Київський національний університет імені Тараса Шевченка
Факультет радіофізики, електроніки та комп’ютерних систем
Кафедра комп’ютерної інженерії

«Створення застосунку для відслідковування персонального прогресу прочитаної літератури»

Кваліфікаційна робота бакалавра студента
спеціальність 123 «Комп'ютерна інженерія»
Денис БЕЛІК
_____(підпис)

Науковий керівник, кандидат
фізико-математичних наук, асистент
Галина СТРІЛЬЧУК
_____(підпис)

Рецензент, кандидат фізико-математичних наук,
доцент Ірина ГАВРИЛЬЧЕНКО
_____(підпис)

До захисту допускаю:

Завідувач кафедрою
Юрій БОЙКО

Ухвалено на засіданні кафедри “____”_____2022 р., протокол № ____

Київ 2022

Реферат

Випускна кваліфікаційна робота бакалавра містить 81 сторінку, 42 ілюстрації, 26 джерел посилань, 2 таблиці, 3 додатки, 36 лістингів коду.

ПРОГРЕС ПРОХОДЖЕННЯ ТЕХНІЧНОЇ ЛІТЕРАТРИ, ВІДСЛІДОВУВАННЯ ПРОГРЕСУ, ІНТЕРФЕЙС КОРИСТУВАЧА, МОБІЛЬНИЙ ДОДАТОК, КРОС-ПЛАТФОРМЕНІСТЬ, ФОРМАТ ДЛЯ СТУДЕНТІВ, БІЗНЕС ЛОГІКА, ВАЛІДАЦІЯ, ДЕКЛАРАТИВНИЙ UI, BLoC ПАТЕРН, АВТОМАТИЗОВАНЕ ПРОФІЛЮВАННЯ, ОПТИМІЗАЦІЯ ПОКАЗНИКА FPS

Розроблено модель даних для відслідковування прогресу проходження технічної літератури. Визначено ключові архітектурні засоби для реалізації функціоналу дій користувача. Розроблено інтерфейс користувача відповідно до моделі даних, зокрема для інтеграції додатку з навчальним закладом. Виявлено та усунуто проблему низького показника FPS, за допомогою автоматизованого профілювання наведено результати оптимізації.

Зміст

Вступ

Одним із способів здобувати певні знання є прочитання літератури. При вивченні нової галузі знань потрібен структурований підхід до вивчення матеріалу, особливо якщо необхідно одночасно займатись вивченням різних сфер певної діяльності. Наприклад, розробник програмного забезпечення бажає покращити свої знання алгоритмів, а також поглибити розуміння популярної системи контролю версій Gi та читати про патерни проектування. На першому кроці потрібно записати список необхідних тем для вивчення та назви одиниць літератури, які розробник хоче прочитати. Далі, для ефективного прогресу, варто зазначити проміжок часу, за який потрібно пройти матеріал. Проміжок часу, зазвичай, досить великий - тобто декілька тижнів або місяців. Таким чином, маючи декілька цілей, які потрібно виконати за певний довгий період, отримуємо щось схоже на Objective Key Results (OKR)[1], але індивідуально. Важливо зауважити, що при читанні технічної літератури часто треба прочитати не всю книгу, а певні її частини. Обравши потрібні розділи, можна порахувати загальну кількість сторінок у всьому списку та на основі цього порахувати скільки потрібно проходити за день, за тиждень, або за інший невеликий відносно всього проміжку час, який розробник вважає найзручнішим для нього. Впродовж зазначеного періоду розробник слідкуватиме за тим, де він зупинився в тій чи іншій книзі, щоб наступного разу продовжити звідти, що є логічно.

Щоб структурувати списки цілей, спростити процес відслідковування прогресу та візуалізувати його - можна створити додаток. Застосунок з таким функціоналом звучить як такий, що має бути ергономічним, у швидкому доступі, та мінімально залежним від таких факторів, як доступ до мережі інтернет або підключення до джерела живлення. Тому було обрано зробити такий застосунок для мобільних девайсів - смартфонів.

Такий додаток може бути використаний і як допоміжний в навчальному процесі, адже формат даних також є схожим на формат семестру навчання в навчальному закладі - за проміжок часу в декілька місяців студент вивчає декілька предметів.

1 Огляд літератури

1.1 Дослідження існуючих рішень

Очевидно, що вже існує багато застосунків для відслідковування прогресу читання літератури[2,3,4]. Багато з них мають комплексний та сучасний дизайн інтерфейсу користувача, вони мають функціонал в деяких аспектах багатший за функціонал додатку, що розглядається в роботі. Тому мета цього розділу не є сказати про те, що існуючі додатки є поганими або недосконалими, а лише підкреслити те, що вони не націлені на формат, описаний у попередньому розділі (читання технічної літератури поєднане з Objective Key Results), та що використання даного додатку матиме деякі переваги для такого формату.

Було проаналізовано 5 існуючих застосунків, які є доступними на Play Market по пошуку “book tracker”, “reading tracker”, “progress tracker”, “okr”.

Результати “book tracker”, “reading tracker”:

1) MyBook (10M+ downloads)

Відсутня опція додавання книги мануально, а пошук в додатку не дав результатів для деяких прикладів технічної літератури. Безкоштовно можно прочитати перший розділ будь-якої книги, але для продовження потрібно оформити підписку. Прогрес оновлюється тільки при проходженні літератури

всередині застосунку, тобто при читанні реальної книги або на іншому девайсі прогрес неможливо буде оновити.

2) ReadMore (100K+ downloads)

Є можливість додавати літературу мануально, а також відмічати прогрес. Але не можна обрати окремі розділи в книзі. Також, в додатку можна встановлювати проміжок часу, але тільки на рівні однієї книги, тобто декілька не можна об’єднати під один проміжок часу.

3) Bookly

Відрізняєтья від двох попередніх застосунків тим, що ціллю є декілька книг за проміжок часу, але користувач може обрати тільки кількість книг, які хоче прочитати. Тобто точно вказати на книги (цілі), які потрібно пройти, немає можливості. При додаванні першої книги автоматично встановлюються два періоди - один місяць та один рік, що є не так гнучко, як можливість для користувача обирати проміжок часу самому.

Результати “progress tracker”, “okr”:

Потрібно зазначити, що найбільш популярні додатки по таким запитам мають формат To do list та тісно пов’язані з плануванням окремих невеликих завдань по конкретних датах в календарі (такі як “Any.do”, “Goal Tracker”, “Goal meter”); або націлені на предметну область спорту, тому додатки, описані нижче, мають відносно малу кількість завантажувань, але більше схожі на додаток цієї роботи.

4) Simple Progress Tracker (5K+ downloads)

Додаток є інтуітивно зрозумілим, для додавання цілі потрібно назвати її та вказати кількість балів (сторінок в контексті книжок), але немає можливості вказати проміжок часу, за який потрібно виконати цілі. Також, може бути тільки один набір цілей, а не декілька. Це позбавляє можливості створювати нові набори цілей зі збереженням старих.

5) Plai, Profit.co, Ally OKRs

Ці 3 додатки та багато інших націлені на OKR на рівні команди або компанії. Щоб користуватися ними, потрібно спочатку зареєструвати обліковий запис компанії, тому для персонального використання вони не є доцільними.

Отже, підсумовуючи, можна виділити декілька груп існуючих додатків, функціонал яких перетинається з функціоналом застосунку цієї роботи, але повністю не задовольняє всіх вимог.

1) додатки, які надають можливість безпосередньо читати (або слухати) всередині додатку. Інтеграція рідера обмежує можливість вибору девайсу, на якому користувачеві зручно читати. Також, якщо книгу можна обрати тільки з каталогу, який надає застосунок, є велика ймовірність, що потрібної користувачеві літератури там може не бути.

2) додатки для слідкування за прогресом читання. Такі додатки не мають за мету об’єднати групу цілей та встановити проміжок часу для їх виконання, а більше сконцентровані на слідкуванні за динамікою активності користувача.

3) додатки планування задач формату “to do list”. Мета таких застосунків допомогти з організацією невеликих за часом завдань, які користувач виконує протягом дня, тому для планування цілей на тривалий періоду їх використання недоцільне.

4) додатки для OKR для команд або компаній. Такі додатки є близькими за функціоналом: планування цілей, які потрібно виконати за тривалий проміжок часу, але вони не призначені для персонального використання.

1.2 Технологія

Вибір технології, за допомогою якої буде створюватись мобільний додаток, залежав від наступних факторів:

1) крос-платформеність - чи можливо використовувати одну кодову базу для запуску додатку на операційних системах Android та iOS;

2) популярність та підтримка - цей пункт включає у себе кількість та якість документації, регулярні оновлення та покращення технології; спільнота, яка бере участь у житті технології;

3) перфоманс - показник кадрів в секунду та інтенсивність споживання ресурсів девайсу при виконанні певного сценарію;

4) інструментарій розробника - графічні редактори з підтримкою технології, інструменти відлагодження, профілювання, автоматизація тестів тощо [5].

Найкращі показники перфомансу мають додатки, зроблені нативно під Android (мовою Java або Kotlin) та iOS (мовою Swift або Objective-C). Проте нативна розробка означає різні кодові бази для кожної ОС. Це ускладнює розробку та зменшує консистентність додатку. Крос-платформені технології дозволяють розробляти додаток під декілька ОС, використовуючи єдину кодову базу [6]. На даний момент найбільш поширеними є такі фреймворки для крос-платформеної розробки нативних додатків: Flutter, React Native, Xamarin, Ionic [7].

Як видно зі статистики заданих питань на StackOverflow з тегами технологій, Xamarin та Ionic не мали великої популярності та наразі втрачають її, в той час як питання по Flutter та React Native обговорюються все більше і більше. За останні роки, Flutter навіть піднявся вище за Swift - основну мову у нативній розробці під iOS [8].

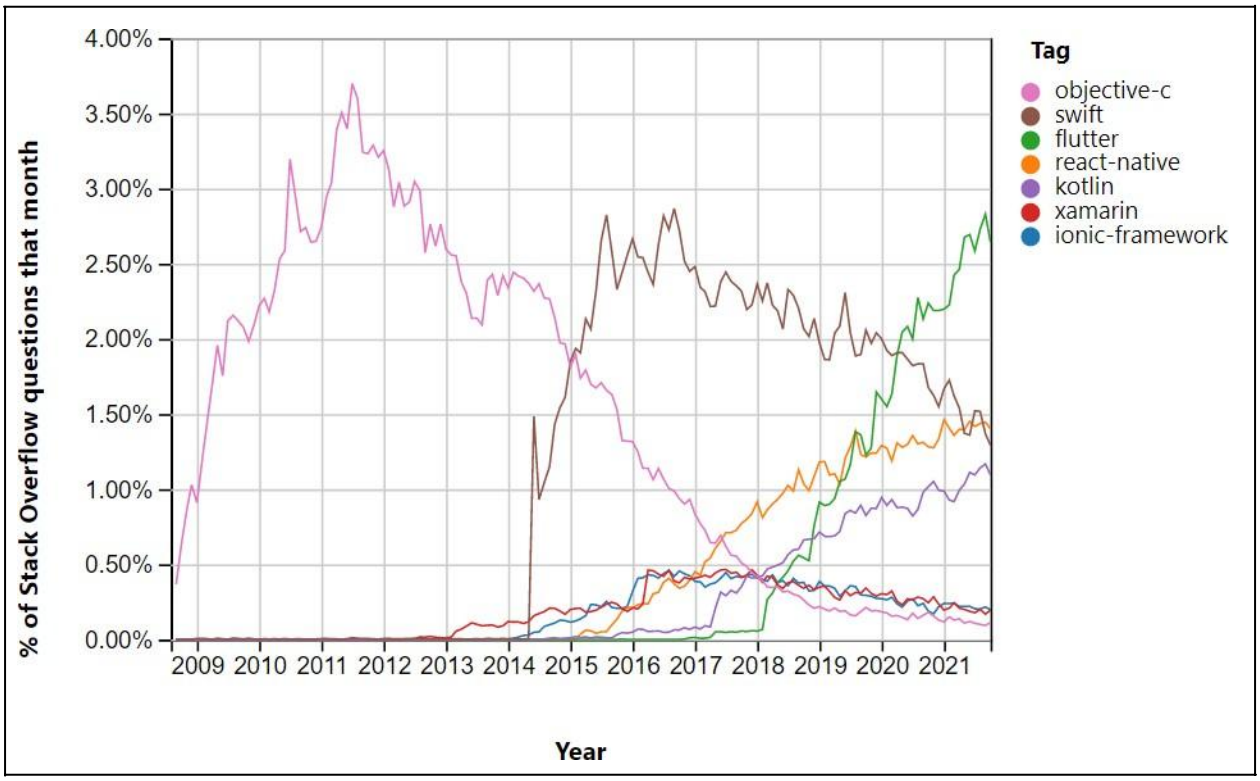

Рис. 1.1 - динаміка обговорення питань на StackOverflow

Також, можна порівняти кількість Stars на GitHub репозиторіях фреймворків.

Репозиторій	flutter	react-native	ionic-framework	Xamarin.Forms
К-сть Stars	133k	99.6k	45.7k	5.5k

Табл. 1 - Кількість Stars на GitHub репозиторіях фреймворків

Отже, наразі найбільш поширеними є два фреймворки для крос-платформеної розробки - Flutter та React Native.

Хоча стилі розробки на Flutter і на React Native схожі (декларативний UI, який будується із невеликих компонентів), реалізація крос-платформеності відрізняється і це впливає на перфоманс [9].

Застосунки React Native пишуться мовою JavaScript (або TypeScript). При запуску додатку, створюється JS Virtual Machine, яка обмінюється повідомленнями з нативними модулями ОС через React Native Bridge для рендеру інтерфейсу. Така комунікація створює додатковий оверхед і погіршує показники перфомансу [10].

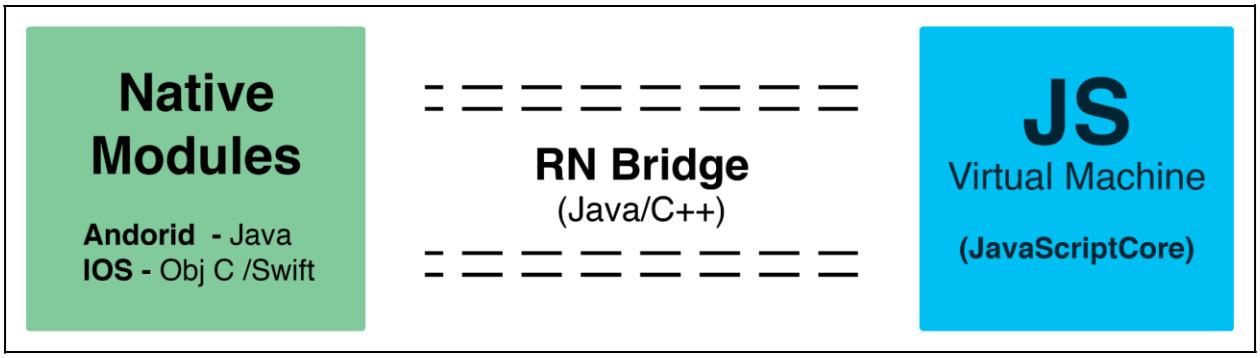

Рис. 1.2 - схема комунікації React Native з нативними модулями

Навідміну від React Native, Flutter не використовує міст для спілкування з нативними модулями, а компілює код мовою Dart (який пише розробник) у нативний код відповідною до ОС мовою. Таким чином Flutter виявляється швидшим за React Native [11]. Така компіляція є Ahead Of Time (AOT), але також в режимі Debug є можливість Just In Time компіляції, що дозволяє оновлювати додаток набагато швидше порівняно з AOT компіляцією. Це дає змогу писати маленькими частинами функціоналу та одразу тестувати їх.

За замовчуванням в проекті Flutter використовується лінтер, рекомендований Flutter, що аналізує код та підтримує його консистентність. Окрім стандартного відлагоджувача, Flutter також має Widget Inspector, що дозволяє переглядати поточний стан дерева компонентів інтерфейсу запущеного застосунку; профайлери для моніторингу перфомансу

(оперативна пам’ять, кількість кадрів за секунду тощо); та інструмент для автоматизації виконання сценаріїв у застосунку - Flutter Driver. Поєднуючи профілювання та Flutter Driver, можна консистентно збирати дані про перфоманс застосунку.

Отже, Flutter є наразі найбільш поширеним крос-платформеним фреймворком для створення нативних застосунків, має показники перфомансу, що конкурують з іншими фреймворками. Також, детальна документація та інструментарій розробника дозволяють підтримувати правильність роботи додатку.

1.3 Користь додатку

Мета додатку - допомогти користувачеві організувати процес проходження технічної літератури. З цього випливає питання: яким чином додаток допоможе в організації? Користувач матиме змогу:

1) зберігати списки літератури та інформацію про кількість сторінок;

2) встановлювати проміжок часу на кожен список;

3) дізнаватися поточну інформацію про проміжок часу (скільки минуло, скільки залишилось днів тощо) та про прогрес виконання (скільки сторінок пройдено, скільки залишилось пройти тощо).

Справедливим буде поставити запитання кожному пункту вище: чому ця функція є корисною для користувача?

1) зберігати списки літератури та інформацію про кількість сторінок - при зменшенні інформації, яку потрібно пам’ятати, зменшується рівень стрессу [12]. Записування цілей у 1,2–1,4 рази збільшує шанс їх досягнення [13].

2) встановлювати проміжок часу на кожен список - дедлайн дозволяє розрахувати, яку частину цілей потрібно проходити за відносно маленький,

зручний для людини, проміжок (день, тиждень тощо), тобто налаштовує регулярність проходження матеріалу [14].

3) дізнаватися поточну інформацію про проміжок часу та прогрес виконання - найбільш розвиненим у людини є зорове сприйняття [15]. Візуалізація прогресу підтримує мотивацію та зацікавленість користувача продовжувати рух до цілі [16].

1.4 Постановка задачі

Розробити мобільний додаток, який дозволить слідкувати за прогресом проходження технічної літератури, а також додати можливість інтеграції із зовнішнім каталогом літератури для студентів університету. Для цього: проаналізувати технології розробки мобільних додатків та обрати найбільш доцільну на момент написання; спроектувати модель даних, що підходить під формат та якою оперуватиме користувач при взаємодії з додатком, визначити можливі дії користувача та розробити архітектуру додатку, що дозволить їх реалізацію. Провести профілювання та оптимізацію недосконалих місць інтерфейсу користувача, якщо такі виявляться.

2 Теоретична частина

2.1 Модель даних

Для прикладу моделі візьмемо семестр навчання в університеті. Студент має декілька дисциплін. Нехай це: математика, фізика, програмування. З кожного предмету студент має певний набір рекомендованої літератури. Таким чином, маємо: за проміжок часу, який дорівнює тривалості семестру, студент має прочитати набір літератури з ймовірністю того, що деякі книжки потрібно буде опрацювати не повністю, а лише певні розділи, які важливі для програми семестру. Так само може бути для попереднього семестру, та для наступного. Отже, маємо наступну схему:

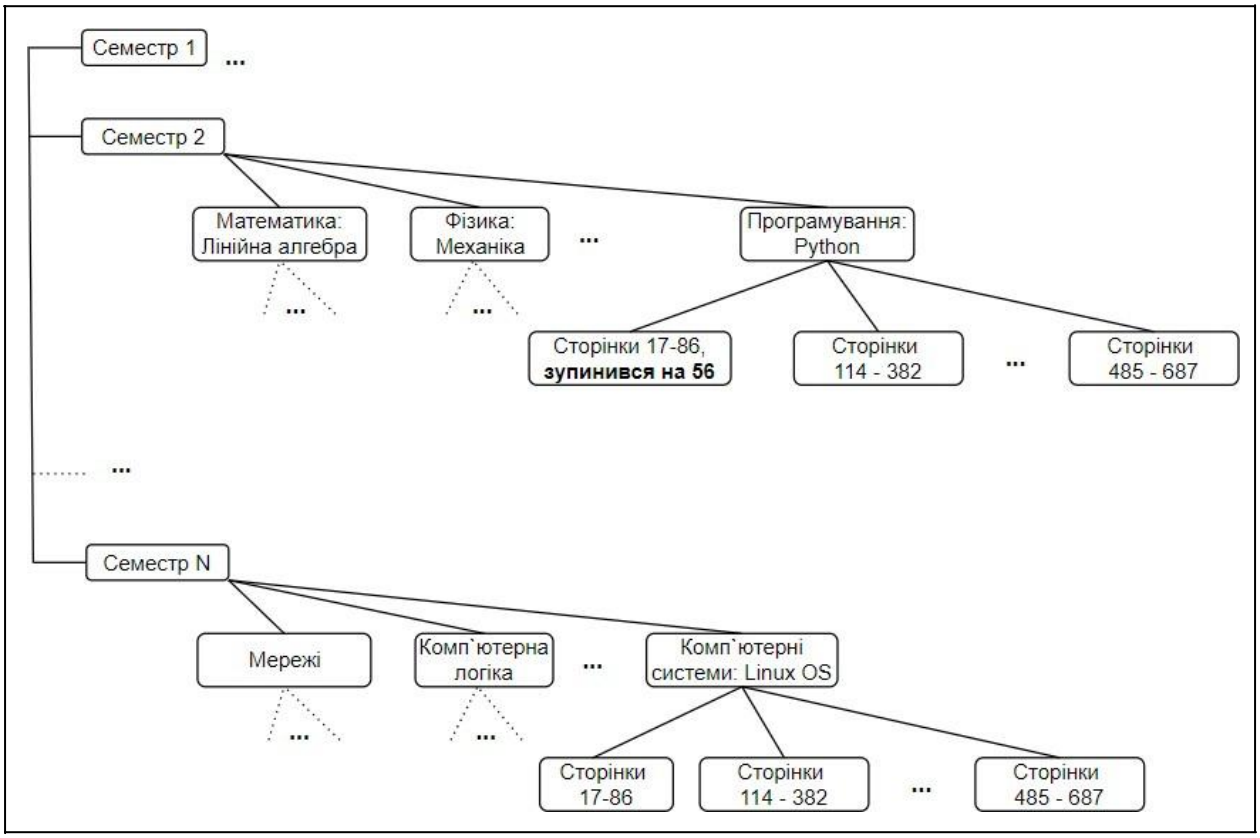

Рис. 2.1 - приклад ієрархії моделі даних

На основі прикладу з рисунку 3, можна побудувати абстрактну схему моделі даних:

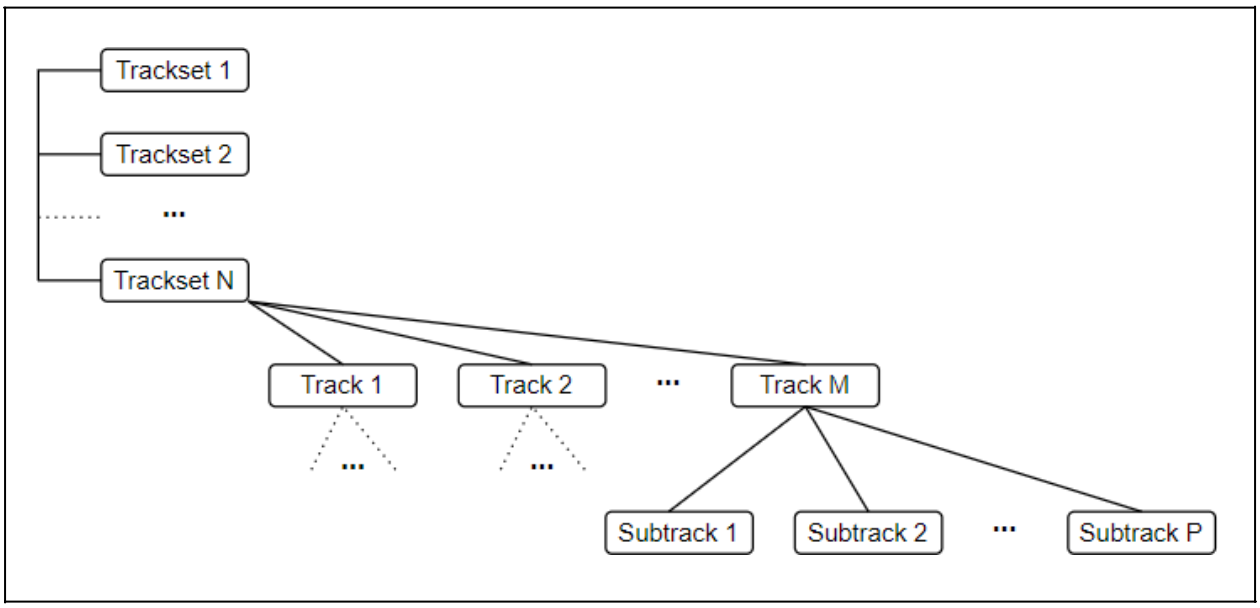

Рис. 2.2 - ієрархія основних сутностей

Як можна побачити на рисунку 2.2, найвищою сутністю є Trackset (на рисунку 2.1 це - семестр). Trackset відповідає за проміжок часу та містить у собі колекцію з сутностей Track. Track (на рисунку 2.1 - книга) містить у собі колекцію з сутностей Subtrack. Subtrack (на рисунку 2.1 - діапазон сторінок) точно визначає частини з Track, які потрібно опрацювати, а також зберігає інформацію про те, де зупинився користувач. Так на рисунку 2.1 виділено Subtrack з інформацією про те, на якій сторінці зупинився користувач. Track підраховує свій прогрес із дочірніх Subtracks, а Trackset, відповідно, із дочірніх Track. Таким чином отримується інформація про загальний прогрес певного Trackset.

Хоча основним застосуванням додатку є технічна література, сутності абстраговані від цього, адже такий формат даних можна використовувати й для інших активностей. Наприклад, це може бути певний навчальний курс, де

сам курс це Trackset, розділи курсу - набір з Track, та підрозділи - набір з Subtrack.

2.2 Дії користувача

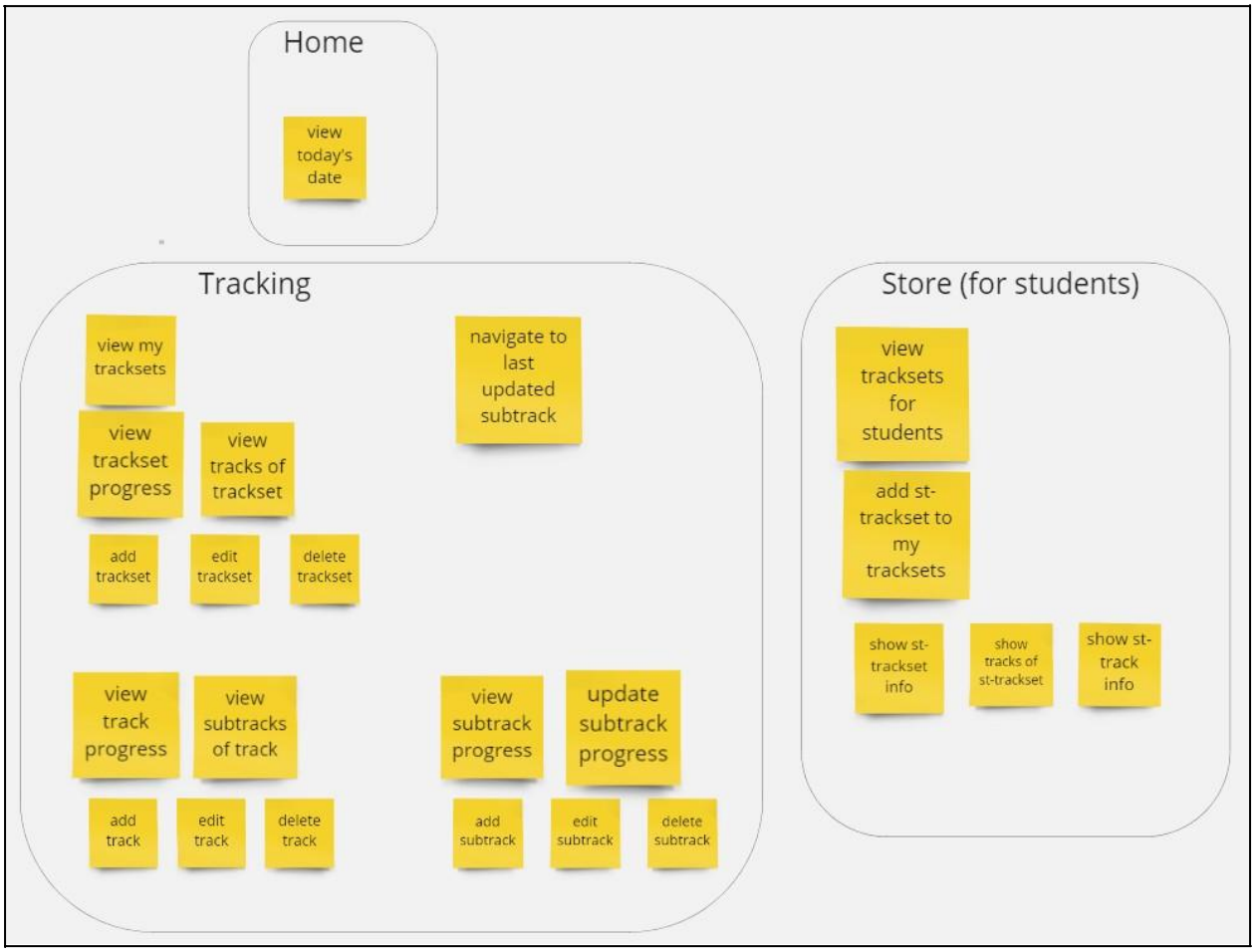

Рис. 2.3 - дії користувача, розбиті по модулям

На рисунку 2.3 зображено дії користувача у додатку. Розмір стікера підкреслює частоту виконання дії, а також її важливість. Так, наприклад, важливою дією є update subtrack progress, або add st-trackset to my tracksets.

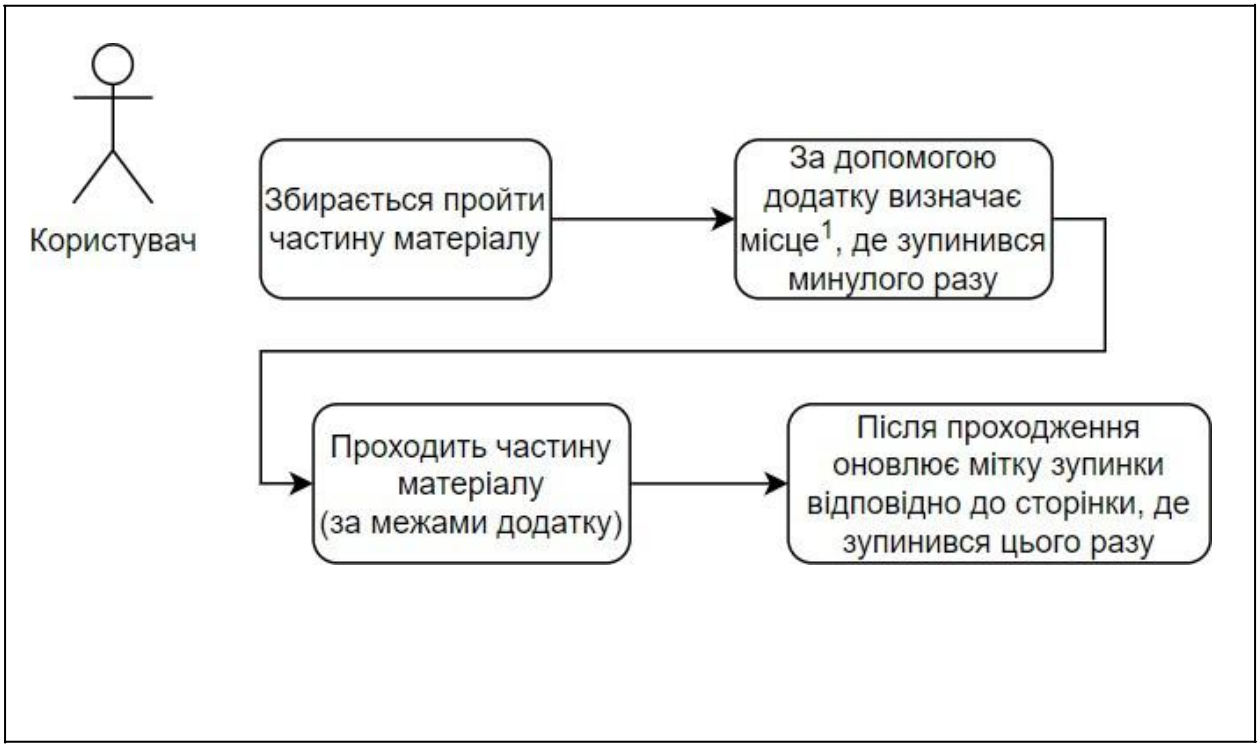

Рис. 2.4 - найчастіший сценарій використання додатку

Передбачається, що сценарій з рисунку 2.4 користувач може виконувати один або декілька разів на день. Його виконання повинно бути максимально простим. Це означає, що не повинно бути складних способів вводу, кількість дій має бути мінімізована, та користувач не має запам’ятовувати або вираховувати власноруч свій прогрес. Тому маємо наступні вимоги для даного сценарію:

1) користувач повинен мати можливість відкрити останній оновлений Subtrack в одне натискання, після заходу у додаток;

2) оновлення прогресу після сесії проходження матеріалу повинно відбуватись таким чином: з джерела прочитання користувач отримує позначку (номер сторінки), на якій зупинився - та змінює попередню позначку на поточну, при чому зміна числової позначки повинна відбуватись прокручуванням списку з чисел, мінімізуючи складність дій. Таким чином

користувачу потрібно на декілька секунд запам’ятати одне число та внести його в додаток, не тримаючи у пам’яті кількість пройдених сторінок та іншого.

Окрім швидкого доступу до останнього оновленого Subtrack, потрібно виділити такі загальні вимоги:

3) користувач повинен мати доступ до перегляду будь-якого Trackset, Track та Subtrack, які є у його списку.

Очевидно, що для проходження матеріалу, користувач має створити Trackset, набір Tracks, та в кожному Track набір Subtracks, або додати до свого списку Trackset із бібліотеки створених для студентів.

Тому маємо вимоги для сценарію створення сутностей:

4) користувач повинен мати змогу створити Trackset, при створенні вказати назву трексета, дату початку проходження та дату закінчення проходження;

5) користувач повинен мати змогу створити Track у певному Trackset, при створенні вказати назву трека;

6) користувач повинен мати змогу створити Subtrack у певному Track, при створенні вказати діапазон сабтрека: першу позначку включно та останню позначку включно. Система має перевірити, що діапазон не перетинається із попередньо створеними.

Для додавання заготовленого Trackset із бібліотеки для студентів маємо такі вимоги:

7) користувач повинен мати змогу переглянути усю ієрархію трексета перед додаванням;

8) користувач повинен мати змогу додати Trackset з бібліотеки до свого списку, при додаванні вказати дату початку проходження та дату закінчення проходження;

Також другорядними діями користувача є редагування та видалення сутностей - в разі помилки при створенні, зміни плану проходження або інших факторів. Тому маємо такі вимоги:

9) користувач повинен мати змогу редагувати та видаляти будь-який Trackset, Track, Subtrack зі свого списку.

Сценарії додавання, редагування та видалення не передбачаються бути частими та не є основними, тому вимоги щодо їх реалізації інтерфейсу не є строгими;

10) користувач повинен мати змогу переглядати статистику відносно кожного Trackset, Track та Subtrack зі свого списку, а саме:

Trackset: кількість днів минуло, кількість днів залишилось, кількість одиниць всього, кількість одиниць пройдено, кількість одиниць лишилось, мінімальна кількість одиниць для проходження щоденно для завершення Trackset вчасно.

Track: кількість одиниць всього, кількість одиниць пройдено, кількість одиниць лишилось.

Subtrack: кількість одиниць всього, кількість одиниць пройдено.

2.3 Архітектура додатку

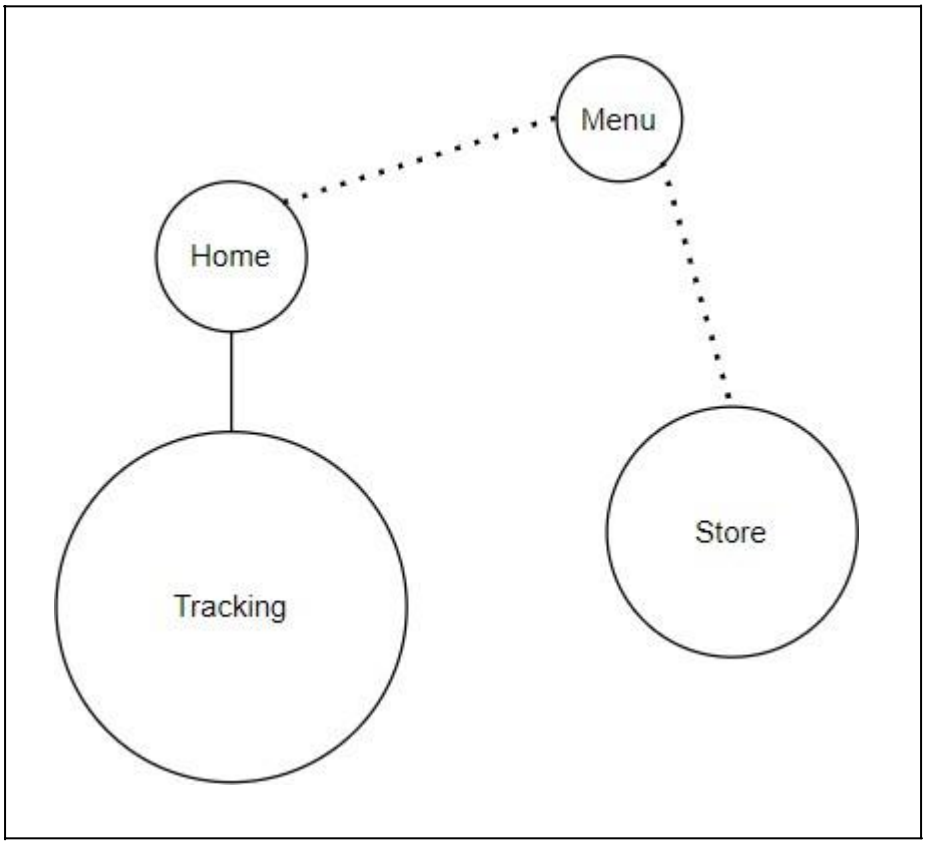

Рис. 2.5 - взаємодія модулів у додатку

Модулі Home та Menu допоміжні для Tracking та Store.

Home є першим модулем при запуску додатку та він відповідальний за те, щоб надати користувачеві доступ до модулів Menu та Tracking.

Tracking це основний модуль, в якому користувач працює з основними сутностями (переглядає прогрес, оновлює прогрес, створює, редагує, видаляє сутності тощо). За допомогою Menu користувач може перейти до модулю Store, який відповідає за відображення заготовлених макетів трексетів для студентів (які зберігаються поза девайсом). В цьому модулі користувач може додати такий макет до свого списку і макет буде ковертований у Trackset, який буде доступний в модулі Tracking.

Таким чином, Tracking та Store не є повністю ізольованими один від одного, адже має бути можливість обраний макет трексету зі Store додати в Tracking. Така операція означає, що модуль Store повинен мати можливість ініціювати зміну стейту модулю Tracking, а Tracking в свою чергу надати інтерфейс для цього таким чином, щоб отримати об’єкт від Store та далі оперувати над ним. Додати Trackset користувач може також і з середини модулю Tracking, створивши його власноруч. При додаванні Trackset щонайменше потрібно оновити інтерфейс, щоб показати доданий Trackset користувачу.

Отже, маємо:

Можливі такі процеси, які відрізняються один від одного вхідними даними, джерелом ініціації, але мають однаковий кінцевий результат, на основі якого інтерфейс додатку має змінитись за однаковим принципом.

Тому в модулі Tracking потрібно виокремити компонент, який відповідатиме за збереження та оновлення стейту (даних) модуля, а інтерфейс модуля буде прослуховувати зміни стейту та відповідно перебудовуватись. Таким чином може існувати декілька джерел ініціації зміни стейту, та декілька джерел, що прослуховують зміни стейту. Коли стейт змінюється - всі компоненти, що були підписани на зміни, зможуть відповідно зреагувати на зміну, при цьому вони не повинні залежати від джерела зміни.

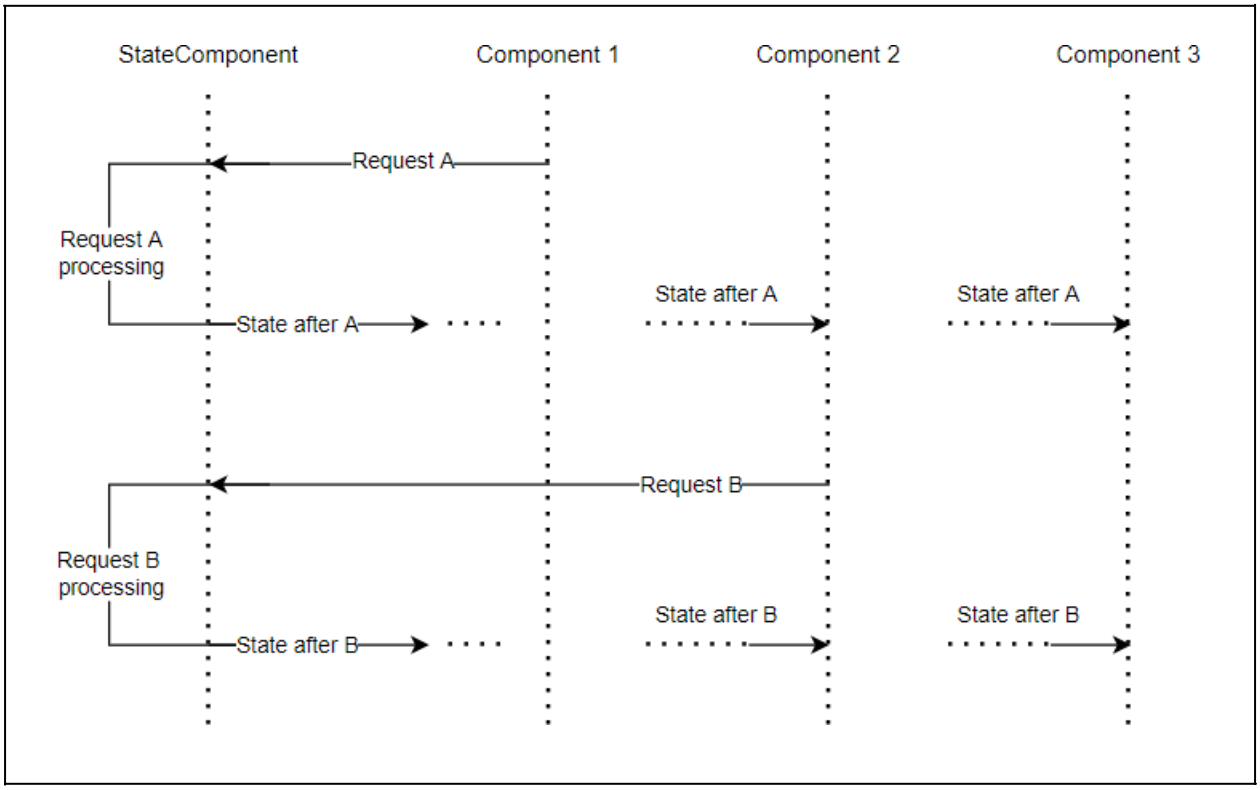

Рис. 2.6 - принцип роботи компонента стейту

На рисунку 2.6 Component 2 та Component 3 підписані на зміни стейту компонента StateComponent. Component 1 відправляє Request A до компонента стейту, в результаті якого стейт змінюється і стає State after A, та підписані компоненти можуть відреагувати на зміну. Потім Component 2 відправляє Request B і відбувається теж саме, що і при Request A. Оголошення зміни стейту відображено з використанням пунктирних ліній, тому що StateComponent напряму не взаємодіє з жодним компонентом. Компоненти залежать від StateComponent, але не навпаки.

Такий компонент стейту дозволить зберігати дані в пам’яті, але також є потреба у постійному сховищі даних, щоб після перезаходу у додаток дані користувача не були втрачені. Дані з модуля Tracking не потребують великого об’єму пам’яті та критерій швидкодії та незалежності додатку переважає

критерій надійності збереження даних, тому було вирішено використовувати локальну базу даних мобільного девайсу.

Дані з модуля Store повинні бути доступні всім користувачам додатку, також має бути можливість змінювати ці дані незалежно від певного девайсу користувача. Отже такі дані мають зберігатися у віддаленій базі даних, а користувачі зможуть отримати доступ до них за допомогою мережі інтернет.

Маємо більш детальну схему модулів Tracking та Store:

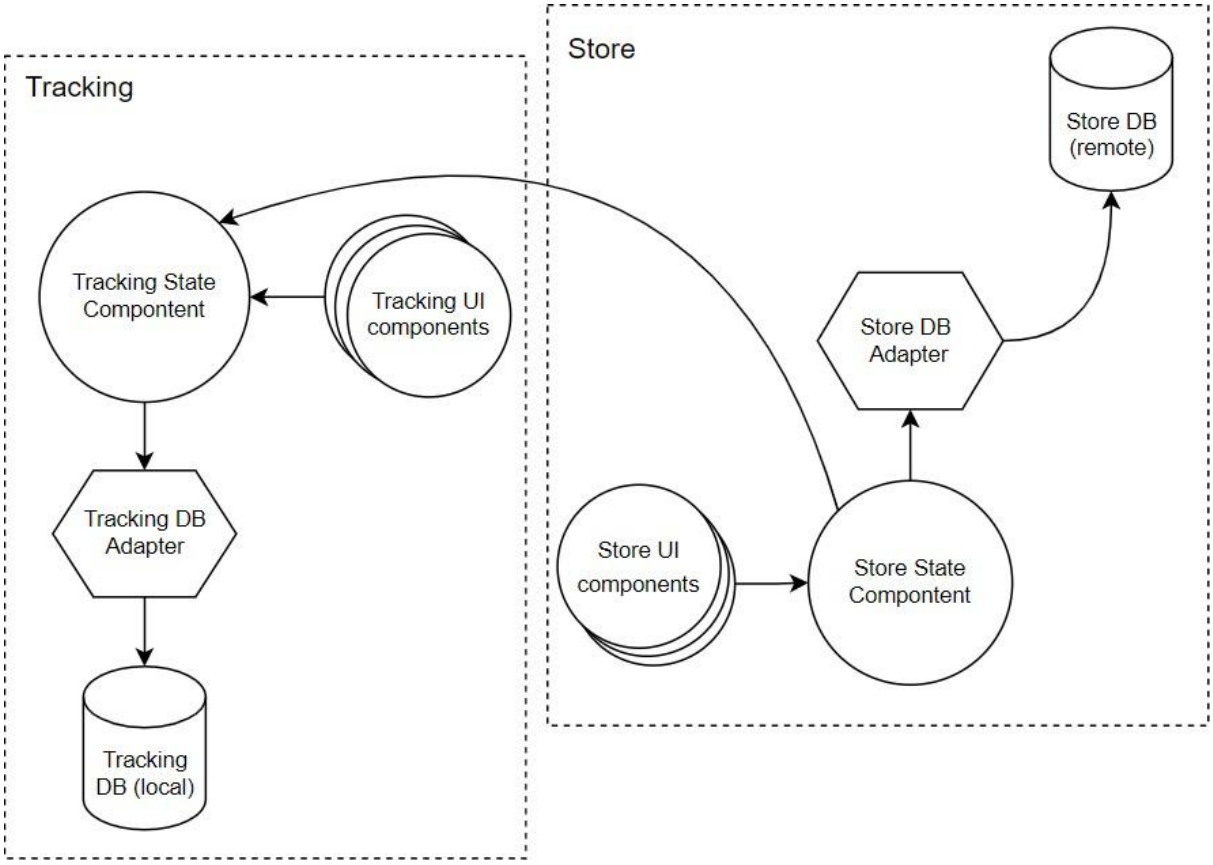

Рис. 2.7 - архітектура модулів Tracking та Store

Напрям стрілки на рисунку 2.7 показує залежність між компонентами. Компонент, від якого прямує стрілка, залежить від компоненту, до якого прямує стрілка.

Компоненти UI це, наприклад, відповідальний за список трексетів компонент, відповідальний за створення трексету компонент тощо.

Компоненти UI залежні від стейт компоненту. Стейт компонент залежний від компоненту DB Adapter. Компоненти типу DB Adapter інкапсулюють у собі роботу з базою даних для того, щоб не забруднювати інші компонентами деталями реалізації бази даних. Це зменшує відповідальність компонентів стейту, а також дозволяє підміняти компонент DB Adapter при тестуванні інших частин додатку.

2.4 Макет інтерфейсу

Скріншоти всіх видів користувача у додатку наведено у додатку В.

Як було описано у розділі 2.1, основні сутності вкладаються одна в одну - Trackset має набір Tracks, а Track - набір Subtracks. Така вкладеність має бути відображена в інтерфейсі. Слід врахувати, що розмір екрана мобільного девайса є відносно невеликим, тому в один момент часу не доцільно розміщувати забагато інформації на екрані та ускладнювати його розуміння користувачем. Тому було вирішено поділити інтерфейс основних сутностей на такі частини:

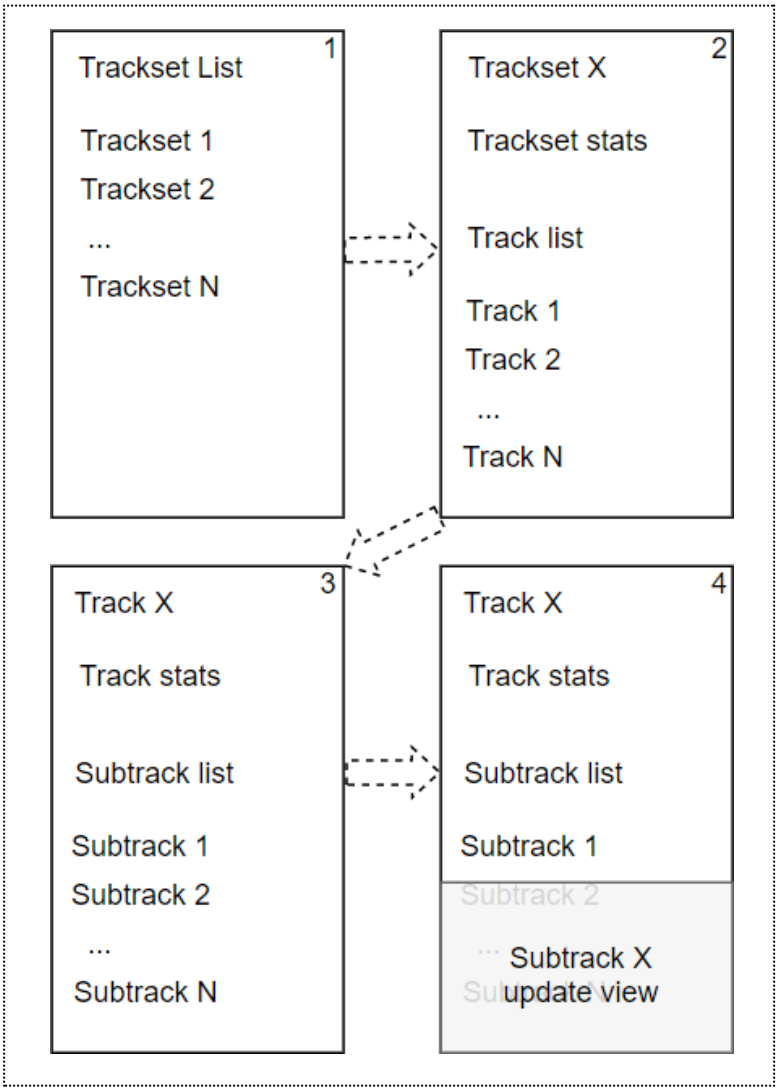

Рис. 2.8 - інтерфейс основних сутностей

Перше, що бачить користувач після запуску додатку - це список трексетів з мінімальною інформацією про трексет - для того, щоб користувач міг ідентифікувати йому потрібний та перейти до другого виду. Другий вид відповідає за один трексет. На ньому користувач має доступ до статистики трексету та списку дочірніх треків. Також, звідси користувач може почати процес редагування обраного трексету. При натисканні на один з треків, користувачеві відкривається третій вид, який відповідає за один трек. На ньому зображено елемент управління для початку редагування трека,

статистика відносно трека, та список дочірніх сабтреків. При натисканні на сабтрек користувач відкриє компонент для оновлення сабтрека, в якому і відбувається безпосередньо дія оновлення прогресу.

Відчуття вкладеності трексетів, треків та сабтреків було досягнено завдяки підходу до переходів між видами інтерфейсу. Переходи 1-2, 2-3 (рис. 2.8) є анімованими таким чином, що натиснутий елемент списку опиняється вгорі екрана, а внутрішній елемент збільшує висоту щоб займати простір, що залишився, і зміщує нижчі елементи списку. На рисунку 10 зображено відкриття виду для Trackset 3:

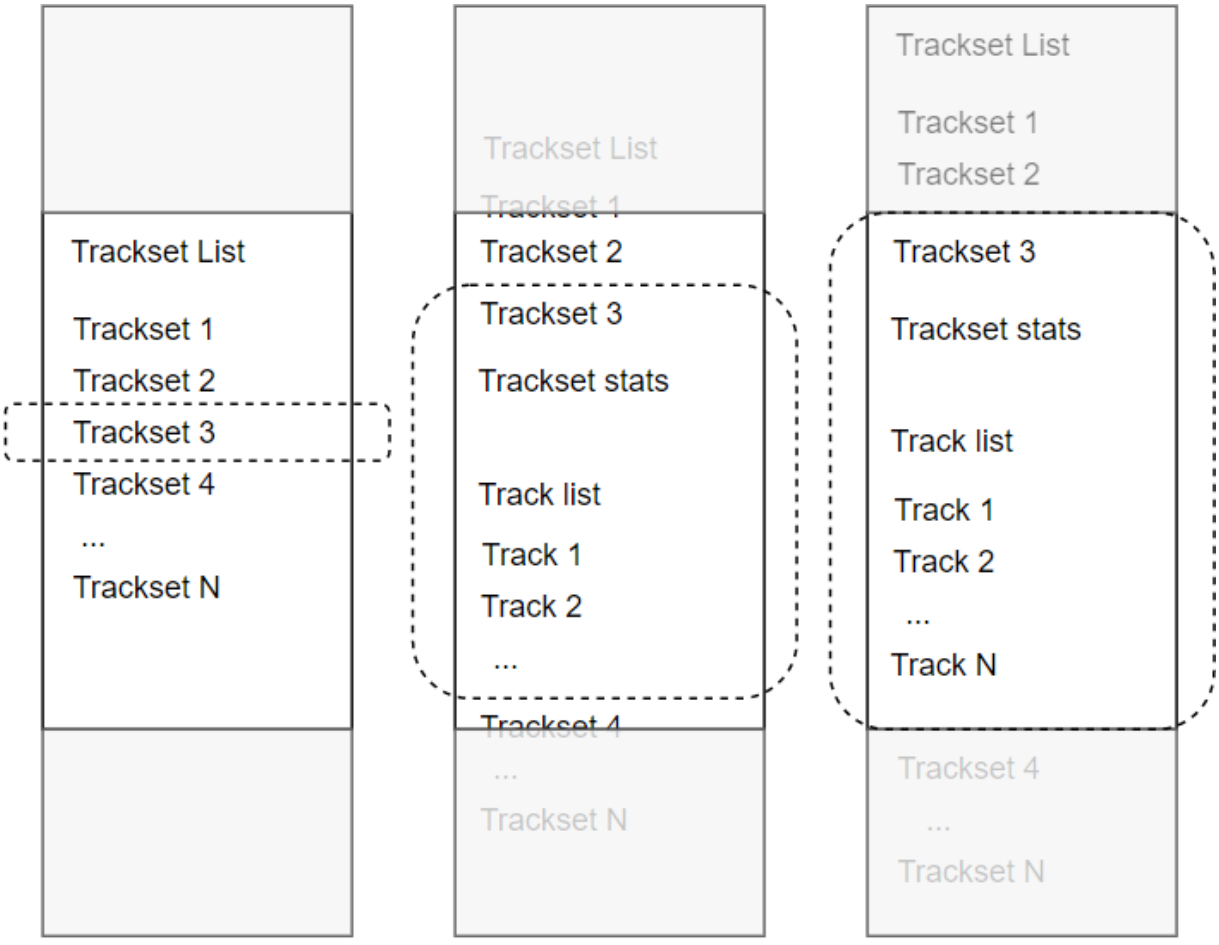

Рис. 2.9 - процес переходу видів користувача

В модулі Store компоненти макетів основних сутностей побудовані за таким самим принципом, але вони не мають статистики та можливості редагування та видалення, тому що призначені для огляду сутності.

3 Практична частина

Як було зазначено у розділі 1.2 “Технологія”, було обрано фреймворк Flutter на мові Dart.

3.1 Модель даних

Сутності Trackset, Track, Subtrack у коді представлені класами:

```
class Trackset extends Equatable {
  final String id;
  final String userId;
  final String name;
  final DateTime startAt;
  final DateTime endAt;
  final NormalizedList<Track, String> tracks;
  final bool isDeleted;
  ...
}
```

Лістинг 3.1 - клас Trackset

```
class Track extends Equatable {
  final String id;
  final String tracksetId;
  final String name;
  final NormalizedList<Subtrack, String> subtracks;
}
```

Лістинг 3.2 - клас Track

```
class Subtrack extends SubtrackRange {
  final String id;
  final String trackId;
  final DateTime? updatedAt;
}

class SubtrackRange extends Range {
  final int pointer;
}
```

```
class Range extends Equatable {
  final int start;
  final int end;
  ...
}
```

Лістинг 3.3 - клас Subtrack

Як можна побачити, всі класи наслідуються від Equatable. Об’єкти класів-нащадків Equatable порівнюються між собою значенням полів, а не хешкодів. Це сприяє порівнянню стейта у компоненті стейта, щоб визначити, чи дійсно змінився стейт, та чи потрібно публікувати нове значення стейта у потік (stream).

Також слід зауважити, що поле tracks у Trackset та поле subtracks у Track обидва є об’єктами класу NormalizedList<T, I>.

```
class NormalizedList<T, I> extends Equatable {
  final Map<I, T> byId;
  final Set<I> all;
  ...
}
```

Лістинг 3.4 - клас NormalizedList

NormalizedList це обгортка над класом Map<I, T>, який дозволяє доступ до елемента T по ключу I із складністю O(1), навідміну від пошуку елемента списку List<T> із складністю O(N). Операція пошуку елемента часто використовується у операціях додавання, видалення та оновлення. Наприклад, оновлення Subtrack означає знаходження по id батьківського Trackset та Track.

Моделі сутностей також містять у собі логіку обрахунку статистики. Найнижча сутність в ієрархії - Subtrack - обраховує дані про прогрес (скільки всього, скільки пройдено, скільки лишилось) на основі своїх полей start, end та pointer:

```
int get length => end - start + 1;

int get done {
  if (isOnStart) return 0;
  if (isOnEnd) return length;
  return pointer - start;
}

int get left {
  if (isOnStart) return length;
  if (isOnEnd) return  0;
  return end - pointer + 1;
}

double get progress => func.progress(done, length);
```

Лістинг 3.5 - обрахунок статистики Subtrack

Для обрахунку статистики на рівні Track потрібно просто підсумувати статистику дочірніх Subtrack:

```
int get length =>
      subtracks.entities.fold(0, (sum, subtrack) => sum += subtrack.length);
int get done =>
      subtracks.entities.fold(0, (sum, subtrack) => sum += subtrack.done);
int get left =>
      subtracks.entities.fold(0, (sum, subtrack) => sum += subtrack.left);

double get progress => func.progress(done, length);
```

Лістинг 3.6 - обрахунок статистики Track

Відповідно, для статистики на рівні Trackset потрібно підсумувати статистику дочірніх Track:

```
int get length =>
      tracks.entities.fold(0, (sum, track) => sum += track.length);
int get done => tracks.entities.fold(0, (sum, track) => sum += track.done);
```

```
int get left => tracks.entities.fold(0, (sum, track) => sum += track.left);

double get progress => func.progress(done, length);
```

Лістинг 3.7 - обрахунок статистики Trackset

Trackset має також рахувати скільки одиниць потрібно проходити щоденно:

```
int get dailyGoal {
  final _daysLeft = daysLeft();
  if (_daysLeft == 0) {
    return 0;
  }
  return (left / _daysLeft).ceil();
}
```

Лістинг 3.8 - обрахунок щоденного проходження Trackset

Та дані щодо проміжку часу (скільки всього днів, скільки минуло, скільки лишилось):

```
int get totalDays => endAt.difference(startAt).inDays + 1;

bool hasStarted([DateTime? today]) {
  var _today = toDateOnly(today ?? DateTime.now());
  return !_today.isBefore(startAt);
}

bool hasEnded([DateTime? today]) {
  var _today = toDateOnly(today ?? DateTime.now());
  return _today.isAfter(endAt);
}

int daysPassed([DateTime? today]) {
  if (!hasStarted(today)) {
    return 0;
  } else if (hasEnded(today)) {
    return totalDays;
  } else {
```

```
    final _today = toDateOnly(today ?? DateTime.now());
    return _today.difference(startAt).inDays;
  }
}

int daysLeft([DateTime? today]) {
  if (!hasStarted(today)) {
    return totalDays;
  } else if (hasEnded(today)) {
    return 0;
  } else {
    final _today = toDateOnly(today ?? DateTime.now());
    return endAt.difference(_today).inDays + 1;
  }
}
```

Лістинг 3.9 - обрахунок часової статистики Trackset

Комбінація даних та логіки в одному класі слідує ідеї об’єктно-орієнтовного програмування та спрощує розуміння коду, додаючи йому згуртованості (cohesion).

Такі класи практично не мають залежностей та є “ванільним” Dart, тому добре піддаються юніт-тестуванню. Прикладом є один з тестів логіки Subtrack:

```
test('pointer-is-between-start-and-end', () {
  const subtrack = SubtrackRange(
    start: 3,
    end: 10,
    pointer: 6,
  );
  expect(subtrack.done, 3);
  expect(subtrack.left, 5);
});
```

Лістинг 3.10 - приклад тесту логіки Subtrack

Описані вище методи можна знайти по шляху lib/features/tracking/models. Відповідні їм моделі для модуля Store знаходяться по шляху lib/features/store/models, а також там є методи маппінгу моделей Store у моделі Tracking. Загальні моделі, які не відносяться до певного модулю, знаходяться по шляху lib/core/models. Прикладом такої моделі є клас NormalizedList.

3.2 Інфраструктура

Додаток Flutter складається з дерева компонентів. Компонент може шукати інші компоненти, які знаходяться вище по дереву. За таким принципом копоненти інтерфейса отримують доступ до компонента стейту (TrackingBloc у модулі Tracking, TrackingStoreBloc у модулі Store). Ініціалізація компонентів стейту відбувається у найвищому компоненті MyApp, та перед цим відбувається ініціалізація бази даних TrackingDb модуля Tracking. Після ініціалізації, об’єкт db передається у компонент MyApp:

```
void main() async {
  ...

  final db = TrackingDb(dbName: DbConfig.dbName);
  await db.open();

  final myApp = MyApp(db: db);
  runApp(myApp);
}
```

3.11 - ініціалізація компонента БД TrackingDb

MyApp всередині наповнює контейнер залежностей базами даних та компонентами стейту. Як контейнер використовується бібліотека provider, а саме MultiRepositoryProvider (для БД), та MultiBlocProvider (для компонентів стейту). Також, компоненти стейту отримують потрібні їм залежності через конструктор при створенні.

```
class MyApp extends StatelessWidget {
  ...
  final TrackingDb db;

  @override
  Widget build(BuildContext context) {
    return MultiRepositoryProvider(
      providers: [
        RepositoryProvider<TrackingDb>.value(value: db),
        RepositoryProvider<TrackingStoreDb>.value(
          value: TrackingStoreDb(),
        ),
      ],
      child: MultiBlocProvider(
        providers: [
          BlocProvider(
            lazy: false,
            create: (context) {
              final db = Provider.of<TrackingDb>(context, listen: false);
              return TrackingBloc(db: db);
            },
          ),
          BlocProvider(
            create: (context) {
              final db = Provider.of<TrackingStoreDb>(
                                    context, listen: false);
              return TrackingStoreBloc(db: db);
            },
          )
        ],
        child: ...    );
  }
}
```

Лістинг 3.12 - регістрація залежностей

Після заповнення залежностей дерево компонентів продовжується іншими глобальними компонентами: навігатором та компонентом снекбару (для відображення випливаючих повідомлень). Повний код інфраструктури можна знайти по шляху lib/main.dart.

Дерево компонентів після запуску додатку виглядає таким чином:

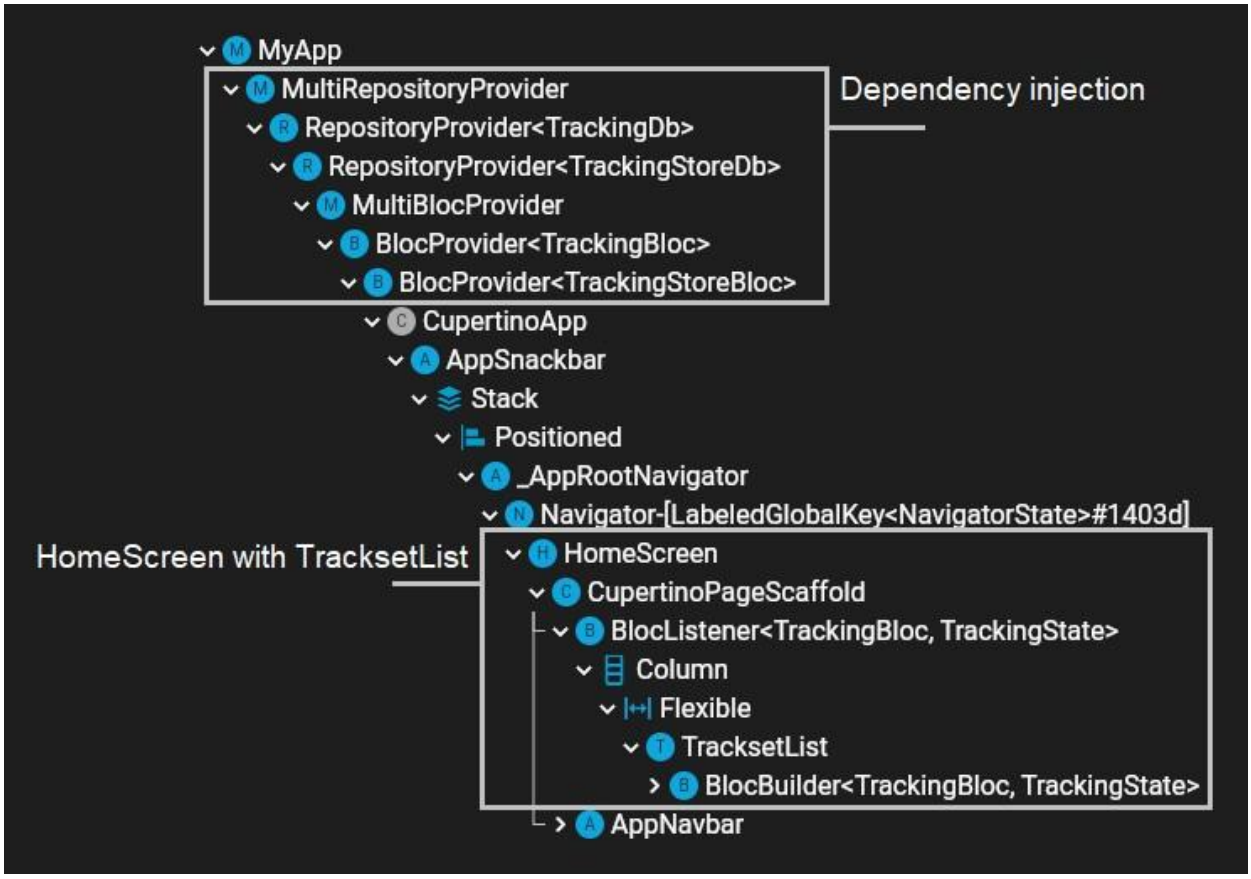

Рис. 3.1 - дерево компонентів після запуску додатку

У верхній частині можна побачити компоненти, які відповідають за dependency injection, а в нижній частині - компонент HomeScreen із TracksetList в ньому. Нижче від TracksetList також знаходяться багато компонентів, але вони приховані на рисунку 3.1.

3.3 Стейт

Модулі Tracking та Store мають стейт всередині модуля. Компоненти стейтів є центральними у взаємодії. Вони відповідають за тимчасове зберігання стейту в пам’яті, читання та запис в базу даних (за допомогою проміжних компонентів), а також інтерфейс будується відповідно поточного стейту.

Основою для компонентів стейту було обрано бібліотеку bloc. Принцип роботи компонента BLoC відповідає принципу, описаному у розділі 2.3 “Архітектура додатку”. Бібліотека надає клас Bloc, який є безпосередньо компонентом стейту і відповідає за зберігання стейту, приймання запитів на обробку (request), та керування потоком стейтів. Компоненти TrackingBloc та TrackingStoreBloc наслідують Bloc:

```
class TrackingBloc extends Bloc<TrackingEvent, TrackingState> {
  ...
}
```

Лістинг 3.13 - клас TrackingBloc

```
class TrackingStoreBloc extends Bloc<TrackingStoreEvent, TrackingStoreState>
{
  ...
}
```

Лістинг 3.14 - клас TrackingStoreBloc

Також, бібліотека надає декілька компонентів для слухання за зміною стейту: BlocBuilder, який знаходиться у дереві компонентів, та при зміні стейту викликається метод builder, який вертає наступний компонент. У наступному прикладі функція builder вертає ExpandableList, який всередині будує список трексетів state.tracksets, отриманих із стейту.

```
BlocBuilder<TrackingBloc, TrackingState>(
  builder: (context, state) {
    final tracksets = state.tracksets;
    final errorMessage = _buildErrorMessage(state);
```

```
    final isLoading = state is TrackingLoadingState;
    return ExpandableList(...);
  },
)
```

Лістинг 3.15 - приклад використання BlocBuilder

BlocListener, який знаходиться у дереві компонентів, та при зміні стейту викликає колбек listener. На прикладі BlocListener використовується для того, щоб показувати спливаюче повідомлення про помилку або про те, що трексет із списку для студентів був доданий до списку користувача.

```
BlocListener<TrackingBloc, TrackingState>(
  listener: _listenToTrackingBloc,
  child: ...
),
```

Лістинг 3.16 - приклад використання BlocListener (1)

```
void _listenToTrackingBloc(BuildContext context, TrackingState state) {
  void show(String text) {
    AppSnackbar.of(context)!.show(text: text);
  }

  if (state is TrackingErrorState) {
    if (state.shouldShowNotification) {
      show(state.description);
    }
    return;
  }
  if (state is TrackingUpdatedState) {
    if (state.isAfterTracksetSoAdded) {
      show('Added trackset ${state.updatedTrackset!.name}');
    }
  }
```

```
}
```

Лістинг 3.16 - приклад використання BlocListener (2)

Компоненти всередині дерева мають можливість знайти компонент стейту (далі блок) за допомогою Provider, якщо вище по дереву блок було інжектовано в контейнер залежностей:

```
final trackingBloc = Provider.of<TrackingBloc>(context);
```

Лістинг 3.17 - отримання TrackingBloc з провайдера залежностей

Context в даному випадку визначає позицію компонента в дереві та дозволяє взаємодіяти з іншими компонентами. Таким чином інші компоненти можуть додавати запити в блок. По конвенціям бібліотеки реквести називають івентами (event). На прикладі компонент знаходить TrackingBloc та додає запит (івент) TracksetsDeleted, та ініціює цим видалення обраних (selectedIds) трексетів:

```
TrackingBloc.of(context).add(
  TracksetsDeleted(selectedIds),
);
```

Лістинг 3.18 - додавання запиту у TrackingBloc

При додаванні запиту, Bloc всередині себе викликає метод mapEventToState, де по типу івента викликається метод-обробник:

```
@override
Stream<TrackingState> mapEventToState(TrackingEvent event) async* {
  ...
  } else if (event is TracksetsDeleted) {
    yield* _mapTracksetsDeletedToState(event);
  }
  ...
}
```

Лістинг 3.19 - виклик обробника запиту в блоці

Метод-обробник спочатку публікує TrackingLoadingState стейт для того, щоб інтерфейс показав користувачу індикатор, що відбувається операція. Далі,

використовуючи змінну normalized, відбувається видалення трексетів. Для збереження результату в базі даних, на об’єкті _db викликається метод deleteTracksets. Після успішного оновлення стейту в пам’яті та в БД, компонент публікує TrackingUpdatedState з оновленим списком tracksets, а інтерфейс відповідно зреагує на цю зміну стейту і перебудується. Якщо під час обробки станеться помилка, то компонент опублікує TrackingErrorState із блоку catch, щоб повідомити користувача про помилку.

```
Stream<TrackingState> _mapTracksetsDeletedToState(
    TracksetsDeleted event,
  ) async* {
    try {
      yield TrackingLoadingState(state);

      var normalized = state.tracksets;
      for (var id in event.ids) {
        normalized = normalized.remove(id);
      }

      await _db.deleteTracksets(event.ids.toList());

      yield TrackingUpdatedState(
        state.copyWith(tracksets: normalized),
        isAfterTracksetsDeleted: true,
      );
    } catch (ex) {
      yield TrackingErrorState(
        state,
        description: 'Failed to delete tracksets',
        failedEvent: event,
        shouldShowNotification: true,
      );
    }
```

```
}
```

Лістинг 3.20 - обробник запиту на видалення Trackset

База даних модуля Tracking є локальною на девайсі користувача. Було обрано SQLite як провайдер БД, тому що він підтримується мобільними ОС Android та iOS, а також має підтримку на Flutter, через бібліотеку sqflite.

База даних модуля Store є віддаленою і девайс користувача отримує дані з неї за допомогою мережі інтернет. Було обрано Firestore, тому що ця БД надається як сервіс у хмарі та не потребує адміністрування. Клієнт (мобільний додаток) в свою чергу має лише авторизуватись та може брати дані з віддаленої БД.

Як і для Tracking, так і для Store, для баз даних використовуються компоненти-адаптери, які розмежовують деталі комунікації з БД та інші частини додатку. Таким чином зміна провайдера БД не буде проблемою, адже модифікувати потрібно буде тільки ці компоненти-адаптери. На прикладі наведено TrackingStoreDb. Компонент стейту TrackingStoreBloc не знає про те, що ця БД є Firestore. Він використовує публічні методи (init та getTracksets), інтерфейси яких не зав’язані на конкретній реалізації:

```
class TrackingStoreDb {
  late final FirebaseFirestore _db;

  Future<void> init() async  {
    await Firebase.initializeApp();
    _db = FirebaseFirestore.instance;
  }

  Future<List<TracksetSo>> getTracksets() async {
    final snapshot = await _db.collection('tracksets').get();
    final tracksets =
        snapshot.docs.map((x) => TracksetSoMaps.fromRaw(x.data())).toList();
    return tracksets;
  }
```

```
}
```

Лістинг 3.21 - реалізація TrackingStoreDb

3.4 Інтерфейс та дії користувача

Всі види користувача наведено у додатку В. Мета цього розділу показати реалізацію декількох частин інтерфейсу та дій користувача.

3.4.1 Симуляція вкладеності компонентів

Демонстрацію розкриття наведено у додатку Б.

Для досягнення ефекту розкривання елемента списку використовуються два класи: ExpandableList та Expandable. Їх розміщення відносно один одного можно описати таким псевдокодом:

```
ExpandableList(
  children: [
    ...
    Expandable(
      header: SomeHeader(),
      body: SomeBody(),
    ),
    ...
  ]
)
```

Лістинг 3.22 - розміщення Expandable відносно ExpandableList

ExpandableList приймає в себе дочірні компоненти (не обов’язково типу Expandable) та розміщує їх вертикально на полотні прокрутки. Ефект розкривання досягається двома одночасними анімаціями. Перша - це прокрутка полотна до такої позиції, щоб верхня точка компонента Expandable, який розкривається, опинилася у верхній точці області полотна прокрутки, яку бачить користувач. Друга анімація - це збільшення розміру висоти компонента, який передано як параметр body у Expandable.

Через те, що ExpandableList знаходиться вище Expandable у дереві, Expandable може знайти ExpandableList та таким чином комунікувати. Процес розкривання починається з Expandable викликом його методу toggle(). Всередині toggle() повідомить ExpandableList про розкриття, а потім запустить анімацію збільшення висоти компонента, переданого як body.

```
void toggle() async {
  final wasExpanded = _isExpanded;
  ...
  _expandableListState.onToggle(toggledExpandable: this);
  if (wasExpanded) {
    await _animationController.reverse();
  } else {
    await _animationController.forward();
  }
  ...
}
```

Лістинг 3.23 - метод розкривання Expandable

ExpandableList дізнається позицію Expandable відносно видимої області екрану, порахує величину зміщення прокрутки, та запустить анімацію прокрутки на обраховане зміщення (offset):

```
void onToggle({
  required ExpandableState toggledExpandable,
}) {
  ...
  if (_isExpanded) {
    ...
    _expandedExpandable = toggledExpandable;
    final expandedHeaderBox = toggledExpandable.getBox();
    final nextOffset = _calcOffsetFor(expandedHeaderBox);
    _scrollTo(nextOffset);
  }
  ...
```

```
}
```

Лістинг 3.24 - метод розкривання ExpandableList

Наступна діаграма ілюструє взаємодію між Expandable та ExpandableList:

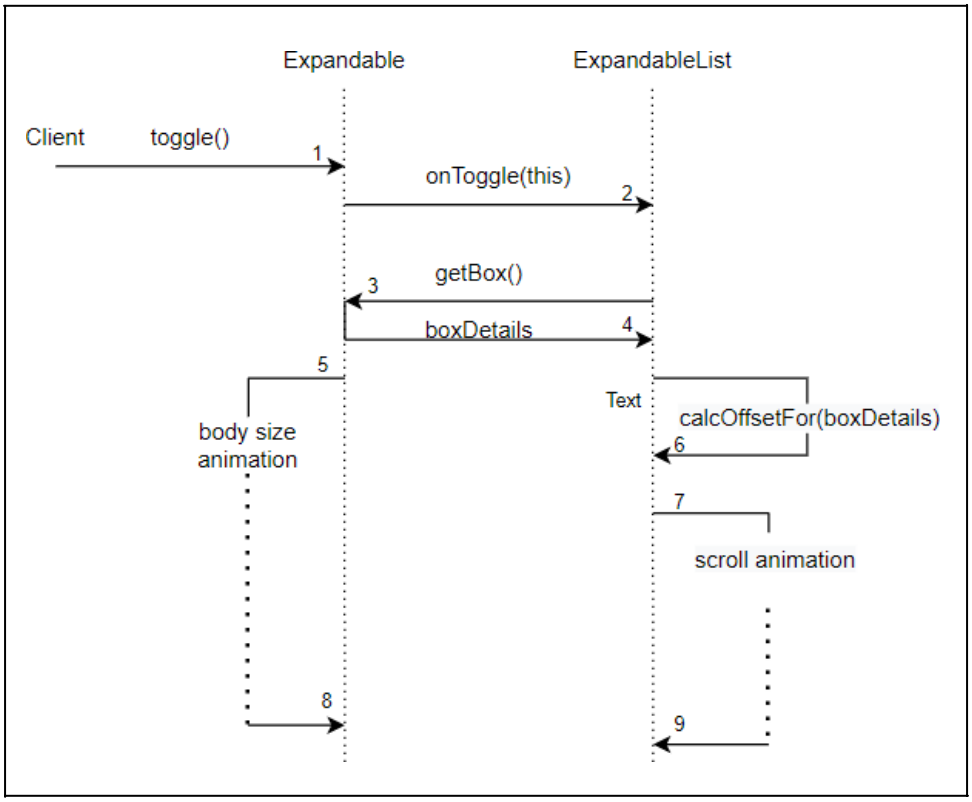

Рис. 3.2 - взаємодія Expandable та ExpandableList

Компоненти expandable є загальними та використовуються у додатку в декількох місцях: розкривання TracksetView, розкривання TrackView, розкривання відповідних компонентів TracksetSoView, TrackSoView у модулі Store.

Наведені вставки коду та діаграми є спрощеними та не ілюструють всіх деталей. Повний код реалізації expandable можна знайти по шляху lib/core/ui/components/expandable.

3.4.2 Відображення прогресу

Як було зазначено в розділі 3.1 “Модель даних”, обрахунок прогресу відбувається у класах сутностей. Методи (властивості) обрахунку потім

використовуються у компонентах інтерфейсу, щоб відобразити дані користувачу. Кожна сутність має свій відповідний інтерфейс компонент (або декілька), який приймає відповідну сутність як параметр та зображує дані про прогрес.

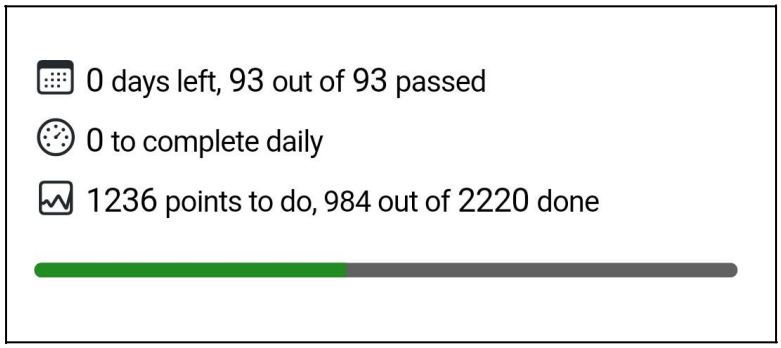

Рис. 3.3 - прогрес Trackset

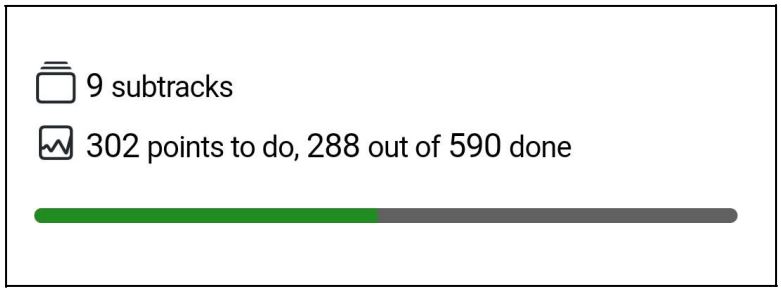

Рис. 3.4 - прогрес Track

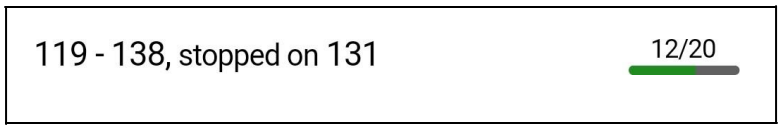

Рис. 3.5 - прогрес Subtrack

Зображення прогресу Trackset та Track мають однаковий принцип. Вони візуально відокремлені від інших елементів інтерфейсу за допомогою великих відступів (види у повному розмірі наведені у додатку В). Також візуально виділені передуючими іконками та збільшеним шрифтом чисел (безпосередньо даних прогресу). Дані про прогрес також мають прогрес бар для кращого візуального сприйняття.

3.4.3 Навігація до останнього оновленого Subtrack

Демонстрацію навігації наведено у додатку Б.

Як було зазначено у розділі 2.2 “Дії користувача”, оновлення прогресу є часто виконуваною дією, тому користувач повинен мати можливість легко здійснювати цю операцію. Прогрес оновлюється на рівні Subtrack - змінюючи значення поля pointer. При звичайна навігації до Subtrack потрібно знайти та відкрити трексет, потім знайти та відкрити трек, знайти на відкрити сабтрек, оновити прогрес. При швидкому доступі користувач має натиснути спеціально відведений для цього елемент, та додаток знайде останній оновлений сабтрек та відкриє його:

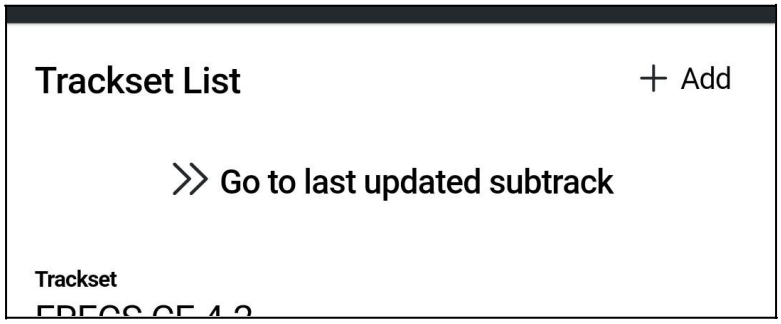

Рис. 3.6 - кнопка для навігації до останнього оновленого Subtrack

В результаті буде відкрито по черзі трексет, трек, на діалогове вікно оновлення прогресу, де користувач обирає наступне значення pointer:

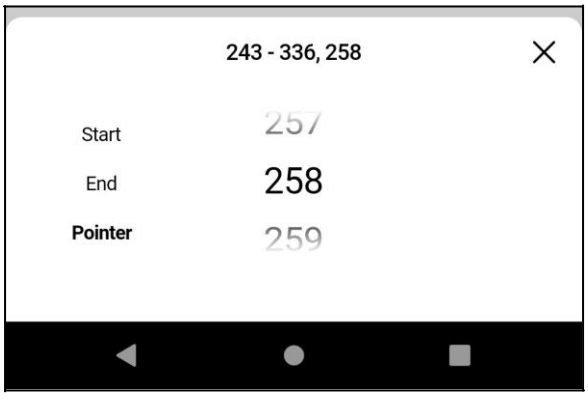

Рис. 3.7 - діалогове вікно оновлення прогресу

Діалогове вікно відкривається не на весь розмір екрану, скріншот у повному розмірі наведений у додатку В (рис. В.11).

Компонент кнопки - OpenLastUpdatedSubtrackWidget - знаходиться у компоненті TracksetList. При натисненні компонент викликає колбек onTap:

```
OpenLastUpdatedSubtrackWidget(
  onTap: _openToLastUpdatedSubtrack,
),
```

Лістинг 3.25 - використання OpenLastUpdatedSubtrackWidget у TracksetList

Обробник _openToLastUpdatedSubtrack перевіряє чи можливо зробити навігацію та додає запит (івент) в блок:

```
void _openToLastUpdatedSubtrack() {
  if (_listSelector.selectionModeEnabled ||
      listKey.currentState?.isExpanded == true) {
    return;
  }
  TrackingBloc.of(context).add(
    LastUpdatedSubtrackOpened(),
  );
}
```

Лістинг 3.26 - обробка натискання на OpenLastUpdatedSubtrackWidget

У блоці обробник запита шукає останній оновлений сабтрек (_findLastUpdatedSubtrack) та оновлює поле стейту lastUpdatedSubtrackPath, яке містить у собі шлях до сабтреку - tracksetId, trackId, subtrackId. Якщо сабтрек було знайдено, то блок послідовно публікує стейти TrackingUpdatedState з інтервалом 900 мілісекунд. Інтервал потрібен для того, щоб відкривання компонентів було послідовним та не занадто різким для коритувача.

```
Stream<TrackingState> _mapLastUpdatedSubtrackOpenedToState(event) async* {
  yield* _findLastUpdatedSubtrack();
  if (state.lastUpdatedSubtrackPath != null) {
    const interval = Duration(milliseconds: 900);
```

```
    final path = state.lastUpdatedSubtrackPath!;
    yield TrackingUpdatedState(state, tracksetIdToBeOpened: path.tracksetId);
    await Future.delayed(interval);
    yield TrackingUpdatedState(state, trackIdToBeOpened: path.trackId);
    await Future.delayed(interval);
    yield TrackingUpdatedState(state, subtrackIdToBeOpened: path.subtrackId);
  }
}
```

Лістинг 3.27 - керування навігацією до останнього оновленого Subtrack

Компоненти, що відповідають за ініціацію розкривання, мають у собі BlocListener, за допомогою якого підписани на зміни стейту та можуть ініціювати розкривання. Розкривання трексетів починається в компоненті TracksetHeader. Якщо значення поля tracksetIdToBeOpened не є null та id трексета, до якого відноситься компонент TracksetHeader, дорівнює id трексета, який потрібно розкрити, знайти Expandable да почати розкривання:

```
BlocListener<TrackingBloc, TrackingState>(
  listener: (context, state) {
    if (state is TrackingUpdatedState &&
        state.tracksetIdToBeOpened != null) {
      if (trackset.id == state.tracksetIdToBeOpened) {
        ExpandableState.of(context)!.toggle();
      }
    }
  },
  child: ...
)
```

Лістинг 3.28 - BlocListener, що слухає та ініціює розкривання Trackset

За таким самим принципом ініціюється розкривання відповідного трека та сабтрека.

3.4.4 Додавання та редагування сутностей

Додавання та редагування сутностей відбувається за наступною схемою:

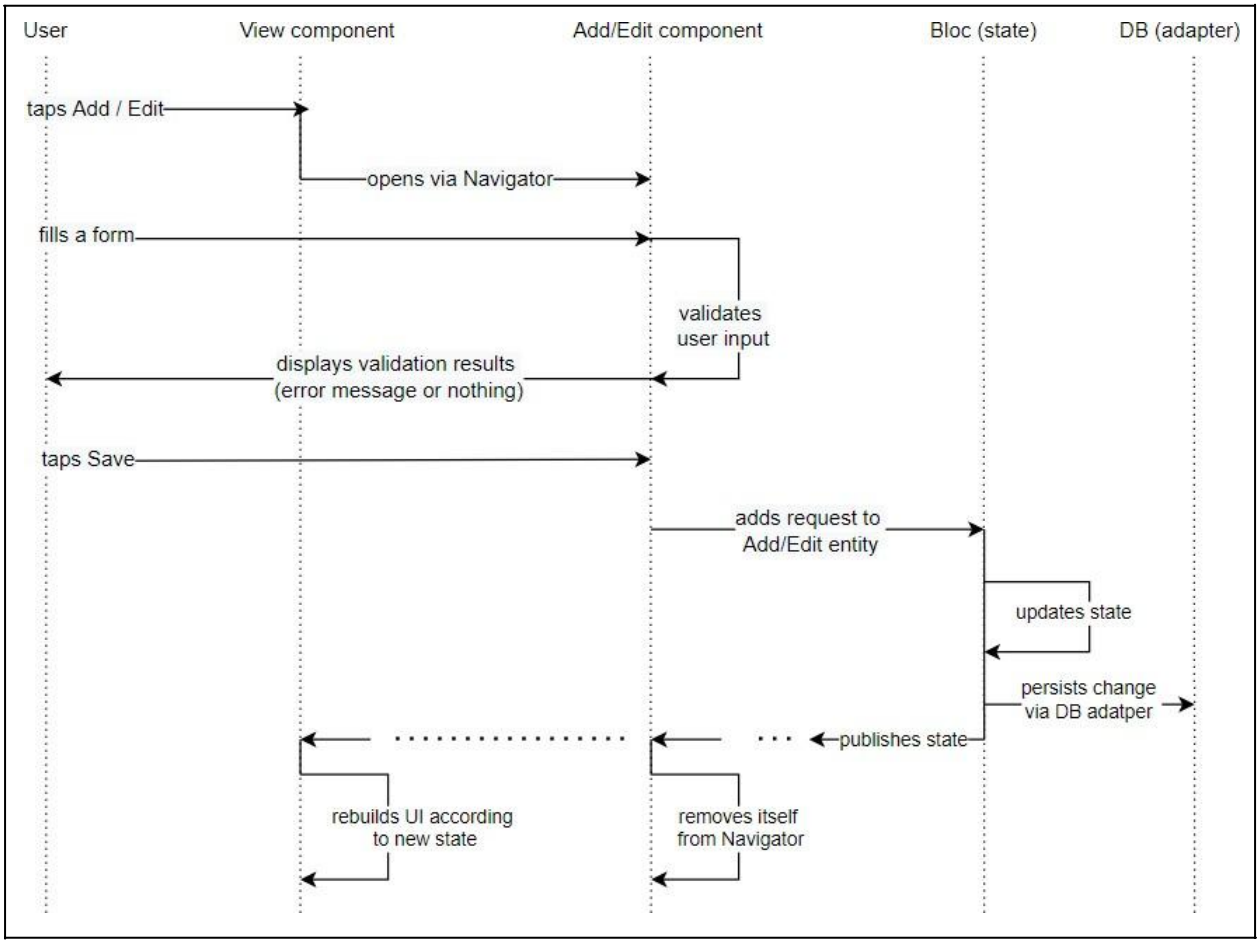

Рис. 3.8 - процес додавання та редагування сутностей

Користувач відкриває певний Add/Edit component з View component та заповнює форму. Якщо валідація пройшла успішно і помилок вводу немає, користувачеві дозволено натиснути Save, при цьому у компонент стейту буде відправлено запит, обробка якого оновить стейт, збереже зміну в БД та опублікує оновлений стейт. Add/Edit component повинен слухати зміни стейту та при певних умовах стейту видалити себе зі стеку навігації, тобто закрити, і користувач буде повернений на попередній вид. Форми додавання та редагування наведені у додатку В.

3.4.5 Видалення сутностей

Було реалізовано видалення сутностей методом вибору зі списку (рис. В.4 з додатку В). Мод виділення (selection mode) активується довгим натисканням одного з елементів списку. Після цього користувач натисканням обирає елементи для видалення. Процес видалення є схожим на процес додавання та редагування, зображений на рис. 3.8, але при видалення замість відкривання ще одного компонента Add/Edit, компонент View змінює свій інтерфейс (переключається у selection mode), взаємодіє з компонентами елементів списку, відслідковує обрані елементи, та додає запит на видалення у компонент стейту.

3.4.6 Додавання Trackset із списку для студентів

Демонстрацію додавання наведено у додатку Б.

За заготовлені трексети для студентів відповідає модуль Store. Користувач переходить до нього за допомогою меню, натискаючи на ‘For Students’:

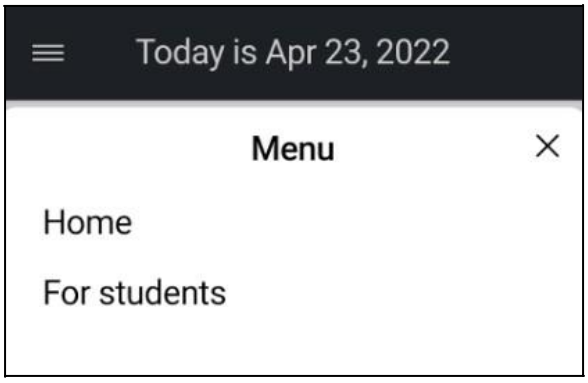

Рис. 3.9 - меню додатку

За допомогою Navigator поверх роуту Home відкривається роут Store. При створенні компонента TracksetStore, у компонент стейта (блок) TrackingStoreBloc додається запит на під’єднання до бази даних:

```
@override
void initState() {
  ...
```

```
  final bloc = TrackingStoreBloc.of(context);
  if (!bloc.state.isInitialized) {
    bloc.add(
      TrackingStoreInitialized(isWithFetch: true),
    );
  }
}
```

Лістинг 3.29 - запит на ініціалізацію компонента БД TrackingStoreDb

З’єднання з цією БД відбувається тільки при переході у модуль Store тому, щоб залишити можливість користуватися модулем Tracking без доступу до мережі інтернет. Тобто для того, щоб помилка з’єднання за БД не впливала на весь додаток.

Блок ініціалізує БД та робить ще один запит для завантаження даних про трексети для студентів (add(TrackingStoreFetched())):

```
Stream<TrackingStoreState> _mapTrackingStoreInitializedToState(
  TrackingStoreInitialized event,
) async* {
  try {
    yield TrackingStoreLoadingState(state);
    await _db.init();
    final nextState = state.copyWith(isInitialized: true);
    if (event.isWithFetch) {
      yield TrackingStoreLoadingState(nextState);
      add(TrackingStoreFetched());
    } else {
      yield TrackingStoreUpdatedState(nextState);
    }
  } catch (ex) {
    yield TrackingStoreErrorState(
      state,
      description: 'Failed to fetch data',
      failedEvent: event,
    );
  }
```

```
}
```

Лістинг 3.30 - обробка запиту на ініціалізацію TrackingStoreDb

Після завантаження даних користувач може переглянути трексети та їх треки і сабтреки (рисунки повного розміру наведені у додатку В):

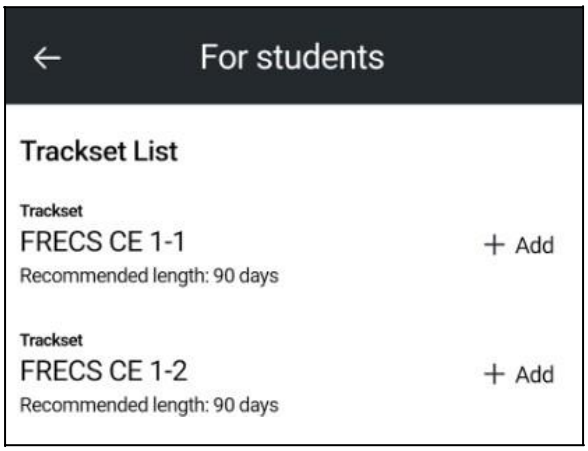

Рис. 3.10 - трексети для студентів

При натисканні на Add випливає діалогове вікно, в якому користувач повинен обрати часовий проміжок, з яким обраний трексет буде збережено до списку трексетів користувача (рис. В.14 додатку В). Після додавання користувач має можливість робити такі ж дії, якби він створив трексет самостійно (рис. В.3 додатку В). Модуль Store має залежність на компоненті стейту модуля Tracking - для додавання запиту на збереження обраного трексета:

```
void _onSubmit() {
  _validateForm();
  if (_isFormValid) {
    _formKey.currentState!.save();
    final bloc = TrackingBloc.of(context);
    bloc.add(
      TracksetSoAdded(
        tracksetSo: widget.trackset,
        dateRange: _value,
      ),
    );
  }
```

```
}
```

Лістинг 3.31 - обробка натискання Save

Метод-обробник запита TracksetSoAdded зберігає доданий трексет та його дочірні сутності в базу даних модуля Tracking (локальну) та публікує новий стейт з доданим трексетом.

3.4.7 Оптимізація expandable

Підхід до навігації з розкриванням, запропонований у розділі 2.4 “Макет інтерфейсу”, та реалізований у розділі 3.4.1 “Симуляція вкладеності компонентів”, виявився занадто дорогим для плавної анімації у 60 fps (frames per second).

Як зазначалося раніше, Expandable змінює висоту компонента, переданого в нього як параметр body. Тому, коли Expandable не є розкритим, висота body = 0 і компонент не видно, проте він присутній в дереві віджетів та Flutter при побудові (яка відбувається кожен фрейм (кадр) при анімації) інтерфейсу витрачає ресурси на цей компонент. Візьмемо ієрархію основних сутностей (рис 2.2). Кожна сутність в ієрархії умовно має відповідний їй компонент інтерфейсу. Таким чином, якщо користувач має N трексетів, по M треків в кожному, та по P сабтреків в кожному треку, то маємо N*M*P компонентів, на які потрібно витрачати ресурси, хоча при розкриванні користувач може бачити максимально N + M + P компонентів - всі трексети, треки обраного трексета, та сабтреки обраного трека.

Якщо взяти N = M = P = 10, то без оптимізації кількість умовних компонентів дорівнює 1000, хоча потрібно лише 30.

Для оптимізації expandable було обернено компонент body у компонент Offstage, яким можна керувати, чи буде потрібно малювати дочірній компонент (та цим витрачати ресурси девайса). Якщо Expandable не є

розкритим (_isExpanded == false), то offstage == true, та компонент widget.body не займає ресурси:

```
Widget buildBody(BuildContext context) {
  return Flexible(
    child: SizeTransition(
      sizeFactor: CurvedAnimation(
        parent: _animationController,
        curve: widget.animationData.curve,
      ),
      child: Offstage(
        offstage: !_isExpanded,
        child: widget.body,
      ),
    ),
  );
}
```

Рис. 3.11 - зміна коду для оптимізації Expandable

Через додавання Offstage, перед запуском анімації потрібно робити додатковий білд (будування) Expandable, щоб значення offstage змінилось на true, та body було видно при розкриванні. Тому код метода toggle() має наступне: якщо це розкривання, а не закривання, тоді змінити значення _isExpanded на протилежне та методом setState спровокувати перебудову компонента, щоб внутрішні компоненти отримали оновлене значення _isExpanded та змінили свій стейт відповідно до нового значення.

```
final wasExpanded = _isExpanded;
if (!wasExpanded) {
  setState(() {
    _isExpanded = !_isExpanded;
  });
}
```

```
_expandableListState.onToggle(toggledExpandable: this);
...
```

Лістинг 3.32 - доданий код для тригеру перебудови по зміні _isExpanded

Так само, при закриванні Expandable, треба знову зробити offstage = true, але вже після того, як пройде анімація закривання.

Для перевірки результату оптимізації було використано інструмент flutter driver для автоматизації сценарію виконання та профілювання.

Сценарій виконання полягав у тому, щоб розкривати компонент трексета. Для цього було реалізовано спрощений варіант додатку, мета якого була виключити або замінити компоненти, які не впилвають на тестування, щоб спростити процес тестування. Таким чином, було взято компонент TracksetList, та над ним поставлено компоненти, які необхідні для запуску додатку та фукнціонування самого TracksetList:

```
CupertinoApp(
  home: BlocProvider(
    create: (context) {
      return TrackingBloc(db: _MockTrackingDb());
    },
    child: const SafeArea(
      child: CupertinoPageScaffold(
        child: TracksetList(),
      ),
    ),
  ),
);
```

Лістинг 3.33 - інфраструктура для профілювання expandable

TracksetList та компоненти всередині нього використовують TrackingBloc, щоб на основі його стейту будувати інтерфейс, проте компонент адаптера бази даних було підмінено іншим - _MockTrackingDb, який повертає дані з пам’яті, які були заготовлені спеціально для тестування:

```
class _MockTrackingDb extends BaseMockTrackingDb {
  final _tracksets = buildTrackingData();

  @override
  Future<NormalizedList<Trackset, String>> getEnrichedTracksets() {
    return Future.value(_tracksets);
  }
}
```

Лістинг 3.34 - mock клас для підміни бази даних

Для тестування було заготовлено 10 трексетів, по 10 треків в кожному, по 10 сабтреків в кожному треку:

```
NormalizedList<Trackset, String> buildTrackingData() {
  final tracksets = TracksetFactory.buildTracksets(
    count: 10,
    trackCount: 10,
    subtrackCount: 10,
  );

  return normalizeTracksets(tracksets);
}
```

Лістинг 3.35 - тестові дані для профілювання expandable

Одна ітерація тестування натисканням на компонент списку трексетів розкриває перший трексет 10 разів та замірює перфоманс операції:

```
final tracksetHeader = find.byType(TracksetHeader).first;
for (var i = 0; i < 10; i++) {
  await binding.traceAction(
    () async {
      await tester.tap(tracksetHeader);
      await tester.pumpAndSettle();
    },
  );
  // collapse expandable  back
  await tester.tap(tracksetHeader);
  await tester.pumpAndSettle();
```

```
}
```

Лістинг 3.36 - сценарій профілювання expandable

Було проведено 10 таких ітерацій до оптимізації та 10 ітерацій після оптимізації. У порівнянні приймали участь наступні показники: середня кількість кадрів в секунду (frame count), середній час створеня кадру (frame build time), середній час створення кадру растеризації (rasterizer frame build time). Те, наскільки довгим є час двох останніх показників, впливає на перший показник.

Маємо наступні результати тестування:

	Frame count	Frame build time (ms)	Rasterizer frame build time (ms)
Before (avg)	14	55.7	16.2
After (avg)	68.3	7.7	10.1
Optimized diff (percents)	387%	-86%	-37%

Табл. 3.1 - результати оптимізації Expandable

Кількість кадрів в секунду зросло з 14 до 68.3 (+387%), час створення кадра зменишвся з 55.7ms до 7.7ms (-86%), час створення кадра растеризації зменшився з 16.2ms до 10.1ms (-37%).

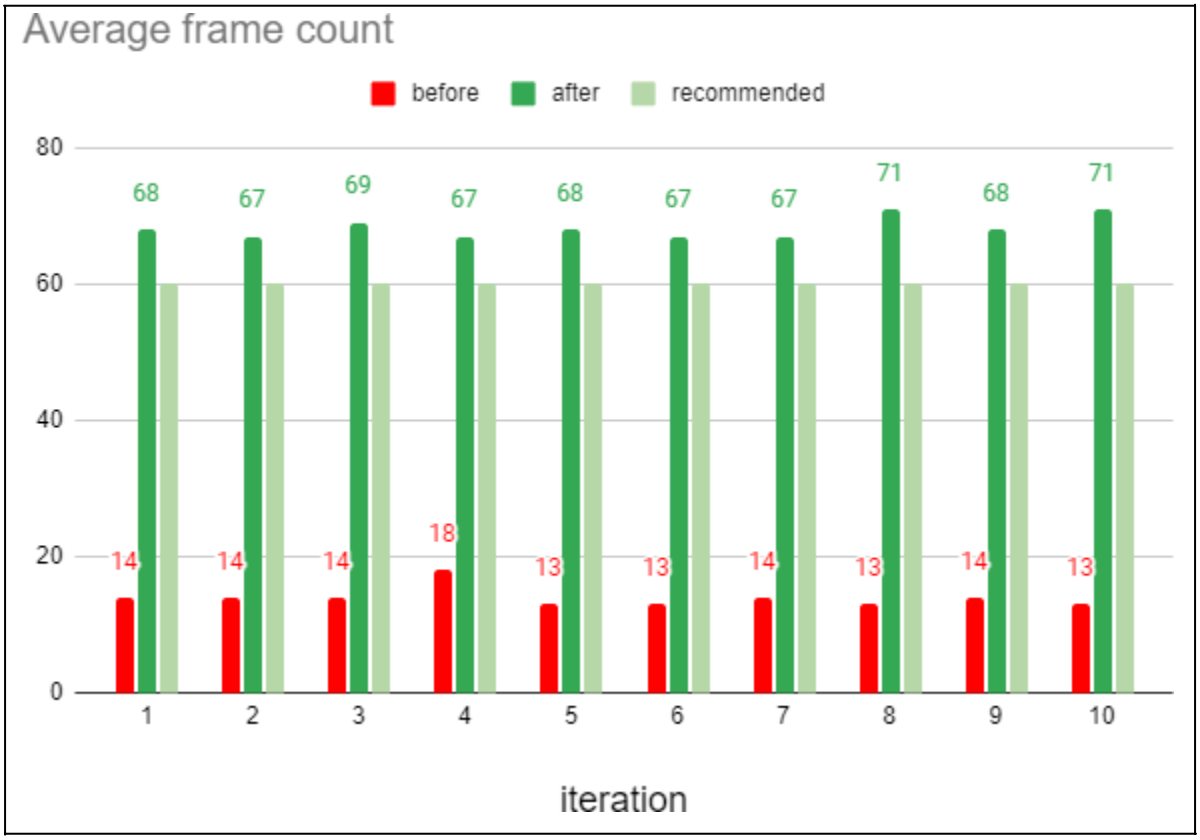

Рис. 3.12 - результати оптимізації expandable (frame count)

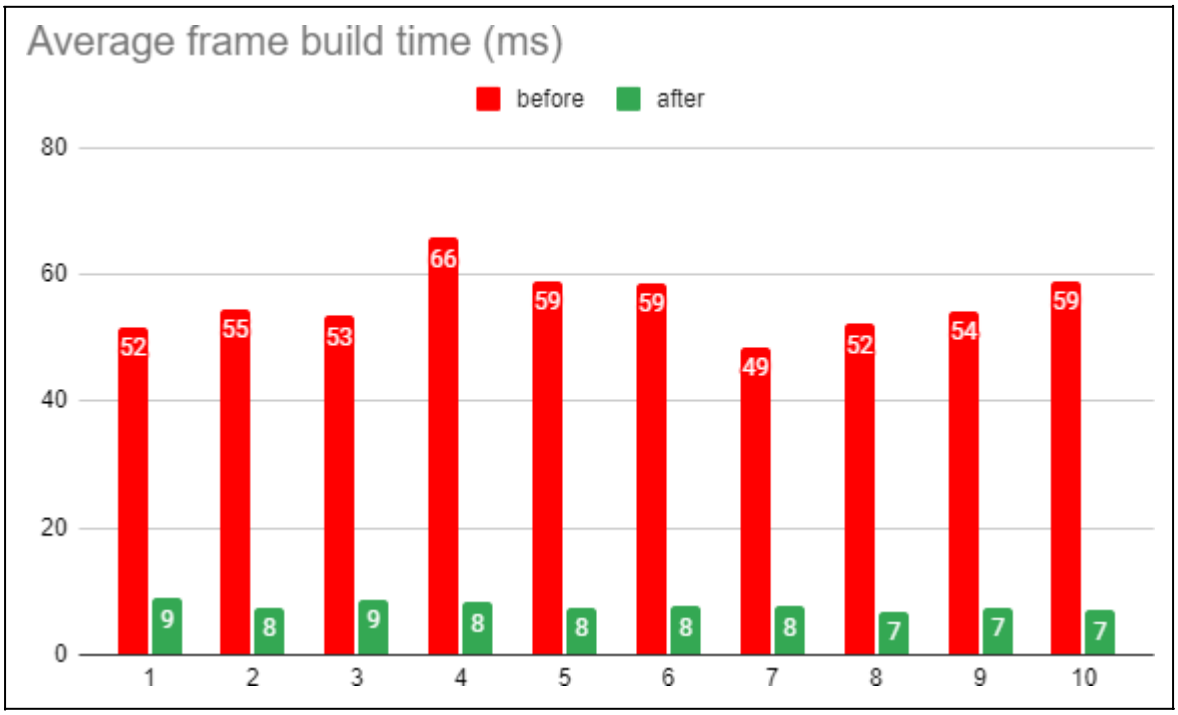

Рис. 3.13 - результати оптимізації expandable (frame build time)

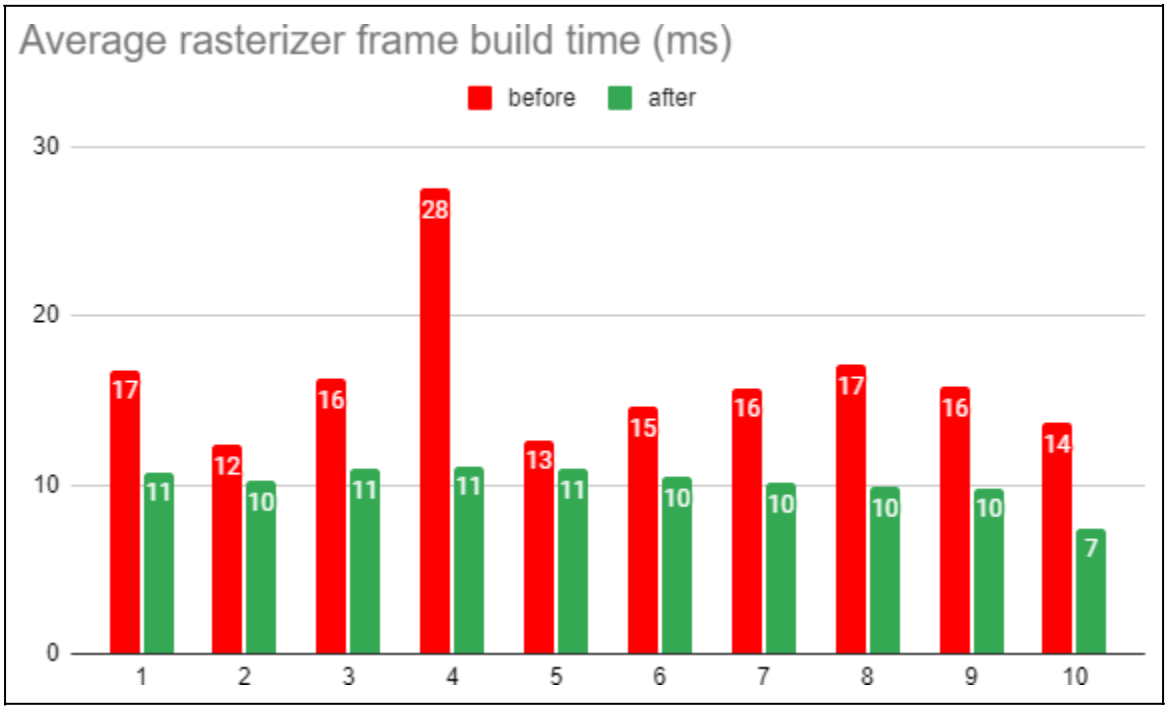

Рис. 3.14 - результати оптимізації expandable (rasterizer frame build time)

Висновки

1. Проаналізовано доступні рішення додатків для відслідковування літератури та виявлено варіанти покращення: можливість об'єднувати декілька одиниць літератури одним проміжком часу, точно вказувати діапазони сторінок у кожній одиниці літератури.
2. Розроблено архітектуру додатку без обмежень додавання сутностей тільки з наявного у додатку каталогу, інтегрованого середовища відображення тексту читання та автоматичного оновлення прогресу та без залежності від доступу до мережі інтернет.
3. Обрано мобільний девайс як цільовий пристрій на основі вимог до ергономічності та доступності додатку.
4. Додано можливість інтеграції додатку у виді допоміжного засобу у навчальних закладах, адже навчальний семестр підходить під формат даних.
5. В якості основного фреймворку обрано Flutter, який на момент аналізу виявився найбільш вдалим за критеріями крос-платформеності, популярності, перфомансу та інструментарію розробника.
6. Було візуально виділено елементи інтерфейсу, що зображують основну інформацію про прогрес, оскільки зорове сприйняттся сприяє мотивації користувача.
7. Формат даних проходження технічної літератури дозволив виділити три основні сутності (trackset, track, subtrack), які утворили три-рівневу ієрархію. На основі ієрархії було визначено, що відповідні компоненти інтерфейсу повинні слідувати формату ієрархії для кращого розуміння користувачем. Це дозволило розробити кастомний підхід до переходів між видами користувача.

8. На основі аналізу дій користувача визначено основні модулі, а саме: Tracking та Store.
9. Виявлено необхідність підтримки сценарію, коли декілька джерел ініціаціюють зміни стейту, та декілька джерел прослуховують зміни стейту. Проблему було вирішено виділенням окремого компонента стейту, який був реалізований на основі патерну BLoC з використанням бібліотеки bloc.
10. Код комунікації з БД було енкапсульовано у компоненти-адаптери, що спростило тестування компонентів інтерфейсу методом підміни компонента-адаптера.
11. Функціонал готових трексетів для студентів було виокрмлено в окремий модуль Store, для позбавлення обов’язкової залежності від доступу в інтернет.
12. Оновлення прогресу Subtrack було виявлено найчастішою дією користувача у додатку. Задля покращення UX було додано функцію швидкої навігації до останнього оновленого Subtrack, а також визначено простий для користувача спосіб оновлення прогресу - прокрутка списку чисел.
13. Було виявлено незадовільний (менше за 60) показник кадрів в секунду при переходах між видами користувача. Реалізацію було підкориговано так, щоб фреймворк не витрачав ресурси при побудові нерозкритої (невидимої) частини компонентів. Для точності результатів сценарій розкривання було автоматизовано. В результаті оптимізації показник кадрів в секунду збільшився від 14 до 68, що свідчить про адекватність FPS після оптимізації (більше за 60).

Перелік посилань

1. Objective Key Results[Електронний ресурс], Режим доступу: URL: https://en.wikipedia.org/wiki/OKR
2. Існуючий додаток MyBook[Електронний ресурс], Режим доступу: URL: https://play.google.com/store/apps/details?id=ru.mybook
3. Існуючий додаток ReadMore[Електронний ресурс], Режим доступу: URL: https://play.google.com/store/apps/details?id=com.shunan.readmore
4. Існуючий додаток Bookly[Електронний ресурс], Режим доступу: URL: https://play.google.com/store/apps/details?id=com.twodoor.bookly
5. Вплив спільноти на технологію[Електронний ресурс], Режим доступу: URL: https://www.sciencedirect.com/science/article/pii/S1319157820305139
6. Порівняння крос-платформеної та нативної розробки[Електронний ресурс], Режим доступу: URL: https://www.uptech.team/blog/native-vs-cross-platform-app-development
7. Топ мобільних фреймворків[Електронний ресурс], Режим доступу: URL: https://technostacks.com/blog/mobile-app-development-frameworks
8. Популярність Flutter[Електронний ресурс], Режим доступу: URL: https://stackoverflow.blog/2022/02/21/why-flutter-is-the-most-popular-cross-platform-mobile-sdk/
9. Порівняння показників перфомансу Flutter та React Native[Електронний ресурс], Режим доступу: URL: https://inveritasoft.com/blog/flutter-vs-react-native-vs-native-deep-performance-comparison

10. Реалізація крос-платформеності React Native[Електронний ресурс], Режим доступу: URL: https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html
11. Реалізація крос-платформеності Flutter[Електронний ресурс], Режим доступу: URL: https://docs.flutter.dev/resources/architectural-overview
12. Важливість запису цілей (1)[Електронний ресурс], Режим доступу: URL: https://www.forbes.com/sites/markmurphy/2018/04/15/neuroscience-explains-why-you-need-to-write-down-your-goals-if-you-actually-want-to-achieve-them
13. Важливість запису цілей (2)[Електронний ресурс], Режим доступу: URL: https://management30.com/blog/writing-down-goals-for-success/
14. Важливість дедлайнів[Електронний ресурс], Режим доступу: URL: https://eightysixfourhundred.com/why-are-deadlines-important/
15. Важливість бачення прогресу (1)[Електронний ресурс], Режим доступу: URL: https://lucemiconsulting.co.uk/writing-down-your-goals/
16. Вважливість бачення прогресу (2)[Електронний ресурс], Режим доступу: URL: https://www.happybrainscience.com/blog/visualize-progress/
17. Документація використання SQLite на Flutter[Електронний ресурс], Режим доступу: URL: https://docs.flutter.dev/cookbook/persistence/sqlite

18. Документація BLoC патерну[Електронний ресурс], Режим доступу: URL: https://bloclibrary.dev/#/whybloc
19. Документація використання BLoC на Flutter[Електронний ресурс], Режим доступу: URL: https://bloclibrary.dev/#/flutterbloccoreconcepts
20. Документація бази даних від Firebase[Електронний ресурс], Режим доступу: URL: https://firebase.google.com/docs/firestore
21. Документація CustomPainter інтерфейсу[Електронний ресурс], Режим доступу: URL: https://api.flutter.dev/flutter/rendering/CustomPainter-class.html
22. Профілювання перфомансу для Flutter[Електронний ресурс], Режим доступу: URL: https://docs.flutter.dev/cookbook/testing/integration/profiling

Додаток А

Посилання на GitHub репозиторій з вихідним кодом:
https://github.com/denbell5/prtmobile

Додаток Б

Демонстрація ефекту розкриття (expandable)
https://drive.google.com/file/d/1jW2eIz1UWWqE5HO76Fj5TWPqS_abFmq6/view?usp=sharing

Демонстрація додавання Trackset зі списку для студентів
https://drive.google.com/file/d/1HufxHLa1CZEcZNq6DL4vV_rbMLr2_H5b/view?usp=sharing

Демонстрація швидкої навігації до останнього оновленого Subtrack та оновлення прогресу
https://drive.google.com/file/d/1KZHLXqBUuB3os1mC6CTBJ-TuqqnJY5W-/view?usp=sharing

Додаток В

Нижче представлені всі види користувача в додатку, розділені по модулям.

Menu модуль

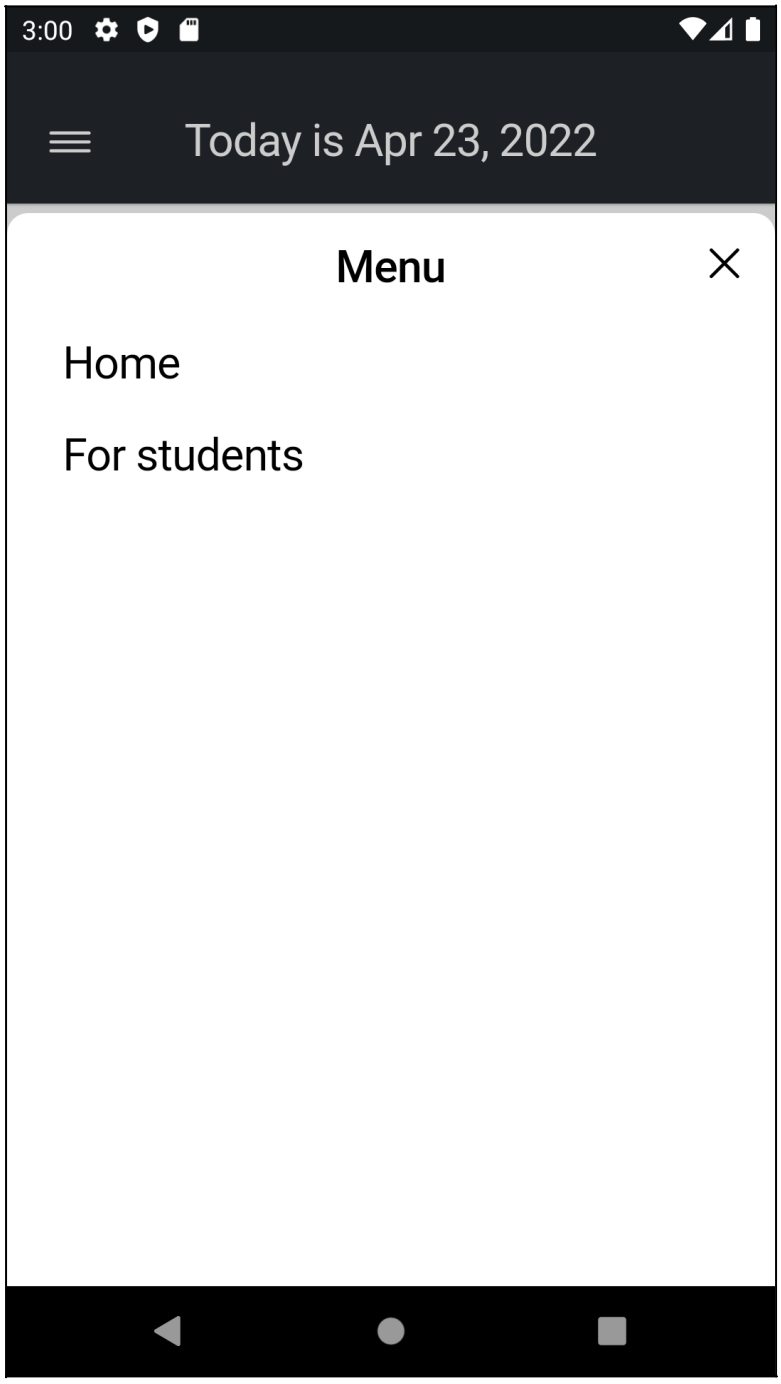

Рис. В.1 - Menu

Tracking модуль

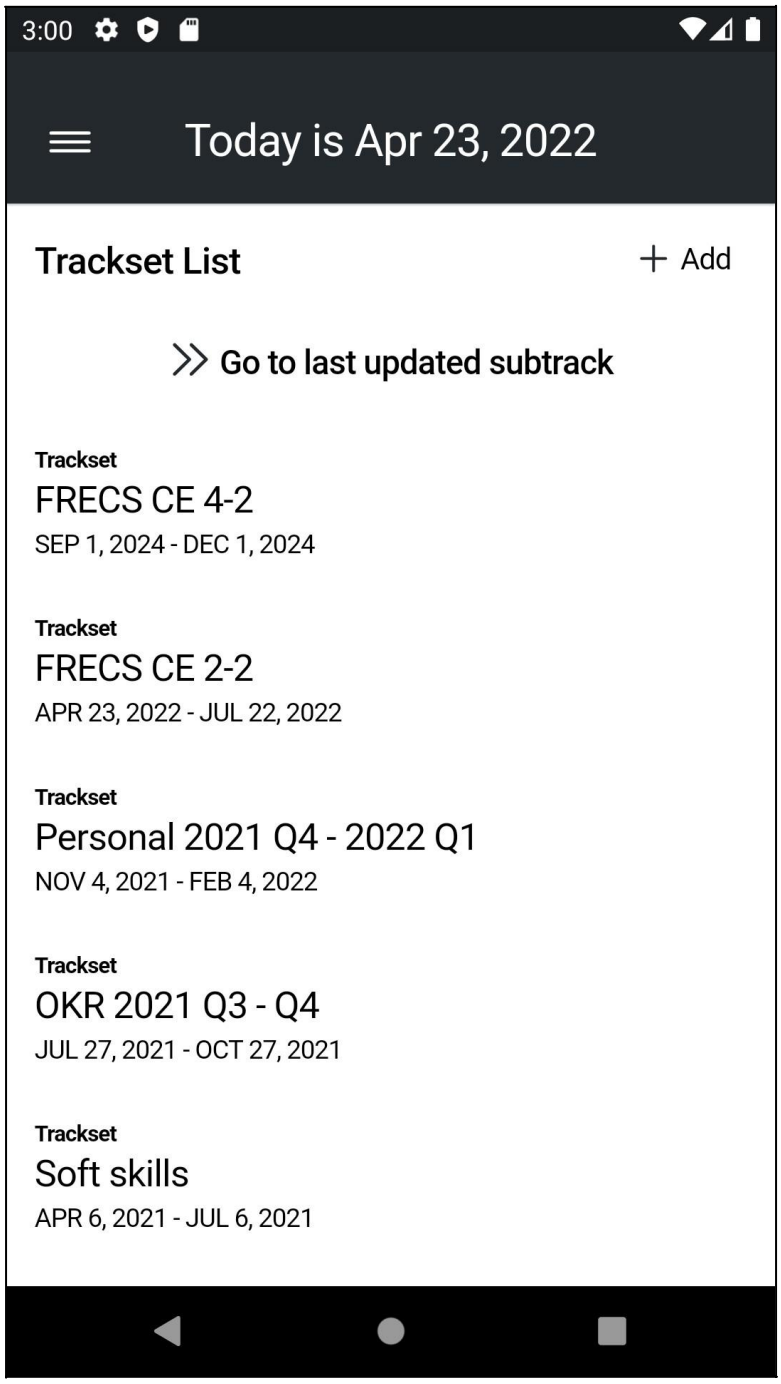

Рис. В.2 - trackset-list

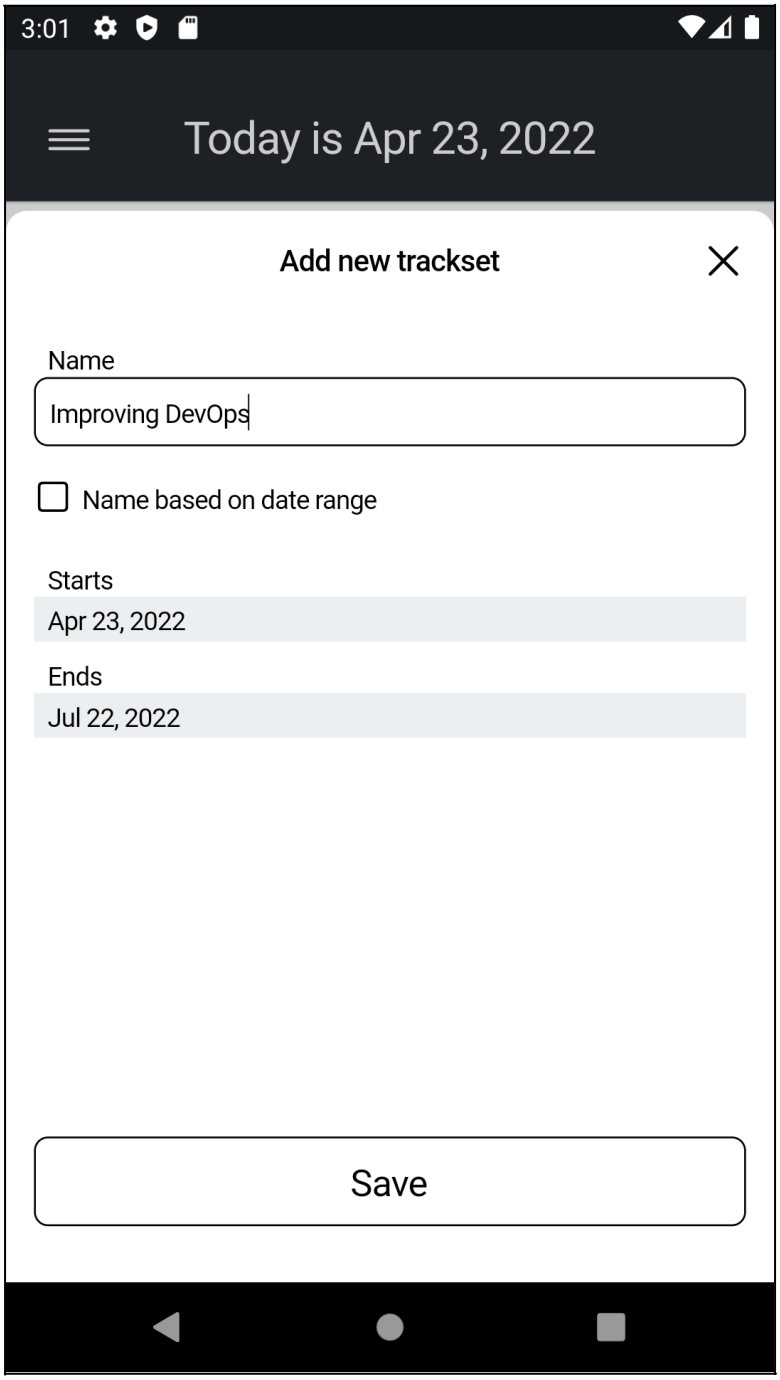

Рис. В.3 - trackset-add

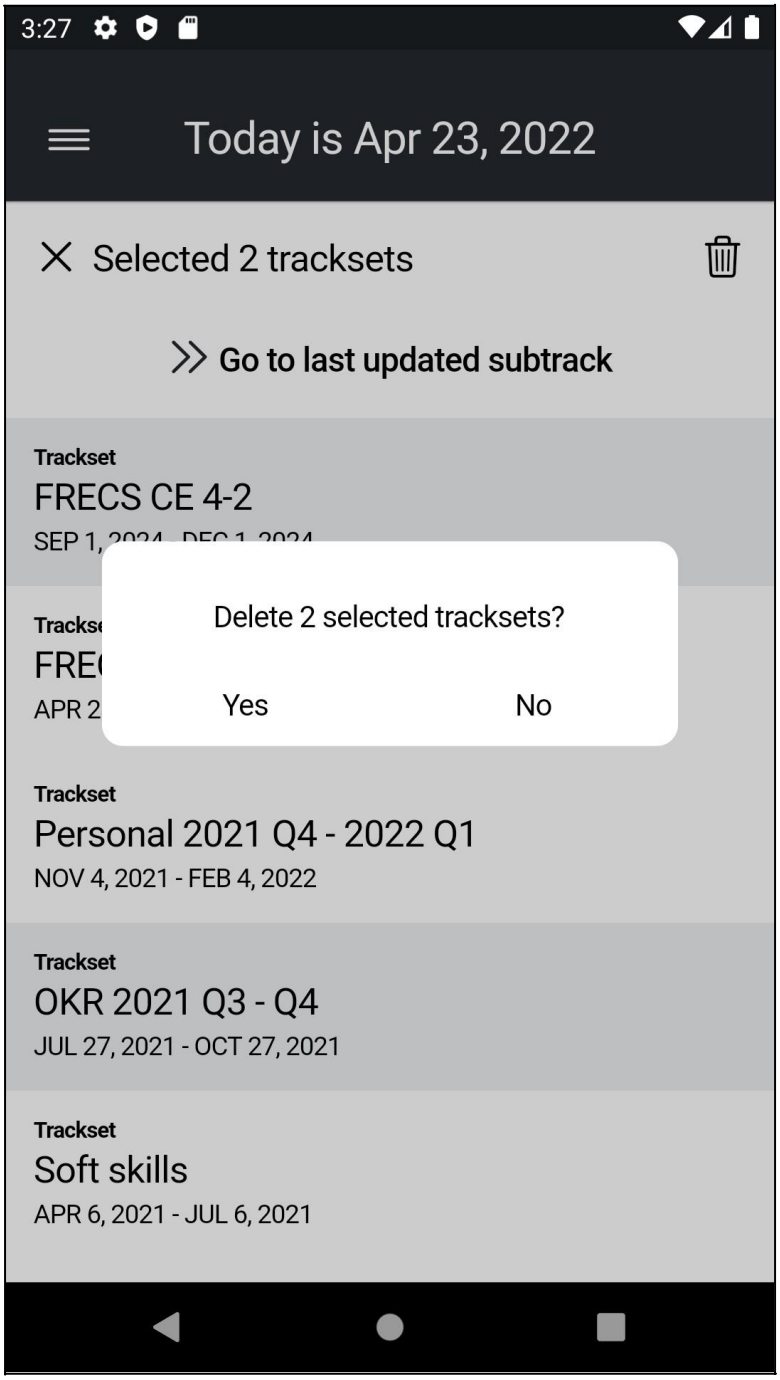

Рис. В.4 - trackset-delete-multiple

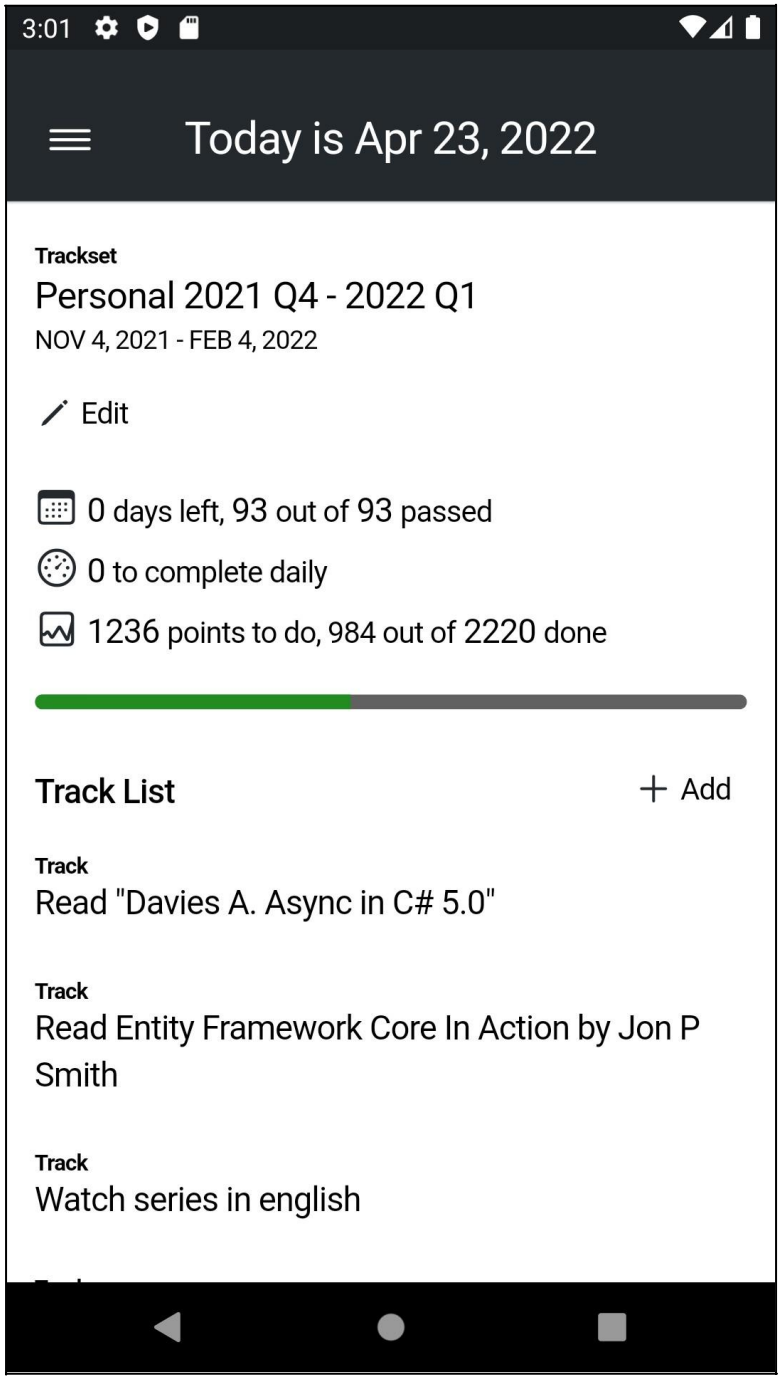

Рис. В.5 - trackset-view

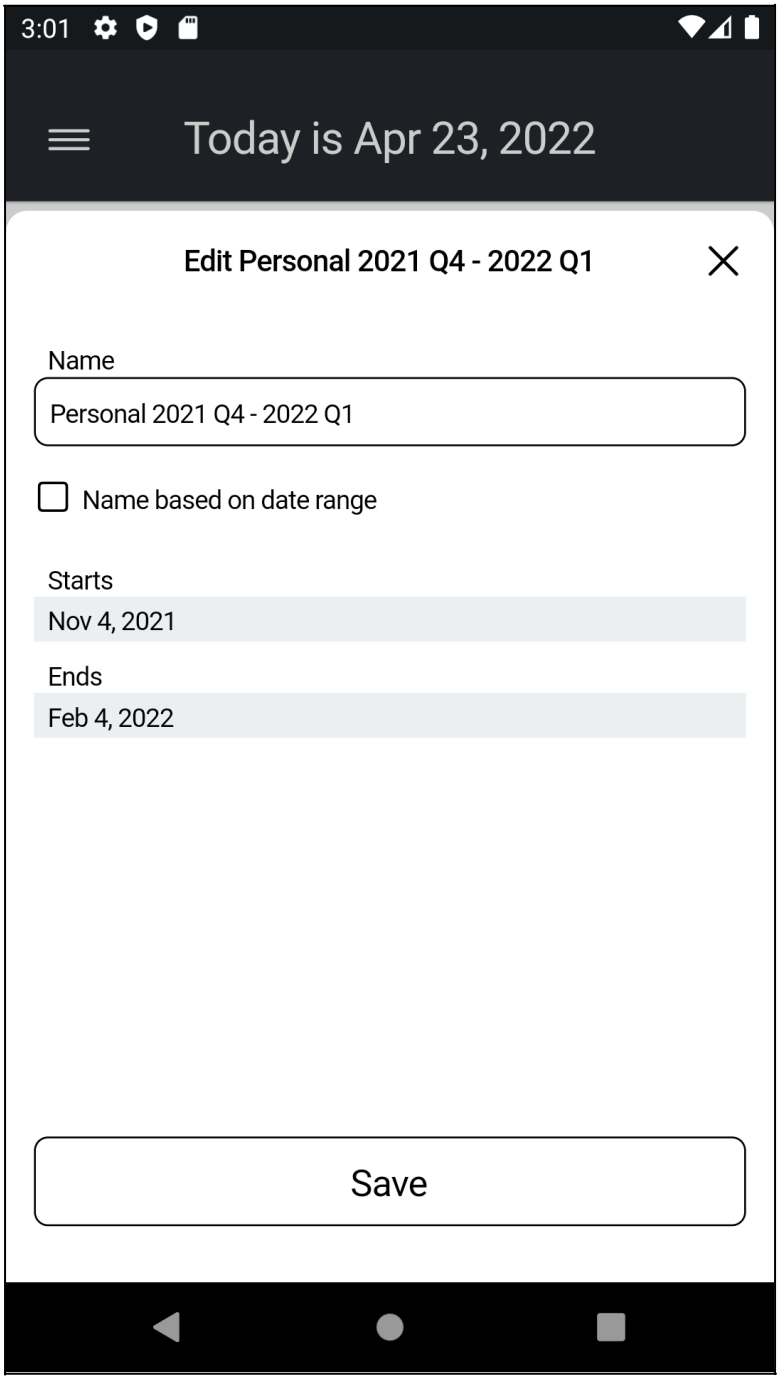

Рис. В.6 - trackset-edit

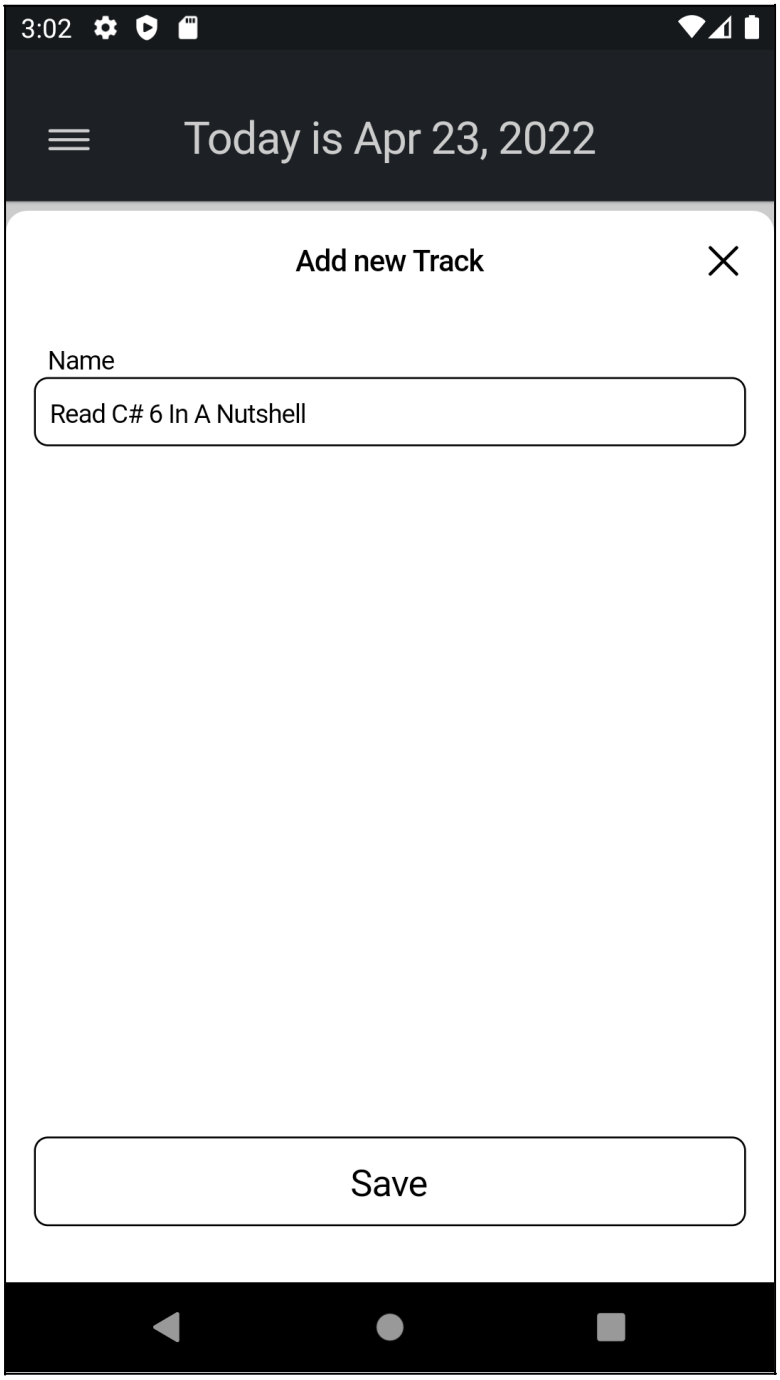

Рис. В.7 - track-add

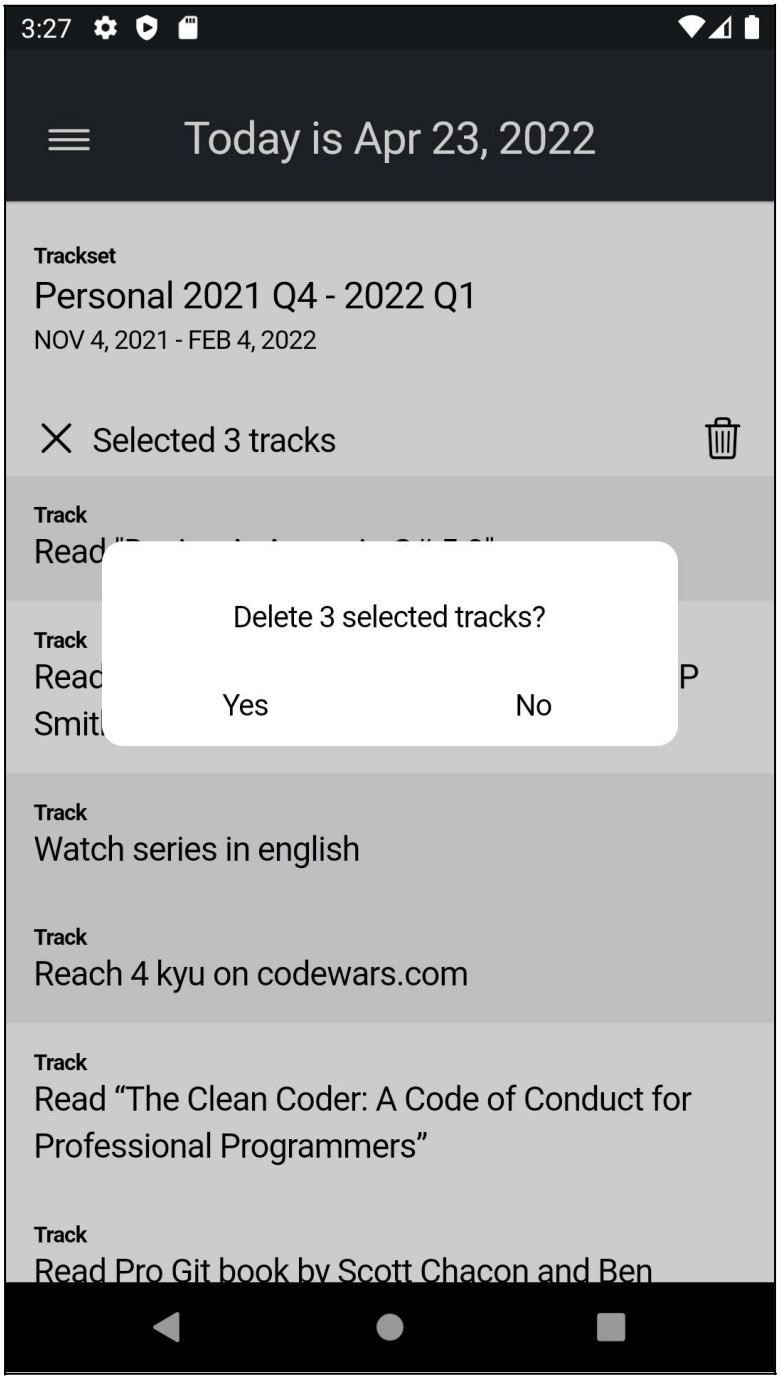

Рис. В.8 - track-delete-multiple

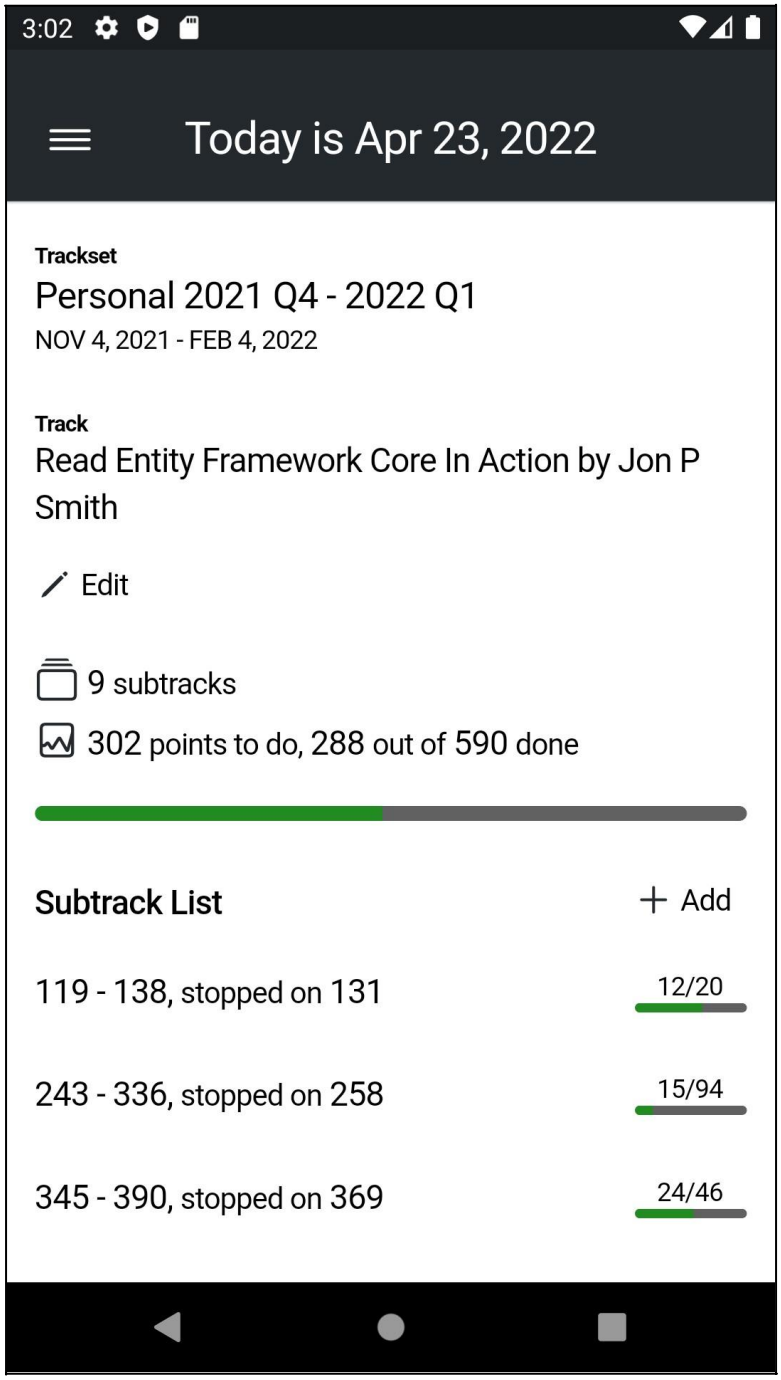

Рис. B.9 - track-view

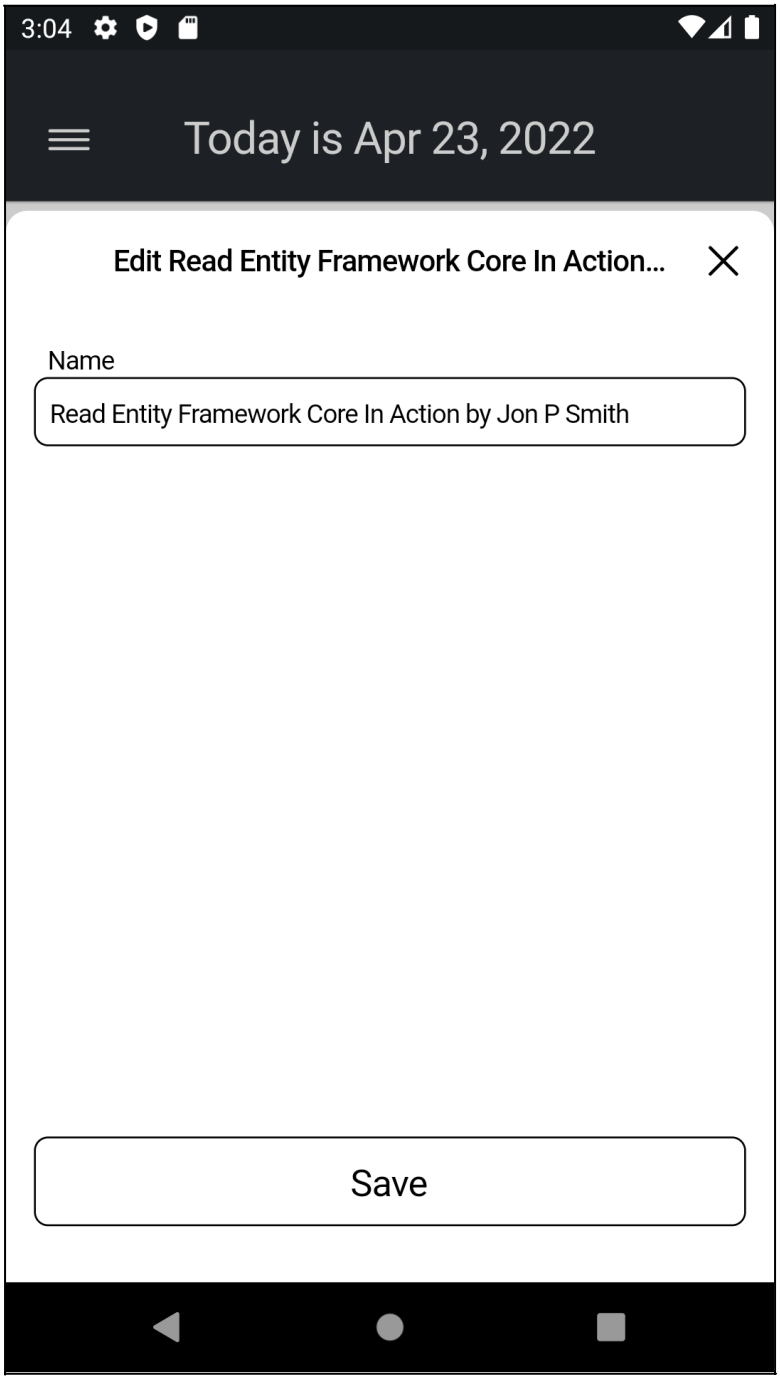

Рис. B.10 - track-edit

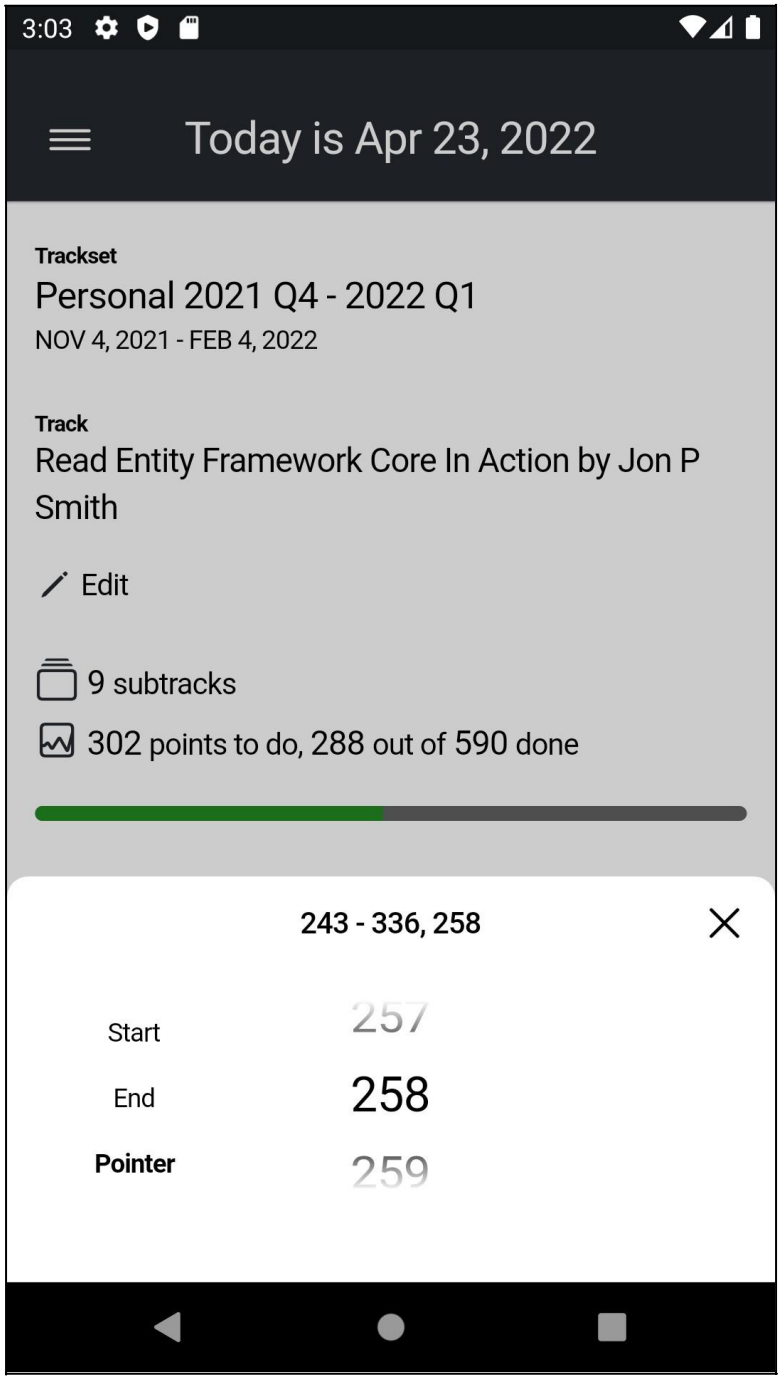

Рис. B.11 - subtrack-edit

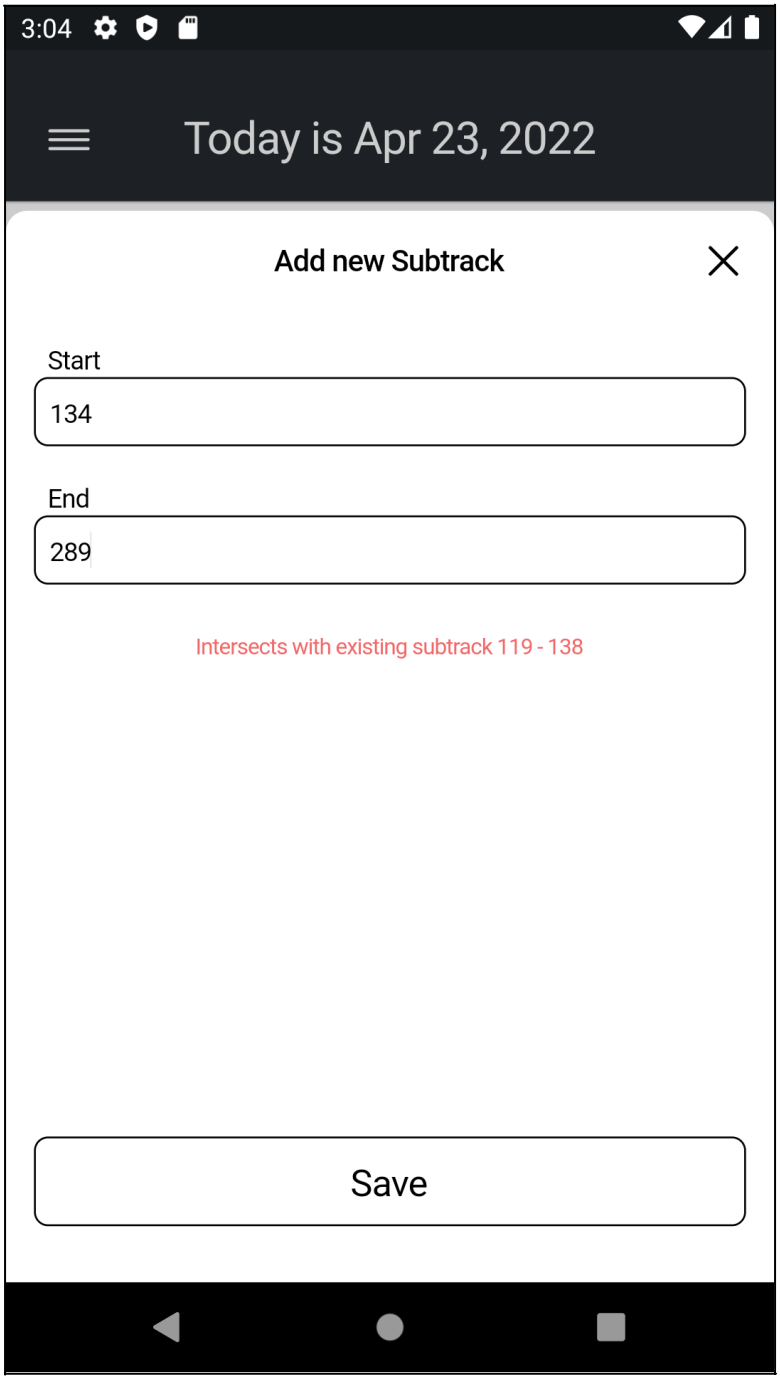

Рис. В.12 - subtrack-add

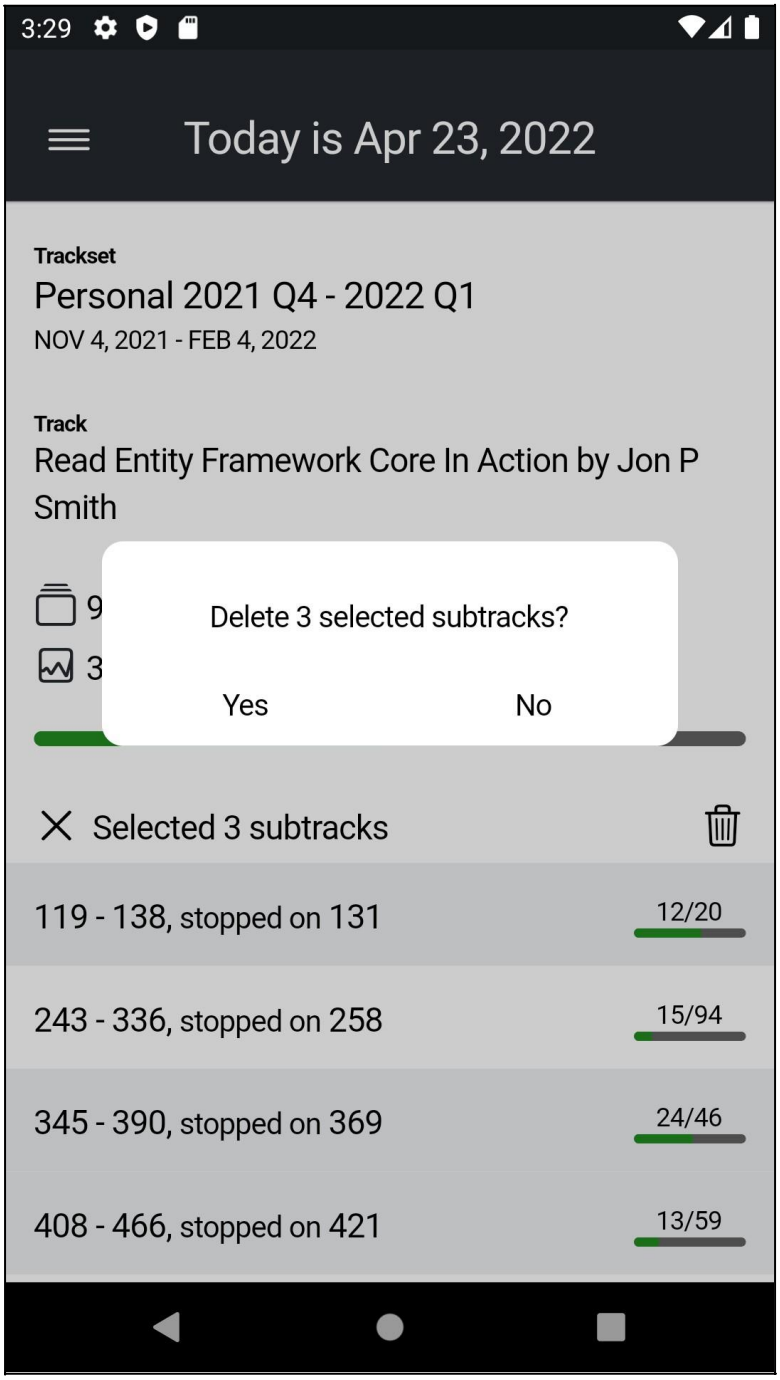

Рис. B.13 - subtrack-delete-multiple

Store модуль

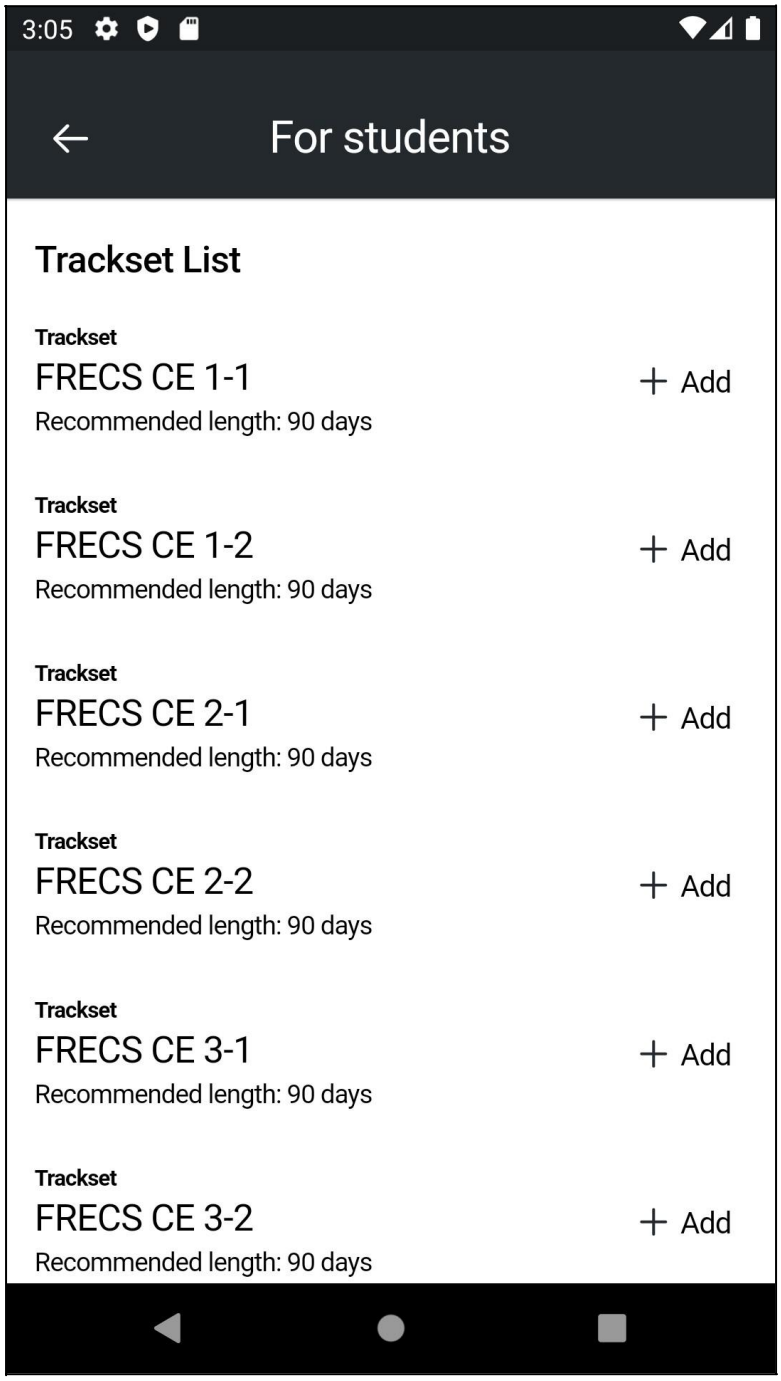

Рис. В.14 - store-trackset-list

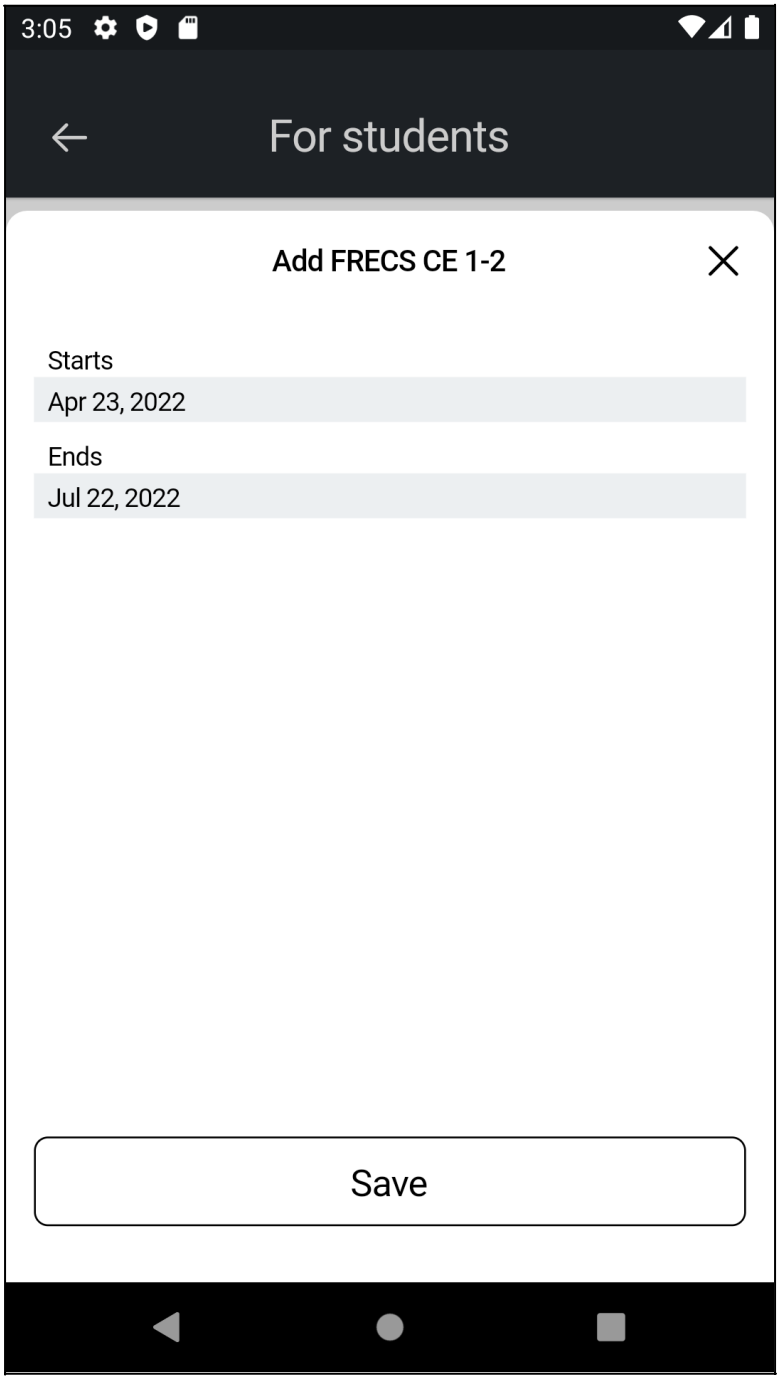

Рис. B.15 - store-trackset-add

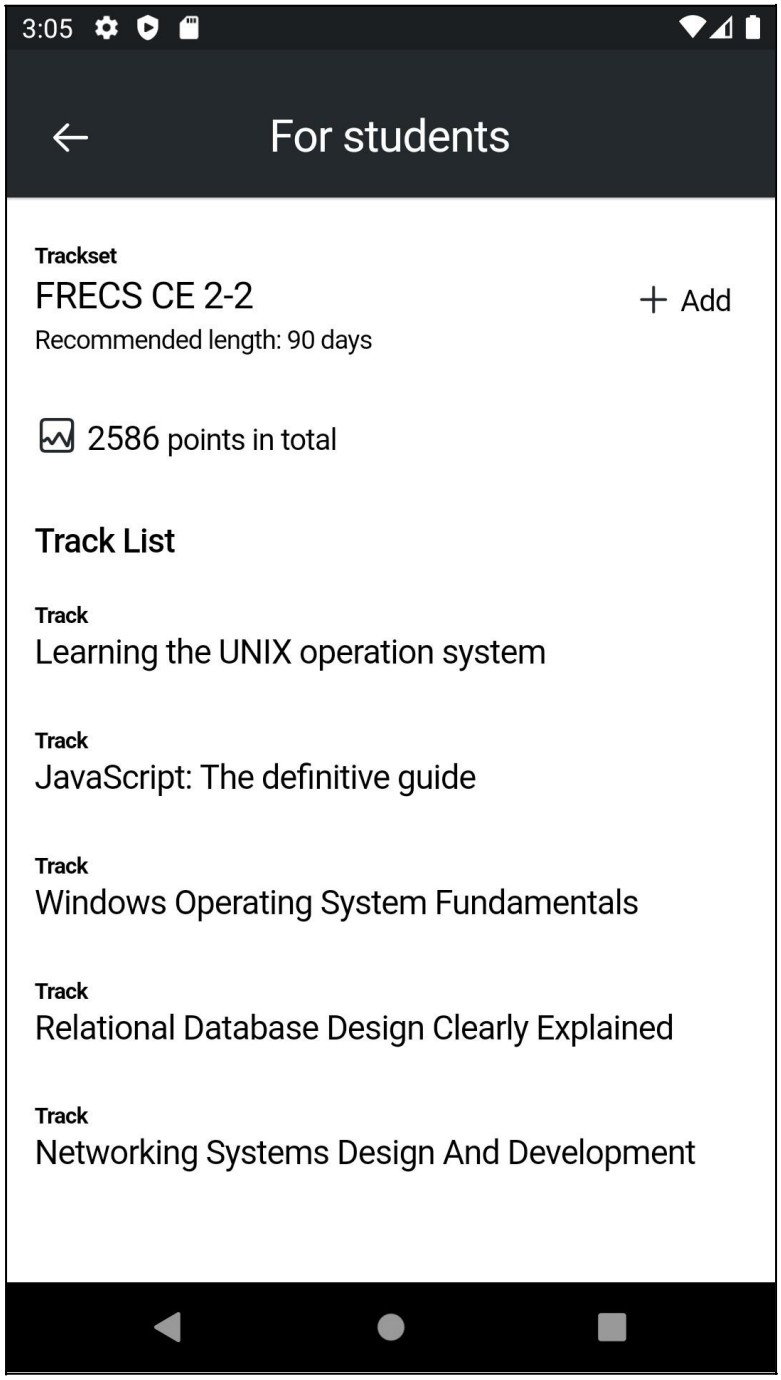

Рис. В.16 - store-trackset-view

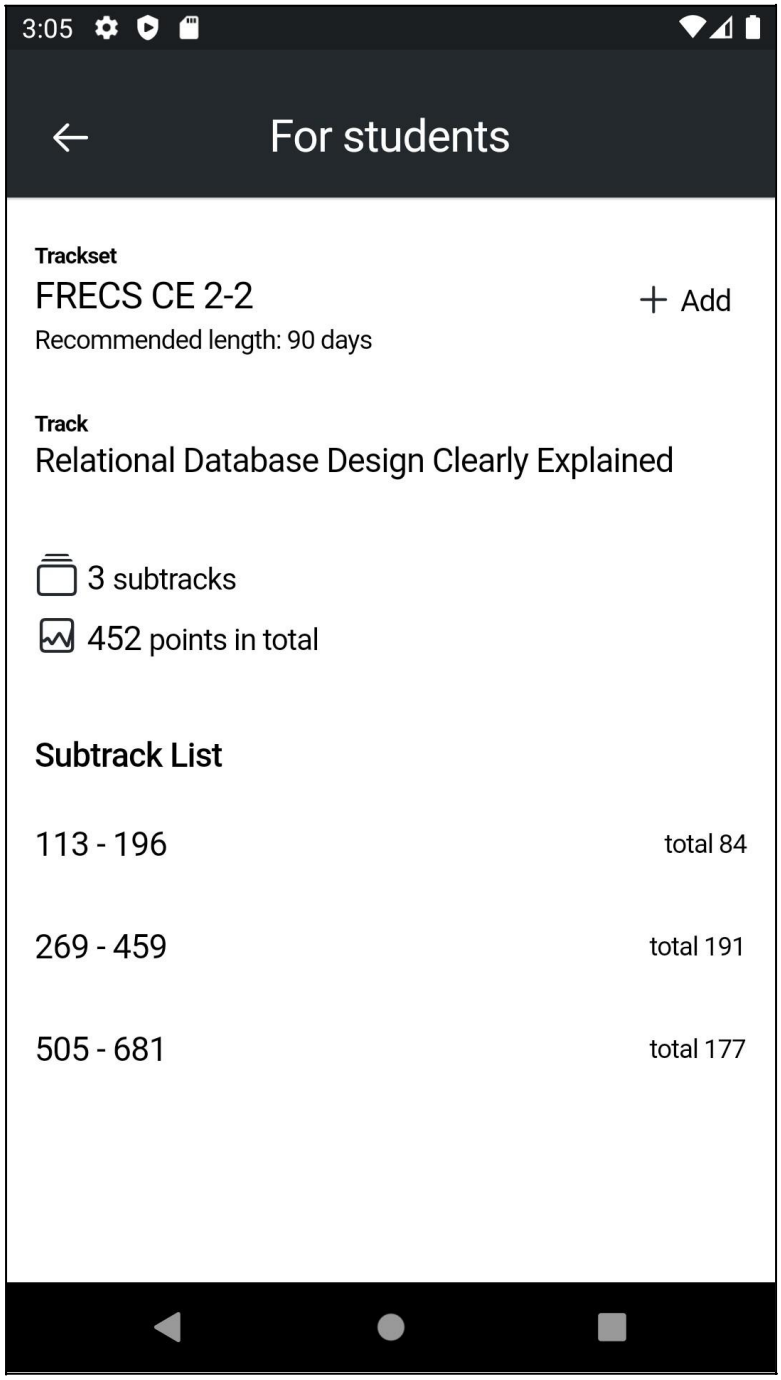

Рис. B.17 - store-track-view