POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Łukasz Zajkowski
Nr albumu s16347

Kreator tworzenia prostych aplikacji mobilnych na smartfonie z systemem Android

Praca magisterka napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, lipiec 2022

Streszczenie

Praca dotyczy możliwości stworzenia kreatora prostych aplikacji mobilnych bezpośrednio na urządzeniu końcowym z systemem Android. Rozwiązania umożlwiającego osobom bez wiedzy oraz umiejętności programistycznych wygenerowania aplikacji przydatnych do codziennych zadań w domu bądź małym przedsiębiorstwie. Praca rozważa zastosowanie oraz sens takiego programu w obecnym świecie, w którym smartfony z niewielkimi ekranami dotykowymi służą głównie do przeglądania treści głównie w Internecie, niż pracy konstruktywnej.

Słowa kluczowe:
android, kotlin, xml, json, firebase, gradle, dagger, hilt, jetpack compose, android studio, aplikacja mobilna, kreator, programowania generyczne

Spis treści

1. Wstęp

Urządzenia mobilne w obecnych czasach są używane przez większość ludzi na świecie. Statystki pokazują że 83.07% ludzi posiada smartfona [11]. Korzystanie z urządzeń takiego typu opiera się głównie na używaniu dostępnych aplikacji, które dostarczają szereg funkcjonalności. Niektóre aplikacje pomagają nam w udostępnianiu danych, inne służą do komunikacji z pozostałymi użytkownikami czy też nawigacji podczas kierowania samochodem. Większość organizacji i firm posiada własna aplikacje mobilną. Aby taką stworzyć często praca jest zlecana zewnętrznym firmą informatycznym. Mniej stosowanym rozwiązaniem lecz coraz bardziej popularnym są platformy bez kodowa (*ang. no-code*). Kreatory tego typu są głównie aplikacjami sieciowymi przeznaczonymi do korzystania na komputerze. Niniejsza praca skupia się na próbie stworzenia aplikacji mobilnej tworzącej osobną aplikacje według preferencji użytkownika bezpośrednio na smartfonie.

1.1. Cel pracy

Celem pracy jest zbadanie możliwości stworzenie prostego kreatora aplikacji mobilnej bezpośrednio na smartfonie z systemem Android. Program powinien być przeznaczony dla osób bez znajomości kodowania i zagadnień programistycznych. Praca porusza zagadnienia związane z możliwościami rozwiązania od strony technologicznej oraz użyteczności. Aplikacje mobilne pozwalające na stworzenie użytkownikom rozwiązania według swoich potrzeb nie są zbyt popularne oraz dostępne. Oprogramowanie na smartfonach dla użytkowników końcowych w obecnych czasach ma głównie zastosowanie w przeglądaniu dostępnych treści lub dodawaniu własnych. Ideą pracy jest sprawdzenie ograniczeń, które wynikają z samego urządzenia jakim jest smartfon, który posiada niewielkie ekrany dotykowy jako główny interfejs wejściowy i wyjściowy komunikacji z użytkownikiem, jak również przedstawienie procesu kreowania w sposób zrozumiały dla odbiorcy. Celem pracy jest również stworzenie prototypu kreatora, który został wykorzystany do przeprowadzenia badań na temat praktyczności zaproponowanego rozwiązania wśród użytkowników nie technicznych.

1.2. Organizacja pracy

W rozdziale drugim przestawione są dostępne rozwiązania na rynku pozwalające na stworzenie własnych aplikacji bezpośrednio na smartfonie. Zaprezentowano w nim najbardziej interesujące funkcjonalności samego kreatora jak również efekty jakie można uzyskać. Opisano również w jaki sposób można uzyskać dostęp do końcowej aplikacji oraz odczucia użytkownika podczas eksploracji.

W rozdziale trzecim opisano główne narzędzia i technologie jakie wykorzystano do stworzenia prototypu.

Rozdział czwarty zawiera szczegółowy opis stworzonego prototypu, wybrane podejście dla implementacji poszczególnych elementów systemu. Rozważania na temat wybranych funkcji. Przedstawiono główny funkcjonalności kreatora i jakie opcje oferuje. Zaprezentowano od strony technicznej działanie modułu generatora aplikacji końcowej na podstawie zapianych ustawienia jak również sposób przechowywania danych.

Rozdział piąty przedstawia przykładowe rozwiązanie jakie zostały zbudowane przy wykorzystaniu prototypu oraz porównanie z założeniami wstępnymi.

W rozdziale szóstym przedstawiono wyniki badań, które sprawdziły użyteczności aplikacji oraz wady i zalety według ankietowanych.

Ostatni rozdział siódmy zawiera podsumowanie oraz wnioski nabyte podczas realizacji pracy.

1.3. Rezultaty pracy

W rezultacie wykonanych prac, autor przeanalizował możliwość stworzenia aplikacji typu kreator bezpośrednio na urządzeniu końcowym. Głównym poruszanymi aspektami są możliwości technologiczne oraz użyteczności. Praca wskazuje wady i zalety na podstawie zaimplementowanego prototypu. Proces budowy oraz końcowy rezultat dostarczył licznych wniosków wraz z ograniczeniami zaproponowanej aplikacji. Poruszono również możliwości rozwoju danego prototypu, czy też elementy które musi zawierać każda aplikacja o podobnym działaniu.

2. Porównanie istniejących kreatorów aplikacji mobilnych

Poniższy rozdział porównuje najpopularniejsze rozwiązania umożliwiające stworzenie własnej aplikacji bez posiadania umiejętności kodowania. Opisane narzędzia zostały wyszukane w oficjalnym katalogu aplikacji Android (Sklep Play) za pomocą angielskich fraz takich jak: „*creator*”, „*create app*”, „*builder*”.

Kreatory zostały przetestowane oraz porównane pod względem możliwości oraz procesu kreowania w podstawowych demonstracyjnych wersjach:

- APPER Make an App without coding. Easy and fast [1]
- App Builder - Create own app (FREE App maker) [2]
- MobEasy : Create Mobile Apps without coding [3]

Testy powyższych aplikacji zostały wykonane na smartfonie OnePlus 7T Pro z systemem Android 10.

2.1. APPER Make an App without coding. Easy and fast

Twórcy *APPER* [1] opisują swoje narzędzie jako najłatwiejszy, najtańszy oraz rewolucyjny sposób utworzenia aplikacji mobilnych nawet jeśli nie mamy wiedzy na temat ich tworzenia. Według twórców dzięki *APPER* stworzymy profesjonalna, ładnie wyglądające rozwiązanie dla swojej firmy bądź organizacji.

APPER pobrano z Sklepu Play ponad 1 mln. razy i oceniono na 3,9 w skali 5. [1]

Dostępne funkcje:

- Wybór stylu menu
- Dobranie własnych ikon
- Dołączanie kanałów społecznościowe
- Obsługa linków do stron www

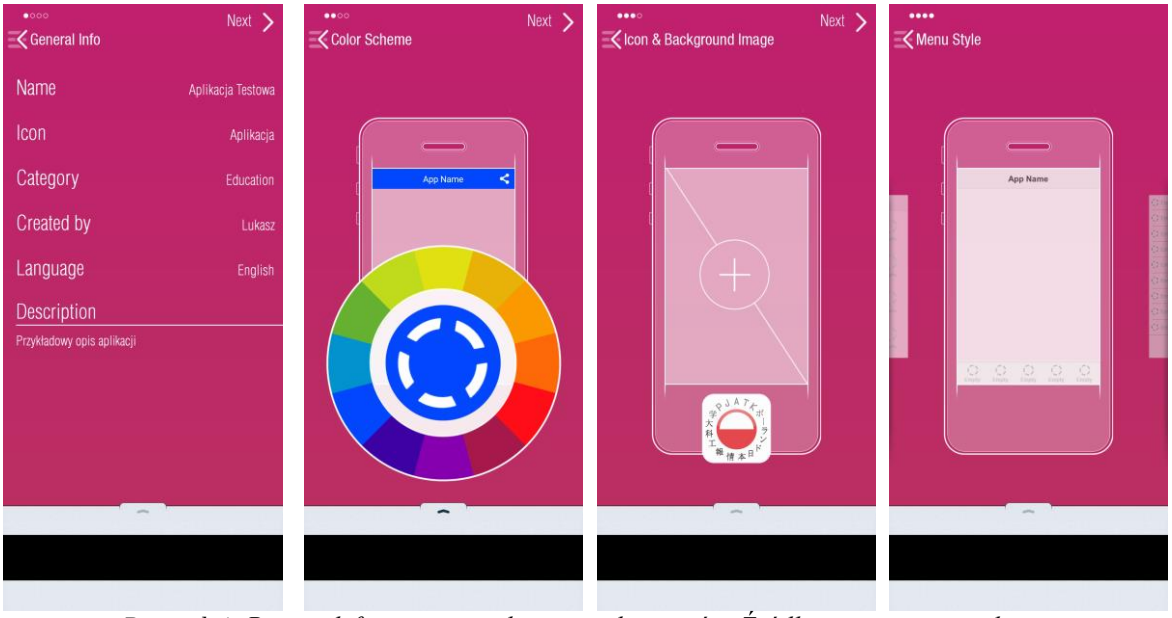
Autor również podaje, iż stworzyć można takie rozwiązania jak:

- Zarządzanie spotkaniami
- Sprzedaż produktów
- Otrzymywać darowizny/ płatności

W celu skorzystania z aplikacji należy zarejestrować konto za pomocą adresu e-mail.

Proces tworzenia

Pierwszym krokiem jest podanie podstawowych danych tworzonej aplikacji: nazwa, kategoria, nazwa autora rozwiązania, język oraz opis.



Rysunek 1. Proces definiowania podstawowych wartości. Źródło: opracowanie własne

Następnie określamy domyślny kolor dla całej naszej aplikacji spośród dostępnej palety kolorów oraz własna ikonkę jako wgrywany obraz z galerii telefonu. Opcjonalnie istnieje opcja dodania tła aplikacji.

Proces szczegółowego działania i wyglądu zaczynami od zdefiniowania stylu menu z kilku dostępnych.

W zależności od wybranego menu wybieramy typ ekrany spośród:

- Ekran z obrazami
- Lista
- Mapa
- Odtwarzacz video
- Usługi zewnętrzne
- Połączenie z serwisami społecznościowymi
- Strona informacyjna

Dany typ ekranu zawiera szablony, funkcjonalności które ma spełniać. Opis trzech wybranych typów ekranów i ich możliwości ustawień:

1. Lista

   W celu stworzenia ekranu z listą najpierw należy wybrać styl listy takie jak proste wyświetlanie elementów bez dodatkowych kontrolek lub razem z obrazkami, polami wyboru lub data i godzina. Kolejnym krokiem jest uzupełnienie listy elementami i wartościami, które są danymi stałymi. Nie jesteśmy w stanie dodawać czy modyfikować zawartości listy jako użytkownik końcowy aplikacji co mocno ogranicza tę funkcję.

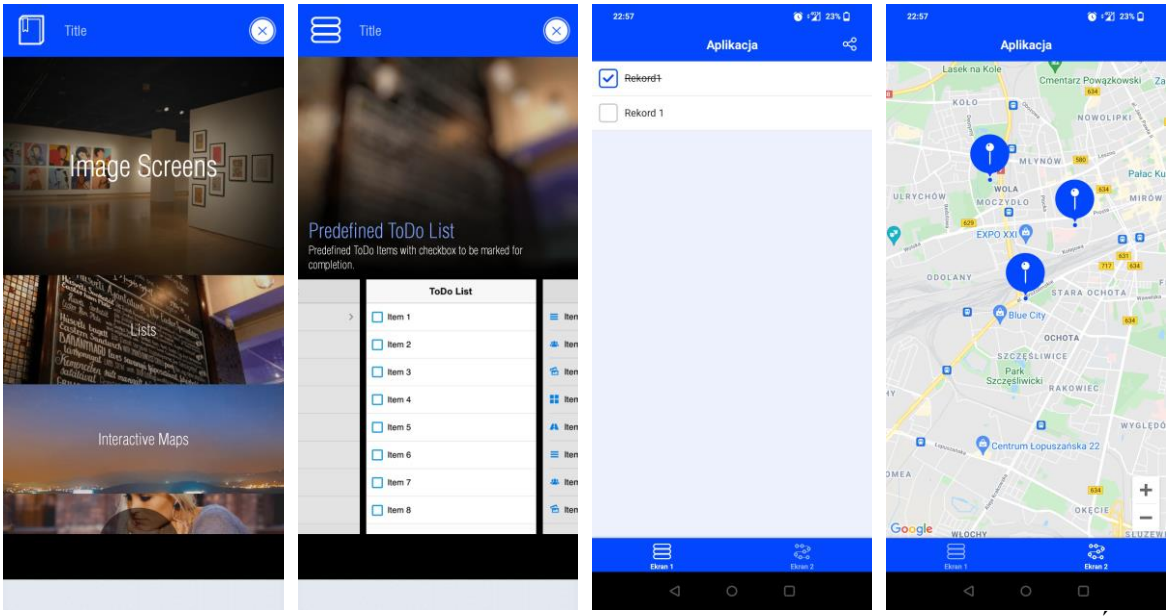
2. Mapa

   W przypadku mapy mamy dostępne tylko trzy opcje: pojedyncza lokalizacja, wiele lokalizacji, trasa. Tak samo jako dla listy wybrane lokalizacje oraz trasa służą tylko i wyłączanie jako funkcja informacyjna i nie podlegają zmiana w gotowej aplikacji.

3. Usługi zewnętrzne

Szereg możliwości oferuję nam typ ekranu zawierającego usługi zewnętrzne. Możemy wybrać od najprostszych typu przekierowanie na daną stronę www lub wysłanie e-mail po rozwiązania dostarczane przez zewnętrzne usługi: ankieta – SurveyMonkey [8], organizacja wydarzenia – Eventbrite [9], Sklep – Shopify [10] oraz wiele innych. Wybierając dana usługę zewnętrzna zostajemy przekierowani na stronę dostawcy w celu sparametryzowania.

Podczas całego procesu towarzyszy nam przed każdym krokiem krótka wskazówka w jaki sposób wybrać element.



Rysunek 2. Proces wyboru typu ekranu wraz z szablonem oraz efekt końcowy ekranu typu lista i mapa. Źródło: opracowanie własne

Podgląd

Z dostępnego menu narzędzia możemy w każdej chwili wybrać podgląd tworzonego rozwiązania bez konieczności generowania finalnego produktu.

Udostępnianie aplikacji

W celu ukończenia aplikacji należy wybrać opcje udostępnienia po której zostaniemy przeniesieni na stronę internetową narzędzia z możliwością instalacji. Finalny produkt nie jest rozwiązaniem natywnym, lecz skrótem do aplikacji mobilnej webowej, którą z łatwością możemy udostępnić innym osobą poprzez przesłanie wygenerowanego linku.

Podsumowanie

APPER [1] oferuję stworzenie w prosty sposób aplikacji informacyjnej np. dla wydarzenia, w której możemy zawrzeć harmonogram, mapę z lokalizacja jak również film promocyjny. Szereg usług zewnętrznych możliwych do podłączenia jest niewątpliwa zaleta, ponieważ przyczynia się do wykroczenia tylko poza funkcje informacyjne. Narzędzie ma przemyślany i intuicyjny interfejs użytkownika, lecz sposób zaimplementowania, stylu i rozmiaru niektórych kontrolek przyczynia się do wrażenia toporności i zacinającego się co chwila oprogramowania.

Rozwiązania poza usługami zewnętrznymi nie oferuję innej możliwości zapisu czy edycji danych przez użytkownika końcowego.

2.2. App Builder - Create own app (FREE App maker)

App Builder [2] określana przez twórców jako pierwsze narzędzie do zbudowania swojej mobilnej aplikacji dla firmy w kilka minut. Do korzystania nie potrzebne są wiedza techniczna oraz umiejętności kodowania. Oprogramowanie umożliwić ma wykreowanie pięknie wyglądającej i funkcjonalnej aplikacji.

App Builder pobrano ponad 1 mln razy i oceniono na 4,6 w skali 5. [2]

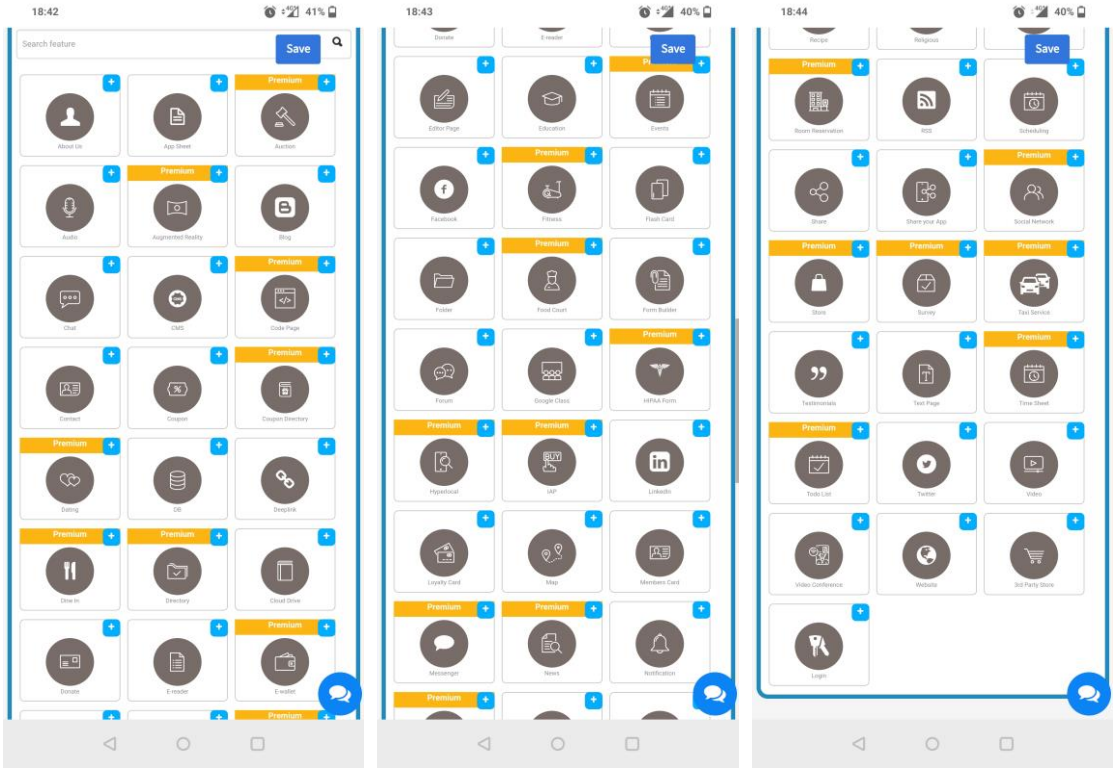
Dostępne niektóre funkcje:

- Dostarczenie pliku *.apk
- Wiele kategorii aplikacji do wyboru
- Motywy dostosowane do klienta
- Możliwość natychmiastowych aktualizacji do użytkownika końcowego
- Czat

Wymagane założenie konta oraz podpięcie rachunku PayPal [5] przed jakimkolwiek skorzystaniem z aplikacji.

Proces tworzenia

Tworzenie naszego rozwiązania rozpoczyna się od konwersacji z sztucznym asystentem za pomocą której ustawiamy parametry jak nazwa aplikacji, kategoria, funkcjonalności z pokaźnej listy, styl oraz motyw. Na podstawie kategorii asystent dobiera nam ikonkę. Po wybraniu powyższych rzeczy przechodzimy do dostosowania rozwiązania, które możemy podzielić na trzy sekcje.



Rysunek 3. Dostępne funkcje do wybrania

1. Funkcjonalności

Najbardziej rozbudowana sekcja narzędzia z niezliczonymi opcjami konfiguracji. Poprzez wybranie danej funkcjonalności otrzymujemy szablon ekranu możliwy do edycji oraz konfigurację usługi bądź też funkcji.

Opis kilku wybranych funkcji/ ekranów:

a. Strona „O nas”

Ekran zdefiniowany za pomocą gotowego szablonu z możliwością edycji tylko treści m.in. lista członków, opis czy godziny kontaktu.

b. DB – baza danych

Funkcja udostępniająca opcje połączenia z bazą danych stworzoną przez użytkownika ręcznie w systemie chmurowy Firebase [6]. Cała zawartość tabeli zostanie wyświetlona użytkownikowi końcowemu, lecz bez możliwości ich edycji. Modyfikacja jest dostępna tylko z poziomu panelu administracyjnego narzędzia. Administrator ma możliwość określenia kolumn tabeli. Każda kolumna posiada swoją nazwę, typ danych (tekst, telefon, adres URL itp.), określenie czy jest wymagana oraz opcje wyłączenia wyświetlania zawartości danej kolumny użytkownikowi końcowemu.

c. Formularz

Możliwość stworzenie formularza z pytaniami, polami otwartymi do uzupełnienia przez użytkownika końcowego. Wypełniony formularz po zatwierdzeniu będzie wysyłany na adres e-mail jaki określimy.

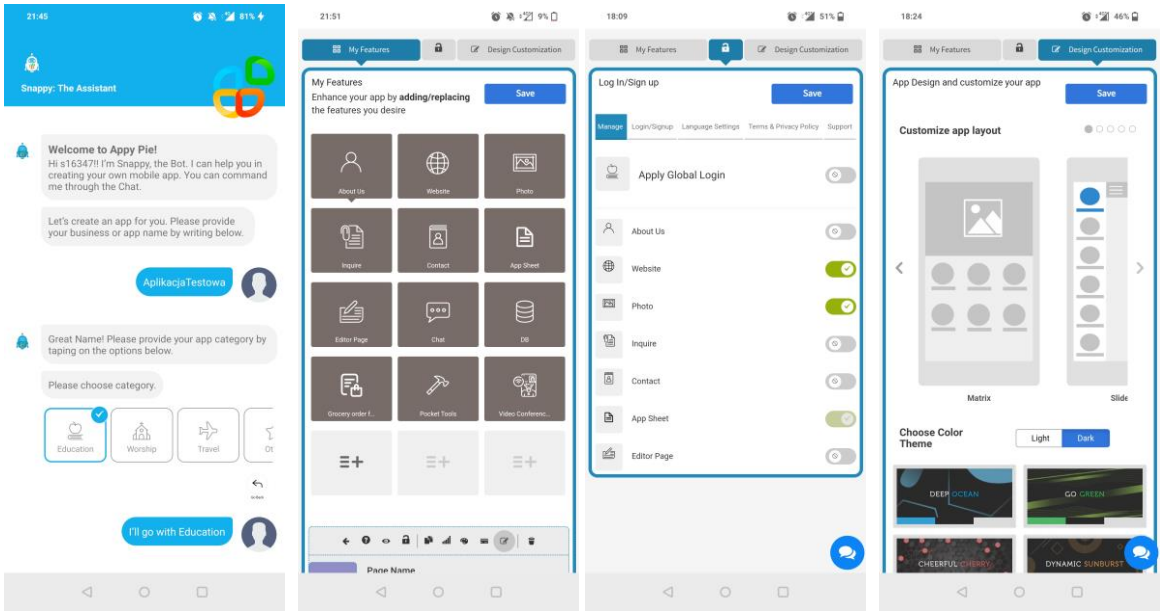
2. Ustawienia prywatności i językowe

Ta sekcja pozwala na wprowadzanie do naszego końcowego rozwiązania mechanizmu autoryzacji użytkowników. Określić możemy pojedyncze ekrany do których dostęp mogą mieć tylko użytkownicy zalogowaniu bądź też do całej aplikacji końcowej. Formularz logowania/rejestracji nowego użytkownika pozwala na dodanie dodatkowych pól poza podstawowymi jak nazwa użytkowania, adres e-mail i hasło.

Kolejnymi opcjami są ustawienia języka, gdzie możemy ręcznie przetłumaczyć na wybrany przez siebie język ekrany logowania, które generowanie są z gotowego szablonu oraz zdefiniować regulamin i politykę prywatności wyświetlaną użytkownikom końcowym.

3. Wygląd

Zakładka udostępnia nam możliwości zmodyfikowania głównego wyglądu, wybranego wcześniej, z kilku popularnych szablonów oraz motywu. Dodatkowymi cechami, którymi jesteśmy w stanie manipulować to styl czcionki, wyrównanie prezentowanych treści, kolory przycisków oraz inne wpływające na końcowym wygląd istotne wartości. Opcjonalnie istnieje możliwość skonfigurowania ekranu powitalnego (ang. *splash screen*)



Rysunek 4. Proces definiowania aplikacji. Ustawienie wartości podstawowych, zarządzanie funkcjami, prywatnością, wyglądem. Źródło: opracowanie własne

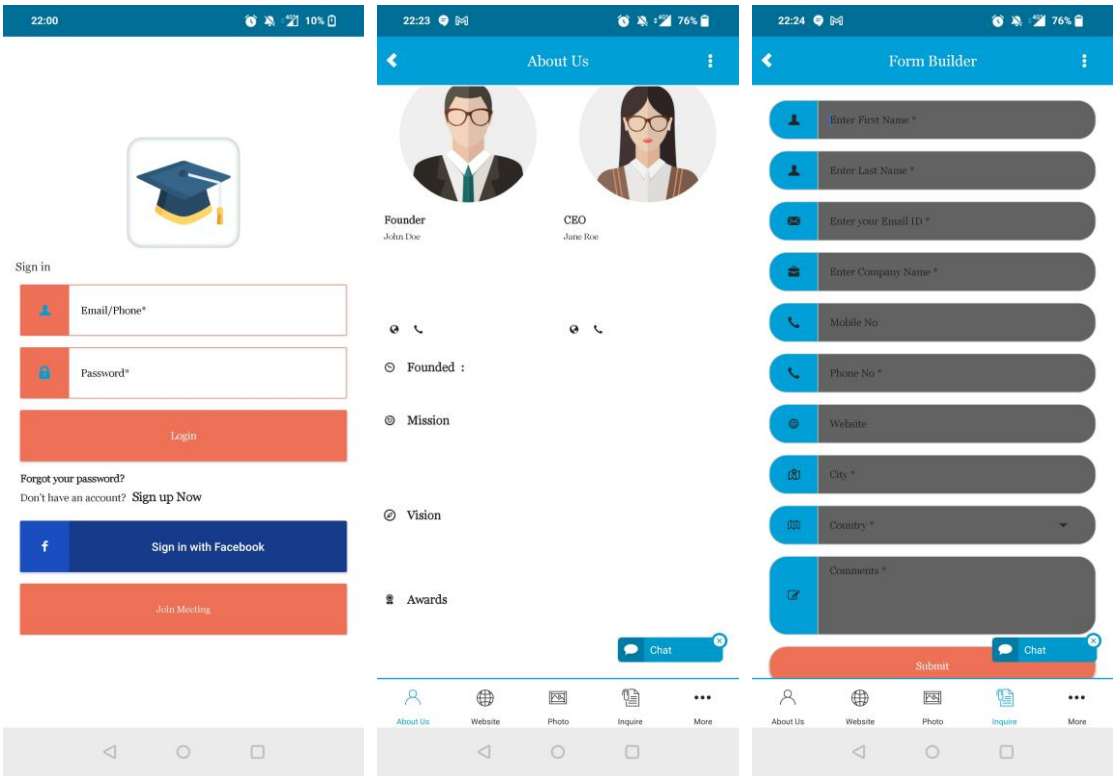
Podgląd

W celu uzyskania podglądu tworzonej aplikacji należy pobrać dodatkową aplikacje Test Lab App [7] gdzie musimy podać adres e-mail wykorzystany do rejestracji konta, po podaniu którego otrzymujemy dostęp do stworzonego rozwiązania w celach przetestowania.

Udostępnianie aplikacji

Istnieje możliwość wygenerowanie pliku *.apk jak również opublikowanie w katalogu Sklep Play tylko i wyłącznie dla klientów opłacających miesięczna subskrypcje serwisu.

Po wydaniu końcowej aplikacji narzędzie jakim jest *App Builder* [2] służy nam jako panel administracyjny w którym dowolnym momencie możemy nanosić poprawki i zmiany do naszej aplikacji oraz zarządzać użytkownikami. Interfejs administratora zezwala również na wysyłanie powiadomień do użytkowników końcowych, reklamowanie produktu na platformach społecznościowych, analizowanie statystyk oraz włączenie wyświetlanie reklam w celu uzyskania przychodu.



Rysunek 5. Wygenerowana aplikacja. Ekran logowanie, strony informacyjnej oraz formularz

Podsumowanie

Z cała pewnością *App Builder* [2] można określić jako platforma nie tylko do stworzenia swojej własnej aplikacji, ale także do późniejszego zarządzania nią z poziomu panelu administratora. Narzędzie udostępnia szereg opcji możliwych do modyfikacji w wielu aspektach takich jak szczegóły wyglądu, polityki dostępności do zasobów a także synchronizacji z usługi zewnętrznymi. Niewątpliwą cecha wyróżniającą rozwiązanie jest opcja konfiguracji mechanizmu autoryzacji użytkowników. Jednym z największych minus jest interfejs graficzny, choć przemyślany i podzielony na sekcje od siebie odizolowane to kontrolki przeznaczone do wpisania czy wybrania wartości zbyt małe w porównaniu do innych. Przy dużej ilości opcji do manipulacji interfejs może sprawić, że użytkownik poczuje się zagubiony.

Program nie udostępnia opcji modyfikacji danych w bazie przez użytkownika końcowego.

2.3. MobEasy : Create Mobile Apps without coding

Autor *Mobeasy* [3] określa tworzenie aplikacji jako proste i łatwe za pomocą jego narzędzia. Opisuje je jako pierwsze rozwiązanie pozwalająca na uzyskanie własnej aplikacji natywnej bez umiejętności kodowania zarówno dla biznesu jak i na własny prywatny użytek.

Mobeasy pobrano ponad 500 tys. razy oraz oceniono na 4 w skali 5 przez użytkowników. [3]

Według twórcy za pomocą jego narzędzia jesteśmy w stanie stworzyć:

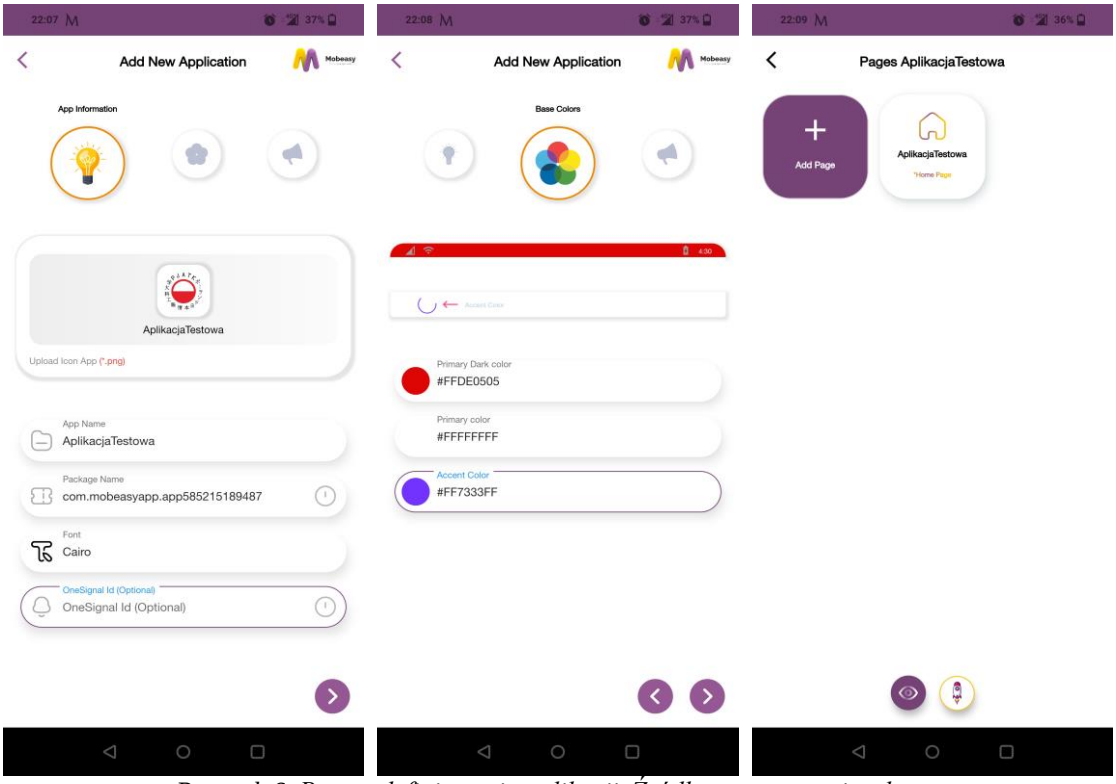
- Skomplikowana wieloekranową aplikacje
- Aplikacje muzyczna
- Gry mobilne takie jak: quiz, wisielec
- Galerię

Przed przystąpieniem do tworzenia wymagana jest wcześniejsze rejestracji użytkownika za pomocą e-mail.

Proces tworzenia

Proces definiowania rozpoczyna się od podania podstawowych informacji taki jaki: nazwa aplikacji, domyślny styl czcionki oraz opcjonalnie identyfikatora do OneSignal [4], narzędzia do wysyłania powiadomień do użytkowników. Na tym etapie można również określić ikonę wybierając ją z naszej galerii zdjęć w smartfonie.

Kolejnym krokiem jest ustawienie domyślnych kolorów głównego, pomocniczego oraz dodatkowego. Kolor możemy podać w postaci szesnastkowej lub wybrać z palety kolorów. Zmiany są prezentowane na podglądzie, przez co widzimy na jakie elementy dany typ koloru wpływa.



Rysunek 6. Proces definiowania aplikacji. Źródło: opracowanie własne

Następnie dodajemy ekrany według własnego pomysły. Do wyboru mamy następujące szablony:

- Pusta strona
- Galeria
- Ekran z przyciskami w postaci obrazków
- Sklep
- Odtwarzacz audio lub video
- Strona informacyjna
- Ekran z listą przycisków
- Quiz
- Wisielec
- Ekran z cytatami

- Odtwarzacz muzyki
- Ekran wykorzystujący język znaczników HTML

W zależności od wybranego szablonu możemy umieszczać na ekranie różne elementy, definiować ich właściwości oraz ich zawartość. Poniżej opis kilku wybranych ekranów:

1. Pusta strona

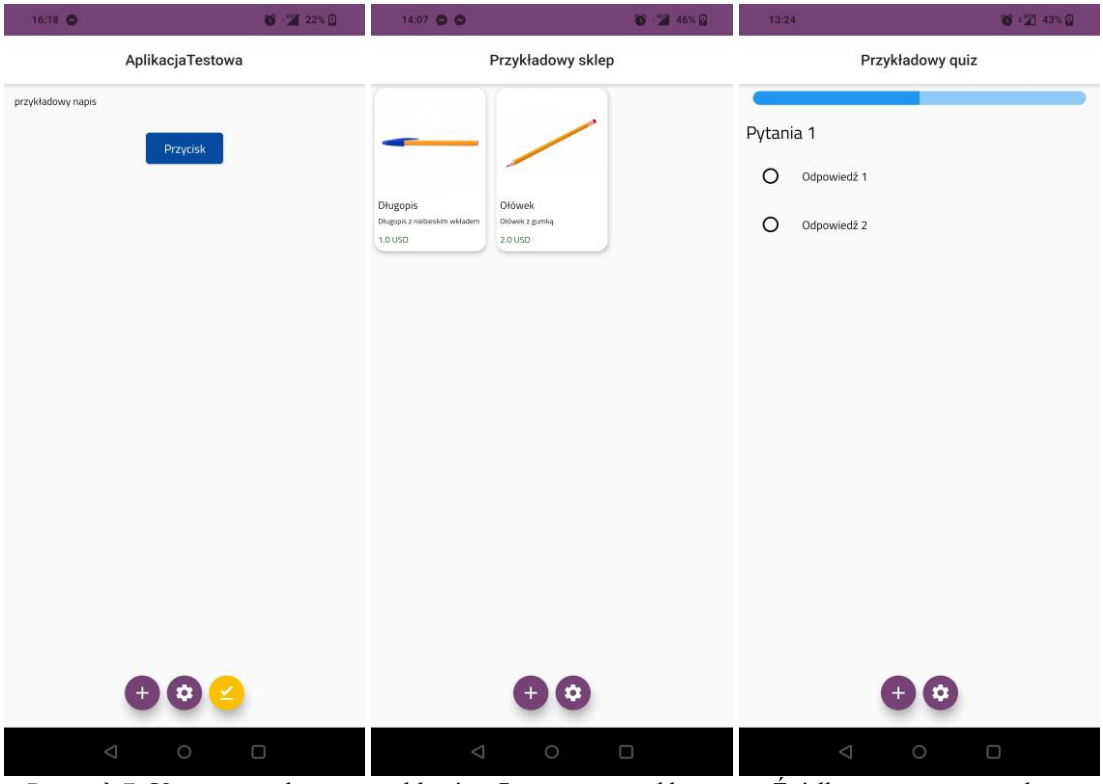
   Ekran ten zezwala na rozmieszczenie elementów bez możliwości przypisania akcji takich jak: tekst, film bądź pliku audio. Możemy również dodać przyciski, boczne wysuwane menu (ang. *drawer*) do których możemy przypisać takie akcje jak: otwarcie innego ekranu, otworzenie strony www w przeglądarce, zadzwonienia.

2. Sklep

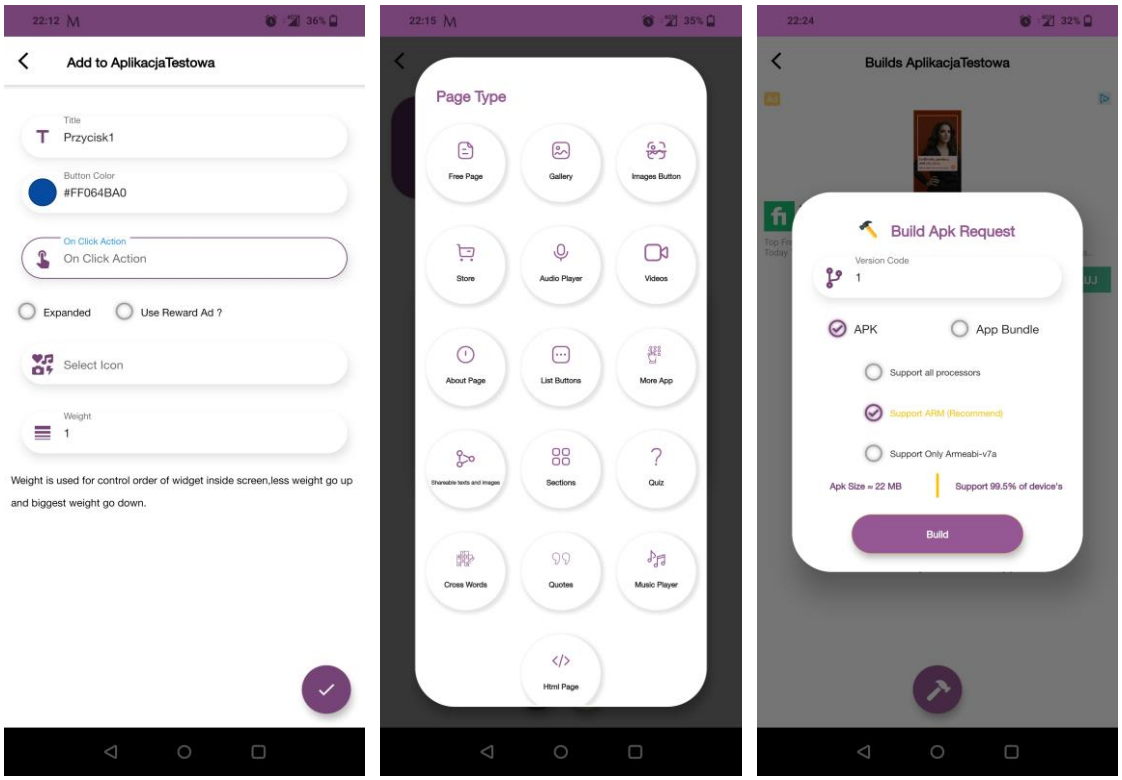
   Ekran umożlwiający stworzenie katalogu produktów. Każdy dodany produkt wymaga obrazu, nazwę, cenę z walutą oraz opis. Tak stworzona strona ma wyłącznie charakter informacyjny, ponieważ nie da się określić jakichkolwiek akcji jak np. złóż zamówienie.

3. Quiz

   Dla ekranu tego typu wymagane jest podanie pytań wraz z odpowiedziami jakie będą wyświetlane w sposób domyślny bez możliwości ingerencji przez użytkownika. Po wykonaniu quizu wyświetlane są statystki poprawnych i błędnych odpowiedzi.



Rysunek 7. Utworzone ekrany z szablonów: Pusta strona, sklep, quiz. Źródło: opracowanie własne



Rysunek 8. Proces definiowania aplikacji. Właściwości przycisku, określanie typu ekranu, tworzenie instalatora. Źródło: opracowanie własne

Podgląd

Rozwiązanie posiada opcje podglądu aplikacji w każdym momencie kreowania dostępne poprzez przycisk.

Udostępnianie aplikacji

Po ukończeniu określenia wszystkich aspektów mamy możliwość wygenerowania instalatora naszej aplikacji w postaci pliku *.apk jak również *.abb. Taki plik jesteśmy w stanie udostępnić znajomym lub opublikować w Sklep Play.

Podsumowanie

Podsumowując *Mobeasy* [3] ma nowoczesny oraz intuicyjny layout jak również proces definiowania aplikacji. Użytkownik jest prowadzony krok po kroku, a w miejscach tego wymagających umieszone są informację z czym związana i jak wpływa na wygląd np. dana właściwość kontrolki. Przygotowane gotowe szablony ekranów zaspokoją z pewnością większość osób chcących stworzyć ładna aplikacje. której głównym zadaniem jest prezentowanie informacji jak również utworzenie gier jakich jak quiz lub wisielec z własnymi zagadnieniami. Niewątpliwą zaleta jest też możliwości dodawania własnego kodu HTML dla osób zaawansowanych. Aplikacja ta wymaga połączenia sieciowego do działania.

Program pozbawiony jest funkcjonalności dzięki którym wygenerowana aplikacja zezwalałaby zapisywać dane podane przez użytkownika oraz współdzielenia ich z innymi.

2.4. Podsumowanie

Wszystkie porównane rozwiązania mają na celu dostarczyć możliwość stworzenia aplikacji mobilnej dla osób bez wiedzy programistycznej i technicznej na ten temat. Każda z nich jest odpowiednim narzędziem jeśli chcemy stworzyć aplikacje informacyjna np. na temat wydarzenia czy organizacji. Tworzenie oparte głównie jest na szablonach ekranów i przypisanych do nich nierozłączalnych funkcjonalności. Każdy ekran jest oddzielna funkcją, która nie ma swojego rozszerzenia i możliwości przejścia do kolejnych stron. Jakiekolwiek opcje edycji danych czy też współdzielenia są dostarczane za pomocą usługi zewnętrznych wymaganych do sparametryzowania przez użytkownika. Tylko jedno z rozwiązań dostarcza mechanizm autoryzacji dla aplikacji końcowej. Graficzne interfejsy użytkownika wydają się być przemyślane choć nie do końca zawsze zoptymalizowane pod względem działania jak i wyglądu pojedynczych elementów.

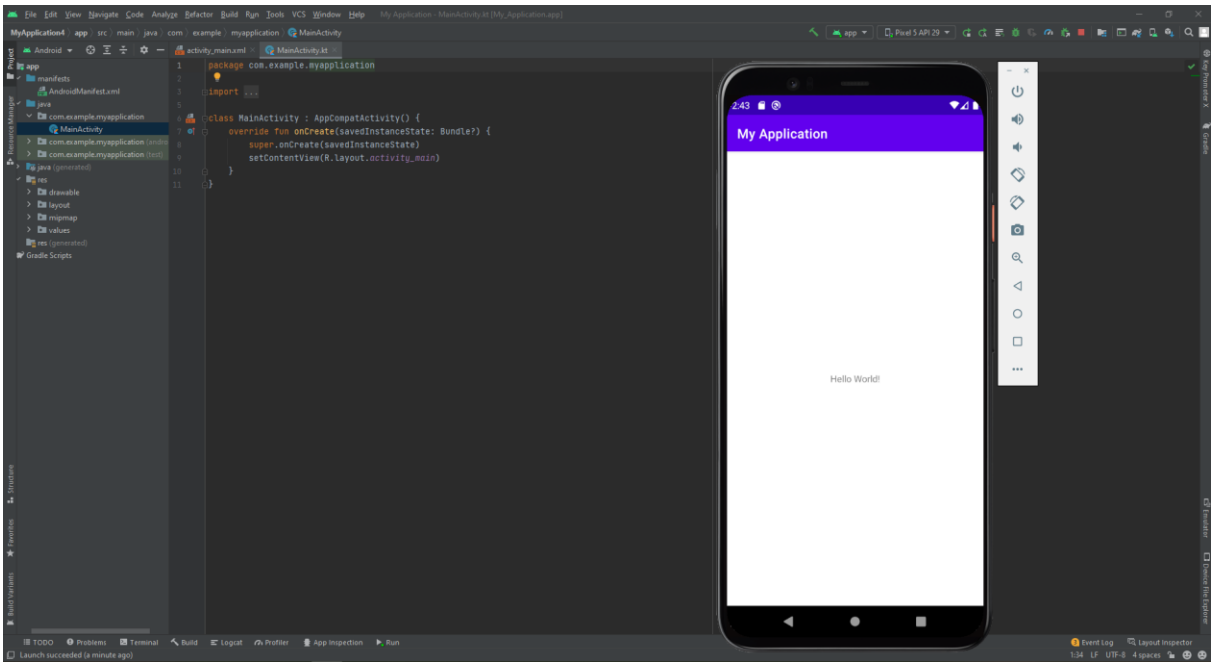
Żadne z narzędzi nie dostarcza opcji zarządzania danymi bezpośrednio z poziomu aplikacji końcowej przez użytkowników, tak aby mogli oni zarządzać współdzielonymi danymi i synchronizować je.

3. Opis narzędzi oraz technologii zastosowanych w pracy

W tym rozdziale opisano technologie oraz narzędzie zastosowane do stworzenia prototypu aplikacji.

3.1. Android Studio

W celu zrealizowania prototypu wykorzystano środowisko programistyczne Android Studio [12]. Jest to darmowe narzędzie do tworzenia aplikacji natywnych na platformę mobilną Android. Zostało zbudowane na podstawie jednego z najbardziej popularnych IDE wśród programistów Javy IntelliJ IDEA od firmy JetBrains [13]. Oficjalna prezentacja nastąpiła w 2013 roku i początkowo środowisko obsługiwało języki C++ oraz Javę, a od roku 2019 również Kotlin. Narzędzie jest ciągle rozwijanie i rozszerzane o nowe funkcjonalności. Wymagane jest w celu poprawnego działania, aby komputer posiadał procesor wspierający wirtualizację oraz minimum 8GB pamięci RAM, lecz zalecane jest więcej z względu na obciążenia podczas używania m.in. dostępnych emulatorów.



Rysunek 9. Android Studio. Źródło: opracowanie własne

Android Studio zawiera wiele funkcji i narzędzi ułatwiających programowanie aplikacji mobilnych i zwiększających produktywność podczas pracy [14]:

- Wbudowany emulator pozwalający na uruchomienie i przetestowanie dotychczasowej implementacji bez posiada fizycznego urządzenia. Opcje zezwalają na konfiguracje wirtualnych urządzeń takich jak: smartfony, smartwatche, tv czy tablety z wskazaną wersja systemu Android. Sam emulator udostępnia funkcje jakimi dysponujemy w fizycznych telefonach jak: lokalizacja, obracanie ekranu czy symulacja aparatu fotograficznego.
- Integracje z narzędziem Gradle z specjalną wtyczką dla Android z dodatkowymi możliwościami
- Debugger pomagający również analizować wydajności napisanego kodu.

- Wbudowane wparcie dla połączenia i integracji z rozwiązaniami chmurowymi od firmy Google jak Google Cloud Platform czy Firebase Cloud.
- Integrację z GitHub pomagająca importować przykładowy kod wskazanych rozwiązań.

3.2. Kotlin

Głównym językiem programowania użytym w pracy magisterskiej do zaimplementowania prototypu jest Kotlin [15]. Jest to statycznie typowany język opracowany przez firmę JetBrains [13] i rozwijany od 2011 roku. Został stworzony z powodu braku dostępności na rynku języka programowania zawierającego pożądane cechy przez twórców środowiska IntelliJ IDEA [16]. W społeczeństwie programistów coraz częściej określany jako następca Javy, z która jest w pełni interoperacyjny, ponieważ sam działa na maszynie wirtualnej Javy. Fakt ten pozwala na migracje kodu etapami bez konieczności przepisania aplikacji jako całość lub dodawanie nowych funkcjonalności już w nowym języku. Według autorów rozwiązuje kilka problemów, które występuję w Javie [17] m.in.:

- Kontrolowanie wartości null przez system typów zmiennych.
- Niezmienność tablic
- Brak wyjątków sprawdzanych na etapie kompilacji

Język od firmy JetBrains zawiera szereg udogodnień dla programistów, których nie zawiera Java. Poniżej opisano kilka z nich [17]:

- Funkcje rozszerzalne. Zapewnienie możliwości rozszerzenia klasy lub interfejsu o nowe metody bez konieczności dziedziczenie czy rozszerzania klasy we wewnątrz. Funkcja taka zawiera typ obiektu w swojej semantyce na jakim może zostać wywołana.
- Inteligentne rzutowanie. Kotlin śledzi sprawdzenie typów zmiennych i automatycznie potrafi je prze rzutować do wybranego inne typu jeśli jest to konieczne.
- Typy danych mogące przyjmować wartość null lub nie. Funkcjonalność język pozwalająca na łatwe kontrolowanie w kodzie braku wartości poprzez specjalny operator *„?”* wykonujący kod tylko jeśli wartość została przypisana.
- Oddzielne interfejsy do kolekcji niemodyfikowalnych i modyfikowalnych. Dzięki temu programista w łatwy sposób może utworzyć kolekcje jakiej potrzebuje w danym rozwiązaniu.

Kotlin od 2019 roku jest oficjalnym językiem programowania natywnych aplikacja na platformę Android, gdzie zastąpił Javę. Firma Google uzasadniając swój wybór odwołała się do czterech głównych argumentów [18]:

- Produktywność. Nowy język pozwala na osiągnięcie tych samych rezultatów z mniejszą ilością kodu dzięki swojej zwięzłości.
- Mniejsza awaryjność aplikacji, ponieważ zastosowano wiele funkcji pomagających uniknąć typowych błędów programistycznych.
- Interoperacyjność. Możliwość zastosowanie języka Kotlin i Javy jednocześnie.
- Współbieżność. Prosta praca z wielowątkowością pozwalająca na uproszczenie tworzenie asynchronicznego kodu.

3.3. Jetpack Compose

Jetpack Compose [20] jest nowoczesnym podejściem do implementacji warstwy interfejsu użytkownika w aplikacjach natywnych na platformę Android. Pierwszą stabilną wersje firma Google wydała w 2021 roku. Znacząco upraszcza i przyspiesza on pracę programistą zastępując poprzednie rozwiązanie, w którym widoki w aplikacjach mobilnych były definiowane za pomocą języka znaczników XML. Jetpack Compose pozwala na definiowanie interfejsu w sposób deklaratywny używając języka Kotlin. Programista jest w stanie w sposób wstępny zdefiniować elementy ekranu, które automatycznie wykrywają zmiany zachodzące w połączonych danych. Takie rozwiązanie pozwala na zwiększenie produktywności i zmniejsza ilość kodu [21]. Nie trzeba implementować obsługi danych kontrolek i ich zachowania w trakcie zmiany zawartości danych do wyświetlenia. Używanie tego samego języka Kotlin do definiowania interfejsu graficznego oraz do logiki biznesowej czy też wymiany danych z serwerem pozwala na łatwa komunikację między tymi warstwami bez używania dodatkowych bibliotek.

Poniżej w Listing 1 przedstawiono przykład z użycia opisywanej technologii. Rozwiązanie daje nam możliwość definiowania zwykłych funkcji, które poprzez oznaczenia adnotacja @Composable mogą zostać użyte w widoku. Metody takie można używać wielokrotnie w innym oznaczonych tą samą adnotacja. Parametr wejściowy „text" typu String przekazany jest do innej funkcji dostarczonej przez twórców, która wygeneruj nam napis. Funkcje takie dostarczają opcjonalne argumenty dzięki którym można modyfikować właściwości m.in: pozycje, kolor czy wielkość. Na poniższym przykładzie oprócz przekazanego tekstu ustawiono również położenie tekstu i kolor. Aby użyć zdefiniowanego elementu wystarczy wywołać funkcję podając wartość tekstu jaka ma być wyświetlana. W przypadku przekazania zmiennej będzie ona śledzona i jej zawartość aktualizowana na ekranie. Dzięki takiemu podejściu programista przekazuje bezpośrednio dane bez potrzeby łączenia kontrolek z widoku z zmiennymi w języku programowania.

Listing 1. Przykładowa kontrolka stworzona przy użyciu Jetpack Compose. Źródło: opracowanie własne

```
@Composable
fun HeaderStepDescription(text: String){
        Text(text = text, textAlign = TextAlign.Center, color = Color.Black)
}
```

3.4. Firebase

Firebase [6] jest to platforma przeznaczona do pomagania w tworzeniu i rozwijaniu gier oraz aplikacji mobilnych. Utworzona przez firmę o tej samej nazwie w 2011 roku, a następnie przejęta przez Google trzy lata później [22]. W tradycyjnym podejściu tworząc aplikację, której dane są współdzielone wymagane jest utworzenie oprogramowania działającego na serwerze, skonfigurowanie własnej bazy danych i zapewnić ujednolicone API. Firebase oferuję szereg usługi chmurowych dzięki którym można zbudować aplikacje mobilną bez budowania własnego rozwiązania serwerowego. Takie rozwiązania znane są jako *MBaaS* [23]. Programista może skupić się tworzeniu oprogramowania przeznaczonego na smartfony tylko konfigurując wybrane usługi. Dostarczane usługi można podzielić na trzy kategorie z względu na przeznaczanie:

- Do budowania aplikacji. Najbardziej obszerny zestaw narzędzi. Zawiera następujące usługi [25]:
    - Realtime Database – przechowywanie i synchronizacja danych w formacie JSON
    - Cloud Firestore – nowsza i ulepszona wersja usługi Realtime Database. Oferuje szybsze i bogatsze zapytania [24].

- Remote Config – umożliwia wysyłanie zmiany konfiguracji bez potrzeby aktualizacji aplikacji.
- Firebase Extensions – dodatkowe funkcje po stronie Firebase np. wysyłanie wiadomości e-mail, usuwanie czy eksportowanie danych.
- App Check – warstwa bezpieczeństwa śledząca i nadzorująca ruch sieciowy pochodzący z aplikacji mobilnej.
- Cloud Functions – możliwość programowania dodatkowych funkcji uruchomianych w odpowiedzi na dany wyzwalacz.
- Authentication – system autoryzacji i logowania.
- Cloud Storage – przechowywanie danych multimedialnych dostarczonych przez użytkowników z aplikacji np. zdjęcia.
- Hosting – pozwala na wdrożenie małej aplikacji sieciowej lub strony internetowej docelowej aplikacji.
- Cloud Messaging – usługa do zarządzanie wysłaniem powiadomień do użytkowników końcowych na smartfonach.
- Firebase ML – usługa w wersji beta przeznaczona do uczenia maszynowego.

- Do utrzymania oraz wydawania wersji. Usługi z tej kategorii pozwalają m.in. na analitykę wykorzystywania aplikacji przez użytkowania, planowania nowych funkcjonalności i monitoring wydajności [26].
- Optymalizacji biznesowej [27].

Niniejsza praca magisterka wykorzystuje tylko usługi z kategorii przeznaczonej do zbudowania aplikacji.

3.5. Gradle

Gradle [28] jest zaraz po Apache Maven [29] jednym z najpopularniejszych narzędzi, które automatyzuje budowania oprogramowania. Przeznaczony jest do wszystkich języków programowania opartych o wirtualna maszynę Javy. Jest odpowiedzialny za kompilację kodu, uruchamianie testów, tworzenie dokumentacji, pobieraniu zależności oraz wielu innych czynności. Do definiowania zadań i zależności używa się języka domenowego – DSL opartego na języku Groovy-m lub w najnowszych wersjach Kotlin-a [30]. Android Gradle plugin jest wtyczką rozszerzającą Gradle dostarczana razem z Android Studio. Oferuje kilka funkcji specyficznych umożliwiających budowania aplikacji mobilnej na ten system.

3.6. Dagger Hilt

Dagger [31] jest to framework dla aplikacji serwerowych dostarczający wzorzec projektowy i architektury jakim jest wstrzykiwanie zależności [32]. Początkowo stworzony przez firmę Square [33], a obecnie utrzymywany przez Google. Jest w pełni statyczny i działa w czasie kompilacji programu. Takie rozwiązanie ma być alternatywną do podobnych rozwiązań, gdzie działanie oparte jest na refleksach i powoduje problemy z wydajnością. Z względu na te cechy stał się standardem przy implementacji wstrzykiwania zależności na Android. Dagger Hilt jest to rozszerzenie freamworku przygotowany specjalnie dla programistów tworzących aplikacje mobilne. Został utworzony, ponieważ stosowanie podstawowej wersji Dagger-a wymagało dużej ilości powtarzalnego kodu tworzonego przez programistę z względu na rozbudowany system Android-a. Korzystając z specjalnie

przygotowanych adnotacji Dagger Hilt [34][35] automatycznie generuje komponenty z bazowego frameworku Dagger. Programista może skupić się dzięki temu w swojej pracy nad tym gdzie i jakie obiekty wstrzyknąć.

4. Propozycji kreatora aplikacji mobilnych

Rozdział ten został poświęcony szczegółowemu opisowi implementacji prototypu. Pierwsza część opisuje wymagania jakie powinien spełniać oraz propozycje rozwiązania. Kolejne punkty przedstawiają jak zbudowana jest zaproponowana aplikacja mobilna oraz w jaki sposób zapisuje i interpretuje pozyskane od użytkownika dane. Następnie zaprezentowane są główne i najciekawsze funkcjonalności kreatora wraz z efektem jaki można uzyskać. Każda funkcjonalność podsumowana jest o swoje ograniczenia oraz możliwości rozwoju.

4.1. Postawione wymagania dla nowego rozwiązania

Pomysłem autora po własnych rozważaniach oraz po zapoznaniu się z istniejącymi rozwiązaniami na rynku jest kreator ukierunkowany na tworzenie prostych aplikacji. Wygenerowany produkt końcowy powinien naśladować aplikacje bazodanowe, czyli dostarczać interfejs graficzny do zarządzania danymi. Rozwiązanie nie powinno przekierowywać do zewnętrznych platform oferujących wymagane funkcjonalności. Oprogramowanie powinno składać się z części kreatora oraz aplikacji wygenerowanej jako końcowa.

Kreator

Część odpowiedzialna za uzyskanie od użytkownika informacji niezbędnych do utworzenia nowej aplikacji według określonych wymagań. Powinna starać się przekazać użytkownikowi nie technicznemu w sposób prosty i zrozumiały instrukcje jak uzupełnić kolejne kroki. Walidacje wprowadzanych danych powinny być sprawdzane w sposób szczegółowy blokując tym samym utworzenie niespójności końcowych parametrów. Proces tworzenia powinien być maksymalnie uproszczony aby nie wprawiać użytkowników w zakłopotanie nadmiarem opcji.

Wygenerowana aplikacja

Druga część oprogramowania, której zadaniem jest (na podstawie parametrów wprowadzonych w kreatorze) utworzenie działającej zgodnie z założeniami aplikacji końcowej. Ten sam zestaw parametrów powinien zawsze być interpretowany w taki sam sposób, dając ten sam powtarzalny efekt. Funkcjonalności jakie powinny być możliwe do uzyskania:

- Możliwość korzystania z aplikacji przez wielu użytkowników.
- Współdzielenie i synchronizacja danych między użytkownikami.
- Dane powinny być reprezentowane w sposób przybliżony do cech encji [36].
- Oferować działania CRUD [37].
- Oferować wyświetlenie danych w postaci listy.
- Grupy użytkowników.
- Proste opcje sparametryzowania wyglądu.
- Definiowanie pozycji dostępnych z menu.

Poniżej zamieszczono przykłady aplikacji jakie powinny być możliwe do utworzenia korzystając z tego rozwiązania:

- Aplikacja do rezerwacji biurek. W utworzonej aplikacji użytkownik po zalogowaniu mógłby wpisać datę, wybrać biurko oraz okres czasu w jakich rezerwuje wybrane przez siebie miejsce w biurze. Dla opisywanej aplikacji podczas tworzenia osoba musiałaby zdefiniować uprawnienia użytkowników (którzy mogliby modyfikować tylko dodane przez siebie instancje) oraz encje „Biurko" , „Rezerwacja" wraz z zdefiniowaniem ekranów do wykonywania operacji CRUD.
- Aplikacja do kontroli stanu produktów w małym magazynie wraz z ich lokalizacją. Utworzona aplikacja umożliwiałaby kontrolowanie stanu magazynu. Użytkownik mógłby dodawać nowe produkty do magazynu, kontrolować ich ilość oraz ew. usunąć ze stanu. Aplikacja wymagałaby encji „Produkt" oraz „Lokalizacja" wraz z ich powiązaniem oraz ekranów do operacji CRUD.
- Prosta aplikacja pozwalająca na zdefiniowanie listy zadań do wykonania z podziałem na kategorię. Przeznaczona tylko dla jednego użytkownika. Aby zagwarantować wymienione funkcje należałoby zdefiniować struktury „Zadanie" oraz „Kategoria".

4.2. Propozycja rozwiązania

Na podstawie zdefiniowanych wymagań poniżej przedstawiono propozycje rozwiązania. Została ona opisana z podziałem na poszczególne moduły oraz kluczowe aspekty.

4.2.1 Generowanie

Przed rozpoczęciem prac przeanalizowano możliwości od strony technicznej w jaki sposób wygenerować aplikacje końcową. Można wyodrębnić dwa główne podejścia. Pierwsze poprzez przygotowanie pliku APK (*ang. Anroid Package Kit)* [38] który można by zainstalować bezpośrednio na smartfonie. Aby uzyskać instalator aplikacji wymagane byłoby narzędzie umożliwiające kompilowanie projektu androidowego na telefonie. Alternatywną ścieżką mogłoby być przesłanie zestawu danych na serwer, który poprzez podstawienie danych do szablonu aplikacji wykonał by budowanie np. poprzez narzędzie Gradle i zwrócił wynik na telefon użytkownika. Drugim podejściem jest utworzenie oprogramowania interpretującego zapisane parametry na bieżąco dostarczając tym samym efekt końcowy.

W ramach proponowanego rozwiązania podjęto decyzje o zaimplementowaniu części generującej aplikacje w czasie działania. Głównymi powodami była implementacja nie wymagająca dodatkowych narzędzi programistycznych, języków programowania czy zaplecza serwerowego. Razem z częścią kreatora całość stanowi jedną aplikacje. Utworzony został panel użytkownika bezpośrednio uruchamiany po włączeniu. Z poziomu tego panelu można przejść do opcji utworzenia nowej lub włączenia już istniejącej aplikacji.

4.2.2 Kreator

Moduł odpowiedzialny za poinstruowanie osoby oraz zapisanie właściwości i parametrów aplikacji użytkownika. W proponowanym rozwiązaniu starano się z względu na wielkość ekranów smartfonów, aby wyświetlane kroki zawierały tylko pojedyncze ustawienia. Kolejne formularze do wypełnienia wyświetlają się po kolei utrzymując tym samym spójność parametrów i zapewnieniem, że żaden nie został pominięty nieświadomie. Dzięki temu użytkownik mógłby skupić się tylko na konkretnym zadaniu bez zakłopotania spowodowanego zbyt wieloma opcjami. Poniżej przestawiono podział na konkretne etapy i kroki wraz z krótkim opisem.

- Etap 1 – ustalenie głównych cech aplikacji
  - Krok 1 - Nazwa aplikacji. Zezwala użytkownikowi na wpisanie nazwy swojej aplikacji
  - Krok 2 – Wybór typu aplikacji. Aplikacja może być przeznaczona dla jednego lub wielu użytkowników.

    - Krok 3 – Grupy użytkowników. Określnie grup do jakich będą mogli należeć tworzeni użytkownicy. Grupy mogą być wykorzystywane do wprowadzenia ograniczeń.
- Etap 2 – Definiowanie struktur. Struktura ma w sposób zbliżony odzwierciedlać encje. Przed przystąpieniem do tego etapu możliwe jest wyświetlenie przykładu.
    - Krok 4 – Tworzenie pojedynczej struktury. Wymagane jest podanie nazwy oraz pól struktury wraz z ich właściwościami. W tym kroku można zdefiniować dowolną ilość struktur.
- Etap 3 – Relacje między strukturami i grupami użytkowników.
    - Krok 5 – Opcjonalne ustawienie relacji między utworzonymi wcześniej strukturami / grupami użytkowników.
- Etap 4 – Styl menu
    - Krok 6 – Pozwala na wybranie między dwoma stylami menu.
- Etap 5 – Ustalenie pozycji w menu oraz funkcji poszczególnych ekranów. Etap zezwala na dodawania przycisków do menu odpowiedzialnych za przejście do kolejnych ekranów. Każdy ekran będzie mógł pełnić tylko jedną funkcje: dodanie obiektu, edycja obiektu wraz z usunięciem, pogląd pojedynczego obiektu, lista obiektów.
    - Krok 7 – Wybranie funkcji dla danego ekranu oraz jego właściwości.
- Etap 6 – Zarządzanie użytkownikami
    - Krok 8 – Wybór opcji związanych z tworzeniem nowych kont.
- Etap 7 – Styl aplikacji
    - Krok 9 – Wybór koloru, kształtu przycisków oraz stylu czcionki.
- Krok 10 – Utworzenie konta dla twórcy aplikacji
- Krok ostatni zapisuje wszystkie wprowadzone właściwości w bazie danych. Od tego momentu aplikacja jest dostępna.

Kroki numer 3, 8 i 10 są dostępne tylko podczas tworzenia aplikacji dla wielu użytkowników.

4.2.3 Generator

Moduł programu odpowiedzialny za generowanie aplikacji końcowej. Główne trzy zadania jakie wykonuje tworząc całość to:

- Interpretacja parametrów uruchamianej aplikacji w celu zbudowania poprawnie interfejsu graficznego.
- Wykonywanie działań CRUD na danych. Zapewnienie spójności.
- Dostarczenie wszystkich funkcjonalności określonych w kreatorze.

Moduł nacechowany jest uniwersalnością metod i generycznością, ponieważ bazuje na parametrach zapisanych w ramach kreatora. Zasady jego działania i zastosowanych rozwiązań zostały przedstawione w kolejnych podrozdziałach.

4.2.4 Udostępnianie

Każda stworzona generowana aplikacja uzyskuje unikalny identyfikator nadany przez system. Jego wartość będzie można przekazać innym użytkownikom aplikacji w celu udostępnienia.

4.2.5 Modyfikacja

W ramach niniejszej pracy magisterskiej oferowane rozwiązanie nie zawiera możliwości modyfikacji wygenerowanej aplikacji po jej utworzeniu. Aktualizowanie parametrów przez użytkownika, który jest twórcą aplikacji jest jak najbardziej osiągalne i możliwe do zaimplementowania. Trzeba zwrócić uwagę, aby przy wdrożeniu takiej opcji zapewnić spójność parametrów aplikacji jak i już wprowadzonych danych. Brak jej może doprowadzić do błędów w działaniu końcowej aplikacji lub całkowitym jej zablokowaniu bez możliwości naprawienia. Rozważając sposoby zapobieganiu wspomnianej sytuacji może być zastosowanie rozwiązania kasującego wszystkie dane wprowadzone przez użytkowników aplikacji wygenerowanej. Zapewnienie migracji, wartości domyślnych lub zapobiegnięcie wykasowaniu istotnych danych np. przy usunięciu danego pola struktury może być trudne do zrealizowania. W dalszej części pracy rozważania na temat modyfikacji poszczególnych funkcjonalności nie będą poruszane.

4.3. Architektura rozwiązania

W tym podrozdziale zaprezentowano propozycje przechowywania danych oraz ogólnej architektury prototypu.

4.3.1 Podział kodu

Kod aplikacji, mimo iż stanowi jedną aplikacje został podzielony na trzy główne pakiety starając odseparować je od siebie, aby nie ingerowały i używały swoich elementów.

- Panel użytkownika (*pl.pjatk.board*)
- Kreator (*pl.pjatk.creator*)
- Generator (*pl.pjatk.generator*)

4.3.2 Baza danych

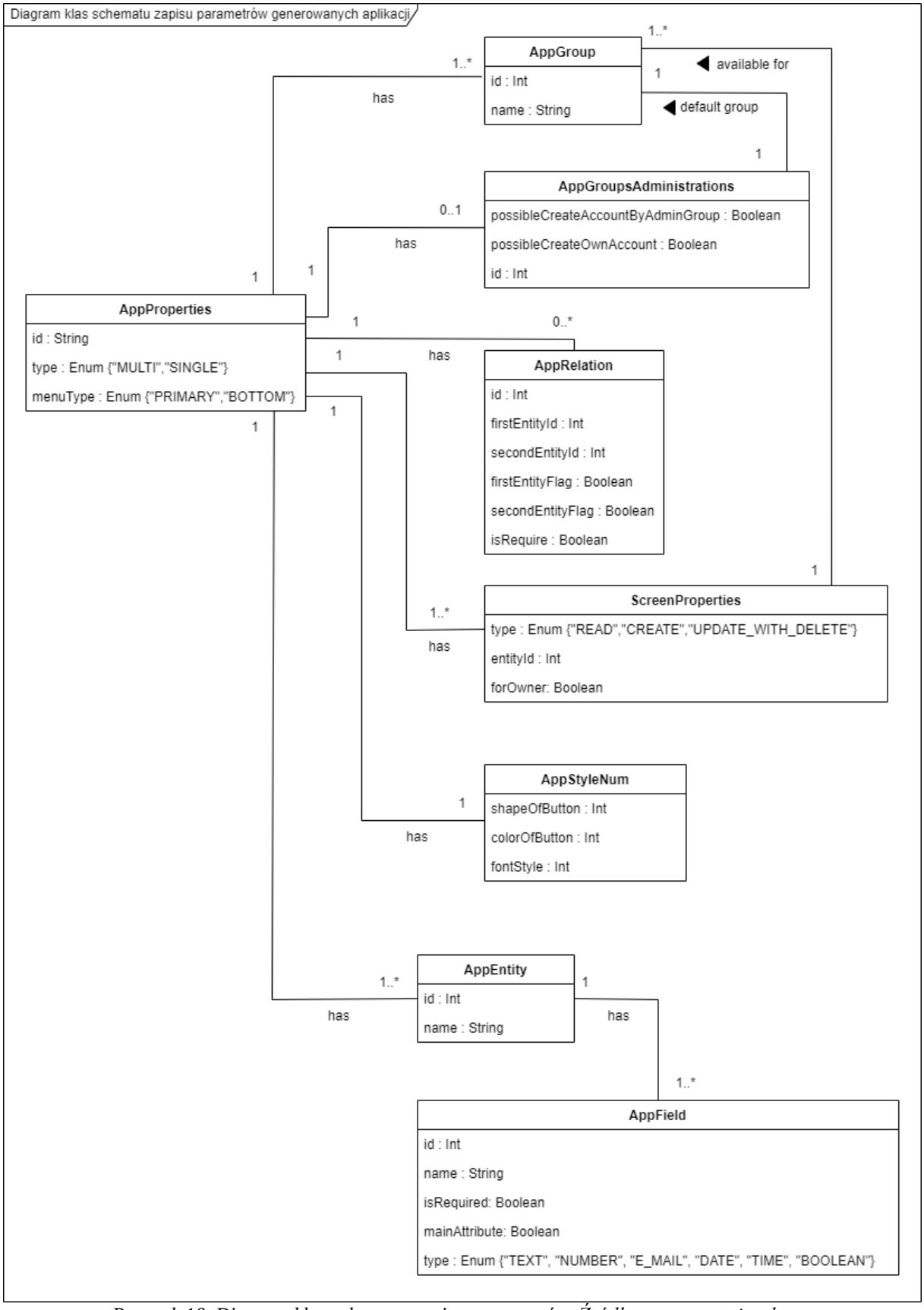
W celu zapewnienia współdzielenia i synchronizacji danych przez użytkowników skorzystano z trzech usług oferowanych przez platformę chmurową Firebase.

- Authentication – system autoryzacji wykorzystywany w prototypie przy rejestrowaniu i logowaniu się użytkowników do wejściowego panelu.
- Realtime Database – baza danych NoSQL zastosowana do przechowywania danych w formacie JSON wprowadzonych przez użytkowników w aplikacji końcowej.
- Firestore Database – baza danych NoSQL, której zadaniem jest przechowywanie konfiguracji generowanych aplikacji oraz zapisanie przypisanych do nich użytkowników.

Wykorzystanie dwóch podobnych usług jest spowodowany chęcią izolacji zestawów danych, aby nie wpływały na siebie podczas implementacji prototypu i testów. Dane wszystkich aplikacji są przetrzymywane w ramach jednego projektu Firebase, instancji stworzonej przez autora pracy magisterskiej. W ramach analizy rozważano również poinstruowanie użytkownika podczas tworzenia jak uruchomić własną usługę Firebase oraz przekazać dane autoryzacyjne. Zaletą takiego podejścia byłoby umożliwienie i zarządzanie wglądem do danych tylko przez użytkownika tworzącego nową aplikacje. Natomiast wadami mogłoby się okazać możliwość ręcznej modyfikacji tych danych przez osobę nie znającą struktur w jakich są przechowywane. Skutkiem mogłoby być nieumyślne zatrzymanie działania oprogramowania ponieważ parametry aplikacji końcowej nie pokrywały by się z zapisanymi danymi. Kolejną wadą byłby nadmierne skomplikowanie procesu zniechęcając użytkownika do jego zakończenia.

4.3.3 Parametry generowanych aplikacji

Parametry tworzonych aplikacji są zapisywane w bazie danych *Firestore Database* po przejściu przez użytkownika całego procesu. Parametry te są przechowywane według ściśle określonego schematu zaimplementowanego przy pomocy klas. Rysunek 10 prezentuje jego koncepcje wraz z objaśnieniem atrybutów w Tabeli 1.



Rysunek 10. Diagram klas schematu zapisu parametrów. Źródło: opracowanie własne

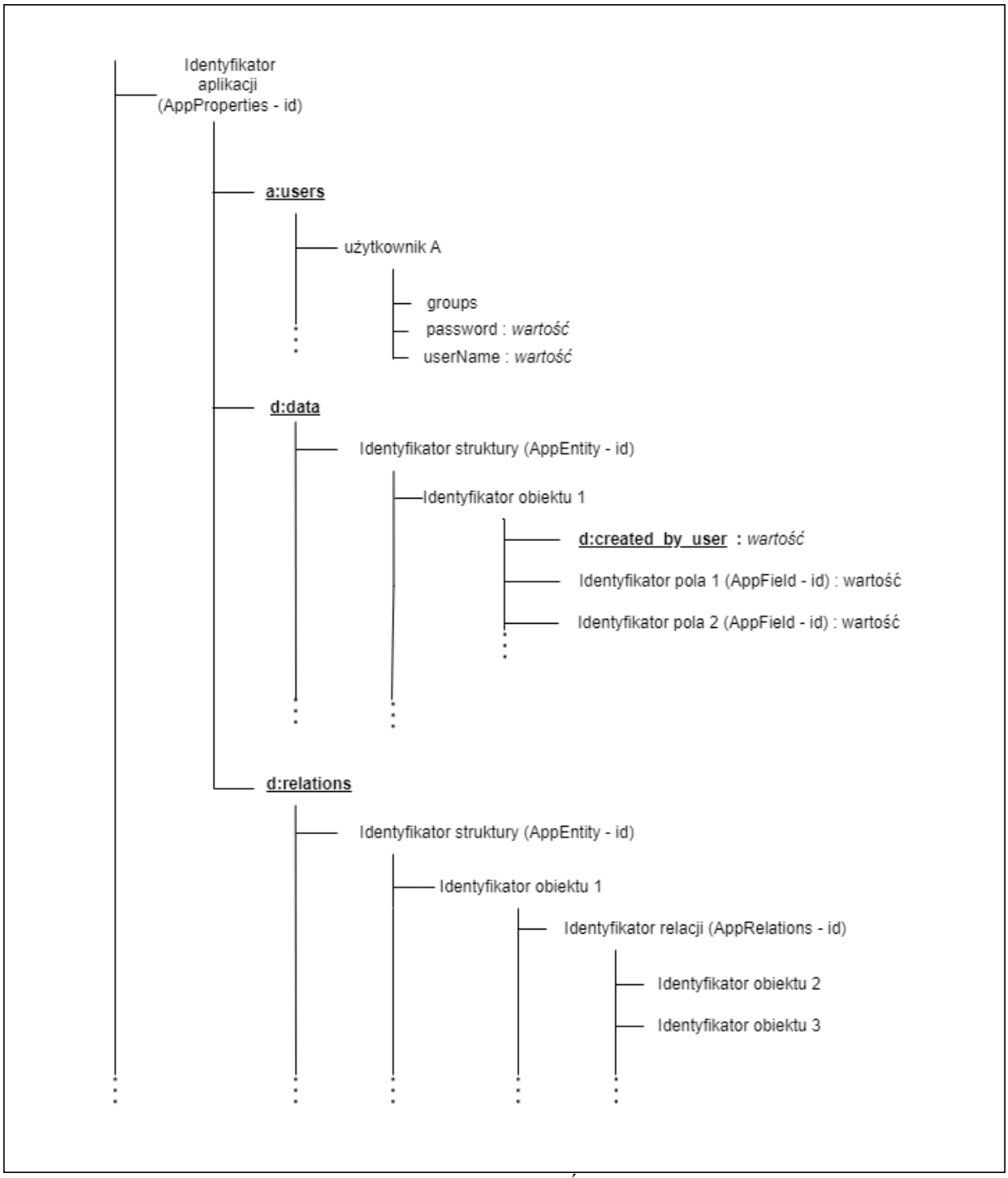
Tabela 1. Opis atrybutów schematu parametrów. Źródło: opracowanie własne

Nazwa atrybutu/ klasy	**Opis**
AppProperties	Główne właściwości aplikacji.
id	Unikalny identyfikator aplikacji składający się z konkatenacji nazwy aplikacji i 7 znaków z otrzymanego UUID.
type	Typ aplikacji. Dostępne wartości: SINGLE – dla pojedynczego użytkownika MULTI – dla wielu użytkowników
menuType	Typ wyglądu menu. Dostępne wartości: PRIMARY – prosta lista przycisków BOTTOM – dolne menu
AppGroup	Właściwości grup użytkowników
id	Unikalny identyfikator numeryczny przypisywany przez moduł kreatora.
name	Nazwa grupy.
AppGroupsAdministrations	Właściwości administracyjne.
possibleCreateAccountByAdminGroup	Parametr administracyjny. Określa czy użytkownicy z grupy administratorów mogą tworzyć konta użytkownikom.
possibleCreateOwnAccount	Parametr administracyjny. Określa czy można utworzyć swoje konto samemu.
AppRelation	Właściwości relacji.
id	Unikalny identyfikator numeryczny opisanej relacji. Jego wartość jest zarządzana przez moduł kreatora.
firstEntityId	Identyfikator pierwszej struktury/ grupy użytkowników w relacji.
secondEntityId	Identyfikator drugiej struktury/ grupy użytkowników w relacji.
firstEntityFlag	Informacja określająca czy pierwsze id jest strukturą czy grupą użytkowników
secondEntityFlag	Informacja określająca czy drugie id jest strukturą czy grupą użytkowników
isRequire	Informacja określająca czy wymagane jest podanie m.in. jednego połączenia
ScreenProperties	Właściwości ekranów
type	Typ ekranu. Dostępne wartości: READ – odczyt CREATE – tworzenie UPDATE_WITH_DELETE – modyfikacja
entityId	Identyfikator struktury której dotyczy ekran.
forOwner	Informacja tylko dla ekranu typu READ. Określa czy mają być wyświetlane dane tylko utworzone przez danego użytkownika.
AppStyleNum	Właściwości stylu aplikacji
shapeOfButton	Kształt przycisków.
colorOfButton	Kolor przycisków.
fontStyle	Styl czcionki.
AppEntity	Właściwości struktur.
id	Unikalny identyfikator numeryczny struktury przypisywany przez moduł kreatora.
name	Nazwa struktury.

AppField	Właściwości pola.
id	Unikalny identyfikator numeryczny pola przypisywany przez moduł kreatora.
name	Nazwa pola.
isRequired	Informacja określająca czy wymagane jest podanie danego pola przy tworzeniu.
mainAttribute	Informacja określająca czy pole jest atrybutem głównym pomagającym w identyfikacji danego obiektu.
type	Typ pola. Dostępne wartości: TEXT – tekst NUMBER – liczba E_MAIL – adres e-mail DATE – data TIME – godzina BOOLEAN – tak/nie

4.3.4 Dane z wygenerowanych aplikacji końcowych

Dane wprowadzane przez osoby korzystające z aplikacji końcowej są zapisywane w usłudze Firebase - *Realtime Database* w formacie JSON. Rysunek 11 prezentuje system w jaki sposób są one przechowywane, aby możliwa była ich interpretacja i modyfikacja.



Rysunek 11. Schemat zapisu danych aplikacji końcowej. Źródło: opracowanie własne

Wszystkie dane są zapisywane w ramach identyfikatora danej aplikacji (parametr `id` z klasy `AppProperties`). Następnie są one skategoryzowane na trzy grupy. Każda określana jest statycznym znacznikiem tekstowym.

Grupa „a:users”

Zawiera informacje o użytkownikach utworzonych w ramach aplikacji końcowej. Komplet danych zawiera: nazwę użytkownika, hasło oraz listę grup (klasa `AppGroup`) do jakich przynależy.

Grupa „d:data”

W tej kategorii zapisywane są dane wypełnione na bazie utworzonych struktur i ich pól. Aby móc odpowiednio się odwołać do tych informacji są one posegregowane według identyfikatora struktury (pole `id` z klasy `AppEntity`). Pod każdym takim kluczem znajduję się lista obiektów, które można odnaleźć za pomocą inkrementowanego klucza głównego. Wartości jakie znajdują się w ramach obiektu są również odnajdywane poprzez swój identyfikator (pole `id` z klasy `AppField`). Dzięki takiemu rozwiązaniu program jest w stanie w prawidłowy sposób zinterpretować wyniki.
Dodatkowym znacznikiem w obiekcie jest „d:created_by_user”, której wartość reprezentuje użytkownika wprowadzającego dane.

Grupa „d:relations”

Ostatnia grupa z informacjami o połączeniach między obiektami. Sposób przechowywania jest zbliżony do systemu w grupie „d:data”. Pierwszym poziomem jest identyfikator struktury (pole `id` z klasy `AppEntity`). Następnie zapisane są utworzone obiekty. Dla których można odczytać występujące relacje (pole `id` z `AppRelation`). Pod kluczem relacji są zapisane wszystkie identyfikatory obiektów z którymi dana struktura w relacji jest połączona.

W celu lepszego zobrazowania opisywanego systemu poniżej zaprezentowano przykłady dostępu do danych.

Aplikacja bazowa do opisania zawiera zadeklarowane dwie struktury „Zadanie” (Tabela 3, Tabela 4) oraz „Pomieszczenie” (Tabela 5, Tabela 6), które są połączone (Tabela 7). Jest przeznaczona dla wielu użytkowników, której celem jest wyznaczanie obowiązków domowych w konkretnych pomieszczeniach (Tabela 2). Jej parametryzacja wygląda następująco:

Tabela 2. Obiekt klasy AppProperties dla aplikacji obowiązki domowe. Źródło: opracowanie własne

Obiekt klasy AppProperties	
Atrybut	**Wartość**
id	Obowiązki domowed20ee114
name	Obowiązki domowe
menuType	PRIMARY
type	MULTI

Tabela 3. Obiekt klasy AppEntity opisujący zadanie. Źródło: opracowanie własne

Obiekt klasy AppEntity	
Atrybut	**Wartość**
entityNo	0
name	Zadanie

Tabela 4. Obiekty klasy AppField opisujące pola dla zadania. Źródło: opracowanie własne

Obiekty klasy AppField należące do obiektu „Zadanie”				
id	**name**	**type**	**required**	**mainAttribute**
0	Data	DATE	True	True
1	Czas	TIME	True	True
2	Opis	TEXT	True	True

Tabela 5. Obiekt klasy AppEntity opisujący pomieszczenie. Źródło: opracowanie własne

Obiekt klasy AppEntity	
Atrybut	**Wartość**
entityNo	1
name	Pomieszczenie

Tabela 6. Obiekty klasy AppField opisujące pola dla pomieszczenia. Źródło: opracowanie własne

Obiekty klasy AppField należące do obiektu „Pomieszczenie”				
id	**name**	**type**	**required**	**mainAttribute**
0	Data	DATE	True	True
1	Czas	TIME	True	True
2	Opis	TEXT	True	True

Tabela 7. Obiekt klasy AppRelation opisujący relacje między zadaniem a pomieszczeniem. Źródło: opracowanie własne

Obiekt klasy AppRelation	
Atrybut	**Wartość**
id	0
firstEntityId	0

secondEntityId	1
firstEntityFlag	True
secondEntityFlag	True
required	True

Tabela 8. Obiekt klasy AppGroup opisujący grupy administrator. Źródło: opracowanie własne

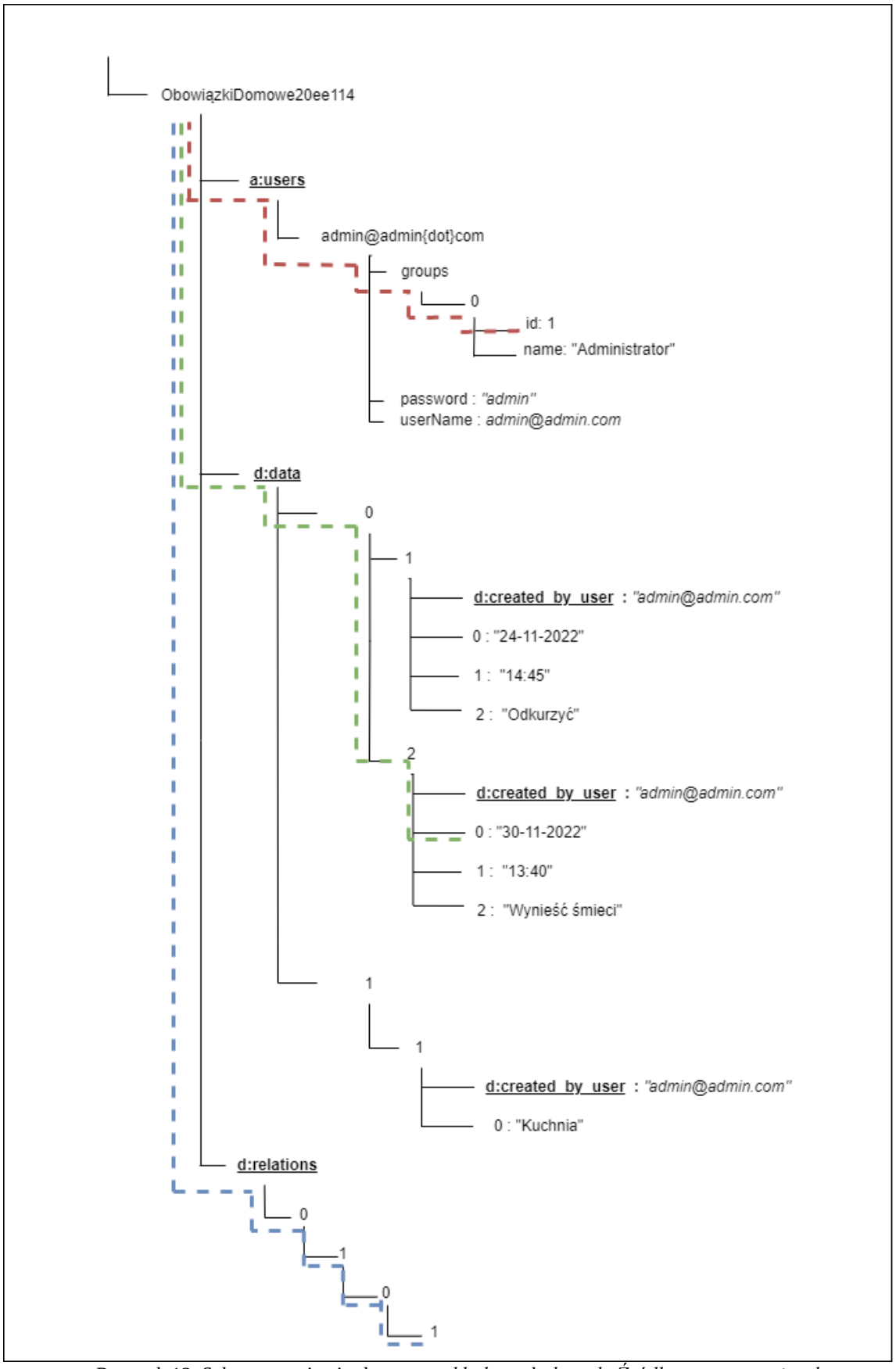
Obiekt klasy AppGroup	
Atrybut	**Wartość**
id	1
name	Administrator

Dane wprowadzone do aplikacji, które przedstawia Rysunek 12.

1. Grupa „Administrator” o `id` = 1 (Tabela 8)
2. Użytkownik „admin” należący do grupy „Administrator”
3. Pomieszczenie „Kuchnia” o `id` = 1
4. Zadanie „Odkurzyć” o `id` = 1 z datą wykonania 24-11-2022 i czasie 14:15.
5. Zadanie „Wynieść śmieci” o `id` = 2 z datą wykonania 30-11-2022 i czasie 13:40
6. Dla zadania „Odkurzyć” przypisano pomieszczenie „Kuchnia”

W ramach przykładów oznaczano poszczególnymi kolorami trzy ścieżki uzyskania danych na Rysunku 12:

- Czerwona. Przypadek sprawdzenia czy użytkownik należy do danej grupy.
- Zielona. Odczyt wartości dla pola przechowującego datę (`AppField` o `id` = 0) w ramach zadania „Wynieść śmieci” (`AppEntity` o `id` = 0). Zapis tej danej pod odpowiednim identyfikatorem (w tym przypadku 0) pozwala aplikacji na prawidłową interpretacje podczas działania odwołując się do obiektu `AppField` z odpowiednim `id` i ustawioną relacje z obiektem `AppEntity`. Omawiane zadania posiada wartość klucza głównego = 2, który został przypisany przy zapisywaniu.
- Niebieska. Odczytanie w ramach struktury zadanie (`AppEntity id` = 0) obiektu o `id` = 1 wszystkich powiązanych elementów dla relacji o `id` = 0. Informację jakiego typu encji jest druga stroną relacji można odczytać z obiektu `AppRelation`.



Rysunek 12. Schemat zapisu i odczytu przykładowych danych. Źródło: opracowanie własne

Klasa `DataServiceFirebase` odpowiedzialna jest za modyfikacje danych w proponowanym systemie przechowywania. Parametry globalne używane przez wszystkie funkcje pokazane są w Listingu 2.

Listing 2. Parametry globalne klasy DataServieFirebase. Źródło: opracowanie własne

```
class DataServiceFirebase(val appProperties: AppProperties) : DataService {

    private val database = Firebase.database
    //Referencja do danych danej aplikacji
    val applicationRef = database.getReference(appProperties.id)
    val DATA = "d:data"
    val RELATIONS = "d:relations"
    val CREATED_BY_USER = "d:created_by_user"
```

Najbardziej rozbudowaną jest metoda dodająca dane – `add` (Listing 3). Jej zadaniem są:

- Ustalenie kolejnej wartości numerycznej dla identyfikatora obiektu w ramach jednej struktury.
- Przekonwertowanie wszystkich wartości do typu String.
- Dodanie informacji o użytkowniku dokonującego wpisu (tylko dla aplikacji dla wielu użytkowników).
- Zapisanie danych dla odpowiedniej struktury dla danej aplikacji.
- Zapisanie relacji jeśli zostały ustanowione.

Metoda `update` (Listing 4) wykonuję analogicznie zadania tylko dla podanego identyfikatora obiektu w parametrach wejściowych.

Listing 3. Implementacja metody add klasy DataServieFirebase. Źródło: opracowanie własne

```
    //Metoda dodająca dane
    override fun add(
        entityNo: Int, //Id modyfikowanej struktury
        values: Map<Int, String>, //Wartości obiektów
        onFinished: (Boolean) -> Unit, // Wywołanie zwrotne
        // Mapa relacja między strukturami
        relations: Map<Int, HashMap<Int, HashMap<Int, HashSet<Int>>>>,
        // Mapa relacja między strukturami a użytkownikami
        relationsUsers: Map<Int, HashMap<Int, HashMap<Int, HashSet<String>>>>) {
        applicationRef.child(DATA).child(entityNo.toString()).get().addOnSuccessListener {
            // Określenie wartości kolejnego klucza głównego lda tworzonego obiektu
            var max = 0;
            for (child in it.children) {
                if (child.key != null && child.key!!.toInt() > max)
                    max = child.key!!.toInt()
            }
            //Przekonwertowanie wszystkie wartości obiektu do typu String
            val mapsOfStrings = mutableMapOf<String, String>()
            mapsOfStrings.putAll(values.mapKeys { entry -> entry.key.toString() }.toMap())
            //Jeśli aplikacja dla wielu użytkowników. Dodanie informacje o autorze.
            if (appProperties.type == TypeApp.MULTI)
                mapsOfStrings[CREATED_BY_USER] = CurrentUser.userApplication!!.userName

            val nextId = max + 1
            // Zapisanie danych dla odpowiedniej struktury z nowym id obiektu
            applicationRef.child(DATA)
                .child(entityNo.toString())
                .child((nextId).toString())
                .setValue(mapsOfStrings)
                .addOnSuccessListener {
                    // Zapisanie relacji
                    if (relations.contains(entityNo) &&
                        relations[entityNo]!!.containsKey(-1) &&
```

```
                        relations[entityNo]!![-1]!!.isNotEmpty()
                    ) {
                        if (relations.containsKey(entityNo) &&
relations[entityNo]!!.containsKey(-1)) {
                            relations[entityNo]!![-1]!!.forEach { (k, v) ->
                                val setOfString = mutableListOf<String>()
                                v.forEach { it -> setOfString.add(it.toString()) }
                                applicationRef
                                    .child(RELATIONS)
                                    .child(entityNo.toString())
                                    .child((nextId).toString())
                                    .child(k.toString())
                                    .setValue(setOfString)
                            }
                        }

                        if (relationsUsers.containsKey(entityNo) &&
relationsUsers[entityNo]!!.containsKey(-1)) {
                            relationsUsers[entityNo]!![-1]!!.forEach { (k, v) ->
                                val setOfString = mutableListOf<String>()
                                v.forEach { it -> setOfString.add(it.toString()) }
                                applicationRef
                                    .child(RELATIONS)
                                    .child(entityNo.toString())
                                    .child((nextId).toString())
                                    .child(k.toString())
                                    .setValue(setOfString)
                            }
                        }
                        onFinished(true)
                    } else
                        onFinished(true)

                }
            }
        }
```

Listing 4. Implementacja metody update klasy DataServieFirebase. Źródło: opracowanie własne

```
    //Metoda aktualizująca dane
    override fun update(
        entityNo: Int, //Id modyfikowanej struktury
        elementId: Int, //Id modyfikowanego obiektu
        values: Map<Int, String>, //Nowe wartości dla obiektu
        onFinished: (Boolean) -> Unit,
        relations: Map<Int, HashMap<Int, HashMap<Int, HashSet<Int>>>>,
        relationsUsers: Map<Int, HashMap<Int, HashMap<Int, HashSet<String>>>>
    ) {
        applicationRef.child(DATA).child(entityNo.toString()).get().addOnSuccessListener {
            //Przekonwertowanie danych do typu String
            var mapsOfStrings = mutableMapOf<String, String>()
            mapsOfStrings.putAll(values.mapKeys { entry -> entry.key.toString() }.toMap())

            //Jeśli aplikacja dla wielu użytkowników. Dodanie informacje o autorze.
            if (appProperties.type == TypeApp.MULTI)
                mapsOfStrings[CREATED_BY_USER] = CurrentUser.userApplication!!.userName

            // Zapisanie danych dla odpowiedniej struktury o podanym id obiektu.
            applicationRef.child(DATA)
                .child(entityNo.toString())
                .child((elementId).toString())
                .setValue(mapsOfStrings)
                .addOnSuccessListener {
                    // Zapisanie relacji
                    relations.forEach { (entityNo, entityNoValues) ->
                        entityNoValues.forEach { (elementId, elementIdValues) ->
                            elementIdValues.forEach { (relationId, relationIdElements) ->
                                val setOfString = mutableListOf<String>()
                                relationIdElements.forEach { it ->
setOfString.add(it.toString()) }
```

```
                                applicationRef
                                    .child(RELATIONS)
                                    .child(entityNo.toString())
                                    .child((elementId).toString())
                                    .child(relationId.toString())
                                    .setValue(setOfString)
                            }
                        }
                    }
                    relationsUsers.forEach { (entityNo, entityNoValues) ->
                        entityNoValues.forEach { (elementId, elementIdValues) ->
                            elementIdValues.forEach { (relationId, relationIdElements) ->
                                val setOfString = mutableListOf<String>()
                                relationIdElements.forEach { it ->
setOfString.add(it.toString()) }
                                applicationRef
                                    .child(RELATIONS)
                                    .child(entityNo.toString())
                                    .child((elementId).toString())
                                    .child(relationId.toString())
                                    .setValue(setOfString)
                            }
                        }
                    }
                    onFinished(true)
                }
            }
        }
```

Najprostszą implementację posiada metoda przeznaczona do usuwania – `delete`. Dla wskazanej struktury oraz identyfikatora obiektu usuwa ona najpierw dane, a następnie relacje.

Listing 5. Implementacja metody delete klasy DataServieFirebase. Źródło: opracowanie własne

```
override fun delete(
        entityNo: Int,//Id modyfikowanej struktury
        elementId: Int,//Id usuwanego obiektu
        onFinished: (Boolean) -> Unit
    ) {
        //Usunięcie obiketu o podanym id dla podanej struktury
        applicationRef.child(DATA)
            .child(entityNo.toString())
            .child((elementId).toString())
            .removeValue()
            .addOnSuccessListener {
                //Usunięcie relacji
                applicationRef
                    .child(RELATIONS)
                    .child(entityNo.toString())
                    .child((elementId).toString())
                    .removeValue().addOnSuccessListener {
                        onFinished(true)
                    }
            }
    }
}
```

4.3.5 Generowanie UI

W proponowanym rozwiązaniu interfejs graficzny jest tworzony szablonowo, dostarczając użytkownikowi ekrany zapewniające kompleksową obsługę zarządzania danymi według struktur. Funkcje jakie może pełnić ekran to:

- Wyświetlanie obiektów
- Dodanie

- Modyfikacja z usunięciem

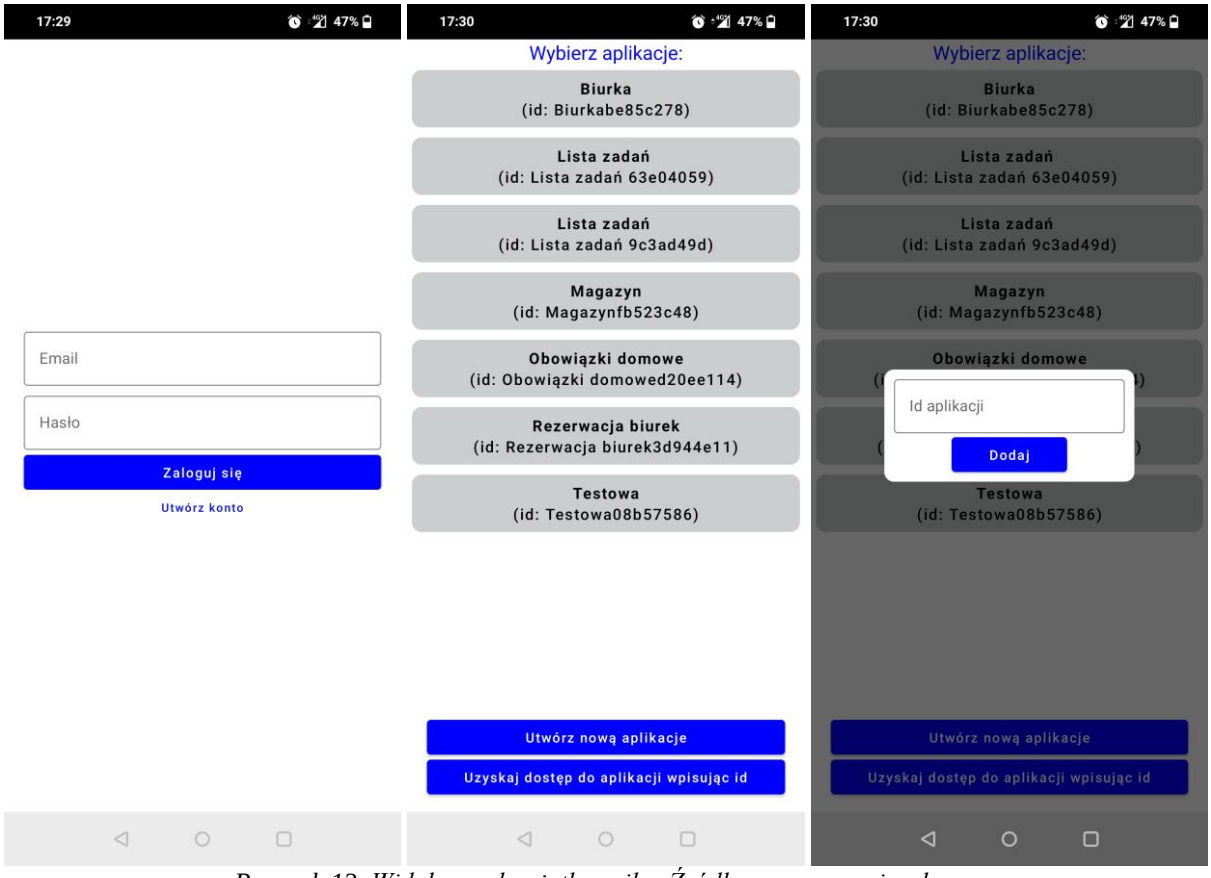
Podczas tworzenia własnego rozwiązania można określić elementy w menu. Aby dodać należy wybrać jakiej struktury ma dotyczyć a następnie jaką funkcje ma pełnić. Parametry te zostają zapisane za pomocą klasy `ScreenProperties`. Ekrany dodawania i modyfikacji generuje formularz na podstawie definicji pól (`AppField`) i relacji (`AppRelation`), który sprawdza poprawność typu wpisywanych danych oraz wymagalność.

4.4. Funkcjonalności

W tym podrozdziale przedstawiono jakie funkcjonalności dostarcza zaproponowany kreator. Każda oprócz opisu zawiera również rozważania nad możliwością rozwoju i ograniczeń. Aby lepiej zobrazować niektóre efekty końcowe przykłady prezentują parametry jakie mógłby być podane do stworzenia aplikacji do rezerwacji biurek w miejscu pracy.

4.4.1 Panel użytkownika

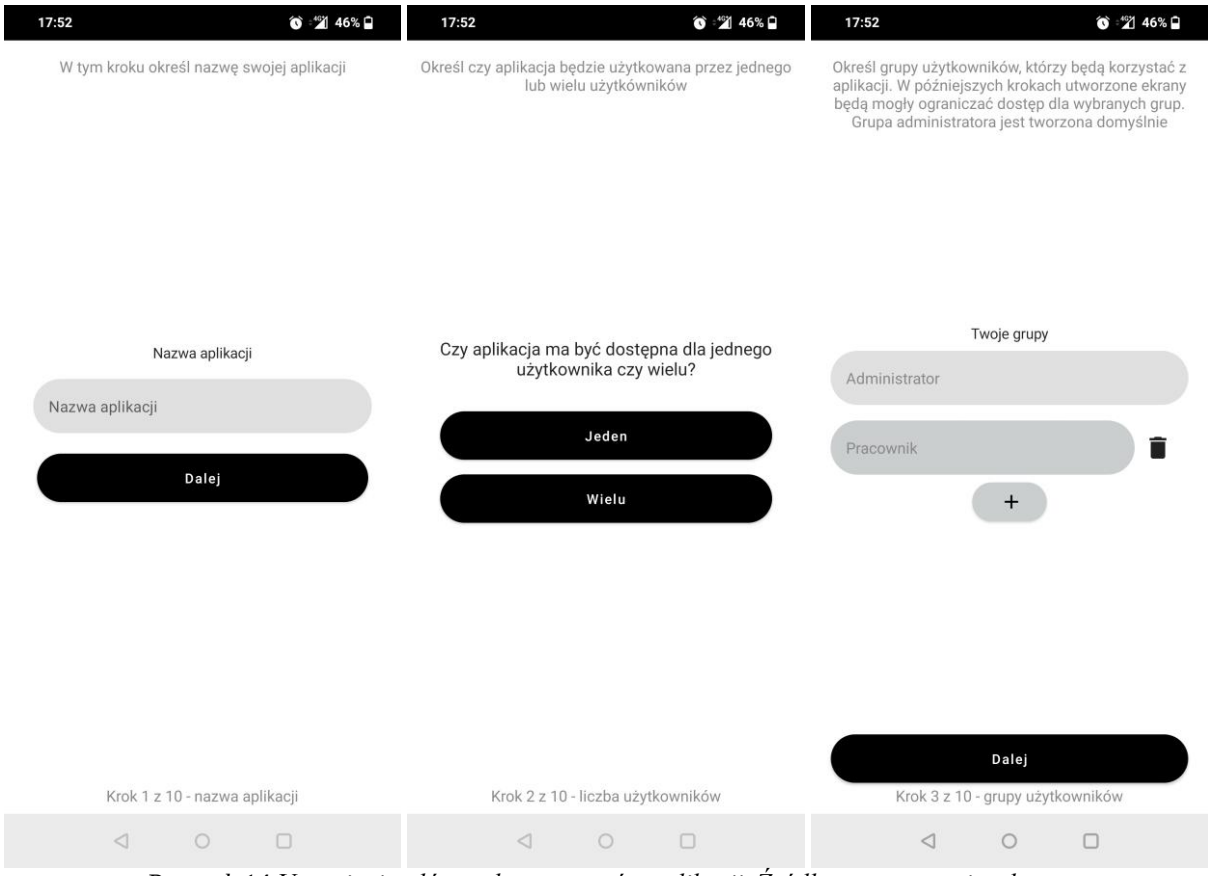
Każdy użytkownik chcący utworzyć swoje rozwiązanie jest zobligowany do założenia konta podstawowego. Wymagany jest adres e-mail oraz hasło. Z tego poziomu można uruchamiać aplikacje już z parametryzowane lub utworzyć nowe. W celu podzielenia się dostępem do własnego pomysłu należy przekazać widoczny identyfikator do innych osób, który następnie można wpisać w wyznaczonym do tego polu. Dzięki takiemu podejściu użytkownicy tylko znający unikalne id mogą korzystać z aplikacji. Jedną z możliwości rozszerzenia jest zaimplementowanie generowania kodu QR, aby przyspieszyć czas udostępniania. Rysunek 13 przedstawia interfejs graficzny panelu.



Rysunek 13. Widok panelu użytkownika. Źródło: opracowanie własne

4.4.2 Główne parametry

W etapie pierwszym kreowania użytkownik proszony jest o podanie i zaznaczenie głównych atrybutów przedstawionych na Rysunku 14. Nazwa aplikacji pozwala na identyfikacje aplikacji. W kroku drugim kreator zezwala na wybór czy program ma być przeznaczony dla wielu czy pojedynczego użytkownika. W zależności od wyboru proces przyjmuje dwie ścieżki. Krótszą dla jednego odbiorcy oraz dłuższa dla wielu, wymagającej podania m.in. grupy użytkowników czy właściwości administracyjnych. Ostatnią opcją do wpisania są grupy użytkowników. Zawsze tworzona jest grupa „Administratora" bez możliwości jej usunięcia. Zadaniem takiego zabiegu jest niedopuszczenie użytkowników do zablokowania możliwości aplikacji. Grupa ta uzyskuję automatycznie ekran do zarządzania innymi osobami i ich przynależności. Przy potencjalnym rozwijaniu prototypu grupa „Administrator" mogłaby uzyskać więcej opcji zarządzania aplikacją. Funkcje administracyjne są opisane bliżej w podrozdziale 4.4.6. Ponadto grupowanie użytkowników pozwala na ograniczenia dostępu do poszczególnych ekranów czy utworzeniu relacji tworzonych w kolejnych etapach. Kreator nie ogranicza ilości tworzonych grup. Jedną z wad definiowania podziału na tym etapie jest trudność zobrazowanie odbiorcy zastosowania tej funkcjonalności w produkcie końcowym. Przykładowa ilustracja utworzy dodatkową grupę „Pracownik".



Rysunek 14.Ustawienia głównych parametrów aplikacji. Źródło: opracowanie własne

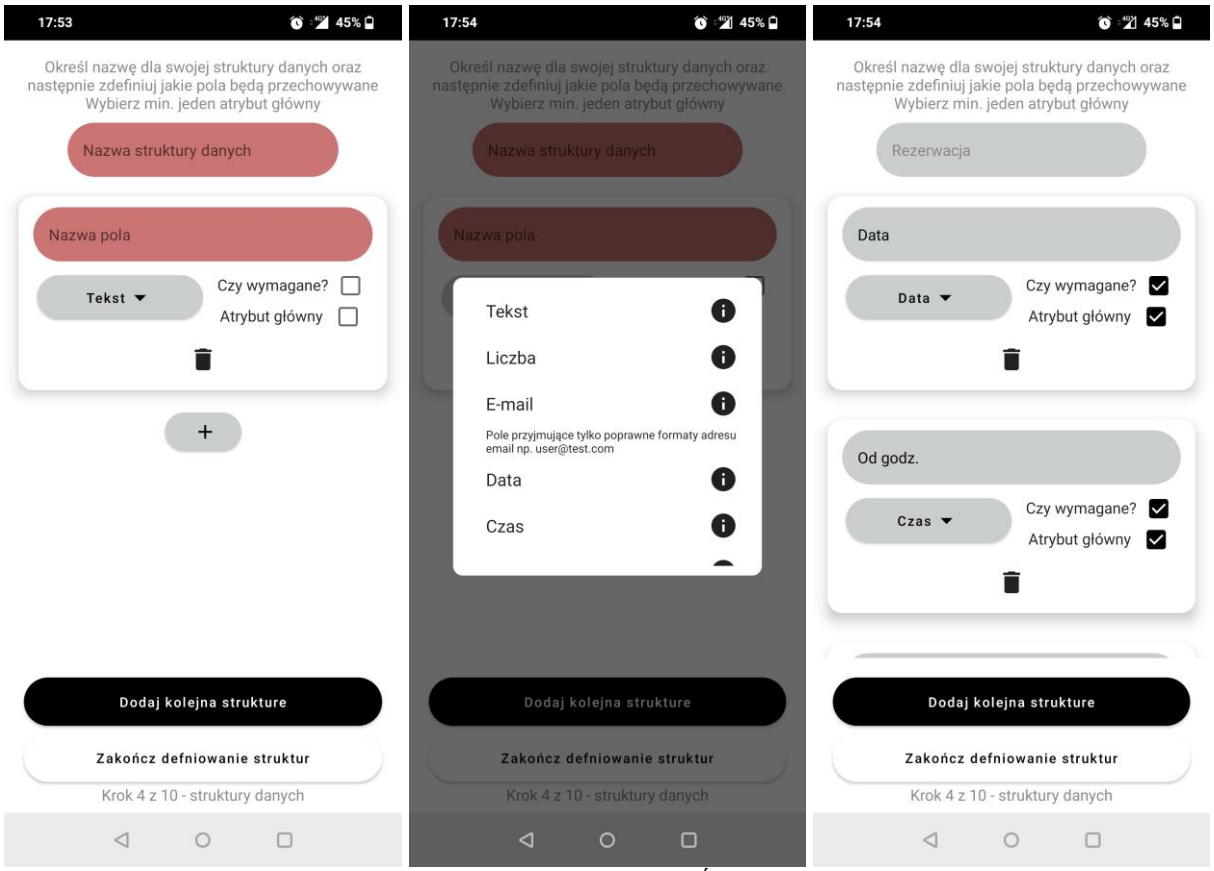
4.4.3 Struktury i relacje

Struktury i relacje są najważniejszą funkcjonalnością zaproponowanego prototypu. Ich sposób zdefiniowania czyni aplikację końcową bardziej lub mniej użyteczną. W celu lepszego zrozumienia przez użytkownika przed przystąpieniem do tego etapu istnieje możliwość podglądu prostego przykładu. Struktury powinny reprezentować rzeczywisty obiekt, byt. Przy modelowaniu baz danych taką reprezentacją określają encje. Z względu na grupę odbiorców jakimi są użytkownicy nie techniczni zdecydowano się na uproszczenia i zastosowanie innego nazewnictwa. Aby określić

strukturę w kreatorze jest wymagane aby podać nazwę struktury oraz minimum jedno pole. Nazwa powinna umożliwić w sposób ogólny zidentyfikować dany byt lub grupę pól np. „Biurko” czy „Rezerwacja”. Ilość pól nie jest w żaden sposób ograniczona przez oprogramowanie. Każde pole musi zawierać:

- Nazwę
- Informacje czy wymagane jest podanie wartości przy tworzeniu nowego obiektu
- Znacznik atrybutu głównego. Atrybut ten czy ich zbiór powinien umożliwić identyfikacje danego obiektu. Cecha wykorzystywana jest przy wyświetlaniu listy obiektów w aplikacji końcowej. Wyświetlenia wszystkich pól przy założeniu braku maksymalnej ich ilości mogłoby sprawić nieczytelność widoku listy.
- Typ danych. Pole musi mieć określony typ aby było użyteczne. Dopuszczenie każdej wartości nie pozwoliło by na kontrolowanie zapisywanych wartości przez użytkowników końcowych. W celu poinstruowania każdy typ zawiera swój opis.

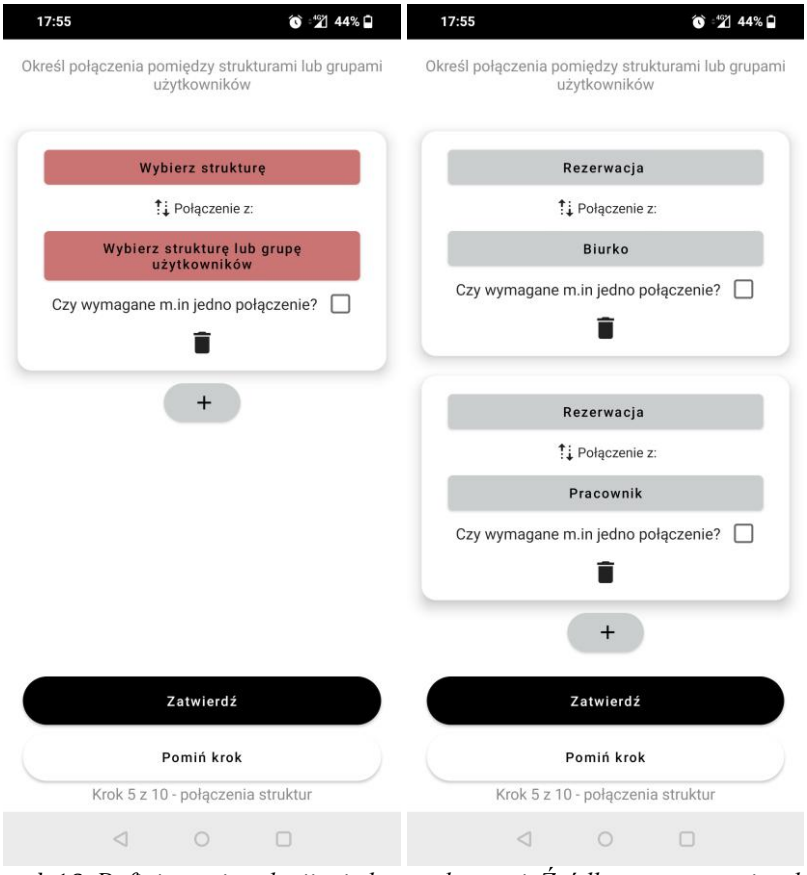
Kreator również nie ogranicza ilości tworzonych struktur. Rysunek 15 przedstawia ekrany do ich wprowadzania.



Rysunek 15. Definiowanie struktur aplikacji. Źródło: opracowanie własne

Kolejno utworzone struktury można łączyć z innymi strukturami lub z grupami użytkowników relacjami (Rysunek 16). W ramach definicji istnieje tylko opcja wyznaczenia czy jest wymagane min. jeden połączony obiekt. Wszystkie relacje są krotności wiele do wielu. Głównym powodem jest brak możliwości prostej ilustracji użytkownikowi wszystkich dostępnych rodzajów połączeń. Dzięki temu w sposób przystępny można bez nadmiarowych opcji ustanowić relacje. Niewątpliwą wadą jest brak sprawdzania poprawności relacji według założeń autora rozwiązania. Jeśli wymagane jest aby

rezerwacja dotyczyła tylko jednego biurka, odpowiedzialność jest po stronie użytkowników. Odbiorcami kreatora jest mała organizacja lub osoby prywatne co powinno zezwolić na kontrolowanie zasad w danej grupie w inny sposób.



Rysunek 16. Definiowanie relacji między strukturami. Źródło: opracowanie własne

Implementacja

Generowanie szablonów do edycji danych (Rysunek 17) według zdefiniowanych struktur odbywa się poprzez iterowanie przez wszystkie obiekty klasy `AppField`. Ich wartości atrybutów są przekazywane do metody `GenerateCreateField` (Listing 3), która dostarcza pole wejściowe z odpowiednimi właściwościami takimi jak:

- Dostarczenie odpowiedniej informacji jeśli wartość nie jest podana
- Sprawdzanie typów podanych danych
- Dostarczenie typowych dla aplikacji mobilnych interfejsów dla podania daty lub czasu

Dane zapisywane są na bieżąco w obiekcie kolekcji typu mapa. Każda wartość jest zapisana pod kluczem według swojego id określonego w klasie `AppField`. Dzięki temu łatwo przekazać informacje do zapisania do metod klasy `DataServieFirebase` zaprezentowanych w Listingu 2 w wcześniejszych podrozdziałach.

Listing 6. Fragment kodu klasy CreateScreen. Wywołanie metody GenerateCreateField. Źródło: opracowanie własne

```
items(entity.fields) { appField ->
    GeneratorCreateField(
        onValueChange = {
```

```
            viewModel.mapsOfField[appField.id] = it
        },
        onUpdateValidation = {
            viewModel.mapsOfFieldValidation[appField.id] = it
        },
        label = appField.name,
        typeOfField = appField.type,
        value = viewModel.mapsOfField[appField.id].toString(),
        colorPrimary = appStyle.colorOfButtons,
        isRequired = appField.isRequired,
        editEnabled = editEnabled
    )
    Spacer(modifier = Modifier.height(8.dp))
}
```

W celu zapewnienia obsługi relacji najpierw wszystkie obiekty klasy `AppRelation` aplikacji są filtrowane w celu znalezieniu połączeń danej struktury (Listing 4).

Listing 7. Fragment kodu klasy CreateScreen. Filtrowanie relacji. Źródło: opracowanie własne

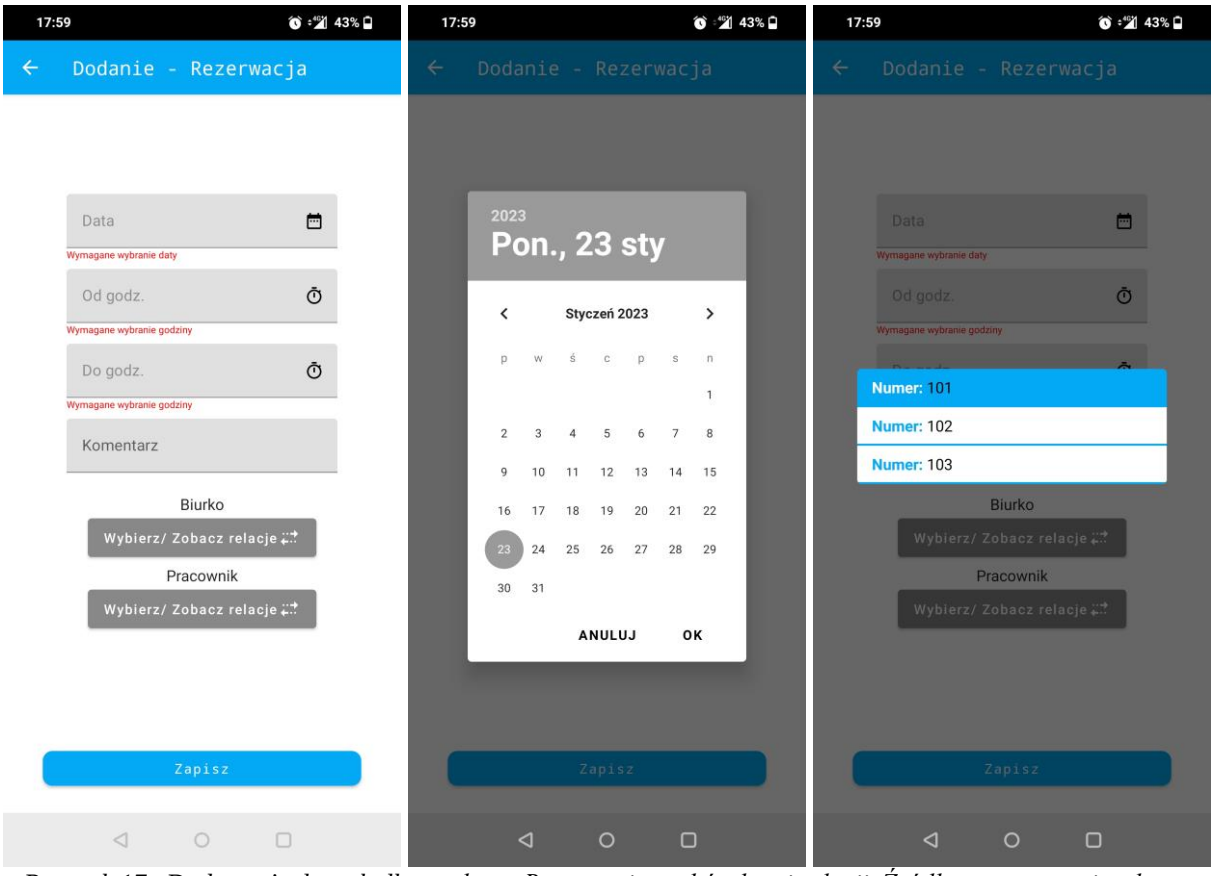
```
val relationsWhereEntityIsFirst: List<AppRelation> =
    appProperties.relations.filter { it.firstEntityId == entityId }
```

Następnie dla każdej znalezionej relacji tworzony jest przycisk po którego kliknięciu ładowane są z bazy danych wszystkie obiekty drugiej strony relacji (Listing 5). Generator wyświetla okienko z załadowanymi danymi, które można następnie zaznaczyć.

Listing 8. Fragment kodu klasy CreateScreen. Generowanie przycisku do obsługi relacji. Źródło: opracowanie własne

```
items(relationsWhereEntityIsFirst) { relation ->
    //Etykieta przycisku
    var foundLabel = ""
    if (relation.secondEntityFlag) {
        foundLabel = appProperties.entities.first { (it.entityNo ==
relation.secondEntity) }.name

        //Przycisk do przeglądania/ wybierania relacji
        GeneratorRelationField(
            onClick = {
                viewModel.loadDataForRelationEntities(
                    relation.secondEntity,
                    onFinished = {
                        viewModel.secondEntity = appProperties.entities.first { it
-> it.entityNo == relation.secondEntity }
                        viewModel.currentRelation = relation
                        if (viewModel.secondData.isEmpty())
                            openDialogWhenEmpty = true
                        else
                            openDialogForRelationWithEntityWhereFirst = true
                    })
            },
            label = foundLabel,
            isRequired = relation.isRequire
        )
```



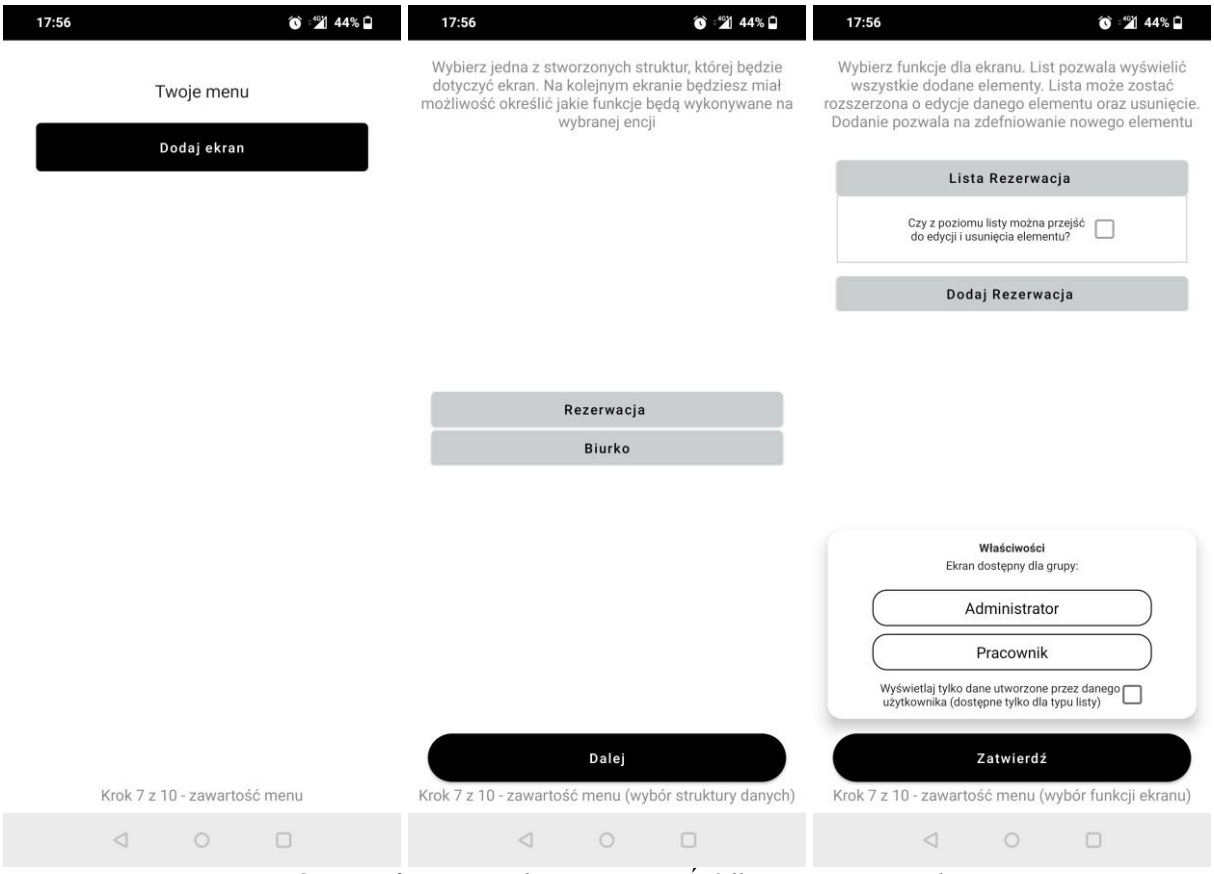
Rysunek 17. Dodawanie danych dla struktury Rezerwacje, wybór daty i relacji. Źródło: opracowanie własne

Wyświetlanie wszystkich obiektów danej struktury w postaci listy (Rysunek 18) odbywa się w analogiczny sposób. Dla przetwarzanej wartości analizowana jest zawartość `AppField` o tożsamy identyfikatorze. Lista wyświetla tylko atrybuty główne aby uniknąć nadmiaru informacji. Ciekawym rozszerzeniem mogłoby być opcja modyfikacji atrybutów głównych według preferencji każdego użytkownika i zapisanie ich w swego rodzaju osobistych ustawień. Dodatkowo aby lista była bardziej użyteczna warto aby została rozszerzona o możliwości wybrania po jakich wartościach powinna być sortowana.

Rysunek 18. Lista prezentująca przykładowe rezerwacje. Źródło: opracowanie własne

4.4.4 Zawartość menu

W celu zapewnienia nawigacji w aplikacji końcowej udostępniono opcje zdefiniowania elementów menu (Rysunek 19). Jak wspomniano w podrozdziale 4.3.5 każdy ekran pełni jedną funkcje. Określenie rozpoczyna się od wybrania struktury na jakiej podstawie zostanie wygenerowany szablon do modyfikacji danych. Następnie należy wybrać jedną z dostępnych funkcji. Z ekranu typu lista można dodatkowo przejść do edycji obiektu lub usunięcia. Dodatkowo dostępność do ekranu można ograniczyć dla danych grup użytkowników. Na przykład pracownik nie powinien móc dodawać biurek. Użytkownik ma podgląd na elementy już dodane do menu. Pilnowane jest, aby dodany został min. jeden element do menu i min. jedna grupa miała dostęp do ekranu, aby nie ustanowić aplikacji bezużytecznej. Nazwa przycisku nawigującego jest tworzona automatycznie poprzez poprzedzenie nazwy struktury funkcją jaka oferuje np. „Dodaj Biurko”.



Rysunek 19. Definiowanie elementu menu. Źródło: opracowanie własne

Implementacja

Generowanie menu (Rysunek 20) odbywa się poprzez przetwarzanie zapisanych na etapie konfiguracji obiektów klasy `ScreenProperties`. Zawiera ona wszystkie dane potrzebne do uruchomienia poszczególnych ekranów. Instrukcje zawierają również sprawdzanie typu aplikacji oraz czy dany użytkownik ma uprawnienia do danego ekranu i czy jest administratorem, aby dostosować odpowiednio menu. Fragment kodu obrazuje Listing 6.

Listing 9. Fragment klasy MenuScreen odpowiedzialny za generowanie menu. Źródło: opracowanie własne

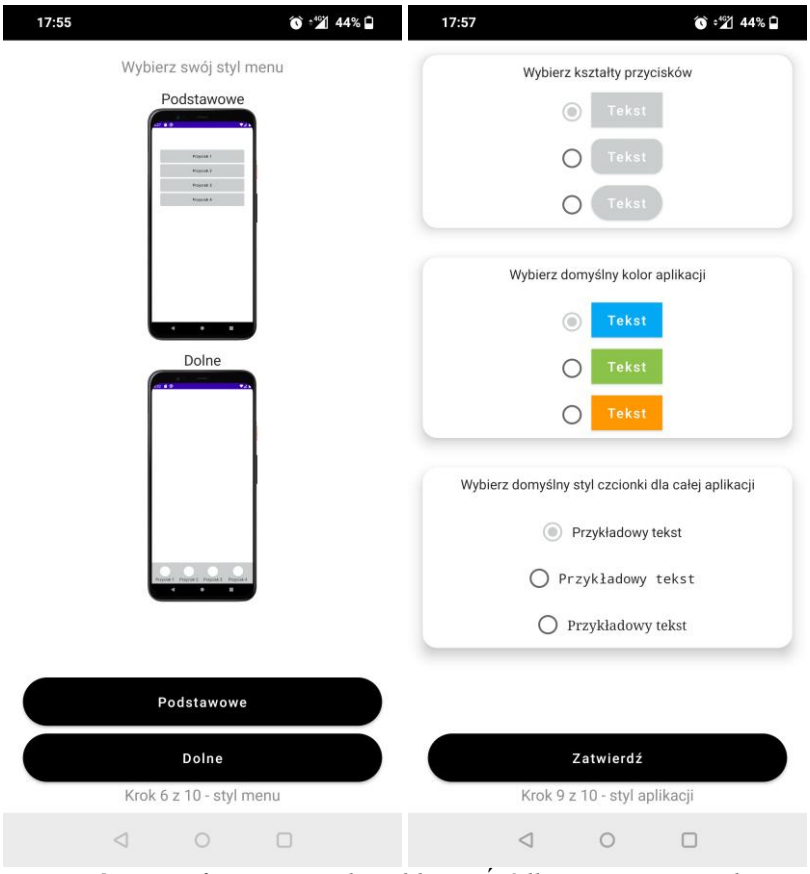
```
LazyColumn() {
    // Iterowanie obiektów
    items(appProperties.screens) { screen ->
        // Sprawdzenie czy zalogowany użytkownik posiada uprawnienia do
ekranu
        if ((appProperties.type == TypeApp.MULTI &&
CurrentUser.checkGroups(
                screen.accessGroups
            )) || appProperties.type == TypeApp.SINGLE
        ) {
            val editEnabled = screen.additionalTypeForRead != null
            // Wygenerowanie przycisku wraz z defnicją akcji po klinięciu
            // w zależności od typu ekranu.
            GeneratorPrimaryButton(
                modifier = Modifier.fillMaxWidth(0.8f),
                onButtonClick = {
                    if (screen.type == AppMenuElementType.CREATE) {
```

```
viewModel.onEvent(MenuEvent.OpenCreateScreen(screen.entityId))
                    } else {
                        viewModel.onEvent(
                            MenuEvent.OpenReadScreen(
                                screen.entityId,
                                editEnabled,
                                screen.forOwner
                            )
                        )
                    }
                },
                text = screen.name,
                colorPrimary = appStyle.colorOfButtons,
                fontStyle = appStyle.fontStyle,
                shape = appStyle.shapeOfButtons
            )
            Spacer(modifier = Modifier.height(5.dp))
        }
    }
    // Wygenerowanie dodatkowego przycisku dla administratorów
    item {
        if (appProperties.type == TypeApp.MULTI &&
CurrentUser.isAdministrator()) {
            GeneratorPrimaryButton(
                modifier = Modifier.fillMaxWidth(0.8f),
                onButtonClick = {
                    viewModel.onEvent(MenuEvent.OpenAdminScreen)
                },
                text = "Zarządzanie użytkownikami",
                colorPrimary = appStyle.colorOfButtons,
                fontStyle = appStyle.fontStyle,
                shape = appStyle.shapeOfButtons
            )
        }
    }
}
```

Rysunek 20. Przykład wygenerowanego menu. Źródło: opracowanie własne

4.4.5 Styl aplikacji

Mobilne programy obecnie cechują się przyjaznym w swym wyglądzie i stylu interfejsem graficznym. Z tego powodu zdecydowano się na implementacje również możliwości dostosowania wyglądu aplikacji (Rysunek 21) o kilka cech jakimi są: styl menu, kształt przycisków, kolor dominujący oraz główną czcionkę. Ten element kreatora jest jedną z opcji najłatwiejszą do potencjalnego rozbudowania o nowe cechy, wartości i jednocześnie dostarczającą wiele możliwości wizualnych.



Rysunek 21. Definiowanie stylu aplikacji. Źródło: opracowanie własne

Implementacja

Zapewnienie wybranych stylów elementów odbywa się poprzez pobranie właściwości z bazy danych otwieranej aplikacji. Wartości te są zapisywane w formacie liczb naturalnych w klasie `AppStyleNum` (Listing 10), a następnie mapowane do odpowiednich klas stylu i przetrzymywane w ramach `AppStyleValue` (Listing 11). Klasa ta jest przekazywana do każdego ekranu. Dzięki Jetpack Compose użycie tych wartości jest proste, ponieważ można nadpisać wartości domyślne kontrolek takie jak: shape, color czy fontFamily jeszcze przed wygenerowanie elementów na smartfonie. Na Listingu 12 pokazano metodę tworzącą przycisk w module generatora. Przyjmuje ona trzy parametry dotyczące wyglądu: `colorPrimary`, `shape` oraz `fontyStyle`. Przekazanie parametrów danej aplikacji do tej funkcji można zaobserwować na Listingu 13.

Listing 10. Implementacja map z wartościami stylu. Źródło: opracowanie własne

```
@Transient
private val shapeOfButtonMap: HashMap<Int, RoundedCornerShape> = hashMapOf(
    0 to RoundedCornerShape(0.dp),
    1 to RoundedCornerShape(10.dp),
    2 to RoundedCornerShape(50.dp)
)

@Transient
private val fontFamilyMap: HashMap<Int, FontFamily> = hashMapOf(
    0 to FontFamily.Default,
    1 to FontFamily.Monospace,
    2 to FontFamily.Serif
)

@Transient
```

```
private val buttonColorsMap: HashMap<Int, Color> = hashMapOf(
    0 to StyleColorBlue,
    1 to StyleColorGreen,
    2 to StyleColorOrange
)
```

Listing 11. Definicja klasy AppStyleValue. Źródło: opracowanie własne

```
data class AppStyleValue(
    var shapeOfButtons: RoundedCornerShape = RoundedCornerShape(0.dp),
    var colorOfButtons: Color = Color.Black,
    var fontStyle: FontFamily = FontFamily.Default)
```

Listing 12. Definicja metody GeneratorPrimaryButton. Źródło: opracowanie własne

```
@Composable
fun GeneratorPrimaryButton(
    modifier: Modifier = Modifier,
    text: String = "",
    onButtonClick: () -> Unit,
    colorPrimary: Color = MaterialTheme.colors.primary,
    shape: RoundedCornerShape = RoundedCornerShape(0.dp),
    fontStyle: FontFamily = FontFamily.Default
) {
    Button(
        modifier = modifier,
        onClick = { onButtonClick() }, shape = shape,
        colors = ButtonDefaults.textButtonColors(
            backgroundColor = colorPrimary,
            contentColor = Color.White
        ),
    ) {
        Text(text = text, fontFamily = fontStyle)
    }
}
```

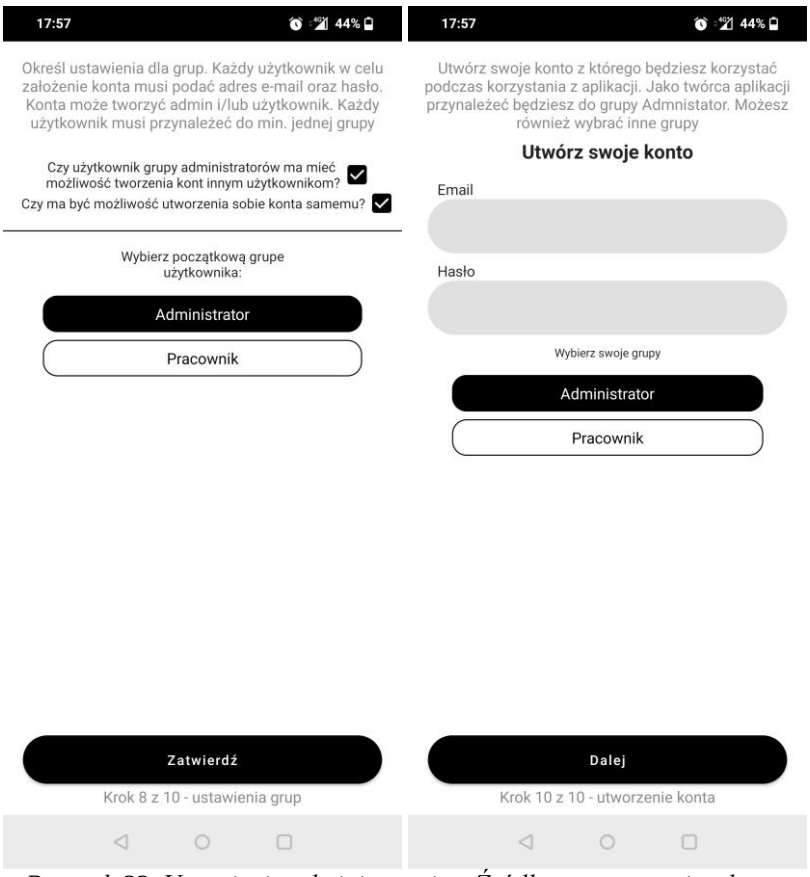
Listing 13. Wywołanie metody GeneratorPrimaryButton. Źródło: opracowanie własne

```
GeneratorPrimaryButton(
    modifier = Modifier.fillMaxWidth(0.8f),
    onButtonClick = {
        dataService.delete(
            entityNo = entityId,
            elementId = elementId,
            onFinished = { navController.navigateUp() })
    },
    text = "Usuń",
    //Wartości stylu aplikacji
    colorPrimary = appStyle.colorOfButtons,
    fontStyle = appStyle.fontStyle,
    shape = appStyle.shapeOfButtons
)
```

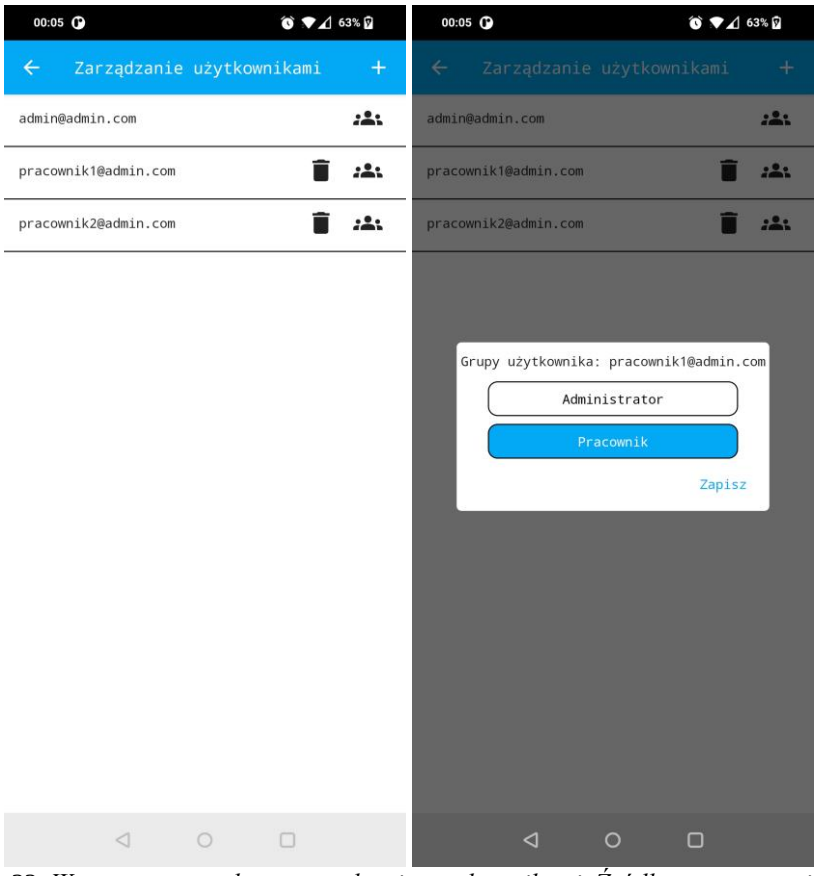
4.4.6 Ustawienia administracyjne

Ustawienia administracyjne dostarczają podstawowych parametrów jakie należy ustawić z względu na koncepcje przyjęte w proponowanym rozwiązaniu. Ostatnim krokiem jest utworzenie własnego konta przez autora rozwiązania, aby nie zablokował on sobie dostępu do własnej aplikacji.

Użytkownik ten jest automatycznie przypisywany do grupy „Administrator” oraz może zdecydować o swojej przynależności do innych utworzonych grup. Tworząc konto zarówno przez twórcę jak i innych osób wymagane jest aby loginem był adres e-mail. Adres ten może stanowić jedyne źródło kontaktu dla administratora do użytkowników, co jest według autora pracy magisterskiej zaletą. Nowo tworzone konta muszą zostać przypisane do jednej z dostępnych grup, dlatego należy ją określić. Jej ustawienie dostępne jest w kroku 8. Możliwe jest wybranie pomiędzy dwoma opcjami rejestrowania nowych osób. Pierwszy to sposób standardowy, czyli samodzielna rejestracja. Druga to dostarczenie funkcji rejestrowania grupie administratorów, dzięki czemu to oni mają kontrole kto korzysta z aplikacji, w przypadku udostępnienia id aplikacji osobą trzecim. Aby przejść dalej wymagane jest wybranie min. jednej opcji. Ponadto aby umożliwić zarządzanie dla osób z grupy administracyjnej generowany jest dodatkowy ekran dostępny z poziomu menu „Zarządzanie użytkownikami”. Zezwala on na usuwanie, dodawanie (jeśli została wybrana ta opcja) i zmianę przynależności grup. Zablokowane jest usunięcie siebie samego z sekcji „Administrator” w celu uchronienia sytuacji z brakiem osoby zarządzającej dla aplikacji. Na Rysunku 22 przedstawiono moduł kreatora umożlwiający konfiguracje tych ustawień. Natomiast na Rysunku 23 widać wygląd ekranu do zarządzania.



Rysunek 22. Ustawienia administracyjne. Źródło: opracowanie własne



Rysunek 23. Wygenerowany ekran zarządzania użytkownikami. Źródło: opracowanie własne

5. Przykładowe aplikacje stworzone przy pomocy kreatora

W tym rozdziale opisano aplikacje jakie zostały utworzone dzięki prototypowi na podstawie przykładów z podrozdziału 4.1.

5.1. Lista zadań

Prosta aplikacja pozwalająca na zarządzanie rozmaitego typu zadaniami do wykonania. Dzięki kreatorowi użytkownik powinien mieć możliwość zdefiniowania atrybutów jakie chce przechowywać według własnych potrzeb. Rozwiązanie przeznaczone tylko dla jednego użytkownika jako pomoc w codziennym życiu. Rzeczy do wykonania mogą być przypisane do różnych kategorii jak „Dom" czy „Praca". Ważnym elementem jest również opcja zaznaczenia które zadania nie zostały jeszcze zrobione.

5.1.1 Parametryzacja

Kluczowe parametry jakie zostały podane w procesie kreowania:

- Typ aplikacji: *dla jednego użytkownika*
- Struktura „*Zadanie*" z atrybutami:
    - *Opis (wymagane, atrybut główny)*
    - *Data (wymagane, atrybut główny)*
    - *Czas*
    - *Czy wykonano*
- Struktura „*Kategoria*" z atrybutami:
    - *Nazwa (wymagane, atrybut główny)*
- „*Zadanie*" oraz „*Kategoria*" zostały połączone relacją.
- Menu zawiera opcje do *dodania* i *wyświetlenie wraz z edycją* każdej struktury.

Dodatek A przedstawia pełną parametryzację w formacie JSON wyeksportowaną z bazy danych.

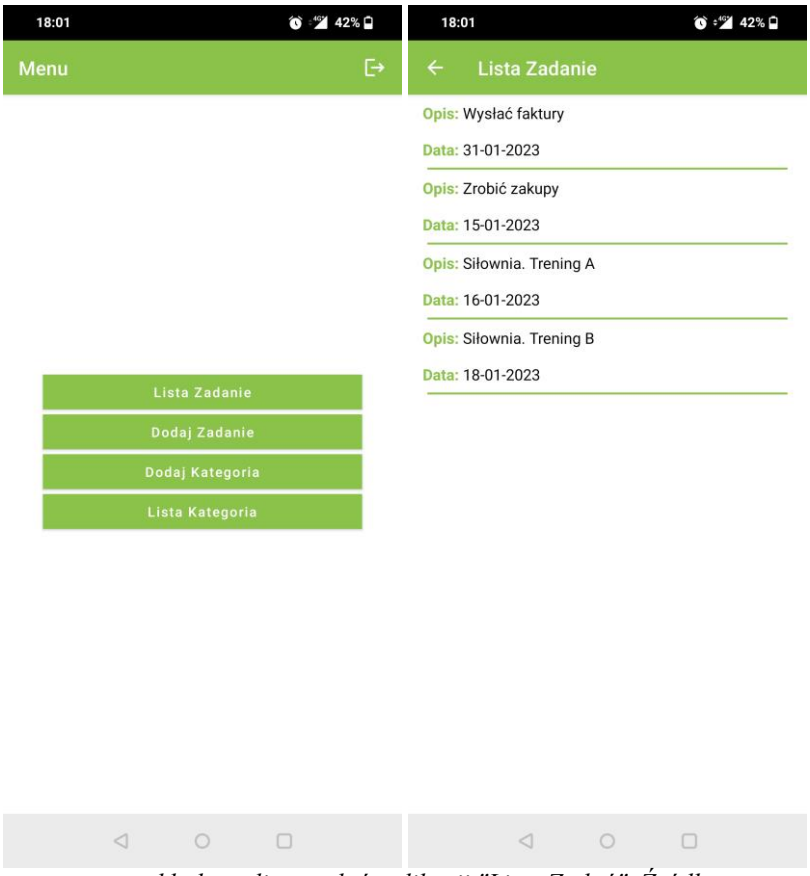
5.1.2 Efekt końcowy

Na podstawie określonej konfiguracji została wygenerowana aplikacja mobilna w której użytkownik może zarządzać swoimi zadaniami. Udostępnione funkcje w menu oferują:

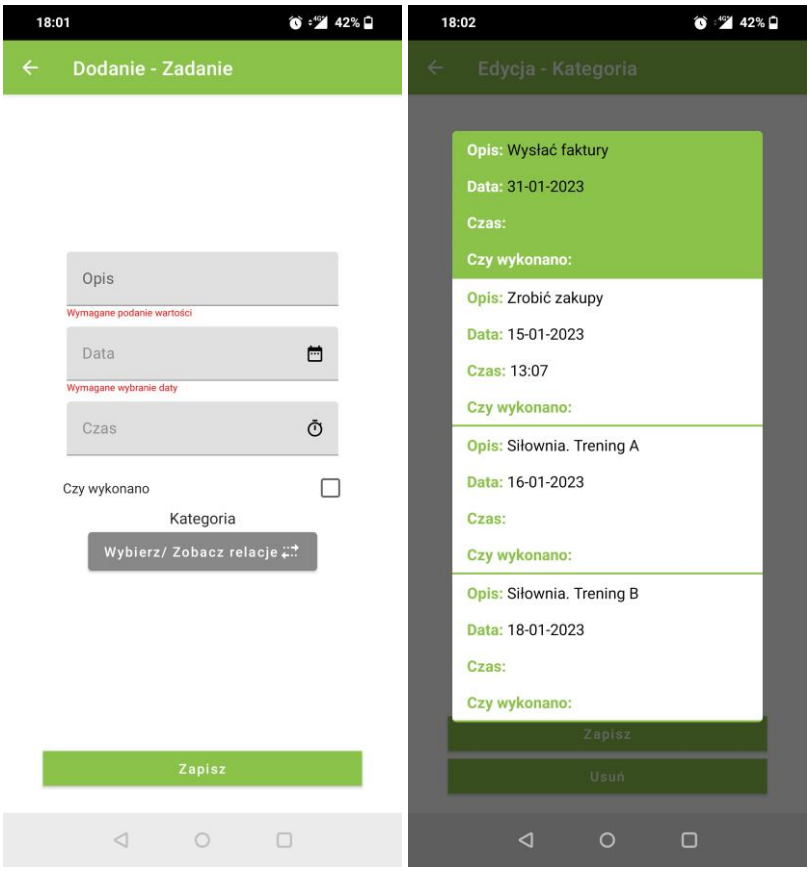
- Dodanie nowych zadań lub kategorii dla których wymagane jest podanie opisu oraz daty. Opcjonalnie można wybrać czas. Po zakończeniu można zaznaczyć za pomocą pola wyboru czy zostało zrealizowane. Dodatkowo zadania można usuwać ręcznie np. w ramach tygodniowych podsumowań. Dla pól jakimi są data i czas istnieje opcja wyboru wartości za pomocą standardowych kontrolek androidowych, dostępnych pod ikonkami kalendarza i zegara.
- Wylistowanie wszystkich zapisanych zadań lub kategorii prezentując tylko atrybuty główne. Kliknięcie w pojedynczy element przechodzi do ekranu edycji danych.

Oferowane również jest połączenie zadań z danymi kategoriami. Niewątpliwą zaletą w takiego typu aplikacjach jest działanie relacji obustronne. Z poziomu zadania możemy sprawdzić do jakiej

kategorii jest przypisane. Natomiast wybierając kategorię można przeszukać wszystkie zadania do niej należące. W prawym górnym rogu ekranu menu umieszczony jest przycisk pozwalający na wyjście z aplikacji. Styl został dostosowany według preferencji twórcy. Rysunek 24 oraz Rysunek 25 przedstawiają przykładowe zrzuty ekranu z wygenerowanego programu.



Rysunek 24. Ekran menu oraz przykładowa lista zadań aplikacji "Lista Zadań". Źródło: opracowanie własne



Rysunek 25. Ekran dodania nowego zadania oraz ekran obrazujący edycję relacji w aplikacji „Lista Zadań”. Źródło: opracowanie własne

5.2. Magazyn

Program mogący pomóc w zarządzaniu stanem małego magazynu w warunkach terenowych. Chcąc sparametryzować taką aplikację należałoby zdefiniować prostą strukturę określającą daną rzecz i jej stan. Określić można również lokalizacje takiego produktu np. na której półce regału się znajduję. Aplikacja powinna być przeznaczona dla wielu użytkowników, którymi zarządzał by kierownik magazynu.

5.2.1 Parametryzacja

Poszczególne opcje jakie zostały wybrane w celu utworzenia:

- Typ aplikacji: *dla wielu użytkowników*
- Grupy użytkowników: *Administrator*, *Pracownik*
- Struktura „*Produkt*” z atrybutami:
    - *Nazwa (wymagane, atrybut główny)*
    - *Ilość (wymagane, atrybut główny)*
    - *Data modyfikacji*
- Struktura „*Lokalizacja*” z atrybutami:
    - *Numer regału (wymagane, atrybut główny)*
    - *Numer półki (wymagane, atrybut główny)*

- „*Lokalizacja*” oraz „*Produkt*” zostały połączone relacjom.
- Możliwość *utworzenia kont tylko przez administratora*
- Opcja do *dodania* i *edytowania „Lokalizacja”* tylko dla administratora.
- Opcja do *dodania* i *edytowania „Produkt”* dla wszystkich grup.

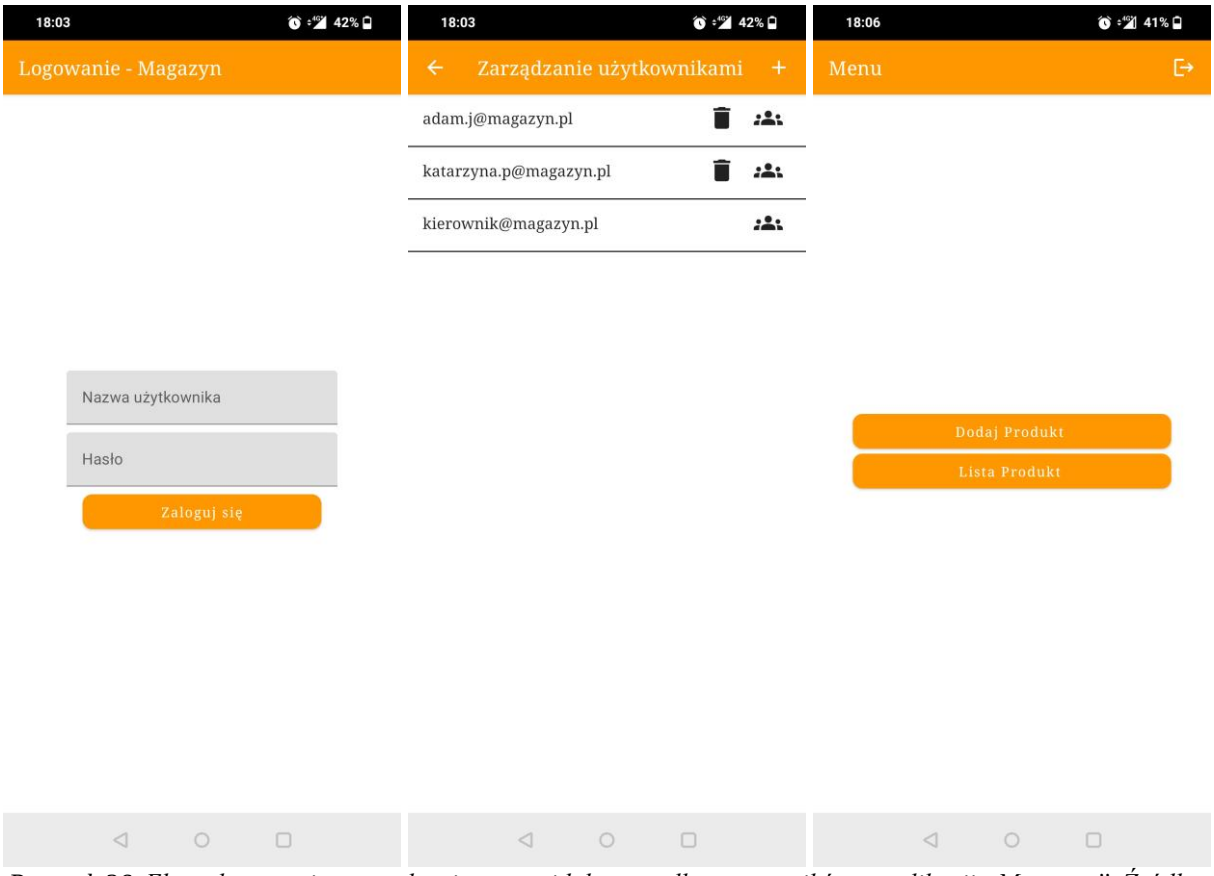
W Dodatku B została przedstawiona pełna konfiguracja opisywanej aplikacji.

5.2.2 Efekt końcowy

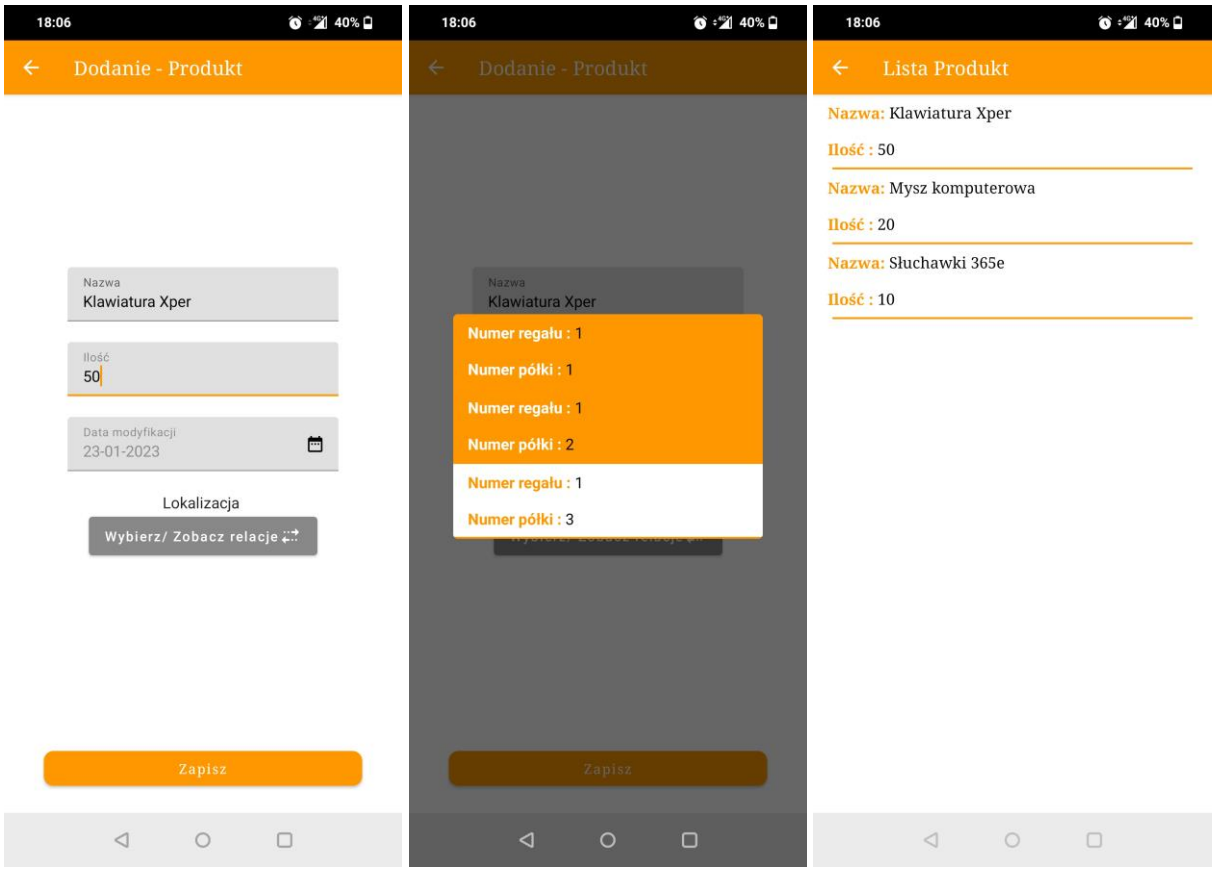
Aplikacja przeznaczona dla wielu użytkowników. Wymagana jest autoryzacja poprzez podanie adresu e-mail oraz hasła. Tworzenie, usuwanie oraz przypisywanie do grup zarządzane jest przez administratorów. Dzięki temu nikt w sposób niekontrolowany nie uzyska dostępu do danych i ich modyfikacji. Użytkownicy mogą przynależeć do dwóch grup: administrator (kierownika) i/lub pracownik (zawierającej ograniczenia). Opcje dostępne z poziomu menu umożliwiają:

- Wylistowanie i edytowanie lokalizacji określanej przez numer regału w magazynie i półki. (opcja dostępna tylko dla administratorów)
- Dodanie nowej lokalizacji (opcja dostępna tylko dla administratorów)
- Zarządzanie użytkownikami (opcja dostępna tylko dla administratorów)
- Dodanie nowego produktu z wymaganiem podania nazwy oraz ilości na stanie i przypisaniem lokalizacji. Opcjonalnie można określić datę modyfikacji. (opcja dla wszystkich użytkowników)
- Wylistowanie wszystkich produktów oraz ich edycji (np. w celu zmniejszenia/powiększenia ilości dostępnej, opcja dla wszystkich użytkowników)

  Zrzuty ekranu na Rysunku 26 przedstawiają ekran logowania, zarządzania użytkownikami oraz menu z poziomu użytkownika z grupy „Pracownicy” z widocznym ograniczonym dostępem do niektórych funkcji. Kolejny Rysunek 27 obrazuje modyfikację i wyświetlenie produktów oraz lokalizacji.



Rysunek 26. Ekran logowania, zarządzania oraz widok menu dla pracowników w aplikacji „Magazyn”. Źródło: opracowanie własne



Rysunek 27. Ekran dodawania nowego produktu, określania relacji między produktem, a lokalizacja i lista produktów w aplikacji „Magazyn”. Źródło: opracowanie własne

6. Przeprowadzone badania wśród potencjalnych użytkowników

Rozdział przedstawia wyniki badań jakie przeprowadzono udostępniając zainstalowany prototyp aplikacji na smartfonie Oneplus 7T Pro.

6.1. Cel badań i badana grupa

Grupa badana stanowiła cztery osoby, które na co dzień nie wykonują pracy zawodowej związanej z projektowaniem lub tworzeniem oprogramowania (radca prawny, student kierunku lekarskiego, tester gier komputerowych czy kierownik do spraw rozwoju). Przedział wiekowy badanych to pomiędzy dwudziestym piątym a trzydziestym rokiem życia. Celem badania było sprawdzenie między innymi:

- Poziomu zrozumienia instrukcji przekazywanych w kreatorze w celu utworzenia własnego rozwiązania
- Opini na temat sposobu definiowania poszczególnych opcji (użyteczność interfejsu graficznego)
- Czy aplikacja końcowa jest zgodna z wizją osoby badanej i czy jest użyteczna
- Czy osoba była by w stanie udostępnić swoją aplikację innym użytkownikom

6.2. Założenia i opis badań

W przeprowadzonych badaniach każda osoba otrzymała zadanie utworzenia za pomocą kreatora aplikacji mobilnej, której zadaniem byłoby kontrolowanie obowiązków domowych. Powinna ona dostarczać informacje kto z domowników jest odpowiedzialny za daną czynność w konkretnym pomieszczeniu. Ponadto poinformowano o braku ograniczeń w zastosowaniu opcji jakie oferuje kreator (np. dodatkowe informacje o zadaniach).

Przed wykonaniem zadania określono założenia końcowe:

- Przeznaczona dla wielu użytkowników
- Zdefiniowana min. jedna dodatkowa grupa
- Struktury i relacje:
    a) Minimum jedna struktura w której sparametryzowano pola określające opis czynności oraz pomieszczenia. Dodatkowo połączona z grupą użytkowników opisująca domowników
    b) Utworzone zostaną trzy struktury: pomieszczenie, zadanie i czynność wraz z atrybutami je opisujące, które zostaną odpowiednio połączone relacjami z dodatkową grupą użytkowników
- Zostanie dodany dostęp min. do opcji utworzenia i przeglądania struktur, w celu zapewnienia użyteczności.

Każdej z badanych osób zostały również zadane następujące pytania po wykonaniu zadania:

- Które kroki sprawiły najwięcej trudności w zrozumieniu lub definiowaniu?
- Które kroki okazały się najłatwiejsze w zrozumieniu lub definiowaniu?

- Czy byłbyś/ byłabyś w stanie udostępnić swoją aplikację innym osobom?
- Czy aplikacja jest zgodna z Twoją wizją?

6.3. Wyniki

Poniżej zaprezentowano skrócone wyniki badań, które zawierają opis utworzonego rozwiązania przez użytkownika i odpowiedzi na postawione pytania. Dodano również istotne spostrzeżenia autora pracy magisterskiej podczas obserwowania procesu tworzenia.

- Osoba 1

  Tabela 9 przedstawia główne cechy utworzonej aplikacji „Kontrola obowiązków domowych”:

Tabela 9. Cechy aplikacji utworzonej przez osobę 1. Źródło: opracowanie własne

Parametry	Wartości
Grupy użytkowników	„Domownicy”
Struktury	„Obowiązek” z polami: o „nazwa” (tekst) o „wykonawca” (tekst) o „częstotliwość” (tekst)
Relacje	Jedna relacja pomiędzy grupą domowników a obowiązkami
Menu	Opcje do wyświetlenia obowiązków (bez edycji) i dodanie nowego
Ustawienia administracyjne	Tworzenie konta na dwa sposoby. Początkowa grupa: „Domownicy”

  Odpowiedzi na pytania i spostrzeżenia:

  - Pod względem interfejsu, trudności sprawił sposób wyboru grup użytkowników w definicji ekranów (podrozdział 4.4.4) / grupa domyślna (podrozdział 4.4.6). Brak jednoznacznego wskazania czy można wybrać wiele/ jedną grupę.
  - Brak wskazań na najłatwiejsze elementy.
  - Badany przyznał, że nie miałby problemu z udostępnieniem swojej aplikacji innym osobom.
  - Końcowy efekt zakwalifikowano jako satysfakcjonujący

- Osoba 2

  Główne cechy stworzonej aplikacji „Czysty domek” zawiera Tabela 10.

Tabela 10. Cechy aplikacji utworzonej przez osobę 2. Źródło: opracowanie własne

Parametry	Wartości
Grupy użytkowników	„Domownicy zarządzający”, „Domownicy szeregowi”
Struktury	„Odkurzanie salonu” z polami: o „osoba odpowiedzialna” (tekst) o „częstotliwość” (liczba)

	○ „czy wykonano minimalne założenia częstotliwości” (tak/nie) „Wynoszenie śmieci” z polami: ○ „osoba odpowiedzialna” (tekst) ○ „częstotliwość” (liczba) ○ „czy wykonano minimalne założenia częstotliwości” (tak/nie) „Wycieranie kurzy” z polami: ○ „osoba odpowiedzialna” (tekst) ○ „częstotliwość” (liczba) ○ „czy wykonano minimalne założenia częstotliwości” (tak/nie)
Relacje	Utworzono relacje: ○ „Odkurzanie salonu” z „Domownicy szeregowi” ○ „Wynoszenie śmieci” z „Domownicy szeregowi” ○ „Wycieranie kurzy” z „Domownicy szeregowi”
Menu	Zdefiniowanie elementy menu: ○ Lista (z edycją) dla „Odkurzanie salonu” ○ Lista (z edycją) dla „Wynoszenie śmieci” ○ Lista (z edycją) dla „Wycieranie kurzy”
Ustawienia administracyjne	Tworzenie konta na dwa sposoby. Początkowa grupa: „Administrator”

Odpowiedzi na pytania i spostrzeżenia:

- Końcowy efekt nie spełniał wizji osoby badanej, która myślała że operacje dodania lub edytowania struktur zostaną dodane automatycznie. Wskazuje to na trudności z przekazaniem informacji na temat budowania menu (podrozdział 4.4.4)
- Problemem okazało się definiowanie struktury, które zostały podane jako już konkretne dane a nie jako szablon dla danego bytu (podrozdział 4.4.3).
- Dodatkowe opisy pól struktury nie są intuicyjne, aby je wyświetlić (podrozdział 4.4.3)
- Trudności z sposobem wyboru grup użytkowników w definicji ekranu (podrozdział 4.4.4) / grupa domyślna (podrozdział 4.4.6)
- Najłatwiejszy element: zdefiniowanie stylu (podrozdział 4.4.5)
- Brak problemu z udostępnieniem swojej aplikacji innym osobom (podrozdział 4.4.1).
- Określenie grupy początkowej jako „Administrator” może wskazywać na brak konkretnej informacji w kreatorze jakie są jej zadania (podrozdział 4.4.6)

- Osoba 3

Tabela 11 prezentuje cechy dla rozwiązania „Podział obowiązków”.

Tabela 11. Cechy aplikacji utworzonej przez osobę 3. Źródło: opracowanie własne

Parametry	Wartości
Grupy użytkowników	„Domownicy”

Struktury	„Zakupy” z polami: o „zrobione” (tak/nie) o „kto” (tekst) o „pomieszczenie” (tekst) „Zmywarka” z polami: o „zrobione” (tak/nie) o „kto” (tekst) o „pomieszczenie” (tekst) „Pranie” z polami: o „zrobione” (tak/nie) o „kto” (tekst) o „pomieszczenie” (tekst)
Relacje	Utworzono relacje: o „Zakupy” z „Domownicy” o „Zmywarka” z „Domownicy” o „Pranie” z „Domownicy”
Menu	Zdefiniowanie elementy menu: o Dodanie dla „Zakupy” o Lista (bez edycji) dla „Zmywarka” o Lista (bez edycji) dla „Pranie”
Ustawienia administracyjne	Tworzenie kont tylko przez administratorów. Początkowa grupa „domownicy”

Odpowiedzi na pytania i spostrzeżenia:

- Osoba nie była zadowolona z końcowego efektu, ponieważ aplikacja nie jest funkcjonalna. Operacje dodania nie zostały zdefiniowane. Trudności z przekazaniem informacji na temat budowania menu.
- Największa trudność w zrozumieniu: czym ma być struktura oraz w budowaniu elementów menu
- Brak wyjaśnienie czym jest atrybut główny
- Badany przyznał, że miałby problem z udostępnieniem swojej aplikacji.
- Najłatwiejszym krokiem: definiowanie stylu

- Osoba 4

Badany utworzył aplikacje „Obowiązki domowe” a jej cechy pokazane są w Tabela 12.

Tabela 12. Cechy aplikacji utworzonej przez osobę 4. Źródło: opracowanie własne

Parametry	Wartości
Grupy użytkowników	„Dzieci”, „Rodzice”, „Dziadkowie”

Struktury	„Zadanie” z polami: o „Zadanie” (tekst) o „Termin” (data) o „Odpowiedzialność” (tekst) o „Komentarz” (tekst)
Relacje	Utworzono relacje: o „Zadanie” z „Dzieci” o „Zadanie” z „Rodzicami” o „Zadanie” z „Dziadkowie”
Menu	Elementy menu: o Lista (z edycją) dla „ Zadania”
Ustawienia administracyjne	Tworzenie kont tylko przez administratorów. Początkowa grupa „Administrator”

Odpowiedzi na pytania i spostrzeżenia:

- Osoba była częściowo zadowolona z aplikacji końcowej. Spełniała jej wizję, natomiast nie dostarczała funkcjonalności
- Instrukcje odnośnie kroków na górze ekranu mało widoczne
- Łatwe definiowanie struktur
- Przyjazny interfejs użytkownika
- Badany oczekiwałby więcej instrukcji i przykładów
- Brak problemów z udostępnieniem swojej aplikacji.

6.4. Interpretacja wyników

Na podstawie zebranych wyników i obserwacji użytkowników podczas badania można wyodrębnić kilka segmentów, które należałoby poprawić lub zaprojektować inaczej:

- Instrukcje i przykłady

  Proces tworzenia powinien zawierać więcej instrukcji dotyczących poszczególnych kroków. Powinny one być również umieszczone w innym miejscu np. jako wyskakujące okienko, aby zmusić osobę do zapoznania się z nimi. Oprócz krótkich instrukcji dobrze byłoby zobrazować działanie i wykorzystanie tychże elementów w aplikacji generowanej (np. jak działają encje, do czego mogą być wykorzystywane grupy). Nazwa określająca encje („Struktura”) powinna zostać zmieniona na inne określenie np. „Szablon danych”, „Szablon struktury” itp.

- Struktury i relacje

  Główny obszar wymagający dostosowania w celu lepszego zrozumienia przez użytkowników. W ramach poprawy należałoby połączyć definiowanie struktur i relacje jako jeden obszar. Łączenie powinno odbywać się analogicznie jak dodanie pól do struktury, czyli oprócz wyboru typu pola wyświetlałyby się również już zdefiniowane encje i grupy użytkowników. W obecnej propozycji prototypu osoby badane często dodawały pole np. „wykonawca” nie wiedząc o

możliwości sparametryzowania takiej opcji w późniejszym kroku, często określając niejasność takiego podejścia.

- Metoda określania menu

  Definiowanie menu okazało się główną przyczyną niepowodzeń efektu końcowego. Udostępnienie opcji do określania pojedynczych ekranów w ramach jednej struktury było nie jasne i osoby nie dodawały opcji „Dodaj" co pozbawiało jakąkolwiek użyteczność. Aby poprawić ten proces można by domyślnie tworzyć wszystkie ekrany jak lista czy dodanie. Pozostałaby tylko i wyłącznie opcja z ograniczeniem dostępu dla danego ekranu z względu na grupę użytkownika.

- Sposób wybierania elementów lub ich zestawu

  Każda osoba badana wskazała na problem dotyczący interfejsu graficznego wyboru grupy lub funkcji ekranu. Nie precyzuje on czy można kliknąć w daną kontrolkę oraz czy można wybrać wiele lub jeden element. Naprawa powinna zaimplementować spopularyzowane kontrolki typu opcji wyboru (*ang. radio buton)* i zaznaczenia (*ang. checkbox)* jak wykonano to w ekranie do wyboru stylu.

- Grupy użytkowników

  Definiowanie grup nie stanowiło większego problemu. Każda osoba dodała grupy według swojego pomysłu i prawie wszyscy powiązali je jakąś relacją. Natomiast nikt nie wykorzystał tej funkcjonalności do ograniczenia dostępu do ekranów/ danych. Nie można jednoznacznie stwierdzić, iż jest to opcja nadmiarowa i niepotrzebna. Może wiązać się to z brakiem takiej potrzeby w zadanym poleceniu.

- Nawigacja

  Poruszanie się podczas etapu kreowania powinno uwzględnić menu pozwalające wrócić do żądanego kroku. Chcąc coś zmodyfikować osoby musiały cofać się poprzez przycisk „wstecz" co nie było zbyt komfortowe.

Po wykonaniu zadania i odpowiedzeniu na pytania, osobom została zaprezentowana aplikacja końcowa (dotycząca tego samego zagadnienia) utworzona przez autora pracy magisterskiej za pomocą kreatora. Przedyskutowano również wybory, które odbiegały od założeń i zademonstrowano funkcjonalności z których nie skorzystano. Wszyscy badani stwierdzili, że rozwiązanie jest interesujące z perspektywami rozwoju i dopracowania. Jako główne zastosowanie wskazali na aplikacje do kontrolowania jakiegoś prostego zbioru informacji, lecz bez konkretnych pomysłów.

7. Podsumowanie

Niniejsza praca magisterska dotyczyła opracowania koncepcji aplikacji mobilnej na system Android, która pomogłaby tworzyć własne proste rozwiązania na smartfony bez znajomości kodowania i technologii. Utworzony prototyp realizuje postawione wymagania w tej pracy, natomiast istnieje wiele możliwości i kierunków rozwoju jego funkcjonalności. Poniżej znajduję się krótka ocena wyników pracy.

Pierwszą i zasadniczą kwestią jest cel jaki chce się osiągnąć za pomocą kreatora aplikacji. Od niego zależy czy efekt końcowy spełni oczekiwania użytkownika. Analizując podobne rozwiązania i implementując własne można stwierdzić, że tego typu programy mają zawsze swoje ograniczenia. Ciężko jest zapewnić rozwiązanie uniwersalne, tak aby umożliwiało użytkownikowi utworzenie dowolnego typu aplikacji. Nawet jeśli oferowane byłoby takie rozwiązanie na rynku, to osoba bez znajomości wszystkich jego funkcji nie wykorzystałaby ich w sposób jaki zostały zaprojektowane. Wniosek ten pokazuje, że praca projektantów, architektów i programistów jest niezastąpiona przy tworzeniu nawet prostej aplikacji.

Kolejnym wnioskiem nasuwającym się podczas implementacji jest ścisłe określenie funkcjonalności. Tworząc można wymyślać i tworzyć kolejne opcje i możliwości bez kończącego się procesu programowania i udostępnienia produktu końcowego. Jest to spowodowane chęcią stworzenia każdej opcji jaką mógłby zaimplementować programista, co w obecnych czasach wydaje się być niemożliwe. W tego typu aplikacji nie ma ograniczeń.

Niewątpliwą trudnością jest opisanie poleceń dla użytkownika i utworzenie przyjaznego procesu kreowania, aby użytkownik czuł się komfortowo z instrukcjami jakie ma wykonać. W odniesieniu do wykonanych badań wynika, że należy przeprowadzać testy interfejsu graficznego i zrozumienia od samego początku, ponieważ coś co może wydawać się zrozumiałe na etapie projektowania, nie jest takim dla odbiorców. Takie testy powinno się powielać i na podstawie nich dopracowywać aplikacje aż do uzyskania satysfakcjonujących wyników. Zbieranie informacji od użytkownika do wygenerowania aplikacji powinno opierać się na bezwzględnym minimum bez nadmiarowych opcji.

Nie należy zakładać, że pierwsza próba utworzenia przez osobę swojej aplikacji będzie sukcesem. Jak każde oprogramowanie, wymagane jest udostępnienie jakiegoś rodzaju instrukcji. Użytkownik musi przejść przez proces pierwszy próbny raz, aby określić jak mógłby wykorzystać dane narzędzie i na co zwrócić uwagę przy drugiej i kolejnych próbach.

Małe ekrany w smartfonach jak i obecna dostępność technologii nie powinna być przeszkodą. Większe ekrany monitora czy laptopa dostarczają możliwość umieszczenia dużej ilości funkcji widocznych naraz, natomiast dobre i pomysłowe rozplanowanie ekranów na telefonie nie powinno wpływać na użyteczność. Jedyną dostrzeżoną wadą jest udostępnienie opcji dostosowania stylu, rozmieszczenia kontrolek i innych elementów na ekranie co mogłoby być trudne do osiągnięcia, aby zaoferować pełną dowolność na smartfonie. Ze względu na dostępność różnych technologii jest możliwe utworzenie sposobu przechowywania danych i implementacji innych opcji oferowanych przez kreator. Wymaga to tylko autorskiego pomysłu na jego wykonanie.

Bibliografia

[1] „APPER Make an App without coding. Easy and fast” [Online] https://play.google.com/store/apps/details?id=com.xpous.myapps&gl=PL [Data dostępu: 18.10.2021]

[2] „App Builder - Create own app (FREE App maker)” [Online] https://play.google.com/store/apps/details?id=com.letsappbuilder&gl=PL [Data dostępu: 18.10.2021]

[3] „MobEasy : Create Mobile Apps without coding” [Online] https://play.google.com/store/apps/details?id=com.mobeasy&gl=PL [Data dostępu: 18.10.2021]

[4] „One Signal” [Online] https://onesignal.com/ [Data dostępu: 18.10.2021]

[5] „PayPal” [Online] https://www.paypal.com/pl/webapps/mpp/home [Data dostępu: 18.10.2021]

[6] „Firebase” [Online] https://firebase.google.com/ [Data dostępu: 18.10.2021]

[7] „Test Lab App” https://play.google.com/store/apps/details?id=com.app.testpiwikapp&hl=pl&gl=PL [Data dostępu: 18.10.2021]

[8] „Survey Monkey” [Online] https://www.surveymonkey.com/ [Data dostępu: 18.10.2021]

[9] „Eventbrite” [Online] https://www.eventbrite.com/ [Data dostępu: 18.10.2021]

[10] „Shopify” [Online] https://www.shopify.com/ [Data dostępu: 18.10.2021]

[11] „How many smartphones are in the word?” [Online] https://www.bankmycell.com/blog/how-many-phones-are-in-the-world [Data dostępu: 15.12.2022]

[12] „Android Studio” https://pl.wikipedia.org/wiki/Android_Studio [Online] [Data dostępu: 15.12.2022]

[13] „Jetbrains” https://www.jetbrains.com/idea/ [Online] [Data dostępu: 15.12.2022]

[14] „Meet Android Studio” https://developer.android.com/studio/intro [Online] [Data dostępu: 15.12.2022]

[15] „Kotlin” https://kotlinlang.org [Online] [Data dostępu: 15.12.2022]

[16] „Kotlin język programowania” https://pl.wikipedia.org/wiki/Kotlin_(j%C4%99zyk_programowania) [Online] [Data dostępu: 15.12.2022]

[17] „Comparsion to Java” https://kotlinlang.org/docs/comparison-to-java.html [Online] [Data dostępu: 15.12.2022]

[18] „Android’s Kotlin-first approach” https://developer.android.com/kotlin/first [Online] [Data dostępu: 15.12.2022]

[20] „Jetpack Compose” https://developer.android.com/jetpack/compose [Online] [Data dostępu: 15.12.2022]

[21] „Why adopt Compose” https://developer.android.com/jetpack/compose/why-adopt [Online] [Data dostępu: 15.12.2022]

[22] „Firebase” https://en.wikipedia.org/wiki/Firebase [Online] [Data dostępu: 15.12.2022]

[23] „Mobile backend as a service” https://en.wikipedia.org/wiki/Mobile_backend_as_a_service [Online] [Data dostępu: 15.12.2022]

[24] „Wybierz bazę danych: Cloud Firestore lub Baza danych czasu rzeczywistego” https://firebase.google.com/docs/database/rtdb-vs-firestore [Online] [Data dostępu: 15.12.2022]

[25] „Firebase Products / Build” https://firebase.google.com/products-build [Online] [Data dostępu: 15.12.2022]

[26] „Firebase Products / Release & Monitor” https://firebase.google.com/products-release [Online] [Data dostępu: 15.12.2022]

[27] „Firebase Products / Engage” https://firebase.google.com/products-engage [Online] [Data dostępu: 15.12.2022]

[28] „Gradle” https://gradle.org/ [Online] [Data dostępu: 15.12.2022]

[29] „Apache Maven” https://maven.apache.org/ [Online] [Data dostępu: 15.12.2022]

[30] „Gradle” https://en.wikipedia.org/wiki/Gradle[Online] [Data dostępu: 15.12.2022]

[31] „Dagger” https://dagger.dev/ [Online] [Data dostępu: 15.12.2022]

[32] „Wstrzykiwanie zależności” https://pl.wikipedia.org/wiki/Wstrzykiwanie_zale%C5%BCno%C5%9Bci [Online] [Data dostępu: 15.12.2022]

[33] „Square” https://square.github.io/ [Online] [Data dostępu: 15.12.2022]

[34] „Dependency injection with Hilt” https://developer.android.com/training/dependency-injection/hilt-android [Online] [Data dostępu: 15.12.2022]

[35] „Dagger Hilt” https://dagger.dev/hilt/ [Online] [Data dostępu: 15.12.2022]

[36] „Encja (bazy danych)” https://pl.wikipedia.org/wiki/Encja_(bazy_danych) [Online] [Data dostępu: 27.12.2022]

[37] „CRUD” https://pl.wikipedia.org/wiki/CRUD [Online] [Data dostępu: 27.12.2022]

[38] „APK” https://pl.wikipedia.org/wiki/APK [Online] [Data dostępu: 28.12.2022]

Wykaz listingów

Wykaz tabel

Wykaz rysunków

Dodatek A: Konfiguracja aplikacji „Lista Zadań”


```
{
   "id":"Lista zadań 9c3ad49d",
   "administration":null,
   "entities":[
      {
         "fields":[
            {
               "name":"Opis",
               "required":true,
               "mainAttribute":true,
               "id":0,
               "type":"TEXT"
            },
            {
               "name":"Data",
               "mainAttribute":true,
               "type":"DATE",
               "required":true,
               "id":1
            },
            {
               "required":false,
               "type":"TIME",
               "name":"Czas",
               "id":2,
               "mainAttribute":false
            },
            {
               "mainAttribute":false,
               "required":false,
               "id":3,
               "type":"BOOLEAN",
               "name":"Czy wykonano"
            }
         ],
         "name":"Zadanie",
         "entityNo":0
      },
      {
         "fields":[
```

```
            {
               "id":0,
               "name":"Nazwa",
               "mainAttribute":true,
               "required":true,
               "type":"TEXT"
            }
         ],
         "name":"Kategoria",
         "entityNo":1
      }
   ],
   "groups":[

   ],
   "menuType":"PRIMARY",
   "name":"Lista zadań ",
   "relations":[
      {
         "id":0,
         "secondEntity":1,
         "require":false,
         "firstEntityId":0,
         "secondEntityFlag":true,
         "firstEntityFlag":true
      }
   ],
   "screens":[
      {
         "entityId":0,
         "name":"Lista Zadanie",
         "accessGroups":[

         ],
         "additionalName":"Zadanie",
         "additionalTypeForRead":"UPDATE_WITH_DELETE",
         "additionalAccessGroups":[

         ],
         "type":"READ",
         "forOwner":false
      },
      {
         "additionalTypeForRead":null,
```

```
      "name":"Dodaj Zadanie",
      "entityId":0,
      "additionalAccessGroups":[

      ],
      "accessGroups":[

      ],
      "additionalName":"Zadanie",
      "type":"CREATE",
      "forOwner":false
   },
   {
      "type":"CREATE",
      "additionalAccessGroups":[

      ],
      "additionalTypeForRead":null,
      "entityId":1,
      "name":"Dodaj Kategoria",
      "forOwner":false,
      "accessGroups":[

      ],
      "additionalName":"Kategoria"
   },
   {
      "accessGroups":[

      ],
      "forOwner":false,
      "entityId":1,
      "additionalName":"Kategoria",
      "type":"READ",
      "additionalTypeForRead":"UPDATE_WITH_DELETE",
      "name":"Lista Kategoria",
      "additionalAccessGroups":[

      ]
   }
],
"style":{
   "colorOfButton":1,
   "shapeOfButton":0,
```

```
        "fontStyle":0
      },
      "type":"SINGLE"
```

Dodatek B: Konfiguracja aplikacji „Magazyn”


```
{
   "id":"Magazynfb523c48",
   "administration":{
      "emailRequired":true,
      "possibleCreateAccountByAdminGroup":true,
      "possibleCreateOwnAccount":false,
      "adminGroup":{
         "id":1,
         "name":"Administrator"
      },
      "startGroup":{
         "name":"Administrator",
         "id":1
      }
   },
   "entities":[
      {
         "name":"Produkt ",
         "fields":[
            {
               "type":"TEXT",
               "id":0,
               "name":"Nazwa",
               "mainAttribute":true,
               "required":true
            },
            {
               "name":"Ilość ",
               "id":1,
               "required":true,
               "type":"NUMBER",
               "mainAttribute":true
            },
            {
               "id":2,
               "required":false,
               "name":"Data modyfikacji ",
               "mainAttribute":false,
               "type":"DATE"
            }
```

```
      ],
      "entityNo":0
   },
   {
      "entityNo":1,
      "fields":[
         {
            "mainAttribute":true,
            "id":0,
            "required":true,
            "type":"NUMBER",
            "name":"Numer regału "
         },
         {
            "mainAttribute":true,
            "required":true,
            "name":"Numer półki ",
            "type":"NUMBER",
            "id":1
         }
      ],
      "name":"Lokalizacja "
   }
],
"groups":[
   {
      "id":1,
      "name":"Administrator"
   },
   {
      "name":"Pracownik ",
      "id":2
   }
],
"menuType":"PRIMARY",
"name":"Magazyn",
"relations":[
   {
      "firstEntityFlag":true,
      "secondEntity":1,
      "firstEntityId":0,
      "secondEntityFlag":true,
      "id":0,
      "require":false
```

```
    }
],
"screens":[
    {
        "additionalAccessGroups":[
            {
                "name":"Administrator",
                "id":1
            },
            {
                "id":2,
                "name":"Pracownik "
            }
        ],
        "type":"CREATE",
        "entityId":0,
        "additionalName":"Produkt ",
        "additionalTypeForRead":null,
        "name":"Dodaj Produkt ",
        "forOwner":false,
        "accessGroups":[
            {
                "id":1,
                "name":"Administrator"
            },
            {
                "name":"Pracownik ",
                "id":2
            }
        ]
    },
    {
        "forOwner":false,
        "name":"Lista Produkt ",
        "type":"READ",
        "accessGroups":[
            {
                "id":1,
                "name":"Administrator"
            },
            {
                "name":"Pracownik ",
                "id":2
            }
```

```
    ],
    "entityId":0,
    "additionalAccessGroups":[
       {
          "name":"Administrator",
          "id":1
       },
       {
          "id":2,
          "name":"Pracownik "
       }
    ],
    "additionalTypeForRead":"UPDATE_WITH_DELETE",
    "additionalName":"Produkt "
 },
 {
    "name":"Lista Lokalizacja ",
    "type":"READ",
    "accessGroups":[
       {
          "name":"Administrator",
          "id":1
       }
    ],
    "forOwner":false,
    "entityId":1,
    "additionalName":"Lokalizacja ",
    "additionalTypeForRead":"UPDATE_WITH_DELETE",
    "additionalAccessGroups":[
       {
          "name":"Administrator",
          "id":1
       }
    ]
 },
 {
    "additionalTypeForRead":null,
    "entityId":1,
    "accessGroups":[
       {
          "name":"Administrator",
          "id":1
       }
    ],
```

```
            "type":"CREATE",
            "additionalAccessGroups":[
               {
                  "id":1,
                  "name":"Administrator"
               }
            ],
            "name":"Dodaj Lokalizacja ",
            "forOwner":false,
            "additionalName":"Lokalizacja "
         }
      ],
      "style":{
         "colorOfButton":2,
         "fontStyle":2,
         "shapeOfButton":1
      },
      "type":"MULTI"
   }
```