDisplayView". Той отговаря за показването на списък с приключения, извлечени от източник на данни ("AdventureFetcher"). Изгледът е организиран с помощта на "LazyVGrid", който подрежда изгледите му във вертикална мрежа, оптимизирана за мързеливо зареждане. Всяко приключение е представено от "AdventureView" в рамките на "NavigationLink", което позволява на потребителите да преминат към подробен изглед на всяко приключение. В "AdventureView" се показва основна информация за приключението, като например неговото заглавие и свързаните с него снимки.

Изгледът също така има "NavigationView" и "NavigationBarTitle", което показва, че е предназначен да бъде част от навигационен стек и ще показва заглавие в навигационната лента.

Обивките на свойствата "@ObservedObject" се използват за наблюдение на промените в обектите "adventureFetcher" и "viewModel", като задействат актуализации на потребителския интерфейс при промяна на тези обекти. Обектът "adventureFetcher" извлича данни асинхронно, вероятно от отдалечен източник, и актуализира потребителския интерфейс, когато данните са налични.

3.10.2 DetailedAdventureView

```
struct DetailedAdventureView: View {
    var adventure: Adventure
    @ObservedObject var viewModel: Connection
    @ObservedObject var viewModelAdv: AttendedAdventuresVModel

    @State private var isAddedToWishlist = false
    var body: some View {
        VStack {
```

Фиг.3.72. Детайлен изглед на приложението

DetailedAdventureView", който представлява подробен изглед на конкретно приключение съдържа:

- Изглед на изображение, показващ основната снимка на приключението.

- Изглед за хоризонтално превъртане, показващ допълнителни снимки, свързани с приключението.
- Информация за създателя на приключението, включително профилното му изображение и потребителско име.
- Бутони за добавяне на приключението в списъка с желания и за присъединяване към приключението.
- Текстов изглед, показващ описанието на приключението, с максимум три реда.
- Разделител за целите на оформлението.

"@ObservedObject" се използва за наблюдение на промените в обектите "viewModel" и "viewModelAdv", което показва, че тези обекти отговарят за управлението на данните и съответно за актуализирането на потребителския интерфейс.

```
var body: some View {
    VStack {
        ImageView(urlString: adventure.urlPhoto)
            .aspectRatio(contentMode: .fit)
        ScrollView(.horizontal, showsIndicators: false) {
            HStack {
                ForEach(0..<5) { _ in
                    ImageView(urlString: adventure.urlPhoto)
                        .frame(width: 80, height: 80)
                        .cornerRadius(10)
                }
            }
        }
        .padding()
        VStack(alignment: .leading) {
            HStack {
                ImageView(urlString: adventure.creatorName.url)
                    .clipShape(Circle())
                    .frame(width: 50, height: 50)
                Text(adventure.creatorName.username).font(.headline)
                Spacer()
                Button(action: {
                    viewModel.addToWishlist(adventurerId: currentAdventurer.id, adventureId: adventure.id)
                    viewModel.fetchWishlistData()
                }) {
                    Image(systemName: isAddedToWishlist ? "heart.fill" : "heart")
                }
                .padding()
                Button(action: {
                    viewModel.attendAdventures(adventurerId: currentAdventurer.id, adventureId: adventure.id)
                    viewModelAdv.getCurrAttendedAdventures()
                }) {
                    Text("Jump in")
                }
            }
        }
        Text(adventure.description)
            .lineLimit(3)
    }
    .padding()
}
```

Фиг.3.73. Детайллен изглед на приключението

"@State" се използва за "isAddedToWishlist", като показва дали приключението в момента е добавено към списъка с желания, или не..

3.10.3 AdventureView

Редицата от код, която предоставихте, съдържа няколко пресметнати свойства в Swift. Всяко от тези свойства представлява изглед или модификатор на изглед.

```
7
0 struct AdventureView: View {
1     private let squareFrame:CGFloat = 320
2     let title: String
3     let adventurePhoto: String
4     let profilePhoto: String
5     var body: some View {
6         ZStack{
7             showzone
8             whiteBase
9             VStack{
10                 adventureImage
11                 //Spacer()
12                 HStack{
13                     titleText
14                     Spacer()
15                     profileImage
16
17                 }
18             }
19             .frame(width: 280, height: 280)
20             .cornerRadius(25)
21             // .shadow(radius: 10)
22             .padding(.top)
23         }
24         .shadow(radius: 5)
25     }
```

Фиг.3.74.Изглед на приключението

- showzone (Фиг.3.74) е променлива, която връща изглед от тип "DiagonalGradientView" с определен квадратен формат ("squareFrame"),

използван за създаване на градиентен фон с диагонална форма. Изгледът има закръглен ъгъл с радиус 30.

- `adventureImage` (*Фиг.3.74*) е променлива, която връща изглед от тип "Image", който показва снимката на приключението. Снимката може да се променя размера си, мащабирана, за да се побере в наличното пространство, и има максимална ширина, равна на ширината на контейнера. Също така е ограничена до закръглен ъгъл с радиус 25.
- `profileImage` (*Фиг.3.74*) променлива, която връща изглед от тип "Image", който показва профилната снимка. Снимката е `resizable`, `scaledToFill` (мащабирана, за да запълни наличното пространство), с фиксирана ширина и височина от 40 единици. Формата му е ограничена до кръгла форма.
- `titleText` (*Фиг.3.74*) е променлива, връщаща изглед от тип "Text", който показва заглавието на приключението. Текстът е стилизиран с шрифт за заглавие, смело тегло и черен цвят. Той е подравнен в началото на контейнера и е паднат хоризонтално.
- `whiteBase` (*Фиг.3.74*) е променлива, която връща изглед от тип "Rectangle", запълнен с бял цвят, образуващ бяла база за други изгледи.

```
var showzone: some View {
    DiagonalGradientView(squareFrame: squareFrame).cornerRadius(30)
}

var adventureImage: some View {
    Image(adventurePhoto)
        .resizable()
        .scaledToFit()
        .frame(maxWidth: .infinity)
        .cornerRadius(25)
        .clipped()
}

var profileImage: some View {
    Image(profilePhoto)
        .resizable()
        .scaledToFill()
        .frame(width: 40, height: 40)
        .clipShape(Circle())
}

var titleText: some View {
    Text(title)
        .font(.title)
        .fontWeight(.bold)
        .foregroundColor(.black)
        .frame(maxWidth: .infinity, alignment: .leading)
        .padding(.horizontal)
}

var whiteBase: some View {
    Rectangle()
        .fill(Color.white)
        .frame(width: 300, height: 300)
        .cornerRadius(30)
}
```

Фиг.3.75. Детайллен изглед на приложението

Четвърта глава

Ръководство на потребителя

4.1. Инсталация

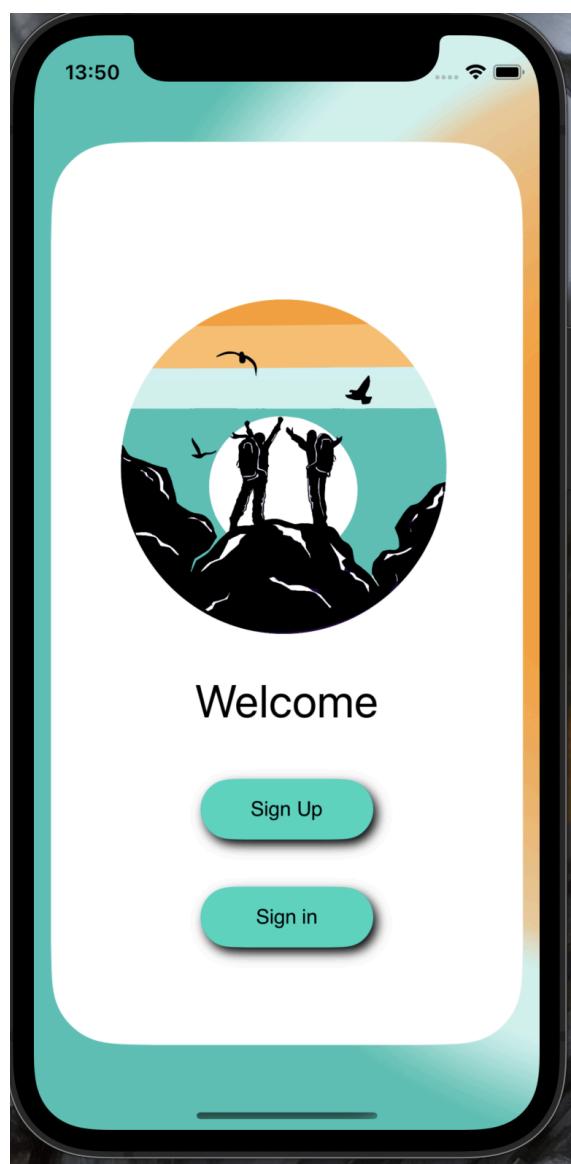
За стартиране на приложението е нужно да се изпълнят няколко предварителни стъпки:

- Инсталиране на пакетен мениджър за JavaScript - npm, npx.
- Инсталиране среда за разработка по избор, която поддържа JavaScript и TypeScript- VisualStudio Code,
- Инсталиране на Nest.js и неговите зависимости за съответната операционна система.
- Стартиране на Nest сървъра чрез избрания мениджър.
- Инсталиране среда за разработка по избор, която поддържа езика Swift - XCode, VisualStudio Code
- Стартиране на приложението.
- Кодът на проекта се намира на този линк -
<https://github.com/iv4pa4/HiJourney-1.2>

4.2. Въведение през еcranите

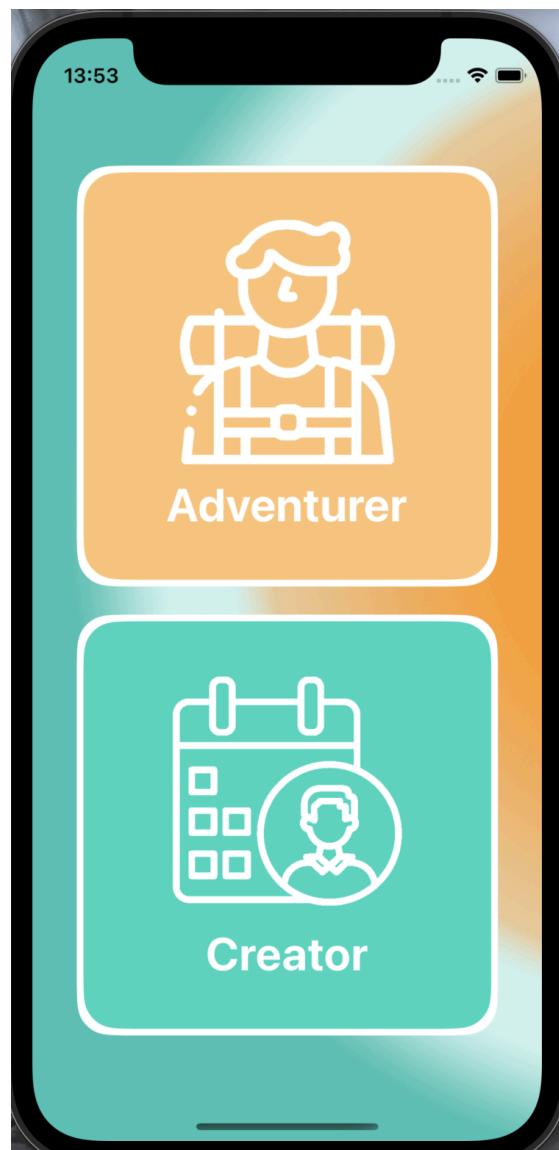
4.2.1. Welcome screen

Първият еcran (*Фиг. 4.1.*), който се появява след стартиране на приложението е еcranът за добре дошли в приложението. Това е еcran, предоставящ на потребителите си две възможности - да си създадат профил или да влязат, ако вече имат такъв.



Фиг.4.1. Еcran за добре дошли

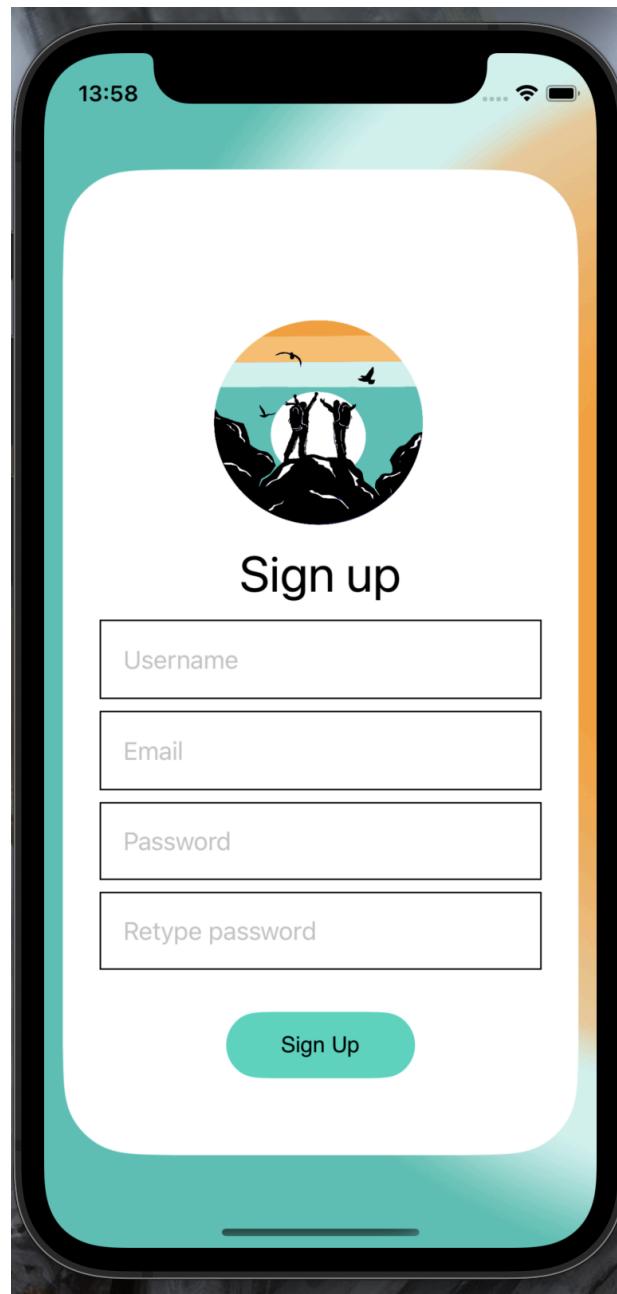
4.2.2. Choosing Role Screen



Фиг.4.2. Екран за избор на тип профил

При избиране на опция за регистрация или вход, потребителя е препратен към еcran, подтикващ го да избере типа профил, който да му бъде създаден. Предоставени са две възможности приключенец или създател.

4.2.3. Sign up screen

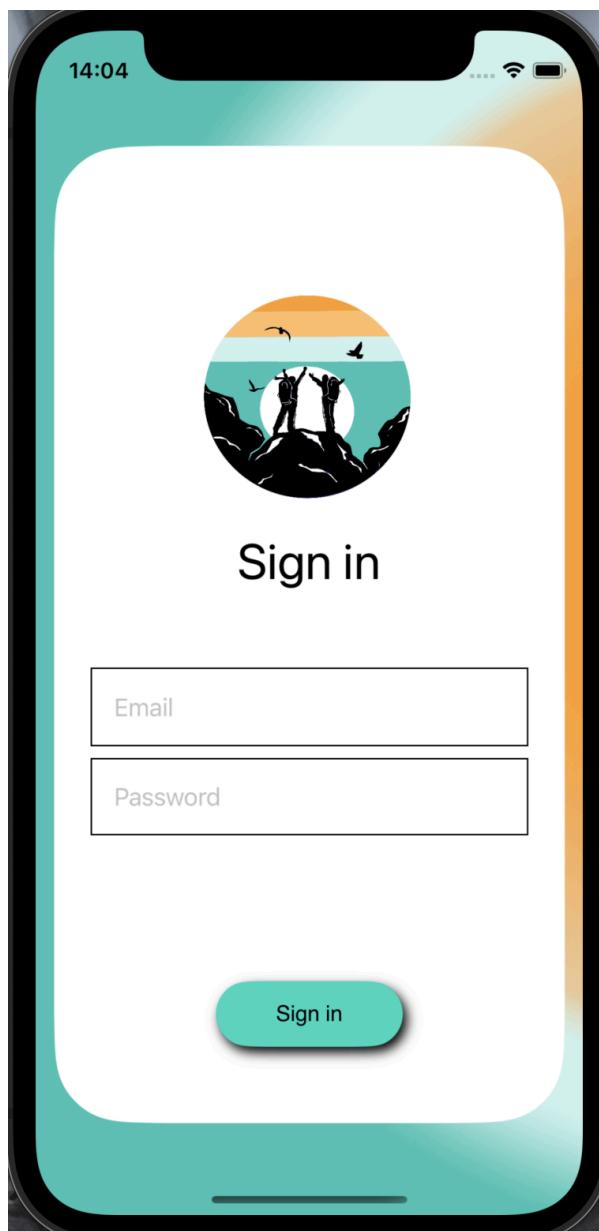


Фиг. 4.3. Екран за регистрация

След избора на потребителя, той бива препратен към съответния екран за регистрация или вход, като типа избран потребител. При избиране на регистрация потребителят е отведен към екрана за регистрация, който ще

поеме неговите входни данни и след натискане на бутона, ще му създаде профил. Данните които задължително се изискват са потребителското име, с което ще бъде представена на другите потребители, имейл, чрез който след това да може да достъпва профила си и парола, която трябва да потвърди.

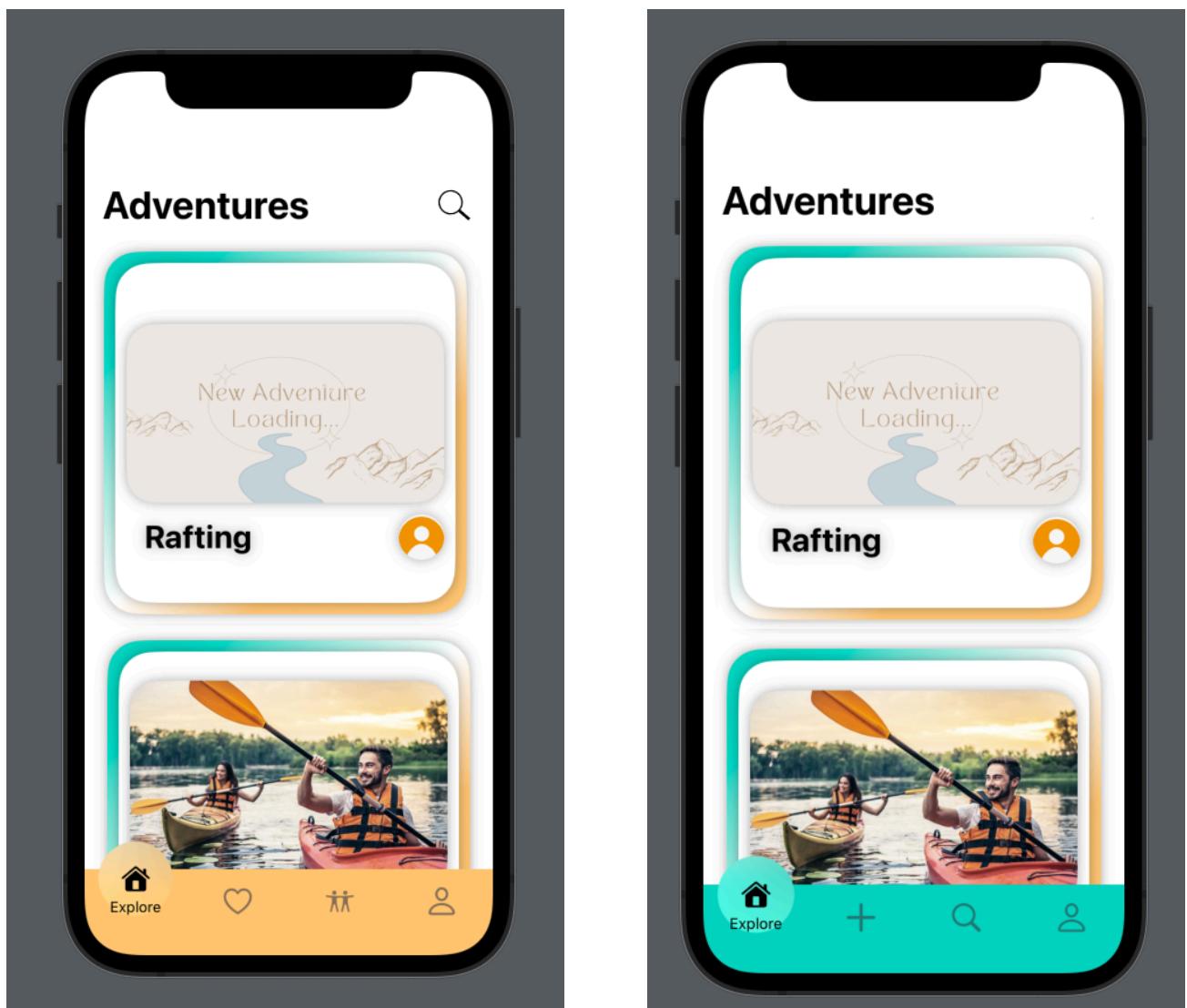
4.2.4. Sign in screen



Фиг.4.4.. Екран за вход

Другата опция, която се предоставяше от първият екран (*Фиг.4.1*) , потребителят ще бъде отведен във формата за вход в платформата, в която трябва да въведе имейл адреса, с който се е регистрирал и парола, която също да отговаря на паролата, избрана при регистрация.

4.2.5. Explore screen

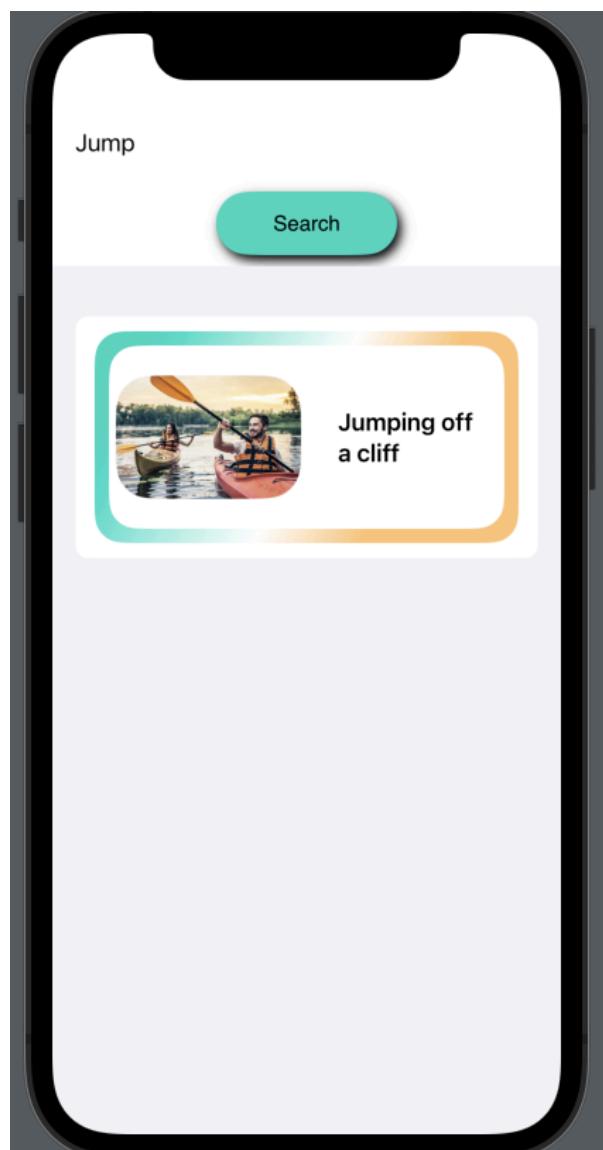


Фиг.4.5. Екрани за разглеждане на приключения. Вляво - еcran за приключенците, вдясно - за създателите

След успешна регистрация на нов потребител, в зависимост от типа потребителски профил сме препратени към основната страница за откриване на приключения. Тук потребителят може да разглежда

създадените от всички създатели приложения. Ако се заинтересува от някое, при неговото натискане потребителят бива отведен към страница с детайлите на това събитие. В горният десен ъгъл има бутон, който позволява търсенето на събития по ключова дума на приключения.

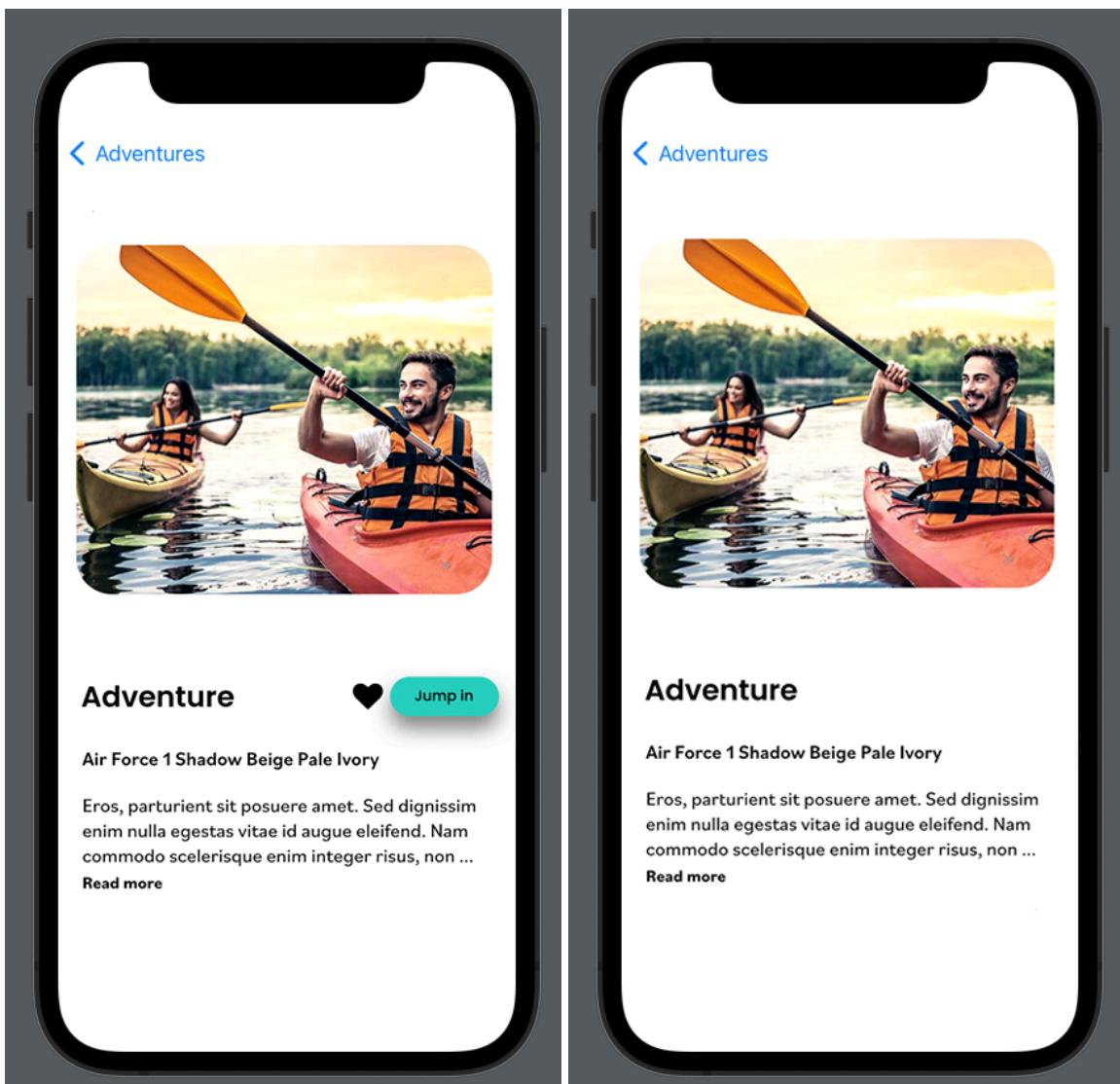
4.2.6. Search screen



Фиг.4.6. Еcran за търсене на ключова дума от приключенец

Екранът за търсене по ключови думи представлява търсачка, която при търсене, показва всички приключения, за които има съвпадение. При тяхното натискане пак се отваря страница с детайли за приключението.

4.2.7. Detailed Adventure screen]

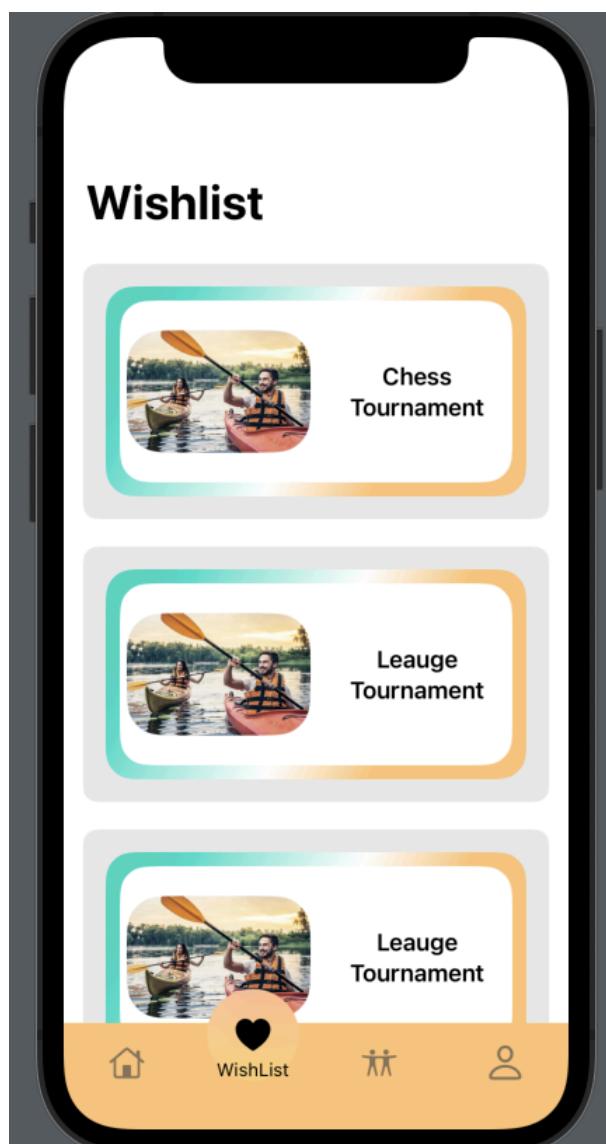


Фиг.4.7. Екрани за детайллен изглед на приключение. Вляво - экран за приключенците, вдясно - за създателите

Страницата с детайлите за всяко приключение се състои от заглавието му, създателя му, описанието, бутон, чрез който потребителят може да добави

събитието си в любими и бутон, чрез който може да го добави във списъка с посетени събития.

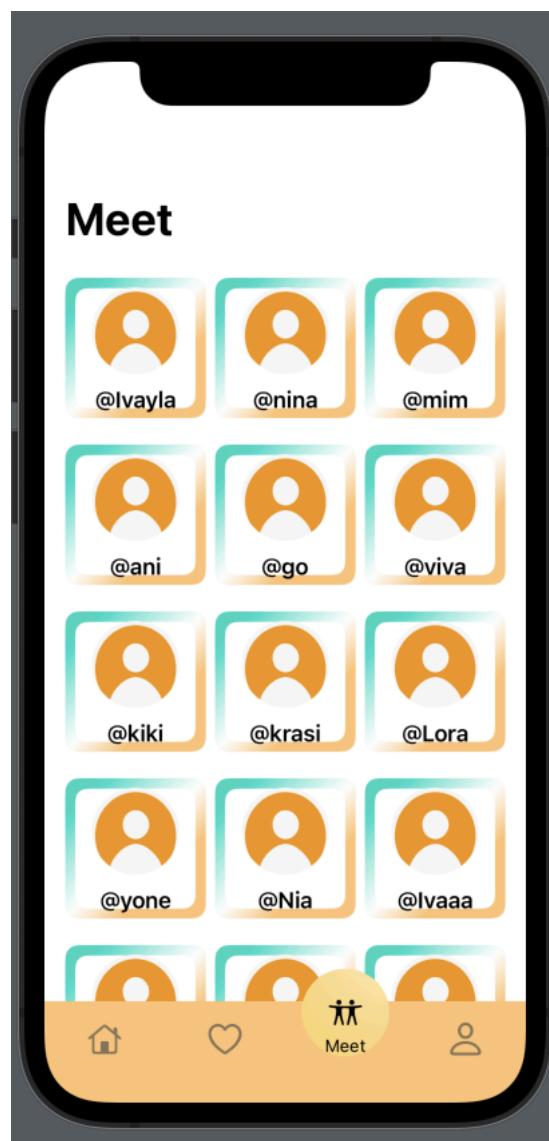
4.2.8. Wishlist screen



Фиг.4.8. Екран за разглеждане на списъка с любими приключения

При избиране от навигационното меню секцията “Wishlist”, приключенецът е отведен до еcran с всички запазени от него събития. При избиране на събитие потребителя отново бива препратен към екрана с детайли за приключението, което е избрал.

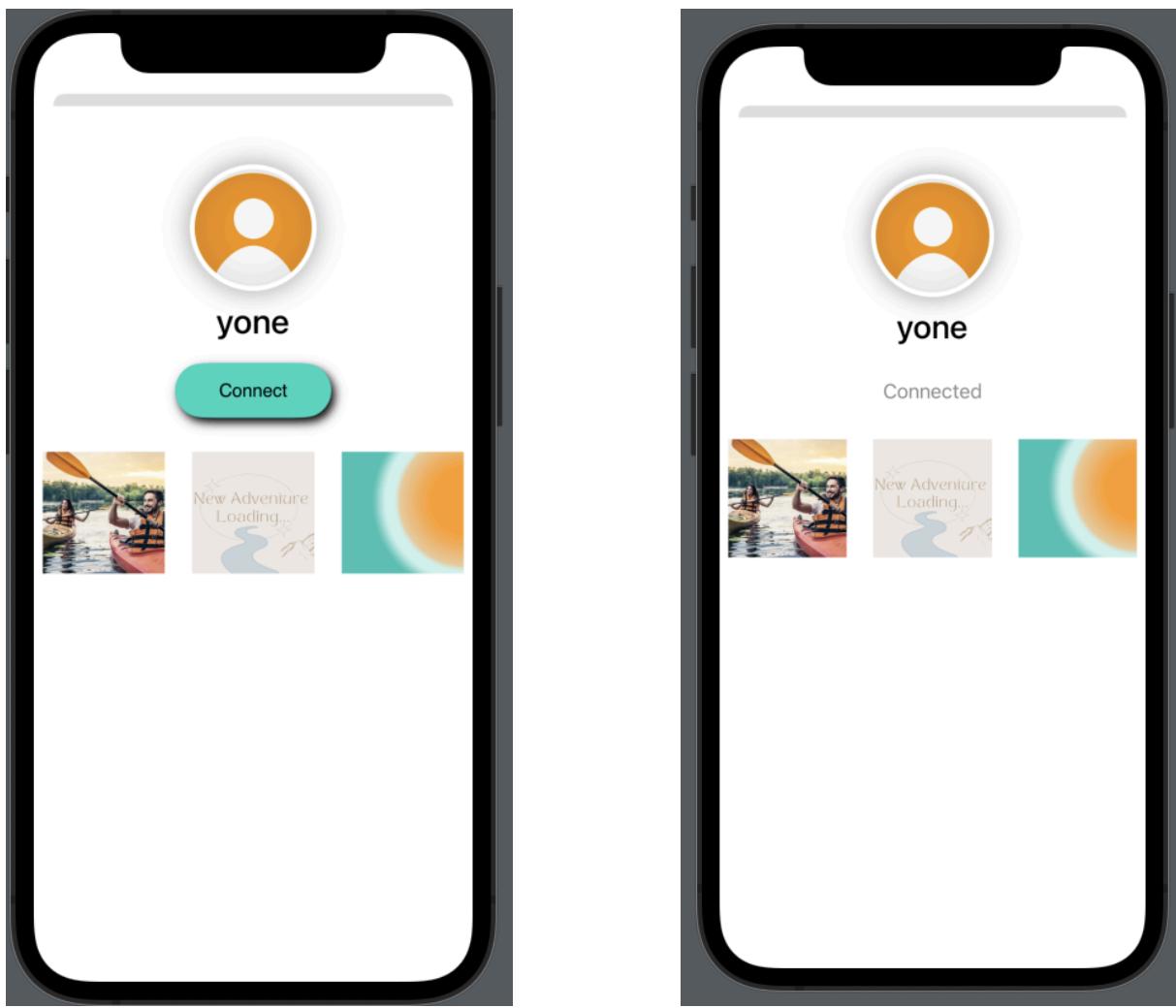
4.2.9. Meet screen



Фиг.4.9. Екрани за разглеждане на приключенците, регистрирани в платформата.

При избиране на секцията “Meet” се стига до еcran с приключенците, регистрирани в платформата. Приключенецът разполага с възможността да се свърже с тях, чрез бутона “Connect”.

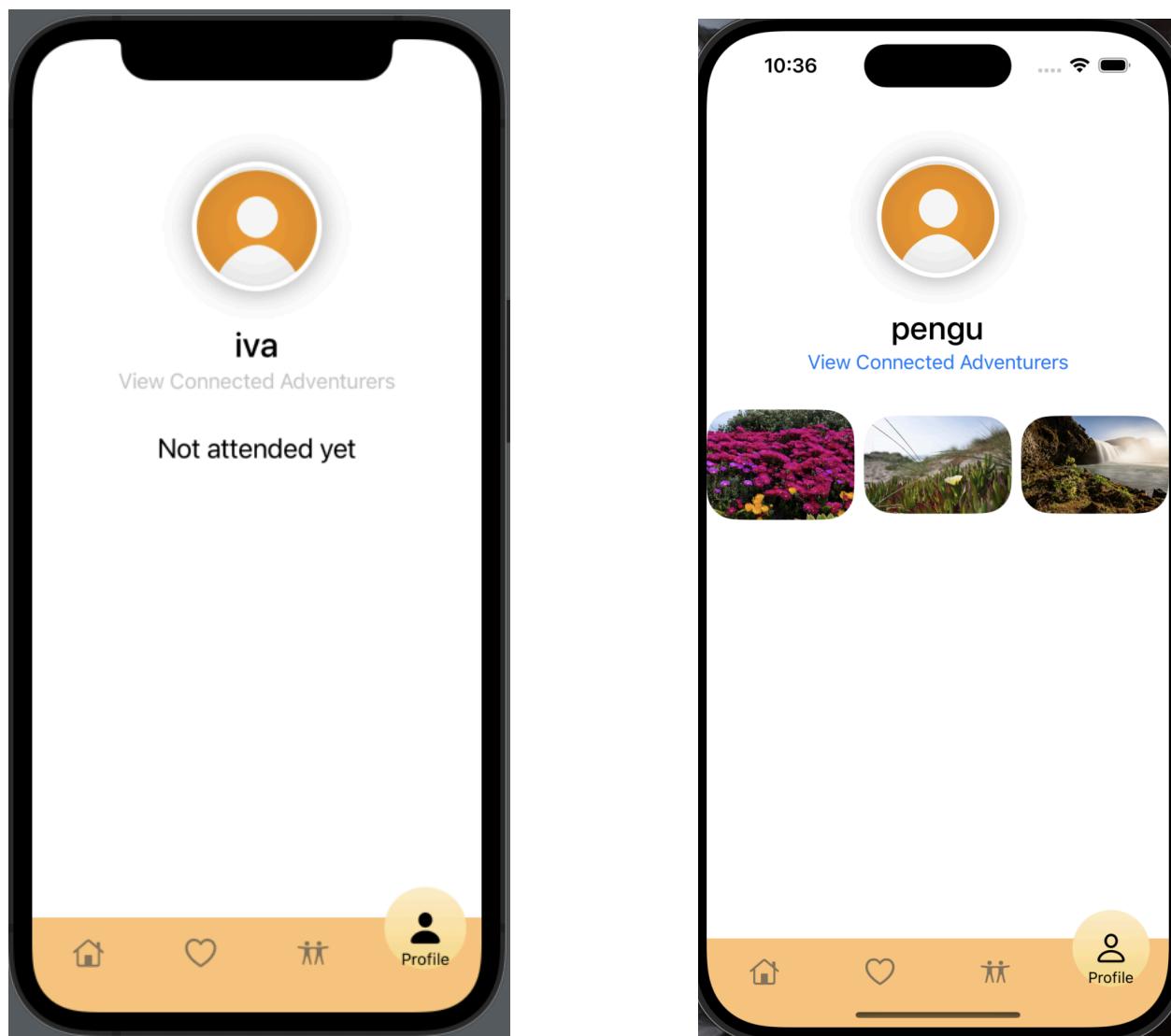
4.2.10. Connect to adventurers screen



Фиг.4.10. Екран за свързване с приключенец. Вляво - екран за приключенец, с когото текущият потребител не е свързан, вдясно - приключенец с вече съществуваща връзка.

При успешно свързване бутона за връзка е изключен, а потребителят е добавен в секцията със свързани приключенци, която може да се достъпи от секция профил в навигационното меню.

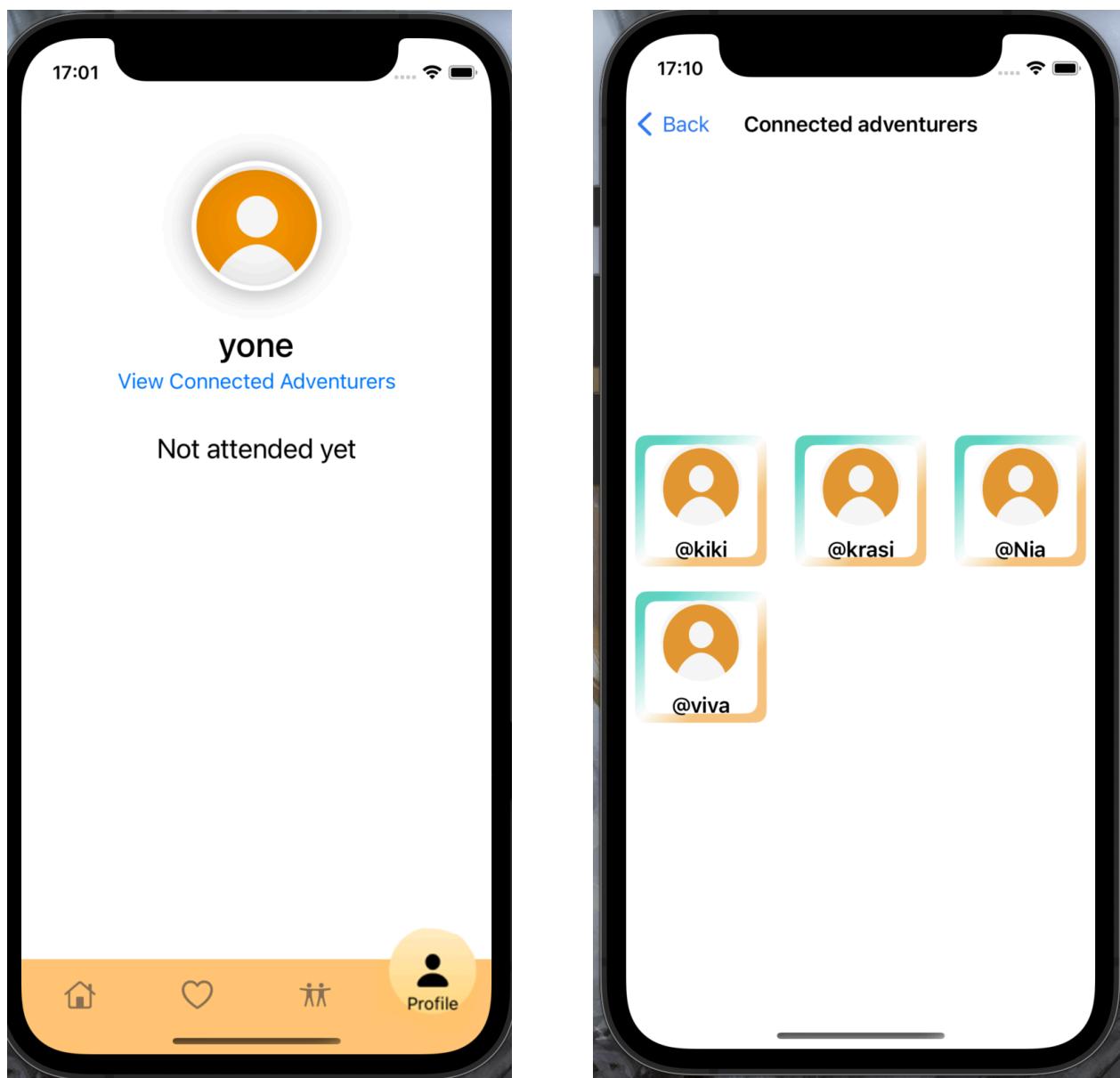
4.2.11. Profile adventurer screen



Фиг.4.11. Екран за разглеждане на потребителски профил и приключения, в които е взето участие. Вляво - екран за приключенец, който не е участвал в събития, вдясно - за вече участвалите.

При избиране на секция профил, приключенецът може да разгледа собствения си профил. Чрез бутона “View Connected Adventurers” се показват приключенците с които потребителят е свързан. В профила също така са показани и приключенията в които потребителят е взел участие. При избиране на някое от тези приключения, потребителят е отведе до страницата с детайли за приключението

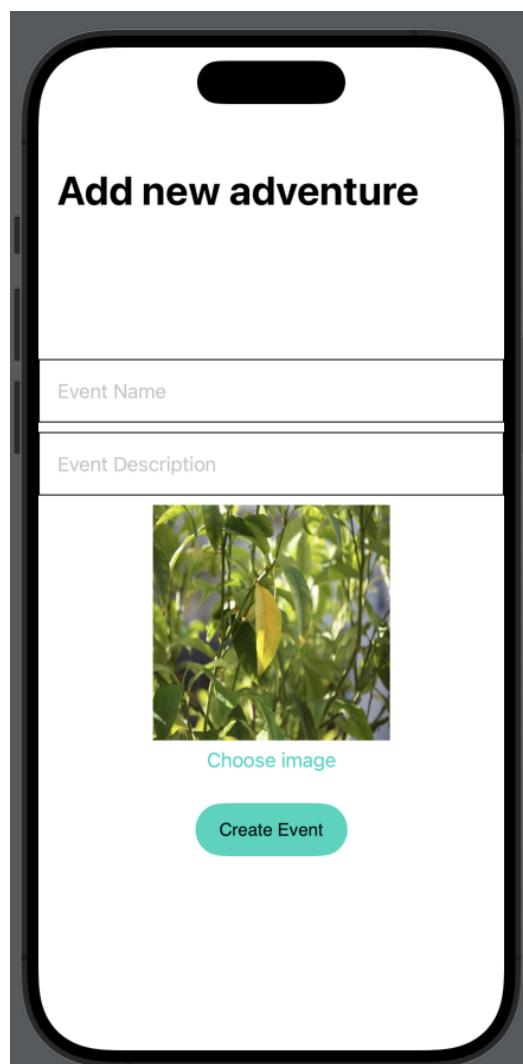
4.2.12. Connected adventurers screen



Фиг.4.12. Екран за разглеждане на свързаните потребители. Вляво - экран изглед на профила, вдясно - свързаните потребители

Когато приключенец е свързан с поне един друг приключенец, бутона “View Connected Adventurers” става активен и потребителят може да разгледа връзките си.

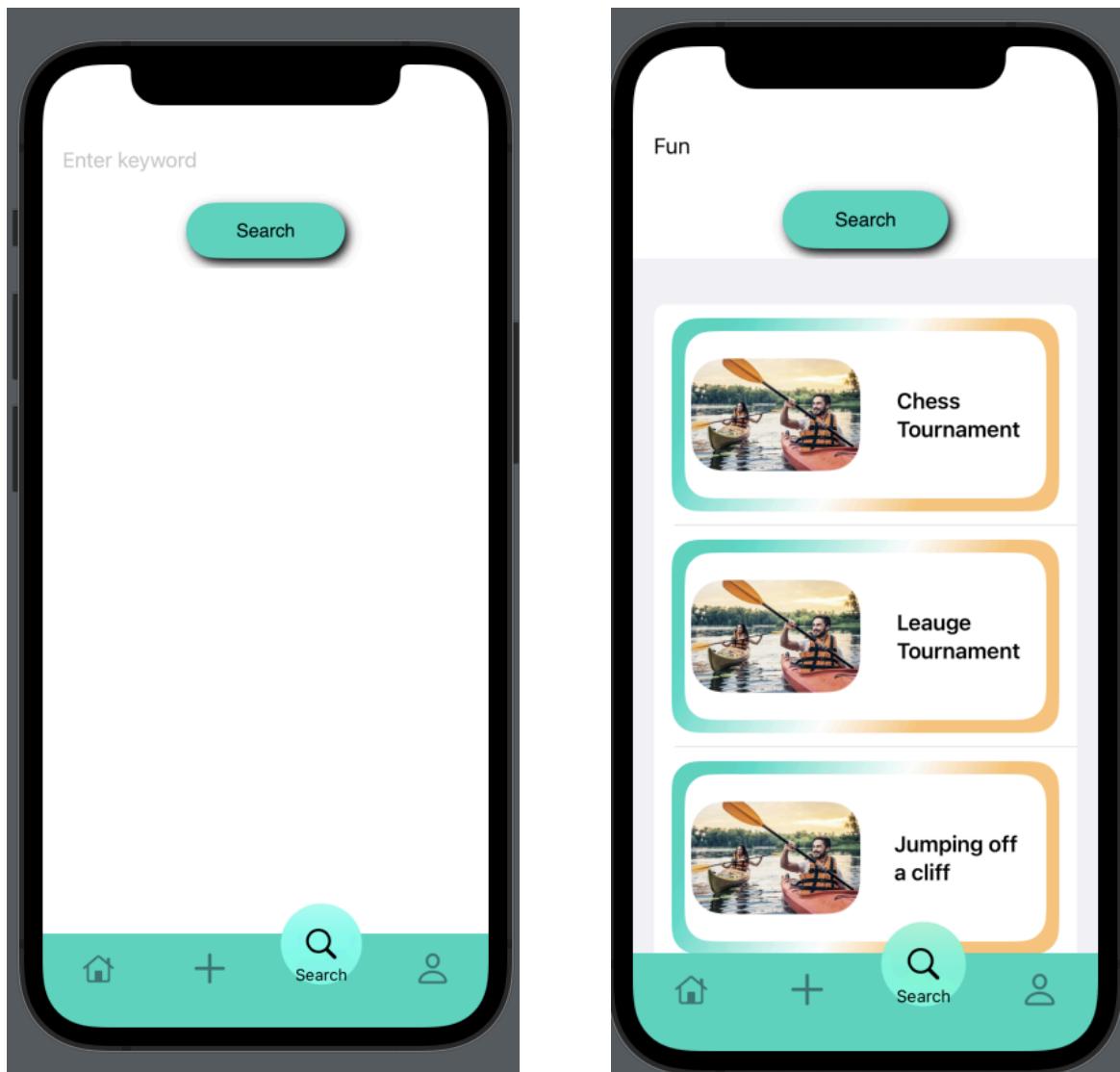
4.2.13. Add new adventure screen



Фиг. 4.13. Екран за добавяне на ново приключение

При избор на секцията “Add” в навигационното меню на създателите, създал може да създаде ново събитие. Има форма в която се въвеждат задължителните данни - име и описание на събитието и незадължителни, които са снимките.

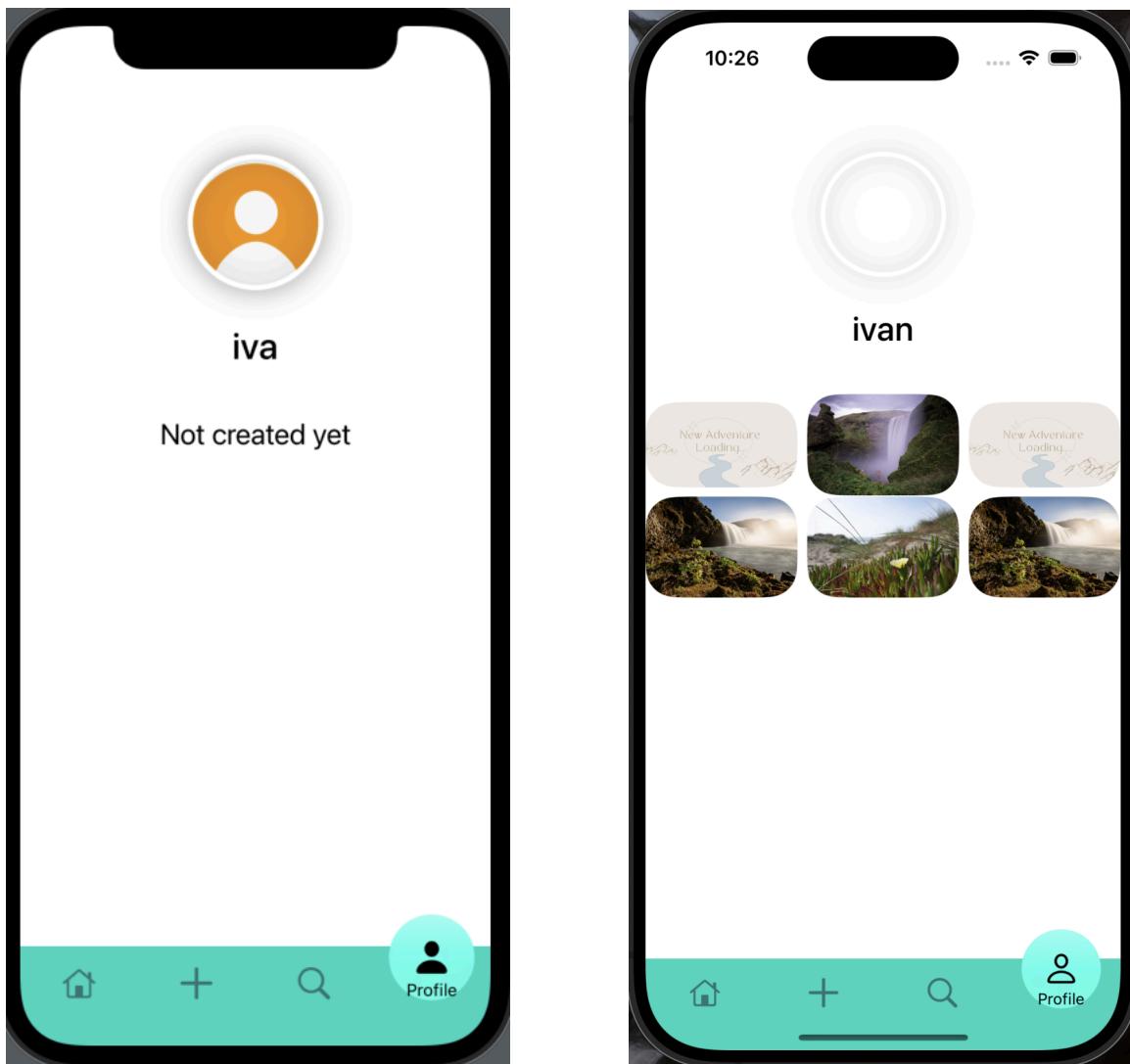
4.2.14. Search creator screen



Фиг.4.14. Екран за търсене на събития по ключови думи от създалел

Ако се избере секция “Search” в създалелското навигационно меню, потребителят е отведен до форма в която може да търси събития по ключови думи. При избор на някое от събитията, това ще отведе създалеля към детайлната страница за дадено събитие.

4.2.15. Profile creator screen



Фиг.4.15. Екран за изглед на потребителски профил. Вляво - экран на създалел, който няма създадени приключения, вдясно - за създалелите които имат такива.

Секцията профил, позволява на създалеля да преглежда собствения си профил. Там той може да разглежда събитията, които е създал. Ако няма създадени събития получава надпис, че все още няма създадени събития.

Заключение

Всички основни функционалности на приложението са изпълнени. Потребителите успешно ще могат да се регистрират и да влизат с профилите си, да разглеждат създадените събития и да търсят такива по ключови думи. Приключенците могат да взимат участие в събития и да се свързват с други приключенци. Те разполагат и със списък с любими събития, където могат да добавят своите избрани. Създалите могат да създават нови приключения. Мобилното приложение може да бъде развито в няколко посоки. Като бъдеща реализация може да се добави изглед към това колко от свързаните приключенци на текущия са участвали в дадено събитие, чат с който приключенците да могат да уговорят съвместното си участие. Друга функционалност ще бъде да се добави меню за настройки за по-лесни промени по профилите на потребителите. Система за класация на потребителите спрямо брой взети участия в преживявания, с цел стимулиране на по-голяма активност. Като заключение, платформата за създаване и участие в различни приключенски събития представлява една удобна и лесна за използване платформа, която ще помогне на потребителите да намират нови нестандартни интереси и хобита и запознанства с хора имащи подобни такива.

Използвана литература

Swift UI documentation - <https://developer.apple.com/xcode/swiftui/>

Swift UI Course - <https://cs193p.sites.stanford.edu/2023>

Nest JS documentation - <https://docs.nestjs.com/>

TypeORM documentation - <https://typeorm.io/>

Amazon s3 documentation - https://docs.aws.amazon.com/?nc2=h_ql_doc_do

MVVM -

<https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm>

Objective - C -

<https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>

Flutter - <https://docs.flutter.dev/>

Java - <https://www.java.com/en/>

Kotlin - <https://kotlinlang.org/docs/home.html>

Sencha - <https://docs.sencha.com/>

Xamarin - <https://dotnet.microsoft.com/en-us/apps/xamarin>

Xcode - <https://developer.apple.com/xcode/>

VisualStudioCode - <https://code.visualstudio.com/docs>

React Native - <https://reactnative.dev/docs/getting-started>

Explorers Connect - <https://www.explorersconnect.com/>

Adventure IO - <https://www.adventureapp.io/>

Meet up - <https://www.meetup.com/>

Adrenaline - <https://www.adrenaline.com/>

Използвани термини

ORM - Object Relational Mapping - обектно релационно картографиране

API - Application Programming Interface - функциите му се изразяват в това да позволява на програмите да си комуникират.

Framework - софтуер, който носи в себе си базови функционалности в зависимост от типа приложение, който искаме да изграждаме.

JVM - Java Virtual Machine - Виртуална машина за изпълнение на

Java - обектно-ориентиран език за програмиране от високо ниво, известен с платформената си независимост и широкото си приложение за създаване на надеждни и мащабируеми софтуерни приложения.

SDK - комплект за разработка на софтуер

MVVM - Model-View-ViewModel

Framework - Софтуерна рамка

IDE - Integrated Development Environment - Развойна среда

UI - Потребителски интерфейс

JSON - JavaScript Object Notation - Текстово базирана обектна нотация