Богуславский Семен Сергеевич
студент
Войнов Даниил Игоревич
студент
ФГБОУ ВО «Кубанский государственный университет»
г. Краснодар, Краснодарский край

ПЕРСОНАЛИЗАЦИЯ ПОЛЬЗОВАТЕЛЬСКОГО КОНТЕНТА НА ОСНОВЕ ПРОФИЛЯ ПРЕДПОЧТЕНИЙ: РЕАЛИЗАЦИЯ В ВЕБ И МОБИЛЬНЫХ ПРИЛОЖЕНИЯХ

Аннотация: *в статье рассматриваются архитектурные и алгоритмические подходы к персонализации пользовательского контента на основе явно заданного профиля предпочтений. Проведён сравнительный анализ реализации механизма персонализации в двух учебных проектах: веб-приложении для совместного подбора контента (стек FastAPI, React, PostgreSQL, Redis) и мобильном приложении для мониторинга показателей здоровья (Flutter, Firebase). Описаны структуры данных пользовательского профиля, алгоритмы фильтрации и ранжирования контента, а также механизм онбординга как инструмент первичного сбора предпочтений. Показано, что хранение профиля на стороне сервера с кэшированием в Redis обеспечивает персонализацию, согласованную между клиентскими платформами, тогда как локальное хранение профиля во Flutter позволяет работать в офлайн-режиме без потери персонального контекста.*

Ключевые слова: *персонализация, пользовательский профиль, онбординг, FastAPI, Flutter, Firebase, Redis, PostgreSQL, фильтрация контента, цифровые образовательные сервисы, рекомендательные системы, профиль предпочтений.*

Введение.

Персонализация – один из ключевых факторов вовлечённости пользователя в цифровой сервис. В образовательном контексте это особенно значимо: адаптация интерфейса и контента под индивидуальный профиль обучающегося сни-

жает когнитивную нагрузку и повышает мотивацию [1]. Промышленные рекомендательные системы (Netflix, Spotify, YouTube) решают задачу персонализации методами коллаборативной и контентной фильтрации, требующими значительных объёмов данных и вычислительных ресурсов. Для учебных приложений с небольшой аудиторией более практичен подход на основе явно заданного профиля предпочтений: пользователь сам указывает свои интересы при регистрации, а система немедленно адаптирует контент без необходимости накапливать поведенческую статистику.

В рамках разработки двух учебных проектов – веб-приложения для совместного подбора медиаконтента и мобильного приложения для мониторинга показателей здоровья – были реализованы различные стратегии персонализации, обусловленные спецификой платформ и предметных областей. Анализ и сравнение этих стратегий составляют основное содержание настоящей статьи.

Структура профиля пользовательских предпочтений.

Профиль предпочтений – структурированная совокупность параметров, описывающих интересы и характеристики конкретного пользователя. В зависимости от предметной области состав профиля существенно различается.

В веб-приложении для подбора контента профиль хранится в таблице user_preferences базы данных PostgreSQL и включает следующие поля: массив предпочтительных жанров (genres: integer[]), диапазон допустимых годов выпуска (year_from, year_to: integer), минимальный рейтинг (min_rating: numeric), предпочтительные языки (languages: varchar[]) и флаг включения контента для взрослых (adult_content: boolean). Использование типа integer[] для жанров позволяет выполнять фильтрацию на уровне SQL с применением оператора overlaps, не загружая логику в код приложения [2].

В мобильном приложении для мониторинга здоровья профиль включает целевые значения биометрических показателей: целевую частоту сердечных сокращений (target_heart_rate: int), целевой вес (target_weight: double), норму сна (sleep_goal_hours: double), дневную норму шагов (steps_goal: int) и целевой уро-

вень потребления воды (water_goal_ml: int). Профиль хранится локально в защищённом хранилище устройства через flutter_secure_storage, что обеспечивает доступ к персональным настройкам без подключения к сети, и дублируется в Firebase Firestore для синхронизации между устройствами пользователя.

Принципиальное различие двух подходов состоит в семантике профиля: в первом случае профиль определяет фильтр для серверной выборки контента, во втором – эталонные значения для сравнения с фактическими измерениями и визуализации отклонений.

Онбординг как механизм первичного сбора предпочтений.

Онбординг – последовательность экранов, предъявляемых новому пользователю при первом входе в приложение. Его основная техническая функция – заполнение профиля предпочтений до того, как пользователь начнёт взаимодействовать с основным контентом. Без онбординга система вынуждена либо показывать всем одинаковую «холодную» выдачу, либо применять эвристики по умолчанию [3].

В веб-приложении онбординг реализован как многошаговый React-компонент с локальным состоянием Zustand. Каждый шаг соответствует одной группе параметров профиля: выбор жанров (чекбоксы с постерами), настройка временного диапазона (слайдер), выбор минимального рейтинга. При завершении онбординга накопленные данные отправляются единственным POST-запросом на эндпоинт /api/v1/users/preferences, изображенный на рисунке 1. На стороне FastAPI данные валидируются моделью Pydantic и сохраняются в PostgreSQL. Флаг onboarding_completed в таблице users переключается в true, после чего при последующих входах онбординг пропускается.

```
POST /api/v1/users/preferences
Content-Type: application/json
Authorization: Bearer <access_token>
{
  "genres": [28, 12, 878],
  "year_from": 2000,
  "year_to": 2024,
  "min_rating": 6.5,
  "languages": ["ru", "en"]
}
```

Рис. 1. Схема запроса к эндпоинту сохранения профиля

В мобильном приложении онбординг реализован как StatefulWidget с PageView, позволяющим перелистывать шаги свайпом. Пользователь последовательно задаёт целевые показатели через числовые поля ввода и слайдеры. Данные сохраняются локально немедленно при изменении (без кнопки «Сохранить»), что исключает потерю введённых значений при случайном закрытии приложения.

Алгоритм персонализированной фильтрации контента.

После заполнения профиля система должна использовать его при формировании каждого ответа с контентом. Рассмотрим реализацию этого механизма в веб-приложении.

При запросе ленты фильмов бэкенд выполняет следующую последовательность операций. Во-первых, извлекается профиль текущего пользователя. Для снижения нагрузки на базу данных профиль кэшируется в Redis с ключом user_preferences:{user_id} и временем жизни 300 секунд. При обновлении профиля соответствующий ключ инвалидируется. Во-вторых, профиль преобразуется в параметры запроса к внешнему API кинопоиска: жанры передаются через параметр genres, диапазон лет – через yearFrom/yearTo, минимальный рейтинг – через ratingFrom. В-третьих, полученный список фильмов дополнительно фильтруется на стороне сервера для исключения уже просмотренных пользователем позиций (выборка из таблицы swipes) [2].

Описанный алгоритм реализует контентную фильтрацию на основе явных атрибутов: совпадение жанров, временного периода и рейтинга. Это детерминированный подход – при одинаковом профиле результат всегда воспроизводим, что упрощает тестирование и отладку по сравнению с вероятностными рекомендательными моделями.

В мобильном приложении персонализация реализована иначе: вместо фильтрации входящего потока данных система сравнивает фактические измерения пользователя с целевыми значениями профиля и визуализирует отклонение. Например, если фактический средний пульс за неделю составляет 82 уд/мин при целевом значении 70 уд/мин, виджет пульса отображает предупреждение и цветовой индикатор. Логика сравнения инкапсулирована в классе HealthGoalEvaluator, принимающем объект профиля и список измерений за период.

Хранение и синхронизация профиля: *серверный и локальный подходы.*

Выбор места хранения профиля предпочтений определяет поведение приложения при работе с несколькими устройствами и в офлайн-режиме. Сравнение двух реализованных подходов представлено в таблице 1.

Таблица 1

Сравнение стратегий хранения профиля предпочтений

Характеристика	Серверное хранение (PostgreSQL + Redis)	Локальное хранение (flutter_secure_storage + Firestore)
Согласованность между устройствами	Автоматическая	Через синхронизацию Firestore
Офлайн-доступ к профилю	Недоступен без сети	Полный доступ
Время доступа к профилю	~1 мс (Redis) / ~5 мс (PostgreSQL)	<1 мс (локальное)
Защита от потери данных	Высокая (СУБД)	Средняя (зависит от резервного копирования устройства)
Нагрузка на сервер при запросе	Минимальная при кэшировании	Отсутствует

Серверное хранение с кэшированием в Redis обеспечивает единый источник истины для всех клиентов пользователя: изменение профиля на веб-сайте немедленно отражается при следующем запросе с мобильного устройства. Локальное

хранение во Flutter оправдано для приложений с высокой частотой чтения профиля (при каждом отображении виджета показателей) и требованием офлайн-работы. Гибридный подход – локальный кэш с периодической синхронизацией с сервером – сочетает преимущества обеих стратегий и применяется в производственных системах [4].

Обновление профиля и инвалидация кэша.

Изменение пользователем своих предпочтений должно немедленно отражаться на выдаче контента. В веб-приложении это реализовано через PATCH-эндпоинт /api/v1/users/preferences, принимающий частичное обновление профиля. После записи в PostgreSQL обработчик выполняет принудительную инвалидацию Redis-ключа:

await redis.delete(f»user_preferences:{user_id}")

При следующем запросе контента профиль будет заново прочитан из базы данных и записан в кэш с обновлёнными значениями. Такой подход гарантирует отсутствие «грязного чтения»: пользователь никогда не получит контент, отфильтрованный по устаревшему профилю [5].

В мобильном приложении обновление профиля обрабатывается реактивно: класс HealthProfileNotifier (Riverpod) хранит текущее состояние профиля и уведомляет все зависимые виджеты при его изменении. Запись в flutter_secure_storage и отправка обновления в Firestore выполняются асинхронно в фоновом потоке, не блокируя интерфейс пользователя. Конфликты при одновременном редактировании с двух устройств разрешаются по стратегии last-write-wins на стороне Firestore.

Заключение.

В статье проведён сравнительный анализ реализации персонализации на основе профиля предпочтений в двух учебных приложениях с различными архитектурными подходами. Показано, что серверное хранение профиля с кэшированием в Redis обеспечивает автоматическую согласованность между клиентами и минимальную задержку при чтении, тогда как локальное хранение во Flutter с

синхронизацией через Firebase Firestore обеспечивает полноценный офлайн-доступ. Механизм онбординга позволяет заполнить профиль до первого взаимодействия с контентом, устраняя проблему холодного старта без необходимости накапливать поведенческую статистику. Алгоритм контентной фильтрации на основе явных атрибутов профиля детерминирован и легко тестируем, что делает его предпочтительным выбором для учебных проектов по сравнению с вероятностными рекомендательными моделями. Реализованные подходы применимы к широкому классу образовательных платформ, в которых адаптация контента под профиль обучающегося является ключевым требованием.

References

1. Ameisen E. Building Machine Learning Powered Applications: Going from Idea to Product. Sebastopol: O'Reilly Media, 2020. 257 p.

2. FastAPI Documentation. SQL (Relational) Databases. URL: https://fastapi.tiangolo.com/tutorial/sql-databases/ (дата обращения: 18.03.2026).

3. Hurff S. Designing Products People Love. Sebastopol: O'Reilly Media, 2016. – 196 p.

4. Redis Documentation. Caching. URL: https://redis.io/docs/manual/patterns (дата обращения: 18.03.2026).

5. Firebase Documentation. Cloud Firestore Data Model. URL: https://firebase.google.com/docs/firestore/data-model (дата обращения: 18.03.2026).