CHARMR.
MOBILE DATING APP

Full Stack Client-Server
Based Mobile Application.
Documentation

Developer:
Alexsandar Ganchev

Contacts: ganchev.professional@gmail.com

Sofia, 2025

Съдържание

1. Увод: Когато технологиите срещат емоциите

В съвременният дигитален свят социалните взаимодействия и изграждането на нови връзки все по-често се пренасят в дигиталното онлайн пространство. Промените в начина на живот, динамиката на ежедневието и глобализация на комуникацията, породени от бързите темпове, с които технологиите се развиват в наши дни, оказват силно влияние върху начина, по който хората се запознават и създават взаимоотношения един между друг. Традиционните методи за запознанства – чрез общи познати, социални събития или случайни срещи – постепенно отстъпва място на иновативни мобилни решения, които предлагат по-голяма достъпност, удобство и най-вече повишен избор за правилния партньор.

Тази трансформация разкрива една основна необходимост: съществуването на дигитални платформи, които не само да улесняват процеса на откриване на нови хора, но и да го правят по ефективен, ангажиращ и сигурен начин. Модерният потребител очаква не просто приложение за запознанства, а завършено преживяване, адаптирано към индивидуалните му нужди, предпочитания и ценности. От тук произтича нуждата от платформи, които комбинират технологични иновации с внимателно обмислен потребителски интерфейс и фокус върху защита на личните данни.

В този контекст възниква **Charmr** – мобилно приложение за запознанства, създаденото с идеята да предложи съвременна реализация на до тук описаната концепция за това какво трябва да представлява една такава апликация. Charmr е проектиран не просто като средство за съвпадение между двама души, а като дигитална платформа, която подкрепя изграждането на автентични човешки взаимоотношения в условията на бързо развиващ се и често отчуждаващ се свят.

В основата на Charmr стои разбирането, че качествените връзки се градят не само върху визуално привличане, а върху споделени ценности, интереси и емоционална съвместимост. Приложението поставя акцент върху преживяването на потребителя – от първия контакт с платформата до възможността за смислено взаимодействие с други хора, като всеки елемент от потребителския път е внимателно обмислен. Charmr насърчава не просто търсенето на мимолетна комуникация, а възможността за създаване на истински връзки, основани на доверие, уважение и откритост.

Проектът е вдъхновен от желанието да се създаде по-човечно и по-съзнателно пространство за запознанства – място, където дигиталните технологии служат на автентичните човешки нужди, а не ги заместват.

2. Анализ на съществуващи разработки

За да се изгради приложение, което наистина отговаря на нуждите на съвременния потребител в сферата на онлайн запознанствата, е необходимо да се направи критичен анализ на вече съществуващи и утвърдени решения. Този анализ предоставя не само основа за вдъхновение, но и ясно очертава възможностите за подобрение и иновация. В настоящия раздел ще бъдат разгледани три популярни мобилни приложения за запознанства – **Tinder,**

Bumble и **Hinge** – като ще се акцентира върху техните силни и слаби страни от гледна точка на реализация и потребителско преживяване.

2.1 Tinder

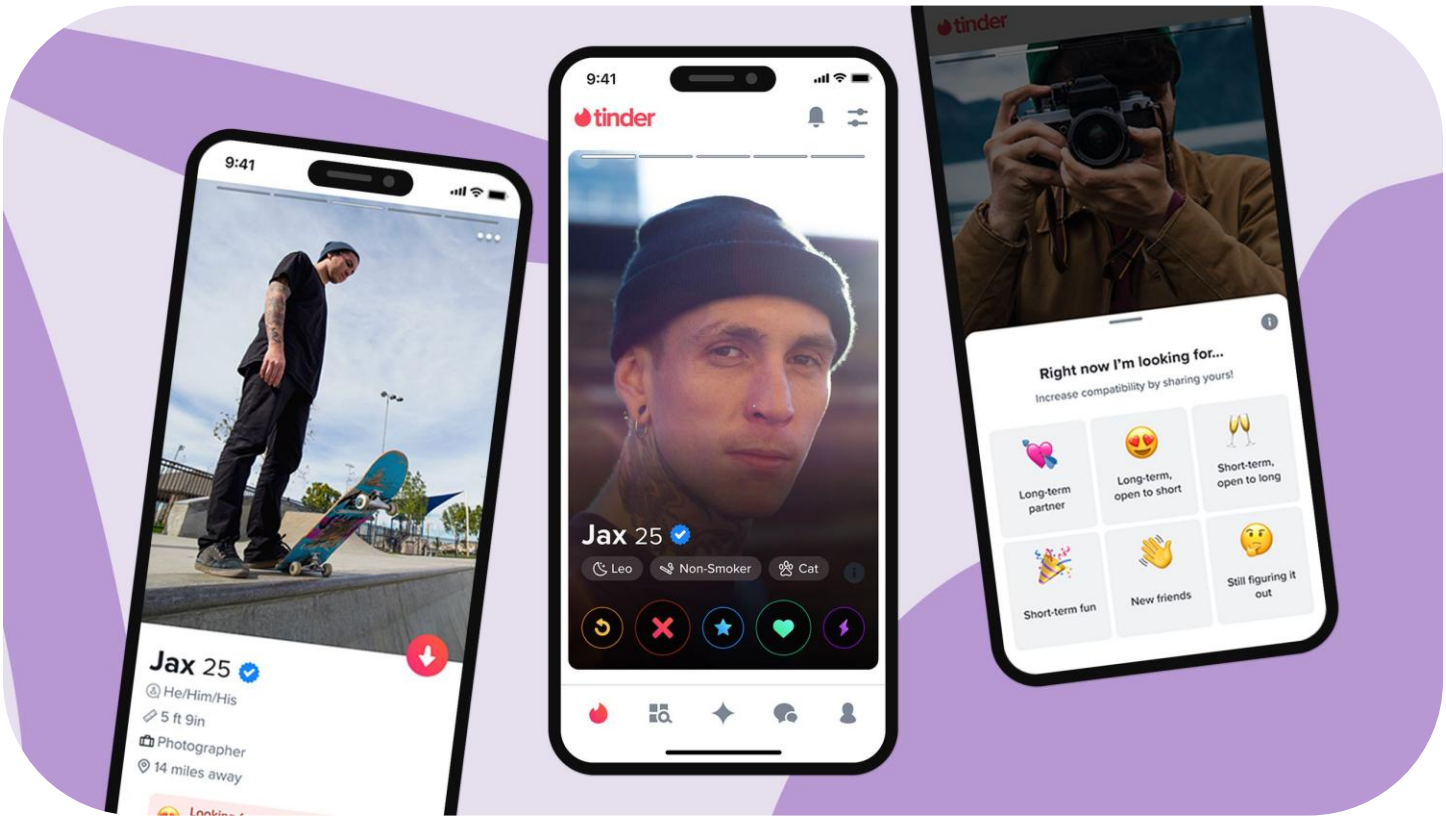
Tinder е безспорно едно от най-популярните приложения за запознанства в световен мащаб. Въведената от него **swipe-механика** (*механика на плъзгане, под формата на тесте карти*) революционизира начина, по който хората взаимодействат дигитално при търсене на партньор.

Положителни страни:

- Изключително интуитивен и опростен потребителски интерфейс.
- Голяма потребителска база, която увеличава вероятността от съвпадения.
- Бърз и лесен процес на регистрация и започване на използване.

Негативните страни:

- Приложението често е критикувано за това, че насърчава повърхностно взаимодействие, базирано основно на външен вид.
- Липса на дълбока персонализация и контекст около профилите.
- Проблеми, свързани с фалшиви профили и безопасността на потребителите.



2.2 Bumble

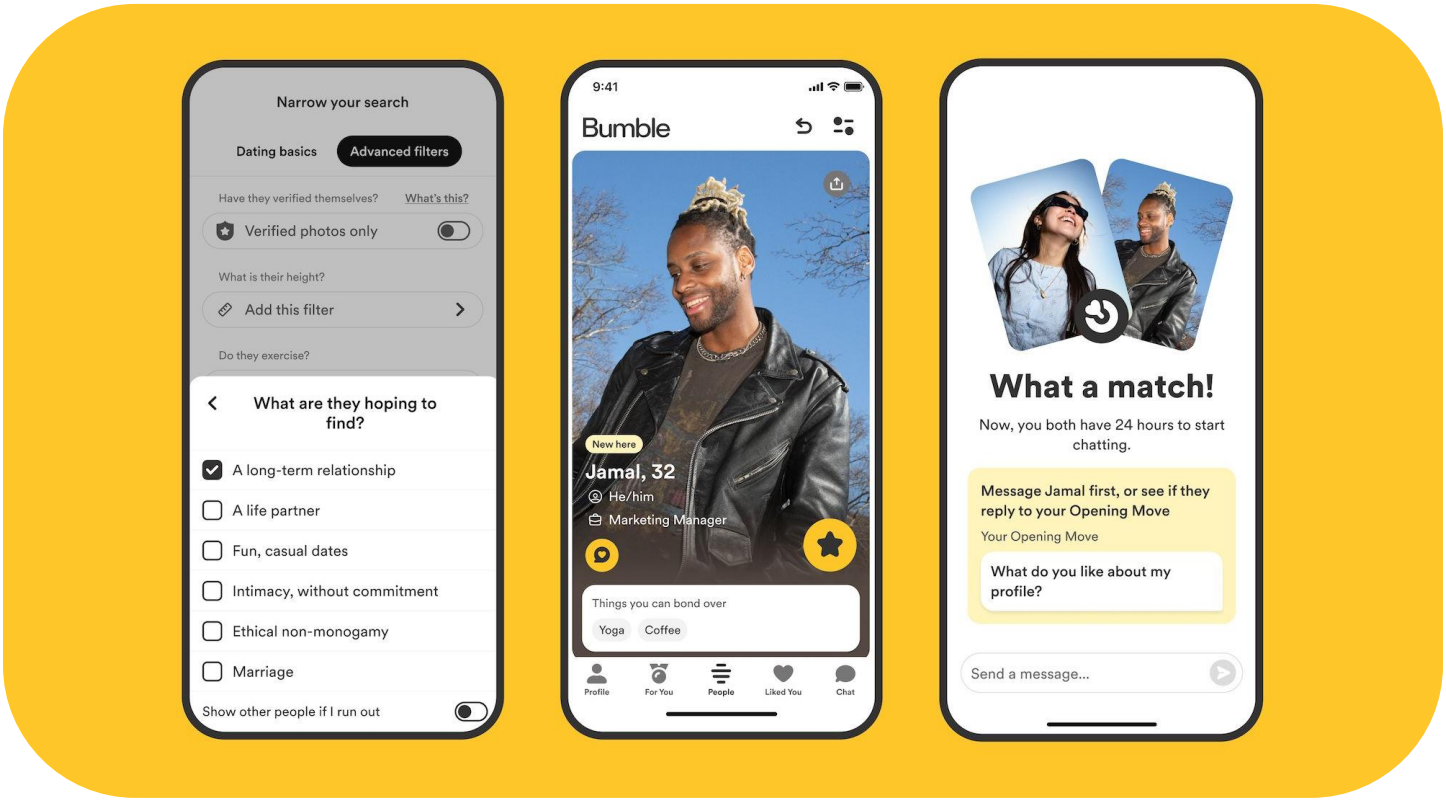
Bumble се отличава с иновативния си подход, при който при хетеросексуални съвпадения, жената е тази, която трябва да направи първата стъпка в комуникацията. Това поставя фокус върху по-равнопоставена и по-съзнателна форма на общуване.

Положителни страни:

- Стимулира по-качествени разговори чрез ограничението във времето за започване на чат.
- Подход, насърчаващ по-голямо уважение към жените и по-малко натрапчиво поведение.
- Допълнителни режими за професионално свързване и създаване на приятелства (Bumble Bizz и Bumble BFF).

Негативните страни:

- Ограничен прозорец от време за започване на разговор може да доведе до пропуснати възможности.
- Все още запазва основно визуалния принцип на оценка.
- Някои потребители намират интерфейса за претрупан и по-малко интуитивен.



2.3 Hinge

Hinge се рекламира като приложението, създадено "за изтриване" – с идеята, че потребителите ще намерят партньор и ще напуснат платформата. То насърчава по-дълбоки взаимодействия чрез въпроси, персонализирани подсказки и повече детайли за всеки профил.

Положителни страни:

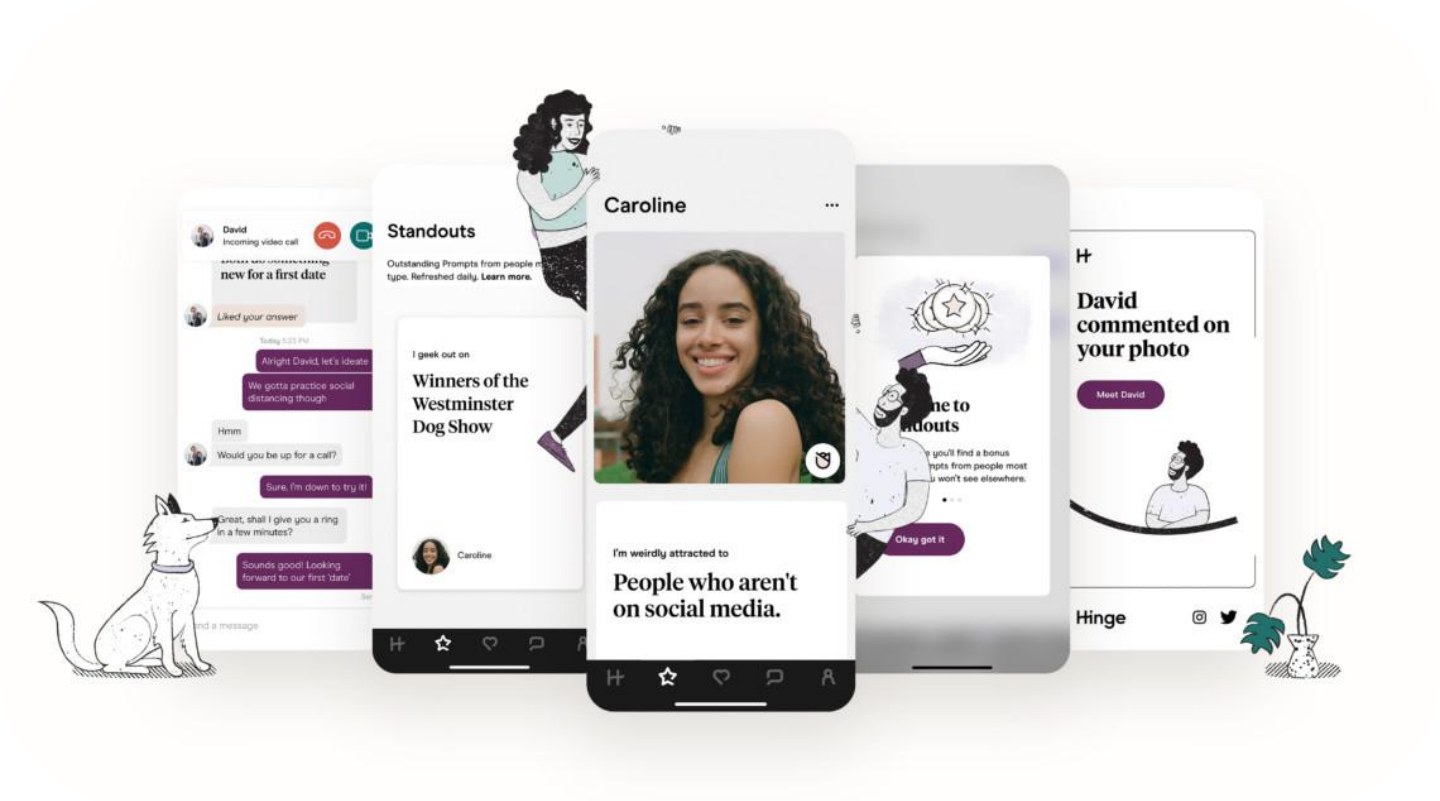
- Стимулира по-смислена комуникация чрез профилни подсказки и тематични въпроси.
- Възможност за изразяване на индивидуалност и характер, а не само визуална презентация.
- По-високо ниво на ангажираност от страна на потребителите.

Негативните страни:

- По-сложният процес на създаване на профил може да бъде възпиращ за някои потребители.
- По-малка потребителска база в сравнение с гиганти като Tinder.

- Понякога алгоритмите за съвпадения не отразяват добре реалните интереси и предпочитания.

Този анализ показва, че въпреки разнообразието от подходи в съвременните приложения за запознанства, съществуват общи слабости – липса на дълбочина в комуникацията, повърхностност при избора на партньор и често пренебрегване на аспекти, свързани със сигурността и автентичността. Именно тези аспекти Charmr се стреми да адресира и подобри, предлагайки ново, по-балансирано изживяване за потребителите.



3. Проектиране

3.1 Целева аудитория и профили на потребителите

Успешното проектиране на едно мобилно приложение започва с ясно разбиране на неговата аудитория. Charmr е насочено към съвременни, дигитално активни потребители, които търсят не просто мимолетни срещи, а автентична комуникация и реални възможности за изграждане на отношения. Това са хора, които често са заети в своето ежедневие, ценят удобството на мобилните технологии и предпочитат структуриран, но гъвкав подход при запознанствата.

Целевата група на Charmr обхваща **мъже** и **жени** на възраст между **20 и 40 години**, предимно живеещи в **градска среда**, със *средно до високо ниво на дигитална грамотност*. Повечето от тях ще са от **масата на студенти**, **млади професионалисти** или **активни хора** с динамичен начин на живот, които все още не са успели да намерят своя партньор в живота. За тази аудитория е особено важно приложението да бъде не само визуално привлекателно, но и **функционално, интуитивно и сигурно**.

Важно е също така да се отбележи, че Charmr е проектирано така, че да бъде приобщаващо и отворено към хора с **различна сексуална ориентация**, като предоставя възможности за *персонализиране на предпочитанията при търсене на партньор*. По този начин се създава **по-широк социален обхват** и се насърчава усещането за принадлежност към една дигитална общност, в която уважението и автентичността са водещи принципи.

3.2 Анализ на данните и бизнес логика

За да се реализира пълноценно функциониращо приложение за запознанства, е необходимо внимателното структуриране на информацията, която се извлича, обработва и съхранява. Charmr използва добре дефиниран модел от данни, който обединява потребителска информация, интеракции между потребителите, комуникация и географски контекст.

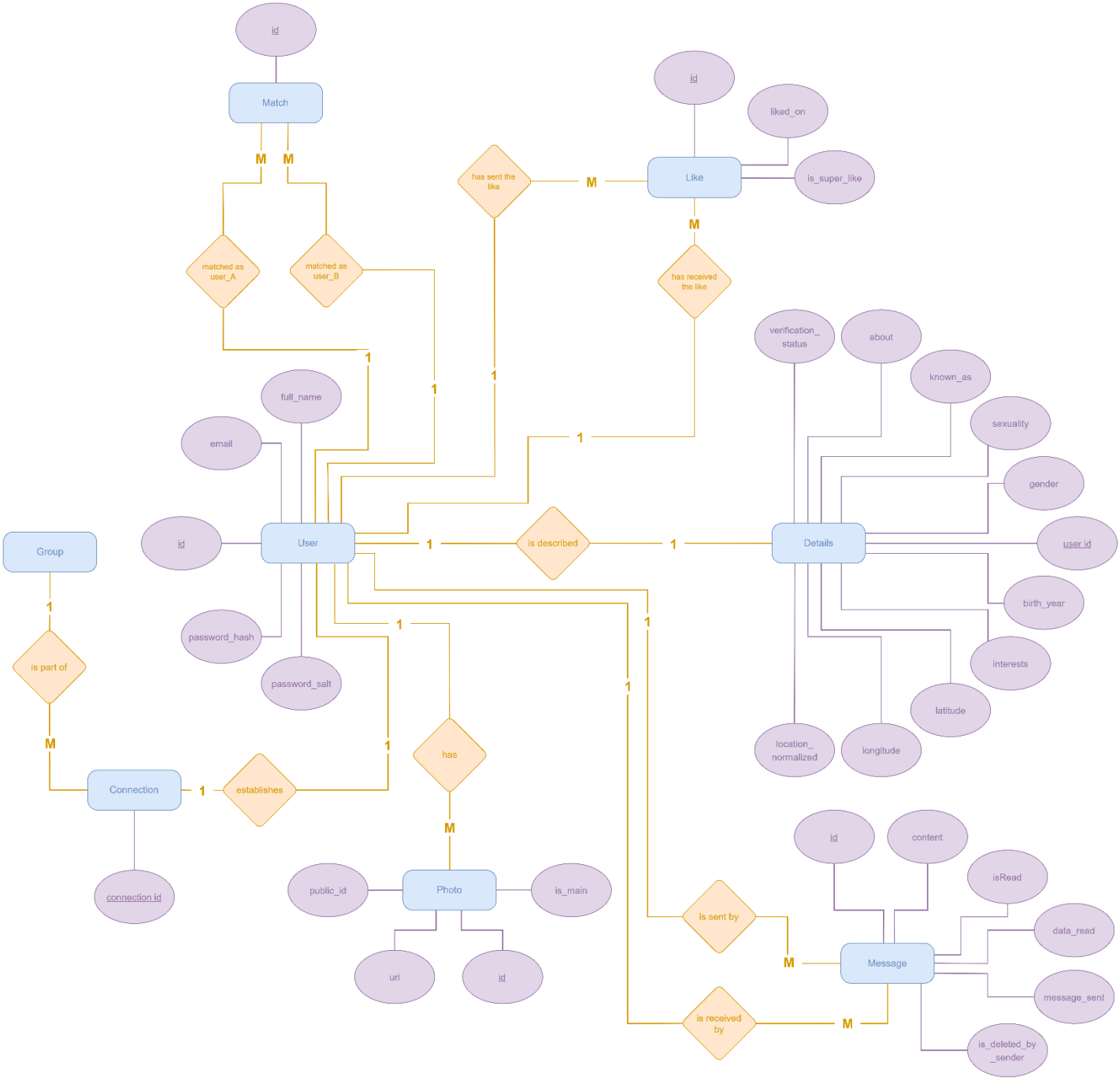
В основата на архитектурата стои entity-то **User**, който съдържа основна информация като име, имейл и криптирана парола. Допълнителните лични детайли, които играят важна роля в подбора на съвпадения, се съхраняват в свързания entity-то **Details**. Там се събират атрибути като пол, сексуалност, интереси, година на раждане, геолокация и кратко описание, както и флаг за верификация. Тази структура позволява богато персонализиране на профила, без да натоварва основния потребителски обект.

Снимките на потребителите са организирани в entity-то **Photo**, който поддържа информация за URL, публичен идентификатор (*използван от интегрирания Cloud Storage Service за изображения)* и флаг, оказващ дали снимката е зададена като профилна. Всеки потребител може да има множество снимки до максимум 6, които са свързани към неговия профила.

Цялата логика по изграждането на съвпадения се управлява чрез два основни entity-та – **Like** и **Match**. При всеки засечен *swipe* надясно се създава запис в таблицата **Like**, който съдържа информация дали става въпрос за *super-like*, както и текущото времето на действието. Когато и двете страни са изразили интерес, се създава запис в **Match**, което задава и основата за комуникацията из между дадените потребителите, които са част от същия този запис.

Обменът на съобщения се осъществява чрез **Message** entity-то, който е структуриран така, че да включва съдържание, дата на изпращане, флагове за прочетено съобщение и изтриване, както и маркира кой е подателя и получателя на същото такова. Това дава възможност за управление на чатовете и историята на разговорите в съответствие с очакванията на потребителите за поверителност и контрол.

В архитектурата на базата данни са включени entity-та отговорни за поддръжката на следенето активното състояние на потребителите *(дали те са онлайн или офлайн)* и това те да получават актуална информация за статуса на своите изпратени съобщения (*дали съобщението е видяно от потребителят, с когото си пишат, дали то е изпратено успешно и др.*). Това е възможно благодарение на интегрирането на логиката базирани на групови връзки/chat-rooms приспособени чрез таблиците **Connections** и **Groups**, които потенциално могат да бъдат използвани за бъдещо разширение на функционалността – например групови събития или тематични групи по интереси в зададена област.



Изградената структура позволява гъвкавост и лесно разширяване на бизнес логиката, като същевременно осигурява ясна и ефективна организация на данните. Тази модулна и релационна организация позволява Charmr не само да отговаря на нуждите на настоящите потребители, но и лесно да се адаптира към бъдещи функции като алгоритми за препоръки, анализ на поведение или социални активности в общността.

Съвпадения и Swipe логика

Основната механика за взаимодействие между потребителите е реализирана чрез т.нар. **dating swiper**, който поддържа следните действия:

- Swipe наляво – *dislike*
- Swipe надясно – *like*
- **Swipe нагоре** – *super like*

Освен gesture-базираната навигация, са налични и **бутони** за всяко действие. Профилите, които се визуализират в swiper-а, се **филтрират по зададени предпочитания**, включващи: *пол, възраст, дистанция (геолокация)*.

Всеки профил може да бъде **разгледан в подробности**, преди да се извърши конкретна swipe-действие.

Система за Likes и Matches

Потребителят има възможност да **следи всички свои интеракции** чрез специална секция с likes:

- **Получени likes** – потребителят вижда кой го е харесал и може директно да направи нов match или да откаже.
- **Отдадени likes** – показва се статусът на отдадените likes (дали са все още активни или са довели до match).
- Потребителят може да **премахне даден like**, ако прецени, че вече няма смисъл от него.

Всяко взаимодействие може да бъде последвано от **match**, който отваря възможност за комуникация между двете страни.

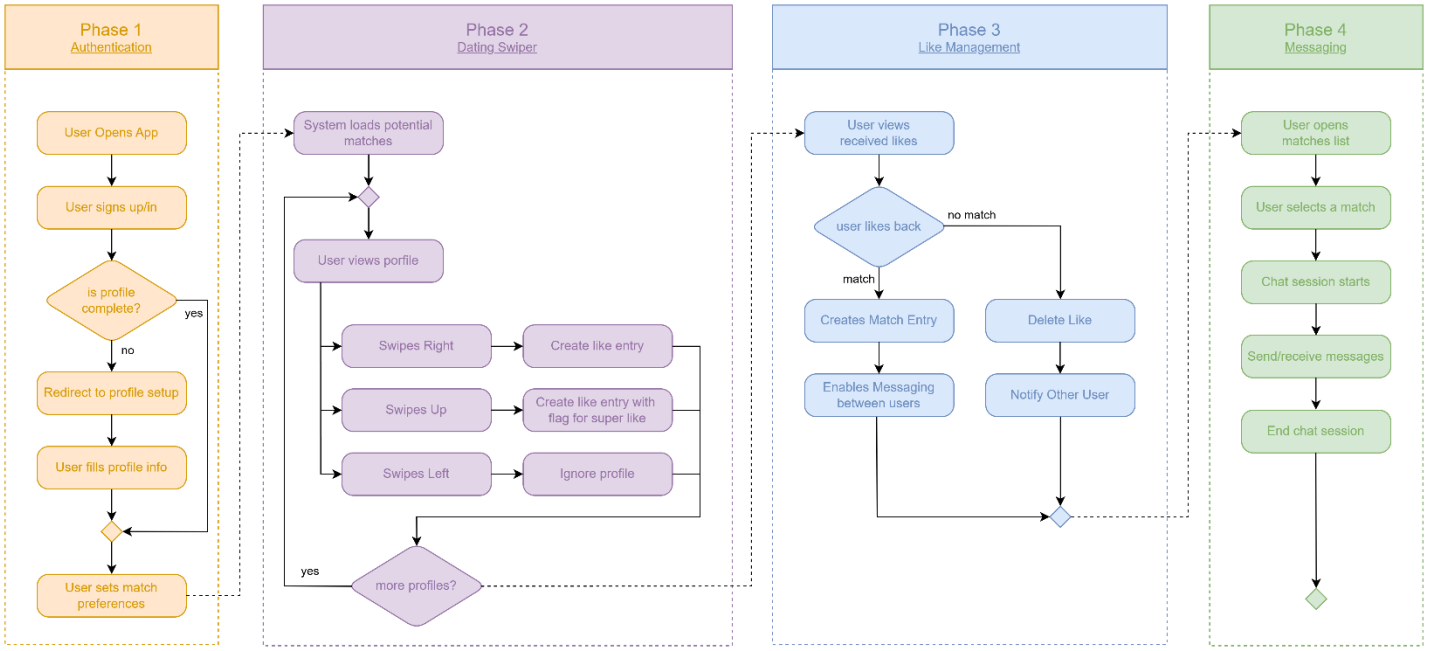
Комуникация и съобщение

Функционалността за чат е базирана на match логиката – съобщения могат да се изпращат само между потребители с двустранен like. Основните възможности включват:

- Inbox с всички съществуващи matches.
- Проследяване на това дали дадените matches са онлайн или не.
- Изпращане на текстови съобщения.
- Визуализация на статус – дали съобщението е прочетено („Seen“)
- Изтриване на съобщения.
- Опция за **Unmatch** директно от чата.

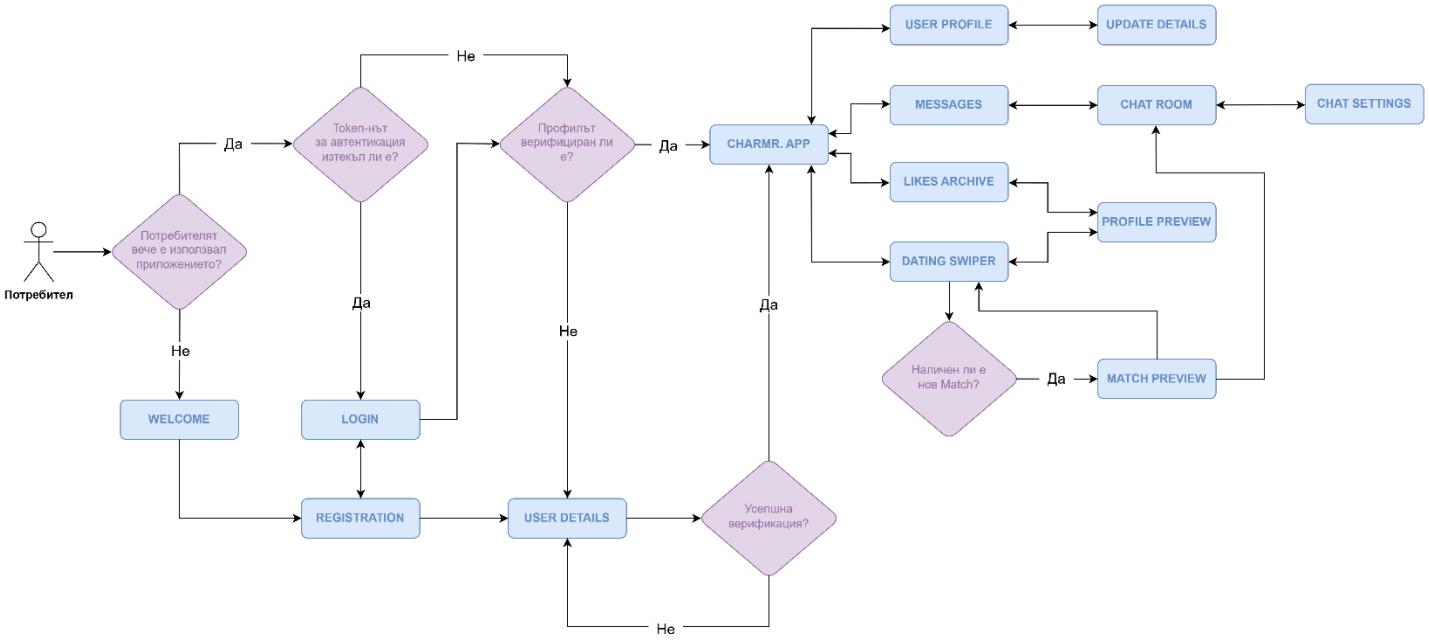
Профил на потребителя

Потребителският профил съдържа **подробна лична информация**, която може да бъде редактирана по всяко време. Потребителят може да **вижда как изглежда профилът му** за останалите в платформата.



3.3 Потребителски интерфейс и достъп до функционалностите

Навигацията в мобилното приложение Charmr е организирана с цел да осигури плавен потребителски поток, съобразен както с новите потребители, така и с тези, които вече са преминали през процеса на регистрация. Основните функционалности са достъпни чрез логически подредени екрани, групирани в начален модул (onboarding), основен интерфейс и допълнителни екрани за взаимодействие и настройки. Приложението използва **stack navigation** и **bottom tabs navigation** подходи за реализацията на преходите из между отделните екрани, като използва и внедряване (nesting) на допълнителни такива от същия тип като така се постига дълбочина, изразяваща се в слоеста структура, разделена спрямо смисловата функционалност на отделните части на апликацията.



3.4 Архитектура и Използвана технология

Приложението **Charmr** следва **класическия модел клиент–сървър**, при който мобилният клиент, разработен с **React Native**, комуникира с backend services, изградена с **.NET**, посредством добре структуриран **REST API**. Това разделение позволява **ясно разграничаване между отговорностите на frontend-а и backend-а**, повишавайки както мащабируемостта, така и сигурността на системата.

Клиентската част (React Native) е отговорна за:

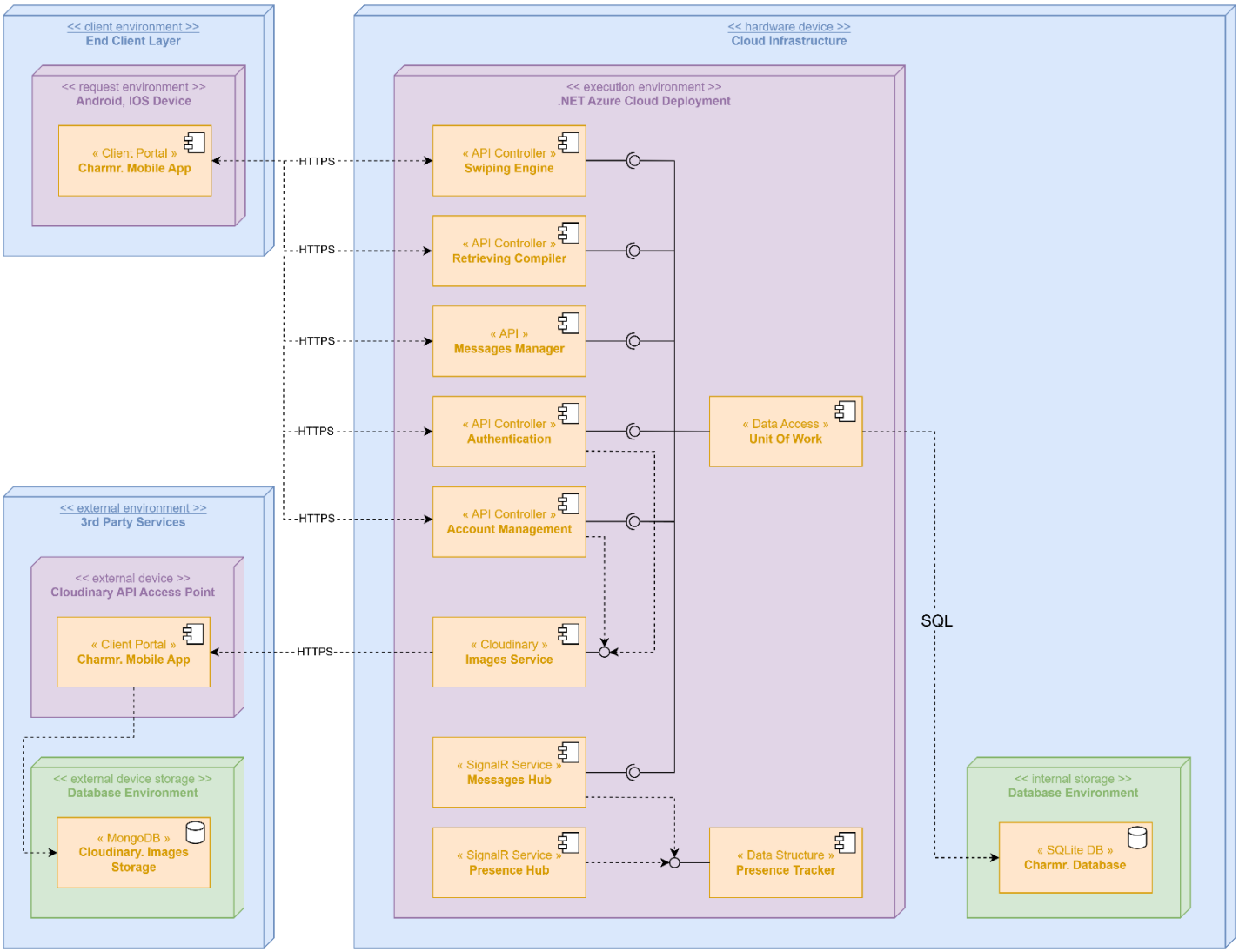
- визуализацията на интерфейса
- обработка на потребителски действия
- изпращане на заявки към сървъра чрез HTTP.

Сървърната част (.NET API) поема:

- бизнес логиката (Match Engine, Authentication, Messaging)
- валидиране и обработка на входни данни
- взаимодействие с базите данни и управление на състояния

Управление на данни:

- **Локална база (SQLite)** – използва се за съхранение на потребителска информация, съобщения и останалата структурирана информация, предоставена в ER диаграмата, представена малко по-рано в документа.
- **Cloudinary** – външен облачен провайдър за съхранение и достъп до потребителски изображения. Системата използва URL адреси, предоставени от Cloudinary, без да запазва самите файлове локално.



Проектът **Charmr** е изграден чрез съвременни и широко утвърдени технологии, които осигуряват стабилност, разширяемост и добро потребителско изживяване както от страна на клиента, така и в сървърната логика.

Клиентска част (Mobile Frontend):

- **React Native**: основната рамка за кроc-платформена мобилна разработка.
- **TypeScript**: добавя статична типизация, подобряваща четимостта и надеждността на кода
- **Redux**: централизирано управление на състоянието на проекта, улесняващо синхронизацията между екраните и отделните компоненти
- **React Native Gesture Handler & Reanimated**: за плавни, адаптивни анимации и жестове *(както е при swipe механиката)*
- **React Native Navigation**: осигурява надеждна и мащабируема система за навигация между екрани

- **Expo**: използван за ускорена разработка и лесна интеграция с различни мобилни функционалности

Сървърна част (Backend):

- **.NET (C#)**: стабилна платформа за изграждане на REST API, осигуряваща висока производителност и сигурност
- **Entity Framework**: ORM технология, която улеснява работата с базата данни чрез обектно-ориентиран подход
- **Repository Pattern + Unit of Work**: осигурява абстракция на достъпа до данни и улеснява поддръжката на кода
- **Dependency Injection (DI)**: използван за ефективно управление на зависимостите и подобрена тестируемост
- **Cloudinary DotNet SDK**: за интеграция с облачното хранилище на изображения, включително качване, достъп и манипулация на медийно съдържание.

Комбинацията от тези технологии и добри практики позволява реализация на мобилно приложение, което е **функционално, мащабируемо и подготвено за реална употреба от крайни потребители**.

4. Реализация

Предвид мащаба и комплексността на проекта **Charmr**, процесът по неговата реализация е структуриран и разгледан в две основни части – **клиентска (mobile frontend)** и **сървърна (backend)**. Това разделение позволява по-добро представяне и анализ на отделните функционални компоненти, архитектурни решения и използвани технологии в контекста на съответната отговорност в системата.

Клиентската част обхваща визуализацията на потребителския интерфейс, навигационната логика и взаимодействието с крайните потребители, докато сървърната страна включва управлението на бизнес логиката, комуникацията с базите данни и обработката на заявките чрез REST API. Заедно те формират една цялостна, модулна и съвременна архитектура, която съчетава гъвкавост, производителност и лесна поддръжка.

5. Сървърна имплементация

Сървърната част на приложението **Charmr** е изградена с помощта на **.NET** и следва принципите на **N-tier архитектура**, с цел ясното разграничаване на отговорностите из между отделените компоненти, подобрена четимост, поддръжка и възможност за разширяване на проекта. Архитектурата е разделена на няколко основни слоя, всеки със специфична роля:

- **Data Access Layer**: отговаря за комуникацията с базата данни посредством **Entity Framework**. В този слой се намират и реалната логика обвързана с Repository интерфейсите и техните реализации, чрез които се имплементира Repository pattern-на за абстрактен достъп до данни.

- **Model Layer:** съдържа основните домейн модели, използвани в приложението. Това включва всички класове, описващи потребители, съобщения, харесвания, изображения и други основни структурни единици, както и свързаните с тях DTO обекти.
- **Web Layer (Public Server Endpoints)** – представлява входната точка на системата и включва:
    - **Controllers** – REST API контролери, които приемат HTTP заявки и ги насочват към съответните услуги.
    - **SignalR Hubs** – използвани за реализация на реално-времевата комуникация в приложението, както при обмена на съобщения.
    - **Services** – слоят, съдържащ бизнес логиката, който стои между контролерите и достъпа до данни. Чрез **Dependency Injection** се осигурява гъвкавост и ниска степен на свързаност между компонентите.

Този многослоен подход гарантира, че всеки компонент на системата е ясно дефиниран, независимо тестируем и лесен за развитие. В резултат се получава **модулен и устойчив backend**, готов да поеме както текущата, така и бъдеща функционална сложност на системата.

5.1 Model Implementation

Съставянето на моделите следва конвенцията наложена от шаблони архитектури като MVC и MVVM, а именно че обозначените класове затова трябва да описват „състоянието“ на съхраняваните от базата данни **entities**. Поради тази причина моделите в базата данни имат следния вид: *property fields (полета)* и *атрибути* към тях, очертаващи допълнителни и по-точни рамки на това, за което е предназначено даденото поле.

```
namespace Server.Models
{
    22 references
    public class User
    {
        [Key]
        22 references
        public Guid Id { get; set; }
        13 references
        public required string FullName { get; set; }
        8 references
        public required string Email { get; set; }
        3 references
        public required byte[] PasswordHash { get; set; }
        3 references
        public required byte[] PasswordSalt { get; set; }
        50 references
        public Details Details { get; set; }
        11 references
        public ICollection<Photo> Photos { get; set; }

        2 references
        public ICollection<Like> LikesGiven { get; set; } = new List<Like>();
        1 reference
        public ICollection<Like> LikesReceived { get; set; } = new List<Like>();

        2 references
        public ICollection<Match> MatchesAsUserA { get; set; } = new List<Match>();
        2 references
        public ICollection<Match> MatchesAsUserB { get; set; } = new List<Match>();
        1 reference
        public ICollection<Message> MessagesSent { get; set; } = new List<Message>();
        1 reference
        public ICollection<Message> MessagesReceived { get; set; } = new List<Message>();
    }
}
```

5.2 Application Database Context

ApplicationDbContext представлява **основният клас за достъп до базата данни** в сървърната част на проекта **Charmr**, изградена с помощта на **Entity Framework Core**. Той наследява от *DbContext* и изпълнява ролята на **контекст на връзката с базата**, чрез който се извършват всички CRUD операции върху данните.

Основните му отговорности включват:

- **Дефиниране на таблиците (DbSet свойства)** – всеки **DbSet<T>** отговаря на конкретен модел (напр. Users, Messages, Likes, Photos) и указва Entity Framework да създаде съответна таблица в базата.
- **Конфигуриране на релации и ограничения** – чрез метода OnModelCreating() се настройват зависимостите между Entity–мата (напр. one-to-many, many-to-many), каскадни изтривания, уникални индекси и др.
- **Проследяване на промените** – EF автоматично следи за модифицирани обекти, което позволява да се прилагат промените с едно извикване на SaveChanges() или SaveChangesAsync().
- **Интеграция с DI контейнера** – ApplicationDbContext се регистрира в услугите на приложението, което позволява инжектирането му в repositories и услуги.

Накратко, този клас е **централната точка**, чрез която backend логиката взаимодейства с данните – от извличането им, през тяхната валидация и модификация, до съхранението им в избраната база (в случая – локална SQLite).

```
namespace Server.DataAccess.Database
{
    41 references
    public class ApplicationDBContext : DbContext
    {
        0 references
        public ApplicationDBContext(DbContextOptions<ApplicationDBContext> options) : base(options) { }
        1 reference
        public DbSet<User> Users { get; set; }
        1 reference
        public DbSet<Details> Details { get; set; }
        0 references
        public DbSet<Like> Likes { get; set; }
        0 references
        public DbSet<Match> Matches { get; set; }
        1 reference
        public DbSet<Message> Messages { get; set; }
        2 references
        public DbSet<Group> Groups { get; set; }
        0 references
        public DbSet<Connection> Connections { get; set; }

        0 references
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Details>()
                .Property(d => d.Interests)
                .HasConversion(
                    v => JsonSerializer.Serialize(v, new JsonSerializerOptions()),
                    v => JsonSerializer.Deserialize<List<string>>(v, new JsonSerializerOptions()) ?? new List<string>()
                );

            modelBuilder.Entity<Like>()
                .HasOne(like => like.Liker)
                .WithMany(liker => liker.LikesGiven)
                .HasForeignKey(like => like.LikerId)
                .OnDelete(DeleteBehavior.Restrict);
```

5.3 Repository Pattern

В сървърната реализация на **Charmr** е приложен **Repository Pattern** с цел да се абстрахира достъпът до базата данни и да се отдели бизнес логиката от логиката за работа с данни. Реализацията включва съставяне на **общо (generic/base) repository**, което дефинира стандартните **CRUD операции** (*Get, GetAll, Add, Delete*), валидни за всеки модел в системата, независимо от неговата конкретна структура или предназначение.

Този подход осигурява:

- **повторна използваемост на кода**;
- **по-добра организация** и поддръжка;
- възможност за по-лесно **тестване и мащабиране** на логиката за достъп до данни.

```
namespace Server.DataAccess.Repository.Implementation
{
    17 references
    public class Repository<T> : IRepository<T> where T: class
    {
        private readonly ApplicationDBContext _context;
        internal DbSet<T> dataBaseSet;

        8 references
        public Repository(ApplicationDBContext context)
        {
            this._context = context;
            this.dataBaseSet = this._context.Set<T>();
        }

        14 references
        public void Add(T entity)
        {
            this.dataBaseSet.Add(entity);
        }

        11 references
        public void Delete(T entity)
        {
            this.dataBaseSet.Remove(entity);
        }
```

```
        28 references
        public async Task<T?> Get(Expression<Func<T, bool>>? whereClause = null, string? includeProperties = null, bool tracked = false)
        {
            IQueryable<T> entityQuery;
            if(tracked)
            {
                entityQuery = this.dataBaseSet;
            }
            else
            {
                entityQuery = this.dataBaseSet.AsNoTracking();
            }

            if(whereClause != null)
            {
                entityQuery = entityQuery.Where(whereClause);
            }

            if(!string.IsNullOrEmpty(includeProperties))
            {
                foreach (var prop in includeProperties.Split(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))
                {
                    entityQuery = entityQuery.Include(prop);
                }
            }

            return await entityQuery.FirstOrDefaultAsync();

        }
```

```
        8 references
        public async Task<IEnumerable<T>> GetAll(Expression<Func<T, bool>>? whereClause = null, string? includeProperties = null)
        {
            IQueryable<T> entityQuery = this.dataBaseSet;
            if (whereClause != null)
            {
                entityQuery = entityQuery.Where(whereClause);
            }

            if (!string.IsNullOrEmpty(includeProperties))
            {
                foreach (var prop in includeProperties.Split(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))
                {
                    entityQuery = entityQuery.Include(prop);
                }
            }

            return await entityQuery.ToListAsync();
        }
    }
}
```

Тъй като обаче, всеки от моделите сам по себе си може да притежава определени нужди, касаещи неговото менажиране в базата данни, това основно Repository се надгражда с допълнително индивидуално такова, с цел покриването на същите такива. Такава операция например се явява Update. Макар и за повечето модели да е присъща тази операция, нейната логика може да варира спрямо зададения модел.

```
namespace Server.DataAccess.Repository.Implementation
{
    2 references
    public class UserRepository : Repository<User>, IUserRepository
    {
        private ApplicationDBContext _context;
        1 reference
        public UserRepository(ApplicationDBContext context) : base(context)
        {
            this._context = context;
        }

        2 references
        public void Update(User user, string email)
        {
            user.Email = email;
            _context.Users.Update(user);
        }
    }
}
```

Такива имплементации като примерната посочена са налични за абсолютно всеки фигуриращ модел в базата данни, като това позволява по-нататъшното модифициране на логиката на конкретния модел, без нарушаването на работещата такава за останалите такива.

5.4 Unit Of Work

В допълнение към **Repository Pattern**, в сървърната архитектура на **Charmr** е внедрен и подходът **UnitOfWork**, който играе ключова роля при координиране на промените върху множество **repositories** в рамките на една обща транзакция.

Идеята зад UnitOfWork е да **обедини всички операции по достъп до данни**, извършвани чрез различни repositories, под единен контекст. Така се гарантира, че **всички свързани промени ще бъдат записани наведнъж** (или напълно отменени при грешка), което е особено важно за **поддържане на целостта на данните**.

Конкретната реализация включва:

- **единен интерфейс IUnitOfWork**, който предоставя достъп до всички нужни repositories
- централизирано извикване на метода **SaveTransaction**(), което управлява записа в базата;
- **инжектиране на UnitOfWork чрез Dependency Injection**, което позволява лесна употреба в контролери и услуги.

Този подход води до по-ясен и устойчив код, улеснява проследимостта на действията върху данните и минимизира риска от частични или неконсистентни състояния в базата.

```
namespace Server.DataAccess.Repository
{
    14 references
    public class UnitOfWork : IUnitOfWork
    {
        private ApplicationDBContext _context;
        18 references
        public IUserRepository userRepository { get; private set; }
        7 references
        public IDetailsRepository detailsRepository { get; private set; }
        12 references
        public IPhotoRepository photoRepository { get; private set; }
        15 references
        public ILikeRepository likeRepository { get; private set; }
        7 references
        public IMatchRepository matchRepository { get; private set; }
        11 references
        public IMessageRepository messageRepository { get; private set; }
        6 references
        public IConnectionRepository connectionRepository { get; private set; }
        7 references
        public IGroupRepository groupRepository { get; private set; }

        6 references
        public UnitOfWork(ApplicationDBContext context) {
            this._context = context;
            this.userRepository = new UserRepository(context);
            this.detailsRepository = new DetailsRepository(context);
            this.photoRepository = new PhotoRepository(context);
            this.likeRepository = new LikeRepository(context);
            this.matchRepository = new MatchRepository(context);
            this.messageRepository = new MessageRepository(context);
            this.connectionRepository = new ConnectionRepository(context);
            this.groupRepository = new GroupRepository(context);
        }

        18 references
        public async Task SaveTransaction()
        {
            await this._context.SaveChangesAsync();
        }
    }
}
```

5.5 Authentication Token Service

Token Service е отговорен за **създаването на JSON Web Token (JWT)** за всеки потребител след успешна автентикация. Служи за **осигуряване на сигурен достъп** до защитени ресурси в

приложението чрез издаване на authenticated token, който съдържа ключови данни (claims) като ID и имейл на потребителя. Token-ът е с валидност 30 дни и се аутентикира със симетричен ключ, конфигуриран от appsettings. Този подход позволява на клиента да използва token за удостоверяване при всяка следваща заявка.

```
namespace Server.Services.Implementation
{
    2 references
    public class TokenService : ITokenService
    {
        private readonly SymmetricSecurityKey _key;

        0 references
        public TokenService(IConfiguration configuration)
        {
            _key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration.GetValue<string>("JwtSettings:TokenKey")));
        }
        3 references
        public string CreateToken(User user)
        {
            var claims = new List<Claim>
            {
                new Claim("NameId", user.Id.ToString()),
                new Claim("Email", user.Email),
            };

            var credentials = new SigningCredentials(_key, SecurityAlgorithms.HmacSha512Signature);

            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(claims),
                Expires = DateTime.Now.AddDays(30),
                SigningCredentials = credentials,
            };

            var tokenHandler = new JwtSecurityTokenHandler();

            return tokenHandler.WriteToken(tokenHandler.CreateToken(tokenDescriptor));
        }
    }
}
```

5.6 Images Service

PhotoService реализира логиката по **качване и изтриване на снимки** в облачната услуга **Cloudinary**. Чрез нея се осигурява централизирано и мащабируемо съхранение на изображения, което освобождава сървъра от нуждата да обработва и съхранява файлове локално. Снимките се качват асинхронно, а при нужда – лесно могат да бъдат изтрити по тяхното publicId. Този service позволява ефективна и сигурна работа с мултимедийно съдържание в приложението.

```
namespace Server.Services.Implementation
{
    2 references
    public class PhotoService : IPhotoService
    {
        private readonly Cloudinary _cloudinary;

        0 references
        public PhotoService(IOptions<CloudinarySettings> config)
        {
            _cloudinary = new Cloudinary(new Account(
                config.Value.CloudName,
                config.Value.ApiKey,
                config.Value.ApiSecret
            ));
        }
```

```
        3 references
        public async Task<ImageUploadResult> AddPhotoAsync(IFormFile imageFile)
        {
            var uploadedResult = new ImageUploadResult();

            if(imageFile.Length > 0)
            {
                using var stream = imageFile.OpenReadStream();
                var uploadParams = new ImageUploadParams
                {
                    File = new FileDescription(imageFile.FileName, stream),

                };
                uploadedResult = await _cloudinary.UploadAsync(uploadParams);
            }

            return uploadedResult;
        }

        2 references
        public async Task<DeletionResult> DeletePhotoAsync(string publicId)
        {
            var deleteParams = new DeletionParams(publicId);
            var deletionResult = await _cloudinary.DestroyAsync(deleteParams);

            return deletionResult;
        }
    }
}
```

5.7 Архитектура на контролерите

Целта на контролерите е да дефинират API End Points на сървъра, през които client страната може да прави requests и да си взаимодейства с данните фигуриращи в базата. Тъй като сами по себе си контролерите в приложението споделят подобна архитектура и взаимодействат по сходен начин с Data Access Layer-а, ще бъде разгледан само един от тях в детайл са да представи генералната логика за това как се извършват CRUD операциите при настъпването на request.

Както бе споменато взаимодействието с данните става индиректно през дефинирания UnitOfWork, който е част от Data Access Layer-а. Чрез него ние вече може да използваме вече съставените операции за CRUD и да се фокусираме върху изграждането на бизнес логиката в рамките на зададения endpoint.

За съставянето на контролера той трябва да унаследи базовите характеристиките на класа BaseController предоставен от .NET. Чрез използването на атрибути дефинираме и самия общ URL path, чрез който ще бъде достъпно правенето на request-ми от страна на клиента. При съставянето на контролера се очаква и инжектирането на инстанция на ApplicationDbContext, с цел съставянето и на инстанция UnitOfWork, през която ще стане възможно взаимодействието с данните.

```
namespace Server.Controllers
{
    [ApiController]
    [Route("api/swiping")]
    1 reference
    public class SwipingActionsController : ControllerBase
    {
        private readonly UnitOfWork _unit;

        0 references
        public SwipingActionsController(ApplicationDBContext context)
        {
            _unit = new UnitOfWork(context);
        }
```

За изграждането на endpoint се съставя метод в рамките на контролера, като отново чрез атрибут се дефинират неговия краен URL Path. Като параметри към тези методи, се подават реално и очакваните данни фигуриращи в request-а, за да бъде възможно изпълнението на исканата операция. Нека разгледаме няколко примера за извличане, добавяне, изтриване, модифициране на данни:

Съставяне на тестето с карти (потребителски профили)

```
        [Authorize]
        [HttpPost("deck")]
        0 references
        public async Task<ActionResult<List<UserSwipeCard>?>> PrepareDeck([FromBody] DeckFiltering filtering)
        {
            var userId = User.ExtractUserId();

            if (userId == null)
            {
                return Unauthorized();
            }

            var user = await _unit.userRepository.Get(u => u.Id == userId, "Details,LikesGiven,MatchesAsUserA,MatchesAsUserB");
            var allUsers = await _unit.userRepository.GetAll(u => u.Id != userId, "Details,Photos");

            if(allUsers==null || user == null)
            {
                return NotFound();
            }

            var likedByTheUser = user.LikesGiven.Select(like => like.LikedId).ToHashSet();

            var matchedUserIds = user.MatchesAsUserA.Select(match => match.UserBId)
                                    .Concat(user.MatchesAsUserB.Select(match => match.UserAId))
                                    .ToHashSet();

            string defaultPreferedGender;
            switch (user.Details.Sexuality)
            {
                case "heterosexual":
                    defaultPreferedGender = user.Details.Gender == "male" ? "female" : "male"; break;
                case "homosexual":
                    defaultPreferedGender = user.Details.Gender; break;
                case "bisexual":
                case "without_preference":
                    defaultPreferedGender = "both"; break;
                default:
                    defaultPreferedGender = "both"; break;
            }
```

```
            var deck = allUsers
                .Where(u => u.Details != null &&
                    (filtering.Gender == defaultPreferedGender ? (user.MatchesOrientation(u)) : (filtering.Gender == "both" ? true : (u.Details.Gender == filtering.Gender)) ) &&
                    filtering.AgeRange[0] <= DateTime.Today.Year - u.Details.BirthYear &&
                    filtering.AgeRange[1] >= DateTime.Today.Year - u.Details.BirthYear &&
                    user.GetDistance(u) <= filtering.LocationRadius &&
                    !likedByTheUser.Contains(u.Id) &&
                    !matchedUserIds.Contains(u.Id))
                .Select(u => new UserSwipeCard
                {
                    UserId = u.Id,
                    Name = u.FullName,
                    Age = DateTime.UtcNow.Year - u.Details.BirthYear,
                    About = u.Details.About,
                    Distance = Convert.ToInt32(user.GetDistance(u)),
                    ProfilePicture = u.Photos.FirstOrDefault(p => p.isMain)?.Url
                })
                .Take(15)
                .ToList();

            return Ok(deck);
        }
```

Запазването на потребителските действия: Likes, Passes, Super Likes, New Matches

```
[Authorize]
[HttpPost("actions")]
0 references
public async Task<IActionResult> ManageSwipingActions([FromBody] SwipingAction[] swipingActions) {
    var likerUserId = User.ExtractUserId();

    if (likerUserId == null)
    {
        return Unauthorized();
    }

    List<MatchPayload> matches = new List<MatchPayload>();

    foreach(var swipingAction in swipingActions)
    {
        switch(swipingAction.ActionType)
        {
            case "like":
                var newMatch = await ManageLike((Guid)likerUserId, Guid.Parse(swipingAction.LikedId), false);
                if (newMatch != null) matches.Add(newMatch);
                break;
            case "super_like":
                var newSuperMatch = await ManageLike((Guid)likerUserId, Guid.Parse(swipingAction.LikedId), true);
                if (newSuperMatch != null) matches.Add(newSuperMatch);
                break;
            case "pass":
                await ManageSkip((Guid)likerUserId, Guid.Parse(swipingAction.LikedId));
                break;
            default: break;
        }
    }

    await _unit.SaveTransaction();

    if(matches.Count > 0)
    {
        return Ok(new { matches });
    }

    return Created();
}
```

Изтриването на харесване (Like)

```
[Authorize]
[HttpDelete("remove-like")]
0 references
public async Task<IActionResult> RemoveLike([FromQuery] Guid likedId)
{
    var userId = User.ExtractUserId();

    if(userId == null)
    {
        return Unauthorized();
    }

    var likeToRemove = await _unit.likeRepository.Get(l => l.LikerId == userId && l.LikedId == likedId);

    if(likeToRemove != null)
    {
        _unit.likeRepository.Delete(likeToRemove);
    }

    await _unit.SaveTransaction();

    return Created();
}
```

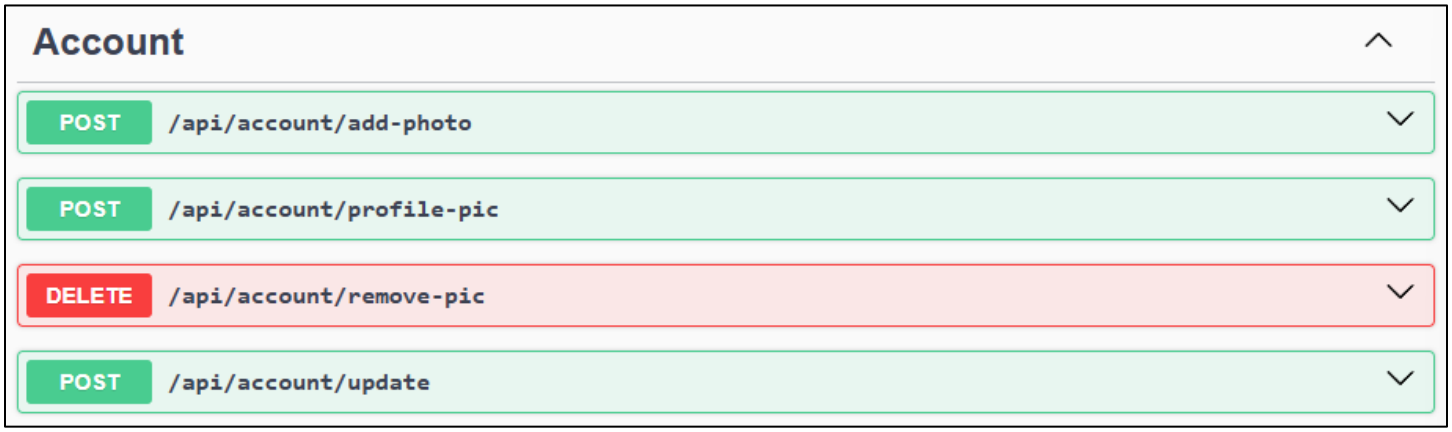
5.8 API Endpoints Definition

Както бе споменато самите контролери следват своята бизнес логика и взаимодействат по сходен начин с UnitOfWork и поради това няма да бърне обърнато задълбочено внимание на това кой как е имплементиран. Вместо това в тази секция ще бъде предоставена обща

информация за дефинираните API endpoint-ове и каква функция те изпълняват в рамките на цялостната апликация:

Account Controller

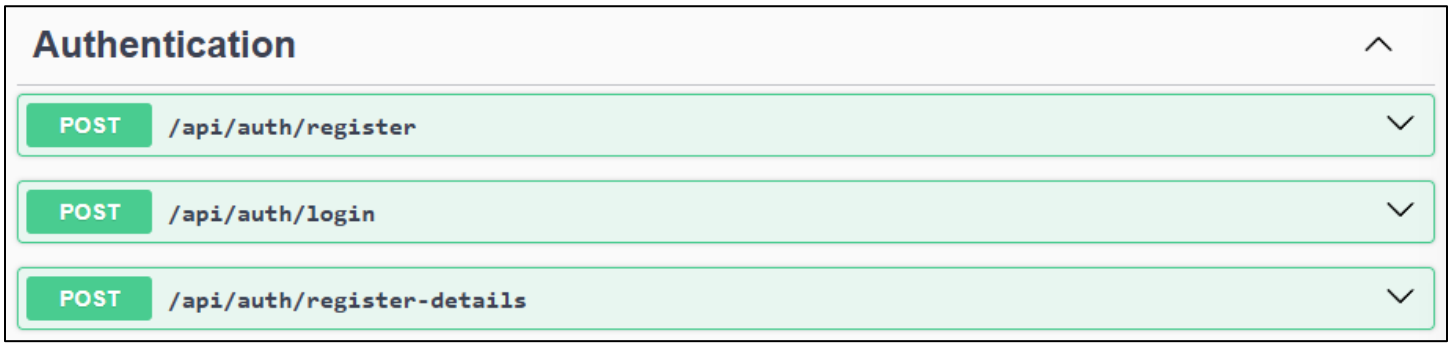
Дефинира функционалност обвързана с персоналното менажиране на вече съществуващ профил от страна на потребителя:



- **/api/account/add-photo**: потребителят има възможността да добави снимка
- **/api/account/profile-pic**: потребителят може да зададе нова профилна снимка, като избере от вече съществуващите такива в неговия профил
- **/api/account/remove-pic**: потребителят задава снимка, която да изтрие от своя профил
- **/api/account/update**: потребителят запазва промени, които е направил по своя профил

Authentication

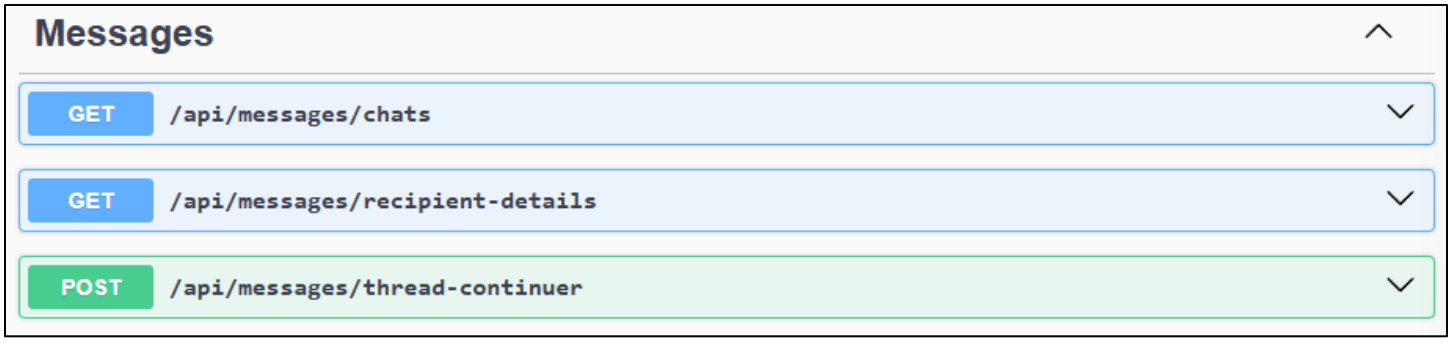
Дефинира функционалностите относно верификацията на потребителите:



- **/api/auth/register**: позволява на потребителите да регистрират своя акаунт
- **/api/auth/login**: позволява на потребителите да достъпът апликацията чрез вход за достъп
- **/api/auth/register-details**: менажира регистрационния flow относно съставянето на акаунта и неговата верификация

Messages

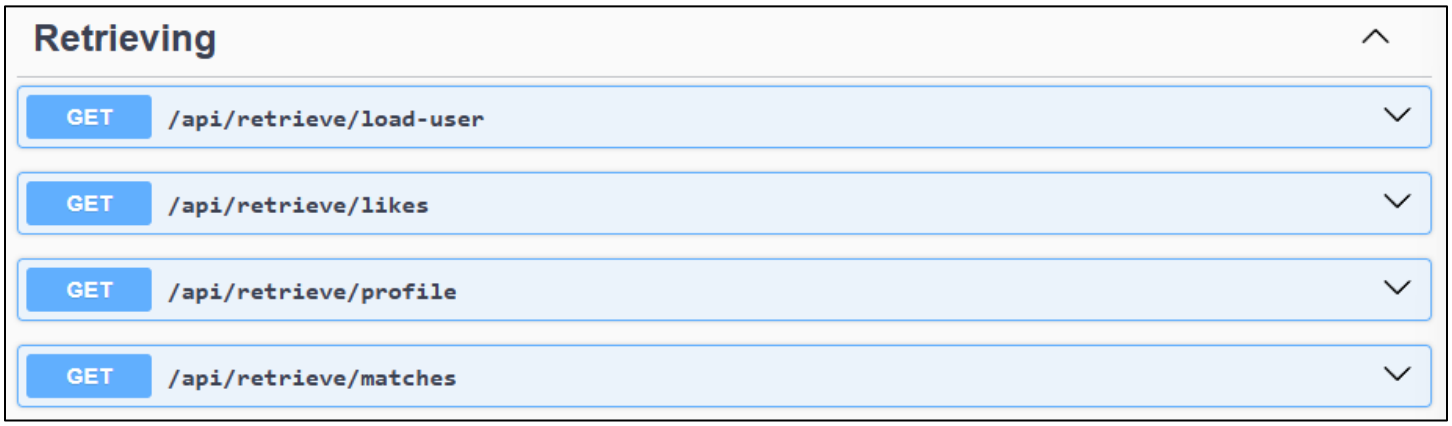
Предоставя допълнителна надградена функционалност към Messages Hub-а, изграден посредством SignalR.



- **/api/messages/chats**: предоставя всички чатове, които дадения потребител има, както и inbox нотификациите им.
- **/api/messages/recipient-details:** предоставя детайли за профила на събеседника си, с когото дадения потребител провежда комуникация в чата.
- **/api/messages/thread-continuer:** предоставя следващата част от съобщенията (по-старите такива постепенно), които фигурират в даден чат

Retrieving Compiler

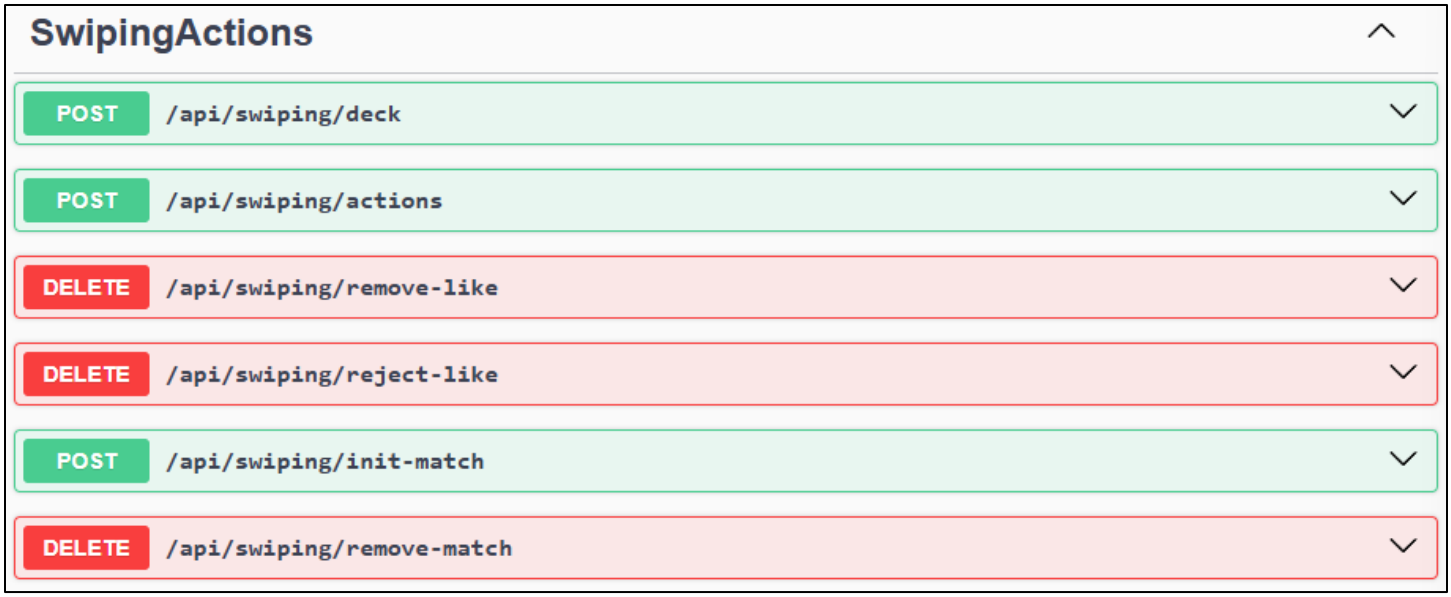
Съставя механизъм за събирането и предоставянето на данни под определен формат:



- **/api/retrieve/load-user**: предоставя базовите данни за потребителя при стартирането на апликацията
- **/api/retrieve/likes**: предоставя колекциите (архив) от like-ове на дадения потребител
- **/api/retrieve/profile**: зарежда пълните данни за даден потребител относно неговия профил
- **/api/retrieve/matches**: предоставя колекцията от съвпадения (matches), които дадения потребител притежава

Swiping Engine

Централния контролер, отговорен за осъществяването на интеракциите между потребителите посредством dating swiper-а.



- **api/swiping/deck**: съставя тестето от профили на база преференцията на текущия потребител
- **/api/swiping/actions**: реализира интеракциите на текущия потребител спрямо неговото заредено тесте
- **/api/swiping/remove-like**: премахва like от архива по усмотрение на текущия потребител
- **/api/swiping/reject-like**: текущия потребител отказва like предоставен му от друг потребител
- **/api/swiping/init-match**: даден потребител създава match на база получен like от друг
- **/api/swiping/remove-match**: изтриване на match по желание на потребителя, като това изтрива и всички негови съобщения със същия този потребител

5.9 Presence Hub (Статус за активност из между потребителите)

Компонентите в пространството Server.SignalR осигуряват функционалност за проследяване на присъствието на потребители в реално време в рамките на приложението. Системата е изградена около два основни класа: PresenceTracker, който управлява състоянието на онлайн потребителите, и PresenceHub, който служи като SignalR комуникационен център за известяване на клиентите относно промени в присъствието.

Presence Tracker

PresenceTracker е **централизирано, thread-safe хранилище (in-memory)**, което поддържа информация кои потребители (userId) са онлайн и списък с техните активни SignalR връзки (connectionId). Той позволява един потребител да има множество връзки едновременно.

- **UserConnected**: Регистрира нова връзка за потребител. Сигнализира (true), ако това е първата връзка и потребителят става онлайн.
- **UserDisconnected**: Премахва връзка. Сигнализира (true), ако това е последната връзка и потребителят става офлайн.

- **GetOnlineUsers**: Връща списък с ID-тата на всички текущо онлайн потребители.
- **GetUserConnections**: Връща списъка с връзки за конкретен потребител.

```
namespace Server.SignalR
{
    [Authorize]
    4 references
    public class PresenceHub : Hub
    {
        private readonly PresenceTracker _tracker;
        0 references
        public PresenceHub(PresenceTracker tracker)
        {
            _tracker = tracker;
        }

        0 references
        public override async Task OnConnectedAsync()
        {
            var userId = Context.User?.ExtractUserId();

            if(userId == null)
            {
                return;
            }

            var isOnline = await _tracker.UserConnected((Guid)userId, Context.ConnectionId);

            if(isOnline)
            {
                await Clients.Others.SendAsync("UserIsOnline", userId);
            }

            var onlineUsers = await _tracker.GetOnlineUsers();
            await Clients.Caller.SendAsync("GetOnlineUsers", onlineUsers);
        }
```

```
        0 references
        public override async Task OnDisconnectedAsync(Exception exception)
        {
            var userId = Context.User?.ExtractUserId();

            if (userId == null)
            {
                await base.OnDisconnectedAsync(exception);
                return;
            }

            var isOffline = await _tracker.UserDisconnected((Guid)userId, Context.ConnectionId);

            if(isOffline)
            {
                await Clients.Others.SendAsync("UserIsOffline", userId);
            }

            await base.OnDisconnectedAsync(exception);
        }
    }
}
```

Presence Hub

PresenceHub е SignalR Hub, който използва PresenceTracker за уведомяване на клиентите за промени в присъствието.

- OnConnectedAsync: При свързване на клиент, регистрира връзката му в PresenceTracker. Ако потребителят току-що е станал онлайн, уведомява останалите клиенти (UserIsOnline). Изпраща на свързващия се клиент списъка с всички онлайн потребители (GetOnlineUsers).
- OnDisconnectedAsync: При прекъсване на връзка, я от-регистрира от PresenceTracker. Ако потребителят току-що е станал офлайн, уведомява останалите клиенти (UserIsOffline).

```
namespace Server.SignalR
{
    [Authorize]
    4 references
    public class PresenceHub : Hub
    {
        private readonly PresenceTracker _tracker;
        0 references
        public PresenceHub(PresenceTracker tracker)
        {
            _tracker = tracker;
        }

        0 references
        public override async Task OnConnectedAsync()
        {
            var userId = Context.User?.ExtractUserId();

            if(userId == null)
            {
                return;
            }

            var isOnline = await _tracker.UserConnected((Guid)userId, Context.ConnectionId);

            if(isOnline)
            {
                await Clients.Others.SendAsync("UserIsOnline", userId);
            }

            var onlineUsers = await _tracker.GetOnlineUsers();
            await Clients.Caller.SendAsync("GetOnlineUsers", onlineUsers);
        }
```

```
        0 references
        public override async Task OnDisconnectedAsync(Exception exception)
        {
            var userId = Context.User?.ExtractUserId();

            if (userId == null)
            {
                await base.OnDisconnectedAsync(exception);
                return;
            }

            var isOffline = await _tracker.UserDisconnected((Guid)userId, Context.ConnectionId);

            if(isOffline)
            {
                await Clients.Others.SendAsync("UserIsOffline", userId);
            }

            await base.OnDisconnectedAsync(exception);
        }
    }
}
```

5.10 Messages Hub

Класът MessageHub управлява комуникацията в реално време за **директни съобщения между двама потребители**. Той използва SignalR групи за изолиране на чатове и си взаимодейства с базата данни отново през дефинирания Data Access Layer, а именно UnitOfWork, като се уповава и на логиката предоставена от PresenceTracker и PresenceHub за цялостно управление на съобщенията и известията.

- **Управление на Групи:** При свързване (OnConnectedAsync), hub-ът идентифицира двамата участници в чата (единият от контекста, другият от query параметър userId). Създава уникален идентификатор за групата (ComposeGroupName) и добавя връзката на потребителя към нея както в SignalR (Groups.AddToGroupAsync), така и в базата данни (AddToMessageGroup). При прекъсване на връзка (OnDisconnectedAsync), записът за връзката се премахва от базата данни (RemoveFromMessageGroup).
- **Зареждане на Чат:** При свързване, hub-ът извлича най-актуалната историята на съобщенията между двамата потребители (_unit.messageRepository.GetMessageThread), маркира всички непрочетени съобщения към свързващия се потребител като прочетени в базата данни и изпраща цялата история на съобщенията на клиента (ReceiveMessageThread). Също така уведомява другия потребител в групата, че събеседникът му е влязъл в чата (InterlocutorEnteredChat).

```
namespace Server.SignalR
{
    2 references
    public class MessageHub : Hub
    {
        private readonly IHubContext<PresenceHub> _pressenceHub;
        private readonly PresenceTracker _tracker;
        private readonly IUnitOfWork _unit;

        0 references
        public MessageHub(IHubContext<PresenceHub> pressenceHub, PresenceTracker tracker, IUnitOfWork unit)
        {
            _pressenceHub = pressenceHub;
            _tracker = tracker;
            _unit = unit;
        }
```

```
        0 references
        public override async Task OnConnectedAsync()
        {
            var coreInterlocutor = Context.User?.ExtractUserId();

            if(coreInterlocutor == null)
            {
                return;
            }

            var httpContext = Context.GetHttpContext();
            var otherInterlocutor = httpContext?.Request.Query["userId"].ToString();
            var pageIndex = httpContext?.Request.Query["pageIndex"].ToString();
            var pageSize = httpContext?.Request.Query["pageSize"].ToString();

            var groupIdentifier = ComposeGroupName((Guid)coreInterlocutor, Guid.Parse(otherInterlocutor));
            await Groups.AddToGroupAsync(Context.ConnectionId, groupIdentifier);
            var group = await AddToMessageGroup(groupIdentifier, (Guid)coreInterlocutor);
            await Clients.OthersInGroup(groupIdentifier).SendAsync("InterlocutorEnteredChat", coreInterlocutor);

            var messages = await _unit.messageRepository.GetMessageThread((Guid)coreInterlocutor, Guid.Parse(otherInterlocutor), int.Parse(pageIndex), int.Parse(pageSize));

            var unreadMessages = messages
                .Where(m => m.RecipientId == coreInterlocutor && !m.isRead)
                .ToList();

            foreach(var message in unreadMessages)
            {
                message.isRead = true;
                message.DateRead = DateTime.UtcNow;

                _unit.messageRepository.Update(message);
            }

            await _unit.SaveTransaction();

            await Clients.Caller.SendAsync("ReceiveMessageThread", messages);
        }
```

```
0 references
public override async Task OnDisconnectedAsync(Exception exception)
{
    var group = await RemoveFromMessageGroup();

    await base.OnDisconnectedAsync(exception);
}
```

- **Изпращане на Съобщения (SendMessage):** Когато потребител изпрати съобщение:
    1. Съобщението се записва в базата данни.
    2. Проверява се дали получателят е **в момента в същата SignalR група** (чат прозорец). Ако да, съобщението се маркира като прочетено веднага в базата данни.
    3. Ако получателят *не е* в групата, но е **онлайн някъде другаде** (проверено чрез PresenceTracker), му се изпраща известие за ново съобщение чрез PresenceHub (NewMessageReceived).
    4. Съобщението се изпраща до **всички членове на SignalR групата** (NewMessage).
- **Изтриване на Съобщения (DeleteMessage):** Когато потребител изтрие съобщение:
    1. Съобщението се маркира като изтрито от подателя в базата данни (soft delete).
    2. Подобно на изпращането, ако получателят *не е* в групата, но е онлайн, му се изпраща известие за изтриването чрез PresenceHub (DeletionMessageReceived).
    3. Актуализираното (изтрито) съобщение се изпраща до **всички членове на SignalR групата** (DeletedMessage).

```
0 references
public async Task SendMessage(SentMessagePayload sentMessage)
{
    var senderId = Context.User?.ExtractUserId();

    if (senderId == null)
    {
        return;
    }

    var message = new Models.Message
    {
        Id = new Guid(),
        SenderId = (Guid)senderId,
        RecipientId = sentMessage.RecipientId,
        Content = sentMessage.MessageContent,
        isRead = false,
        DateRead = null,
        MessageSent = DateTime.UtcNow,
        IsDeletedBySender = false
    };

    _unit.messageRepository.Add(message);

    var groupIdentifier = ComposeGroupName((Guid)senderId, sentMessage.RecipientId);
    var group = await _unit.groupRepository.Get(g => g.Identifier == groupIdentifier, "Connections");

    if(group == null)
    {
        return;
    }
```

```
    if(group.Connections.Any(g => g.UserId == sentMessage.RecipientId))
    {
        message.isRead = true;
        message.DateRead = DateTime.UtcNow;
    }
    else
    {
        var connections = await _tracker.GetUserConnections(sentMessage.RecipientId);
        if(connections != null)
        {
            await _pressenceHub.Clients.Clients(connections).SendAsync("NewMessageReceived", message);
        }
    }

    await _unit.SaveTransaction();
    await Clients.Group(groupIdentifier).SendAsync("NewMessage", message);
}
```

```
0 references
public async Task DeleteMessage(DeleteMessagePayload deletedMessage)
{
    var senderId = Context.User?.ExtractUserId();

    if (senderId == null)
    {
        return;
    }

    var messageToDelete = await _unit.messageRepository.Get(mess => mess.Id == deletedMessage.MessageId);

    if (messageToDelete == null) return;

    messageToDelete.IsDeletedBySender = true;
    _unit.messageRepository.Update(messageToDelete);

    var groupIdentifier = ComposeGroupName((Guid)senderId, deletedMessage.RecipientId);
    var group = await _unit.groupRepository.Get(g => g.Identifier == groupIdentifier, "Connections");

    if (group == null)
    {
        return;
    }

    if (!group.Connections.Any(g => g.UserId == deletedMessage.RecipientId))
    {
        var connections = await _tracker.GetUserConnections(deletedMessage.RecipientId);
        if (connections != null)
        {
            await _pressenceHub.Clients.Clients(connections).SendAsync("DeletionMessageReceived", messageToDelete);
        }
    }

    await _unit.SaveTransaction();
    await Clients.Group(groupIdentifier).SendAsync("DeletedMessage", messageToDelete);
}
```

6. Клиентска имплементация

Клиентската част на приложението е изградена в съответствие с принципите на **React Native**, с цел осигуряване на крос-платформена поддръжка и висока производителност. В основата ѝ стои **Root компонент App**, който играе централна роля в инициализацията и обвързването на всички останали части на интерфейса. Структурата на проекта е ясно организирана чрез **екрани (Screens)**, които дефинират основните визуални и функционални рамки на различните изгледи в приложението – като вход, регистрация, профил, съобщения и други.

Всеки от тези екрани е допълнително **декомпозиран на отделни компоненти**, отговарящи за по-малки и ясно дефинирани части от UI. Тази **йерархична структура** позволява по-добра **модулност, повторно използване на компоненти** и по-лесно поддържане. Благодарение на подобно разделение, всеки екран може да бъде развиван или рефакториран самостоятелно, като се запазва цялостната стабилност на приложението. Това дава възможност за целенасочена работа върху отделни елементи от интерфейса, като същевременно се запазва добра архитектурна последователност в рамките на цялата клиентска логика.

6.1 App Root. Definition of Root Navigation

Компонентът App стои в сърцето на клиентската архитектура и има ключова роля в определянето на **началната навигационна логика** на приложението. Неговата основна задача е да създаде **базовата навигационна рамка**, използвайки навигационен стек (stack-based навигация) и условни маршрути, в зависимост от състоянието на аутентикация на потребителя. При стартиране, App проверява за **наличен token за достъп (authentication token)** – ако такъв съществува, потребителят автоматично се пренасочва към **реалната,**

функционална част на приложението (като Swiping, Messages, Profile и др.). В противен случай, приложението го води през **началния Onboarding/Welcome flow**, включително Login и Register екрани.

Този подход демонстрира правилна **реализация на User Flow**, осигурявайки безпроблемен и логичен преход между етапите на използване на приложението – от първоначален достъп до активна интеракция. Така App компонентът не само служи като входна точка, но и като интелигентен навигационен контролер, който определя посоката на потребителското преживяване.

```
const Stack = createStackNavigator();

export default function App() {
  const [initialToken, setInitialToken] = useState("");
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const checkForExistingToken = async function () {
      setLoading(true);
      try {
        const storedToken = await SecureStore.getItemAsync("token");
        if (storedToken) {
          setInitialToken(storedToken);
        }
      } catch (error) {
        setInitialToken("");
      } finally {
        setLoading(false);
      }
    };

    checkForExistingToken();
  }, []);
```

```
  return (
    <>
      <StatusBar
        backgroundColor={colors.primaryBackground}
        barStyle="dark-content"
      />
      <NavigationContainer>
        <Provider store={store}>
          {loading ? (
            <Loading />
          ) : (
            <Stack.Navigator
              initialRouteName={initialToken ? "app" : "welcome"}
              screenOptions={{
                headerShown: false,
              }}
            >
```

```
            <Stack.Screen name="welcome" component={Welcome} />
            <Stack.Screen
              name="authenticate"
              component={Authenticate}
              options={{ animation: "reveal_from_bottom" }}
            />
            <Stack.Screen
              name="preferences"
              component={Preferences}
              options={{ animation: "slide_from_right" }}
            />
            <Stack.Screen
              name="app"
              component={Application}
              options={{ animation: "fade" }}
            />
          </Stack.Navigator>
        )}
      </Provider>
    </NavigationContainer>
  </>
 );
}
```

6.2 Nesting Navigation Stacks. AppTabs

В рамките на навигационната логика на приложението са внедрени **под-навигации**, които осигуряват **смислово организирано и контекстуално ориентирано преминаване** между различните секции.

Един от ключовите примери е AppTabs, изграден чрез createBottomTabNavigator, който дефинира **долна таб навигация** между основните модули на приложението – Swiping (main), Likes, Messages и Profile. Всеки таб визуално се отличава чрез **динамична икона**, която се променя при фокусиране, като същевременно осигурява и **плавна потребителска анимация и стилово унифициран вид**. Това разделение позволява на потребителя **лесно и бързо да се ориентира** в основната структура на апликацията, а логиката зад самата навигация е организирана в съответствие с **React Navigation принципите** за модулност и четимост.

```
const AppTabsNavigator = createBottomTabNavigator<AppTabsParamList>();
const BaseNavigator = createStackNavigator<ApplicationStackParamList>();

const AppTabs: React.FC = function () {
  return (
    <AppTabsNavigator.Navigator
      initialRouteName="main"
      screenOptions={({ route }) => ({···
      })}
    >
      <AppTabsNavigator.Screen name="main" component={DatingSwiper} />
      <AppTabsNavigator.Screen name="likes" component={Likes} />
      <AppTabsNavigator.Screen name="messages" component={Messages} />
      <AppTabsNavigator.Screen name="profile" component={Profile} />
    </AppTabsNavigator.Navigator>
  );
};
```

6.3 Обща компонентна архитектура

Архитектурата на компонентите в приложението следва **React парадигмата**, като всеки компонент е реализиран чрез **функционални компоненти** (React.FC). Тези функции са изградени по утвърден модел, включващ три основни части. Първо, се дефинират **входните параметри (props)**, чрез които компонентът приема данни и callback функции за обработка. След това следва **функционалното тяло**, където се извършват логически и state-управлявани операции, като например използването на React hook-ове (useState, useEffect) за динамично поведение на компонента. Последната част е **return блокът**, в който се описва визуалната структура с помощта на базовите компоненти на **React Native**, като View, Text, Image, TextInput, Pressable и други.

Под самата компонентна функция обикновено се намира и **стиловият обект**, дефиниран чрез StyleSheet.create, който централизирано управлява **визуалното оформление** на елементите. Това разграничение между логика, структура и стил прави компонентите **лесни за поддръжка, рециклиране и модификация**, като същевременно допринася за четимостта и доброто структуриране на кода.

```
let iconSrc: any;

const CredentialInput: React.FC<{
  type: "name" | "email" | "password" | "confirm_password";
  label: string;
  inputValue: string;
  inputHandler: (
    credentialKey:
      | "name"
      | "email"
      | "password"
      | "confirm_password"
      | "terms_agreement",
    value: string
  ) => void;
}> = function ({ type, label, inputValue, inputHandler }) {
  const [isTextHidden, setIsTextHidden] = useState(false);

  useEffect(() => {
    type.includes("password") && setIsTextHidden(true);
  }, []);

  switch (type) {
    case "name":
      iconSrc = require("../../../assets/icons/profile.png");
      break;
    case "email":
      iconSrc = require("../../../assets/icons/email.png");
      break;
    case "password":
      iconSrc = require("../../../assets/icons/password.png");
      break;
    case "confirm_password":
      iconSrc = require("../../../assets/icons/password_check.png");
  }
```

```
  return (
    <View style={styles.container}>
      <Text style={styles.label}>{label}</Text>
      <Image source={iconSrc} style={styles.icon} />
      <TextInput
        style={styles.input}
        autoComplete="off"
        keyboardType={type === "email" ? "email-address" : "default"}
        secureTextEntry={isTextHidden}
        value={inputValue}
        onChangeText={(text) => inputHandler(type, text)}
        autoCapitalize="none"
      />
      {type.includes("password") && (
        <Pressable
          onPress={() => setIsTextHidden((prevState) => !prevState)}
          style={[
            styles.icon,
            { justifyContent: "center", alignItems: "center" },
          ]}
        >
          <Image
            style={{ width: "65%", height: "65%" }}
            source={
              isTextHidden
                ? require("../../../assets/icons/hidden.png")
                : require("../../../assets/icons/visible.png")
            }
          />
        </Pressable>
      )}
    </View>
  );
};
```

```
const styles = StyleSheet.create({
  container: {
    width: "100%",
    height: 47.5,
    margin: "auto",
    marginTop: 15,
    marginBottom: 15,
    borderWidth: 1.5,
    borderColor: colors.textSecondary,
    borderRadius: 12.5,
    position: "relative",
    overflow: "visible",
    flexDirection: "row",
    alignItems: "center",
    paddingLeft: 10.5,
    paddingRight: 9.5,
  },
  label: {
    position: "absolute",
    backgroundColor: colors.secondaryBackground,
    top: -12.5,
    left: 15,
    paddingLeft: 7.5,
    paddingRight: 7.5,
    color: colors.textSecondaryContrast,
    fontFamily: "hn_regular",
  },
```

6.4 Обща Screen архитектура

Screen компонентите в приложението следват същата архитектурна логика и синтаксис, утвърдени от React Native и използвани при изграждането на обикновени компоненти. И те са реализирани като **функционални компоненти**, изградени чрез React.FC синтаксис, включващи ясно разграничени секции за **параметри**, **функционално тяло** и **визуално структуриране чрез return**. Основната разлика обаче се крие в **смисловата роля**, която изпълняват в архитектурата на приложението — Screen-овете служат като **контейнерни модули**, в които се интегрират и композират различни по-малки компоненти, за да се изгради цялостен изглед или функционален фрагмент от потребителския интерфейс.

Това структурно разделение позволява всеки Screen да дефинира **собствен контекст и логика**, като например зареждане на данни, управление на състояние и взаимодействие с навигационната логика. Подобно на останалите компоненти, и тук стиловете са централизирано дефинирани чрез StyleSheet.create, което улеснява визуалната консистентност и поддръжката на кода. С тази организация Screen компонентите играят ключова роля за **структурирането на потребителския поток и навигацията в приложението**, като същевременно запазват високата модуларност и повторна използваемост на кода.

```
const Messages: React.FC = function () {
  const { token } = useCharmrSelector((state) => state.authentication);
  const [chats, setChats] = useState<ChatDetails[]>([]);
  const [isLoading, setIsLoading] = useState(false);

  const loadInbox = useCallback(async () => {
    setIsLoading(true);

    try {
      const response = await fetch(`${API_ROOT}/messages/chats`, {
        headers: {
          Authorization: `Bearer ${token}`,
          "Content-Type": "application/json",
        },
      });

      if (response.ok) {
        const responseData: { chats: ChatDetails[] } = await response.json();
        setChats(responseData.chats);
      }
    } catch (error) {
      console.error("Error loading inbox:", error);
    } finally {
      setIsLoading(false);
    }
  }, [token, isLoading]);

  useFocusEffect(
    useCallback(() => {
      loadInbox();
    }, [])
  );
```

```
  return (
    <View style={styles.screen}>
      <MessagesHeader />
      <View style={styles.main}>
        <FlatList
          data={chats}
          keyExtractor={(item) => item.interlocutorId}
          renderItem={({ item }) => <ChatPreview payload={item} />}
          onRefresh={loadInbox}
          refreshing={isLoading}
          ListEmptyComponent={
            <View style={styles.fallback}>
              <Text style={styles.fallback_title}>⚛ No Chats Yet</Text>
              <Text style={styles.fallback_message}>
                Your inbox is empty! 💌 Start a conversation by picking one of
                your matches and get to know them. Love could be just a message
                away! ❤️
              </Text>
            </View>
          }
        />
      </View>
    </View>
  );
};
```

6.5 Redux Store Implementation

За целите на **централизираното управление на състоянието** в приложението е имплементиран Redux Store, използвайки **Redux Toolkit** – съвременен и опростен начин за работа с Redux в среда като React Native. Store-ът служи като глобален източник на ключови данни в приложението и гарантира **последователност и контрол** върху тяхното управление, особено при динамични операции като аутентикация, потребителски настройки или връзка в реално време.

Структурата на Redux Store-а е **модулно организиран** чрез отделни **Slice-ове**, всеки от които отговаря за конкретен аспект от приложението. Това модулно разделение позволява ясно структуриране и по-лесна поддръжка на логиката, като същевременно поддържа висока степен на четимост и разделение на отговорностите.

```
import accountDataManagementSlice from "./slices/account";
import authenticationSlice from "./slices/authentication";
import detailsManagementSlice from "./slices/details";
import signalRManagementSlice from "./slices/signalR";

export const store = configureStore({
  reducer: {
    accountDataManager: accountDataManagementSlice,
    authentication: authenticationSlice,
    detailsManager: detailsManagementSlice,
    signalRDataManager: signalRManagementSlice,
  },
});

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;

export const useCharmrDispatch = () => useDispatch<AppDispatch>();
export const useCharmrSelector: TypedUseSelectorHook<RootState> = useSelector;
```

Например, authenticationSlice отговаря за управлението на токена за достъп на потребителя. В него са дефинирани **начални стойности**, **редуктори**, както и **асинхронни взаимодействия със SecureStore**, където токенът се съхранява сигурно. Така всеки Slice съдържа собствено локализирано състояние и действия, които могат да бъдат извиквани в различни части от приложението чрез персонализирани hook-ове като useCharmrDispatch и useCharmrSelector.

```
interface IAuthenticationState {
  token: string | null;
}

const initialState: IAuthenticationState = {
  token: null,
};

const authenticationSlice = createSlice({
  name: "authentication",
  initialState,
  reducers: {
    tokenFromStorageLoader: (state) => {
      state.token = SecureStore.getItem("token");
    },
    fetchedTokenAssigner: (state, action: PayloadAction<string>) => {
      state.token = action.payload;
      SecureStore.setItem("token", action.payload);
    },
    tokenCleaner: (state) => {
      state.token = null;
      SecureStore.deleteItemAsync("token");
    },
  },
});

export const { tokenFromStorageLoader, fetchedTokenAssigner, tokenCleaner } =
  authenticationSlice.actions;
export default authenticationSlice.reducer;
```

Цялата конфигурация на store-а се осъществява чрез configureStore, където отделните Slice-ове се комбинират в единен reducer. По този начин се постига **мащабируема и подредена структура**, която лесно може да бъде разширена с допълнителна логика или модули в бъдеще.

6.7 Комуникация със сървъра и взаимодействие с Redux Store

Комуникацията между клиентската част на приложението и сървъра се реализира посредством **асинхронни функции**, извиквани в контекста на отделните компоненти. Общият подход е **унифициран** и следва няколко основни стъпки: инициализиране на заявката, изпращане на HTTP заявка към сървъра (с включени headers, най-често с Authorization токен), обработка на получената информация и последваща синхронизация със състоянието в Redux Store.

Всеки компонент, който комуникира със сървъра, следва този модел, като **отговорът от сървъра бива обработван в зависимост от използвания endpoint** и неговото значение за

текущия потребителски flow. Това осигурява гъвкавост при работа с различни типове данни, но запазва структурираността и предвидимостта на кода.

```
const loadUser = async function () {
  setLoadingState(true);
  try {
    const response = await fetch(`${API_ROOT}/retrieve/load-user`, {
      headers: {
        Authorization: `Bearer ${token}`,
      },
    });

    switch (response.status) {
      case 200:
        const responseData: ILoadUserRes = await response.json();
        if (responseData.userData.credentials.verificationStatus) {
          dispatch(filterInitiliazer(responseData.userFiltering));
          dispatch(userDataInitializer(responseData.userData));
        } else {
          navigation.navigate("preferences");
        }
        break;
      case 401:
        setModalVisibility(true);
    }
  } catch (error) {
    console.log(error);
  } finally {
    setLoadingState(false);
  }
};
```

Пример за подобен подход е функцията loadUser, която:

1. Инициализира **визуално състояние на зареждане** чрез setLoadingState(true).
2. Извършва fetch заявка към зададен endpoint, включвайки **токена за аутентикация** в заглавието.
3. На база HTTP статуса предприема различни действия – при успешен отговор (200), отговорът се **parsed до обект** и съответните данни се изпращат към Redux Store чрез dispatch. Така се гарантира **централизирано съхранение и достъп** до потребителските данни в останалите части на приложението.
4. В случаите на определени статуси (напр. 401), могат да се предприемат действия като визуализация на modal или навигация към различен екран.
5. След приключване на операцията – независимо от резултата – състоянието на зареждане се прекратява чрез setLoadingState(false).

6.8 SignalR Service

За осигуряване на **реално-времева комуникация** между клиентското приложение и сървъра, в архитектурата е внедрен модул SignalRService, който използва **SignalR протокола на ASP.NET Core**. Той представлява самостоятелен клас, отговорен за инициализация, поддръжка и управление на двете основни SignalR хъб връзки: **Presence Hub** и **Message Hub**.

Основна логика и структура

Класът SignalRService поддържа две отделни хъб връзки:

- presenceHubConnection – следи **онлайн статуса на потребителите**.
- messageHubConnection – управлява **чат комуникацията** между потребители.

Всеки хъб се инициализира с accessToken, използван за автентикация на връзката със сървъра. След успешна свързаност, към всяка връзка се добавят **събитийни слушатели (event listeners)**, които очакват входящи съобщения или действия от сървърната страна. Получените данни се предават директно към **Redux store** чрез dispatch функции, осигурявайки централизирано управление и реактивност на приложението спрямо входящата информация.

Примери за събития в Presence Hub:

- **UserIsOnline** – добавя ID на потребител, който е онлайн.
- **UserIsOffline** – премахва ID на потребител, който е офлайн.
- **GetOnlineUsers** – запазва текущите онлайн потребители.

В Message Hub логиката е по-богата, тъй като включва:

- Получаване и визуализиране на нови съобщения.
- Обновяване на цялата нишка на комуникация.
- Синхронизация на прочетени съобщения и изтривания.

Използване в компонентите

SignalRService се инстанцира чрез useRef, за да бъде **персистентен през целия жизнен цикъл на компонента**, а при useEffect:

1. Се инициализира връзката с конкретния хъб.
2. Се предава token, потребителски ID и callback за loading състояние.
3. В return на ефекта се прекратява хъб връзката чрез .terminateMessageHub() за предотвратяване на нежелано поведение и утечки на ресурси.

Предимства на тази имплементация:

- **Инкапсулираност** – цялата логика е капсулирана в един клас, улесняваща повторната ѝ употреба.
- **Гъвкавост** – позволява работа с различни хъбове чрез различни методи.
- **Интеграция с Redux** – осигурява моментално обновяване на UI състоянието при всяка нова активност от сървъра.

```
class SignalRService {
  private presenceHubConnection: HubConnection | null = null;
  private messageHubConnection: HubConnection | null = null;
  private dispatch: Function;

  constructor(dispatch: Function) {
    this.dispatch = dispatch;
  }
```

```
  initPressenceHub(token: string) {
    this.presenceHubConnection = new HubConnectionBuilder()
      .withUrl(`${API_ROOT}/hubs/presence`, {
        accessTokenFactory: () => token,
      })
      .withAutomaticReconnect()
      .configureLogging(LogLevel.Information)
      .build();

    this._startPressenceHubConnection();

    this.presenceHubConnection.on("UserIsOnline", (userId) => {
      this.dispatch(onlineUserAppender(userId));
    });

    this.presenceHubConnection.on("UserIsOffline", (userId) => {
      this.dispatch(onlineUserRemover(userId));
    });

    this.presenceHubConnection.on("GetOnlineUsers", (users) => {
      this.dispatch(onlineUsersSaver(users));
    });
  }

  initMessageHub(
    interlocutorId: string,
    token: string,
    onSetLoadingState: React.Dispatch<React.SetStateAction<boolean>>,
    pageIndex: number = 1,
    pageSize: number = 25
  ) {
    this.messageHubConnection = new HubConnectionBuilder()
      .withUrl(
        `${API_ROOT}/hubs/message?userId=${interlocutorId}&pageIndex=${pageIndex}&pageSize=${pageSize}`,
        {
          accessTokenFactory: () => token,
        }
      )
      .withAutomaticReconnect()
      .configureLogging(LogLevel.Information)
      .build();

    this._startMessageHubConnection(onSetLoadingState);

    this.messageHubConnection.on("newMessage", (message) => {
      this.dispatch(newMessageAppender(message));
    });
```

```
  private async _startPressenceHubConnection() {
    try {
      await this.presenceHubConnection?.start();
      console.log(
        "Connection to the Precence Service was established successfully."
      );
    } catch (error) {
      console.log("Couldn't connect to the Precence Service.\nError: ", error);
    }
  }

  private async _startMessageHubConnection(
    onSetLoadingState: React.Dispatch<React.SetStateAction<boolean>>
  ) {
    try {
      onSetLoadingState(true);
      await this.messageHubConnection?.start();
      console.log(
        "Connection to the Message Service was established successfully."
      );
    } catch (error) {
      console.log("Couldn't connect to the Message Service.\nError: ", error);
    } finally {
      onSetLoadingState(false);
    }
  }
}
```

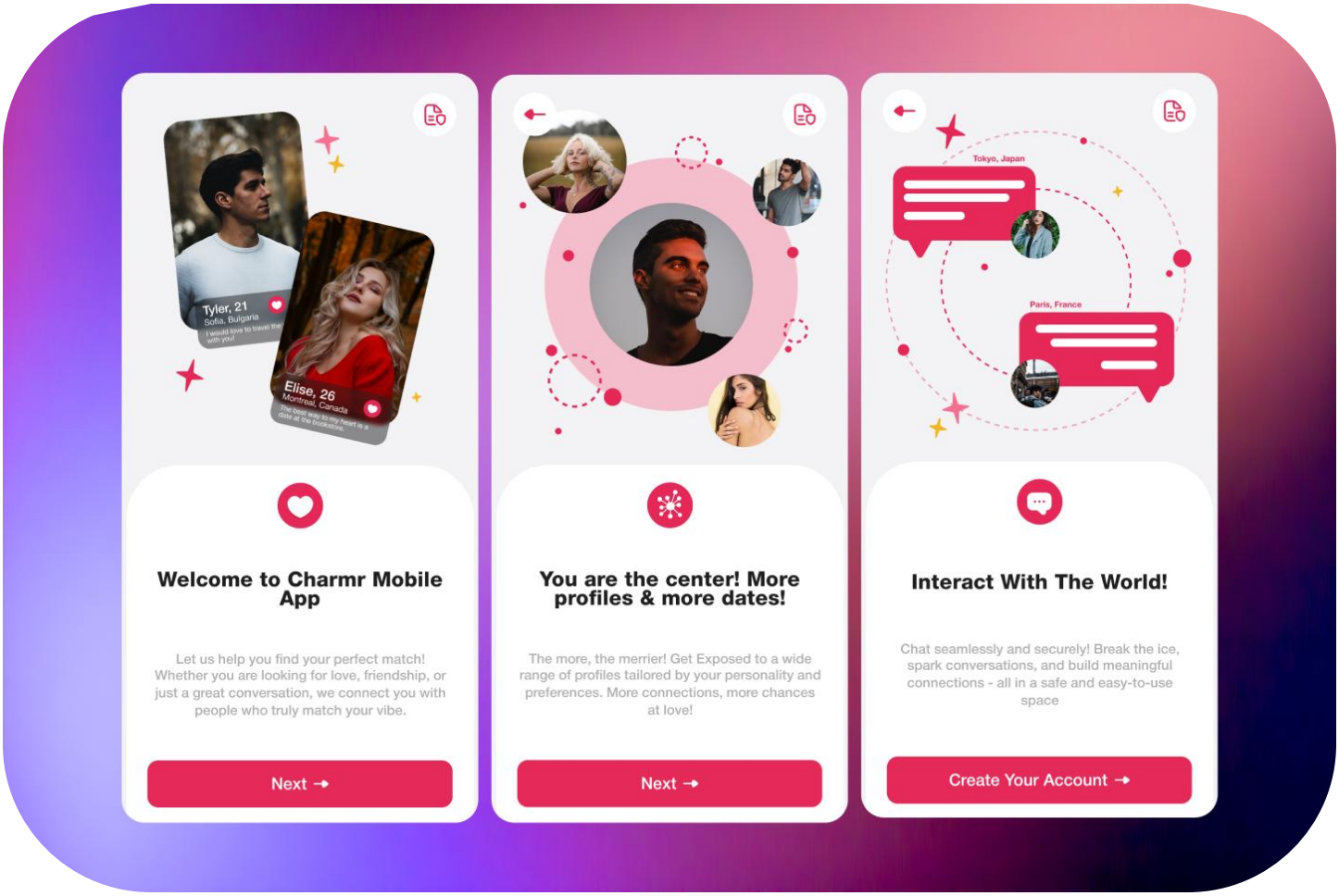
```
sendMessage(recipientId: string, messageContent: string) {
  if (this.messageHubConnection) {
    this.messageHubConnection.invoke("SendMessage", {
      recipientId,
      messageContent,
    });
  } else {
    console.log("SignalR connection is not established.");
  }
}

deleteMessage(messageId: string, recipientId: string) {
  if (this.messageHubConnection) {
    this.messageHubConnection.invoke("DeleteMessage", {
      messageId,
      recipientId,
    });
  } else {
    console.log("SignalR connection is not established.");
  }
}

terminatePresenceHub() {
  if (this.presenceHubConnection) {
    this.presenceHubConnection.stop();
  }
}

terminateMessageHub() {
  if (this.messageHubConnection) {
    console.log("Message Hub Terminated");
    this.messageHubConnection.stop();
    this.dispatch(threadPaginationParamsInitializier());
  }
}
```

7. Потребителски интерфейс (Ръководство)

7.1 Welcome Screen

Welcome екранът представлява първия досег на потребителя с приложението Charmr и играе ключова роля за създаване на силно първоначално впечатление. Основната му цел не е просто да насочи потребителя към регистрация или вход, а да предизвика интерес, емоционален отклик и доверие още от първата секунда.

Интерфейсът е изграден така, че да комуникира с потребителя визуално и концептуално – чрез атрактивен дизайн, свежи цветове и кратки, вдъхновяващи текстове, които подсказват стойността на Charmr като модерен инструмент за създаване на смислени връзки. Тук потребителят получава ясно послание: Charmr не е просто поредното приложение за запознанства, а платформа, която поставя личността, индивидуалността и автентичността в центъра на вниманието.

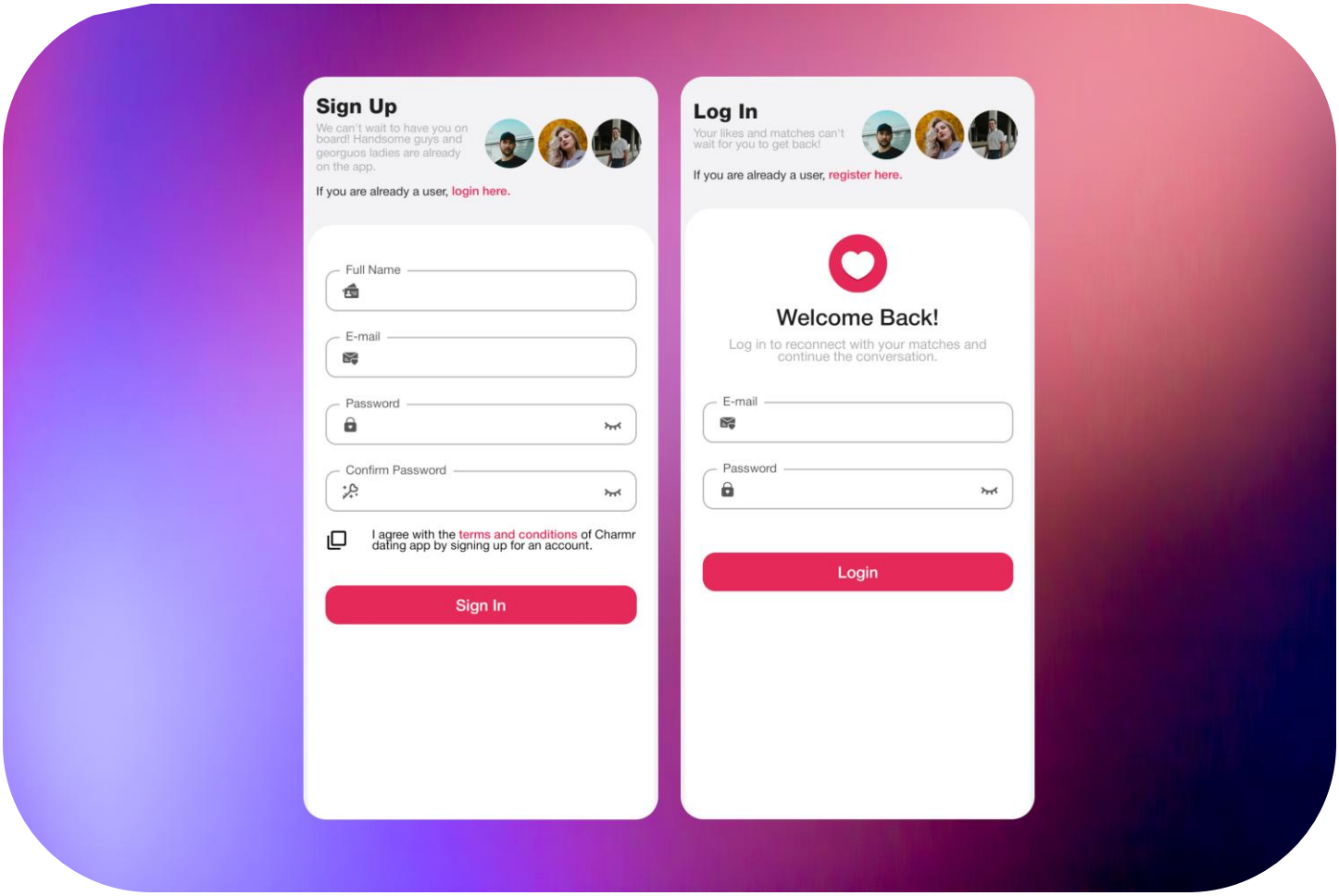


7.2 Registration/Login Screens

След първоначалния досег с приложението през **Welcome екрана**, потребителят бива насочен към един от двата основни входни пътя: **Login** или **Registration**.

Login екранът позволява на вече регистрирани потребители да влязат в своя профил. Интерфейсът тук е изчистен и интуитивен, като набляга на бърз достъп чрез имейл и парола, а при нужда – и опция за възстановяване на парола. Основната му цел е да осигури **бърза, сигурна и безпроблемна автентикация**.

Registration екранът е насочен към нови потребители, които желаят да създадат акаунт в платформата. От потребителя се изисква въвеждане на основна информация като имейл и парола, като процесът е оптимизиран за минимален *friction*. Този екран служи като **входна точка към по-задълбочения Registration Details Flow**.

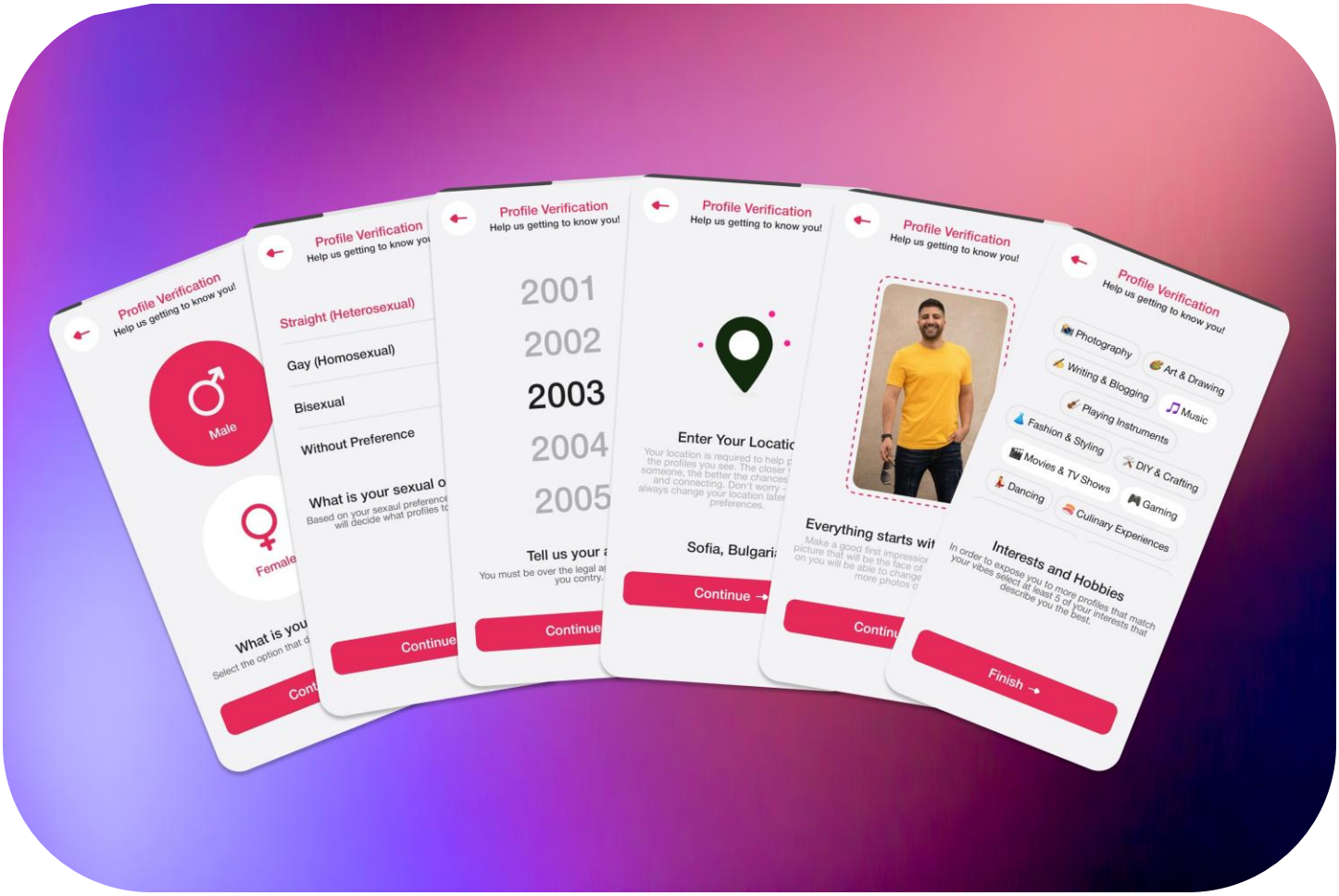


След успешна регистрация, потребителят преминава през **детайлизиран процес по създаване и верификация на профила**, който има ключова роля в качеството на съвпаденията и цялостното потребителско изживяване.

В този flow се събират допълнителни данни като:

- име, псевдоним и демографска информация
- интереси, предпочитания и сексуална ориентация
- геолокация
- профилна снимка и основни детайли

Тази стъпка изпълнява **двойна функция**: от една страна обогатява потребителския профил, а от друга – гарантира **сигурност и автентичност** на участниците в платформата. Само след успешна верификация, потребителят получава пълен достъп до приложението.

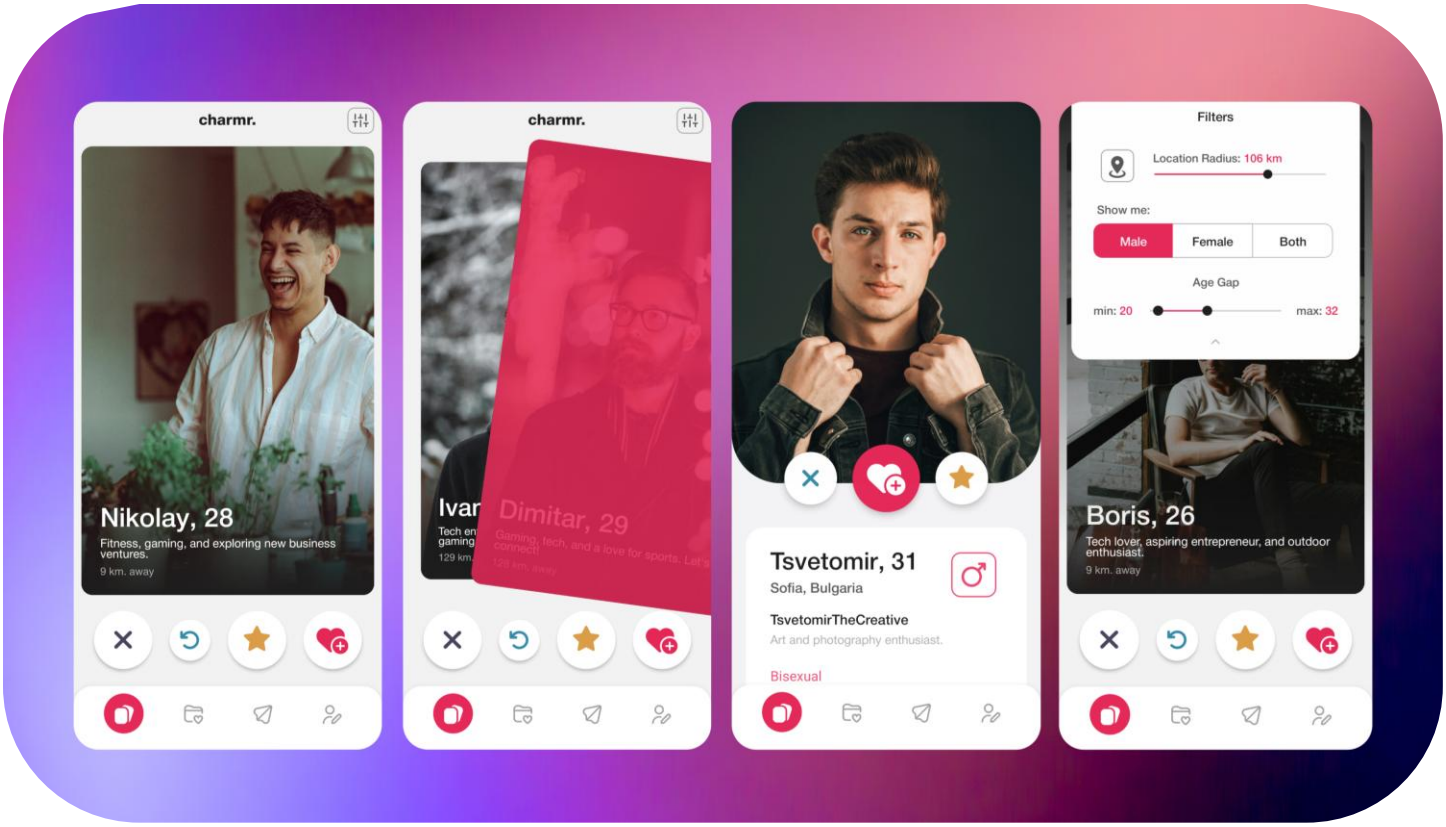


7.3 Dating Swiper Screen

Dating Swiper екранът представлява основното място, където потребителите взаимодействат с потенциални съвпадения. Интерфейсът следва предоставянето на тесте от с карти от потребителски профили, с които той може да упражнява интуитивна **"swipe" механика**, при която потребителят плъзга картите на профили надясно за интерес (like) или наляво за пропуск (pass). При взаимно харесване се създава **match**, който отключва чат функционалност.

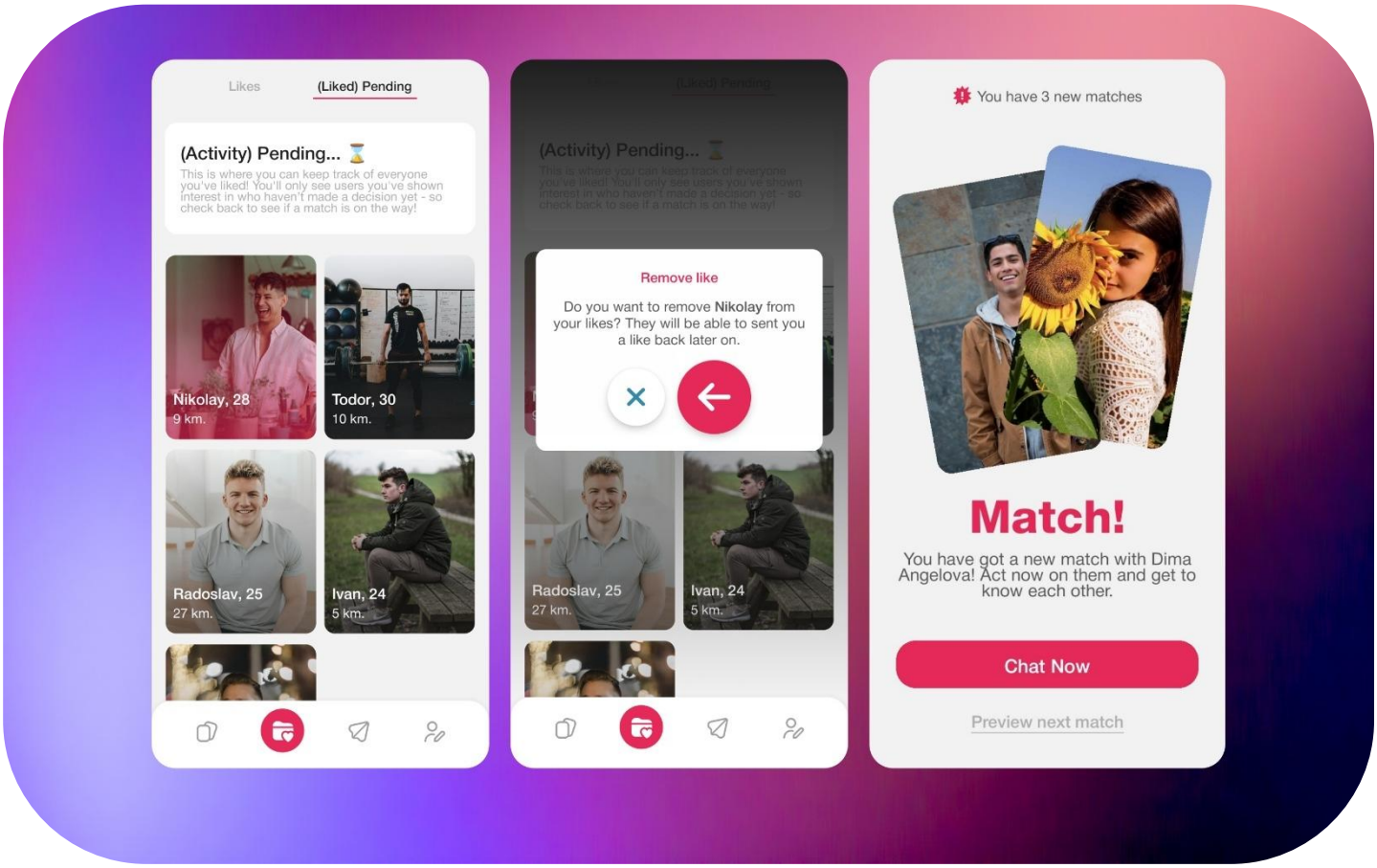
В допълнение към основния swipe, екранът предлага:

- **Възможност за филтриране** на профили според предпочитания (възраст, пол, разстояние и интереси), което позволява по-прецизно търсене.
- **Опция за преглед на пълния профил** на даден потребител (био, снимки, локация и допълнителни детайли) преди вземане на решение.
- Поддръжка на **Super Like** функция за изразяване на специален интерес.



7.4 Likes Activity Screen

Likes Archive предоставя на потребителя **централизирано място за преглед и управление на всички получени харесвания**, свързани с неговия профил. В този екран се



визуализират както обикновените likes, така и **Super like** реакциите, които се отличават чрез **розов градиент**, за да сигнализират по-силен интерес от другия потребител.

Потребителят има възможност:

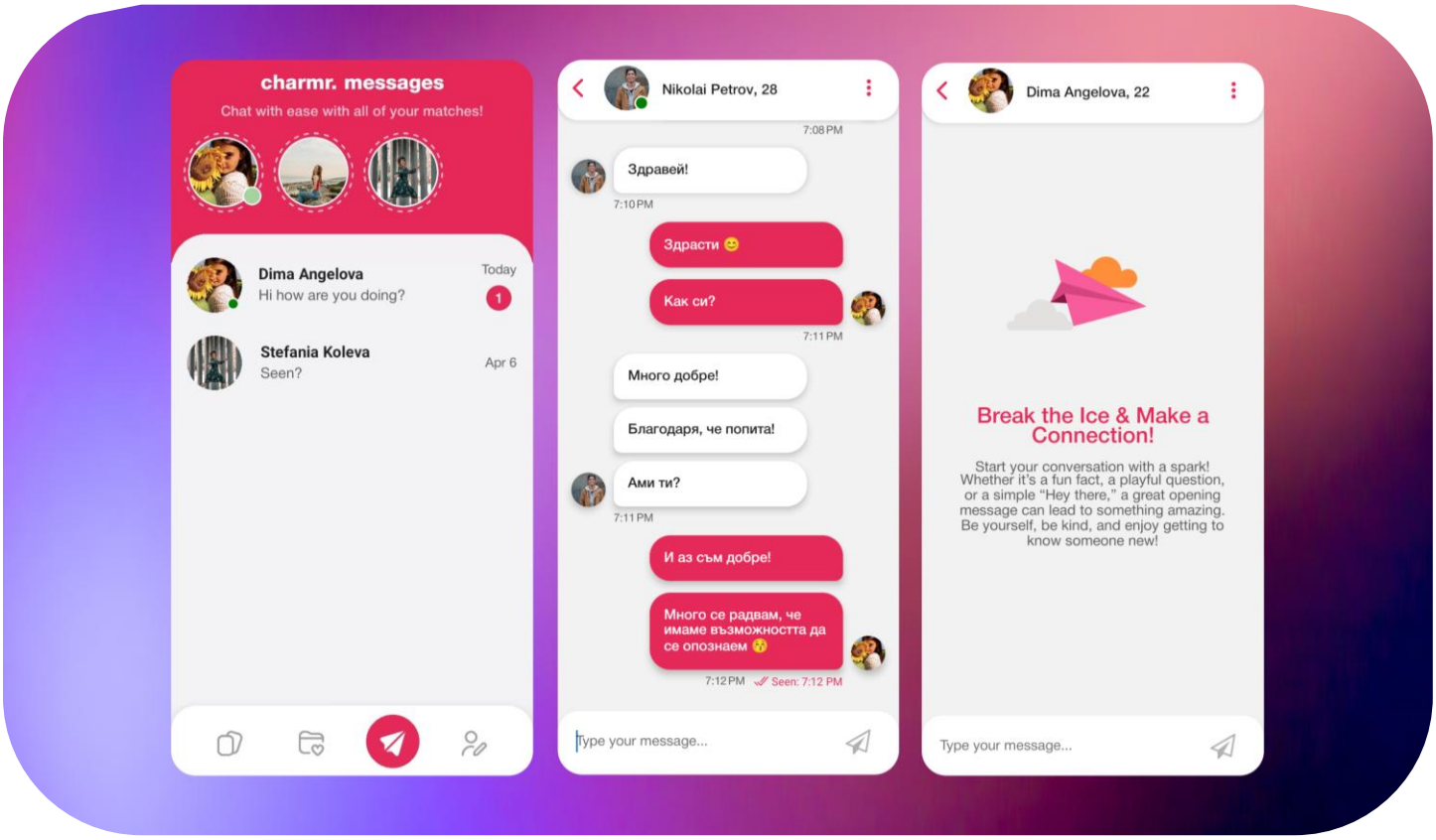
- Да преглежда историята на получените likes.
- Да предприема действия като харесване обратно (Match) или отхвърляне
- Да получава по-добър контрол и прозрачност върху потребителската си активност

Този екран е важен елемент от **ангажираността и обратната връзка** в платформата, като поддържа усещането за реална интеракция и динамика в процес а на запознанства.

7.5 Messages Screen

Messages екранът е комуникационният хъб на приложението, където потребителите могат да преглеждат, управляват и водят разговори с потребители**,** с които са **Match-нали**. Екранът е разделен функционално на три основни части:

- **Match селекция** – визуализира най-новите съвпадения в хоризонтален списък, улеснявайки бърз достъп до нови връзки.
- **Inbox** – списък с активни чатове, подредени по последно получено или изпратено съобщение. Позволява лесна навигация между разговори.
- **Chat Room интерфейс** – поддържа стандартна функционалност за чат: изпращане и получаване на текстови съобщения, визуализиране на състояния (напр. "прочетено") и плавна интеракция в реално време.

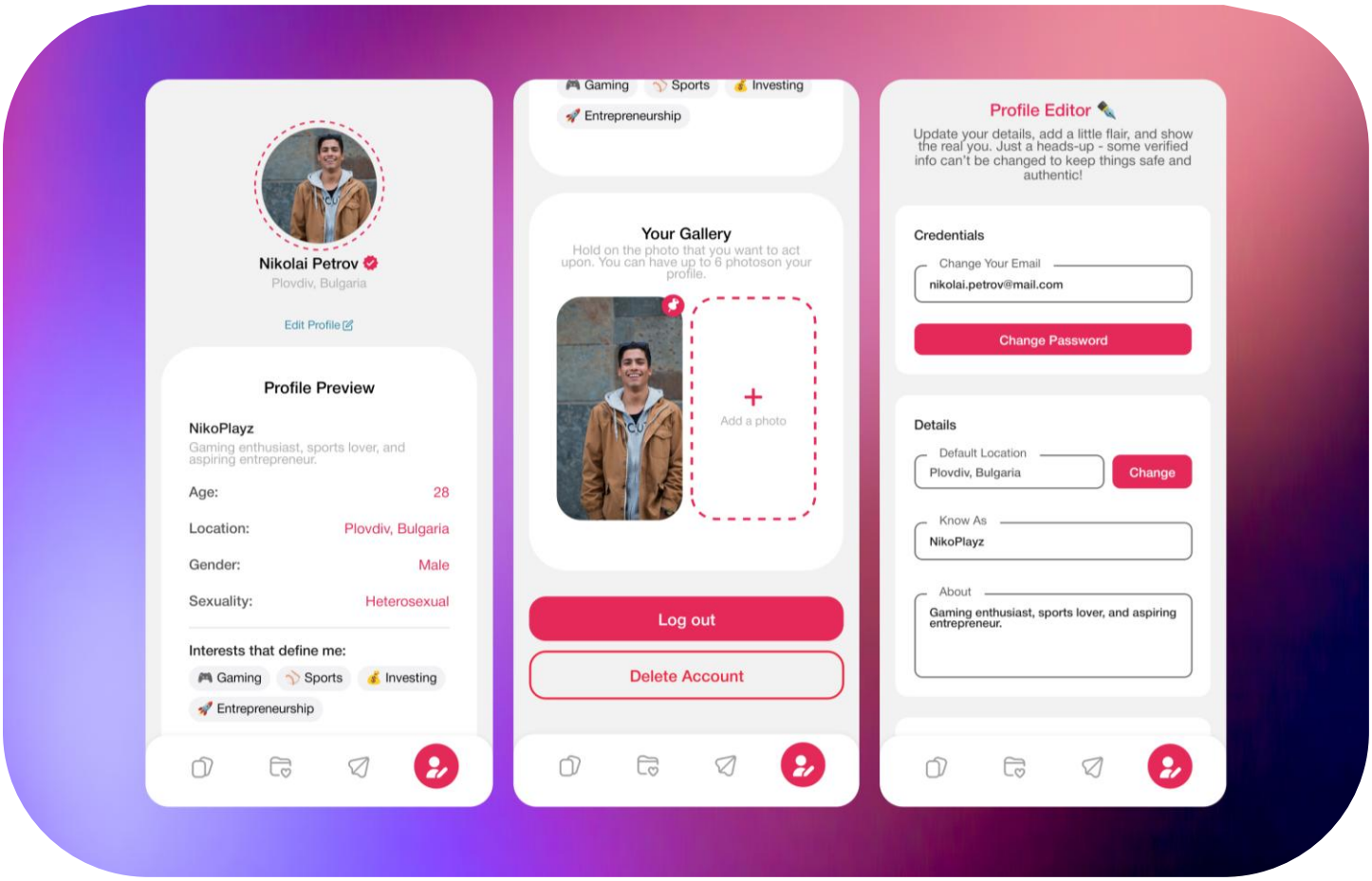


7.6 Profile Management Screen

Profile екранът предоставя на потребителя достъп до неговия собствен профил, като служи като **център за управление на личната информация и представяне в платформата**. Оттук потребителят може:

- Да преглежда и редактира основни данни като имейл, био и предпочитания.
- Да добавя, премахва или пренарежда профилни снимки.

Целта на екрана е да осигури **гъвкав контрол и персонализация**, така че всеки потребител да се представи автентично и в най-добрата си светлина. Интерфейсът е изчистен и лесен за навигация, като насърчава поддръжка на актуален и ангажиращ профил.



8. Място за подобрения, надграждане и възможни бъдещи промени

С цел непрекъснато подобряване на потребителското изживяване и функционалната стойност на приложението, са заложени редица бъдещи надграждания и подобрения. Те целят както разширяване на функционалността, така и допълнително повишаване на сигурността и ангажираността на потребителите:

- **По-добра интеракция през Dating Swiper**: Предвижда се развиването на по-динамични и ангажиращи механики за взаимодействие между потребителите в основната част на приложението – Swiper интерфейса. Целта е преживяването да бъде по-близко до реално социално взаимодействие и да бъде колкото се може по моментално.

- **Оптимизация на дизайна за различни устройства**: В ход е разработка по подобряване на responsiveness-а на интерфейса за различни екрани, включително поддръжка на **таблетна версия**, което ще разшири обхвата на използваемостта.
- **Имейл верификация и управление на акаунт**: Включва се реална възможност за **верифициране на имейл адреса** чрез външен email service. Наред с това ще се предостави възможност за **смяна на лични данни**, като имейл и парола, директно от профила на потребителя, чрез същия този email service.
- **Символична такса при регистрация**: С цел ограничаване на фалшиви профили и изграждане на по-сериозна потребителска база, се обмисля въвеждането на **еднократна такса до 5 лв. при регистрация**. Това ще помогне за естествено филтриране на ангажираните потребители.
- **Развитие на Like Archive секцията**: Ще се въведе **филтриране и търсене на потребители**, с цел по-лесна навигация и организация на историята от харесани профили.
- **Надграждане на чат функционалността**. Планира се въвеждане на:
    - Реакции към съобщения (emoji).
    - Възможност за изпращане на **снимки и видеа**.
    - Поддръжка на **гласов и видео чат** в реално време.

9. Заключение

Приложението е в завършен вид и успешно покрива всички базови функционалности, типични за съвременните социални и dating платформи. Реализирани са основните потребителски потоци – регистрация, автентикация, интеракция между профили, взаимодействие чрез Swipe механика, комуникация в реално време и управление на лични данни – като всяка от тези функционалности е изградена с внимание към потребителското изживяване и поддръжка на добра производителност.

Въпреки ограничените ресурси и краткия времеви обхват на разработката, архитектурният подход, използван в проекта, е модерен, устойчив и лесен за разширение. Използвани са актуални технологии и добри практики от React екосистемата, Redux Toolkit за управление на състоянието, както и SignalR за двупосочна комуникация в реално време. Това осигурява не само стабилна основа, но и възможност бъдещи бъгове и неточности да бъдат бързо локализирани и отстранени.

Важно е да се подчертае, че структурата на проекта е изградена така, че позволява **лесна интеграция на нови функционалности**, дори и от екип, който не е участвал в първоначалната разработка. Това прави решението не само завършено в настоящия си вид, но и **устойчиво в дългосрочен план**.

В този си етап приложението е напълно **готово за реална употреба**, като отговаря на нуждите на крайните потребители и разполага с потенциал да бъде конкурентоспособно спрямо съществуващите решения в нишата.