МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра информатики и программирования

РАЗРАБОТКА ПРИЛОЖЕНИЯ ДЛЯ ЗНАКОМСТВ С РЕКОМЕНДАТЕЛЬНОЙ СИСТЕМОЙ

БАКАЛАВРСКАЯ РАБОТА

студента 4 курса 441 группы
направления 02.03.03 — Математическое обеспечение и администрирование
информационных систем
профиль «Технологии программирования»
факультета КНиИТ
Уталиева Султана Едильбаевича

Научный руководитель
ст.преп. кафедры ИиП _____ Казачкова А. А.

Заведующий кафедрой
к. ф.-м. н., доцент _____ Огнева М. В.

Саратов 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние годы наблюдается устойчивый рост интереса к персонализированным цифровым сервисам, что обусловлено стремлением пользователей получать релевантный контент и улучшенный пользовательский опыт. Одним из ключевых инструментов персонализации являются рекомендательные системы — интеллектуальные алгоритмы, позволяющие адаптировать предложения под индивидуальные предпочтения. Их значение особенно велико в приложениях для знакомств, где качество рекомендаций напрямую влияет на успешность социальных взаимодействий и удовлетворённость пользователей [1, 2].

Современные приложения, такие как Tinder, OkCupid и Hinge, активно используют алгоритмы рекомендаций, включая коллаборативную фильтрацию, обучение представлений и гибридные методы, зачастую основанные на больших объёмах пользовательских данных и методах машинного обучения [3–5]. Однако многие существующие решения сталкиваются с проблемами в условиях дефицита данных — например, на ранних этапах использования приложения [6].

В данной работе рассматривается разработка мобильного приложения для знакомств, включающего в себя встроенную рекомендательную систему. В отличие от большинства существующих решений, предлагаемый подход предусматривает использование не только анкетных данных, но и результатов тестов-опросов, которые пользователи могут проходить по собственному желанию. Каждая карточка с вопросом позволяет выбрать один из трёх вариантов ответа — «да», «нет» или «пропустить», что позволяет формировать тернарные признаки $(-1, 0, 1)$, лежащие в основе профиля предпочтений пользователя.

Целью дипломной работы является разработка приложения для знакомств с рекомендательной системой, обеспечивающей релевантные и разнообразные рекомендации потенциальных партнёров на основе результатов тестирования. Для достижения этой цели решаются следующие задачи:

- анализ существующих подходов к построению рекомендательных систем в контексте мобильных приложений для знакомств;
- формализация задачи рекомендаций с учётом специфики представления пользовательских признаков;
- проектирование архитектуры приложения и рекомендательной системы;
- реализация рекомендательной подсистемы на основе методов контентной фильтрации и эвристических правил;

— разработка подхода к оценке качества рекомендаций с использованием оффлайн-метрик.

Практическая значимость работы заключается в создании масштабируемого мобильного решения, сочетающего в себе функции приложения для знакомств и интерпретируемой рекомендательной системы, адаптированной к условиям ограниченных вычислительных ресурсов и высокой динамики пользовательских предпочтений.

1 Анализ предметной области и существующих решений

1.1 Особенности рекомендательных систем

Рекомендательные системы являются неотъемлемой частью современных цифровых платформ, предоставляя пользователям персонализированные предложения продуктов, услуг или контента. Они находят широкое применение в различных областях, включая электронную коммерцию, стриминговые сервисы, социальные сети и онлайн-знакомства.

Современные рекомендательные сталкиваются с рядом проблем:

— отсутствие информации о новых пользователях и объектах (проблема холодного старта) [6];
— высокая разреженность пользовательско-объектных матриц [1];
— необходимость обеспечения справедливости и отсутствия предвзятости в рекомендациях [2].

Современные исследования направлены на преодоление этих ограничений, в том числе с использованием больших языковых моделей, методов объяснимого машинного обучения и расширения пользовательского контекста.

1.2 Обзор рекомендательных систем в сфере онлайн-знакомств

Онлайн-сервисы знакомств предъявляют особые требования к алгоритмам рекомендаций. Поскольку конечной целью является установление реального или виртуального контакта между людьми, системы должны максимально точно учитывать совместимость по широкому спектру признаков, при этом оставаясь достаточно лёгкими в вычислении и объяснимыми.

1.2.1 OkCupid

Платформа OkCupid применяет нетривиальный подход к построению рекомендаций, фокусируясь на глубоком анализе анкет пользователей и их ответов на вопросы. В отличие от многих систем, которые ориентируются лишь на поведение пользователей, OkCupid делает упор на содержательные признаки совместимости.

Пользователи проходят опросы, отвечая на вопросы о ценностях, привычках, интересах и взглядах. Для каждого вопроса пользователь:

— указывает свой собственный ответ;
— выбирает приемлемые для него ответы потенциального партнёра;
— определяет важность соответствия по этому вопросу.

Таким образом, для каждого пользователя можно получить богатый профиль предпочтений, включающий не только их мнения, но и ожидания от других.

Основой рекомендательной системы OkCupid служит собственная метрика совместимости, вычисляемая по формуле, напоминающей обобщённую версию взвешенного совпадения. Пусть пользователь A ответил на n вопросов, по которым можно сопоставить ответы с пользователем B. Тогда их совместимость вычисляется по следующей схеме:

1. Для каждого вопроса i определяется s_i^A — совпадает ли ответ B с приемлемыми ответами A; w_i^A — важность вопроса по мнению A.
2. Вычисляется доля удовлетворения B ожиданий A:

$$S_{AB} = \frac{\sum_{i=1}^{n} s_i^A \cdot w_i^A}{\sum_{i=1}^{n} w_i^A}$$

3. Аналогично считается S_{BA}.
4. Итоговая совместимость:

$$C(A, B) = \sqrt{S_{AB} \cdot S_{BA}}$$

Такой симметричный подход позволяет учитывать желания обеих сторон, что особенно важно в сфере знакомств [3].

Кроме ответов на вопросы, система также может учитывать:

— географическое положение пользователей;
— возраст и пол;
— поведенческие метрики (например, активность и отклики);
— интересы, указанные в профиле;
— алгоритмы коллаборативной фильтрации на основе лайков.

Однако основным источником данных остаются именно опросы, что выгодно отличает OkCupid от других платформ.

Рекомендательная система OkCupid строится на принципах симметричной оценки, учитывающей как личные предпочтения, так и взаимные ожидания. Модель сочетает элементы экспертных систем и идеи коллаборативной фильтрации, что позволяет выдавать персонализированные рекомендации, основанные на содержательных признаках.

Тем не менее, использование опросов как основной основы для рекомендаций имеет ряд ограничений:

— Самоотчетность. Пользователи могут отвечать неискренне, выбирая социально одобряемые ответы или предполагаемые предпочтения других, а не собственные.
— Ограниченность охвата. Даже при большом числе вопросов многие аспекты личности, поведения и совместимости остаются неохваченными.
— Неполные профили. Пользователи часто не отвечают на все доступные вопросы, что затрудняет построение точных рекомендаций.
— Фиксированные веса. Веса важности выставляются вручную, и пользователь может переоценить или недооценить значимость отдельных тем.
— Изменчивость во времени. Ответы могут устаревать, но система не всегда способна учитывать динамику изменения взглядов и предпочтений.

Таким образом, хотя подход OkCupid обладает значительной выразительной силой, его эффективность может снижаться при недостаточной мотивации пользователей честно и подробно заполнять анкету.

1.2.2 Tinder

Платформа Tinder делает акцент на скорости взаимодействия и минимализме в пользовательском интерфейсе. Это определяет и архитектуру рекомендательной системы: она почти полностью основана на поведенческих данных, а не на заранее структурированных опросах [7].

Рекомендации Tinder опираются на наблюдение за действиями пользователя в реальном времени, применяя идеи из области ранжирования, коллаборативной фильтрации и машинного обучения.

— Пользователь не заполняет анкету или тесты – основным источником сигналов становится поведение (свайпы, матчи, отклики).
— Цель системы – ранжировать потенциальных партнёров по вероятности положительного отклика.
— Взаимная симпатия (мэтч) служит основной меткой релевантности.

Ранее Tinder применял модификацию рейтинговой системы Elo, аналогичной той, что используется в шахматах. Каждому пользователю приписывался скрытый рейтинг R, который обновлялся при каждой паре действий:

$$R_A^{\text{new}} = R_A + K \cdot (S - E)$$

где:

— S — фактический результат (1, если A получил лайк от B, 0 — если дизлайк);
— E — ожидаемая вероятность положительного отклика, например:

$$E = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

— K — коэффициент обучения.

Со временем эта система была признана слишком статичной и неспособной учитывать сложные паттерны предпочтений.

В настоящее время Tinder использует более гибкий и масштабируемый подход на основе моделей обучения ранжированию и нейронных сетей. Ключевые особенности [5]:

1. Использование исторических свайпов как обучающего датасета.
2. Обогащение признаков за счёт:
   — времени суток, геолокации, возраста и пола;
   — информации о взаимодействиях (ответы в чатах, продолжительность общения);
   — изображений (модели компьютерного зрения извлекают визуальные эмбеддинги).
3. Применение моделей типа learning-to-rank, включая градиентный бустинг и глубокие нейросети.
4. Возможное использование sequence-based моделей (например, RNN или трансформеров), учитывающих порядок свайпов.

Хотя такой подход даёт хорошее качество персонализации, он не лишён ограничений:

— Холодный старт. Новым пользователям сложно получить релевантные рекомендации до накопления истории свайпов.
— Смещение данных. Пользователи чаще свайпают по привлекательности, а не по глубинной совместимости.
— Неустойчивость. Поведение может быть ситуативным, но алгоритм воспринимает его как предпочтение.
— Мало объяснимости. Модель сложно интерпретировать или объяснить пользователю, почему показан тот или иной профиль.

Модель рекомендаций Tinder эволюционировала от простой системы рейтингов к сложной системе поведенческого ранжирования. Она эффективно мас-

штабируется и адаптируется под предпочтения пользователя, но при этом страдает от отсутствия прозрачности и возможной поверхностности критериев совместимости [8].

1.2.3 Hinge

Платформа Hinge позиционирует себя как сервис для «удаления» приложения после нахождения подходящего партнёра. Это отражается и в её подходе к построению рекомендаций: модель ориентирована не на максимум свайпов, а на вероятность качественного взаимодействия. Рекомендательная система Hinge учитывает как поведенческие сигналы, так и контекстные данные, стремясь построить эффективные персонализированные предложения.

В основе лежат несколько ключевых принципов:

— Система стремится обучаться на успешных взаимодействиях — прежде всего на лайках, приводящих к продолжительным диалогам.
— Учитывается не только сам факт лайка, но и качество последующего общения.
— Рекомендации строятся с использованием ранжирования, основанного на моделях типа learning-to-rank.

Ключевым источником вдохновения послужил алгоритм Gale–Shapley, использующийся в задаче стабильного брака. В оригинальной постановке каждый участник ранжирует партнёров, и цель — найти устойчивое соответствие. В адаптации Hinge этот принцип реализуется эвристически, через поиск так называемого «самого совместимого» партнёра дня, при этом система моделирует предпочтения обеих сторон [9].

Если обозначить:

— $P(u, v)$ — вероятность успешного взаимодействия между пользователями u и v;
— R_u — рейтинг, отражающий склонность u к взаимодействию с разными типами партнёров;
— C_v — контекстные характеристики пользователя v;

то задача рекомендации может быть сведена к оценке:

$$\hat{y}_{uv} = f(R_u, C_v)$$

где f — обученная модель, приближающая вероятность успешного взаи-

модействия. Под успешностью понимается не просто лайк, а наличие значимого чата или повторного контакта.

Кроме основных моделей, в системе используются дополнительные эвристики:

— подавление повторяющихся шаблонов (например, слишком частые лайки одному типажу);
— учёт предпочтений по контенту профиля (фото, ответы на подсказки);
— отслеживание реакций на предложенные анкеты и их отложенное влияние.

Важной частью рекомендаций Hinge является система подбора пары дня. Она основывается на анализе двусторонних предпочтений, активности и недавнего поведения. Рекомендуемый партнёр имеет высокий прогнозируемый шанс на взаимную симпатию и заинтересованный диалог.

Несмотря на успех такой модели, она имеет определённые ограничения:

— зависимость от истории пользователя, что создаёт проблему холодного старта;
— возможные локальные оптимумы: пользователь может застревать в узком профиле предложений;
— невысокая прозрачность — пользователю сложно понять, почему предложен тот или иной контакт.

В отличие от Tinder, Hinge делает ставку не на частоту свайпов, а на глубину взаимодействий. Это требует от системы учёта более сложных поведенческих метрик, включая динамику общения после совпадения. Такой подход требует более тонкой настройки, но лучше отвечает цели платформы — формированию устойчивых связей [4].

1.2.4 eHarmony

Рекомендательная система eHarmony изначально создавалась как экспертная модель совместимости, основанная на глубоком психологическом тестировании. Платформа ориентирована на долгосрочные отношения, а не на быстрые знакомства. Это определяет как архитектуру системы, так и методологию сбора и обработки данных.

В отличие от более поведенчески-ориентированных платформ, eHarmony делает ставку на анкеты, основанные на психологической типологии. При регистрации каждый пользователь заполняет обширную анкету, содержащую от 100 до 150 вопросов, касающихся:

— личностных черт (экстраверсия, добросовестность, невротизм и др.);
— ценностей и жизненных установок;
— отношения к конфликтам, компромиссам, религии и карьере;
— предпочтений в партнёрстве и стиле общения.

Из ответов формируется вектор признаков $x \in \mathbb{R}^d$, где d — число выделенных скрытых характеристик. Далее используется модель оценки совместимости между двумя пользователями u и v на основе расстояния между их признаковыми векторами:

$$S(u, v) = 1 - \frac{\|x_u - x_v\|}{D}$$

где D — нормирующий коэффициент (максимально возможное расстояние), а $S(u, v)$ — мера совместимости, принимающая значения от 0 до 1.

Алгоритм может включать дополнительные поправки:
— усиление совпадений по наиболее значимым признакам;
— штрафы за критические несовпадения (например, по отношению к детям или религии);
— предпочтение партнёров, близких по возрасту, географии или культурному фону.

Система реализует фильтрацию на основе заранее определённой модели совместимости, а не обучения на пользовательском поведении. Это означает, что:

1. рекомендации стабильны во времени и не зависят от текущей активности;
2. пользователи получают небольшой, но тщательно отобранный список совпадений;
3. основной целью алгоритма является структурная совместимость, а не привлекательность.

Несмотря на высокую психологическую обоснованность, такой подход имеет ряд ограничений:
— высокая когнитивная нагрузка при регистрации, что отпугивает часть пользователей;
— отсутствие адаптации под поведение — система не обучается на реальных откликах;
— возможная переориентация на типаж, а не на реальное разнообразие предпочтений.

Тем не менее, eHarmony демонстрирует устойчивую эффективность в своей нише, благодаря глубокой проработке модели совместимости и ориентации на фундаментальные ценности и личностные черты. Такой подход хорошо подходит для пользователей, ищущих стабильные и продолжительные отношения [10].

1.3 Особенности сбора и представления пользовательских признаков

В рекомендательных системах, применяемых в онлайн-знакомствах, центральную роль играет сбор и интерпретация информации о предпочтениях пользователей. Эти данные служат основой для построения персонализированных профилей и определения потенциально совместимых кандидатов. Анализ существующих решений в данной предметной области выявляет два крайних подхода к сбору информации:

— длинные анкеты с детализированными вопросами, характерные для платформ вроде eHarmony, обеспечивают богатое представление о пользователе, но требуют значительных усилий при заполнении и приводят к высокой доле отказов;

— минималистичные интерфейсы типа Tinder предлагают быструю оценку по принципу свайпа пользователей, обеспечивая высокую конверсию, но теряя глубину предпочтений.

В качестве компромиссного решения рассматриваются тернарные опросы, в которых каждый вопрос допускает три варианта реакции: положительную, отрицательную и нейтральную. Это позволяет выразить отношение к различным характеристикам без излишней нагрузки. Каждый ответ кодируется значением $q_i \in \{-1, 0, 1\}$, где i — номер вопроса. Такая шкала обеспечивает:

— компактность векторного представления профиля;

— возможность учитывать отсутствие чёткой позиции;

— поддержку различных методов машинного обучения и заполнения пропусков.

Результаты сбора формируются в разреженную матрицу $R = (q_{u,i})$, где u — пользователь, i — вопрос, а $q_{u,i}$ — реакция. Пропущенные значения соответствуют отсутствию взаимодействия. На практике возможно использование стратегий заполнения: нулями, средним значением по вопросу, либо построение модели, устойчивой к разреженности.

В дополнение к ответам могут быть включены и другие признаки:

— демографические данные — возраст (нормированный), пол (бинарная переменная);
— описания «о себе», конвертируемые в эмбеддинги с помощью языковых моделей;
— временные характеристики — длительность прохождения, количество пропусков, уверенность в ответах.

Особенность предметной области знакомств заключается в том, что каждый пользователь является одновременно субъектом и объектом рекомендаций. Это накладывает дополнительные требования к симметричности представлений и способности учитывать двустороннюю заинтересованность. Более того, успешность рекомендации здесь не сводится к лайку, а может быть оценена через цепочку взаимодействий: переписка, ответный интерес, встречи.

На этапе анализа задач были также выявлены следующие характеристики, которые следует учитывать при проектировании системы:

— необходимость работать с разреженными признаковыми матрицами и отсутствием части информации;
— поддержка холодного старта для новых пользователей и вопросов;
— ориентация на сравнение пользователей между собой, а не на ранжирование объектов фиксированной природы.

Таким образом, представление предпочтений в тернарной форме, дополненное демографией и эмбеддингами, формирует универсальную основу для построения профилей. Эти профили могут использоваться в различных методах рекомендации — от коллаборативной фильтрации до кластеризации и нейросетевых моделей.

2 Методы построения рекомендательной системы

2.1 Коллаборативная фильтрация

Коллаборативная фильтрация является одним из наиболее популярных подходов в рекомендательных системах, особенно в условиях, когда отсутствует явное описание объектов или пользователей. Основная идея заключается в том, что предпочтения пользователей могут быть предсказаны на основе поведения других пользователей с похожими вкусами.

Существует два основных типа коллаборативной фильтрации: на основе памяти (memory-based) и на основе модели (model-based). Первый подход использует метрики сходства между пользователями или объектами (например, косинусное расстояние, корреляцию Пирсона) и агрегирует оценки соседей. Второй — строит параметризованную модель на основе данных о взаимодействиях, чаще всего через факторизацию матрицы.

Пусть имеется матрица взаимодействий $R \in \mathbb{R}^{m \times n}$, где m — количество пользователей, n — количество объектов (например, анкет потенциальных партнёров). Элемент $r_{u,i}$ может обозначать бинарную оценку (лайк/не лайк), числовой рейтинг или иной сигнал предпочтения. Коллаборативная фильтрация предполагает, что в матрице присутствует скрытая структура, отражающая закономерности во вкусах пользователей [11].

Одним из распространённых методов является сингулярное разложение (SVD) или его модификации (например, Funk-SVD, ALS), при которых R аппроксимируется как

$$R \approx UV^\top,$$

где $U \in \mathbb{R}^{m \times k}$ и $V \in \mathbb{R}^{n \times k}$ содержат латентные векторы пользователей и объектов соответственно, а k — число латентных признаков. Значение $\hat{r}_{u,i} = U_u \cdot V_i^\top$ интерпретируется как предсказанная степень интереса пользователя u к объекту i.

Интересным обобщением является применение коллаборативной фильтрации к данным не только о лайках, но и о признаках, таких как ответы пользователей на опросы. В этом случае каждый пользователь представлен тернарным или категориальным вектором, а матрица $Q \in \{-1, 0, 1\}^{m \times p}$ (где p — число вопросов) также может быть факторизована аналогичным способом:

$$Q \approx U'Z^\top.$$

Полученные векторы могут использоваться для оценки схожести между пользователями, для восстановления пропущенных ответов или как источник признаков для гибридных моделей.

Коллаборативная фильтрация демонстрирует высокую эффективность при наличии большого количества пользовательских взаимодействий, однако страдает от проблемы холодного старта и может усиливать популярность одних и тех же объектов, снижая разнообразие рекомендаций [12].

2.2 Контентная фильтрация

Контентная фильтрация представляет собой один из классических подходов к построению рекомендательных систем. Основная идея заключается в том, чтобы рекомендовать объекты, схожие с теми, которые пользователь оценил положительно ранее, основываясь на характеристиках самих объектов. В отличие от коллаборативной фильтрации, здесь не учитываются предпочтения других пользователей.

Каждый объект описывается вектором признаков, которые могут быть бинарными, числовыми или категориальными. Пусть объект j представлен вектором признаков $x_j \in \mathbb{R}^d$. Модель пользователя строится как агрегированное представление объектов, с которыми у него были положительные взаимодействия. Например, если пользователь i взаимодействовал с объектами $j_1, \ldots, j_k$, то вектор предпочтений можно получить как среднее:

$$p_i = \frac{1}{k} \sum_{s=1}^{k} x_{j_s}.$$

Рекомендации формируются на основе сходства между вектором предпочтений пользователя и векторами новых объектов. Чаще всего используется косинусное расстояние:

$$\mathrm{sim}(p_i, x_j) = \frac{p_i^\top x_j}{\|p_i\| \cdot \|x_j\|}.$$

Объекты с наибольшим значением sim включаются в топ рекомендаций [13].

Данный подход имеет ряд достоинств:

— высокая интерпретируемость: можно объяснить, почему был рекомендован тот или иной объект;

— независимость от количества других пользователей;
— устойчивость к проблеме холодного старта для объектов (если их описание доступно).

Однако контентная фильтрация имеет и ограничения:

— рекомендации ограничиваются областью уже проявленных интересов;
— трудно учитывать сложные зависимости между признаками;
— качество зависит от полноты и выразительности признакового описания.

Для повышения гибкости могут применяться методы машинного обучения. Например, обучающая выборка может включать пары (x_j, y_{ij}), где y_{ij} — бинарная переменная, указывающая наличие положительного отклика со стороны пользователя i на объект j. На этой основе можно обучить логистическую регрессию, SVM или градиентный бустинг, предсказывающий вероятность интереса к новому объекту.

Также возможны гибридные модели, объединяющие контентную и коллаборативную фильтрацию. Например, контентные признаки могут использоваться для регуляризации матричной факторизации или служить входом для нейронных сетей. Такие подходы позволяют улучшить обобщающую способность модели и преодолеть узость интересов, характерную для чисто контентной фильтрации [14].

Контентные методы особенно полезны в системах, где объекты имеют чётко выраженные признаки: тексты, категории, изображения, ответы на тесты или анкеты. При наличии информативного описания они позволяют получать рекомендации уже на самых ранних этапах использования системы, что делает их важным компонентом гибридных решений.

2.3 Эвристики: совпадения по ответам, популярность, фильтры

Эвристические методы в рекомендательных системах основаны на наборе простых правил и предположений, позволяющих быстро и эффективно формировать рекомендации. Несмотря на относительную простоту, такие подходы остаются актуальными, особенно в условиях ограниченности данных или требований к объяснимости.

Одним из базовых эвристических подходов является сравнение пользователей по их ответам на вопросы анкет или опросов. Если ответы представлены в тернарной шкале (например, -1 — несогласие, 0 — нейтрально, 1 — согласие), то схожесть между пользователями можно оценить по доле совпадающих

ответов:

$$\mathrm{sim}(u, v) = \frac{1}{|P|} \sum_{i \in P} (q_{u,i} = q_{v,i}),$$

где P — множество вопросов, на которые оба пользователя дали ответ. Этот подход применим как в системах знакомств, так и в других областях, где важна совместимость взглядов, предпочтений и интересов.

Другим эвристическим приёмом является использование показателя популярности. Объекты (например, профили или товары), получившие наибольшее количество положительных оценок, могут предлагаться новым пользователям в качестве стартовых рекомендаций. Популярность может быть нормирована, например:

$$\mathrm{pop}(i) = \frac{\text{число лайков объекта } i}{\text{максимальное число лайков среди всех объектов}}.$$

Популярные рекомендации часто дополняются фильтрацией по демографическим и другим признакам, таким как возраст, пол, географическое местоположение, язык, наличие общих интересов и т. д [15].

Такие фильтры применяются до или после основного ранжирования и позволяют исключить очевидно нерелевантные варианты. Например, пользователь, заинтересованный только в кандидатах определённого возраста или пола, должен получать только соответствующие предложения.

Также может применяться эвристика совпадения по ключевым признакам. Если в профиле пользователя указаны предпочтения (например, любимые фильмы, занятия, взгляды), система может искать совпадения с другими профилями и ранжировать их по количеству совпавших интересов.

Комбинирование эвристик позволяет построить гибкую систему, способную адаптироваться к условиям отсутствия данных, начальной загрузки, а также повысить обоснованность рекомендаций. При этом эвристические методы легко интерпретируемы, что важно в чувствительных сферах, таких как онлайн-знакомства или подбор персонала.

2.4 Применение кластеризация (k-means, DBSCAN) в рекомендациях

Кластеризация — это метод обучения без учителя, направленный на группировку объектов в кластеры таким образом, чтобы элементы одного кластера были похожи друг на друга и отличались от элементов других кластеров. В

контексте рекомендательных систем кластеризация применяется для:

— сегментирования пользователей (или объектов) по интересам, поведению или признакам;
— повышения масштабируемости рекомендаций за счёт ограничения поиска релевантных кандидатов внутри кластера;
— выявления нишевых предпочтений и персонализированных паттернов.

Одним из наиболее популярных алгоритмов является k-means. Он принимает на вход число кластеров k и итеративно минимизирует внутрикластерную дисперсию:

$$\sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2,$$

где C_j — кластер, а μ_j — его центр. Алгоритм эффективен при компактных и сферических кластерах, но чувствителен к выбору k и неустойчив к выбросам.

Для более гибкой кластеризации используется алгоритм DBSCAN. Он группирует точки по плотности: кластером считается связная по плотности область, где каждая точка имеет хотя бы $minPts$ соседей в пределах радиуса ε. DBSCAN способен находить кластеры произвольной формы и автоматически игнорирует шум (выбросы), что делает его особенно полезным в разнородных данных.

В рекомендательных системах кластеризация применяется по-разному:

— На пространстве пользователей: сегментация по поведенческим или опросным признакам позволяет формировать кластеры пользователей с похожими предпочтениями. Рекомендации для нового пользователя можно извлекать из наиболее близкого кластера.
— На пространстве объектов: группировка товаров, фильмов, анкет и пр. по тематике, стилю или целевой аудитории позволяет адаптировать рекомендации к интересам пользователя.
— В латентных пространствах: кластеризация векторов после факторизации (например, в SVD или autoencoder-подходах) даёт более сжатое и семантически значимое представление.

Кластеризация также применяется для визуализации и анализа структуры пользовательской базы, выявления целевых групп и построения тематических подборок. Её эффективность во многом зависит от выбора признаков и масштабов данных, поэтому нередко она используется в комбинации с другими

методами [16].

2.5 Стратегии холодного старта

Проблема холодного старта возникает, когда система не располагает достаточной информацией о пользователях или объектах, чтобы формировать персонализированные рекомендации. Выделяют два основных сценария: появление нового пользователя и добавление нового объекта (например, анкеты).

Для новых пользователей могут применяться следующие подходы:

- заполнение вступительных тестов или анкет — позволяет собрать первичные признаки и использовать их при формировании рекомендаций;
- использование демографических данных — рекомендации подбираются на основе поведения пользователей с аналогичными характеристиками (возраст, пол, география и т.д.);
- показ популярных объектов — временная стратегия, при которой пользователю демонстрируются анкеты с высокой оборачиваемостью, что помогает быстрее сформировать профиль предпочтений.

При появлении новых объектов, которые ещё не получили откликов, возможны такие меры:

- временное повышение приоритета в выдаче — например, показ новым или активным пользователям для ускоренного накопления статистики;
- использование признаков схожести — на основе анкетных данных, внешности (в случае CV), текста описания (в случае NLP) или ответов на вопросы;
- размещение в релевантных сегментах — объект может быть временно включён в выдачу по кластерам, в которые он потенциально попадает по признаковому пространству.

Кроме того, существуют универсальные стратегии, применимые и к новым пользователям, и к новым объектам:

- инициализация признаков с помощью доступных внешних данных — анкет, биографий, метаинформации;
- использование гибридных моделей, сочетающих элементы content-based и коллаборативной фильтрации — это снижает чувствительность к отсутствию истории;
- активное обучение — выбор контента, максимально полезного для уточнения предпочтений, что позволяет за минимальное число взаимодействий

улучшить качество рекомендаций.

Проблема холодного старта наиболее критична для систем, основанных на коллаборативной фильтрации, поскольку они требуют исторических данных о взаимодействии пользователей с объектами. Поэтому многие современные решения строятся с использованием дополнительных эвристик и предварительной инициализации признаков, что позволяет обеспечить устойчивость системы на ранних этапах использования [17].

2.6 Методы глубокого обучения в рекомендательных системах

Развитие нейросетевых архитектур оказало существенное влияние на область рекомендательных систем. Благодаря способности извлекать сложные скрытые зависимости из разнородных данных, модели глубокого обучения применяются для повышения качества рекомендаций как в традиционных задачах (например, предсказание рейтингов), так и в более сложных сценариях, включая мультимодальные рекомендации, учет временной динамики и персонализацию на основе контекста.

Одним из первых направлений стало расширение классической матричной факторизации с помощью нейронных сетей. Вместо простой линейной факторизации матрицы взаимодействий $R \in \mathbb{R}^{m \times n}$, где m — количество пользователей, n — количество объектов, и R_{ij} — факт взаимодействия, используются обучаемые эмбеддинги и нелинейные функции активации. Примером такой модели является Neural Collaborative Filtering (NCF) [18]. В NCF пары эмбеддингов (u_i, v_j) передаются через многослойный перцептрон:

$$\hat{r}_{ij} = \mathrm{MLP}([u_i, v_j]),$$

где $[\cdot, \cdot]$ обозначает конкатенацию векторов. Модель обучается по функции потерь, например, бинарной кросс-энтропии в задаче предсказания лайка/дизлайка.

Другое направление связано с использованием рекуррентных и трансформерных архитектур для моделирования последовательности взаимодействий. Так называемые sequence-based recommenders учитывают порядок взаимодействий пользователя с объектами. Например, модель GRU4Rec применяет Gated Recurrent Unit (GRU) [19] для обработки истории действий пользователя. Базовая идея заключается в следующем: пусть $x_1, x_2, \ldots, x_T$ — последовательность

взаимодействий, тогда на каждом шаге рассчитывается скрытое состояние h_t:

$$h_t = \mathrm{GRU}(x_t, h_{t-1}),$$

и предсказывается следующий элемент x_{t+1} с помощью softmax-слоя.

Для учёта более длинных зависимостей и параллельной обработки была предложена модель SASRec, основанная на механизме внимания. В отличие от рекуррентных моделей, здесь используется позиционно-кодированная последовательность эмбеддингов, проходящая через слои трансформера. Это позволяет учитывать контекст всех предыдущих действий при выборе следующей рекомендации [20].

Особое место занимают графовые нейронные сети, применяемые для моделирования взаимодействий в виде графов. В Graph Convolutional Matrix Completion [21], каждый пользователь и объект представляются вершинами, соединёнными ребром при наличии взаимодействия. Представления вершин обновляются по правилу:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)}\right),$$

где $\mathcal{N}(i)$ — соседи вершины i, $W^{(l)}$ — матрица весов на l-м слое, c_{ij} — коэффициент нормализации. Такой подход позволяет учитывать структуру взаимодействий и взаимосвязь объектов, что особенно актуально для задач, где важны не только пользовательские предпочтения, но и социальные или контекстные связи.

Ещё одно активно развивающееся направление — мультимодальные рекомендации. Здесь помимо взаимодействий учитываются дополнительные признаки, такие как текст описания, изображения, аудио. Для обработки таких данных применяются CNN, BERT и другие специализированные архитектуры. В модели VBPR визуальные признаки изображений используются для расширения латентного пространства:

$$\hat{r}_{ij} = u_i^\top v_j + u_i^\top E x_j,$$

где x_j — визуальный вектор, полученный из изображения объекта, E — обучаемая матрица проекции [22].

Основные преимущества методов глубокого обучения:

— способность моделировать сложные нелинейные зависимости между пользователями и объектами;
— возможность использовать разнородные источники информации;
— высокая гибкость и расширяемость архитектур.

Тем не менее, существуют и ограничения:

— высокая требовательность к вычислительным ресурсам;
— потребность в большом объёме размеченных данных;
— трудность интерпретации и объяснимости результатов.

Таким образом, методы глубокого обучения открывают широкие перспективы для построения более точных и персонализированных рекомендательных систем, особенно в условиях, когда доступны дополнительные источники информации и достаточно ресурсов для обучения. Однако выбор таких подходов должен быть обоснован задачами проекта, размером аудитории и доступной инфраструктурой.

2.7 Гибридные подходы

Гибридные рекомендательные системы объединяют преимущества различных методов, включая коллаборативную фильтрацию, контентную фильтрацию и эвристические алгоритмы. Такая интеграция позволяет компенсировать слабые стороны отдельных подходов и достичь более высокой точности, устойчивости к холодному старту и разнообразия рекомендаций.

Существуют разные стратегии гибридизации:

— объединение выходов нескольких моделей (late fusion);
— комбинирование признаков на входе одной модели (early fusion);
— использование одного подхода в качестве фильтра, а другого — для ранжирования.

Один из классических примеров — модель, в которой одновременно учитываются схожесть пользователей (коллаборативная составляющая) и сходство объектов (контентная составляющая). Пусть r_{ui} — предсказанная оценка пользователя u для объекта i. Тогда комбинированная формула может иметь следующий вид:

$$r_{ui} = \alpha \cdot r_{ui}^{\text{collab}} + (1 - \alpha) \cdot r_{ui}^{\text{content}},$$

где r_{ui}^{collab} — предсказание по коллаборативной модели, r_{ui}^{content} — результат контентного ранжирования, а $\alpha \in [0, 1]$ — параметр, регулирующий вклад каждой части.

Другой подход основан на поэтапной фильтрации. Например, можно сначала применить контентную фильтрацию для предварительного отбора релевантных объектов, а затем выполнить коллаборативное ранжирование по пользовательским лайкам или оценкам. Такой двухшаговый процесс снижает вычислительную нагрузку и повышает релевантность [23].

Гибридные системы особенно полезны в следующих сценариях:

— наличие разреженной матрицы пользовательских взаимодействий, при этом имеются структурированные признаки объектов;

— необходимость рекомендаций для новых пользователей или новых объектов;

— желание учитывать не только историю взаимодействий, но и семантическое содержание;

— ориентация на объяснимость модели и возможность интерактивной настройки предпочтений.

В последние годы широкое распространение получили модели, встраивающие оба подхода в общую латентную структуру. Например, в модели Factorization Machines (FM) и её нейронных расширениях (Neural FM, DeepFM) признаки пользователей и объектов подаются в общий обучаемый слой, позволяющий учитывать как контентную, так и взаимодействующую информацию. Обозначим входной вектор признаков как x, тогда предсказание в FM-модели имеет вид:

$$\hat{y}(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle v_i, v_j \rangle x_i x_j,$$

где v_i — латентный вектор i-го признака. Такая модель способна улавливать взаимодействия между признаками без явного задания структуры [24].

Гибридные методы на практике показывают высокую гибкость и адаптивность, особенно в условиях изменяющихся предпочтений и ограниченных пользовательских данных. Они успешно применяются в рекомендательных системах крупных платформ (Netflix, YouTube, LinkedIn), где важно сочетать поведенческие паттерны с контекстной и персонализированной информацией.

Совокупность перечисленных подходов делает гибридные методы универсальным инструментом, который можно адаптировать под особенности конкретной предметной области, в том числе в сфере онлайн-знакомств.

2.8 Методы оценки рекомендательной системы

Оценка качества рекомендательной системы в приложении знакомств играет ключевую роль для обеспечения релевантности и привлекательности предложений, предоставляемых пользователю. Цель такой оценки — убедиться, что система способствует установлению взаимного интереса и активному взаимодействию между пользователями.

В рамках мобильного приложения знакомств эффективность рекомендаций оценивается с двух сторон: на основе исторических данных (оффлайн) и в реальном времени (онлайн).

Оффлайн-оценка проводится до запуска системы на основе известных пользовательских взаимодействий. При этом выбирается контрольная выборка, в которой определённые события (например, взаимные лайки) скрываются, а алгоритм должен их предсказать. Основное внимание в данном случае уделяется метрике HitRate@K — доле случаев, в которых хотя бы один релевантный объект оказался среди топ-K рекомендаций. Эта метрика отражает способность системы «попадать» в интересы пользователя и используется в качестве основной целевой метрики.

Также применяются следующие метрики:

— Precision@K — доля релевантных пользователей среди K рекомендованных;

— Recall@K — доля релевантных пользователей, которые удалось рекомендовать;

— MRR (Mean Reciprocal Rank) — учитывает позицию первого релевантного объекта в списке;

— NDCG — нормированная кумулятивная дисконтированная полезность, учитывающая ранжирование;

— Coverage — доля пользователей, которым система способна выдать хотя бы одну осмысленную рекомендацию. Высокий coverage свидетельствует о способности системы охватывать широкую аудиторию.

Оффлайн-оценка обычно проводится по схеме leave-one-out, когда одна интеракция пользователя исключается из тренировочного множества и исполь-

зуется для тестирования [25].

После внедрения системы важно отслеживать эффективность рекомендаций по поведенческим метрикам в режиме онлайн. Наиболее информативные из них:

1. Доля взаимных лайков среди выданных рекомендаций;
2. Конверсия в диалог — отношение числа начатых чатов к количеству рекомендованных профилей;
3. Среднее время до первого взаимодействия (лайка или сообщения);
4. Удержание пользователей — как долго и регулярно пользователь взаимодействует с рекомендованными контактами;
5. Повторные взаимодействия — оценивается, продолжается ли активность с рекомендованным пользователем спустя время.

Помимо точности, важно учитывать качественные характеристики системы, которые напрямую влияют на пользовательское восприятие и вовлечённость:

- Разнообразие — рекомендации не должны быть однотипными;
- Новизна — включение ранее не встречавшихся кандидатов повышает интерес;
- Персонализация — учёт индивидуальных особенностей конкретного пользователя;
- Равномерность — отсутствие систематического перекоса в сторону ограниченной группы пользователей.

Для оценки рекомендательной системы в приложении знакомств важно сочетать количественные метрики точности и охвата с показателями пользовательского поведения. Такой подход позволяет обеспечить релевантность, персонализацию и устойчивую вовлечённость аудитории.

3 Теоретические основы разработки мобильных приложений

3.1 Общие особенности мобильной разработки

Мобильные приложения функционируют в условиях, отличающихся от настольной или серверной среды. Эти отличия накладывают определённые ограничения и формируют особые требования к проектированию, реализации и тестированию.

Во-первых, мобильные устройства имеют ограниченные ресурсы. Смартфоны и планшеты уступают настольным системам по вычислительной мощности, объёму оперативной памяти и возможностям хранения данных. Кроме того, приложения должны быть чувствительны к расходу батареи, особенно при активном использовании мультимедиа, анимации и фоновых процессов.

Во-вторых, в экосистеме мобильных устройств наблюдается значительное разнообразие. Существует множество моделей устройств с различными размерами экранов, плотностью пикселей, аппаратными возможностями и версиями операционных систем. Это требует особого внимания к адаптивности интерфейса и совместимости приложения с широким спектром устройств.

Помимо технических ограничений, важным аспектом является пользовательский опыт. Пользователи ожидают от приложения высокой скорости отклика, визуальной плавности и интуитивной структуры. Низкое качество интерфейса или перегруженность функциональностью могут привести к отказу от использования, независимо от пользы приложения.

Среди ключевых особенностей мобильной разработки можно выделить:

- ограниченные вычислительные ресурсы устройства;
- требования к экономии энергии и управлению фоновыми задачами;
- разнообразие устройств, экранов и версий операционных систем;
- необходимость обеспечения высокого уровня интерактивности и отзывчивости интерфейса;
- ориентация на кратковременные, но частые сценарии использования.

Эти особенности определяют специфику проектирования мобильных решений. Для успешной реализации приложения разработчику необходимо соблюдать ряд принципов, направленных на обеспечение стабильности, производительности и удобства использования.

К таким принципам относятся:

1. адаптация пользовательского интерфейса под различные размеры и ори-

ентации экранов;

2. минимизация использования ресурсов устройства;
3. соблюдение рекомендаций по дизайну, принятых в целевой платформе;
4. обеспечение плавной и предсказуемой навигации по приложению;
5. проведение тестирования на разных устройствах и версиях операционной системы.

Таким образом, мобильная разработка представляет собой область, требующую сочетания инженерной дисциплины, внимания к деталям и ориентации на пользовательский опыт. Эти аспекты определяют основу архитектурных и технологических решений в мобильных системах.

3.2 Общие подходы к проектированию мобильных систем

Проектирование мобильных приложений требует системного подхода, включающего выбор архитектурной модели, структурирование компонентов и определение принципов взаимодействия между ними. Эти решения оказывают значительное влияние на надёжность, масштабируемость и удобство сопровождения системы.

К числу ключевых факторов, влияющих на архитектуру мобильного приложения, относятся:

— целевая платформа (Android, iOS или обе);
— требования к производительности и времени отклика;
— предполагаемый объём пользовательского трафика и сценарии нагрузки;
— организационные ограничения (время, бюджет, ресурсы);
— ожидаемая эволюция проекта и возможность масштабирования.

Современные мобильные приложения обычно строятся как распределённые системы, в которых клиент и сервер выполняют разные роли. Распространённые архитектурные подходы включают:

— монолитную архитектуру с полной локальной логикой (редко применяется в современных системах);
— клиент-серверную архитектуру с разделением обязанностей между устройством пользователя и серверной частью;
— модульную или микросервисную архитектуру, обеспечивающую масштабируемость и гибкость.

При этом важную роль играет концепция разделения ответственности: пользовательский интерфейс, бизнес-логика и работа с данными разносится

по разным слоям приложения. Это способствует улучшению тестируемости, повторному использованию компонентов и упрощению сопровождения [26].

Независимо от конкретной реализации, архитектура мобильной системы должна обеспечивать:

1. слабую связанность между модулями;
2. чёткие границы между слоями;
3. стандартизированное взаимодействие между компонентами (например, через REST API);
4. возможность независимого обновления и масштабирования подсистем.

Такой подход обеспечивает устойчивую основу для развития и поддержки мобильного приложения в условиях изменяющихся требований и роста нагрузки.

3.3 Клиент-серверная архитектура

Клиент-серверная архитектура является одним из наиболее устойчивых и широко применяемых подходов при построении современных мобильных приложений. Её основным принципом является логическое разделение системы на два взаимосвязанных компонента: клиентскую часть, работающую на пользовательском устройстве, и серверную часть, выполняющую обработку запросов, управление данными и реализацию бизнес-логики.

Данный архитектурный подход обеспечивает целый ряд преимуществ:

— централизованное управление и консистентность данных;
— разгрузка клиентской части за счёт переноса вычислений на сервер;
— возможность переиспользования серверной логики в различных клиентских интерфейсах (веб, мобильных и пр.);
— упрощение обновления клиентских приложений без необходимости модификации серверного кода;
— гибкость масштабирования и балансировки нагрузки на стороне сервера.

Роль клиентской части заключается в следующем:

— предоставление пользовательского интерфейса и взаимодействие с пользователем;
— сбор и первичная обработка пользовательских данных;
— формирование сетевых запросов и обработка полученных ответов;
— управление локальным состоянием приложения и навигацией.

Серверная часть, в свою очередь, обеспечивает:

— реализацию бизнес-правил и логики приложения;
— аутентификацию и авторизацию пользователей;
— централизованное хранение и обработку данных;
— взаимодействие с внешними сервисами;
— логирование, мониторинг и обеспечение безопасности.

Взаимодействие между клиентом и сервером, как правило, осуществляется через стандартизированные протоколы (например, HTTP) и форматы обмена данными (чаще всего JSON или XML). Такой подход обеспечивает платформонезависимость и простоту интеграции.

Для реализации клиентской части всё более широкое распространение получают кроссплатформенные фреймворки, такие как Flutter. Он позволяет разрабатывать приложения с единой кодовой базой, сохраняя при этом высокую производительность и выразительность интерфейса.

Серверная логика зачастую реализуется с использованием зрелых и гибких платформ, таких как Spring, обеспечивающих поддержку REST API, управление транзакциями, безопасность и масштабируемость [27].

Таким образом, клиент-серверная архитектура остаётся актуальной и надёжной основой для построения мобильных систем, сочетающих в себе адаптивность, расширяемость и удобство сопровождения.

3.4 Безопасность и конфиденциальность

В современных мобильных приложениях вопросы безопасности и конфиденциальности данных занимают центральное место. Пользователи ожидают, что их личная информация будет защищена от несанкционированного доступа, утечки или подмены. Это особенно актуально для приложений, обрабатывающих чувствительные персональные данные, включая профили пользователей, переписку, предпочтения и другую информацию личного характера.

Обеспечение безопасности требует комплексного подхода, охватывающего как клиентскую, так и серверную часть приложения. В числе ключевых задач:
— защита канала передачи данных от перехвата и модификации;
— надёжная аутентификация и авторизация пользователей;
— контроль доступа к защищённым ресурсам;
— предотвращение распространённых уязвимостей (включая XSS, CSRF, SQL-инъекции и другие);
— обеспечение целостности и актуальности пользовательской сессии;

— соблюдение требований к обработке и хранению персональных данных.

Одной из стандартных практик является реализация авторизации на основе токенов. Наиболее распространённым решением в этом направлении являются компактные самодостаточные токены, которые позволяют хранить информацию о пользователе и его правах доступа в зашифрованном или подписанном виде. Использование токенов удобно в распределённых системах и облегчает реализацию масштабируемой безсессионной архитектуры [28].

Для защиты пользовательских данных от перехвата при передаче, общепринятым стандартом является использование защищённых транспортных протоколов. В частности, применяется HTTPS, основанный на TLS, который обеспечивает шифрование и аутентификацию соединения между клиентом и сервером.

На серверной стороне безопасность реализуется посредством встроенных или внешних фреймворков, позволяющих централизованно управлять доступом, разграничивать привилегии и применять политики безопасности. Такие решения также обеспечивают:

— предварительную фильтрацию входящих запросов;
— управление сессиями или токенами;
— обработку исключений, связанных с неавторизованным доступом;
— регистрацию действий пользователей для целей аудита.

С точки зрения пользовательского опыта и доверия, защита конфиденциальности является неотъемлемой частью проектирования. Приложение должно обеспечивать безопасное хранение пользовательских данных, возможность их удаления, а также отказоустойчивость в случае попыток несанкционированного доступа [29].

На практике к задачам обеспечения безопасности подходят с учётом принципов минимизации данных, разделения ответственности и поэтапного усиления контроля. Это позволяет проектировать надёжные и устойчивые к угрозам системы, соответствующие современным ожиданиям и требованиям.

4 Проектирование архитектуры приложения для знакомств

4.1 Проектирование архитектуры мобильного приложения

Проектирование архитектуры мобильного приложения было выполнено на основе системного анализа предполагаемой функциональности, пользовательских сценариев и требований к масштабируемости. Приложение реализуется как клиентская часть, взаимодействующая с сервером через REST API, и играет роль интерфейса между конечным пользователем и рекомендательной системой.

Архитектура приложения строится вокруг концепции модулярности и разделения ответственности. Это означает, что каждый экран и компонент реализует строго определённую роль, что упрощает тестирование, расширение и поддержку.

В качестве основного навигационного каркаса был выбран стек-навигации с возможностью модального перехода, поскольку пользователи часто выполняют действия в глубину (например, редактирование профиля, просмотр чужих профилей, переход в чат) с последующим возвратом к предыдущему контексту.

Начальной точкой проектирования стало обеспечение контроля доступа и персонализации взаимодействия. Приложение начинается с экрана входа, где пользователь может авторизоваться или зарегистрироваться. Выбор между этими ветками предопределяет дальнейший маршрут в навигации.

На рисунке 1 представлена схема, иллюстрирующая данный процесс. При регистрации пользователь предоставляет базовую информацию (имя пользователя, пол, краткое описание), которая сохраняется на сервере и может быть использована как часть признаков в системе рекомендаций.

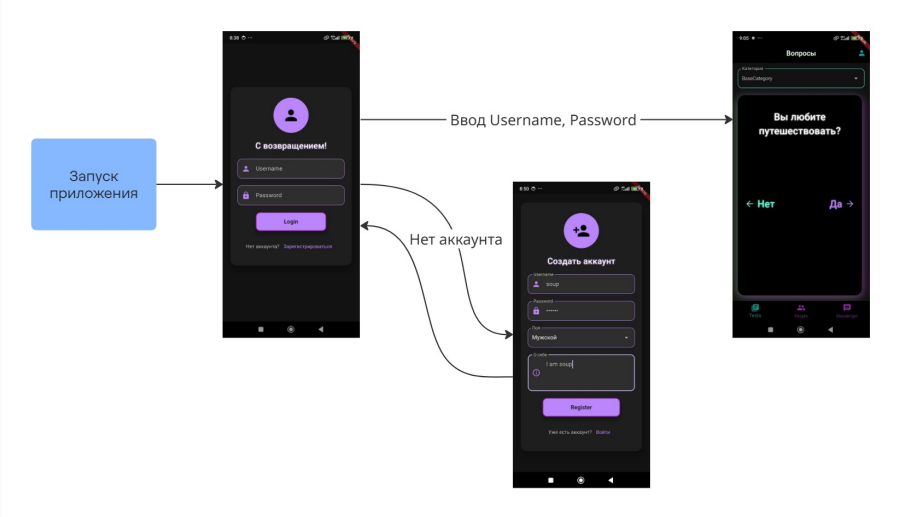

Рисунок 1 – Процесс входа и регистрации

Каждому пользователю доступен экран профиля, где отображается личная информация. В процессе проектирования было определено, что взаимодействие с профилем должно быть двухуровневым: просмотр и редактирование. Таким образом, экран "Профиль"реализован как статическое представление данных, а редактирование (аватара и описания) выполняется через отдельные маршруты. Это позволяет минимизировать когнитивную нагрузку и избежать случайных изменений.

На рисунке 2 показаны возможные действия пользователя с личным профилем. Архитектурно, экран реализован как обособленный компонент, который получает и обновляет данные через API, сохраняя локальное состояние до момента отправки изменений.

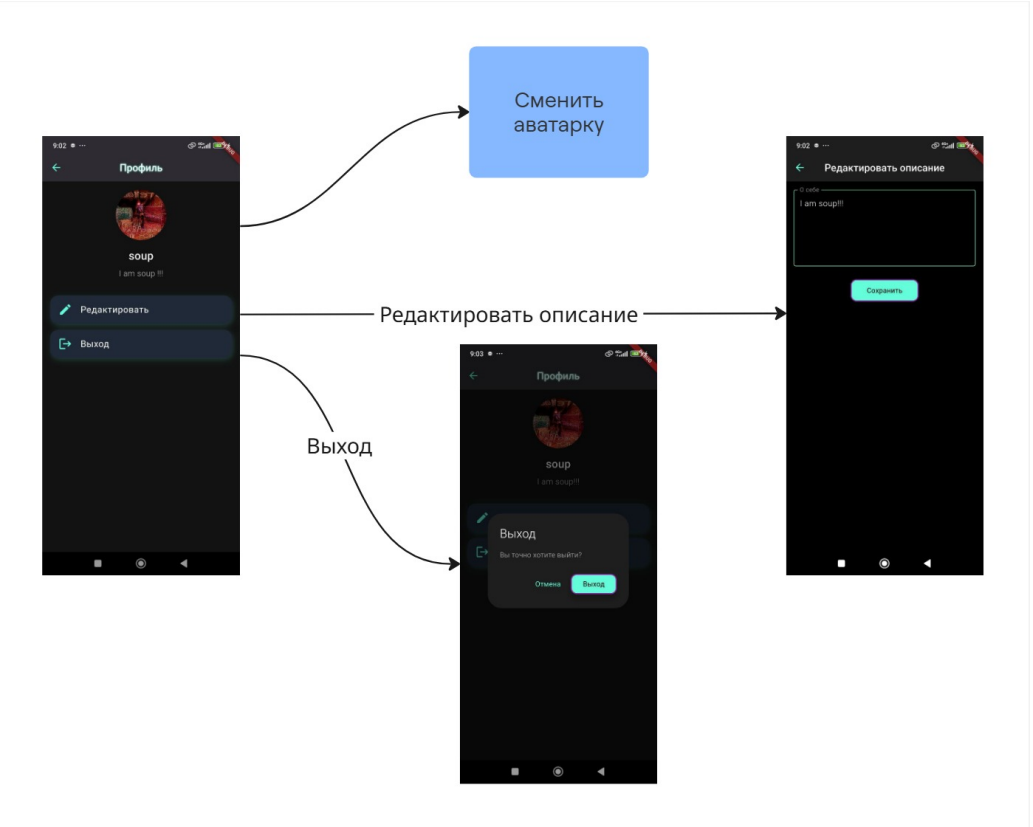

Рисунок 2 – Редактирование профиля и выход

Сердцем рекомендательной системы является вектор признаков, формируемый из ответов пользователя на серию вопросов. Было принято решение реализовать интерфейс прохождения теста через свайпы, поскольку это сочетает простоту использования и скорость. Такой формат также является привычным для пользователей в контексте приложений знакомств.

Как показано на рисунке 3, пользователь выбирает категорию, после чего отображается последовательность карточек с вопросами. Ответ осуществляется

свайпом влево (отрицательный ответ), вправо (положительный) или вверх (нейтральный). При достижении конца набора карточек приложение сообщает об этом. В архитектуре эта часть реализована как отдельный модуль с локальным буфером вопросов, загружаемых с сервера по категориям.

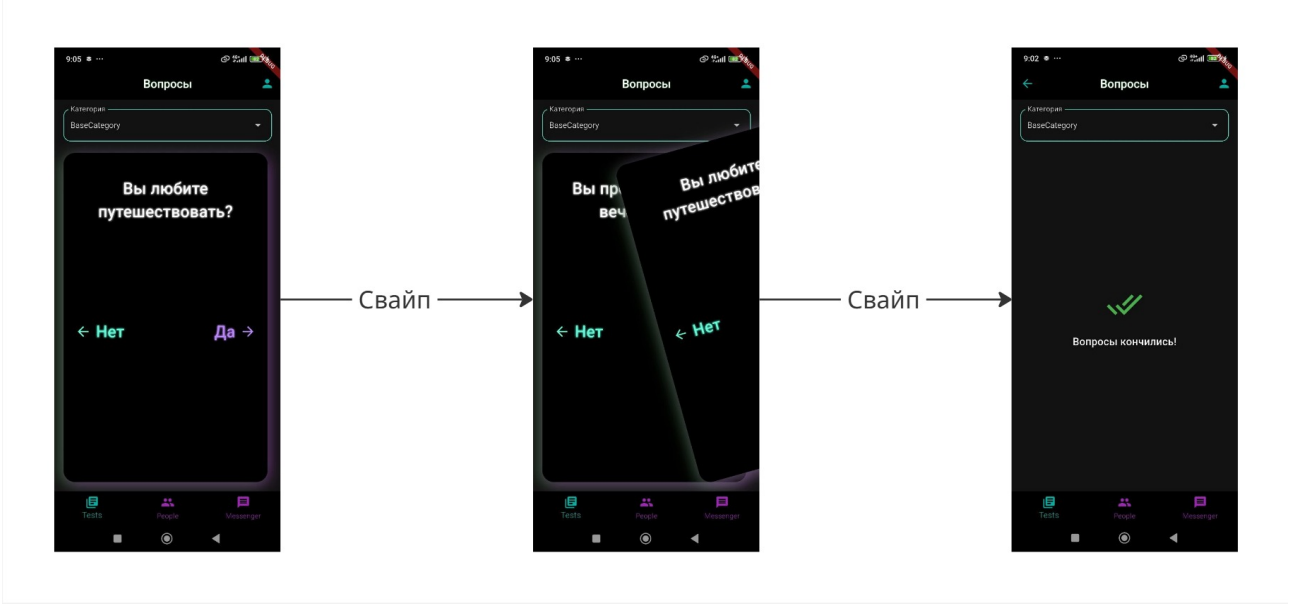

Рисунок 3 – Процесс ответов на вопросы

На основе сформированных векторов признаков сервер предоставляет список пользователей, наиболее близких по сходству. Эта информация отображается на экране рекомендаций . Данный экран проектировался как динамическая таблица, получающая данные из внешнего источника с возможностью фильтрации и сортировки.

Как показано на рисунке 4, каждая карточка рекомендации предоставляет краткую информацию: имя, сходство и кнопку для запроса на чат. Нажатие на карточку открывает модальное окно с расширенным профилем. Эта структура проектировалась для поддержки масштабируемости — при появлении новых фильтров или параметров сортировки можно расширить интерфейс без изменения существующего кода отображения.

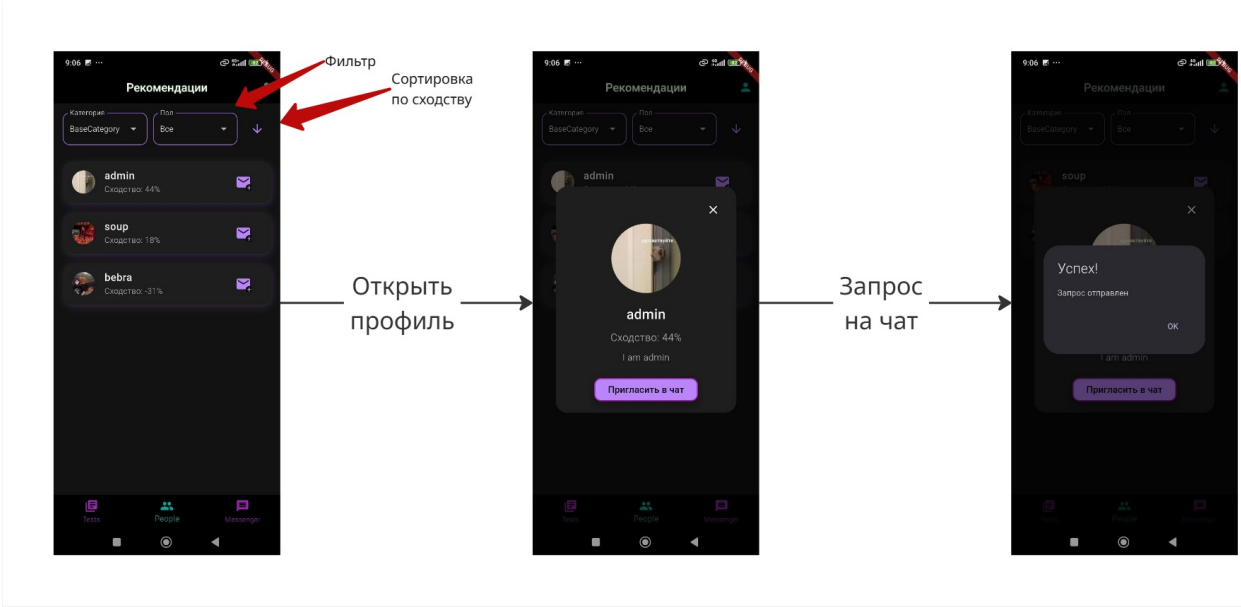

Рисунок 4 – Рекомендации, просмотр профиля и отправка запроса

Переход от рекомендаций к активному взаимодействию осуществляется через систему запросов на чат. С точки зрения архитектуры, чат требует наличия согласия двух сторон. Для этого реализован модуль управления заявками (Рисунок 5), в котором разделены входящие запросы и активные чаты.

Входящие запросы представлены в виде списка с возможностью принятия или отклонения. Принятые запросы автоматически перемещаются в раздел активных чатов. Таким образом, логика работы с чатами разделена на два уровня: согласование и непосредственное общение.

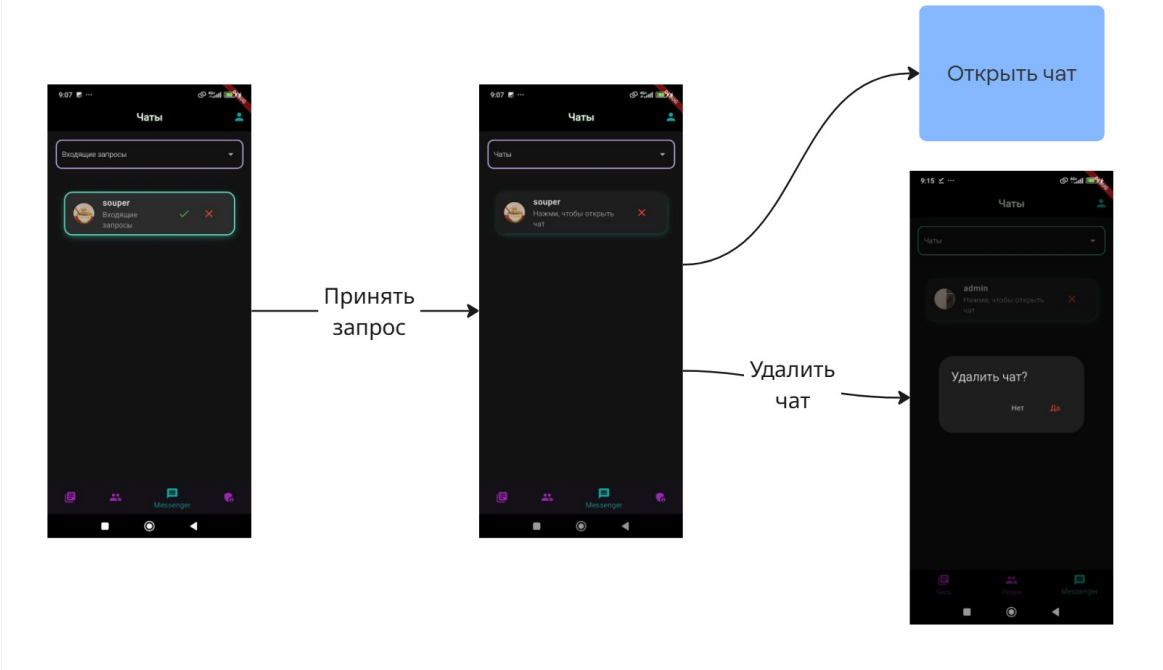

Рисунок 5 – Управление чатами и запросами на чат

Чат реализован как асинхронный поток сообщений. Было принято архитектурное решение о поддержке мультимедиа, что потребовало внедрения системы предварительного просмотра и управления прикреплёнными файлами. На рисунке 6 представлена логика интерфейса: отправка текста и вложений.

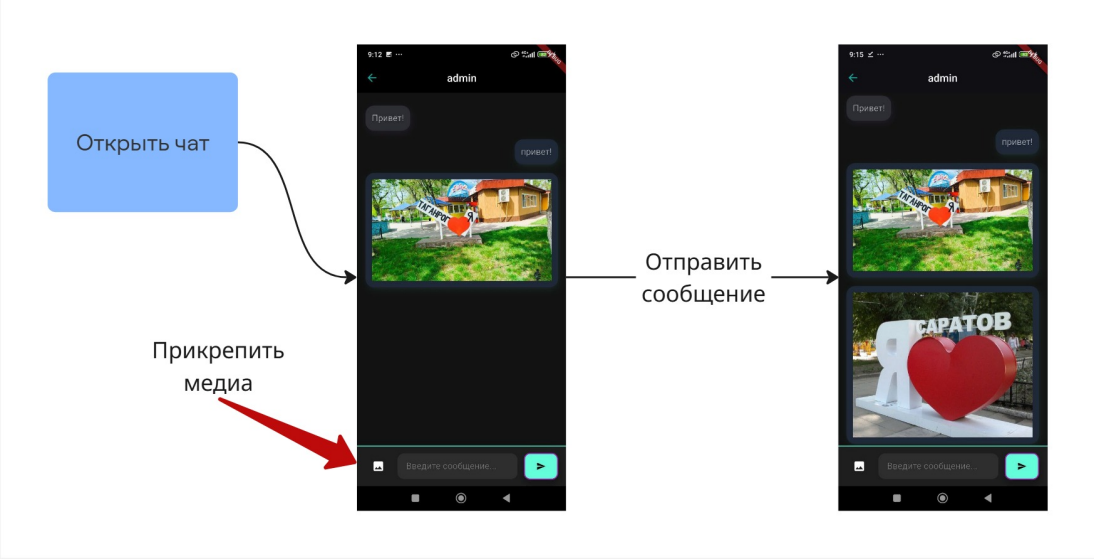

Рисунок 6 – Функционал чата: отправка сообщений и медиа

На этапе проектирования была предусмотрена возможность наполнения базы вопросов через интерфейс администратора. Как показано на рисунке 7, архитектура реализует проверку прав пользователя при запуске. При наличии административных прав активируется специальный экран, где можно вводить новые карточки. Таким образом, архитектура поддерживает разграничение ролей и безопасную изоляцию функционала.

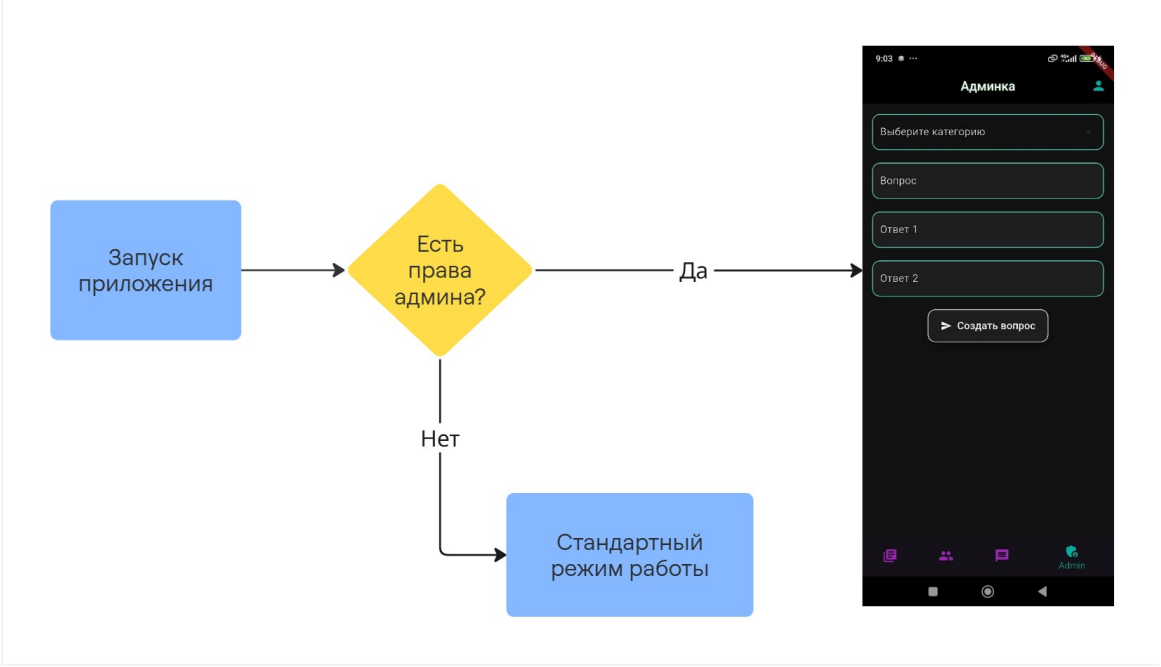

Рисунок 7 – Логика доступа к административной панели

Итоговая архитектура мобильного приложения представляет собой иерархическую структуру, основанную на пользовательских сценариях:

— экраны входа и регистрации — проверка подлинности;
— экран профиля — просмотр и редактирование данных;
— интерфейс вопросов — сбор данных для рекомендаций;
— рекомендации — выдача и фильтрация результатов;
— запросы на чат и чаты — коммуникация между пользователями;
— панель администратора — внутренняя поддержка наполнения базы.

Каждый из компонентов был спроектирован с учётом независимости, повторного использования и расширяемости. Такая архитектура обеспечивает устойчивость к росту функциональности и позволяет легко интегрировать рекомендательные алгоритмы на стороне сервера без необходимости внесения значительных изменений в клиентское приложение.

4.2 Проектирование архитектуры серверной части

Проектирование серверной части приложения для знакомств начинается с определения ключевых требований к системе. В первую очередь, необходимо обеспечить надёжное взаимодействие с мобильным клиентом, реализацию бизнес-логики приложения, управление пользовательскими данными, хранение медиафайлов, поддержку масштабируемой подсистемы рекомендаций, а также гибкость при развёртывании и сопровождении.

Для обмена данными между клиентским и серверным слоями требуется простой, расширяемый и широко поддерживаемый протокол. Учитывая эти факторы, архитектура взаимодействия строится на основе REST. Это решение обеспечивает стандартизованный обмен информацией через HTTP, совместимость с мобильными платформами и широкую поддержку в инструментах автоматизации.

При проектировании интерфейсов важно предусмотреть их документирование и проверку. Для этих целей подходит спецификация OpenAPI, поддерживающая автоматическую генерацию описания REST-эндпоинтов. Интеграция со Swagger позволяет предоставить наглядную, интерактивную документацию, упрощающую как разработку, так и тестирование API.

Серверная часть должна хранить структурированные пользовательские данные, такие как анкеты, результаты тестов, предпочтения, заявки на чат. Все эти данные обладают чёткой схемой и требуют транзакционной целостности.

Поэтому в качестве основной системы управления базами данных обоснован выбор реляционной СУБД. Среди доступных решений наиболее оптимальным по сочетанию стабильности, расширяемости и соответствию промышленным стандартам является PostgreSQL. Эта СУБД поддерживает богатый набор функций, включая работу с JSON, индексацию, расширения, и хорошо интегрируется с инструментами миграций.

Поддержка непрерывного развития и возможность безопасного обновления схемы базы данных требует внедрения системы управления миграциями. Это позволяет отслеживать изменения, выполнять откаты и обеспечивать согласованность структуры данных во всех окружениях. Исходя из популярности, удобства и хорошей интеграции со Spring Boot, в качестве системы миграций выбирается Liquibase.

Поскольку приложение позволяет пользователям загружать изображения профиля и отправлять вложения в чатах, необходимо предусмотреть механизм хранения бинарных данных. Хранение таких файлов непосредственно в базе данных было бы неэффективным и плохо масштабируемым. Вместо этого архитектура должна предусматривать использование объектного хранилища. Оно обеспечивает отдельное хранение медиафайлов с возможностью доступа через HTTP-интерфейс. В качестве подходящего решения выбирается MiniO — объектное хранилище с S3-совместимым API, которое легко разворачивается локально или в облачной среде и обеспечивает высокую отказоустойчивость.

Особое внимание в архитектуре уделяется подсистеме рекомендаций. Она требует специализированной обработки данных, может использовать иные языки программирования, библиотеки машинного обучения, и нуждается в независимом масштабировании. В этой связи было принято решение спроектировать рекомендательную систему как отдельный сервис. Это соответствует принципам микросервисной архитектуры и обеспечивает логическую и техническую изоляцию. Взаимодействие между основной серверной частью и рекомендательным сервисом происходит через HTTP-клиент, реализованный в модуле client.

Общая структура серверного приложения организована на основе модульного подхода с логическим разделением по уровням ответственности. Это упрощает поддержку кода, повышает читаемость и способствует соблюдению принципов SOLID. Каждый модуль отвечает за строго определённую функцию в рамках архитектуры.

Общая структура серверного приложения организована на основе модульного подхода, обеспечивающего логическое разделение по уровням ответственности. Это облегчает поддержку, расширение и тестирование системы.

- controller — REST-контроллеры, принимающие HTTP-запросы, валидирующие входные данные и передающие их в бизнес-логику;
- service — реализация бизнес-правил, координация взаимодействия между слоями и внешними сервисами;
- repository — интерфейсы доступа к базе данных на основе Spring Data JPA;
- entity — JPA-сущности, отражающие структуру таблиц и связи между ними;
- dto — модели передачи данных между слоями и при взаимодействии с клиентом;
- client — модуль HTTP-взаимодействия с внешней рекомендательной системой;
- config — конфигурационные классы, включая настройки безопасности, CORS, Swagger и подключение к внешним сервисам;
- exception — иерархия пользовательских исключений и глобальная обработка ошибок;
- mapper — преобразование между сущностями и DTO;
- utils — вспомогательные методы и классы общего назначения.

Такое разделение позволяет обеспечить читаемость, переиспользуемость и модульность, а также упрощает интеграцию дополнительных компонентов в архитектуру.

Подобная структура способствует высокой модульности, улучшает читаемость проекта и упрощает тестирование. Чёткое разграничение ответственности между слоями позволяет удобно масштабировать приложение, добавлять новые функции и интегрировать внешние сервисы без нарушения архитектурной целостности.

Для обеспечения воспроизводимости окружения и удобства развертывания на различных машинах проектируется контейнеризированная инфраструктура. Это особенно важно при наличии нескольких сервисов (основного API, рекомендательной системы, базы данных, хранилища медиафайлов). В этих условиях использование Docker становится естественным выбором. С его помо-

щью можно описать каждый компонент как независимый контейнер, настроить их взаимодействие в рамках одной сети и обеспечить надёжное воспроизводимое развёртывание, как в процессе локальной разработки, так и в продуктивной среде.

Таким образом, архитектура серверной части проектируется на основе требований надёжности, масштабируемости и гибкости. В ходе анализа выбираются технологии и подходы, соответствующие поставленным задачам. Итоговая архитектура включает REST-интерфейс, документацию через OpenAPI, использование PostgreSQL с миграциями Liquibase, объектное хранилище MiniO для медиафайлов, отдельный микросервис рекомендаций и контейнеризацию с помощью Docker. Эти решения образуют устойчивую и расширяемую платформу, готовую к дальнейшему развитию.

5 Реализация приложения для знакомств

5.1 Реализация кроссплатформенной клиентской части

Клиентская часть мобильного приложения для знакомств была разработана с использованием фреймворка Flutter и языка программирования Dart. Выбор данной технологии обусловлен ее способностью обеспечивать кроссплатформенную разработку с единой кодовой базой для операционных систем iOS и Android, а также высокой производительностью и гибкостью в создании пользовательских интерфейсов. Основной задачей при разработке клиентской части являлось создание интуитивно понятного, отзывчивого и функционального интерфейса, обеспечивающего комфортное взаимодействие пользователя с системой. Архитектура приложения ориентирована на модульность и переиспользование компонентов, что упрощает поддержку и дальнейшее развитие проекта.

Точкой входа в приложение является файл main.dart, который инициализирует корневой виджет и определяет глобальные настройки, такие как тема оформления и система навигации. Этот процесс инициализации и определения маршрутов можно увидеть в следующем фрагменте кода. В приложении реализована единая темная тема для консистентного визуального восприятия на всех экранах. Маршрутизация между экранами осуществляется с помощью именованных маршрутов, что позволяет структурировать навигацию и упрощает переходы.

```
// Фрагмент main.dart
class MyApp extends StatelessWidget {
    const MyApp({super.key});
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
        title: 'Bim Bim App',
        theme: ThemeData(
        brightness: Brightness.dark,
        scaffoldBackgroundColor: //...
        ),
        initialRoute: '/login',
        routes: {
            '/login': (context) => const LoginScreen(),
            '/register': (context) => const RegisterScreen(),
            '/home': (context) => const MainScreen(),
            '/messenger': (context) => const MessengerPage(),
```

```
            '/editProfile': (context) => const EditProfilePage(),
        },
        );
    }
}
```

Взаимодействие с серверной частью осуществляется через специально разработанный сервис ApiClient, расположенный в services/api_client.dart. Этот класс инкапсулирует логику отправки HTTP-запросов (GET, POST, PUT, DELETE), а также загрузки файлов на сервер. Базовый URL для всех API-запросов вынесен в файл констант constants/constants.dart.

Ключевым аспектом безопасности при взаимодействии с API является использование JWT-токенов. После успешной аутентификации пользователя сервер возвращает JWT-токен, который сохраняется в локальном хранилище устройства с помощью пакета shared_preferences. ApiClient автоматически извлекает этот токен и добавляет его в заголовки всех последующих защищенных запросов в виде «Authorization: Bearer <token> ». Пример реализации этого механизма представлен ниже.

```
// Фрагмент services/api_client.dart
Future<Map<String, String>> _getHeaders({Map<String, String>? extraHeaders}) async {
    final prefs = await SharedPreferences.getInstance();
    final token = prefs.getString('jwt_token');
    final headers = <String, String>{
        'Content-Type': 'application/json',
        if (token != null) 'Authorization': 'Bearer ' + token,
        ...?extraHeaders,
    };
    return headers;
}

Future<http.Response> post(String endpoint,
    {Object? body, Map<String, String>? headers}) async {
    final fullHeaders = await _getHeaders(extraHeaders: headers);
    final url = Uri.parse(endpoint);
    return http.post(url, headers: fullHeaders, body: body);
}
```

Модуль аутентификации включает экраны входа (screens/login_screen.dart) и регистрации (screens/register_screen.dart). Экран входа позволяет пользовате-

лю ввести свои учетные данные, которые затем отправляются на сервер через ApiClient на эндпоинт /auth/login. В случае успеха, полученный JWT-токен сохраняется, и пользователь перенаправляется на главный экран приложения. Логика этого процесса показана в следующем фрагменте кода.

```
// Фрагмент screens/login_screen.dart
Future<void> _login() async {
    final String username = _usernameController.text;
    final String password = _passwordController.text;
    // ...
    try {
        final response = await _apiClient.post(
        '$baseUrl/auth/login',
        body: jsonEncode({'username': username, 'password': password})
        );

        if (response.statusCode == 200) {
            final Map<String, dynamic> responseData = jsonDecode(response.body);
            if (responseData.containsKey('token')) {
                final String token = responseData['token'];
                final prefs = await SharedPreferences.getInstance();
                await prefs.setString('jwt_token', token);
                Navigator.pushReplacementNamed(context, '/home');
            } // ...
        } // ...
    } catch (e) { /* ... */ }
}
```

Главный экран приложения (screens/main_screen.dart) служит центральным узлом навигации после аутентификации. Он содержит BottomNavigationBar для переключения между основными разделами: «Вопросы», «Рекомендации», «Чаты» и «Админка» (для администраторов).

Управление профилем пользователя реализовано на странице screens/pages/-profile_page.dart. Здесь отображается информация о пользователе, предоставляется возможность загрузки нового аватара и перехода к редактированию профиля. Загрузка аватара, механизм которой приведен ниже, осуществляется методом uploadMultipart класса ApiClient.

```
// Фрагмент screens/pages/profile_page.dart
Future<void> _uploadAvatarToBackend(File imageFile) async {
```

```
    try {
        final file = await http.MultipartFile.fromPath('image', imageFile.path);
        final response = await _apiClient.uploadMultipart(
          endpoint: '$baseUrl/user/updateAvatar',
          files: [file],
          fields: {'type': 'image'}
        );
        if (response.statusCode == 200) {
            _fetchUserData();
        } // ...
    } catch (e) { /* ... */ }
}
```

Важной частью функционала является система подбора партнеров, основанная на ответах пользователей на вопросы в разделе «Тесты» (screens/pages/-tests_page.dart). Пользователи отвечают на вопросы, свайпая карточки. Использование виджета SwipeCards для этой цели демонстрируется в коде ниже. Эти ответы отправляются на сервер и используются для формирования профиля интересов.

```
// Фрагмент screens/pages/tests_page.dart
void _initializeSwipeItems() {
    if (_questions.isNotEmpty) {
        _swipeItems = _questions.map((question) {
            QuestionItem questionItem = QuestionItem(
              id: question['id'].toString(),
              content: question['content'],
              answerLeft: question['answerLeft'],
              answerRight: question['answerRight'],
            );
            return SwipeItem(
              content: questionItem,
              likeAction: () => _onAnswer(questionItem.id, 1),
              nopeAction: () => _onAnswer(questionItem.id, -1),
              superlikeAction: () => _onAnswer(questionItem.id, 0),
            );
        }).toList();
        setState(() {
            _matchEngine = MatchEngine(swipeItems: _swipeItems);
        });
    }
```

На странице «Рекомендации» (screens/pages/people_page.dart) отображаются профили других пользователей с указанием процента «сходства». Загрузка таких рекомендаций и их последующее отображение в виде списка проиллюстрированы в следующем фрагменте.

```
// Фрагмент screens/pages/people_page.dart
Future<void> _fetchPeople(int? categoryId) async {
    // ...
    final response = await _apiClient.get('$baseUrl/matching/$categoryId');
    if (response.statusCode == 200) {
        final data = json.decode(response.body) as List<dynamic>;
        setState(() {
            _people = data.map((e) => e as Map<String, dynamic>).toList();
            // ...
        });
    }
    // ...
}
ListView.builder(
    itemCount: sortedPeople.length,
    itemBuilder: (context, index) {
        final person = sortedPeople[index];
        return ListTile(
          leading: CircleAvatar(backgroundImage: NetworkImage(person['avatar'])),
          title: Text(person['username']),
          subtitle: Text('Сходство: ${person['similarity']}%'),
           // ...
        );
    },
)
```

Модуль обмена сообщениями состоит из списка чатов (screens/pages/-messenger_page.dart) и экрана самого чата (screens/pages/chat_page.dart). Экран чата отображает историю переписки и позволяет отправлять текстовые сообщения и изображения. Для обработки реального времени и получения новых сообщений используется периодический опрос сервера (Timer.periodic). Реализация этого механизма обновления показана ниже.

```
// Фрагмент screens/pages/chat_page.dart
@override
void initState() {
```

```
        super.initState();
        _loadMessages();
        _updateTimer = Timer.periodic(const Duration(seconds: 3), (timer) {
            if (mounted) {
                _loadMessages();
            }
        });
}
Future<void> _loadMessages() async {
        // ...
        final response = await _apiClient.get('$baseUrl/chat/${widget.chatId}/messages');
        // ...
}
```

Таким образом, клиентская часть мобильного приложения для знакомств реализована с использованием современных подходов и инструментов фреймворка Flutter. Особое внимание уделено структурированию кода, обеспечению безопасности передачи данных через JWT-токены и созданию удобного пользовательского интерфейса для всех ключевых функций приложения.

5.2 Реализация масштабируемой серверной части

Серверная часть мобильного приложения для знакомств разработана с использованием фреймворка Spring Boot, который был выбран благодаря его способности обеспечивать быструю разработку, встроенной поддержке множества технологий и обширному сообществу. Spring Boot упрощает создание автономных производственных приложений на основе Spring, минимизируя необходимость в сложной конфигурации.

Архитектура серверной части следует классической многоуровневой модели, включающей уровень контроллеров (Controller), сервисов (Service) и репозиториев (Repository), что обеспечивает четкое разделение ответственности и повышает тестируемость и поддерживаемость кода.

Контроллеры отвечают за обработку входящих HTTP-запросов, их валидацию и передачу данных на уровень сервисов. Они определяют API эндпоинты приложения. Например, контроллер AuthController обрабатывает запросы, связанные с аутентификацией и регистрацией пользователей, как показано в следующем фрагменте кода, где определяется эндпоинт для входа пользователя.

```
@RestController
```

```
@RequiredArgsConstructor
@RequestMapping("/api/auth")
public class AuthController {
    private final UserServiceImpl userService;
    // ... другие методы ...
    @PostMapping("/login")
    public JwtDto loginUser(@RequestBody UserLoginRequest userLoginRequest) {
        return userService.loginUser(userLoginRequest);
    }
}
```

Уровень сервисов инкапсулирует основную бизнес-логику приложения. Сервисы координируют взаимодействие между контроллерами и репозиториями, выполняют операции над данными и реализуют специфические для домена правила. Пример реализации метода loginUser в классе UserServiceImpl, который проверяет учетные данные пользователя и генерирует JWT-токен, представлен ниже.

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService {
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final JwtUtils jwtUtils;
    // ... другие зависимости ...
    @Override
    public JwtDto loginUser(UserLoginRequest userLoginRequest) {
        Optional<User> user = userRepository
            .findByUsername(userLoginRequest.username());
        if (user.isEmpty()) {
            throw new UnauthorizedException("Username not found");
        }
        if (!passwordEncoder.matches(userLoginRequest.password(),
            user.get().getPassword())) {
            throw new UnauthorizedException("Wrong password");
        }
        return jwtUtils.generateToken(user.get().getUsername(),
            user.get().getId(), user.get().getRoles());
    }
    // ... другие методы ...
}
```

Для взаимодействия с базой данных используется Spring Data JPA, который значительно упрощает создание уровня доступа к данным. Интерфейсы репозиториев, такие как UserRepository, наследуются от JpaRepository, что автоматически предоставляет стандартные CRUD-операции и возможность определения кастомных запросов.

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

Модель данных представлена JPA-сущностями (Entities), такими как User, Category, Question, Chat, Message. Эти классы аннотированы для маппинга на таблицы реляционной базы данных. Сущность User, например, содержит информацию о пользователе, его учетные данные и связи с другими сущностями.

```
@Getter
@Setter
@Entity
@Table(name="users")
public class User extends AbstractEntity {
    @Column(nullable = false, unique = true)
    private String username;
    @Column(nullable = false)
    private String password;
    // ... другие поля и связи ...
}
```

Для обмена данными между клиентом и сервером, а также между различными слоями приложения, активно используются объекты передачи данных (DTO). Они представляют собой простые Java-классы (часто реализуемые как records), которые определяют структуру данных для запросов и ответов. Пример DTO для запроса на вход пользователя UserLoginRequest:

```
public record UserLoginRequest(String username, String password) {
}
```

Безопасность приложения обеспечивается с помощью Spring Security и механизма аутентификации на основе JSON Web Tokens (JWT). JWT-токены используются для аутентификации пользователей после успешного входа в систему, позволяя им получать доступ к защищенным ресурсам. Центральным

элементом системы аутентификации является фильтр JwtAuthenticationFilter. При каждом запросе он извлекает JWT-токен из заголовка Authorization. Если токен присутствует и валиден, из него извлекаются данные пользователя (имя, идентификатор, роли), на основе которых создается объект аутентификации и помещается в SecurityContextHolder, делая пользователя доступным для последующих компонентов системы. Логика работы фильтра показана в следующем фрагменте кода.

```
@Override
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response,
        FilterChain filterChain) throws ServletException, IOException {
        String authHeader = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
                JwtDto jwtDto = // ... извлечение токена;
                if (jwtUtils.validateToken(jwtDto)) {
                        // ... извлечение необходимых параметров из токена
                        UsernamePasswordAuthenticationToken authentication =
                                new UsernamePasswordAuthenticationToken(
                                        userDetails, null, authorities
                                );
                        SecurityContextHolder.getContext().setAuthentication(authentication);
                }
        }
        filterChain.doFilter(request, response);
}
```

Генерация и валидация токенов осуществляется классом JwtUtils, который использует секретный ключ и заданное время жизни токена, настраиваемые в конфигурационном файле application.yaml. Пример метода генерации токена:

```
public class JwtUtils {
        private String secretKey;
        private Long expirationTime;
        public JwtDto generateToken(String username, Long id, String roles) {
                return new JwtDto(Jwts.builder()
                        .setSubject(username)
                        .claim("id", id)
                        .claim("roles", roles)
                        .setIssuedAt(new Date())
                        .setExpiration(new Date(System.currentTimeMillis() + expirationTime))
```

```
                .signWith(SignatureAlgorithm.HS512, secretKey)
                .compact());
    }
    // ... другие методы валидации и извлечения данных ...
}
```

Конфигурация безопасности Spring Security определяется в классе SecurityConfig. В фрагменте кода ниже показано, как настраиваются правила доступа к различным эндпоинтам, отключаются стандартные механизмы (например, CSRF, form login), и JwtAuthenticationFilter встраивается в цепочку фильтров.

```
@Bean
public SecurityFilterChain securityFilterChain(
    JwtAuthenticationFilter jwtAuthenticationFilter,
    HttpSecurity http) throws Exception {
    http
        .csrf(AbstractHttpConfigurer::disable)
        .formLogin(AbstractHttpConfigurer::disable)
        .sessionManagement(config -> config.sessionCreationPolicy(
            SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/auth/**").permitAll()
            .requestMatchers("/api-docs/**").permitAll()
            // ... другие разрешенные пути ...
            .anyRequest().authenticated()
        )
        .addFilterBefore(jwtAuthenticationFilter,
            UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

Разграничение доступа на уровне методов контроллеров или сервисов осуществляется с использованием аннотаций, таких как @PreAuthorize("hasRole('ADMIN')"), что позволяет гибко управлять правами доступа к отдельным операциям.

Для хранения пользовательских изображений (аватары, изображения в чатах) используется интеграция с S3-совместимым хранилищем MinIO. Сервис ImageServiceImpl отвечает за загрузку файлов в MinIO, генерацию уникальных имен файлов и предоставление URL для доступа к ним. Загрузка происходит через MinioClient, настроенный в MinioConfig.

Для реализации функции подбора подходящих пользователей (мэтчинга) серверная часть приложения интегрируется с внешним специализированным сервисом рекомендаций. Взаимодействие с этим сервисом осуществляется посредством HTTP-клиента, реализованного с использованием декларативного подхода Spring. Интерфейс MatchingClient определяет контракт взаимодействия, используя аннотацию @PostExchange для указания эндпоинта и типа запроса. Пример объявления клиента представлен ниже.

```
public interface MatchingClient {
    @PostExchange("/api/matching")
    List<MatchingResponse> getMatching(
    @RequestBody MatchingRequest request);
}
```

Конфигурация данного клиента, включая базовый URL внешнего сервиса, вынесена в отдельный класс свойств MatchingClientProperties, значения для которого загружаются из основного конфигурационного файла приложения application.yaml. Это обеспечивает гибкость настройки адреса сервиса рекомендаций без необходимости изменения кода.

```
@Data
@Configuration
@ConfigurationProperties(prefix = "rest.client.matching",
ignoreUnknownFields = false)
public class MatchingClientProperties {
    private String baseUrl;
}
```

Создание и настройка экземпляра MatchingClient происходит в конфигурационном классе RestClientConfig. В коде ниже показано, как используется RestClient.Builder для базовой настройки HTTP-клиента (например, указания baseUrl из MatchingClientProperties) и HttpServiceProxyFactory для создания прокси-объекта, реализующего интерфейс MatchingClient. Такой подход позволяет абстрагироваться от низкоуровневых деталей выполнения HTTP-запросов, сосредотачиваясь на бизнес-логике.

```
@Bean
public MatchingClient matchingClient(RestClient.Builder builder,
MatchingClientProperties properties) {
```

```
        RestClient restClient = builder
                .baseUrl(properties.getBaseUrl())
                .build();
        RestClientAdapter adapter = RestClientAdapter
                .create(restClient);
        HttpServiceProxyFactory factory = HttpServiceProxyFactory
                .builderFor(adapter)
                .build();
        return factory.createClient(MatchingClient.class);
}
```

При необходимости получения рекомендаций, соответствующий сервис серверной части (MatchingServiceImpl) подготавливает объект MatchingRequest, содержащий данные о текущем пользователе, других пользователях и их ответах на вопросы. Этот объект передается методу getMatching клиента MatchingClient, который выполняет HTTP-запрос к внешнему сервису и возвращает список подходящих кандидатов в виде объектов MatchingResponse.

Управление схемой базы данных и ее миграциями осуществляется с помощью Liquibase. Изменения схемы описываются в XML или YAML файлах, что обеспечивает версионирование и контролируемое обновление структуры БД. Пример изменения схемы для создания таблицы категорий:

```
databaseChangeLog:
  - changeSet:
        id: add_base_category_and_questions
          author: soup
          changes:
                - insert:
                  tableName: category
                  columns:
                        - column:
                          name: name
                          value: 'BaseCategory'
                        - column:
                          name: question_count
                          valueNumeric: 0
```

Все основные конфигурационные параметры приложения, такие как настройки подключения к БД, параметры JWT, адреса внешних сервисов и MinIO,

вынесены в файл application.yaml, что позволяет гибко настраивать приложение для различных окружений.

Таким образом, серверная часть приложения представляет собой структурированную систему, использующую современные подходы и технологии для обеспечения функциональности, безопасности и масштабируемости мобильного приложения для знакомств.

6 Разработка рекомендательной системы приложения для знакомств

6.1 Анализ предметной области и выбор данных для исследования

Разработка эффективной рекомендательной системы является ключевым аспектом современных мобильных приложений для знакомств. Целью данной работы является создание такой системы, основной особенностью которой является использование опросов с тернарными ответами (да, нет, пропустить) для формирования профилей пользователей и последующего подбора потенциальных партнеров. Данный подход предоставляет пользователям простой и интуитивно понятный способ выражения своих предпочтений и интересов, одновременно позволяя системе собирать структурированные данные для анализа.

Для моделирования и первичного тестирования предлагаемой рекомендательной системы был выбран публично доступный набор данных «Speed Dating Experiment» [30]. Этот датасет был собран профессорами Колумбийской бизнес-школы Рэйем Фисманом и Шиной Айенгар в ходе экспериментальных мероприятий по быстрым знакомствам, проводившихся с 2002 по 2004 год. Выбор данного набора данных обусловлен его высокой релевантностью поставленной задаче. Во-первых, он содержит информацию о демографических характеристиках участников, их интересах, самооценках по ключевым атрибутам (привлекательность, искренность, интеллект, веселье, амбициозность, общность интересов) и предпочтениях в потенциальном партнере. Во-вторых, что наиболее важно, датасет включает реальные исходы четырехминутных «первых свиданий» – решение участников о желании продолжить общение (переменные dec – решение участника, match – обоюдное согласие). Наличие как анкетных данных, так и результатов реальных взаимодействий позволяет оценить, насколько предпочтения, выраженные в опросах, коррелируют с фактическим выбором.

Ключевой особенностью датасета, имеющей значение для данного исследования, является его структура по «волнам» (переменная wave). Каждая «волна» представляет собой отдельное мероприятие по быстрым знакомствам, и участники взаимодействовали только с ограниченным подмножеством партнеров противоположного пола внутри своей волны. Это означает, что данные о взаимодействии каждой возможной пары пользователей в рамках всего датасета отсутствуют. Число участников в волнах варьировалось, что также вносит определенную разнородность в данные. Данная особенность структуры накладывает

ограничения на интерпретацию метрик качества рекомендаций, поскольку система может предложить объективно подходящего партнера, но если они не пересекались в рамках одной «волны», это взаимодействие не будет зафиксировано как совпадение, что потенциально занижает показатели точности и полноты.

Для адаптации данных к тернарному формату, используемому в разрабатываемом приложении, были выбраны признаки, отражающие интересы участников (например, sports, tvsports, exercise, dining и т.д.) и их самооценки/предпочтения по шести ключевым атрибутам (например, attr1_1 – важность привлекательности для себя, attr3_1 – самооценка привлекательности). Эти признаки, представленные в датасете оценками по 10-балльной шкале, были преобразованы в тернарный формат: значения от 1 до 3 интерпретировались как «нет» (-1), от 4 до 6 – как «пропустить» (0), а от 7 до 10 – как «да» (1). Такой подход позволяет симулировать механизм сбора предпочтений, предполагаемый в мобильном приложении.

Несмотря на упомянутые ограничения, датасет «Speed Dating Experiment» предоставляет ценную основу для первоначального исследования и обоснования базовых механизмов предлагаемой рекомендательной системы. Он позволяет проверить гипотезу о том, что сходство пользователей, выраженное через их ответы на тернарные опросы, может служить основой для формирования релевантных рекомендаций.

6.2 Разработка и сравнительный анализ моделей рекомендаций

Основной задачей данного этапа исследования являлась оценка эффективности построения рекомендаций, базирующихся на сходстве профилей пользователей, сформированных на основе их ответов на тернарные опросы. Для этого был проведен сравнительный анализ нескольких подходов к получению векторных представлений пользователей и последующему расчету их схожести.

Первоначальным шагом являлась загрузка и предварительная обработка данных из датасета «Speed Dating Experiment» [30]. Данные были загружены из CSV-файла с использованием библиотеки pandas. Одной из задач предварительной обработки было восстановление идентификаторов партнеров (pid) для некоторых записей, где они отсутствовали, на основе сопоставления номера волны (wave) и идентификатора партнера внутри волны (partner). Этот процесс показан в следующем фрагменте кода:

```
import pandas as pd

# ... загрузка df из CSV ...
df = pd.read_csv(csv_path, encoding='latin1')

id_lookup = df[['wave', 'id', 'iid']].drop_duplicates().set_index(['wave', 'id'])['iid']

df['pid'] = df.apply(lambda row: id_lookup.loc[(row['wave'], row['partner'])]
    if pd.isna(row['pid']) else row['pid'], axis=1)
```

Методология исследования включала несколько ключевых шагов. Во-первых, данные из датасета, касающиеся интересов участников и их оценок различных атрибутов, были преобразованы в тернарный формат. Как упоминалось ранее, значения от 1 до 3 были отображены в -1 («нет»), от 4 до 6 – в 0 («пропустить»), а от 7 до 10 – в 1 («да»). Пример функции для такого преобразования и ее применение к выбранному набору признаков (ternary_features, включающему интересы и самооценки атрибутов) приведен ниже:

```
def ternarize(value):
    if pd.isna(value) or (4 <= value <= 6):
        return 0
    elif value >= 7:
        return 1
    elif value <= 3:
        return -1
    return 0

ternary_features = ["sports", "tvsports", ..., "amb3_1"]
df_ternary = df["iid"] + ternary_features].copy()

for feature in ternary_features:
    df_ternary[feature] = df_ternary[feature].apply(ternarize)

df_ternary = df_ternary.drop_duplicates(subset=['iid'], keep='first')
```

Таким образом, для каждого уникального пользователя (идентифицируемого по iid) был сформирован вектор тернарных ответов. Далее, для уменьшения размерности и извлечения скрытых признаков из этих векторов, были применены и сравнены различные методы: сингулярное разложение (SVD), метод главных компонент (PCA), нелинейное снижение размерности с помощью

UMAP, а также автоэнкодер (Autoencoder) и вариационный автоэнкодер (VAE). Сходство между пользователями противоположного пола затем рассчитывалось с использованием косинусного расстояния между их полученными эмбеддингами. Функция для построения рекомендательной системы на основе PCA, например, имела следующую структуру:

```
def build_pca_recommender(df_ternary: pd.DataFrame, df_profiles: pd.DataFrame,
        n_components: int = 10):
        # df_profiles содержит уникальные iid и соответствующий пол (gender)
        features = df_ternary.drop(columns=['iid'])
        pca = PCA(n_components=n_components, random_state=42)
        reduced = pca.fit_transform(features)
        df_reduced = pd.DataFrame(reduced, index=df_ternary['iid'])

        genders = df_profiles.drop_duplicates('iid').set_index('iid')['gender']
        df_reduced['gender'] = genders

        def recommend(user_id: int, k: int = 10):
                if user_id not in df_reduced.index:
                        return []
                user_row = df_reduced.loc[user_id]
                user_vec = user_row.drop('gender').values.reshape(1, -1)
                user_gender = user_row['gender']
                opposite_gender = 1 if user_gender == 0 else 0

                candidates = df_reduced[df_reduced['gender'] ==
                        opposite_gender].drop(columns='gender')
                candidate_ids = candidates.index
                candidate_vectors = candidates.values

                similarities = cosine_similarity(user_vec, candidate_vectors)[0]
                top_indices = np.argsort(similarities)[::-1][:k]
                top_user_ids = candidate_ids[top_indices]
                return list(top_user_ids)
        return recommend
```

Для оценки качества полученных рекомендаций использовались стандартные метрики: точность на K позиции (Precision@K), полнота на K позиции (Recall@K), коэффициент попадания (HitRate@K) и покрытие (Coverage). В качестве данных о взаимодействиях (df_interactions) использовались записи из

исходного датасета, где было зафиксировано решение пользователя (dec) и фактическое совпадение (match).

Результаты сравнительного анализа моделей, использующих исключительно тернарные данные для построения эмбеддингов пользователей, представлены в таблице 1. В качестве метрик использовались Precision@K, Recall@K, HitRate@K и Coverage@K при K=20. Размерность латентного пространства (эмбеддинга) для моделей PCA, SVD, UMAP и VAE была установлена равной 10, для Автоэнкодера – 16.

Таблица 1 – Сравнение метрик качества для моделей на тернарных признаках (K=20)

Модель	**Precision@20**	**Recall@20**	**HitRate@20**	**Coverage@20**	**Размерность эмбеддинга**
SVD	0.0581	0.0735	0.6788	0.9728	10
PCA	0.0583	0.0731	0.6788	1.0000	10
UMAP	0.0595	0.0765	0.6751	0.9982	10
Автоэнкодер (AE)	0.0544	0.0707	0.6733	0.9819	16
Вариационный AE (VAE)	0.0535	0.0719	0.6842	0.9982	10

Из таблицы видно, что более простые методы снижения размерности, такие как PCA, SVD, и нелинейный UMAP, продемонстрировали в целом сопоставимую и в некоторых аспектах лучшую производительность по сравнению с нейросетевыми подходами (Автоэнкодер и VAE) на данном датасете. Например, UMAP показал наилучшие значения Precision@20 (0.0595) и Recall@20 (0.0765). PCA и SVD следовали близко (P@20: 0.0583/0.0581, R@20: 0.0731/0.0735). Автоэнкодер и VAE показали несколько более низкие значения по этим метрикам (P@20: 0.0544/0.0535, R@20: 0.0707/0.0719). Интересно, что Вариационный Автоэнкодер (VAE) продемонстрировал самый высокий показатель HitRate@20 (0.6842).

Возможное снижение производительности нейросетевых моделей по метрикам Precision и Recall может быть обусловлено относительно небольшим размером уникальных профилей для обучения эмбеддингов (551 пользователь) и разреженностью тернарных векторов признаков. Эти факторы могут затруднять обучение сложных нелинейных зависимостей нейронными сетями без риска переобучения. График зависимости метрик от значения K для модели PCA, показавшей сбалансированные результаты, приведен на рисунке 8.

В свою очередь, график для автоэнкодера (Рисунок 9) иллюстрирует тенденцию к несколько более низким значениям метрик, особенно при малых значениях K, по сравнению с PCA.

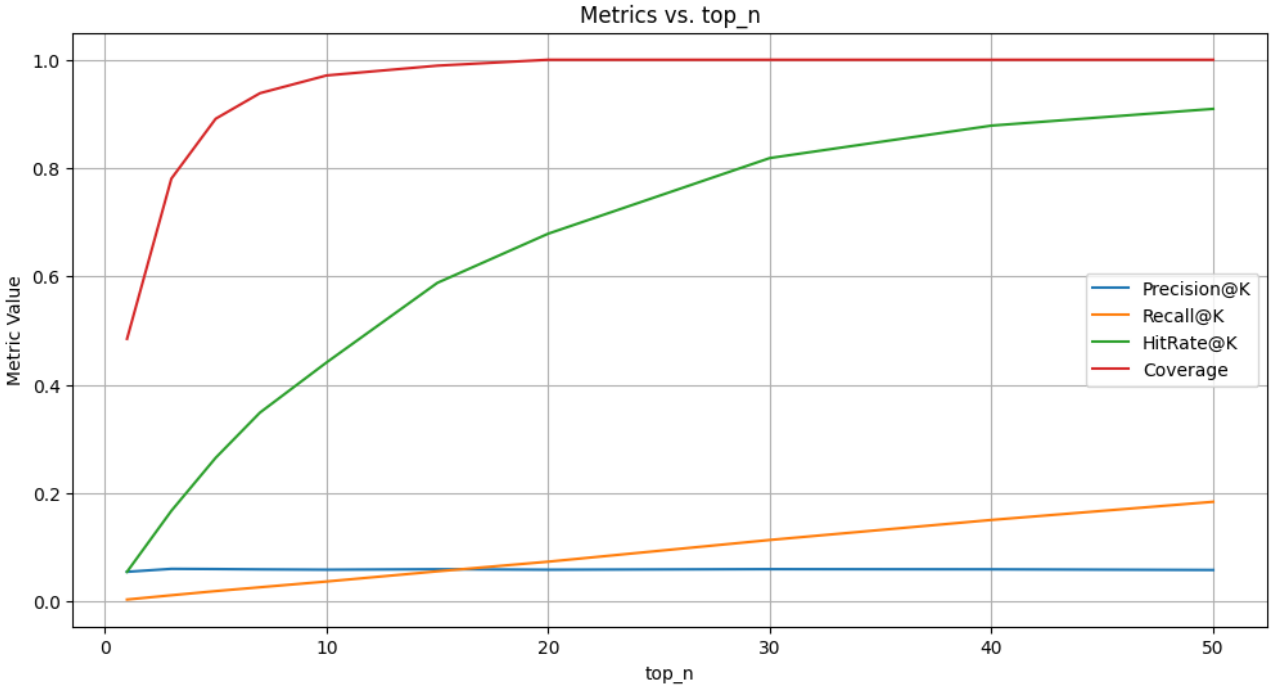

Рисунок 8 – Зависимость метрик Precision@K, Recall@K, HitRate@K и Coverage от числа рекомендаций K для модели PCA

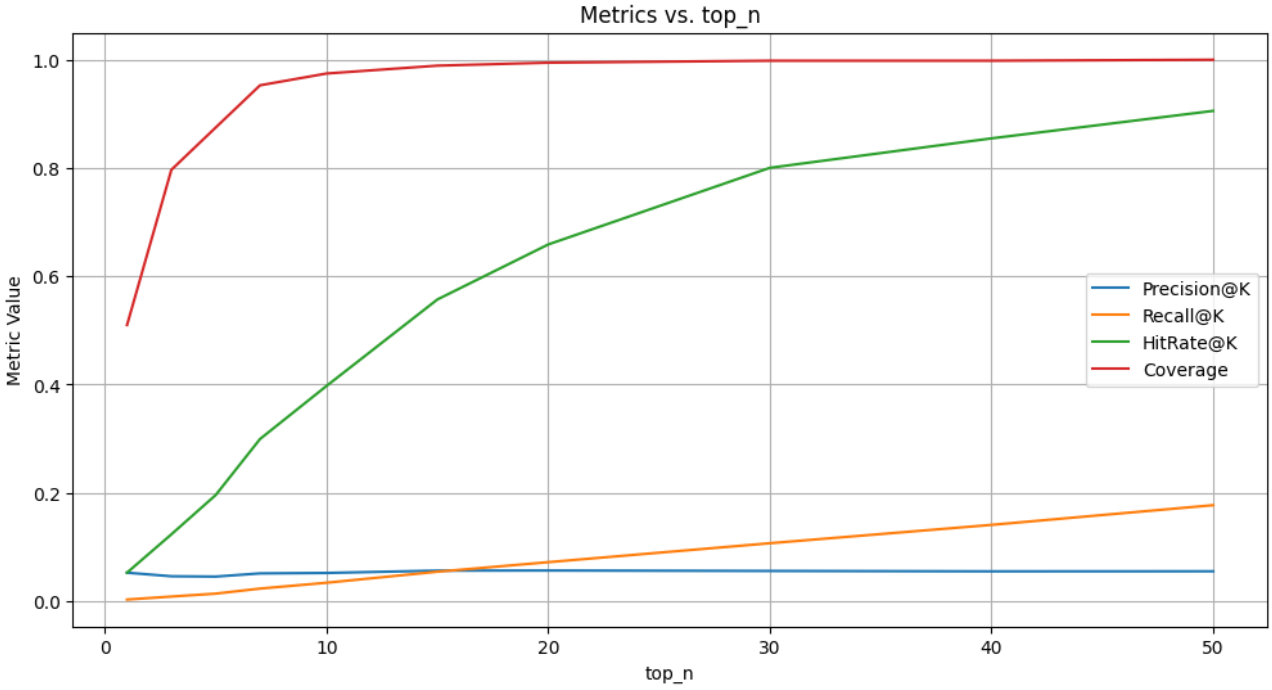

Рисунок 9 – Зависимость метрик Precision@K, Recall@K, HitRate@K и Coverage от числа рекомендаций K для автоэнкодера

При интерпретации этих значений важно учитывать ранее «волновую» структуру датасета. Поскольку участники взаимодействовали лишь с ограниченным числом партнеров в рамках своей волны, многие потенциально релевантные рекомендации не могли быть подтверждены как фактические совпаде-

ния, что искусственно занижает показатели Precision@K и Recall@K. В этом контексте более показательным является HitRate@K, который для K=20 варьировался от 0.6733 (Автоэнкодер) до 0.6842 (VAE). Это означает, что примерно для 67-68% пользователей система смогла найти хотя бы одно реальное совпадение (из тех, кто действительно встретился на мероприятии) среди топ-20 предложенных кандидатов. Показатель Coverage для всех моделей был близок к единице, что указывает на способность моделей рекомендовать почти всех доступных кандидатов.

Полученные умеренные значения метрик на одних лишь тернарных данных, а также тот факт, что более сложные нейросетевые модели не продемонстрировали явного преимущества на данном этапе, указывают на то, что, хотя опросы и несут полезную информацию, их одних может быть недостаточно для формирования высокоточных рекомендаций. Это создает предпосылки для разработки комбинированной (гибридной) рекомендательной системы.

В рамках данной дипломной работы предлагается расширить базовую систему, основанную на тернарных векторах, путем включения анализа текстовых описаний профилей пользователей. Предполагается, что текстовые описания могут содержать нюансы и детали, которые сложно выразить через ограниченный набор тернарных ответов. Для этого текстовые описания преобразуются в векторные представления. Итоговое сходство между пользователями рассчитывается как взвешенная сумма сходства их тернарных профилей и сходства их текстовых эмбеддингов: $Sim(u, v) = \alpha \cdot sim_{ternary}(u, v) + (1 - \alpha) \cdot sim_{text}(u, v)$, где $sim_{ternary}(u, v)$ – косинусное сходство тернарных векторов пользователей u и v, $sim_{text}(u, v)$ – косинусное сходство их текстовых эмбеддингов, а α – весовой коэффициент. Такой гибридный подход позволяет обогатить информацию о пользователях и потенциально повысить качество и релевантность предлагаемых кандидатур.

6.3 Реализация сервиса рекомендательной системы

Для практической реализации предложенной гибридной рекомендательной системы был разработан прототип в виде микросервиса на языке Python. Выбор Python обусловлен его богатой экосистемой библиотек для обработки данных, машинного обучения и веб-разработки, а также простотой и скоростью разработки. В качестве основного фреймворка для создания API был выбран FastAPI, известный своей высокой производительностью, асинхронной приро-

дой и удобной системой валидации данных на основе Pydantic. Для запуска приложения используется ASGI-сервер Uvicorn.

Архитектура сервиса спроектирована с учетом ключевой особенности – динамичности системы, то есть способности работать с произвольным набором вопросов для формирования тернарных профилей. Сервис не привязан к заранее определенной структуре опросов и вычисляет сходство на лету на основе данных, передаваемых в запросе.

Для определения структуры входящих запросов и исходящих ответов используются модели Pydantic, описанные в файле models.py. Модель MatchingRequest инкапсулирует данные о текущем пользователе, для которого запрашиваются рекомендации, списке всех доступных пользователей для подбора и перечне вопросов (QuestionMatchingRequest) с их идентификаторами и содержанием. Модель UserMatchingRequest содержит информацию о конкретном пользователе, включая его идентификатор, текстовое описание и словарь его тернарных ответов на вопросы. На выходе сервис возвращает список объектов MatchingResponse, каждый из которых содержит профильную информацию рекомендованного пользователя и вычисленный коэффициент сходства. Структура модели MatchingResponse представлена ниже:

```
from pydantic import BaseModel
from typing import Optional, List

class MatchingResponse(BaseModel):
    id: int
    avatar: Optional[str] = None
    gender: str
    username: str
    description: str
    similarity: float
```

Центральным элементом системы является класс DynamicRecommendationSystem, реализованный в файле recommender.py. При инициализации этого класса загружается предобученная модель «all-MiniLM-L6-v2» из библиотеки Sentence Transformers для преобразования текстовых описаний профилей в векторные представления (эмбеддинги). Выбор модели «all-MiniLM-L6-v2» обусловлен ее оптимальным балансом между качеством получаемых эмбеддингов и вычислительной эффективностью. Данная модель хорошо зарекомендовала себя

в задачах семантического сходства текстов, при этом являясь относительно компактной, что важно для производительности микросервиса. Имя модели задается в конфигурационном файле config.py. Также инициализируется весовой коэффициент α, по умолчанию равный 0.8. Этот коэффициент определяет относительный вклад тернарного сходства и текстового сходства в итоговую оценку. Значение 0.8 было выбрано эмпирически как начальная точка, предполагающая больший вес для явных предпочтений, выраженных через тернарные опросы, но при этом позволяющая текстовым описаниям вносить коррективы. В дальнейшем, на основе анализа реальных данных и A/B тестирования, этот коэффициент может быть более точно настроен.

Процесс получения текстового эмбеддинга для описания пользователя реализован в методе get_text_embedding. Если описание отсутствует, возвращается нулевой вектор соответствующей размерности, что обеспечивает корректную работу системы даже при неполных данных.

```
class DynamicRecommendationSystem:
    def __init__(self, alpha: float = 0.8):
    self.text_embedder = SentenceTransformer(TEXT_EMBEDDING_MODEL)
    self.text_embedding_dim = self.text_embedder.get_sentence_embedding_dimension()
    self.alpha = alpha

def _get_text_embedding(self, description: str) -> np.ndarray:
    if not description:
        return np.zeros(self.text_embedding_dim)
    return self.text_embedder.encode(description)
```

Для обработки тернарных ответов используется вспомогательная функция get_ternary_vector. Она принимает словарь ответов пользователя и упорядоченный список идентификаторов вопросов, на основе которых формирует numpy-вектор тернарных ответов. Важно, что порядок вопросов в этом векторе строго задан, что обеспечивает корректное сопоставление профилей разных пользователей.

```
def _get_ternary_vector(answers: dict, question_ids: List[int]) -> np.ndarray:
    return np.array([answers.get(qid, 0) for qid in question_ids], dtype=float)
```

Расчет косинусного сходства между двумя векторами (тернарными или текстовыми) выполняется функцией compute_similarity. В ней предусмотрена

проверка на наличие нулевых векторов, чтобы избежать ошибок деления на ноль; в таком случае сходство полагается равным нулю.

Основная логика формирования рекомендаций заключена в методе get_recommendations класса DynamicRecommendationSystem. Этот метод последовательно обрабатывает запрос: сначала извлекаются упорядоченные идентификаторы вопросов и данные основного пользователя. Затем для основного пользователя и каждого кандидата вычисляются тернарные и текстовые векторы. На основе этих векторов рассчитываются два типа сходства: тернарное и текстовое. Итоговое комбинированное сходство определяется по формуле $Sim(u, v) = \alpha \cdot sim_{ternary}(u, v) + (1 - \alpha) \cdot sim_{text}(u, v)$. После расчета сходства для всех кандидатов (кроме самого пользователя) формируется список объектов MatchingResponse, который сортируется по убыванию коэффициента сходства. Фрагмент, иллюстрирующий расчет комбинированного сходства и формирование ответа, показан ниже:

```
# ... получение main_ternary, main_text ...
for candidate in request.users:
    if candidate.id == main_user.id:
        continue
    candidate_ternary = _get_ternary_vector(candidate.answers, ordered_qids)
    candidate_text = self._get_text_embedding(candidate.description)
    ternary_sim = _compute_similarity(main_ternary, candidate_ternary)
    text_sim = _compute_similarity(main_text, candidate_text)
    combined_sim = self.alpha * ternary_sim + (1 - self.alpha) * text_sim
    recommendations.append(MatchingResponse(
        id=candidate.id,
        username=candidate.username,
        avatar=candidate.avatar,
        gender=candidate.gender,
        description=candidate.description,
        similarity=combined_sim * 100
    ))
    # ... сортировка recommendations ...
```

API сервиса реализован с использованием FastAPI в файле main.py. Определен единственный эндпоинт /api/matching, который принимает POST-запросы с телом в формате MatchingRequest. Эндпоинт выполняет базовую валидацию входных данных: проверяет наличие списка пользователей и, если предоставлены ответы на вопросы, то и наличие самих вопросов. Затем задача формирова-

ния рекомендаций делегируется объекту класса DynamicRecommendationSystem. В сервисе предусмотрена обработка стандартных исключений FastAPI и общих исключений для возврата корректных HTTP-ответов клиенту, что повышает надежность его работы. Код эндпоинта представлен следующим образом:

```
@app.post("/api/matching", response_model=List[MatchingResponse])
async def get_matching_users(request: MatchingRequest):
    try:
        if not request.users:
            return []
        if not request.questions and any(u.answers for u in request.users):
            raise HTTPException(status_code=400,
                detail="Answers provided but no questions defined to interpret")
        return dynamic_recommendation_system.get_recommendations(request)
    # ... обработка исключений ...
    except Exception as e:
    # ... логирование ошибки ...
    raise HTTPException(status_code=500, detail=f"Internal server error: {str(e)}")
```

Запуск микросервиса осуществляется с помощью Uvicorn, который обеспечивает асинхронное выполнение и способен обрабатывать большое количество одновременных запросов. Таким образом, разработанный прототип представляет собой функциональный и производительный сервис, способный предоставлять динамические рекомендации на основе комбинирования тернарных профилей и текстовых описаний пользователей. Модульная структура и использование современных фреймворков обеспечивают простоту его дальнейшего сопровождения и масштабирования. Весь разработанный в работе код можно увидеть в GitHub репозитории [31].

6.4 Преимущества и перспективы развития

Разработанная гибридная рекомендательная система, сочетающая анализ тернарных опросов и текстовых описаний профилей, обладает рядом преимуществ и открывает перспективы для дальнейшего развития и совершенствования.

Ключевым преимуществом предложенного подхода является его динамичность и гибкость. Система не привязана к фиксированному набору вопросов, что позволяет администраторам приложения легко изменять, добавлять или удалять вопросы в опросах без необходимости переобучения основной модели

сходства. Обработка тернарных векторов и текстовых эмбеддингов происходит на лету на основе актуального набора вопросов и описаний, передаваемых в запросе. Это обеспечивает высокую адаптивность системы к изменяющимся потребностям пользователей и эволюции самого приложения.

Другим важным преимуществом является простота и интерпретируемость взаимодействия для пользователя. Тернарные ответы (да, нет, пропустить) интуитивно понятны и не требуют от пользователя сложных оценок или размышлений, что снижает когнитивную нагрузку и повышает вероятность заполнения опросов. При этом пользователь имеет явный контроль над предоставляемыми данными. Относительно небольшое количество вопросов, необходимых для формирования первичного тернарного профиля, и простота их заполнения также минимизируют проблему «холодного старта» для новых пользователей, позволяя системе достаточно быстро начать генерировать осмысленные рекомендации.

Несмотря на продемонстрированные в ходе исследования на датасете «Speed Dating Experiment» умеренные, но осмысленные результаты базовых моделей на тернарных данных, важно отметить ограничения текущего исследования и прототипа. Основное ограничение связано с самим датасетом: его «волновая» структура не позволяет оценить взаимодействие всех возможных пар, что, как обсуждалось ранее, занижает метрики точности и полноты. Кроме того, текущая реализация гибридной системы использует простую линейную комбинацию для агрегации сходств с фиксированным коэффициентом α.

Тем не менее, разработанная система представляет собой прочный фундамент и открывает широкие перспективы для дальнейшего развития и исследований. Одним из главных направлений является сбор и использование исторических данных о реальных взаимодействиях пользователей внутри разработанного мобильного приложения, таких как лайки, мэтчи и характер общения после мэтча. Эти данные позволят обучать более сложные и персонализированные модели, например, на основе коллаборативной фильтрации или нейросетевых подходов, способных улавливать скрытые предпочтения.

Далее, текущий весовой коэффициент α в гибридной модели может быть оптимизирован с помощью A/B тестирования или методов машинного обучения для более точного взвешивания вклада тернарных и текстовых данных. Возможна также разработка более сложных нелинейных функций для агрега-

ции различных типов сходства.

Перспективным направлением является и динамическая адаптация самих опросов. Можно реализовать механизмы, которые бы предлагали пользователям наиболее релевантные или информативные вопросы на основе их предыдущих ответов или активности, что позволит более эффективно собирать данные. Кроме того, система может быть расширена за счет учета контекстуальных факторов, таких как время, геолокация или недавняя активность пользователя, а также анализа дополнительных аспектов профиля, например, фотографий или музыкальных предпочтений, с использованием соответствующих технологий. Внедрение механизмов обратной связи от пользователей на предложенные рекомендации также будет способствовать непрерывному улучшению модели.

Предложенная и реализованная в виде прототипа микросервиса рекомендательная система, основанная на динамическом анализе тернарных опросов и текстовых описаний, демонстрирует свою жизнеспособность и является перспективной отправной точкой. Ее гибкость и возможность учета различных аспектов пользовательского профиля, в сочетании с потенциалом для интеграции более сложных алгоритмов на основе собираемых данных, делают ее ценным компонентом для разрабатываемого мобильного приложения знакомств.

ЗАКЛЮЧЕНИЕ

В ходе работы был проведен анализ существующих решений и методов построения рекомендательных систем в сфере онлайн-знакомств. На основе результатов исследования была спроектирована масштабируемая и гибкая клиент-серверная архитектура мобильного приложения для знакомств с динамической гибридной рекомендательной системой.

Масштабируемость и гибкость полученного решения достигается за счет комбинации клиентской части на Flutter, серверной на Spring Boot и отдельного рекомендательного микросервиса на Python/FastAPI.

Анализ существующих решений и экспериментальное исследование на датасете «Speed Dating Experiment» подтвердили жизнеспособность предложенного подхода. Было показано, что даже относительно простые модели могут эффективно работать с тернарными данными, а метрика HitRate@K, являющаяся важным показателем в условиях специфической структуры данных, показывает осмысленные результаты. Разработанная система отличается высокой адаптивностью и интерпретируемостью, так как не привязана к фиксированному набору вопросов, что упрощает ее эволюцию и снижает когнитивную нагрузку на пользователей, способствуя решению проблемы «холодного старта».

Дальнейшее развитие разработанного решения предполагает сбор и анализ реальных пользовательских данных для обучения более сложных моделей, оптимизацию параметров гибридной системы и внедрение динамической адаптации опросников. Разработанное приложение и рекомендательная система представляют собой прочный фундамент для создания качественного и востребованного сервиса знакомств.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Wang, X. Data scarcity in recommendation systems: A survey / X. Wang, Y. Liu, M. Chen at al. // arXiv preprint arXiv:2312.10073. — 2023.

2 Jin, D. A survey on fairness-aware recommender systems / D. Jin, L. Wang, H. Zhang, Y. Zheng, W. Ding, F. Xia, S. Pan // arXiv preprint arXiv:2306.00403. — 2023.

3 Rudder, C. Okcupid: The math behind online dating [Электронный ресурс] / C. Rudder // AMS Graduate Student Blog. — 2013. — URL: https://blogs.ams.org/mathgradblog/2016/06/08/okcupid-math-online-dating/ (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

4 Carman, A. Finding love on a first data: Matching algorithms in online dating [Электронный ресурс] / A. Carman // Harvard Data Science Review. — 2021. — URL: https://hdsr.mitpress.mit.edu/pub/i4eb4e8b (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

5 Tinder Engineering Team,. Personalized user recommendations at tinder [Электронный ресурс] // Proceedings of the Machine Learning Conference). — San Francisco, USA: 2017. — URL: https://mlconf.com/sessions/personalized-user-recommendations-at-tinder-the-t/ (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

6 Zhao, Z. Weizhi zhang and yuanchen bei and liangwei yang and henry peng zou / Z. Zhao, W. Fan, J. Li at al. // arXiv preprint arXiv:2501.01945. — 2025.

7 Tinder,. Powering tinder — the method behind our matching [Электронный ресурс]. — 2022. — URL: https://www.help.tinder.com/hc/en-us/articles/7606685697037-Powering-Tinder-The-Method-Behind-Our-Matching4 (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

8 Resnick, B. The tinder algorithm, explained [Электронный ресурс] / B. Resnick // Vox. — 2019. — URL: https://www.vox.com/2019/2/7/18210998/tinder-algorithm-swiping-tips-dating-app-science (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

9 Wells, G. Hinge founder justin mcleod explains how the algorithm finds your match [Электронный ресурс] / G. Wells // Fortune. — 2024. — URL: https://fortune.com/2024/01/18/

hinge-ceo-justin-mcleod-interview-attractiveness-score-algorithm-rose-jail/ (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

10 Relationstips,. How eharmony matches are made: Inside the algorithm [Электронный ресурс] / Relationstips. — 2025. — URL: https://www.relationstips.com/how-eharmony-matches-are-made-inside-the-algorithm/ (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

11 Sugahara, K. Hierarchical matrix factorization for interpretable collaborative filtering / K. Sugahara, K. Okamoto // arXiv preprint arXiv:2311.13277. — 2023.

12 Zhou, Z. Contrastive collaborative filtering for cold-start item recommendation / Z. Zhou, L. Zhang, N. Yang // arXiv preprint arXiv:2302.02151. — 2023.

13 Lops, P. Content-based recommender systems: State of the art and trends / P. Lops, M. d. Gemmis, G. Semeraro // Recommender Systems Handbook. — 2011. — Pp. 73–105.

14 Zhang, S. Deep learning based recommender system: A survey and new perspectives / S. Zhang, L. Yao, A. Sun, Y. Tay // ACM Computing Surveys (CSUR). — 2019. — Vol. 52, no. 1. — Pp. 1–38.

15 Nabil, S. Demographic information combined with collaborative filtering for an efficient recommendation system / S. Nabil, M. Y. Chkouri, J. El Bouhdidi // International Journal of Electrical and Computer Engineering (IJECE). — 2024. — Vol. 14, no. 5. — Pp. 5916–5925.

16 Beregovskaya, I. Review of clustering-based recommender systems / I. Beregovskaya, M. Koroteev // arXiv preprint arXiv:2109.12839. — 2021.

17 Nadimi-Shahraki, M. H. Cold-start problem in collaborative recommender systems: Efficient methods based on ask-to-rate technique / M. H. Nadimi-Shahraki, M. Bahadorpour // Journal of Computing and Information Technology. — 2014. — Vol. 22, no. 2. — Pp. 105–113.

18 Xiangnan, H. Neural collaborative filtering / H. Xiangnan, L. Liao, H. Zhang, L. Nie, X. Hu, T.-S. Chua // arXiv preprint arXiv:1708.05031. — 2017.

19 Hidasi, B. Session-based recommendations with recurrent neural networks / B. Hidasi, A. Karatzoglou, L. Baltrunas, D. Tik // arXiv preprint arXiv:1511.06939. — 2016.

20 Kang, W. Self-attentive sequential recommendation / W. Kang, J. J. McAuley // arXiv preprint arXiv:1808.09781. — 2018.

21 van den Berg, R. Graph convolutional matrix completion / R. van den Berg, T. Kipf, M. Welling // arXiv preprint arXiv:1706.02263. — 2017.

22 Ruining, H. Visual bayesian personalized ranking from implicit feedback / H. Ruining, J. McAuley // arXiv preprint arXiv:1510.01784. — 2015.

23 Cano, E. Hybrid recommender systems: A systematic literature review / E. Cano, M. Morisio // Intelligent Data Analysis. — 2017. — Vol. 21, no. 6. — 1487–1524 p.

24 Guo, H. Deepfm: A factorization-machine based neural network for ctr prediction / H. Guo, R. Tang, Y. Ye, Z. Li, X. He // arXiv preprint arXiv:1703.04247. — 2017.

25 Recommender model evaluation: Offline vs. online [Электронный ресурс] // Shaped Blog. — 2023. — URL: https://www.shaped.ai/blog/evaluating-recommender-models-offline-vs-online-evaluation (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

26 Dhaduk, H. Mobile application architecture: Layers, types, principles, factors [Электронный ресурс] / H. Dhaduk // Simform Blog. — 2024. — URL: https://www.simform.com/blog/mobile-application-architecture/ (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

27 Client-server architecture – system design [Электронный ресурс] // GeeksforGeeks. — 2024. — URL: https://www.geeksforgeeks.org/client-server-architecture-system-design/ (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

28 Best practices - oauth for mobile apps [Электронный ресурс] // Curity. — 2025. — URL: https://curity.io/resources/learn/oauth-for-mobile-apps-best-practices/ (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

29 Zinkus, M. Data security on mobile devices: Current state of the art, open problems, and proposed solutions / M. Zinkus, T. M. Jois, M. Green // arXiv preprint arXiv:2105.12613. — 2021.

30 Fisman, R. Speed Dating Experiment Dataset [Электронный ресурс] / R. Fisman, S. S. Iyengar. — Kaggle. — URL: https://www.kaggle.com/datasets/annavictoria/speed-dating-experiment (Дата обращения 30.04.2025). Загл. с экр. Яз. англ.

31 Репозиторий с разработанным кодом [Электронный ресурс]. — URL: https://github.com/sour-soup/bim-bim-diploma (Дата обращения 30.05.2025).

ПРИЛОЖЕНИЕ А

Инициализация, конфигурация и безопасность серверного приложения

Файл: BimBimApplication.java (Точка входа в Spring Boot приложение)

```
package org.soursoup.bimbim;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class BimBimApplication {
    public static void main(String[] args) {
        SpringApplication.run(BimBimApplication.class, args);
    }

}
```

Файл: application.yaml (Основные настройки приложения)

```
spring:
  application:
    name: bim-bim
  datasource:
    url: jdbc:postgresql://localhost:5432/db
    username: db
    password: db
  jpa:
    hibernate:
      ddl-auto: validate

  liquibase:
    enabled: true
    change-log: classpath:/db/changelog/master-changelog.yaml

rest:
  client:
    matching:
      base-url: http://localhost:8000

jwt:
  secretKey:
```

```
    alskdjsdfhlaksdjffhladkfsfhldaksadfhladasdfhljlfhaklfkjadsfhjadkslffhdajlskffhjldkasdfhjkladasdfhjlka

  expirationTime: 3600000 # 1 hour in milliseconds

springdoc:
  swagger-ui:
    path: /api-docs

minio:
  bucket: images
  access-key: bimbimbambam
  secret-key: bimbimbambam
  url: http://192.168.47.16:9000
```

Файл: SecurityConfig.java (Конфигурация правил безопасности)

```
package org.soursoup.bimbim.config.security;

import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.
    EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configurers.
    AbstractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.
    UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(JwtAuthenticationFilter
```

```
    jwtAuthenticationFilter, HttpSecurity http) throws Exception {
        http
                .csrf(AbstractHttpConfigurer::disable)
                .formLogin(AbstractHttpConfigurer::disable)
                .sessionManagement(config -> config.sessionCreationPolicy(SessionCreationPolicy
    .STATELESS))
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/api/auth/**").permitAll()
                        .requestMatchers("/api-docs/**").permitAll()
                        .requestMatchers("/swagger-ui/**").permitAll()
                        .requestMatchers("/v3/api-docs/**").permitAll()
                        .anyRequest().authenticated()
                )
                .addFilterBefore(jwtAuthenticationFilter,
    UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Файл: JwtAuthenticationFilter.java (Фильтр для JWT-аутентификации)

```
package org.soursoup.bimbim.config.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.soursoup.bimbim.dto.JwtDto;
import org.soursoup.bimbim.utils.JwtUtils;
import org.springframework.http.HttpHeaders;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
```

```
16
17 import java.io.IOException;
18 import java.util.Arrays;
19 import java.util.List;
20
21 @Component
22 @RequiredArgsConstructor
23 public class JwtAuthenticationFilter extends OncePerRequestFilter {
24
25     private final JwtUtils jwtUtils;
26
27     @Override
28     protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response
       , FilterChain filterChain) throws ServletException, IOException {
29         String authHeader = request.getHeader(HttpHeaders.AUTHORIZATION);
30         if (authHeader != null && authHeader.startsWith("Bearer ")) {
31             String tokenValue = authHeader.substring(7);
32             JwtDto jwtDto = new JwtDto(tokenValue);
33
34             if (jwtUtils.validateToken(jwtDto)) {
35                 String username = jwtUtils.extractUsername(jwtDto);
36                 String roles = jwtUtils.extractRoles(jwtDto);
37                 Long id = jwtUtils.extractId(jwtDto);
38
39                 List<SimpleGrantedAuthority> authorities = Arrays.stream(roles.split(","))
40                         .map(role -> new SimpleGrantedAuthority("ROLE_" + role.trim().
       toUpperCase()))
41                         .toList();
42
43                 JwtUserDetails userDetails = new JwtUserDetails(id, username, roles, authorities
       );
44                 UsernamePasswordAuthenticationToken authentication =
45                         new UsernamePasswordAuthenticationToken(userDetails, null, authorities);
46
47                 SecurityContextHolder.getContext().setAuthentication(authentication);
48             }
49         }
50
51         filterChain.doFilter(request, response);
52     }
53 }
```

ПРИЛОЖЕНИЕ Б

Ключевые сущности модели данных

Файл: User.java (Сущность пользователя)

```
package org.soursoup.bimbim.entity;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

@Getter
@Setter
@Entity
@Table(name="users")
public class User extends AbstractEntity {
    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private String gender;

    @Column
    private String description;

    @Column
    private String avatar;

    @Column(nullable = false)
    private String roles = "USER";

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true
    )
    private List<Answer> answers;

    @OneToMany(mappedBy = "fromUser", cascade = CascadeType.ALL, orphanRemoval =
      true)
```

```
    private List<Chat> likesSent;

    @OneToMany(mappedBy = "toUser", cascade = CascadeType.ALL, orphanRemoval =
     true)
    private List<Chat> likesReceived;
}
```

Файл: Question.java (Сущность вопроса для анкеты)

```
package org.soursoup.bimbim.entity;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

@Getter
@Setter
@Entity
@Table
public class Question extends AbstractEntity {
    @Column(nullable = false)
    private String content;

    @Column(nullable = false)
    private String answerLeft;

    @Column(nullable = false)
    private String answerRight;

    @Column
    private String image;

    @OneToMany(mappedBy = "question", cascade = CascadeType.ALL, orphanRemoval =
     true)
    private List<Answer> answers;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", nullable = false)
    private Category category;
}
```

Файл: Chat.java (Сущность чата между пользователями)

```
package org.soursoup.bimbim.entity;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@Entity
@Table
public class Chat extends AbstractEntity {
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "from_user_id", nullable = false)
    private User fromUser;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "to_user_id", nullable = false)
    private User toUser;

    @Column(name = "to_user_confirmed", nullable = false)
    boolean toUserConfirmed = false;

    @Column(name = "is_canceled", nullable = false)
    boolean isCanceled = false;
}
```

ПРИЛОЖЕНИЕ В
Реализация подбора пользователей и чата

Файл: MatchingServiceImpl.java (Сервисный слой для механизма подбора пользователей)

```
package org.soursoup.bimbim.service.impl;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.soursoup.bimbim.client.MatchingClient;
import org.soursoup.bimbim.config.MinioConfig;
import org.soursoup.bimbim.dto.request.MatchingRequest;
import org.soursoup.bimbim.dto.response.MatchingResponse;
import org.soursoup.bimbim.entity.Answer;
import org.soursoup.bimbim.entity.Category;
import org.soursoup.bimbim.entity.Question;
import org.soursoup.bimbim.entity.User;
import org.soursoup.bimbim.exception.NotFoundException;
import org.soursoup.bimbim.repository.*;
import org.soursoup.bimbim.service.MatchingService;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

@Slf4j
@Service
@RequiredArgsConstructor
public class MatchingServiceImpl implements MatchingService {

    private final CategoryRepository categoryRepository;
    private final QuestionRepository questionRepository;
    private final AnswerRepository answerRepository;
    private final UserRepository userRepository;
    private final ChatRepository chatRepository;
    private final MatchingClient matchingClient;
    private final MinioConfig minioConfig;

```

```
37      @Override
38      public List<MatchingResponse> getMatching(Long userId, Long categoryId) {
39          Category category = categoryRepository.findById(categoryId)
40                  .orElseThrow(() -> new NotFoundException("Category not found"));
41
42          List<Question> questions = questionRepository.findAllByCategory(category);
43
44          User currentUser = userRepository.findById(userId)
45                  .orElseThrow(() -> new NotFoundException("User not found"));
46
47          List<User> connectedUserIds = chatRepository.findAllByFromUserOrToUser(
        currentUser, currentUser).stream()
48                  .flatMap(chat -> Stream.of(chat.getFromUser(), chat.getToUser()))
49                  .filter(user -> !user.equals(currentUser))
50                  .distinct().toList();
51
52          List<User> users = userRepository.findAll().stream()
53                  .filter(user -> !connectedUserIds.contains(user))
54                  .toList();
55
56          Map<Long, Map<Long, Long>> answerMap = buildAnswerMap(users, questions);
57
58          List<MatchingRequest.UserMatchingRequest> userMatchingRequests = users.stream()
59                  .map(user -> buildUserMatching(user, answerMap))
60                  .toList();
61
62          List<MatchingRequest.QuestionMatchingRequest> questionMatchingRequests =
        questions.stream()
63                  .map(this::buildQuestionMatching)
64                  .toList();
65
66          MatchingRequest matchingRequest = new MatchingRequest(
67                  userId,
68                  questionMatchingRequests,
69                  userMatchingRequests
70          );
71          log.info("Matching request: {}", matchingRequest);
72
73          var response = matchingClient.getMatching(matchingRequest);
74
75          log.info("Matching response: {}", response);
```

```
76          return response;
77      }
78
79      private Map<Long, Map<Long, Long>> buildAnswerMap(List<User> users, List<
        Question> questions) {
80          return users.stream()
81                  .collect(Collectors.toMap(User::getId, user -> buildAnswers(user, questions)));
82      }
83
84      private Map<Long, Long> buildAnswers(User user, List<Question> questions) {
85          List<Answer> answers = answerRepository.findAllByUser(user);
86
87          return answers.stream()
88                  .filter(answer -> questions.stream().anyMatch(q -> q.getId().equals(answer.
        getQuestion().getId())))
89                  .collect(Collectors.toMap(answer -> answer.getQuestion().getId(), Answer::
        getAnswer));
90      }
91
92      private MatchingRequest.UserMatchingRequest buildUserMatching(User user, Map<Long,
          Map<Long, Long>> answerMap) {
93          return new MatchingRequest.UserMatchingRequest(
94                  user.getId(),
95                  defineAvatar(user.getAvatar()),
96                  user.getGender(),
97                  user.getUsername(),
98                  user.getDescription(),
99                  answerMap.get(user.getId())
100         );
101     }
102
103     private MatchingRequest.QuestionMatchingRequest buildQuestionMatching(Question
        question) {
104         return new MatchingRequest.QuestionMatchingRequest(
105                 question.getId(),
106                 question.getContent(),
107                 question.getAnswerLeft(),
108                 question.getAnswerRight()
109         );
110     }
111
```

```
    private String defineAvatar(String avatar) {
        avatar = Optional.ofNullable(avatar).orElse("unknown.png");

        return minioConfig.getUrl() + "/" + minioConfig.getBucket() + "/" + avatar;
    }
}
```

Файл: ChatServiceImpl.java (Сервисный слой для функциональности чата)

```
package org.soursoup.bimbim.service.impl;

import lombok.RequiredArgsConstructor;
import org.soursoup.bimbim.dto.request.ImageRequest;
import org.soursoup.bimbim.dto.request.UpdateImageRequest;
import org.soursoup.bimbim.entity.Chat;
import org.soursoup.bimbim.entity.Message;
import org.soursoup.bimbim.entity.User;
import org.soursoup.bimbim.exception.ForbiddenException;
import org.soursoup.bimbim.exception.NotFoundException;
import org.soursoup.bimbim.repository.ChatRepository;
import org.soursoup.bimbim.repository.MessageRepository;
import org.soursoup.bimbim.repository.UserRepository;
import org.soursoup.bimbim.service.ChatService;
import org.soursoup.bimbim.service.ImageService;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@RequiredArgsConstructor
@Service
public class ChatServiceImpl implements ChatService {
    private final ChatRepository chatRepository;
    private final MessageRepository messageRepository;
    private final UserRepository userRepository;
    private final ImageService imageService;

    public List<Chat> getSentRequests(Long userId) {
        User user = userRepository.findById(userId)
                .orElseThrow(() -> new NotFoundException("User not found"));
        return chatRepository.findAllByFromUserAndToUserConfirmedFalse(user)
                .stream().filter(chat -> !chat.isCanceled()).toList();
```

```
34
35      }
36
37      public List<Chat> getPendingRequests(Long userId) {
38          User user = userRepository.findById(userId)
39                  .orElseThrow(() -> new NotFoundException("User not found"));
40          return chatRepository.findAllByToUserAndToUserConfirmedFalse(user)
41                  .stream().filter(chat -> !chat.isCanceled()).toList();
42      }
43
44      @Transactional
45      public void acceptChatRequest(Long chatId, Long userId) {
46          Chat chat = chatRepository.findById(chatId)
47                  .orElseThrow(() -> new NotFoundException("Chat request not found"));
48
49          if (!chat.getToUser().getId().equals(userId)) {
50              throw new ForbiddenException("Access denied: You are not allowed to accept this
     chat request");
51          }
52
53          chat.setToUserConfirmed(true);
54          chatRepository.save(chat);
55      }
56
57      @Transactional
58      public void declineChatRequest(Long chatId, Long userId) {
59          Chat chat = chatRepository.findById(chatId)
60                  .orElseThrow(() -> new NotFoundException("Chat request not found"));
61
62          if (!chat.getToUser().getId().equals(userId) && !chat.getFromUser().getId().equals(
     userId)) {
63              throw new ForbiddenException("Access denied: You are not allowed to accept this
     chat request");
64          }
65          chat.setCanceled(true);
66          chatRepository.save(chat);
67      }
68
69      @Override
70      @Transactional
71      public Message uploadImage(Long userId, Long chatId, UpdateImageRequest
```

```
    updateImageRequest) {
72      User user = userRepository.findById(userId)
73              .orElseThrow(() -> new NotFoundException("User not found"));
74
75      Chat chat = chatRepository.findById(chatId)
76              .orElseThrow(() -> new NotFoundException("Chat not found"));
77
78      if (!(chat.getFromUser().getId().equals(userId) || chat.getToUser().getId().equals(userId
    ))) {
79          throw new ForbiddenException("User is not part of the chat");
80      }
81
82      if (!chat.isToUserConfirmed()) {
83          throw new ForbiddenException("Chat is not confirmed");
84      }
85
86      String imageFilename = imageService.upload(new ImageRequest(updateImageRequest.
    image()));
87
88      Message message = new Message();
89      message.setChat(chat);
90      message.setAuthor(user);
91      message.setContent("");
92      message.setImage(imageFilename);
93
94      return messageRepository.save(message);
95  }
96
97
98  public List<Chat> getActiveChats(Long userId) {
99      User user = userRepository.findById(userId)
100             .orElseThrow(() -> new NotFoundException("User not found"));
101
102     return chatRepository.
    findAllByFromUserAndToUserConfirmedTrueOrToUserAndToUserConfirmedTrue(user,
    user)
103             .stream().filter(chat -> !chat.isCanceled()).toList();
104 }
105
106 @Transactional
107 public Message sendMessage(Long chatId, Long authorId, String content) {
```

```
108         Chat chat = chatRepository.findById(chatId)
109                 .orElseThrow(() -> new NotFoundException("Chat not found"));
110 
111         if (!chat.isToUserConfirmed()) {
112             throw new ForbiddenException("Chat is not confirmed");
113         }
114 
115         User author = userRepository.findById(authorId)
116                 .orElseThrow(() -> new NotFoundException("Author not found"));
117 
118         Message message = new Message();
119         message.setChat(chat);
120         message.setAuthor(author);
121         message.setContent(content);
122         return messageRepository.save(message);
123     }
124 
125     public List<Message> getMessages(Long chatId) {
126         Chat chat = chatRepository.findById(chatId)
127                 .orElseThrow(() -> new NotFoundException("Chat not found"));
128 
129         return messageRepository.findAllByChatOrderBySentAt(chat);
130     }
131 
132     @Transactional
133     public Chat createChatRequest(Long fromUserId, Long toUserId) {
134         if (fromUserId.equals(toUserId)) {
135             throw new ForbiddenException("You cannot invite yourself to a chat");
136         }
137 
138         User fromUser = userRepository.findById(fromUserId)
139                 .orElseThrow(() -> new NotFoundException("Sender not found"));
140 
141         User toUser = userRepository.findById(toUserId)
142                 .orElseThrow(() -> new NotFoundException("Recipient not found"));
143 
144         boolean chatExists = chatRepository.existsByFromUserAndToUser(fromUser, toUser)
     ||
145                         chatRepository.existsByFromUserAndToUser(toUser, fromUser);
146 
147         if (chatExists) {
```

```
148             throw new ForbiddenException("Chat request already exists");
149         }
150 
151         Chat chat = new Chat();
152         chat.setFromUser(fromUser);
153         chat.setToUser(toUser);
154         chat.setToUserConfirmed(false);
155 
156         return chatRepository.save(chat);
157     }
158 
159 
160 }
```

ПРИЛОЖЕНИЕ Г

Управление миграциями схемы базы данных с использованием Liquibase

Файл: master-changelog.yaml (Главный файл миграций Liquibase)

```yaml
databaseChangeLog:
  - include:
      file: db/changelog/changelog-category.yaml
  - include:
      file: db/changelog/changelog-question.yaml
  - include:
      file: db/changelog/changelog-question-in-queue.yaml
  - include:
      file: db/changelog/changelog-user.yaml
  - include:
      file: db/changelog/changelog-answer.yaml
  - include:
      file: db/changelog/changelog-chat.yaml
  - include:
      file: db/changelog/changelog-message.yaml
  - include:
      file: db/changelog/changelog-user-category.yaml
  - include:
      file: db/changelog/changelog-insert-admin-user.yaml
  - include:
      file: db/changelog/changelog-add-base-category.yaml
```

Файл: changelog-user.yaml (Пример миграции для таблицы пользователей)

```yaml
databaseChangeLog:
  - changeSet:
      id: user-init
      author: yourname
      changes:
        - createTable:
            tableName: users
            columns:
              - column:
                  name: id
                  type: BIGINT IDENTITY
                  constraints:
                    primaryKey: true
```

```
            nullable: false
            unique: true
        - column:
            name: username
            type: VARCHAR(255)
            constraints:
              nullable: false
              unique: true
        - column:
            name: password
            type: VARCHAR(255)
            constraints:
              nullable: false
        - column:
            name: gender
            type: VARCHAR(10)
            constraints:
              nullable: false
        - column:
            name: description
            type: VARCHAR(255)
        - column:
            name: avatar
            type: VARCHAR(255)
        - column:
            name: roles
            type: VARCHAR(255)
            constraints:
              nullable: false
              defaultValue: "USER"
```

ПРИЛОЖЕНИЕ Д

Инициализация приложения, настройка темы и маршрутизация

```
import 'package:flutter/material.dart';
import 'screens/login_screen.dart';
import 'screens/register_screen.dart';
import 'screens/main_screen.dart';
import 'screens/pages/messenger_page.dart';
import 'screens/pages/edit_profile_page.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Bim Bim App',
      theme: ThemeData(
        brightness: Brightness.dark,
        scaffoldBackgroundColor: Colors.black,
        appBarTheme: const AppBarTheme(
          backgroundColor: Colors.black,
          elevation: 0,
          centerTitle: true,
          titleTextStyle: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
          ),
          iconTheme: IconThemeData(
              color: Color.fromARGB(255, 3, 173, 162)),
        ),
        bottomNavigationBarTheme: const BottomNavigationBarThemeData(
          backgroundColor: Colors.black,
          selectedItemColor:
              Color.fromARGB(255, 3, 173, 162),
          unselectedItemColor: Colors.purple,
        ),
```

```
      textTheme: const TextTheme(
        bodyMedium: TextStyle(color: Colors.white),
      ),
      iconTheme: const IconThemeData(
          color: Color.fromARGB(255, 3, 173, 162)),
      elevatedButtonTheme: ElevatedButtonThemeData(
        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.black,
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(12),
          ),
          elevation: 10,
          side: const BorderSide(
              color: Colors.purple, width: 2),
        ),
      ),
    ),
    initialRoute: '/login',
    routes: {
      '/login': (context) => const LoginScreen(),
      '/register': (context) => const RegisterScreen(),
      '/home': (context) => const MainScreen(),
      '/messenger': (context) => const MessengerPage(),
      '/editProfile': (context) => const EditProfilePage(),
    },
  );
 }
}
```

ПРИЛОЖЕНИЕ Е

Клиент для взаимодействия с API

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:shared_preferences/shared_preferences.dart';

class ApiClient {
  Future<String?> _getToken() async {
    final prefs = await SharedPreferences.getInstance();
    return prefs.getString('jwt_token');
  }

  Future<Map<String, String>> _getHeaders({Map<String, String>? extraHeaders}) async {
    final token = await _getToken();
    final headers = <String, String>{
      'Content-Type': 'application/json',
      if (token != null) 'Authorization': 'Bearer $token',
      ...?extraHeaders,
    };
    return headers;
  }

  Future<http.Response> get(String endpoint, {Map<String, String>? headers}) async {
    final fullHeaders = await _getHeaders(extraHeaders: headers);
    final url = Uri.parse(endpoint);
    return http.get(url, headers: fullHeaders);
  }

  Future<http.Response> post(String endpoint, {Object? body, Map<String, String>? headers}) async {
    final fullHeaders = await _getHeaders(extraHeaders: headers);
    final url = Uri.parse(endpoint);
    return http.post(url, headers: fullHeaders, body: body);
  }

  Future<http.Response> put(String endpoint, {Object? body, Map<String, String>? headers}) async {
    final fullHeaders = await _getHeaders(extraHeaders: headers);
    final url = Uri.parse(endpoint);
    return http.put(url, headers: fullHeaders, body: jsonEncode(body));
```

```
  }

  Future<http.Response> delete(String endpoint, {Map<String, String>? headers}) async {
    final fullHeaders = await _getHeaders(extraHeaders: headers);
    final url = Uri.parse(endpoint);
    return http.delete(url, headers: fullHeaders);
  }

  Future<http.StreamedResponse> uploadMultipart({
    required String endpoint,
    required List<http.MultipartFile> files,
    Map<String, String>? fields,
  }) async {
    final token = await _getToken();
    final request = http.MultipartRequest('POST', Uri.parse(endpoint));

    if (token != null) {
      request.headers['Authorization'] = 'Bearer $token';
    }

    if (fields != null) {
      request.fields.addAll(fields);
    }

    request.files.addAll(files);

    return request.send();
  }
}
```

ПРИЛОЖЕНИЕ Ж

Экран аутентификации пользователя

```
import 'package:flutter/material.dart';
import 'package:bim_bim_app/services/api_client.dart';
import 'dart:convert';
import 'package:shared_preferences/shared_preferences.dart';
import '../constants/constants.dart';

class LoginScreen extends StatefulWidget {
  const LoginScreen({super.key});

  @override
  State<LoginScreen> createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final _apiClient = ApiClient();
  final _usernameController = TextEditingController();
  final _passwordController = TextEditingController();

  Future<void> _login() async {
    final String username = _usernameController.text;
    final String password = _passwordController.text;

    if (username.isEmpty || password.isEmpty) {
      _showError('Please fill in all fields!');
      return;
    }

    try {
      final response = await _apiClient.post(
        '$baseUrl/auth/login',
        body: jsonEncode({
          'username': username,
          'password': password,
        }),
      );

      if (response.statusCode == 200) {
        final Map<String, dynamic> responseData = jsonDecode(response.body);

```

```
      if (responseData.containsKey('token')) {
        final String token = responseData['token'];

        final prefs = await SharedPreferences.getInstance();
        await prefs.setString('jwt_token', token);

        Navigator.pushReplacementNamed(context, '/home');
      } else {
        _showError('Token not received!');
      }
    } else {
      _showError('Error: ${response.statusCode} - ${response.body}');
    }
  } catch (e) {
    _showError('Failed to connect to server');
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: const Color(0xFF121212),
    body: Center(
      child: SingleChildScrollView(
        padding: const EdgeInsets.all(20),
        child: Card(
          color: const Color(0xFF1E1E1E),
          elevation: 15,
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(16),
          ),
          child: Padding(
            padding: const EdgeInsets.symmetric(horizontal: 20, vertical: 30),
            child: Column(
              mainAxisSize: MainAxisSize.min,
              children: [
                const CircleAvatar(
                  radius: 50,
                  backgroundColor: Color(0xFFBB86FC),
                  child: Icon(
                    Icons.person,
```

```
            size: 50,
            color: Colors.black,
          ),
        ),
        const SizedBox(height: 20),
        const Text(
          'Welcome back!',
          style: TextStyle(
            fontSize: 24,
            fontWeight: FontWeight.bold,
            color: Colors.white,
            shadows: [
              Shadow(
                color: Color(0xFFBB86FC),
                blurRadius: 10,
              ),
            ],
          ),
        ),
        const SizedBox(height: 20),
        TextField(
          controller: _usernameController,
          style: const TextStyle(color: Colors.white),
          decoration: InputDecoration(
            labelText: 'Username',
            labelStyle: const TextStyle(color: Colors.white),
            prefixIcon: const Icon(Icons.person, color: Color(0xFFBB86FC)),
            filled: true,
            fillColor: const Color(0xFF2E2E2E),
            border: OutlineInputBorder(
              borderRadius: BorderRadius.circular(12),
            ),
            enabledBorder: OutlineInputBorder(
              borderSide: const BorderSide(color: Color(0xFFBB86FC)),
              borderRadius: BorderRadius.circular(12),
            ),
          ),
        ),
        const SizedBox(height: 20),
        TextField(
          controller: _passwordController,
```

```
122                 obscureText: true,
123                 style: const TextStyle(color: Colors.white),
124                 decoration: InputDecoration(
125                   labelText: 'Password',
126                   labelStyle: const TextStyle(color: Colors.white),
127                   prefixIcon: const Icon(Icons.lock, color: Color(0xFFBB86FC)),
128                   filled: true,
129                   fillColor: const Color(0xFF2E2E2E),
130                   border: OutlineInputBorder(
131                     borderRadius: BorderRadius.circular(12),
132                   ),
133                   enabledBorder: OutlineInputBorder(
134                     borderSide: const BorderSide(color: Color(0xFFBB86FC)),
135                     borderRadius: BorderRadius.circular(12),
136                   ),
137                 ),
138               ),
139               const SizedBox(height: 20),
140               ElevatedButton(
141                 onPressed: _login,
142                 style: ElevatedButton.styleFrom(
143                   padding: const EdgeInsets.symmetric(
144                     horizontal: 80,
145                     vertical: 15,
146                   ),
147                   backgroundColor: const Color(0xFFBB86FC),
148                   shape: RoundedRectangleBorder(
149                     borderRadius: BorderRadius.circular(12),
150                   ),
151                 ),
152                 child: const Text(
153                   'Login',
154                   style: TextStyle(
155                     fontSize: 16,
156                     fontWeight: FontWeight.bold,
157                     color: Colors.black,
158                   ),
159                 ),
160               ),
161               const SizedBox(height: 20),
162               Row(
```

```
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  const Text(
                    'Don\'t have an account?',
                    style: TextStyle(color: Colors.white),
                  ),
                  TextButton(
                    onPressed: () {
                      Navigator.pushNamed(context, '/register');
                    },
                    child: const Text(
                      'Register',
                      style: TextStyle(color: Color(0xFFBB86FC)),
```

ПРИЛОЖЕНИЕ И

Основная структура навигации приложения

```
import 'package:flutter/material.dart';
import 'package:bim_bim_app/constants/constants.dart';
import 'package:bim_bim_app/screens/pages/admin_page.dart';
import 'package:bim_bim_app/services/api_client.dart';

import '../../screens/pages/people_page.dart';
import '../../screens/pages/tests_page.dart';
import '../../screens/pages/messenger_page.dart';
import '../../screens/pages/profile_page.dart';

class MainScreen extends StatefulWidget {
  const MainScreen({super.key});

  @override
  State<MainScreen> createState() => _MainScreenState();
}

class _MainScreenState extends State<MainScreen> {
  final _apiClient = ApiClient();
  int _currentIndex = 0;
  bool _isAdmin = false;

  final List<Widget> _pages = [
    const QuestionsPage(),
    const PeoplePage(),
    const MessengerPage(),
    const AdminPage(),
  ];

  final List<String> _titles = ['Tests', 'Recommendations', 'Chats', 'Admin Panel'];

  @override
  void initState() {
    super.initState();
    _checkAdminStatus();
  }

  Future<void> _checkAdminStatus() async {
    try {
```

```
    final response = await _apiClient.get('$baseUrl/auth/isAdmin');

    if (response.statusCode == 200) {
      final data = response.body;
      setState(() {
        _isAdmin = data == "true";
      });
    } else {
      print('Failed to check admin status: ${response.body}');
    }
  } catch (e) {
    print('Error checking admin status: $e');
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(
        _titles[_currentIndex],
        style: const TextStyle(
          color: Colors.white,
          shadows: [
            Shadow(
              color: Colors.green,
              blurRadius: 5,
            ),
          ],
        ),
      ),
      backgroundColor: Colors.black,
      actions: [
        IconButton(
          icon: const Icon(Icons.person),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => const ProfilePage()),
            );
          },
```

```
        ),
      ],
    ),
    body: _pages[_currentIndex],
    bottomNavigationBar: BottomNavigationBar(
      currentIndex: _currentIndex,
      onTap: (index) {
        setState(() {
          _currentIndex = index;
        });
      },
      items: [
        const BottomNavigationBarItem(
          icon: Icon(Icons.library_books),
          label: 'Tests',
        ),
        const BottomNavigationBarItem(
          icon: Icon(Icons.people),
          label: 'People',
        ),
        const BottomNavigationBarItem(
          icon: Icon(Icons.message),
          label: 'Messenger',
        ),
        if (_isAdmin)
          const BottomNavigationBarItem(
            icon: Icon(Icons.admin_panel_settings),
            label: 'Admin',
          ),
      ],
    ),
  );
 }
}
```

ПРИЛОЖЕНИЕ К

Механизм ответов на вопросы с использованием Swipe-карт

```
import 'dart:convert';
import 'dart:typed_data';
import 'package:flutter/material.dart';
import 'package:bim_bim_app/services/api_client.dart';
import 'package:swipe_cards/swipe_cards.dart';
import '../../constants/constants.dart';

String decodeUtf8(String input) {
  return utf8.decode(Uint8List.fromList(input.codeUnits));
}

class QuestionItem {
  final String id;
  final String content;
  final String answerLeft;
  final String answerRight;

  QuestionItem({required this.id, required this.content, required this.answerLeft, required this
      .answerRight});
}

class QuestionsPage extends StatefulWidget {
  const QuestionsPage({super.key});

  @override
  State<QuestionsPage> createState() => _QuestionsPageState();
}

class _QuestionsPageState extends State<QuestionsPage> {
  final _apiClient = ApiClient();
  String _selectedCategory = '';
  List<String> _categories = [];
  List<Map<String, dynamic>> _questions = [];
  Map<String, String> _categoryMap = {};
  late List<SwipeItem> _swipeItems = [];
  late MatchEngine _matchEngine;

  @override
  void initState() {
```

```
    super.initState();
    _loadCategories();
  }

  Future<void> _loadCategories() async {
    try {
      final response = await _apiClient.get('$baseUrl/category/all');

      if (response.statusCode == 200) {
        final data = json.decode(response.body) as List;
        setState(() {
          _categories = data.map((category) {
            var name = category['name'];
            return name is String
                ? name
                : name.toString();
          }).toList();

          _categoryMap = {
            for (var item in data)
              item['name'] is String ? item['name'] : item['name'].toString():
                  (item['id'] is String
                      ? item['id']
                      : item['id'].toString())
          };

          if (_categories.isNotEmpty) {
            _selectedCategory = _categories[0];
            _loadAllQuestions(_selectedCategory);
          }
        });
      } else {
        throw Exception('Failed to load categories');
      }
    } catch (e) {
      print('Error loading categories: $e');
    }
  }

  Future<void> _loadAllQuestions(String categoryName) async {
    final categoryId = _categoryMap[categoryName];
```

```
80    final intId = int.tryParse(categoryId ?? '0');
81
82    try {
83      final response = await _apiClient.get('$baseUrl/question/remainderByCategory?
      categoryId=$intId');
84
85      print(response.body);
86      print(response.statusCode);
87
88      if (response.statusCode == 200) {
89        final decodedBody = decodeUtf8(response.body);
90        final data = json.decode(decodedBody) as List;
91
92        if (data.isNotEmpty) {
93          setState(() {
94            _questions = data
95                .map((q) => {'id': q['id'], 'content': q['content'], 'answerLeft' : q['answerLeft'], '
      answerRight' : q['answerRight']})
96                .toList();
97            _initializeSwipeItems();
98          });
99        } else {
100         print("No questions found for this category");
101       }
102     }
103   } catch (e) {
104     print('Error loading questions: $e');
105   }
106 }
107
108 void _initializeSwipeItems() {
109   if (_questions.isNotEmpty) {
110     _swipeItems = _questions.map((question) {
111       QuestionItem questionItem = QuestionItem(
112         id: question['id'].toString(),
113         content: question['content'],
114         answerLeft: question['answerLeft'],
115         answerRight: question['answerRight'],
116       );
117
118       return SwipeItem(
```

```
        content: questionItem,
        likeAction: () => _onAnswer(questionItem.id, 1),
        nopeAction: () => _onAnswer(questionItem.id, -1),
        superlikeAction: () => _onAnswer(questionItem.id, 0),
      );
    }).toList();

    setState(() {
      _matchEngine = MatchEngine(swipeItems: _swipeItems);
    });
  }
}

Future<void> _sendAnswer(String questionId, int value) async {
  try {
    final response = await _apiClient.get('$baseUrl/question/setAnswer?questionId=
   $questionId&result=$value');

    if (response.statusCode == 200) {
      print('Answer saved successfully for question $questionId');
    } else {
      print('Failed to save answer: ${response.body}');
    }
  } catch (e) {
    print('Error saving answer: $e');
  }
}

void _onAnswer(String questionId, int answer) async {
  if (_swipeItems.isNotEmpty) {
    await _sendAnswer(questionId, answer);

    setState(() {
      if (_swipeItems.isEmpty) {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('No more questions)')),
        );
      }
    });
  }
}
```

```
159
160 @override
161 Widget build(BuildContext context) {
162   final screenWidth = MediaQuery.of(context).size.width;
163   final screenHeight = MediaQuery.of(context).size.height;
164
165   return Scaffold(
166     backgroundColor: const Color(0xFF121212),
167     body: Padding(
168       padding: const EdgeInsets.all(16.0),
169       child: Column(
170         children: [
171           DropdownButtonFormField<String>(
172             value: _selectedCategory.isEmpty ? null : _selectedCategory,
173             decoration: InputDecoration(
174               labelText: 'Category',
175               labelStyle: const TextStyle(color: Colors.white),
176               border: OutlineInputBorder(
177                 borderRadius: BorderRadius.circular(12),
178               ),
179               enabledBorder: OutlineInputBorder(
180                 borderSide: const BorderSide(color: Color(0xFF64FFDA)),
181                 borderRadius: BorderRadius.circular(12),
182               ),
183             ),
184             dropdownColor: const Color(0xFF1E1E1E),
185             style: const TextStyle(color: Colors.white),
186             items: _categories
187                 .map((category) => DropdownMenuItem(
188                       value: category,
189                       child: Text(category),
190                     ))
191                 .toList(),
192             onChanged: (value) async {
193               if (value != null) {
194                 setState(() {
195                   _selectedCategory = value;
196                   _questions.clear();
197                   _loadAllQuestions(value);
198                 });
199               }
```

```
            },
          ),
          const SizedBox(height: 20),
          Expanded(
            child: _questions.isEmpty
                ? const Center(
                    child: Column(
                      mainAxisAlignment: MainAxisAlignment.center,
                      children: [
                        Icon(
                          Icons.done_all,
                          color: Colors.green,
                          size: 64,
                        ),
                        SizedBox(height: 20),
                        Text(
                          'No more questions!',
                          style: TextStyle(
                            color: Colors.white,
                            fontSize: 18,
                            fontWeight: FontWeight.bold,
                          ),
                        ),
                      ],
                    ),
                  )
                : SwipeCards(
                    matchEngine: _matchEngine,
                    itemBuilder: (context, index) {
                      final question = _swipeItems[index].content.content as String;

                      final leftOption =
                          _swipeItems[index].content.answerLeft as String;

                      final rightOption =
                          _swipeItems[index].content.answerRight as String;

                      return Container(
                        width: screenWidth * 0.9,
                        height: screenHeight * 0.7,
                        decoration: BoxDecoration(
```

```
241                     color: Colors.black,
242                     borderRadius: BorderRadius.circular(20),
243                     boxShadow: [
244                       BoxShadow(
245                         color: Colors.green.withOpacity(0.5),
246                         blurRadius: 20,
247                         offset: const Offset(-5, 5),
248                       ),
249                       BoxShadow(
250                         color: Colors.purple.withOpacity(0.5),
251                         blurRadius: 20,
252                         offset: const Offset(5, 5),
253                       ),
254                     ],
255                   ),
256                   child: Stack(
257                     children: [
258                       Positioned(
259                         top: screenHeight * 0.05,
260                         left: 20,
261                         right: 20,
262                         child: Text(
263                           question,
264                           textAlign: TextAlign.center,
265                           style: TextStyle(
266                             fontSize: question.length > 20 ? 28 : 36,
267                             fontWeight: FontWeight.bold,
268                             color: Colors.white,
269                             shadows: const [
270                               Shadow(
271                                 color: Colors.white,
272                                 blurRadius: 10,
273                               ),
274                             ],
275                           ),
276                         ),
277                       ),
278                       Positioned(
279                         left: 20,
280                         top: screenHeight * 0.35,
281                         child: Row(
```

```
                children: [
                  const Icon(
                    Icons.arrow_back,
                    color: Color(0xFF64FFDA),
                    size: 28,
                  ),
                  const SizedBox(width: 8),
                  Text(
                    leftOption,
                    style: const TextStyle(
                      fontSize: 28,
                      fontWeight: FontWeight.bold,
                      color: Color(0xFF64FFDA),
                      shadows: [
                        Shadow(
                          color: Color(0xFF64FFDA),
                          blurRadius: 10,
                        ),
                      ],
                    ),
                  ),
                ],
              ),
            ),
            Positioned(
              right: 20,
              top: screenHeight * 0.35,
              child: Row(
                children: [
                  Text(
                    rightOption,
                    style: const TextStyle(
                      fontSize: 28,
                      fontWeight: FontWeight.bold,
                      color: Color(0xFFBB86FC),
                      shadows: [
                        Shadow(
                          color: Color(0xFFBB86FC),
                          blurRadius: 10,
                        ),
                      ],
```

```
                                ),
                              ),
                              const SizedBox(width: 8),
                              const Icon(
                                Icons.arrow_forward,
                                color: Color(0xFFBB86FC),
                                size: 28,
                              ),
                            ],
                          ),
                        ),
                      ],
                    ),
                  );
                },
                onStackFinished: () {
                  ScaffoldMessenger.of(context).showSnackBar(
                    const SnackBar(content: Text('No more questions)')),
                  );
                  setState(() {
                    _questions.clear();
                  });
                },
                upSwipeAllowed: false,
                fillSpace: true,
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

ПРИЛОЖЕНИЕ Л

Интерфейс обмена сообщениями в чате

```
import 'dart:async';
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:bim_bim_app/constants/constants.dart';
import 'package:bim_bim_app/services/api_client.dart';
import 'package:http/http.dart' as http;
import 'package:image_picker/image_picker.dart';

class ChatPage extends StatefulWidget {
  final String chatName;
  final int chatId;
  final String avatar;

  const ChatPage(
      {super.key,
      required this.chatName,
      required this.chatId,
      required this.avatar});

  @override
  State<ChatPage> createState() => _ChatPageState();
}

class _ChatPageState extends State<ChatPage> {
  List<Map<String, dynamic>> _messages = [];
  final TextEditingController _controller = TextEditingController();
  final ApiClient _apiClient = ApiClient();
  bool isLoading = true;
  Timer? _updateTimer;

  @override
  void initState() {
    super.initState();
    _loadMessages();

    _updateTimer = Timer.periodic(const Duration(seconds: 3), (timer) {
      _loadMessages();
    });
  }
```

```
40
41  @override
42  void dispose() {
43    _updateTimer?.cancel();
44    _controller.dispose();
45    super.dispose();
46  }
47
48  Future<void> _uploadImage(XFile imageFile) async {
49    try {
50      final file = await http.MultipartFile.fromPath(
51        'image',
52        imageFile.path,
53      );
54
55      final response = await _apiClient.uploadMultipart(
56        endpoint: '$baseUrl/chat/${widget.chatId}/uploadImage',
57        files: [file],
58        fields: {'type': 'image'}
59      );
60
61      print(response);
62      print(response.statusCode);
63      print(response.headers);
64
65      if (response.statusCode != 200) {
66        throw Exception('Failed to upload image');
67      }
68    } catch (e) {
69      ScaffoldMessenger.of(context).showSnackBar(
70        SnackBar(content: Text('Error: ${e.toString()}')),
71      );
72    }
73  }
74
75  Future<void> _pickImage() async {
76    try {
77      final picker = ImagePicker();
78      final pickedFile = await picker.pickImage(source: ImageSource.gallery);
79
80      if (pickedFile != null) {
```

```
81        await _uploadImage(pickedFile);
82      }
83    } catch (e) {
84      ScaffoldMessenger.of(context).showSnackBar(
85        SnackBar(content: Text('Error: ${e.toString()}')),
86      );
87    }
88  }
89
90  Future<void> _loadMessages() async {
91    try {
92      final response = await _apiClient.get('$baseUrl/chat/${widget.chatId}/messages');
93
94      print(response.body);
95
96      if (response.statusCode != 200) {
97        throw Exception('Failed to load messages');
98      }
99
100     final decodedBody = utf8.decode(response.bodyBytes);
101     final List<dynamic> data = json.decode(decodedBody);
102
103     setState(() {
104       _messages = data.map((message) {
105         return {
106           'isMe': message['isMe'],
107           'content': message['content'],
108           'image': message['image'],
109         };
110       }).toList();
111       isLoading = false;
112     });
113   } catch (e) {
114     setState(() {
115       isLoading = false;
116     });
117     ScaffoldMessenger.of(context).showSnackBar(
118       SnackBar(content: Text('Error: ${e.toString()}')),
119     );
120   }
121 }
```

```
122
123 Future<void> _sendMessage() async {
124   if (_controller.text.trim().isEmpty) return;
125
126   try {
127     final response = await _apiClient.post(
128       '$baseUrl/chat/${widget.chatId}/messages',
129       body: _controller.text.trim()
130     );
131
132     if (response.statusCode != 200) {
133       throw Exception('Failed to send message');
134     }
135
136     setState(() {
137       _messages.add({
138         'isMe': true,
139         'content': _controller.text.trim(),
140       });
141       _controller.clear();
142     });
143   } catch (e) {
144     ScaffoldMessenger.of(context).showSnackBar(
145       SnackBar(content: Text('Error: ${e.toString()}')),
146     );
147   }
148 }
149
150 @override
151 Widget build(BuildContext context) {
152   return Scaffold(
153     backgroundColor: const Color(0xFF121212),
154     appBar: AppBar(
155       title: Text(widget.chatName),
156     ),
157     body: Column(
158       children: [
159         Expanded(
160           child: isLoading
161               ? const Center(
162                   child: CircularProgressIndicator(
```

```
            valueColor:
                AlwaysStoppedAnimation<Color>(Color(0xFF64FFDA)),
          ),
        )
      : ListView.builder(
          padding: const EdgeInsets.all(16),
          itemCount: _messages.length,
          itemBuilder: (context, index) {
            final message = _messages[index];
            final isMe = message['isMe'];
            return Align(
              alignment:
                  isMe ? Alignment.centerRight : Alignment.centerLeft,
              child: Container(
                decoration: BoxDecoration(
                  color: isMe
                      ? const Color(0xFF1F2937)
                      : const Color(0xFF2D2D34),
                  borderRadius: BorderRadius.circular(16),
                  boxShadow: [
                    BoxShadow(
                      color: isMe
                          ? const Color(0xFF64FFDA).withOpacity(0.1)
                          : const Color(0xFFBB86FC).withOpacity(0.1),
                      blurRadius: 10,
                      offset: const Offset(2, 4),
                    ),
                  ],
                ),
                margin: const EdgeInsets.symmetric(vertical: 8),
                padding: const EdgeInsets.all(12),
                child: message['image'] != null
                    ? Image.network(
                        message['image'],
                        fit: BoxFit.cover,
                      )
                    : Text(
                        message['content'],
                        style: const TextStyle(
                          color: Colors.white70,
                          fontSize: 16,
```

```
                        ),
                      ),
                    ),
                  );
                },
              ),
            ),
            Container(
              padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 12),
              decoration: const BoxDecoration(
                color: Color(0xFF1E1E1E),
                border: Border(
                  top: BorderSide(color: Color(0xFF64FFDA), width: 1.5),
                ),
              ),
              child: Row(
                children: [
                  IconButton(
                    icon: const Icon(Icons.photo, color: Colors.white),
                    onPressed: _pickImage,
                  ),
                  const SizedBox(width: 12),
                  Expanded(
                    child: TextField(
                      controller: _controller,
                      style: const TextStyle(color: Colors.white),
                      decoration: InputDecoration(
                        hintText: 'Enter a message...',
                        hintStyle: const TextStyle(color: Colors.white38),
                        filled: true,
                        fillColor: const Color(0xFF2E2E2E),
                        border: OutlineInputBorder(
                          borderRadius: BorderRadius.circular(12),
                          borderSide: BorderSide.none,
                        ),
                        contentPadding: const EdgeInsets.symmetric(
                          horizontal: 16,
                          vertical: 12,
                        ),
                      ),
                    ),
                  ),
```

```
245                 ),
246                 const SizedBox(width: 12),
247                 ElevatedButton(
248                   onPressed: _sendMessage,
249                   style: ElevatedButton.styleFrom(
250                     backgroundColor: const Color(0xFF64FFDA),
251                     padding: const EdgeInsets.all(16),
252                     shape: RoundedRectangleBorder(
253                       borderRadius: BorderRadius.circular(12),
254                     ),
255                   ),
256                   child: const Icon(Icons.send, color: Colors.black),
257                 ),
258               ],
```

ПРИЛОЖЕНИЕ М

Реализация рекомендательной системы

Файл: models.py (Описание сущностей)

```
from typing import List, Dict, Optional

from pydantic import BaseModel, Field


class QuestionMatchingRequest(BaseModel):
    id: int
    content: str
    answerLeft: Optional[str] = None
    answerRight: Optional[str] = None


class UserMatchingRequest(BaseModel):
    id: int
    avatar: str
    gender: str
    username: str
    description: str
    answers: Dict[int, int]


class MatchingRequest(BaseModel):
    userId: int
    questions: List[QuestionMatchingRequest]
    users: List[UserMatchingRequest]


class MatchingResponse(BaseModel):
    id: int
    avatar: Optional[str] = None
    gender: str
    username: str
    description: str
    similarity: float
```

Файл: recommender.py (Класс для выдачи рекомендаций)

```
from typing import List
```

```
2
3  import numpy as np
4  from sentence_transformers import SentenceTransformer
5  from sklearn.decomposition import PCA
6  from sklearn.metrics.pairwise import cosine_similarity
7
8  from config import TEXT_EMBEDDING_MODEL
9  from models import MatchingRequest, MatchingResponse, QuestionMatchingRequest
10
11
12 def _get_ordered_question_ids(questions: List[QuestionMatchingRequest]) -> List[int]:
13     return [q.id for q in questions]
14
15
16 def _get_ternary_vector(answers: dict, question_ids: List[int]) -> np.ndarray:
17     return np.array([answers.get(qid, 0) for qid in question_ids], dtype=float)
18
19
20 def _compute_similarity(vec1: np.ndarray, vec2: np.ndarray) -> float:
21     if np.linalg.norm(vec1) == 0 or np.linalg.norm(vec2) == 0:
22         return 0.0
23     return float(cosine_similarity(vec1.reshape(1, -1), vec2.reshape(1, -1))[0][0])
24
25
26 class DynamicRecommendationSystem:
27     def __init__(self, alpha: float = 0.8):
28         self.text_embedder = SentenceTransformer(TEXT_EMBEDDING_MODEL)
29         self.text_embedding_dim = self.text_embedder.get_sentence_embedding_dimension
       ()
30         self.alpha = alpha
31
32     def _get_text_embedding(self, description: str) -> np.ndarray:
33         if not description:
34             return np.zeros(self.text_embedding_dim)
35         return self.text_embedder.encode(description)
36
37     def get_recommendations(self, request: MatchingRequest) -> List[MatchingResponse]:
38         if not request.questions or not request.users:
39             return []
40
41         ordered_qids = _get_ordered_question_ids(request.questions)
```

```
        main_user = next((u for u in request.users if u.id == request.userId), None)
        if not main_user:
            return []

        ternary_matrix = np.array([
            _get_ternary_vector(u.answers, ordered_qids) for u in request.users
        ])

        if ternary_matrix.shape[1] > 3:
            n_components = ternary_matrix.shape[1] // 2
            pca = PCA(n_components=n_components)
            ternary_matrix_reduced = pca.fit_transform(ternary_matrix)
        else:
            ternary_matrix_reduced = ternary_matrix

        id_to_vector = {
            user.id: vec for user, vec in zip(request.users, ternary_matrix_reduced)
        }

        main_ternary = id_to_vector[main_user.id]
        main_text = self._get_text_embedding(main_user.description)

        recommendations = []

        for candidate in request.users:
            if candidate.id == main_user.id:
                continue

            candidate_ternary = id_to_vector[candidate.id]
            candidate_text = self._get_text_embedding(candidate.description)

            ternary_sim = _compute_similarity(main_ternary, candidate_ternary)
            text_sim = _compute_similarity(main_text, candidate_text)

            combined_sim = self.alpha * ternary_sim + (1 - self.alpha) * text_sim

            recommendations.append(MatchingResponse(
                id=candidate.id,
                username=candidate.username,
                avatar=candidate.avatar,
                gender=candidate.gender,
```

```
                description=candidate.description,
                similarity=combined_sim * 100
            ))

        recommendations.sort(key=lambda x: x.similarity, reverse=True)
        return recommendations
```

Файл: main.py (REST-API сервиса рекомендаций)

```
from typing import List

import uvicorn
from fastapi import FastAPI, HTTPException

from models import MatchingRequest, MatchingResponse
from recommender import DynamicRecommendationSystem

app = FastAPI(title="Dynamic Dating Recommendation Service")

dynamic_recommendation_system = DynamicRecommendationSystem()

@app.on_event("startup")
async def startup_event():
    print("FastAPI application started (Dynamic Recommender).")


@app.post("/api/matching", response_model=List[MatchingResponse])
async def get_matching_users(request: MatchingRequest):
    print(request)
    try:
        if not request.users:
            return []
        if not request.questions and any(u.answers for u in request.users):
            raise HTTPException(status_code=400, detail="Answers provided but no
     questions defined to interpret them.")

        return dynamic_recommendation_system.get_recommendations(request)
    except HTTPException as http_exc:
        raise http_exc
    except Exception as e:
        print(f"Error during matching: {e}")
        import traceback
        traceback.print_exc()
```

```
        raise HTTPException(status_code=500, detail=f"Internal server error: {str(e)}")


if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000, reload=True)
```