Aplikacja mobilna do zarządzania i tworzenia interaktywnych notatek

(A mobile application to manage and create interactive notes)

Bartosz Sobocki

Praca inżynierska

Promotor: dr Marcin Młotkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

1 września 2023

Streszczenie

Praca zawiera opis projektu wykonania aplikacji mobilnej MobiNote służącej do tworzenia i zarządzania interaktywnymi notatkami. Aplikacja została zaimplementowana w języku Dart z wykorzystaniem frameworka flutter. Przechowywanie danych możliwe jest dzięki bazie danych SQLite przy użyciu biblioteki sqlite3 oraz drift. Praca zawiera opis implementacji i struktury aplikacji MobiNote oraz podręcznik użytkownika opisujący sposób użytkowania aplikacji. Opisano organizację kodu, zastosowane wzorce projektowe oraz rozwiązania problemów informatycznych napotkane podczas tworzenia aplikacji, jak również implementację parsera języka znaczników wzorowanego na języku Markdown.

The thesis contains a description of the design of the MobiNote mobile application for creating and managing interactive notes. The application was implemented in the Dart language using the flutter framework. The data is stored in an SQLite database using sqlite3 and drift libraries. The thesis contains a description of the structure and implementation of the MobiNote application and a user manual describing how to use the application. Code organisation, applied design patterns and solutions for the computer science problems encountered during application development are described, as well as the implementation of the parser of the markup language based on the Markdown language.

Spis treści

1. Wprowadzenie **9**

1.1. Cel projektu 9

1.2. Motywacja 9

1.3. Opis aplikacji MobiNote 11

2. Podręcznik użytkownika **13**

2.1. Strona główna 13

2.1.1. Opcja *Choose Theme* 14

2.1.2. Opcja *Rebuild Database* 14

2.1.3. Zeszyty 14

2.1.4. Notatki 15

2.2. Edycja notatki 15

2.2.1. Pasek strony 15

2.2.2. Pasek narzędzi 16

2.2.3. Edytor strony 16

2.3. Edycja zawartości notatki 17

2.3.1. Akapity 17

2.3.2. Edycja akapitów tekstowych 18

2.3.3. Edycja akapitów widgetów 21

2.3.4. Obrazy 22

2.3.5. Listy 22

3. Implementacja **25**

3.1. Technologie 25
3.1.1. flutter 25
3.1.2. SQLite 26
3.2. Organizacja repozytorium 26
3.3. Wykorzystane rozwiązania 27
3.3.1. Baza danych 27
3.3.2. Dobre praktyki 28

4. Struktura aplikacji MobiNote 31
4.1. Notatki 31
4.2. Przechowywanie notatki 32
4.2.1. Akapity 32
4.2.2. Widgety 34
4.3. Parsowanie notatki 34
4.3.1. Problem parsowania tekstu ze znacznikami stylu 34
4.3.2. Implementacja parsera 35
4.4. Prezentowanie notatki 39
4.4.1. Widgety 39
4.4.2. Tekst 40

5. Instalacja 43
5.1. Android 43
5.1.1. Instalacja aplikacji z nieznanych źródeł 43
5.1.2. Instalacja z pliku apk-release.apk 44

6. Podsumowanie 45

7. Rozwój aplikacji MobiNote 47
7.1. Struktura 47
7.1.1. Baza danych 47
7.1.2. Strona główna 47
7.2. Rozszerzenie istniejących rozwiązań 48

7.2.1. Lista 48
7.2.2. Widgety wewnętrzne w tekście 48
7.2.3. Dodatkowe widgety w notatce 48
7.3. Funkcjonalność 49
7.3.1. Alarmy 49
7.3.2. Powiadomienia 49
7.3.3. Nagrywanie głosowej notatki 49
7.3.4. Karty step-by-step 49

Bibliografia **51**

Rozdział 1.

Wprowadzenie

1.1. Cel projektu

Celem projektu jest wytworzenie aplikacji mobilnej do zarządzania notatkami w interaktywny sposób. Aplikacja ma umożliwić użytkownikowi prowadzenie i tworzenie notatek pozwalających na formatowanie tekstu i dołączanie załączników. Formatowanie tekstu wzorowane będzie na języku znaczników Markdown [1]. Główną różnicą w stosunku do edytorów używających tego typu formatowania tekstu będą:

- brak konieczności zmian widoków (pomiędzy edycją surowego tekstu, a widokiem zawierającym sformatowany tekst);
- przełączenia całej zawartości pomiędzy trybem edycji, a trybem użytkowym;
- potrzeby zapisu notatki do wyświetlania formatowania i zawartych załączników.

Graficzny interfejs użytkownika powinien być przystępny i pozwalać na formatowanie wybranych elementów, podczas gdy pozostała zawartość notatki jest wyświetlana w trybie widoku. Do przechowywania zawartości notatek powinna zostać użyta baza danych, jak również odpowiedni format zapisu danych do przechowywania zawartości niebędącej tekstem.

1.2. Motywacja

Motywacją do stworzenia projektu MobiNote była potrzeba aplikacji pełniącej rolę brudnopisu używanego podczas wielu codziennych czynności. Jestem osobą, która uwielbia:

- tworzyć notatki;
- prowadzić dzienniki;
- zapisywać przepisy;
- tworzyć wszelakie listy;

- dzielić problemy na mniejsze części i rozpisywać je;
- prowadzić różnego rodzaju zeszyty.

Dzięki temu mogę zawsze sięgać do zapisków swoich myśli i pomysłów w chwilach, gdy są mi potrzebne.

Dostępne na rynku aplikacje, takie jak Evernote [2], czy ostatnio używane przeze mnie proste w użyciu Samsung Notes [3], w wielu aspektach sprawdziło się i przyniosło wiele korzyści. W obu przypadkach interfejs użytkownika jest przejrzysty, a funkcjonalność obszerna. Jednak po dłuższym okresie użytkowania tych, oraz innych aplikacji dostrzegam w nich problemy lub braki, dlatego chciałbym używać notatnika wolnego od nich.

Bardzo lubię korzystać z języka znaczników Markdown. Oferuje on możliwość formatowania tekstu, tworzenia różnych nagłówków i elementów notatki przy użyciu jedynie odpowiedniej składni zawartej w zwykłym tekście. W przeciwieństwie do omawianych wyżej aplikacji nie trzeba:

- przeszukiwać opcji w poszukiwaniu przycisków, za pomocą których użytkownik określa typ i wielkość czcionki;
- przełączać zwykłej klawiatury na klawiaturę zawierającą narzędzia do formatowania tekstu;
- wchodzić w menu, aby zaimportować zdjęcia, dodać listy itd.

Używanym przeze mnie serwisem był HackMD [4]. Posiada on jednak duży minus, którym jest brak aplikacji mobilnej, oraz dwa ekrany: edycji tekstu i sformatowanego widoku. Zależało mi na tym, aby notatka była renderowana i formatowana na bieżąco na jednym ekranie z ewentualnym ujawnianiem i chowaniem znaczników w tekście. Tekst powinien być stylizowany już w polu tekstowym uwzględniając dodawanie i usuwanie znaczników w czasie rzeczywistym.

Chciałem stworzyć aplikację, która będzie posiadała szereg wymaganej przeze mnie funkcjonalności, jak na przykład:

- liczniki, abym mógł w trakcie treningu odliczać wykonane serie;
- alarmów, które odliczałyby przerwy pomiędzy seriami;
- przycisków informacyjnych otwierających okno dialogowe z informacjami dotyczącymi na przykład wymienionego w notatce miejsca;
- edycji stylów tekstu za pomocą znaczników, zamiast przycisków i odrębnej klawiatury.

Aplikacja nie powinna wymagać ode mnie dużego wysiłku podczas tworzenia zapisków:

- pracując nad projektem;
- ucząc się nowych rzeczy;
- zapisując myśli w nocy tuż przed spaniem.

Chciałem, aby aplikacja była szybka, prosta i przejrzysta, jednocześnie pozwalając użytkownikowi na dodawanie, edycję oraz interakcję różnego typu widgetów, alarmów czy powiadomień. Wtedy aplikacja może również służyć w stanie, gdy aktualnie nie jest edytowana. W moich intencjach było, aby interfejs użytkownika był dopasowany do moich potrzeb i gustu. Chciałem, by aplikacja odpowiadała mi pod wieloma względami, nie tylko w kwestii oferowanych funkcji, ale również wyglądu, czy rozmieszczenia i dostępności funkcji, przycisków, czy widgetów tak, aby te najczęściej używane były najłatwiej dostępne i proste w obsłudze.

1.3. Opis aplikacji MobiNote

Aplikacja MobiNote służy do tworzenia i zarządzania notatkami w sposób opisany powyżej. Głównym zamysłem aplikacji jest pomoc korzystającemu w prowadzeniu, tworzeniu i zapisywaniu notatek, w których skład wchodzi nie tylko tekst, ale również przyciski, obrazy, listy, liczniki i różnego rodzaju pomocne widgety. Aplikacja umożliwia tworzenie i utrzymywanie brudnopisu używanego podczas codziennych czynności, jak również przejrzystych, dopracowanych i przystępnych notatek.

Wytworzone notatki mogą być używane w formie brudnopisu, między innymi podczas:

- treningu na siłowni;
- organizacji przyjęcia;
- nauki z wykorzystaniem sesji pomodoro;
- tworzeniu listy zakupów;
- wielu innych codziennych czynności.

Mogą także w przyjemny dla oka sposób przechowywać i wyświetlać informacje przygotowane w celu tworzenia:

- dziennika;
- notatek do nauki;
- spisu pomysłów i ważnych myśli;
- różnego rodzaju list i opisów.

Przykładem może być lista miejsc, które użytkownik chciałby odwiedzić wraz z opisem i zdjęciami miejsc, które chciałby tam zobaczyć.

Interaktywność notatki jest zapewniana poprzez możliwość tworzenia zawartości na bieżąco za pomocą klawiatury i dostępnego interfejsu użytkownika. Użytkownik może tworzyć styl tekstu za pomocą znaczników dodawanych wewnątrz tekstu w odpowiednich miejscach, podobnie do języka znaczników Markdown. Dodając odpowiednie znaczniki w tekście można ustawić wielkość czcionki w danym akapicie, kursywę, podkreślenie, przekreślenie, a także pogrubienie. Tekst jest automatycznie formatowany wraz z dodaniem znaczników, co pozawala na bieżąco obserwować i

dostosowywać style i wielkości czcionki do potrzeb korzystającego.

Notatki będą zapisywane lokalnie, co pozwoli zaoszczędzić czas ładowania, jak również pominąć problemy związane z synchronizacją.

Interfejs użytkownika jest prosty i przejrzysty, posiada motyw ciemny (**dark**), jasny (**light**) oraz ułatwiający użytkowanie osobom słabowidzącym (**easy**).

Rozdział 2.

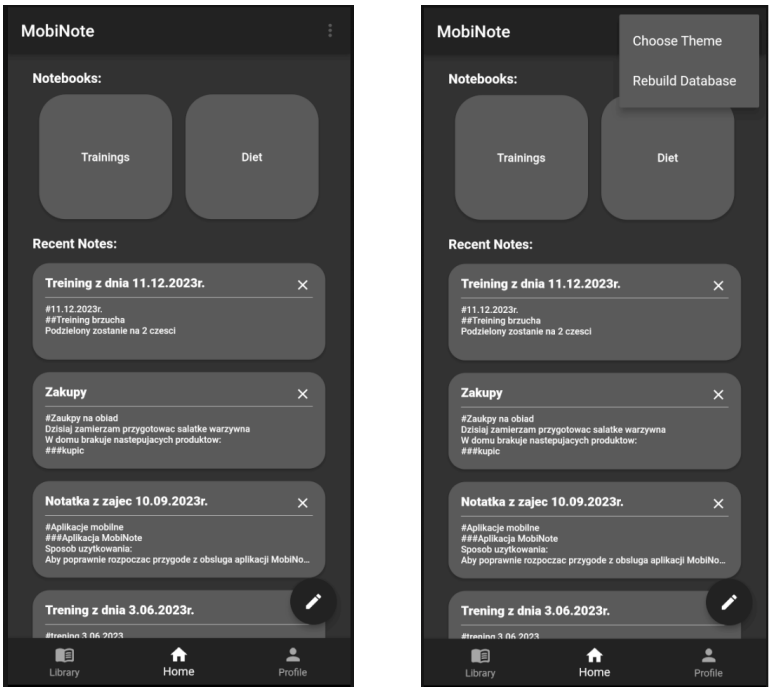
Podręcznik użytkownika

2.1. Strona główna

Po uruchomieniu aplikacji użytkownik zostaje przeniesiony na stronę główną. Znajdzie tam:

- swoje notatki i zeszyty;
- przycisk w prawym dolnym rogu służący do tworzenia nowej notatki;
- dodatkowe elementy w menu górnego paska.

Elementy w menu górnego paska umożliwiają zmianę motywu, czy reset bazy danych w celu usunięcia wszystkich notatek.

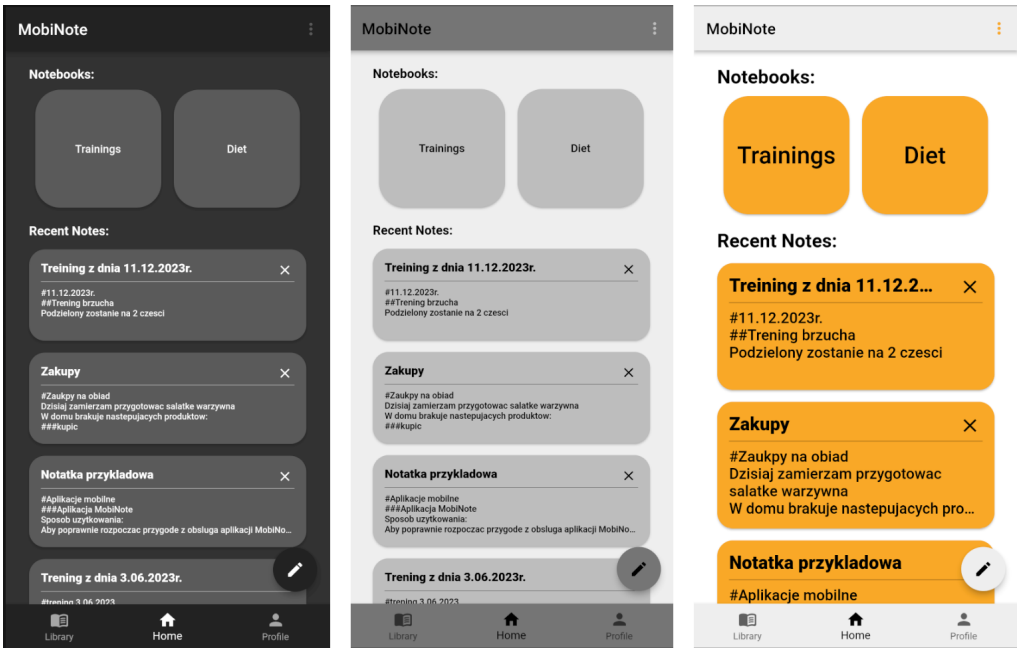


Rysunek 2.1: Strona główna aplikacji MobiNote z wybranym motywem **dark**.

2.1.1. Opcja *Choose Theme*

Po uruchomieniu tej opcji wyświetli się okienko dialogowe z wyborem motywu, jakiego użytkownik chciałby użyć. Do wyboru mamy trzy mowtywy: **dark**, **light** oraz **easy**.

Motywy **dark** oraz **light** służą jako główne motywy aplikacji, zachowując te same wielkości elementów tekstu i całej aplikacji. Dla osób mających problemy z widocznością tekstu przy domyślnych ustawieniach wielkości i kolorów aplikacji przygotowany został motyw **easy**. Motyw ten cechują kontrastujące ze sobą kolory, jak również zwiększone rozmiary czcionek, ikon, przycisków i paska narzędzi.



Rysunek 2.2: Porównanie motywów na tej samej stronie głównej.

2.1.2. Opcja *Rebuild Database*

Opcja ta otwiera okno dialogowe z pytaniem o to, czy na pewno chcemy wykonać operację przebudowy bazy danych. Po zatwierdzeniu baza danych jest usuwana i budowana ponownie.

2.1.3. Zeszyty

Pod etykietą ***Notebooks*** dostępne są dwa przyciski: **Trainings** oraz **Diet**. Są to roboczo dodane przyciski, które są przygotowane do rozszerzenia aplikacji o możliwość układania notatek w zeszyty, dla lepszej organizacji, a także wprowadzać etykiety. Pomysł ten zostanie omówiony w 7. rozdziale.

2.1.4. Notatki

Kolejnym elementem strony głównej jest lista notatek znajdująca się pod etykietą ***Recent Notes***. Każdy element zawiera tytuł oraz pierwsze cztery surowe linie tekstu (zawierające znaki specjalne stylów w przypadku akapitu będącego tekstem, lub string w formacie JSON reprezentujący widget zawarty w danym akapicie). W prawym górnym rogu danych elementów znajduje się przycisk **X** służący do usunięcia z bazy danych notatki reprezentowanej przez ten element. Po naciśnięciu na jeden z omawianych elementów użytkownik zostaje przeniesiony do ekranu wyświetlania i edycji wybranej notatki.

W prawym dolnym rogu ekranu widnieje okrągły przycisk z ikoną ołówka (Rysunek 2.2). Po jego naciśnięciu korzystający zostaje przeniesiony do ekranu edycji, gdzie może stworzyć i zapisać nową notatkę.

2.2. Edycja notatki

Po wybraniu notatki, lub przejściu do tworzenia nowej, na ekranie pojawi się strona edycji notatki. Składa się ona z głównego paska strony, paska narzędzi oraz edytora.

2.2.1. Pasek strony

Przycisk zapisu i powrotu

Aby wyjść i zapisać notatkę korzystający używa przycisku powrotu. Ważne jest, aby zamiast systemowych przycisków nawigacyjnych użyć właśnie tego przycisku nawigacyjnego dostępnego w górnym pasku, ponieważ wraz z powrotem do strony głównej zapisuje on notatkę, jeśli ta uległa zmianie.

Za zmianę notatki uznawane są:

- edycja tytułu;
- edycja tekstu w akapicie tekstowym;
- edycja widgetu w akapicie widgetu.

WAŻNE! Nowa notatka nie zostaje utworzona w momencie otwarcia okna edycji, a dopiero poprzez użycie przycisku powrotu pod warunkiem, że jej tytuł lub zawartość uległy zmianie. Oznacza to, że przejście do edycji nowej notatki, a następnie powrót, nie zapiszą pustej notatki, jednak dodanie i usunięcie znaku umożliwią zapis przy powrocie.

Edytor tytułu

Jest to pole tekstowe widniejące zaraz obok ikony przycisku (zapis i powrót). Służy ono do edycji tytułu notatki.

Przycisk zmiany opcji zapisu

Ostatnim elementem paska jest **przycisk zmiany opcji zapisu**. Jest to przycisk typu **switch** domyślnie ustawiony na true. Każdorazowe kliknięcie zmienia jego logiczną wartość. Wartość **true** oznacza zapis notatki w przypadku jej edycji, natomiast **false** oznacza brak zapisu stanu notatki, nawet jeśli została ona zmieniona.

2.2.2. Pasek narzędzi

W tym pasku znajdują się przyciski oznaczone ikonami. W aktualnej wersji aplikacji dostępne są dwa przyciski: dodanie obrazu (ikona obrazu), oraz dodanie listy (ikona listy).

Dodanie obrazu

Do zawartości notatki można dodawać również obrazy. Aby to zrobić należy kliknąć przycisk z ikoną obrazu na pasku narzędzi edytora notatki, a następnie wybrać z urządzenia zdjęcie, które ma zostać wstawione. Zdjęcie zostanie dodane do notatki.

Dodanie listy

Po wybraniu tej opcji na ekranie pojawi się lista złożona początkowo z jednego elementu. Jest on domyślnie ustawiony na typ **checkbox**. Oznacza to, że jest to wiersz zawierający checkbox oraz puste pole tekstowe.

2.2.3. Edytor strony

Edytor strony zajmuje resztę powierzchni ekranu. Jest polem, w którym następuje edycja tekstu oraz widgetów. Aktualny stan wizualny jest odświeżany na bieżąco, dzięki czemu użytkownik obserwuje zmiany w efekcie końcowym w czasie rzeczywistym.

2.3. Edycja zawartości notatki

Dla lepszego zrozumienia działania aplikacji MobiNote będą używane sformułowania **akapit tekstowy** oraz **akapit widgetów**. Są to nazwy opisujące abstrakcję użytą podczas konstruowania struktury aplikacji.

2.3.1. Akapity

Notatki w aplikacji składają się z akapitów, które dzielimy na akapity tekstowe i akapity widgetów. Każdy akapit może przechodzić w jeden z dostępnych trybów. Obecnie wyróżnione są tryby:

- widoku;
- edycji;
- zaznaczenia;
- niewidoczny.

tryb widoku

Akapit istnieje w trybie widoku, kiedy nie jest aktywny. Akapit przechodzi w tryb widoku, w momencie przeniesienia aktywności na inny akapit (na przykład poprzez kliknięcie).

tryb edycji

Akapit przechodzi w tryb edycji, w momencie przeniesienia na niego aktywności. Aktywność ta może zostać przeniesiona poprzez kliknięcie na elementy akapitu, lub w przypadku próby usunięcia akapitu tekstowego będącego następnikiem danego akapitu widgetów.

tryb zaznaczenia

Akapit widgetów przechodzi w tryb zaznaczenia poprzez przytrzymanie jego elementów niebędących polem tekstowym (pole tekstowe ma już zarezerwowany ten gest na zaznaczanie tekstu).

tryb niewidoczny

Akapit przestaje być widoczny w momencie przejścia edytora w kierunku pionowym wystarczająco do wysunięcia akapitu poza edytor.

2.3.2. Edycja akapitów tekstowych

Każdy tekst zamykany jest w akapicie tekstowym od początku, aż do znaku końca linii. Oznacza to, że styl, jaki zostanie nadany na początku linii zostanie zaaplikowany na całą linię. Przykładem jest ustawienie akapitu jako nagłówka.

Dodawanie

Dodawanie akapitów tekstowych odbywa się za pomocą klawiatury. W momencie, gdy użytkownik podczas edycji doda znak nowej linii utworzy się nowy akapit będący następnikiem edytowanego.

Ważne: Podczas dodawania nowego akapitu widgetów utworzy się nowy akapit tekstowy będący jego następnikiem. Ma to na celu zachowanie możliwości ciągłej pracy z tekstem.

Usuwanie

Usunąć akapit tekstowy można poprzez naciśnięcie przycisku **delete** na samym początku tekstu. Jeśli poprzedni akapit jest akapitem tekstowym, wówczas pozostały tekst kopiowany jest na koniec poprzedniego akapitu.

Ustawianie nagłówka

Do formatowania tekstu używany jest język znaczników wzorowany na Markdown. Ustawienie nagłówka odbywa się poprzez wstawienie znaku # na początku linii, przed tekstem.

Dostępne nagłówki

- nagłówek1 #
- nagłówek2 ##
- nagłówek3 ###
- nagłówek4 ####

Brak nagłówka oznacza, że dana linia jest zwykłym akapitem o domyślnej wielkości czcionki i odstępów oraz nie zawiera pogrubienia.

Odkrywanie znaków nagłówka

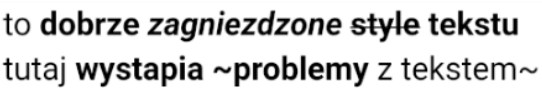
Edytor posiada funkcję odkrywania znaków nagłówka, w celu ułatwienia edycji danego nagłówka. Dzięki temu użytkownik może zobaczyć który aktualnie nagłówek jest wybrany oraz w prosty sposób przechodzić pomiędzy typami nagłówków dodając bądź usuwając znak #.

Rysunek 2.3: Odkryte znaki w edytowanych nagłówkach.

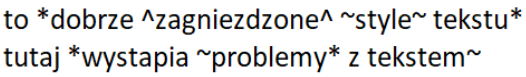
Ustawianie stylów

Kolejnym ważnym elementem aplikacji MobiNote są style tekstu. Użytkownik, chcąc sformatować tekst w odpowiednim stylu, wprowadza do niego symbole specjalne oznaczające konkretne style. Odbywa się to według poniższych zasad:

- za początek oznaczenia stylu uznawany jest wzór:
  `[dowolny znak][symbol stylu][znak niebędący białym znakiem]`
- za koniec oznaczenia stylu uznawany jest wzór:
  `[znak niebędący białym znakiem][symbol stylu]`
- dla każdego znacznika początkowego stylu musi istnieć znacznik końcowy;
- style mogą być łączone – pomiędzy znacznikami stylu A można dodać znaczniki stylu B;
- style nie mogą się przecinać – jeśli dodajemy znacznik początkowy stylu B pomiędzy znacznikami stylu A, to znacznik końca stylu B powinien wystąpić przed znacznikiem końca stylu A.



Rysunek 2.4: Zagnieżdżone style w prawidłowy i nieprawidłowy sposób



Rysunek 2.5: Surowy tekst ukazujący rozmieszczenie znaków.

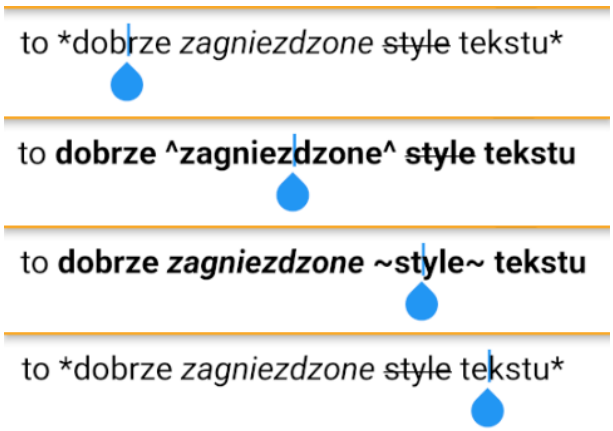
Znaki specjalne

Każdy dostępny styl tekstu oznaczony jest za pomocą znaków:

- * – pogrubienie
- ^ – kursywa
- _ – podkreślenie
- ~ – przekreślenie

Odkrywanie znaków specjalnych

Edytor tekstu posiada funkcję odwijania stylu w momencie, gdy kursor znajduje się bezpośrednio w środku stylu. Odwijany jest tylko styl, którego jeden ze znaków specjalnych znajduje się najbliżej kursora. Funkcjonalność ta została wprowadzona, aby użytkownik mógł łatwiej edytować i usuwać style. Dzięki temu może dostrzec, gdzie konkretnie znajdują się symbole danego stylu w tekście.



Rysunek 2.6: Przykłady odkrytych znaków specjalnych.

Usuwanie i edycja stylów

Aby usunąć dany styl wystarczy usunąć reprezentujące go znaki specjalne. Poza odkrytymi znakami reszta jest ukryta, jednak wszystkie znaki nadal występują w tekście. Edytujący może zatem kliknąć bezpośrednio przed widoczny początkowy symbol lub poza tekst oznaczony danym stylem i naciskając przycisk **delete**, na klawiaturze urządzenia, usunąć dany niewidoczny symbol zakończenia stylu.

2.3.3. Edycja akapitów widgetów

Elementami tych akapitów są widgety. W aktualnej wersji aplikacji dostępne są dwa widgety, które mogą być bezpośrednimi elementami akapitów widgetów. Są to **obrazy** i **listy**.

Dodawanie

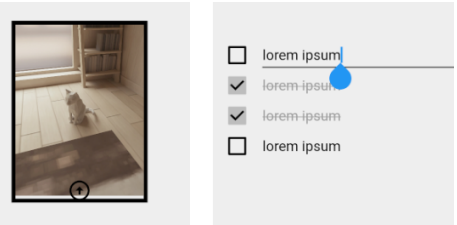
Dodawanie akapitów widgetów odbywa się poprzez użycie przycisków z paska narzędzi edytora notatki.

Usuwanie

Akapit widgetów można usunąć poprzez usunięcie jego wszystkich głównych elementów. W obecnej wersji możliwe jest posiadanie tylko jednego głównego elementu, jednak w planie rozwoju aplikacji przewidziane jest, aby akapity mogły przechowywać i używać większej ilości widgetów, w zależności od rodzajów używanych widgetów.

Tryb Edycji

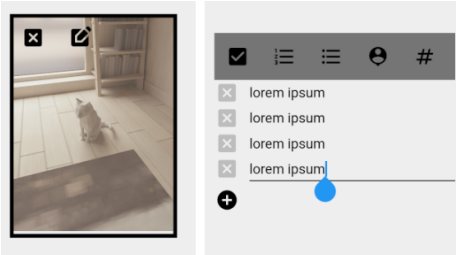
Przejście do trybu edycji odbywa się poprzez interakcję z widgetem, np naciśnięcie na obraz, bądź edycja tekstu w elemencie listy. W przypadku obrazów tryb edycji jest oznaczony poprzez dodanie obramowania do zdjęcia, jak również ikony w dolnej części obramowania służącej do zmiany rozmiaru zdjęcia.



Rysunek 2.7: Tryb edycji przykładowych widgetów głównych.

Tryb zaznaczenia

Przejście w tryb zaznaczenia odbywa się poprzez naciśnięcie i przytrzymanie konkretnych części widgetów. W przypadku obrazu jest to sam obraz, z kolei w przypadku list są to etykiety przypięte do pola tekstowego (w zależności od wyboru mogą być to: checkbox, etykiety tekstowe czy liczniki).



Rysunek 2.8: Tryb zaznaczenia przykładowych widgetów głównych.

2.3.4. Obrazy

Jednym z głównych widgetów są obrazy dodawane poprzez użycie przycisku na pasku narzędzi. Obraz wybierany jest z pamięci urządzenia.

Edycja

Aby zmienić rozmiar zdjęcia użytkownik musi przejść w tryb edycji poprzez kliknięcie na zdjęcie, a następnie przeciągać ikonę ze strzałką (Rysunek 2.7). Automatycznie dostosowywany będzie rozmiar obrazu.

Usunięcie i zmiana

Po wejściu w tryb zaznaczenia ukażą się dwie ikony: ikona usunięcia obrazu oraz ikona zmiany obrazu (Rysunek 2.8). Wybór pierwszej ikony skutkuje usunięciem obrazu wraz z akapitem, natomiast drugiej – przejściem do pamięci urządzenia w celu wyboru zdjęcia.

2.3.5. Listy

Dostępne są różne rodzaje list. Różnią się one głównie etykietą oraz niewielkimi elementami, jak na przykład dostępne przekreślenie i zmiana koloru tekstu przy odznaczonej pozycji.

Edycja

Edycja list następuje poprzez interakcję użytkownika. Użytkownik może naciskać bezpośrednio na etykiety przy polu tekstowym w celu ich edycji (checkbox, licznik), jak również edytować tekst wiersza.

W przypadku niektórych typów wiersza możliwe jest jego odznaczenie. Wówczas tekst wiersza zmienia kolor na szary, a sam tekst zostaje przekreślony.

Dodawanie wierszy

Dodawanie wierszy następuje na dwa sposoby:

1. W dowolnym miejscu listy poprzez dodanie znaku nowej linii.
2. Na koniec listy poprzez kliknięcie przycisku "+"pod etykietami w trybie zaznaczenia.

Dodawanie wierszy poprzez znak nowej linii przenosi tekst na prawo od miejsca dodania znaku nowej linii do nowego wiersza.

Usuwanie wierszy

Usuwanie wierszy jest możliwe w trybie zaznaczania poprzez klikanie na etykiety ze znakiem "x".

Zmiana typu wierszy

Użytkownik może zmienić typ wszystkich wierszy w danej liście poprzez przejście w tryb zaznaczenia, a następnie w górym pasku nad listą (Rysunek 2.8) wybrać jedną z dostępnych opcji.

Dostępne typy wiersza

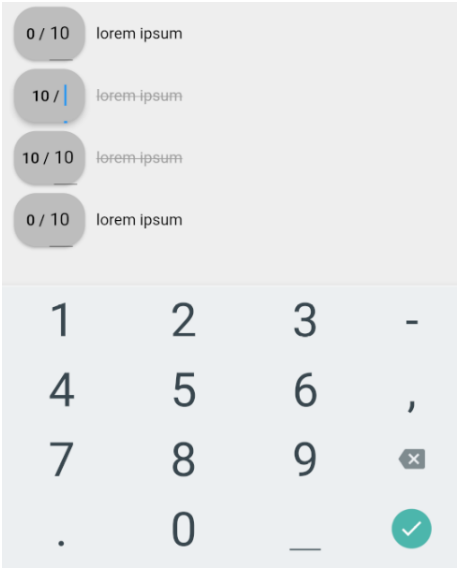
- checkbox;
- numerowane;
- oznaczone przez symbol "-";
- naznaczane przez symbol "*"(w nowszej wersji aplikacji spodziewane jest tworzenie etykiet przez użytkownika);
- licznik.

Aktualnie listy są jednopoziomowe, bez możliwości zmiany wysunięcia od początku wiersza. Możliwość ta jest przewidziana w dalszym rozwoju aplikacji.

Licznik

Licznik służy do odliczania rzeczy opisywanej przez tekst wiersza. Z każdym kliknięciem zwiększana jest wartość licznika (lewa część etykiety), aż do osiągnięcia celu licznika (liczba w prawej części licznika). Wraz z osiągnięciem ustalonego celu wiersz zostaje odznaczony.

Możliwa jest zmiana wartości celu licznika. Aby to zrobić wystarczy nacisnąć na liczbę w prawej części licznika, a następnie wybrać liczbę na klawiaturze i zatwierdzić. Po zmianie następuje reset licznika.



Rysunek 2.9: Edycja celu jednego z liczników.

Rozdział 3.

Implementacja

Implementując aplikację MobiNote starałem się zachowywać czysty kod, podzielony na jak najmniejsze części pod względem odpowiedzialności i zadań klas oraz metod, jak również czyste repozytorium zachowując odpowiedni podział i strukturę katalogów, modułów i pozostałych plików. Przykładałem dużą wagę do jakości oraz wykonania, aby powrót do poszczególnych warstw i miejsc w aplikacji był prosty, a sama jej struktura była przejrzysta i zrozumiała.

3.1. Technologie

3.1.1. flutter

Aplikacja została napisana przy użyciu frameworka flutter w języku Dart.

Wybierając technologie kierowałem się kryteriami takimi jak: prostota, wieloplatformowość, estetyka bez dużego wkładu w kreowanie komponentów na własną rękę oraz wydajność. Jedną z najbardziej polecanych framework'ów na stronie linkedin.com [6] oraz itCraft [7] był flutter. Jak podaje strona główna flutter.dev [8]:

"Flutter is an open source framework by Google for building beautiful, natively compiled, multi-platform applications from a single codebase."

Składnia języka Dart jest stosunkowo prosta i przejrzysta, natomiast framework flutter pozwala na programowanie aplikacji mobilnych w prosty sposób na platformy IOS oraz Android o dużej wydajności zważywszy na to, że język Dart jest kompilowany do natywnego kodu. Pozwala to na tworzenie aplikacji szybko i bez wielkiego nakładu pracy, dając przy tym duże możliwości i przyjemny dla oka rozbudowany interfejs użytkownika.

3.1.2. SQLite

Do przechowywania notatek została wykorzystana baza danych SQLite. Aplikacja wykorzystuje bibliotekę sqlite3 [9], umożliwiającą wykonywanie operacji na lokalnej bazie danych SQLite. Pozwala ona na zapis, odczyt a także manipulację danymi. Dodatkowo wykorzystywany jest pakiet drift [10], który jest rodzajem systemu ORM, umożliwiający mapowanie między obiektami języka Dart oraz tabelami bazy danych SQLite. Pozwala to na pracę bezpośrednio w kodzie Dart używając dostępnej funkcjonalności bez konieczności pisania zapytań bezpośrednio w języku SQL.

3.2. Organizacja repozytorium

Kod aplikacji został podzielony na moduły i zorganizowany w zależności od poziomu abstrakcji i zastosowań.

Główny katalog z kodem źródłowym **lib** zawiera:

- plik **main.dart** z wywołaniem głównej funkcji main budującej aplikację;
- katalog **database**;
- katalog **logic**;
- katalog **screens**.

database

Zawiera definicję bazy danych, struktury oraz metody z nią związane. Posiada wygenerowany na podstawie pliku **database_def.dart** plik **database_def.g.dart** zawierający struktury odzwierciedlające tabele bazy danych w klasy języka Dart.

logic

Znajdują się tam definicje struktur reprezentujących stan komponentów, definicje typów (stylów tekstu, widgetów, znaczników itd.), mapowanie kluczy (String) na style tekstu, znaczniki i elementy, logika parsera, funkcje pomocnicze wraz z funkcjonalnością aplikacji niebędącej bezpośrednią częścią widgetów.

screens

Dostępne są tam definicje stron wraz z funkcjonalnością i metodami ich komponentów, jak również definicje motywów i kolorów aplikacji.

3.3. Wykorzystane rozwiązania

Programowanie Obiektowe

Dart promuje programowanie obiektowe, dlatego też w projekcie został zastosowany paradygmat obiektowego programowania wraz z jego mechanizmami. Przykładami tych mechanizmów może być wielokrotne zastosowanie abstrakcji różnych poziomów, dziedziczenie i polimorfizm.

Przykładem zastosowania wszystkich powyższych mechanizmów jest reprezentacja, tworzenie oraz zarządzanie widgetami. Dzięki zastosowaniu opisanych mechanizmów możliwe było zaimplementowanie fabryki, listy, oraz akapitów trzymanych w edytorze.

Wzorce projektowe

W powyższych opisach możemy zauważyć użycie wzorców projektowych spotykanych podczas implementacji aplikacji mobilnej [11]. Są to między innymi:

- **Fabryka** – użyty w przypadku fabryki widgetów *NoteEditorWidgetFactory*;
- **Singleton** – używany w przypadku dostępu do bazy danych oraz dostępu do ustawionego motywu aplikacji (*MobiNoteDatabase*, *MobiNoteTheme*);
- **Kompozyt** – używany w przypadku struktur drzewiastych takich jak na przykład obiekty *SpanTree*, czy *NoteWidgetData*;

Testy Jednostkowe

Część logiki niezwiązanej bezpośrednio z wyświetlaniem i organizacją komponentów i ich wyglądu testowałem za pomocą testów jednostkowych od razu podczas ich implementacji przy użyciu biblioteki testing [12]. Dzięki temu uniknąłem długich godzin spędzonych na poszukiwaniu problemów, gdy któryś z komponentów mógł nie działać poprawnie. Przyczyną niepoprawnego działania może być nie tylko algorytm, ale i sam widget i funkcjonowanie frameworka flutter. Dostosowując szczegółowość testów jednostkowych starałem się pokryć przypadki użycia jeszcze przed użyciem danych klas, metod i algorytmów w aplikacji. Przetestowany został w ten sposób parser języka znaczników stylów, wraz z pełną funkcjonalnością, jak również konwersja reprezentacji widgetów do formatu JSON.

3.3.1. Baza danych

Do zarządzania dostępem i manipulacją bazy danych tworzony jest jeden obiekt typu **MobiNoteDatabase** dla całego projektu. Jest on tworzony i udostępniany za

pomocą biblioteki do zarządzania stanem **GetX** [13].

W implementacji aplikacji MobiNote, **GetX** pozwala na tworzenie instancji zarządzającej dostępem do bazy danych raz i udostępnianie jej w różnych miejscach aplikacji. Struktura aplikacji wymaga tylko jednej instancji obiektu **MobiNoteDatabase**, dlatego tworzone jest jedno połączenie w jej konstruktorze, a sam obiekt przechowywany jest za pomocą **GetX**.

```
Get.put(MobiNoteDatabase());
```

Następnie za pomocą funkcji **find** może zostać udostępniany

```
final database = Get.find<MobiNoteDatabase>();
```

Umożliwia to zwiększenie wydajności, ponieważ unikamy otwierania połączenia z bazą danych za każdym razem, gdy jest ono potrzebne (obie strony: strona główna, oraz edytor notatki używają połączenia z bazą danych do swoich funkcjonalności). Używany zatem jest tutaj wzorzec Singleton. Z racji tego, że nie ma potrzeby otwierania równoległych połączeń (używana jest tylko jedna ze stron i ich funkcjonalność naraz) wzorzec ten został zastosowany w implementacji.

3.3.2. Dobre praktyki

Leniwe tworzenie widgetów w ListView

Podczas implementacji listy notatek w stronie głównej oraz edytora zawartości notatki użyty został widget **ListView** poprzez wywołanie tzw. *named constructor* **ListView.builder**[14]. Praktyka ta ma dobre zastosowanie przy długich listach, ponieważ używając konstruktora **builder** widgety zawarte w **ListView** budowane są w sposób leniwy, na bieżąco, jeśli istnieje potrzeba ich wyświetlenia. Z racji tego, że budowanie jest szybkie i nieskomplikowane pozwala to uniknąć przechowywania dużej ilości widgetów (znajduje to pozytywne skutki przy długich, skomplikowanych notatkach).

Reprezentacja widgetów

Każdy widget reprezentowany jest poprzez obiekt typu pochodnej klasy **WidgetData** przechowywującej parametry danego widgetu potrzebne do jego prawidłowego wyświetlania oraz funkcjonowania. Oddzielana jest warstwa wyświetlania od warstwy informacji danego widgetu. Pozwala to na zapis stanu widgetu do bazy

danych, ponieważ zawarta w obiektach reprezentujących warstwę informacji jest konwersja do formatu JSON, który jest przechowywany w bazie danych jako fragment zawartości danej notatki.

ID Generator

Widgety zawierają identyfikatory do odróżniania ich w rodzicu lub liście zwierającej je. Ma to zastosowanie m.in. w znajdowaniu miejsca w liście np. do dodawania nowego elementu, czy usuwania i zmiany elementów. Lepiej nadać `id` i na podstawie znalezienia `id` na liście pobierać indeks, zamiast nadawać i aktualizować indeksy elementom przy każdej edycji listy.

Do generowania identyfikatorów w odpowiednich miejscach w kodzie służy obiekt klasy **IdGenerator**.

Fabryka widgetów

Widgety tworzone są na podstawie obiektów reprezentacyjnych (opisanych powyżej) w klasie **NoteEditorWidgetFactory**. Na podstawie pola `type` wybierany jest typ widgetu, który następnie tworzony jest z parametrów obiektu reprezentującego. Używa klasy **IdGenerator** do generowania identyfikatorów, przez co obsługa przydzielania `id` jest robiona w jednym miejscu, w fabryce, która jest przekazywana dzieciom i elementom danych widgetów. Pozwala to zachować spójność w aktualnej implementacji, jak również przy dalszym rozwoju.

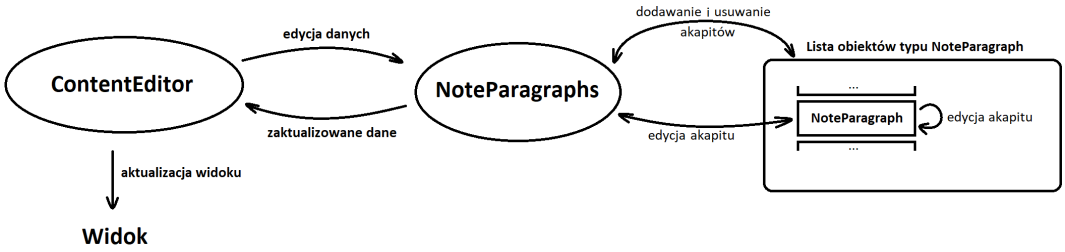
Rozdział 4.

Struktura aplikacji MobiNote

Struktura aplikacji MobiNote została częściowo opisana w 2. rozdziale.

4.1. Notatki

Każda notatka składa się z nagłówka (tytułu) oraz zawartości. Nagłówkiem jest tekst edytowany w polu tekstowym znajdującym się w górnym pasku strony, natomiast zawartość notatki jest ciągiem akapitów dodawanych, edytowanych i usuwanych w edytorze notatki. Akapity mogą przechowywać tekst, jak również widgety. Struktura kodu opiera się na wzorcu MVC (Model View Controller), którą można przedstawić w uproszczony sposób za pomocą schematu:



Rysunek 4.1: Uproszczony schemat struktury i działania edytora notatki.

W obecnej implementacji aplikacji **MobiNote** za widok odpowiada obiekt klasy **ContentEditor**, natomiast kontrolerem danych jest obiekt klasy **NoteParagraphs**, który również przechowuje dane składające się na model. Jest to świadomy zabieg. Model nie jest skomplikowany i rozbudowany, dlatego też nie trzeba tworzyć dodatkowych struktur i klas do zarządzania nim, jak w klasycznym wzorcu MVC. Pozwala to na uproszczenie logiki i struktury kodu, mniejsze wykorzystanie pamięci, oraz skrócenie ścieżki wywołań w przypadku wykonywania konkretnych operacji na danych. Zabieg ten został użyty z myślą o użytkownikach korzystających z mniej wydajnych urządzeń.

4.2. Przechowywanie notatki

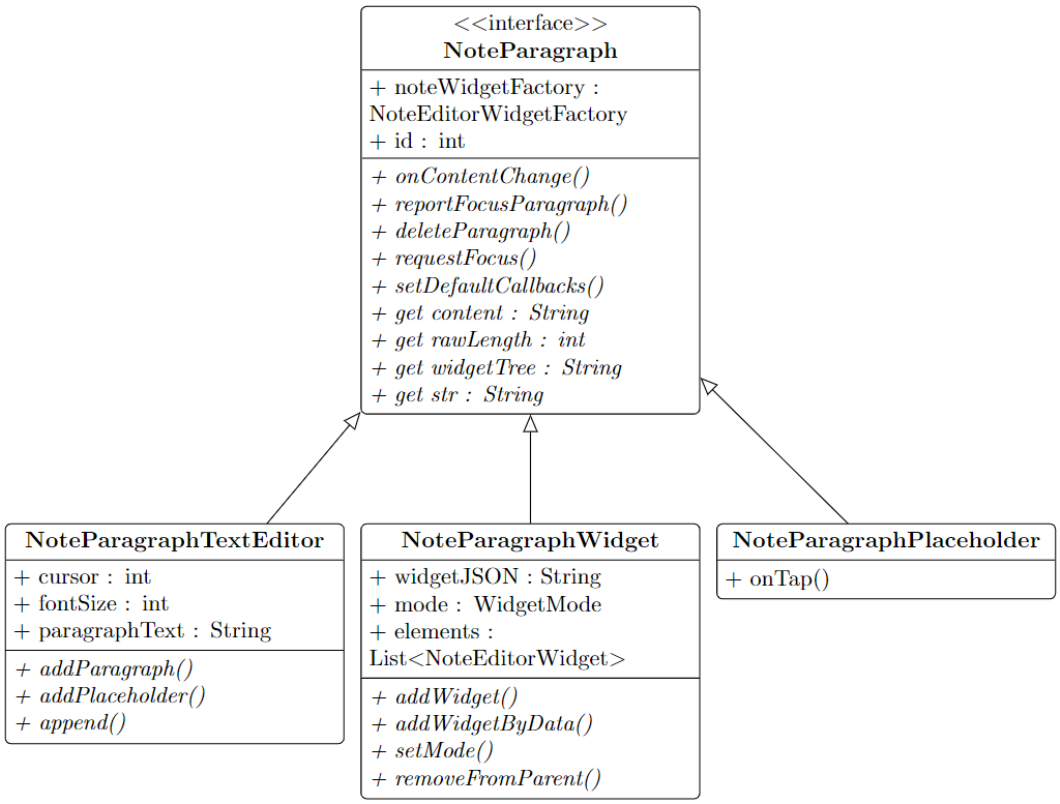
Zawartość notatki przechowywana jest jako lista akapitów w obiekcie klasy **NoteParagraphs**. Każdy akapit jest obiektem klasy dziedziczącej po klasie **NoteParagraph**. W przypadku akapitów tekstowych zawartością przechowywaną w obiekcie będzie tekst, natomiast w przypadku akapitów widgetów będą to parametry widgetów zawartych w danym akapicie. Ten model danych jest zapisywany w bazie danych w odpowiednim formacie oraz odczytywany i konwertowany z powrotem wraz z przejściem do strony edycji danej notatki.

4.2.1. Akapity

Istnieją trzy typy akapitów w edytorze dziedziczące po **NoteParagraph**:

- **NoteParagraphTextEditor**;
- **NoteParagraphWidget**;
- **NoteParagraphPlaceholder**.

Schemat dziedziczenia wygląda następująco:



Każdy z akapitów posiada unikatowy identyfikator `id` oraz referencję do fabryki widgetów tworzonej wewnątrz managera **NoteParagraphs**. Pozwala to na unikatowe identyfikowanie akapitów wewnątrz listy, oraz wszystkich utworzonych widgetów w obrębie przestrzeni edytora.

Używane są również mechanizmy wywołań zwrotnych (callbacks) do zgłaszania aktywności (focus), zaistniałej zmiany, jak również usunięcia samego siebie z listy akapitów, na podstawie własnego `id`, gdy wszystkie jego elementy zostaną usunięte.

NoteParagraphTextEditor

Służy do przechowywania, wyświetlania i edycji tekstu notatki. Posiada mechanizmy dynamicznej zmiany rozmiaru poprzez manipulację stanem oraz wartościami wewnętrznych pól. Posiada tylko jeden element, którym jest pole tekstowe z niestandardowym kontrolerem *NoteTextEditingController* parsującym tekst przy użyciu własnego parsera.

NoteParagraphWidget

Służy do przechowywania, wyświetlania i edycji widgetów notatki. Posiada mechanizmy dynamicznej zmiany rozmiaru wraz ze zmianą rozmiaru widgetu będącego elementem głównym. Mechanizmy zmiany rozmiaru są różne w zależności od rodzaju widgetu.

NoteParagraphPlaceholder

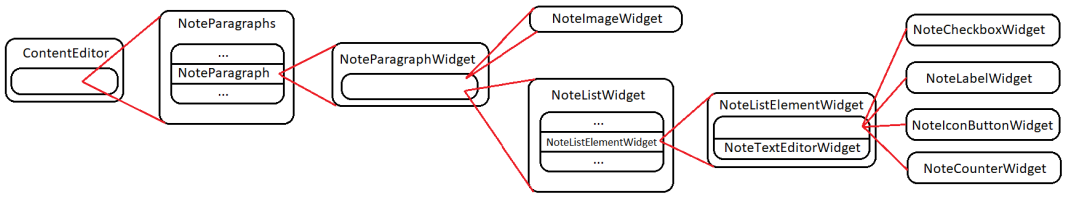
Służy jako wypełnienie pozostałej przestrzeni edytora. Po kliknięciu na niego uruchamia mechanizm przenoszenia aktywności (focus) na ostatni akapit na liście, dzięki czemu użytkownik nie musi znać wielkości i szukać ostatniego akapitu, aby kontynuować edycję notatki.

4.2.2. Widgety

W aktualnej wersji aplikacji widgety dzielą się na:

- widgety główne (będące bezpośrednimi elementami **NoteParagraphWidget**)
- widgety będące elementami widgetów głównych.

W przypadku rozwoju aplikacji struktury widgetów oraz ich hierarchii i tworzenia ulegną zmianie. Aktualnie struktura aplikacji wygląda następująco:



Rysunek 4.2: Struktura widgetów zawartości notatki w aktualnej wersji aplikacji.

Widgety przechowywane są w modelu za pomocą obiektów klas pochodnych od klasy **NoteWidgetData**. Każdy używany typ widgetu posiada opisujący go typ danych **NoteWidgetData**. Wszystkie dostępne typy danych dostępne są w plikach w katalogu *lib/logic/note_editor/widgets/representation/*. Dane te zostają zapisane w bazie danych w formacie JSON.

4.3. Parsowanie notatki

4.3.1. Problem parsowania tekstu ze znacznikami stylu

Do wyświetlania stylizowanego tekstu potrzebne jest jego parsowanie. Funkcjonalność ta będzie wołana stosunkowo często, a dla długich tekstów bez znaku nowej linii będzie to długi do obsługi tekst. Dlatego też ważny jest sposób implementacji.

Zrealizowanym pomysłem rozwiązania problemu jest podział parsowania tekstu na trzy etapy:

1. przekształcanie znaczników stylu w tekście na **znaki unicode**;
2. parsowanie tekstu ze **znakami unicode** na specjalną drzewiastą reprezentację **SpanInfo**;
3. konwersja struktury **SpanInfo** na obiekt **TextSpan**.

4.3.2. Implementacja parsera

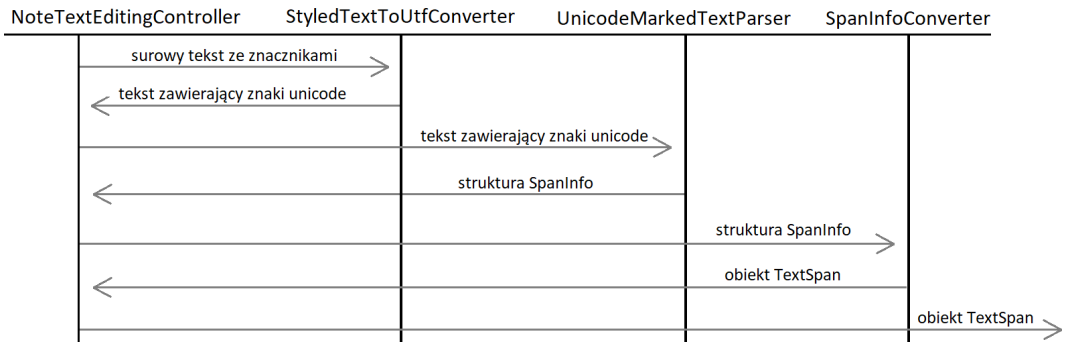
Własna implementacja

Parser wraz z jego strukturą został zaprojektowany i zaimplementowany przeze mnie, zamiast przy użyciu narzędzi generowania parsera dla opisanej gramatyki używanego przeze mnie języka znaczników. Powodem tego jest potrzeba łatwego rozwoju oraz wprowadzenia niestandardowych rozwiązań i dodatkowych funkcjonalności niebędących częścią samej składni gramatyki, jednak potrzebnych wewnątrz implementacji parsera.

Własna implementacja parsera pozwoliła mi od początku do końca zaprojektować i zrozumieć jego strukturę, co pozwoliło mi tworzyć elastyczne środowisko rozwoju tekstu w notatkach. Przykładowo wprowadzanie funkcji odkrywania i ukrywania znaczników stylu oraz nagłówka, bądź dodanie wywołania zwrotnego zmieniającego rozmiar pola tekstowego było prostym zabiegiem niewymagającym dużych zmian i rozszerzeń kodu parsera.

Parsowanie

Każdy z etapów parsowania jest realizowany poprzez obiekt innej klasy. Całość łączona jest w metodzie obiektu **NoteTextEditingController**.



Rysunek 4.3: Schemat przedstawiający etapy parsowania tekstu.

Znaki unicode

Do procesu parsowania zostały użyte znaki unicode z bloku Private Use Area. Są to znaki, które są trudno dostępne dla użytkownika z poziomu klawiatury mobilnej.

Zdefiniowane zostały w następujący sposób:

```
const styleUnicodeNumber = 0xe000;
const widgetUnicodeNumber = 0xe100;
```

```
const elementUnicodeNumber = 0xe1A0;
const paragraphUnicodeNumber = 0xe200;
```

Oznaczają odpowiednio początki przestrzeni znaczników: stylu tekstu, wewnętrznych widgetów i elementów w tekście oraz nagłówków akapitu. Znaczniki początkowe mają parzyste numery, natomiast znaczniki końcowe mają nieparzyste numery następujące po odpowiadających im znacznikach początkowych.

Struktura SpanInfo

Podczas implementacji użyta została struktura **SpanInfo**. Zawiera ona informacje na temat danego węzła w drzewiastej strukturze stylów tekstu, oraz referencję do jego rodzica. Została zaimplementowana w celu usprawnienia przebiegu etapu drugiego parsowania tekstu.

```
class SpanInfo {
  String type;
  String text;
  List<SpanInfo> children;
  late SpanInfo parent;
}
```

Dzięki swojej budowie struktura może zostać przekonwertowana w prosty sposób na obiekt typu **TextSpan**.

StyledTextToUtfConverter

Przekształcanie znaczników odbywa się poprzez przejście przez surowy tekst w ich poszukiwaniu i przetwarzaniu. Znaczniki dzielone są na początkowe i końcowe (zobacz: 2.3.2.). Podczas sprawdzania, czy dany znak jest znacznikiem początkowym, czy końcowym pobierany jest szerszy kontekst, złożony z przylegających do niego znaków.

W momencie, gdy zostanie znaleziony znak będący początkowym znacznikiem, dodawana jest informacja o nim do listy **startBounds** w postaci struktury **SpecialPatternInfo** przechowującej znacznik wraz z jego indeksem.

```
class SpecialPatternInfo {
  final int indexInText;
  final String character;
}
```

W przypadku znalezienia znaku c będącego końcowym znacznikiem, przeszukujemy listę **startBounds** w poszukiwaniu pierwszego wystąpienia znacznika c' o takiej samej wartości. Jeśli nie zostanie znaleziony, wówczas znak c traktowany jest

jako zwykły tekst. Jeśli natomiast zostanie znaleziony, oznacza to, że wystąpił wcześniej początek stylu, a cały tekst pomiędzy aktualnym indeksem, a indeksem zapisanym w strukturze opisującej początkowy znacznik c' jest tekstem zawierającym ten styl. Zatem znalezione znaczniki zamieniane są na swoje odpowiedniki w formie znaków unicode, a znaczniki istniejące na liście **startBounds** będące następnikami znalezionego c' zostają usunięte z listy (granice stylów nie mogą się przecinać), a więc nie zostaną zamienione.

W podobny sposób konwertowane są również widgety i elementy wewnętrzne w tekście. W aktualnej wersji aplikacji nie są one obsługiwane, jednak sam parser został przygotowany na obsługę tego typu problemu. Szukane są zatem również fragmenty tekstu będące tagami widgetów, bądź oznaczeniami elementów.

UnicodeMarkedTextParser

Z racji tego, ze znaki unicode z zakresu Private Use Area są trudno dostępne, na potrzeby samego problemu konwersji tekstu notatek możemy założyć, że znaki w tekście symbolizują początkowe i końcowe znaczniki, które na pewno mają swoje odpowiedniki i są poprawnie użyte (nie przecinają się).

Na początku algorytmu tworzona jest referencja *currentSpan* na obiekt **SpanInfo**, który jest aktualnie przetwarzanym stylem (napotkany został jego znacznik początkowy). Początkowo jest to korzeń całej struktury **SpanInfo**.

Oznaczony w ten sposób tekst przechodzimy ponownie posiadając dokładne informacje, które symbole są znacznikami stylu, a które zwykłymi fragmentami tekstu.

Podczas przechodzenia tekstu zapisujemy znaki niebędące znacznikami do tymczasowego bufora.

Napotykając znacznik początkowy, jeśli w buforze znajduje się tekst, to wówczas jest on dodawany do tekstu przechowywanego w *currentSpan*. Następnie tworzony jest nowy **SpanInfo** (na podstawie stylu reprezentowanego przez znacznik) będący dzieckiem *currentSpan*. Następnie referencja do nowo powstałego obiektu zapisywana jest jako *currentSpan*.

Po napotkaniu znacznika końcowego dodajemy tekst z tymczasowego bufora do *currentSpan*, czyścimy bufor i przestawiamy *currentSpan* przechodząc po węzłach **SpanInfo** (używane jest do tego pole z referencją *parent* wskazującą na rodzica danego węzła) na pierwszy napotkany węzeł przechowujący informacje na temat stylu reprezentowanego przez aktualnie sprawdzany znacznik.

Przetwarzane są również informacje na temat widgetów i elementów wewnętrznych przygotowane w ramach podstawy do rozwoju funkcjonalności w przyszłości.

SpanInfoConverter

Klasa **TextSpan** jest klasą pochodną klasy **InlineSpan**.

SpanInfoConverter buduje strukturę **InlineSpan** złożoną z obiektów klas pochodnych **TextSpan** oraz **WidgetSpan**. Wewnętrzne widgety i elementy, które mają w przyszłości być obsługiwane przy użyciu **WidgetSpan** nie są aktualnie obsługiwane, dlatego przyjęty w opisie model struktury opiera się tylko na obiektach typu **TextSpan**.

Przechodząc przez strukturę **SpanInfo** budujemy strukturę złożoną z obiektów typu **TextSpan**. Zapisywany jest *mainSpan* będący korzeniem struktury, natomiast przejście przez strukturę odbywa się w sposób rekurencyjny.

```
SpanInfo mainSpan = SpanInfo(type: 'null');

InlineSpan getSpans(SpanInfo spanInfo) {
    if (mainSpan.type == 'null') mainSpan = spanInfo;

    List<InlineSpan> spanChildren =
        spanInfo.children.map((e) => getSpans(e)).toList();

    /* ustawianie stylu i typu obiektu
     * oraz zwracanie go wraz z children */
}
```

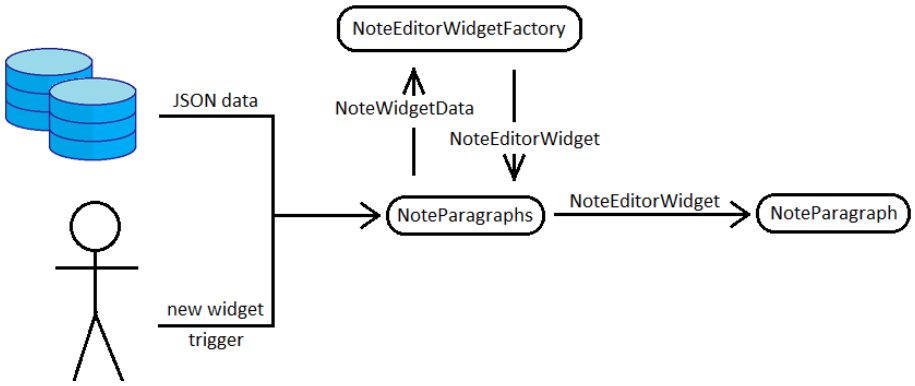
4.4. Prezentowanie notatki

Za widok notatki odpowiada klasa **ContentEditor**. Pozwala ona na wyświetlanie zawartości akapitów oraz ich elementów, jak również za jej pośrednictwem użytkownik może edytować notatkę z poziomu widoku.

4.4.1. Widgety

Tworzenie widgetów

Widgety tworzone są poprzez interakcję użytkownika przy użyciu interfejsu użytkownika (patrz 2.3.3.), jak również poprzez przetwarzanie zawartości notatki z bazy danych. W obu przypadkach tworzone są odpowiednie obiekty typu **NoteWidgetData** na podstawie użytej funkcjonalności bądź przetworzonych danych pobranych z bazy danych. Na ich podstawie tworzone są widgety wyświetlane w edytorze.



Rysunek 4.4: Schemat tworzenia widgetów.

Usuwanie widgetów

Widgety usuwane są poprzez interakcję użytkownika z dostępnym interfejsem graficznym po przejściu w tryb zaznaczenia (patrz 2.3.3.). Widgety usuwane są w swoich rodzicach, a jeśli brakuje widgetów potomnych, wówczas widget listy bądź akapit widgetów zostaje usunięty.

4.4.2. Tekst

Praca z tekstem dzieli się na edycję tekstu z użyciem logiki zawartej w **NoteParagraphTextEditor** i **NoteTextEditorWidget**, jak również wyświetlanie go wraz ze zmianą rozmiarów z pomocą **NoteTextEditingController**.

Tekst edytowany jest za pomocą pola tekstowego TextField udostępnianego jako widget przez framework flutter. Posiada on wewnętrzny padding poziomy (lewa i prawa strona) jako stałą wartość, oraz padding pionowy (góra, dół) obliczany na podstawie różnicy pomiędzy wielkością czcionki użytej w danym akapicie, a domyślnej czcionki z dodatkiem podstawowej wartości niezerowej (w przypadku, gdy dany akapit nie ma ustawionego nagłówka, nie chcemy braku odstępu między akapitami z powodu zerowego paddingu pionowego).

Przy każdej zmianie tekstu sprawdzana jest jego zawartość w poszukiwaniu informacji o stanie akapitu. Stan ten określa się poprzez wielkość czcionki i stylów tekstu użytych w akapicie.

Dodawanie tekstu

Jeśli wewnątrz pola tekstowego został dodany znak nowej linii, wówczas wołane jest wywołanie zwrotne (callback), zainicjalizowane przy tworzeniu danego akapitu przez **NoteParagraphs**, służące do dodania nowego akapitu. Wywołanie to przyjmuje identyfikator danego akapitu i na tej podstawie oblicza miejsce w liście, do którego należy włożyć nowo powstały akapit. Przy tworzeniu nowego akapitu następuje przeniesienie tekstu następującego po znaku końca linii. Znak końca linii jest usuwany.

Usuwanie tekstu

Flutter nie oferuje możliwości przechwytywania zdarzenia naciśnięcia przycisku **delete**, gdy kursor znajduje się na początku tekstu. W tej sytuacji najprościej jest użyć przechwytywania zdarzenia edycji tekstu w akapicie.

W każdym akapicie na początku dodawany jest znak `placeholder = \u200b`. Jest to znak unicode reprezentujący spację o zerowej długości, a więc niewidoczną w tekście. Jest to swego rodzaju wartownik. Jego brak na początku tekstu podczas sprawdzania tekstu po edycji mówi o potrzebie usunięcia akapitu.

Gdy użytkownik zechce usunąć akapit, wówczas przenosi się kliknięciem na początek wiersza (o ile już się na nim nie znajduje), a następnie naciska przycisk **delete**. Dzięki temu usuwa `placeholder`, a przy tym wywoływany jest callback służący do usunięcia akapitu. Jako argument przyjmuje identyfikator akapitu, aby określić miejsce usunięcia.

Aby uniknąć kliknięcia na początek linii przed wartownika został dodany mechanizm zmiany pozycji kursora na drugi znak (bezpośrednio za wartownikiem) za każdym razem, gdy jest on na pozycji zerowej.

Wyświetlanie tekstu

Do wyświetlania stylizowanego tekstu używane są obiekty typu **TextSpan**. Obiekty te posiadają drzewiastą strukturę, ponieważ mogą zawierać dzieci tego samego typu, które będą dziedziczyć dane parametry stylu, o ile nie zostaną dla nich specjalnie zainicjalizowane. Daje to możliwość zorganizowania tekstu bez potrzeby dzielenia go na małe kawałki oraz nadawania za każdym razem łączonych stylów (gdyby obiekty **TextSpan** były przechowywane jako elementy jednej listy). Można zainicjalizować jeden styl wewnątrz drugiego, co bardzo upraszcza strukturę i pozwala na wprowadzanie globalnych zmian struktury (jak na przykład wielkość czcionki całego akapitu) tylko w korzeniu, zapewniając, że wszystkie części tekstu odziedziczą zmianę.

Aktualizacja i renderowanie wyglądu pola tekstowego na bieżąco jest możliwe poprzez wykorzystanie klasy **TextEditingController**. Posiada ona metodę **buildTextSpan**, która służy do budowania wyglądu i stylu tekstu poprzez obiekty **TextSpan**. Dziedzicząc z tej klasy i nadpisując metodę **buildTextSpan** mogłem użyć parsera tekstu zawierającego znaczniki do przekonwertowania go na stylizowany tekst. Dzięki temu przy każdej zmianie tekst jest parsowany, a jego wygląd aktualizowany.

Utworzony w ten sposób **NoteTextEditingController** może nie tylko używać parsera do tworzenia obiektów **TextSpan**, ale również mógł zostać rozszerzony o dodatkowe parametry, takie jak:

- callback **resizeTextField** wołany podczas parsowania, gdy wielkość czcionki w tekście ulega zmianie;
- możliwość przechowywania informacji o tym, czy dany akapit jest aktualnie w użyciu;
- sprawdzanie i manipulacja pozycją kursora.

Callback **resizeTextField** został użyty w kontrolerze, zamiast w widgecie pola tekstowego, ponieważ wtedy wystarczy parsować tekst jednokrotnie. W momencie zwrócenia drzewiastej struktury tekstu **TextSpan** możemy ustawić rozmiar pola tekstowego w jego korzeniu, zamiast edytować zawartość widgetu używając innych mechanizmów i parsować dodatkowo tekst w poszukiwaniu informacji na temat wielkości czcionki.

Informacja na temat aktualnego użycia akapitu jest używana do wyświetlania i ukrywania znaczników stylu nagłówka.

Sprawdzanie i manipulacja pozycją kursora pozwala na odkrywanie znaczników

stylu, jak również ustawianie pozycji, aby kursor nie znajdował się przed znakiem `placeholder` (zobacz 4.4.2.).

Przy każdej zmianie tekstu odświeżany jest widok akapitu, dlatego wystarczy zmienić wartość zmiennej odpowiadającej za jego rozmiar przed końcem całej operacji związanej z przetwarzaniem nowego tekstu.

Rozdział 5.

Instalacja

Aplikacja MobiNote jest dostępna i testowana na urządzeniach mobilnych z systemem operacyjnym Android. Kod źródłowy aplikacji wraz z plikiem **.apk** można uzyskać pobierając repozytorium **MobiNote** pod linkiem: `https://github.com/bsobocki/MobiNote`.

5.1. Android

Do instalowania aplikacji na urządzeniach posiadających system Android, Google wprowadziło platformę **Google Play**. Oferuje ona możliwości sprawdzenia aplikacji pod względem niechcianych zachowań, czy instalacji złośliwego oprogramowania. Aplikacje są instalowane za pomocą dostępnego interfejsu użytkownika. Domyślne ustawienia urządzenia pozwalają wyłącznie na instalacje aplikacji z zaufanego źródła, jakim jest wspomniany **Google Play** [5].

Istnieje również niestandardowy sposób instalacji aplikacji bez użycia oferowanego rozwiązania opisanego powyżej. Użytkownik może zainstalować aplikację za pośrednictwem pliku o rozszerzeniu **.apk**. Użytkownik robi to na własną odpowiedzialność, dlatego ważne, żeby aplikacja pochodziła z zaufanego źródła. Aby to zrobić, należy wyłączyć blokadę instalacji aplikacji z nieznanych źródeł.

5.1.1. Instalacja aplikacji z nieznanych źródeł

Do zainstalowania aplikacji MobiNote potrzebna jest wyłączona blokada instalacji aplikacji z nieznanych źródeł dla menadżera plików. W tym celu należy:

- otworzyć **Ustawienia**
- w **Ustawieniach** przejść do sekcji **Aplikacje**
- następnie kliknąć na ikonę menu (trzy kropki w prawym górnym rogu)
- przejść do opcji **Dostęp specjalny**

- następnie **Zainstaluj nieznane aplikacje**
- wybrać menadżera plików i włączyć dla niego tę opcję

Ważne! Dla bezpieczeństwa po zainstalowaniu aplikacji warto ponownie wyłączyć możliwość instalowania aplikacji z nieznanych źródeł, jeśli więcej aplikacji nie będzie w ten sposób instalowanych.

5.1.2. Instalacja z pliku apk-release.apk

Aby zainstalować aplikację na urządzeniu mobilnym z systemem Android należy:

- pobrać repozytorium za pomocą polecenia
  `git clone git@github.com:bsobocki/MobiNote.git`
- podłączyć urządzenie i przenieść pliki **apk-release.apk** do wybranego przez siebie miejsca docelowego na urządzeniu;
- włączyć możliwość instalacji nieznanych aplikacji (szczegóły powyżej);
- w managerze plików znaleźć miejsce docelowe pliku **apk-release.apk** i uruchomić;
- kliknąć **Zainstaluj**.

Rozdział 6.

Podsumowanie

Stworzyłem aplikację MobiNote na moje własne potrzeby, według moich upodobań. Wygląd i funkcjonalność aplikacji zostały zaprojektowane z myślą o przystępności i wygodzie użytkowania. Jestem zadowolony z aktualnego rezultatu, dlatego używanie aplikacji w wersji, która została wydana wraz z napisaniem danej pracy jest dla mnie przyjemne. Podczas codziennego testowania i używania aplikacji napotykałem problemy, które rozwiązywałem wraz z dalszą implementacją, jak również odkrywałem, jakie funkcjonalności mogą być przydatne i wprowadzałem je. Przykładem takiej funkcjonalności jest możliwość zmiany rozmiaru zdjęcia, która nie została przewidziana podczas planowania aplikacji.

Praca nad rozwojem aplikacji pozwoliła mi na rozwinięcie umiejętności implementowania aplikacji mobilnych, której dotychczas nie posiadałem. Użycie wzorców projektowych oraz dobrych praktyk programowania pozwoliły mi odpowiednio zaprojektować strukturę aplikacji i kodu. Wykorzystałem wiedzę zdobytą podczas studiów między innymi na temat rozwoju oprogramowania, programowania obiektowego, tworzenia i operowania danymi przy użyciu bazy danych, inżynierii oprogramowania, jak również zdobytą umiejętność rozwiązywania problemów.

Aplikacji MobiNote używam od momentu ukończenia parsera, ponieważ już w samej wersji tekstowej aplikacja pozwalała na tworzenie przydatnych notatek i zapisków. Zaprojektowany sposób interakcji z zawartością notatki sprawdził się podczas tworzenia notatek do treningu na siłowni, ponieważ pozwala na sprawne tworzenie i edycję list. Uważam, że dobrym pomysłem było zaimplementowanie własnego parsera, ponieważ podczas użytkowania aplikacji zauważyłem, że ujawnianie znaczników stylów i użytych nagłówków w akapitach tekstowych jest bardzo pomocne przy edycji tekstu i dostosowywaniu formatowania. Ustawianie typów wielkości nagłówka w akapitach okazało się również przydatne podczas tworzenia odstępów pomiędzy akapitami, ponieważ aplikacja umożliwia ustawienie wielkości nagłówka, który może się składać z samej spacji, dzięki czemu odstępy te mogą być zwiększane zgodnie z upodobaniami użytkownika.

Widget listy w różnych trybach sprawdził się podczas różnych aktywności: od treningu na siłowni (lista typu licznik) po robienie zakupów (lista typu checkbox), natomiast dodawanie obrazów pozwala na rozszerzenie notatek o przydatne schematy, czy poprawę wyglądu. Jest to kolejny powód, dzięki któremu pomimo swojej prostoty aplikacja ma dla mnie charakter użytkowy, a nie tylko teoretyczny na potrzeby pracy.

W wykonaniu aplikacji dostrzegam również drobne błędy i niedociągnięcia, jak na przykład niekontrolowane ukrywanie i ujawnianie klawiatury podczas przewijania ekranu, lub dodawania nowego akapitu tekstowego. Dzieje się tak, ponieważ przy tworzeniu nowych obiektów typu **TextField** zgłaszana jest jego aktywność (focus) oraz przebudowywany jest stan (state) widgetu głównego widoku **ContentEditor**. Przy dalszym rozwoju aplikacji niezbędne byłoby użycie menadżera stanu [15], aby lepiej zorganizować odświeżanie widoku i aktualizowanie stanu widgetów.

Podczas pracy nad aplikacją narodziły się pomysły na rozwój istniejących rozwiązań, natomiast podczas użytkowania aplikacji w różnych sytuacjach rósł apetyt na kolejne funkcjonalności. Pomysły te omówię w następnym rozdziale.

Rozdział 7.

Rozwój aplikacji MobiNote

7.1. Struktura

7.1.1. Baza danych

W kolejnych wersjach aplikacji MobiNote przewidziane jest rozszerzenie bazy danych o dodatkowe tabele:

- *Notebooks* przechowującą wpisy dotyczące utworzonych zeszytów;
- *NotesInNotebook* zawierającą wpisy dotyczące notatek w danych zeszytach.

7.1.2. Strona główna

Na stronie głównej dostępne są przygotowane już przyciski **Trainings** oraz **Diet** reprezentujące przykładowy widok, który będzie używany w kolejnej wersji aplikacji MobiNote. Przyciski te reprezentują instancje typu **Notebook**, które będą odzwierciedleniem wpisów tabeli **Notebooks**.

Po naciśnięciu na przycisk użytkownik zostanie przeniesiony do strony z dostępnymi notatkami, będącymi częścią zeszytu reprezentowanego przez dany przycisk.

Przyciski w dolnym pasku strony głównej przygotowane są na przełączanie pomiędzy jej widokami. W obecnej aplikacji dostępny jest tylko widok główny, jednak przewidziany jest jeszcze widok zawierający tylko listę zeszytów (Library), oraz widok profilu użytkownika i jego ustawień (Profile).

7.2. Rozszerzenie istniejących rozwiązań

7.2.1. Lista

Klasa **NoteElementListWidget** posiada zmienną `int depth` służącą do określania poziomu głębokości w liście. Ma to na celu określenie wielkości wcięcia, jak również używanych etykiet do oznaczania danego elementu (różne znaki, numeracja itd).

Dodatkowym elementem do edycji listy jest możliwość własnej definicji etykiet w liście. Będzie to ograniczone do pewnej głębokości. W zależności od głębokości, użytkownik będzie mógł zdefiniować symbol lub napis, który będzie używany jako etykieta.

7.2.2. Widgety wewnętrzne w tekście

Aktualnie w kodzie przygotowane są miejsca do parsowania widgetów wewnętrznych w tekście. Przygotowane są definicje wzorów w tekście, które miałyby zostać przekonwertowane w widgety.

Widgetami wewnętrznymi są między innymi:

- linki stron internetowych;
- linki notatek w bazie;
- inline latex – do dodawania na przykład symboli matematycznych;
- zdjęcia (o rozmiarze nieprzekraczającym rozmiaru czcionki);
- cytowanie tekstu (zmiana koloru tła tekstu oraz czcionki).

Użycie obiektów **InlineSpan** udostępnianych przez flutter pozwala tworzyć drzewiastą strukturę, której korzeniem jest **TextSpan**, natomiast dziećmi są inne obiekty **InlineSpan**, czyli instancje obiektów klas pochodnych **TextSpan** i **WidgetSpan**.

7.2.3. Dodatkowe widgety w notatce

Przygotowane również zostały obiekty typu **NoteWidgetData** oraz miejsca w **NoteListElementWidget** na dodatkowe elementy. Przykładem takiego elementu jest **NoteInfoPage**. Będzie on przyciskiem z ikoną, który po kliknięciu otwiera okno dialogowe z notatką/stroną informacyjną, którą będzie można przewijać.

Planowanymi dodatkowymi widgetami są między innymi:

- notatka głosowa;
- tabele.

7.3. Funkcjonalność

7.3.1. Alarmy

Jedną z głównych funkcjonalności aplikacji MobiNote zaplanowaną na kolejne iteracje jest możliwość dodawania i ustawiania alarmów. Alarmy te będą używane w elementach list. Będą odmierzać czas i informować użytkownika o końcu przerwy, na przykład pomiędzy seriami ćwiczeń, bądź pomiędzy sesjami nauki.

7.3.2. Powiadomienia

Kolejną zaplanowaną funkcjonalnością jest dodanie powiadomień. Powiadomienia mają służyć jako przypomnienia ustawione w notatce dotyczące konkretnych działań. Przykładowo w notatce dotyczącej urodzin bliskiej osoby możemy ustawić przypomnienia o odebraniu tortu, bądź prezentu. Nada to użyteczność notatce nawet, gdy nie będzie ona bezpośrednio w użyciu.

7.3.3. Nagrywanie głosowej notatki

Przydatną funkcjonalnością danej aplikacji będzie możliwość nagrywania notatek głosowych. Będą one przydatne w sytuacji, kiedy użytkownik będzie potrzebował na szybko zapisać daną treść i wrócić do niej w późniejszym czasie.

7.3.4. Karty step-by-step

Ostatnią funkcjonalnością przewidzianą w aplikacji MobiNote będą karty **step-by-step**. Funkcjonalność ta będzie polegać na przygotowaniu listy rzeczy do zrobienia, aby osiągnąć zaplanowany cel. Następnie użytkownik zapisuje daną listę i przechodzi do widoku danej notatki. Widok ten składać się będzie tylko z jednego pola z opisem kroku, na którym aktualnie znajduje się użytkownik. Będą dostępne przyciski nawigacyjne pozwalające na:

- cofnięcie się o jeden krok;
- przejście do następnego kroku.

Karty te mają na celu pokazać użytkownikowi jedynie następny krok, zamiast całej listy. Pozwala to na uniknięcie nadmiernego rozmyślania i poczucia przytłoczenia związanego z ilością rzeczy, jakie trzeba wykonać, aby osiągnąć zaplanowany cel.

Bibliografia

[1] Dokumentacja i opis języka znacznika markdown dostępne są na stronie https://www.markdownguide.org/

[2] Strona główna aplikacji Evernote w języku polskim: https://evernote.com/intl/pl

[3] Opis aplikacji Samsung Notes dostępny jest na stronie https://www.samsung.com/pl/apps/samsung-notes/

[4] Opis serwisu HackMD wraz z funkcjonalnością dostępny jest pod adresem https://hackmd.io/s/features

[5] Opis platformy Google Play dostępny na stronie https://play.google/howplayworks/

[6] SSTech System on linkedin.com (15.03.2023) *BEST 9 MOBILE APP DEVELOPMENT FRAMEWORKS IN 2023*, https://www.linkedin.com/pulse/best-9-mobile-app-development-frameworks-2023-sstech-system/

[7] itCraft (26.05.2023) *Flutter w świecie aplikacji mobilnych: Czy to przyszłość programowania?*, https://itcraftapps.com/pl/blog/flutter-w-swiecie-aplikacji-mobilnych-czy-to-przyszlosc-programowania/

[8] Opis oraz dokumentacja frameworka flutter: https://www.flutter.dev

[9] Opis biblioteki sqlite3 dla języka Dart: https://pub.dev/packages/sqlite3

[10] Opis biblioteki drift dla języka Dart: https://pub.dev/packages/drift, dokumentacja: https://drift.simonbinder.eu/docs/getting-started/

[11] Wzorce projektowe spotykane podczas implementacji aplikacji mobilnych z użyciem frameworka flutter opisane zostały na stronie: https://flutterdesignpatterns.com/

[12] Dokumentacja biblioteki testing dla języka Dart dostępna jest pod adresem: https://dart.dev/guides/testing

[13] Dokumentacja biblioteki GetX dla języka Dart dostępna jest pod adresem: https://pub.dev/packages/get

[14] Zastosowanie **ListView.builder** opisane jest na stronie: `https://docs.flutter.dev/cookbook/lists/long-lists`

[15] Podejścia zarządzania stanem aplikacji mobilnej z użyciem narzędzi i bibliotek frameworka flutter dostępne są pod adresem: `https://docs.flutter.dev/data-and-backend/state-mgmt/options`