Politechnika Śląska
Wydział Matematyki Stosowanej
Kierunek Informatyka
Studia stacjonarne I stopnia

Projekt inżynierski

System obsługi Biblioteki Wydziału Matematyki Stosowanej

Kierujący projektem:
dr inż. Zdzisław Sroczyński

Autorzy:
Karolina Chrząszcz
Szymon Górnioczek
Tomasz Kryg

Gliwice 2018

Karolinie
Szymonowi
Tomkowi

Projekt inżynierski:

System obsługi Biblioteki Wydziału Matematyki Stosowanej

kierujący projektem: dr inż. Zdzisław Sroczyński

1. **Karolina Chrząszcz** – (1%)

   Projekt interfejsu aplikacji mobilnej, implementacja aplikacji na system Android

2. **Szymon Górnioczek** – (1%)

   —

3. **Tomasz Kryg** – (98%)

   Projekt interfejsu aplikacji mobilnej, implementacja aplikacji na system iOS

Podpisy autorów projektu

1. .....
2. .....
3. .....

Podpis kierującego projektem

.....

Oświadczenie kierującego projektem inżynierskim

Potwierdzam, że niniejszy projekt został przygotowany pod moim kierunkiem i kwalifikuje się do przedstawienia go w postępowaniu o nadanie tytułu zawodowego: inżynier.

Data Podpis kierującego projektem

Oświadczenie autorów

Świadomy/a odpowiedzialności karnej oświadczam, że przedkładany projekt inżynierski na temat:
System obsługi Biblioteki Wydziału Matematyki Stosowanej
został napisany przez autorów samodzielnie.
Jednocześnie oświadczam, że ww. projekt:

- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2000 r. Nr 80, poz. 904, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/am w sposób niedozwolony,
- nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów wyższej uczelni lub tytułów zawodowych.
- nie zawiera fragmentów dokumentów kopiowanych z innych źródeł bez wyraźnego zaznaczenia i podania źródła.

Podpisy autorów projektu

1. Karolina Chrząszcz, nr albumu:252525, ... (podpis:) .....
2. Szymon Górnioczek, nr albumu:252252, ... (podpis:) .....
3. Tomasz Kryg, nr albumu:225183, ... (podpis:) .....

Gliwice, dnia .....

Spis treści

Wstęp

Książka – jeden z najbardziej podstawowych przedmiotów. Każdy z nas, nawet jeśli nie lubi czytać, niejednokrotnie w swoim życiu z jakiejś korzystał. Może służyć rozrywce, ale przede wszystkim szeroko pojętej edukacji. Każda przeczytana książka pomaga rozwinąć myślenie jak i kreatywność czytelnika. Każde przeczytane zdanie rozwija zdolność wypowiadania się. Każde przeczytane słowo pozwala poszerzyć zdolności językowe.

W dzisiejszych czasach, książki są coraz częściej pomijane kosztem innych rozrywek, takich jak gry komputerowe, kino czy telewizja, które z roku na rok wypychają książkę na dalszy plan. Jak wynika z raportu Biblioteki Narodowej [1], w roku 2016 jedynie 37% Polaków przeczytało choć jedną książkę. Mogłoby się więc wydawać, że wchodzenie na rynek książek nie jest najlepszym pomysłem. Jednak istnieją książki, które posiadają często wiadomości niezmienne i nie służą rozrywce, a przede wszystkim pozyskiwaniu wiedzy. Są to między innymi książki z działów ścisłych dotyczących bezpośrednio matematyki. Przykładowo ciąg Fibonacciego omówiony w roku 1202 przez Leonarda z Pizy, do dziś został opisany i wykorzystany w wielu książkach związanych z różnorodnymi dziedzinami. Z przytoczonego przykładu wynika, że książki dotyczące matematyki czy też fizyki, bardzo często zawierają dużo wiedzy, nawet w czasach współczesnych. Wszystko ewoluuje, jednak prawa natury pozostają niezmienne.

Współcześnie doszliśmy do punktu, w którym książka zaczyna mieć znaczenie przede wszystkim edukacyjne. Na uczelniach całego świata, książki są synonimem wiedzy, ponieważ bardzo często profesorzy i doktorzy kształcili się za pomocą takich samych (lub być może tych samych) materiałów co ich dzisiejsi uczniowie. Z tego powodu biblioteki – szczególnie na początku semestru – muszą zmagać się z dużymi ilościami studentów, chcącymi poszukać potrzebnych im książek oraz sprawdzić ich ilość i dostępność, aby później je wypożyczyć. Najczęściej staje się to problemem dla pracowników biblioteki jak i dla studentów. Bibliotekarze są przez to bardzo zabiegani, natomiast uczniowie są zdenerwowani sporymi kolejkami, w których czekają często tylko po to, by zadać pytanie.

Skupiając się na książkach naukowych i na własnym doświadczeniu uczelnianym,

gdzie wypożyczanie książek związanych z matematyką jest czymś powszechnym, postanowiliśmy stworzyć system dla Biblioteki Wydziału Matematyki Stosowanej na Politechnice Śląskiej w Gliwicach. Głównym celem takiego systemu byłoby posiadanie informacji o zbiorach biblioteki wydziałowej i łatwe udostępnianie ich studentom. Podstawą byłby prosty interfejs dla administratora uzupełniającego pozycje książek w systemie, jak i dla studenta chcącego szybko sprawdzić dostępność wybranej książki w bibliotece bez potrzeby wychodzenia z domu.

Jednym z typów aplikacji byłaby aplikacja webowa. Musiałaby zapewnić administratorowi pracującemu w bibliotece prostą obsługę bazy danych.

Drugim typem aplikacji obsługujących bibliotekę będą aplikacje na smartfony. Zostaną stworzone dwie aplikacje mobilne, każda na oddzielny system: Android oraz iOS w celu poszerzenia grupy odbiorców na wydziale. Pozwolą one na przeszukanie biblioteki pod względem dowolnej szukanej frazy oraz dostarczą informacji o wszystkich książkach i ich dostępności w bibliotece.

1. Aplikacja na system Android

1.1. Wykorzystane narzędzie

Aplikacja na urządzenia mobilne z systemem Android została napisana w języku *Java 8* w programie *Android Studio* w wersji 3.0.1.

Java jest uniwersalnym językiem obiektowym stworzonym w 1996 roku przez Jamesa Goslinga, Billy'ego Joya oraz Guya Steele [10]. Cechuje go prostota, celem twórców było, aby każdy mógł z łatwością osiągnąć płynność w korzystaniu z *Javy*. Jest mocno powiązana z językami programowania *C* oraz *C++* [10].

AndroidStudio jest oficjalnym środowiskiem programistycznym do tworzenia aplikacji na system Android, ogłoszonym w 2013 roku przez Google [15]. Wersja 3.0.1 weszła w życie w październiku 2017 roku wprowadzając, między innymi, wsparcie najnowszej wersji Android 8 [11]. Widać, że do tworzenia aplikacji mobilnej zostały wybrane najnowsze i jedne z najbardziej znanych narzędzi.

1.2. Wymagania systemowe

Aplikację można zainstalować na telefonach komórkowych oraz tabletach z systemem Android w wersji minimum 5.0 Lollipop (API 21). Według statystyk z dnia 5 lutego 2018 roku [12] przedstawionych na zdjęciu 1 najwięcej użytkowników korzysta z wersji Marshmallow 6.0. Jednak duży procent nadal używa wersji Lollipop, nie jest to na tyle znacząca różnica, aby zignorować tę wersję. Aplikację będzie mogło używać około 24% więcej użytkowników.

1.3. Model danych

Aplikacja posiada następujący model danych stworzony na podstawie informacji dostarczanych z API:

- `Book` – klasa reprezentująca obiekt książki:
  - `title` (`String`) – tytuł,

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.7%
4.2.x		17	2.6%
4.3		18	0.7%
4.4	KitKat	19	12.0%
5.0	Lollipop	21	5.4%
5.1		22	19.2%
6.0	Marshmallow	23	28.1%
7.0	Nougat	24	22.3%
7.1		25	6.2%
8.0	Oreo	26	0.8%
8.1		27	0.3%

Data collected during a 7-day period ending on February 5, 2018.
Any versions with less than 0.1% distribution are not shown.

Rysunek 1: Procent urządzeń z uruchomioną daną wersją Androida

– `responsibility` (`String`) – autorzy,
– `year` (`Integer`) – rok wydania,
– `volume` (`String`) – tom,
– `availability` (`String`) – dostępność,
– `type` (`String`) – typ pozycji,
– `isbnWithIssn` (`String`) – numer ISBN/ISSN,
– `facultySignature` (`String`) – sygnatura biblioteki MS,
– `mainSignature` (`String`) – sygnatura biblioteki głównej,
– `categories` (`List<Category>`) – kategorie przypisane do książki.

- `CategoryResponse` – klasa reprezentująca kategorie:
  - `category` (`Category`) – kategoria,
  - `subcategories` (`List<Category>`) – podkategorie przypisane do kategorii.
- `Category` – klasa reprezentująca obiekt kategorii:
  - `categoryId` (`String`) – id kategorii,
  - `name` (`String`) – nazwa kategorii.
- `Dictionary` – klasa reprezentująca obiekt "słownik" dla książki:
  - `bookTypes` (`String[]`) – typy pozycji (podręcznik, zbiór zadań, inny),
  - `bookAvailabilities` (`String[]`) – dostępność (dostępna, wypożyczona, czytelnia).

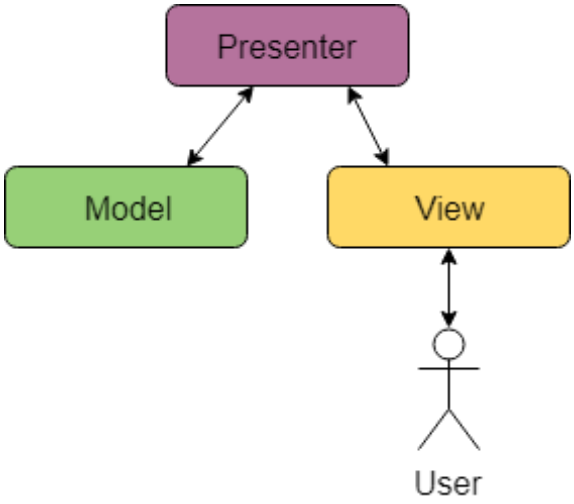
1.4. Realizacja projektu

Poniższe rozdziały opiszą budowę poszczególnych obiektów klasy `Activity`, które w dalszej części pracy będą nazywane aktywnościami.

W celu monitorowania działania aplikacji po wdrożeniu, został do niej przyłączony *Crashlytics* [13]. Jest to jedno z najlepszych urządzeń, które pozwala na śledzenie działania aplikacji. Biblioteka ta jest częścią platformy *Fabric*. Poza analizą ilości użytkowników w danych okresie czasu, pozwala na bieżąco śledzić błędy krytyczne powodujące zatrzymanie działania aplikacji. System Android jest jednym z najbardziej znanych i najczęściej używanych systemów, co niestety rodzi pewne problemu. Na przykład ilość urządzeń oraz ich różnorodność. Aplikacja wspiera cztery główne wersje systemu oraz wszystkie rozmiary telefonów oraz tabletów. Bardzo trudno jest dobrze przetestować taki produkt i znaleźć wszystkie możliwe niepożądane działania. Stąd pomysł na wprowadzenie Crashlyticsa. W każdej chwili można sprawdzić na stronie fabric.io raport błędów wysłany przez bibliotekę. Między innymi zostają wyświetlone data, czas i ilość wystąpienia konkretnego błędu, model urządzenia oraz wersja Androida, na którym wybrany błąd się pojawił, a także sam log błędu.

1.4.1. ModelViewPresenter

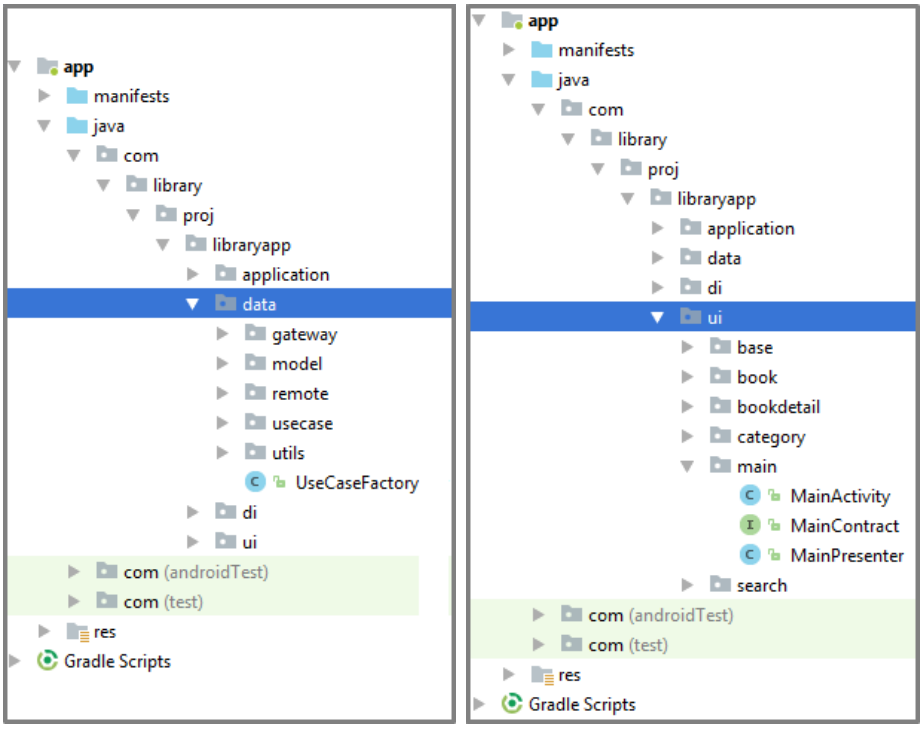
Projekt powstał w oparciu o strukturę *Model View Presenter*. *MVP* jest wzorcem projektowym stworzonym na podstawie wzorca Model View Controller. Podobnie jak w *MVC*, Model odpowiada za reprezentację obiektów danych, a View za wyświetlanie informacji użytkownikowi. Presenter natomiast jest odpowiedzialny za odseparowanie modelu od widoku. Podejmuje decyzję o tym, co powinno być wyświetlone użytkownikowi. Na podstawie dostarczonych danych przez Model, zwraca je w reprezentacyjnej formie do View. W przeciwieństwie do Controllera z MVC, nie posiada w sobie żadnych elementów UI [16]. Rysunek 2 przedstawia uproszczony model *MVP*.



Rysunek 2: Związek między odpowiednimi elementami *MVP*

W projekcie Model został odseparowany przez przeniesienie wszystkich klas związanych z pobieraniem/wysyłaniem danych do paczki `data`, co widać na rysunku 3. Natomiast elementy widoku znajdują się w paczce UI. W nich występuje również konkretny podział na klasy View oraz Presenter. Dodatkowo występuje tu interfejs Contract, który łączy w sobie interfejsy Presentera i View. Użycie interfejsów umożliwia zarządzanie widokiem z poziomu Presentera, bez konieczności posiadania elementów widoku w nim.

W celu uproszczenia i polepszenia czytelności kodu w projekcie została również użyta biblioteka *Dagger2* [17]. Jest to implementacja wzorca Dependency Injection, czyli wstrzykiwania zależności. Celem tego podejścia jest minimalizacja tworzenia obiektów poprzez



Rysunek 3: Podział projektu z uwzględnieniem struktury *MVP*

```
new ClassName()
```

oraz stworzenie repozytorium obiektów, które możemy wstrzyknąć w dowolnym miejscu aplikacji [15].

1.4.2. Pobieranie i wysyłanie danych

Do pobierania i wysyłania danych oraz łączenia się z serwerem wykorzystano biblioteki *Retrofit* [18] oraz *RxJava2* [19].

Retrofit jest darmową biblioteką przeznaczoną do korzystania z API REST z poziomu Androida. Dzięki metodom oraz anotacjom zawartym w bibliotece budowanie zapytań jest szybkie, proste i przede wszystkim przejrzyste (rysunek 4). Jej ogromną zaletą jest również wbudowana obsługa GSON, która zapewnia automatyczne mapowanie obiektów POJO na JSON i odwrotnie [15].

Natomiast biblioteka *RxJava* rozszerza wzorzec Observable, w celu umożliwienia prostej obsługi sekwencji danych/zdarzeń. Umożliwia odpowiednie składanie sekwencji, jednocześnie eliminując możliwe błędy związane ze złą synchronizacją zdarzeń oraz działaniem na wielu wątkach [19].

```
public interface ApiService {

    @POST(ApiConfig.Book.PATH)
    Observable<List<Book>> getBooks(@Body BookRequestData bookRequestData);

    @GET(ApiConfig.Category.PATH)
    Observable<List<CategoryResponse>> getAllCategories();

    @GET(ApiConfig.Dictionary.PATH)
    Single<Dictionary> getDictionary();
}
```

Rysunek 4: Fragment klasy *ApiService* zawierającej metody pobrania książek, kategorii oraz słownika z API

1.4.3. Zarządzanie widokami

Widoki zostały stworzone w plikach z rozszerzeniem `xml`. Przekazanie ich do klas zarządzającymi aktywnościami odbyło się z użyciem biblioteki *ButterKnife* [14]. Jest to bardzo prosta, darmowa biblioteka, która generuje kod związany z dostępem do UI na podstawie anotacji [15]. Dzięki niej łatwiej jest dbać o czystość kodu, niepotrzebne jest używanie `findViewById`, a podpięcie listenerów zajmuje mniej miejsca i jest bardziej czytelne, co widać na obrazku 5.

```
@BindView(R.id.search_availability_btn)
Button searchAvailabilityBtn;
@BindView(R.id.search_category_btn)
Button searchCategoryBtn;

@BindViews({R.id.search_title_et, R.id.search_author_et, R.id.search_inventory_number_et,
        R.id.search_signature_et, R.id.search_main_signature_et, R.id.search_year_et,
        R.id.search_volume_et})
List<EditText> allEtFields;

@OnClick(R.id.search_refresh_iv)
public void onRefreshClick() {
    downloadData();
}

@OnClick(R.id.search_delete_iv)
public void onClearClick() {
    clearAllFields();
}
```

Rysunek 5: Przykład zastosowania biblioteki *ButterKnife* w klasie *SearchActivity*

1.4.4. BaseActivity

Pierwszą, a zarazem najważniejszą klasą obiektu `Activity` jest klasa abstrakcyjna `BaseActivity`, którą wszystkie pozostałe aktywności rozszerzają. W całej paczce `base` znajdują się także interfejs `BaseView`, który aktywność implementuje oraz interfejs i klasę `Presenter`. Jest to zbiór podstawowych klas i interfejsów, które kolejne klasy będą rozszerzać bądź implementować. Poza podstawowymi metodami, takimi jak:

```
getLayoutRes();
performFieldInjection(ActivityComponent activityComponent);
getActivityComponent();
```

które przypominają o konieczności wstrzyknięcia instancji klasy do prezentera, została zaimplementowana również poniższa metoda:

```
final void onDetachView() {
    view = null;
    if(compositeDisposable != null
        && !compositeDisposable.isDisposed()) {
        compositeDisposable.dispose();
    }
}
```

Jej celem jest przerwanie wszystkich połączeń z API, gdy widok zostanie zniszczony, na przykład w przypadku, gdy użytkownik wyłączy aplikację.

1.4.5. MainActivity

Celem `MainActivity` jest wyświetlenie powitalnego ekranu użytkownikowi, tak zwanego *SplashScreena*, który po 1.5 sekundy znika i otwiera widok wyszukiwania. W przyszłości w tym miejscu można dodać opcję logowania lub utworzenia nowego konta do aplikacji.

Cały widok, podobnie jak pozostałe, zawiera się w `ConstraintLayout` [20]. Jest to jeden z najnowszych obiektów typu `ViewGroup` wprowadzony razem z wersją 3.0 Android Studio. Do tej pory najczęściej używanymi elementami były `RelativeLayout` oraz `LinearLayout`, które działają prosto i intuicyjnie, ale mimo wszystko są mocno ograniczone. `ConstraintLayout` jest stworzony przede wszystkim do ułatwienia

tworzenia widoków przy pomocy wbudowanego narzędzia *Layout Editor's*. Umożliwia tworzenie pojedynczych widoków, których każdą z krawędzi można pozycjonować względem dowolnie wybranych innych elementów znajdujących się w grupie. Następnie umożliwia tworzenie łańcuchów widoków i pozycjonowanie ich tak, jakby były jednym obiektem, bez konieczności dodawania i zagnieżdżania kolejnego obiektu typu `ViewGroup`.

1.4.6. SearchActivity

Jest to aktywność otwierana bezpośrednio z poprzedniej, po upływie danego czasu.

Rysunek 6: Projekt ekranu wyszukiwania książek

Jej celem jest umożliwienie wprowadzenia filtrów szukanych książek przez użytkownika. Widok składa się z customowego elementu typu `Toolbar`, elementów `TextInputEditText`, w które użytkownik może wprowadzić tekst oraz obiektów `Button` widocznych na wstępnym projekcie 6.

`Toolbar` został dodany w celu poprawienia czytelności aplikacji, zwiększenia intuicyjnego działania oraz ułatwienia nawigacji pomiędzy widokami. Każda aktywność posiada swój `Toolbar`. W przypadku `SearchActivity` znajduje się tam tekst z informacją, na którym widoku znajduje się użytkownik, po lewej stronie ikona do odświeżenia informacji przychodzących z API (np. typ, dostępność książki oraz kategorie w przypadku braku internetu nie zostaną pobrane), po prawej ikona usunięcia wszystkich, do tej pory, wprowadzonych danych.

Obiekt `TextInputEditText` [21] jest podklasą obiektu `EditText`. Różni się od tradycyjnego wejściowego pola tekstowego tym, że w momencie pisania oraz po wprowadzeniu tekstu przez użytkownika powyżej, mniejszą czcionką znajduje się podpowiedź, która wcześniej była wyświetlana. W tym wypadku jest to informacja, co należy w dane pole wpisać.

Filtry takie jak dostępność książki, typ pozycji oraz kategorię można wybrać po naciśnięciu obiektu `Button`. W przypadku dwóch pierwszych zostanie wyświetlony `Dialog` z listą możliwości do wyboru. W klasie odpowiedzialnej za tworzenie odpowiedniego dialogu znajduje się interfejs, dzięki któremu w łatwy sposób następuje przekazanie wybranych elementów do aktywności.

```
public interface OnDictionaryDialogResult {
    void finish(String selectedItem);
}
```

Po naciśnięciu na element z listy następuje wywołanie metody `finish()` zaimplementowanej w aktywności.

```
adapter.getOnDictionaryItemClickSubject().subscribe(
        dictionary -> {
            onDictionaryDialogResult.finish(dictionary);
            dismiss();
        });
```

`SearchActivity:`

```
dictionaryDialog.setOnDictionaryDialogResult(result -> {
        selectedType = result;
```

```
        searchTypeBtn.setText(result);
    });
```

Natomiast wybór kategorii odbywa się w osobnej aktywności.

1.4.7. CategoryActivity

Rysunek 7: Projekt ekranu wyboru kategorii

Aktywność do wyboru kategorii składająca się z obiektu `Toolbar` oraz obiektu `ExpandableListView`. Elementami są widoki składające się z obiektu `CheckBox` oraz `TextView` z nazwą kategorii, podobne do tych na widocznych na rysunku 7. Podobnie jak poprzednio, w Toolbarze znajduje się nazwa aktualnego widoku, po prawej stronie ikona do usuwania wszystkich do tej pory zaznaczonych elementów, po lewej stronie strzałka powrotu do poprzedniego widoku.

Podczas powrotu do poprzedniej aktywności, zaznaczone elementy są przekazane w obiekcie `Intent`.

```
@Override
```

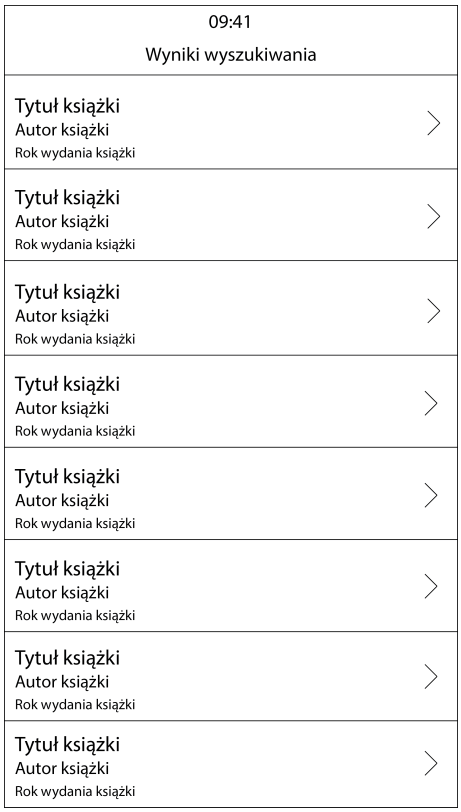
```
public void onBackPressed() {
    Intent intent = new Intent();
    intent.putParcelableArrayListExtra(ON_BACK_CATEGORIES_EXTRA, categories);
    setResult(RESULT_OK, intent);
    finish();
}
```

Jest to możliwe dzięki implementacji przez obiekt `Category` interfejsu `Parcelable`. Aby instancja obiektu mogła zostać zapisana i przechowana w obiekcie `Parcel` konieczne jest stworzenie statycznej zmiennej `CREATOR`.

```
public class Category implements Parcelable {

...

public static final Creator<Category> CREATOR = new Creator<Category>() {
        @Override
        public Category createFromParcel(Parcel in) {
            return new Category(in);
        }

        @Override
        public Category[] newArray(int size) {
            return new Category[size];
        }
    };

    protected Category(Parcel in) {
        categoryId = in.readString();
        name = in.readString();
        isChecked = in.readByte() != 0;
        isExpanded = in.readByte() != 0;
        subcategories = in.createTypedArrayList(Category.CREATOR);
    }
```

```
...

    @Override
    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeString(categoryId);
        parcel.writeString(name);
        parcel.writeByte((byte) (isChecked ? 1 : 0));
        parcel.writeByte((byte) (isExpanded ? 1 : 0));
        parcel.writeTypedList(subcategories);
    }
}
```

1.4.8. BookActivity



Rysunek 8: Projekt ekranu listy książek

Celem `BookActivity` jest wyświetlenie wszystkich książek, zwróconych przez API. Został użyty do tego element `RecyclerView`. Pozwala on przedstawić dużą liczbę danych na ograniczonym ekranie, podobnie jak na rysunku 8. Zarządzanie poszczególnymi elementami odbywa się przy użyciu adaptera. W adapterze zapamiętane są wszystkie elementy, w tym przypadki lista książek `List<Book>`. Dla każdej z nich tworzony jest osobny widok z użyciem nadrzędnego szablonu zapamiętanego w `ViewHolderze`. W adapterze do poszczególnych pól, jak na przykład `TextView titleTv` zostaje przypisany tytuł konkretnej książki. Dzięki zapamiętywaniu dwóch pozycji elementu – pozycja elementu w liście wszystkich obiektów, pozycja elementu z perspektywy `LayoutManagera` – możliwe jest odświeżanie całego widoku w przypadku jakichkolwiek zmian w danych.

W aplikacji API dostarcza ograniczoną ilość książek. W przypadku powolnego internetu lub urządzenia nie ma sensu czekać aż serwer zwróci wszystkie możliwe wyniki, a następnie czekać aż aplikacji je przetrawi i wyświetli. Stąd na początku zostaje pobrane np. pierwszych 50 książek. Po doscrollowaniu listy do końca przez użytkownika pobieranych jest kolejnych 50 książek, przy czym poprzednie wyniki są nadal wyświetlane.

```
bookRv.addOnScrollListener(getOnScrollListener());

...

private RecyclerView.OnScrollListener getOnScrollListener() {
    return new RecyclerView.OnScrollListener() {
        @Override
        public void onScrollStateChanged(RecyclerView recyclerView,
        int scrollState) {
            LinearLayoutManager manager = (LinearLayoutManager)
                recyclerView.getLayoutManager();
            if (isScrollable(scrollState, manager)) {
                currentPage++;
                setupPagination();
                getPresenter().getBooks(bookRequestData);
            }
```

```
        }

        private boolean isScrollable(int scrollState,
            LinearLayoutManager layoutManager) {
            return isEndOfScrollView(layoutManager)
            && scrollState == SCROLL_STATE_IDLE;
}

        private boolean isEndOfScrollView(LinearLayoutManager layoutManager) {
            return layoutManager.findLastVisibleItemPosition()
            == layoutManager.getItemCount() - 1;
        }
    };
}

@Override
public void refreshBooks(List<Book> books) {
    allBooks.addAll(books);
    if (bookRv.getAdapter() != null) {
        bookRv.getAdapter().notifyDataSetChanged();
    }
}
```

Metoda `isEndOfScrollView(LinearLayoutManager layoutManager)` wykorzystuje pozycję ostatniego widocznego elementu i porównuje ją z pozycją ostatniego elementu w całej liście. Jeśli znajdujemy się na końcu listy zostanie wysłane zapytanie do API o kolejną porcję książek. Po odebraniu odpowiedzi lista z wszystkimi (do tej pory) książkami zostaje zwiększona o nowe książki, a widok odświeżony przez wywołanie metody `notifyDataSetChanged()` na adapterze `RecyclerView`.

W tej aktywności `Toolbar` zawiera nazwę aktualnego widoku oraz po lewej stronie ikonę powrotu do widoku wyszukiwania książek.

Rysunek 9: Projekt ekranu szczegółów książki

1.4.9. BookDetailsActivity

Ostatnia aktywność zawiera w sobie szczegółowe informacje o książce (rys. 9). Można do niej przejść przez kliknięciu w element na liście wszystkich wyników. Tak jak poprzednie widoki zawiera `Toolbar` z nazwą aktualnego widoku oraz z ikoną powrotu do listy znalezionych książek. Informacje o książce wyświetlane są w polach `TextView`.

1.5. Interfejs użytkownika

Głównym celem aplikacji było ułatwienie studentom Wydziału Matematyki Stosowanej korzystanie z biblioteki. W tym celu powstała aplikacja na jeden z największych systemów Android. Jednocześnie celem aplikacji była jak największa przejrzystość i intuicyjność, żeby osoby nie korzystające na co dzień z dotykowych telefonów mogły używać jej bez problemów. Projekt był konsultowany na bieżąco z osobą zarządzającą biblioteką. Jednolity `Toolbar` pozwala na łatwą nawigację po aplika-

cji, a dobór odpowiednich kolorów zagwarantował spójność z aplikacją tworzoną na system iOS.

1.6. Splash screen

Aplikacja po uruchomieniu wyświetla ekran powitalny (rys. 10).

Rysunek 10: Ekran startowy aplikacji

Ekran po 1.5 sekundy znika i nie jest możliwe wrócenie do niego po naciśnięciu przycisku *back*.

1.6.1. Wyszukiwarka książek

Ekranu wyszukiwania książek umożliwia użytkownikowi między innymi wpisanie żądanej treści w odpowiednie pola za pomocą klawiatury systemowej (rys. 11). Trzy pola znajdujące się najniżej umożliwiają wybranie pozycji z listy. Dwa pierwsze otwierają dialog (rys. 12), ostatnie przenosi do nowej aktywności z listą kategorii, jak na rysunku 13.

Rysunek 11: Wyszukiwarka książek

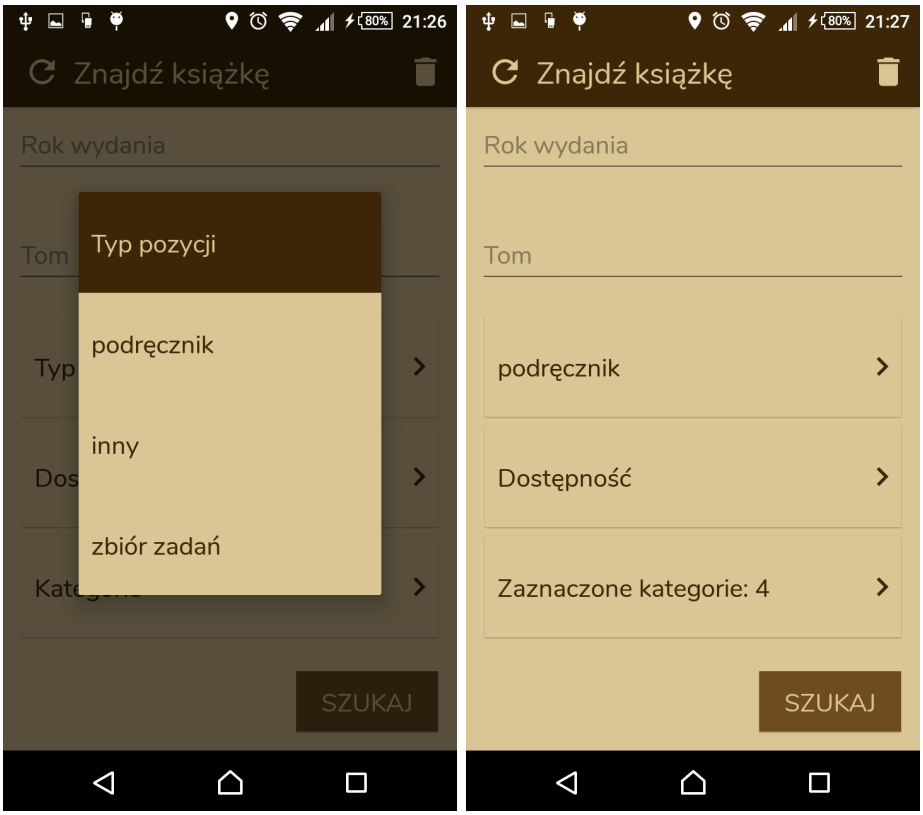
Do poprawnego pobrania danych oraz wyszukania książek aplikacja wymaga połączenia z internetem. W przypadku, gdy nie ma połączenia z internetem, na ekranie pojawi się dialog z informacją o tym oraz *Toast* z informacją o nieudanym pobraniu danych. Przycisk w lewym górnym roku pozwala na odświeżenie danych po uzyskaniu dostępu do internetu.

W dolnym prawy rogu ekranu znajduje się przycisk wyszukiwania. Kliknięcie go otwiera kolejną aktywność i wyszukuje książki.

1.6.2. Wybór kategorii

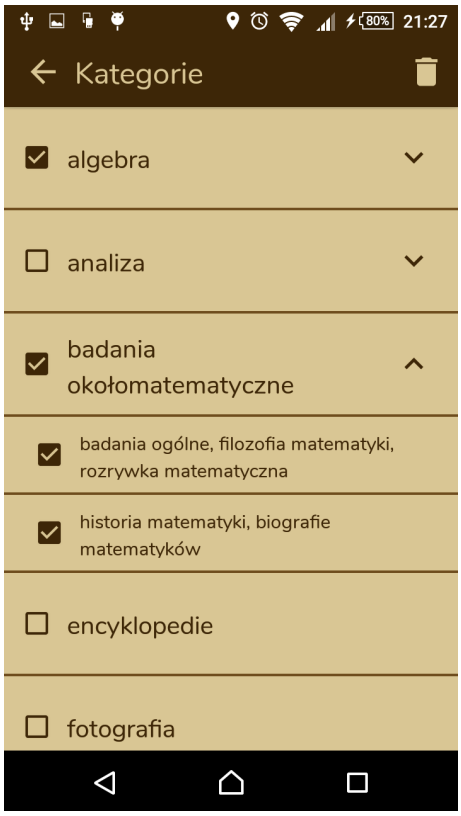
Widok z kategoriami zawiera listę posortowanych elementów oraz podzielonych na sekcję. Można rozróżnić kategorie główne oraz podkategorię, przy czym nie każda kategoria główna zawiera jakieś podkategorie. Jest to oznaczone strzałką znajdującą się z prawej strony elementu, która sugeruje możliwość kliknięcia i rozwinięcia listy.

Na tym etapie użytkownik ma możliwość ograniczenia zakresu poszukiwanych przez niego książek. Może zaznaczyć dowolną ilość kategorii głównych oraz podka-



Rysunek 12: Dialog z wyborem *typu pozycji* książki oraz widok wyszukiwania po wybraniu typu *podręcznik* przez użytkownika

tegorii z możliwość szybkiego zresetowania zaznaczeń, po naciśnięciu ikony kosza w prawym górnym rogu.

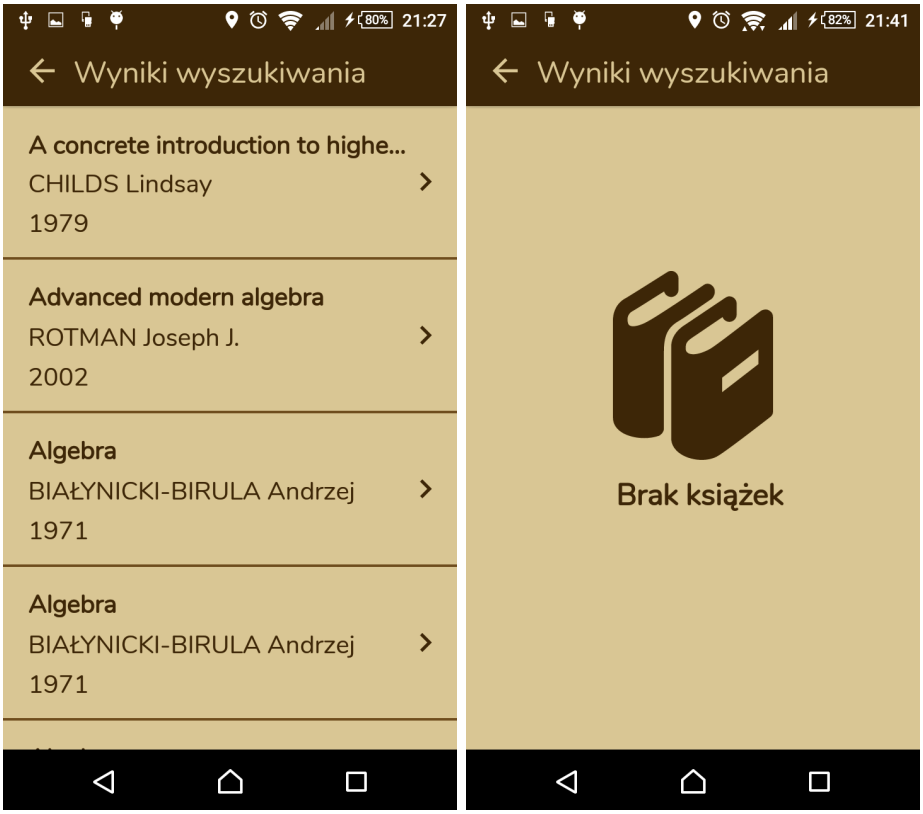


Rysunek 13: Ekran wyboru kategorii

W lewym górnym rogu znajduje się strzałka powrotu do poprzedniego widoku. Kliknięcie jej przenosi użytkownika z powrotem na ekran wyszukiwania. Po przeniesieniu na guziku wyboru kategorii pojawia się ilość zaznaczonych przez użytkownika elementów (rys. 12) lub tekst *Kategorie*, w przypadku gdy nic nie zostało zaznaczone.

1.6.3. Wyniki wyszukiwania

Po naciśnięciu przez użytkownika przycisku *Szukaj* na ekranie wyszukiwania otworzy się nowy widok. W przypadku nie znalezienia żadnej pozycji pasującej do zapytania lub w przypadku błędu serwera, wyświetlony zostanie ekran znajdujący się na rysunku 14 po prawej stronie. Ekran po lewej z rysunku 14, przedstawia listę z poprawnie znalezionymi pozycjami. Na tym etapie użytkownik może sprawdzić część tytułu książki (wyświetlanie zależne od wielkości ekranu urządzenia), autora



Rysunek 14: Przykładowe wyniki wyszukiwania

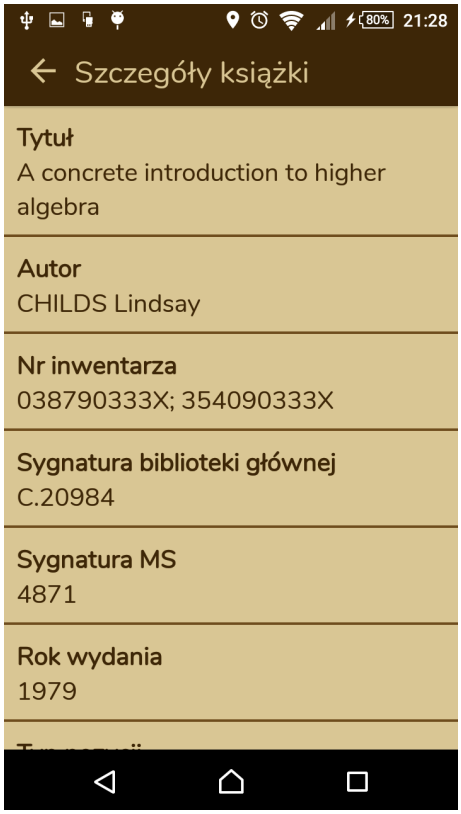
oraz rok wydania. Sortowanie odbywa się alfabetycznie pod względem tytułu. W przypadku takich samych tytułów sortowanie odbywa się po dacie wydania - od najnowszej do najstarszej.

Jeśli użytkownik chce wrócić do ekranu wyszukiwania książki może nacisnąć na strzałkę w lewym górnym rogu lub nacisnąć klawisz systemowy znajdujący się w lewym dolnym rogu na urządzeniach z systemem Android. Po powrocie pola będą wypełnione ostatnio wpisanymi lub wybranymi informacjami.

Więcej szczegółów książki zostanie wyświetlonych po naciśnięciu na konkretną pozycję z listy.

1.6.4. Szczegóły książki

Ekran szczegółów książki został przedstawiony na rysunku 15. Tym razem wszystkie znane informacje wyświetlane są w całości. Jeśli elementów jest dużo i nie mieszczą się na ekranie, użytkownik może scrollować widok w celu wyświetlenia pozostałych informacji.



Rysunek 15: Szczegóły książki

2. Aplikacja na system iOS

2.1. Wykorzystane narzędzia

Aplikacja na system mobilny firmy Apple została napisana w języku *Swift 4*, przy pomocy środowiska *Xcode 9.1*. *Swift* jako język programowania jest dostępny na rynku informatycznym niecałe 4 lata i jest następcą dosyć skomplikowanego języka *Objective–C*. Jego najnowsza, czwarta wersja, która została wykorzystana w projekcie, miała premierę we wrześniu 2017 roku, w tym samym czasie co dziewiąta wersja środowiska programistycznego *Xcode*. Zatem do tworzenia projektu zostały użyte jedne z najnowocześniejszych narzędzi programistycznych na rynku. Warto dodać, że wykorzystane rozwiązania pozwalają na komercjalizację projektu, ponieważ narzędzie *Xcode* jest programem typu *Freeware*, natomiast język *Swift*

bazuje na licencji *Apache License 2.0*, która zezwala na dystrybuowanie i sprzedaż oprogramowania.

2.2. Wymagania systemowe

Aplikacja może zostać zainstalowana na systemach iOS w wersji 10.0 lub nowszej. Wybór minimalnej wersji systemu nie był przypadkowy. System ten posiada zalety dla użytkownika jak i dla osoby tworzącej oprogramowanie.

System iOS 10.0 jest systemem istniejącym na rynku od września 2016 roku. Ma on zatem mniej niż półtora roku. Wspieranie tak nowych systemów wydaje się mało sensowne, jednak jest zupełnie inaczej. W przypadku polityki firmy Apple, systemy iOS wspierają zwykle wiele urządzeń istniejących na rynku dużo wcześniej od oficjalnej premiery systemu. Przykładowo wspierając system iOS 10.0, wspieramy zarazem telefon iPhone 5, który to miał premierę we wrześniu 2012. Zatem aplikacja może być instalowana na telefonach sprzed niemal 5.5 roku. Wpływa to zwykle na dużą liczbę instalacji nowych wersji iOS na urządzeniach. Zgodnie ze statystykami [5] na dzień 18 stycznia 2018 roku, ilość urządzeń posiadających system iOS wygląda następująco w zależności od wersji:

- iOS 11 – 65%
- iOS 10 – 28%
- wcześniejsze wersje iOS – 7%

Z powyższych statystyk wynika, iż pisząc aplikację na system iOS 10.0, można wspierać 93% rynku urządzeń mobilnych z systemem firmy Apple. Dodatkowo wspieranie stosunkowo nowej wersji systemu przynosi także inne korzyści a mianowicie dłuższy okres czasu istnienia na rynku w przypadku rzadkiego aktualizowania aplikacji.

Jeśli chodzi o spojrzenie na zasadność wspierania systemu iOS 10.0 od strony programisty, można z pewnością stwierdzić, że im nowszy system tym więcej narzędzi do zastosowania podczas tworzenia oprogramowania. Nowe wersje systemów zwykle przynoszą łatwiejsze i szybsze rozwiązania, które mogą być wykorzystane. Zatem najwygodniej dla programisty jest wytwarzać produkt stosując coraz to nowocześniejsze metody. Z tego względu wybór wspierania systemu iOS 10.0 jest pewnego

rodzaju kompromisem pomiędzy programistą a klientem. Programista zyskuje odpowiednią wygodę podczas tworzenia aplikacji, a klient zyskuje nowoczesny produkt, który przez długi czas może utrzymać się na rynku aplikacji mobilnych.

2.3. Model danych

Aplikacja posiada następujący model danych:

- `Book` – najważniejsza klasa aplikacji. Reprezentuje obiekt książki. Implementuje protokół `Codable`.
  - `bookTitle` (`String`) – tytuł
  - `bookAuthors` (`String`) – autorzy
  - `bookYear` (`String`) – rok wydania
  - `bookVolume` (`String`) – tom
  - `bookAvailability` (`String`) – dostępność
  - `bookPositionType` (`String`) – typ pozycji
  - `bookIsbn` (`String`) – numer ISBN
  - `bookMathLibrarySignature` (`String`) – sygnatura biblioteki MS
  - `bookMainLibrarySignature` (`String`) – sygnatura biblioteki głównej
  - `bookCategories` (`[Category]`) – kategorie, do których książka jest przypisana
- `MainCategory` – klasa reprezentująca kategorię główną. Implementuje protokół `Decodable`.
  - `category` (`Category`) – kategoria
  - `subcategoriesArray` (`[Category]`) – tablica podkategorii, przypisanych do kategorii głównej
- `Category` – klasa reprezentująca obiekt kategorii. Implementuje protokół `Codable`.
  - `id` (`String`) – id kategorii
  - `name` (`String`) – nazwa kategorii

- `DictionaryTypes` – model opisujący obiekt słownika dla książki. Implementuje protokół `Codable`.
  - `type` (`[String]`) – typy pozycji (np. podręcznik, zbiór zadań, inny)
  - `availability` (`[String]`) – dostępność (np. dostępna, wypożyczona, czytelnia)

Analizując powyższy model danych, który został wykorzystany w aplikacji, warto wspomnieć, że można podzielić kategorie na dwa rodzaje: kategorię główną oraz podkategorię. Każdy z tych rodzajów można jednak przedstawić za pomocą obiektu typu `Category`. Zatem pole `categories` może posiadać w tablicy kategorie jak i kategorie główne. Wystarczy z kategorii głównej wyciągnąć pole `category`.

2.4. Realizacja projektu

Projekt był realizowany przy pomocy natywnych narzędzi. Proces tworzenia widoków został oparty o narzędzie *Interface Builder* dostarczane przez firmę Apple. Umożliwia on budowanie widoków za pomocą prostego przenoszenia komponentów i ustawiania odległości pomiędzy nimi z poziomu interfejsu. Pozwala to na szybkie i proste tworzenie widoków aplikacji.

Główne widoki aplikacji zostały utworzone w pliku o nazwie *Main.storyboard*, który pozwala na wykorzystanie opisanego powyżej narzędzia. Wszystkie podrzędne widoki takie jak komórki tabeli, zostały utworzone w plikach o rozszerzeniu *.xib*. Osobny plik dla każdej z komórek pozwala na ich ułatwione modyfikacje w przyszłości oraz wielokrotne używanie w różnych tabelach.

Każdy z utworzonych widoków ma swoje odzwierciedlenie w pliku typu *.swift*. Dzięki temu programista ma możliwość wpływania na ich wygląd zewnętrzny za pomocą przekazywania im odpowiednich danych.

Proces tworzenia aplikacji na system iOS bazował na ciągłych porównaniach aplikacji z aplikacją na system Android w celu jak najlepszego odwzorowania obu z tych aplikacji. Wiele z wykorzystanych rozwiązań było dostępne wyłącznie dla wybranej platformy. Poniżej zostały przedstawione istotne elementy, jakie użyto podczas tworzenia oprogramowania dla biblioteki wydziałowej na mobilny system firmy Apple.

2.4.1. Biblioteka R.swift

R.swift jest biblioteką [7], która służy do zarządzania zasobami aplikacji. Została ona wciągnięta do projektu na początku jego tworzenia w celu wygody dostępu do elementów aplikacji. Działa bardzo podobnie do klasy `R` występującej w języku Java dla systemu Android. Dzięki tej bibliotece można w bardzo łatwy sposób dostać się do komponentów takich jak obrazek, zainicjowany widok czy też międzynarodowe ciągi znaków zapisane w aplikacji. Przykładem pokazującym zasadność użycia tej biblioteki może być zwykłe rejestrowanie komórki do tabeli.

```
tableView.register(
    UINib(nibName: "BookTableViewCell", bundle: nil),
    forCellReuseIdentifier: "BookTableViewCell"
)
```

Poniższy przykład pokazuje uproszczony sposób rejestrowania komórki przy pomocy biblioteki *R.swift*.

```
tableView.register(
    R.nib.bookTableViewCell(),
    forCellReuseIdentifier: "BookTableViewCell"
)
```

2.4.2. Zarządzanie widokami

Cała aplikacja na system iOS została stworzona z wykorzystaniem natywnego narzędzia jakim jest `UIPageViewController` [6]. Pozwala on na przechodzenie pomiędzy widokami typu `UIViewController` z wykorzystaniem jednej z dwóch natywnych animacji, które można wybrać. W przypadku naszej aplikacji wykorzystana została animacja `pageCurl`, która imituje przewracanie strony w książce. Pozwoliło to na lepszy odbiór aplikacji przez użytkownika.

Obiekt klasy `UIPageViewController` działa na zasadzie tablicy uporządkowanej widoków, co oznacza, że każdy z widoków "wie" jaki widok jest przed nim i za nim. Nasza aplikacja nie ma ustalonej kolejności widoków, ponieważ użytkownik za każdym razem ma pewien wybór i może przejść do jednego widoku, by potem wrócić i przejść do kolejnego. Z tego powodu, w celu skorzystania z tego narzędzia, została utworzona klasa `MainPageViewController`, która dziedziczy po klasie

`UIPageViewController` w celu dostosowania jej do naszych potrzeb. Została ona zaimplementowana w taki sposób, by przed każdym przejściem pomiędzy widokami następowała analiza, jaki widok ma być wyświetlony. Następnie jest on tworzony, uzupełniany odpowiednimi danymi i pokazywany z pewnym wybranym przejściem (w prawo lub w lewo). Dodatkowo obiekt typu `MainPageViewController` ma możliwość zaprezentowania widoku oraz zainicjowania systemowego paska nawigacji. W celu umożliwienia wykonania tych operacji z dowolnego wyświetlonego widoku, został utworzony odpowiedni protokół, który jest implementowany przez tę klasę, a następnie wykorzystywany w widokach, które go wymagają do obsługi interfejsu użytkownika.

```
protocol MainPageViewControllerDelegate: class {
    func presentViewController(_ viewController: UIViewController)
    func next(viewController: UIViewController)
    func previous(viewController: UIViewController)
    func initNavigationBar(withTitle title: String?,
                           leftButton: UIBarButtonItem?,
                           rightButton: UIBarButtonItem?)
}
```

Każdy z widoków podrzędnych posiada pole typu `MainPageViewControllerDelegate?`, które pozwala na przekazanie metod służących do zarządzania interfejsem bezpośrednio z poziomu tego widoku.

2.4.3. MainViewController

Klasa ta dziedziczy po klasie `UIViewController` i została stworzona jedynie w celu możliwości rozszerzenia funkcjonalności aplikacji w przyszłości. Jest ona bowiem dziedziczona przez każdy widok w aplikacji, co pozwala na zunifikowanie interfejsu poprzez dostęp do tych samych metod, które są w stanie wpłynąć na utworzoną instancję widoku.

2.4.4. SearchViewController

Obiekt klasy tego typu jest obiektem pozwalającym przygotować formularz, który następnie będzie użyty w celu wyszukania książki w systemie biblioteki. Składa się on z natywnego elementu tabeli `UITableView`, dwóch rodzajów komórek:

Rysunek 16: Projekt ekranu wyszukiwania książek

`SearchTextTableViewCell` i `SearchCategoryTableViewCell` oraz z przycisku typu `LoadingButton`, który został zaimplementowany na potrzeby projektu. Klasa `SearchViewController` implementuje delegaty obu tych komórek:

- `SearchTextTableViewCellDelegate` – w celu odczytania tekstu z komórki i zapisania odczytanego ciągu znaków w celu użycia go do wyszukiwania
- `SearchCategoryTableViewCellDelegate` – w celu przekazania możliwości otwarcia widoku z kategoriami, po naciśnięciu przycisku

Widok typu `SearchCategoryTableViewCell` posiada możliwość uzupełniania danych przez użytkownika. Powoduje to wysunięcie się klawiatury, która niekiedy zakrywa pole, w które użytkownik wpisuje tekst. Z tego powodu został on wyposażony w rozszerzenie, które pozwala na podwijanie się widoku podczas wysuwania na nim klawiatury.

```
extension SearchViewController {
```

```
    @objc func keyboardWillShow(notification:NSNotification){
        var userInfo = notification.userInfo!
        var keyboardFrame:CGRect =
        (userInfo[UIKeyboardFrameEndUserInfoKey] as! NSValue)
        .cgRectValue
        keyboardFrame = self.view.convert(keyboardFrame, from: nil)
        tableView.contentInset =
        UIEdgeInsetsMake(0.0, 0.0,
            keyboardFrame.size.height + 8.0, 0.0)
    }
    @objc func keyboardWillHide(notification:NSNotification){
        var userInfo = notification.userInfo!
        var keyboardFrame:CGRect =
        (userInfo[UIKeyboardFrameEndUserInfoKey] as! NSValue)
        .cgRectValue
        keyboardFrame = self.view.convert(keyboardFrame, from: nil)
        tableView.contentInset =
        UIEdgeInsetsMake(0.0, 0.0,
            DefaultValues.EDGE_INSET_BOTTOM, 0.0)
    }
}
```

Powyższe metody w celu monitorowania zachowania klawiatury są dodane do centrum notyfikacji w aplikacji jako obserwatory pokazania/ukrycia się natywnej klawiatury.

```
private func initObservers() {
        NotificationCenter.default.addObserver(self,
            selector: #selector(keyboardWillShow),
            name: NSNotification.Name.UIKeyboardWillShow,
            object: nil)
        NotificationCenter.default.addObserver(self,
            selector: #selector(keyboardWillHide),
            name: NSNotification.Name.UIKeyboardWillHide,
            object: nil)
```

```
}
```

Widok posiada u dołu przycisk szukania, który odpytuje usługę za pomocą spreparowanego obiektu klasy `Book` utworzonego na podstawie uzupełnionych pól przez użytkownika.

W pasku nawigacji widoku, zostały podpięte dwa przyciski. Jeden z nich znajduje się po lewej stronie i służy do pobrania typów pozycji, dostępności oraz kategorii książek w przypadku braku internetu przy włączaniu aplikacji. Kliknięcie tego przycisku powoduje także wyświetlenie dialogu informującego użytkownika o pobieraniu danych. Do utworzenia dialogu została wykorzystana biblioteka JGProgressHUD [8]. Drugi z przycisków znajduje się po prawej stronie i służy do wyczyszczenia formularza ze wszystkich wpisanych danych.

2.4.5. CategoriesViewController

Rysunek 17: Projekt ekranu wyboru kategorii

Kolejnym widokiem jest obiekt klasy `CategoriesViewController`. Jest to widok posiadający komponent `UITableView`, jednak w bardziej skomplikowanej kon-

figuracji niż widok `SearchViewController`. W tabeli został zarejestrowany model komórki kategorii `CategoryTableViewCell` oraz model nagłówka głównej kategorii `MainCategoryHeaderView`. Tabela została stworzona jako tabela rozwijalna, tzn. po wyborze sekcji, wyświetlane są komórki do niej należące. Z powodu braku natywnego rozwiązania na tę funkcjonalność, została ona stworzona samodzielnie.

Widok posiada zmienną `expandedHeaders`, która z początku jest wypełniona wartościami typu `false`.

```
var expandedHeaders: [Bool] = []
...
func collapseAllHeaders() {
    let mainCategoriesCount =
        SessionManager.shared.mainCategories.count
    expandedHeaders =
        Array(repeating: false, count: mainCategoriesCount)
}
```

Indeks każdego elementu tabeli odzwierciedla sekcję nagłówka. Pozwala to stwierdzić, które z sekcji są rozwinięte, a które zwinięte. Kliknięcie w przycisk rozwijania sekcji, uruchamia metodę ze specjalnie stworzonego na te potrzeby protokołu obsługującego nagłówek.

```
extension CategoriesViewController: MainCategoryHeaderViewDelegate {
    ...
    func expandSubcategories(
        usingHeader header: MainCategoryHeaderView)
    {
        var headerIsExpanded = expandedHeaders[header.section]
        headerIsExpanded = !headerIsExpanded
        header.isExpanded = headerIsExpanded
        expandedHeaders[header.section] = headerIsExpanded
        tableView.reloadData()
    }
}
```

Powoduje ona przestawienie odpowiedniej wartości w tablicy `expandedHeaders` na

przeciwną, przekazanie jej do widoku i odświeżenie tabeli. Natywne metody delegatowe klasy `UITableView` zostały uzupełnione w sposób obsługujący informacje dotyczące rozwiniętej/zwiniętej sekcji. Implementacja metody odpowiadającej za ilość wierszy, pochodząca z protokołu `UITableViewDataSource` została zaprezentowana poniżej.

```
func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int
{
    if expandedHeaders[section] {
        return SessionManager.shared.mainCategories[section]
            .subcategories.count
    } else {
        return 0
    }
}
```

W przypadku rozwiniętej sekcji zwraca ilość wierszy równą ilości podkategorii należących do kategorii głównej z danej sekcji. Dla zwiniętego nagłówka nie wyświetla żadnego wiersza.

Opisywany widok obsługuje także możliwość wyboru wierszy bądź nagłówków, w celu późniejszego wykorzystania wybranych kategorii. W tym celu również zostały wykorzystane zmienne służące do pamiętania stanu wyboru.

```
var selectedHeaders: [Bool] = []
var selectedCells: [IndexPath:Bool] = [:]
```

Przedstawiają one zaznaczone nagłówki (obiekt typu `Array`) oraz zaznaczone komórki (obiekt typu `Dictionary`). Każde wybranie nagłówka, zmienia jego stan, jednocześnie wpływając na stan komórek należących do danej sekcji. Wybór nagłówka powoduje skorzystanie z metody przypisującej w słowniku `selectedCells` zmienne typu `Bool` do obiektów typu `IndexPath`, które określają pozycję komórki w tabeli. Jednocześnie przypisując obiekty, komórki w zadanych pozycjach są zaznaczane lub odznaczane.

Nie tylko nagłówek może wpłynąć na stan komórki. Istnieje również odwrotna zależność. Wybór komórek także może wpłynąć na stan nagłówka (na przykład

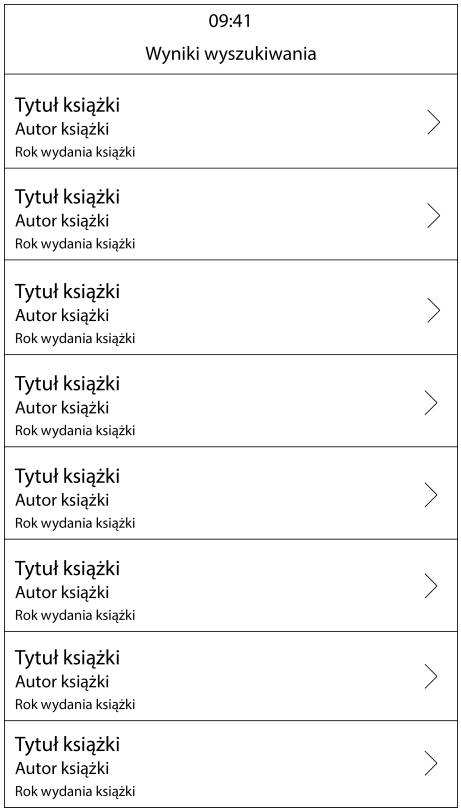
w przypadku zaznaczenia wszystkich podkategorii). Obsługa tego zdarzenia została opisana poniższą metodą.

```
private func trySelectHeader(inSection section: Int) {
    let subcategories = SessionManager.shared
        .mainCategories[section].subcategories
    if subcategories.isEmpty { return }
    for (index, _) in subcategories.enumerated() {
        let indexPath = IndexPath(row: index, section: section)
        guard let selected = selectedCells[indexPath]
        else { return }
        if !selected {
            toggleHeader(inSection: indexPath.section,
                select: false)
            return
        }
    }
    toggleHeader(inSection: section, select: true)
}
```

Dodatkowo obiekt typu `CategoriesViewController` obsługuje zapis wybranych kategorii jak i załadowanie zaznaczonych już wcześniej kategorii, zaraz po wejściu na widok.

2.4.6. BookListViewController

Obiekt klasy `BookListViewController` jest następnym z widoków posiadających tabelę `UITableView`. Służy do wyświetlania wyników wyszukiwania. Została w nim zarejestrowana komórka typu `BookTableViewCell`, która wyświetla tytuł oraz autora znalezionej książki. Tabela została przystosowana w taki sposób, aby wyświetlać pewną ilość książek, a w przypadku zaistniałej potrzeby, dociągać kolejną ich ilość. Metoda `tryFetchMoreBooks(loadedIndexPath: IndexPath)`, która została zaimplementowana, jest wywoływana przy każdym ładowaniu się komórki na widoku. W przypadku, gdy ładowana jest szósta od końca komórka, metoda ta odpytuje usługę o kolejne książki. Gdy je otrzyma, dopisuje je do tablicy książek co powoduje odświeżenie widoku, ze zwiększoną liczbą znalezionych pozycji.



Rysunek 18: Projekt ekranu listy książek

```
fileprivate func tryFetchMoreBooks(loadedIndexPath: IndexPath) {
    let rowWhenFetchNeeded = books.count - 5
    if loadedIndexPath.row == rowWhenFetchNeeded && canFetchMore {
        offset += DefaultValues.BOOKS_PER_FETCH
        RequestManager.shared.getBooks(
            withOffset: offset,
            completion: appendFetchedBooks)
    }
}
fileprivate func appendFetchedBooks(_ fetchedBooks: [Book]) {
    if fetchedBooks.isEmpty {
        canFetchMore = false
        return
    }
    self.books.append(contentsOf: fetchedBooks)
```

```
}
```

Widok przedstawiający listę książek, obsługuje dodatkowo natywne rozwiązanie typu *3D Touch*. Pozwala ono na podgląd zawartości komórki oraz wykonanie dodatkowych akcji przed jej wyborem. W tym celu, zostało stworzone rozszerzenie klasy `BookListViewController` obsługujące akcje mocnego przyciśnięcia wiersza (*Peek*) oraz jego jeszcze mocniejszego dociśnięcia (*Pop*).

```
extension BookListViewController: UIViewControllerPreviewingDelegate {
    //PEEK
    func previewingContext(
        _ previewingContext: UIViewControllerPreviewing,
        viewControllerForLocation location: CGPoint) -> UIViewController?
    {
        guard let indexPath = tableView.indexPathForRow(at: location),
            let cell = tableView.cellForRow(at: indexPath)
        else {
            return nil
        }
        let book = books[indexPath.row]
        let detailsVC = getBookDetailsViewController(forBook: book)
        previewingContext.sourceRect = cell.frame
        return detailsVC
    }
    //POP
    func previewingContext(
    _ previewingContext: UIViewControllerPreviewing,
    commit viewControllerToCommit: UIViewController) {
        let navigationController = UINavigationController(
            rootViewController: viewControllerToCommit)
        self.showDetailViewController(navigationController,
            sender: self)
    }
}
```

Akcja *3D Touch* jest obsługiwana jedynie na telefonach typu iPhone 6s i nowszych. Z tego powodu należy po załadowaniu widoku upewnić się, czy jest sens rejestrowania metod szybkiego podglądu.

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    if traitCollection.forceTouchCapability == .available {
        registerForPreviewing(with: self, sourceView: tableView)
    }
}
```

Widok typu `BookListViewController` posiada dodatkowo obsługę w przypadku, gdy ilość otrzymanych książek z usługi jest równa 0. Do obsługi pustego ekranu wykorzystana została biblioteka *UIEmptyState* [9]. Jest to mało rozpowszechniona biblioteka służąca do analizowania wyświetlanych informacji w tabeli `UITableView` lub kolekcji `UICollectionView`, która w przypadku liczby elementów do wyświetlenia równej 0, pokaże użytkownikowi dowolnie spersonalizowaną informację o braku danych do zaprezentowania.

Biblioteka ta jest napisana w stosunkowo przyjazny sposób dla programisty. W celu zastosowania jej funkcjonalności, należy zaimplementować dwa typy protokołów: `UIEmptyStateDelegate` oraz `UIEmptyStateDataSource`. Następnie można przejść do przypisywania potrzebnych zmiennych, które zostaną automatycznie wyświetlone w przypadku braku elementów do wyświetlenia na widoku.

```
extension BookListViewController:
    UIEmptyStateDelegate, UIEmptyStateDataSource
{
    fileprivate func setEmptyStateDelegates() {
        self.emptyStateDelegate = self
        self.emptyStateDataSource = self
    }
    var emptyStateBackgroundColor: UIColor {
        return .main
    }
    var emptyStateTitle: NSAttributedString {
```

```
        let title = R.string.localizable.noResults()
        let range = (title as NSString).range(of: title)
        let titleAttributedString =
            NSMutableAttributedString(string: title)
        let titleColor = UIColor.tintDark
        titleAttributedString.addAttribute(
            NSAttributedStringKey.foregroundColor,
            value: titleColor, range: range)
        return titleAttributedString
    }
    var emptyStateImage: UIImage? {
        let image = R.image.bookShelf()
            .maskWithColor(color: .tintDark)
        return image
    }
}
```

2.4.7. BookDetailsViewController

Klasa `BookDetailsViewController` przedstawia obiekt widoku informacji o książce. Składa się ona tak jak inne widoki z tabeli `UITableView`, a do wyświetlenia zawartości wykorzystuje komórkę typu `BookDetailTableViewCell` posiadająca jedynie tytuł oraz zawartość pola opisującego książkę. Opisywana klasa widoku, jest dużo prostsza od poprzednich. Elementem wyróżniającym ją spośród pozostałych, jest zmienna typu `[UIPreviewActionItem]`, która odpowiada za szybkie akcje w trybie *Peek*, będąc na poprzednim widoku typu `BookListViewController`.

```
override var previewActionItems: [UIPreviewActionItem] {
    let copyTitleAction = UIPreviewAction(
        title: R.string.localizable.copyTitle(),
        style: .default) {
            (action, vc) in
            UIPasteboard.general.string = self.book?.bookTitle
    }
    let copyAuthorAction = UIPreviewAction(
```

Rysunek 19: Projekt ekranu szczegółów książki

```
        title: R.string.localizable.copyAuthor(),
        style: .default) {
            (action, viewcontroller) in
            UIPasteboard.general.string = self.book?.bookAuthors
    }
    return [copyTitleAction, copyAuthorAction]
}
```

2.4.8. SessionManager

`SessionManager` to klasa odpowiadająca za zarządzanie, przechowywanie i udostępnianie danych do widoków podczas jednej sesji działania aplikacji. Klasa ta posiada trzy pola. Wystarczają one do zarządzania całą sesją, która jest aktywna, dopóki aplikacja znajduje się w pamięci RAM urządzenia.

- `var searchedBook: Book!` – pole odpowiadające za przetrzymywanie szablonowej, poszukiwanej przez użytkownika książki. Pole to jest aktualizowane

z każdą zmianą użytkownika na widoku wyszukiwania.

- `var mainCategories: [MainCategory]` – wszystkie kategorie (główne wraz z podkategoriami) zaciągnięte z usługi po wejściu w aplikację.
- `var dictionaryTypes: DictionaryTypes!` – wszystkie możliwe typy pozycji oraz stany dostępności książek. Otrzymywane są z usługi po otwarciu aplikacji.

Klasa `SessionManager`, z powodu zarządzania całą sesją, została zaimplementowana z zastosowaniem wzorca projektowego Singleton. Zgodnie z jego założeniami, klasa posiada statyczną instancję oraz prywatny konstruktor.

```
class SessionManager {
    private init() {
        searchedBook = Book()
        dictionaryTypes = DictionaryTypes()
    }
    static let shared = SessionManager()
    ...
}
```

2.4.9. RequestManager

Za odpytywanie usługi odpowiada klasa `RequestManager`. Klasa ta jest również Singletonem. Jest ona napisana korzystając jedynie z natywnych rozwiązań języka *Swift 4*. Dzięki implementacji odpowiednich protokołów przez klasy modelowe, metoda ta jest w stanie bezpośrednio stworzyć obiekt z danych, które przyszły z usługi w formacie *.json*. Poniżej został przedstawiony przykład zaciągania z usługi obiektu tablicy wszystkich kategorii głównych.

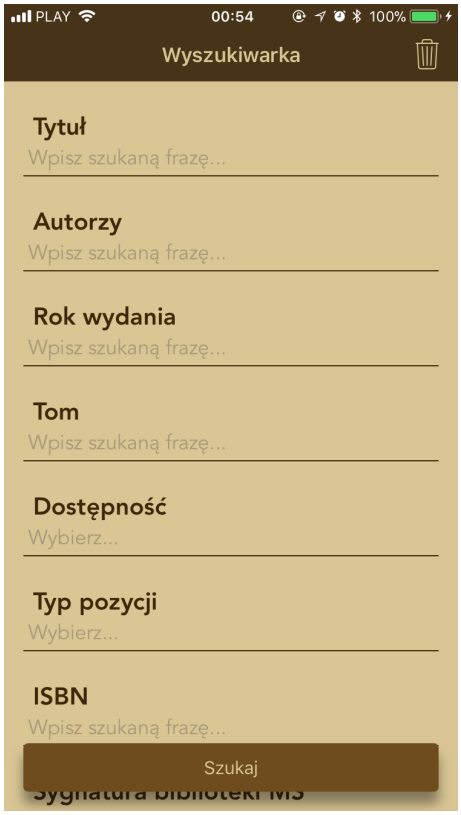
```
func getCategories(completion: @escaping (([MainCategory])->())) {
    guard let request = getRequest(usingHttpMethod: "GET",
        forEndpoint: CATEGORY_ENDPOINT) else { return }
    URLSession.shared.dataTask(with: request) {
        (data, response, error) in
        if let error = error {
            NSLog(error.localizedDescription)
```

```
        }
        guard let data = data else { return }
        do {
            let mainCategories = try
                JSONDecoder().decode([MainCategory].self, from: data)
            completion(mainCategories)
        } catch let jsonError {
            NSLog(jsonError.localizedDescription)
            completion([])
        }
    }.resume()
}
```

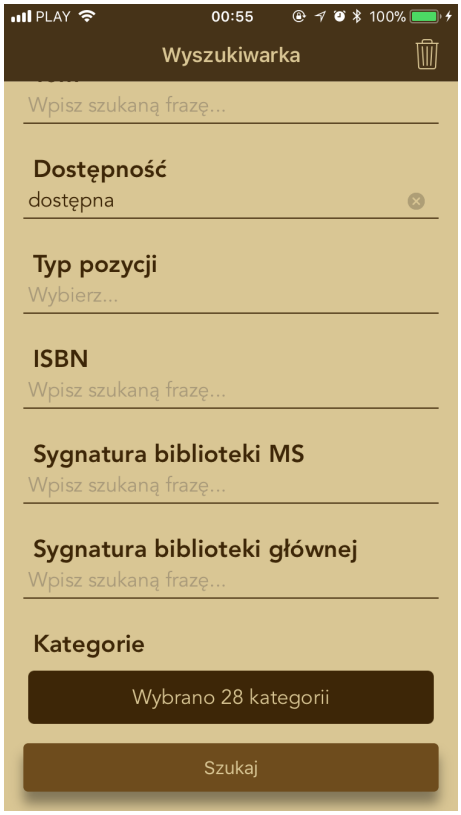
Każda z metod odpytujących usługę zaimplementowana jest korzystając z metody tworzącej typowy obiekt klasy `URLRequest?`. Jest on tworzony na podstawie typu metody (w przypadku naszego projektu korzystamy jedynie z typów *GET* oraz *POST*) oraz końcówki adresu do którego powinno zostać wysłane zapytanie.

```
fileprivate func getRequest(
    usingHttpMethod httpMethod: String?,
    forEndpoint endpoint: String) -> URLRequest?
{
    let address = URL_STRING + endpoint
    guard let url = URL(string: address) else { return nil }
    var request = URLRequest(url: url)
    request.httpMethod = httpMethod
    request.setValue("application/json",
        forHTTPHeaderField: "Content-Type")
    return request
}
```

2.5. Interfejs użytkownika



Rysunek 20: Niewypełniony ekran wyszukiwania książki



Rysunek 21: Częściowo uzupełniony ekran wyszukiwania książki

2.5.1. Wyszukiwarka książek

Aplikacja po uruchomieniu, przechodzi bezpośrednio do ekranu wyszukiwarki książek, jak zostało przedstawione na rysunku 20. Aplikacja do poprawnego działania wymaga połączenia użytkownika z internetem. W przypadku braku połączenia internetowego, na ekranie pojawi się stosowna informacja zaraz po wejściu na widok. Przycisk w lewym górnym roku pozwala na odświeżenie zawartości. Użytkownik ma możliwość uzupełnienia dowolnych pól, w zależności od poszukiwanej przez niego pozycji.

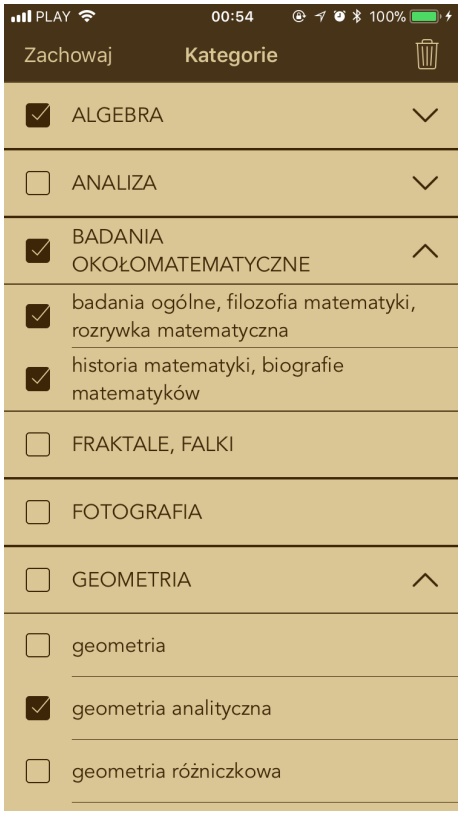
Pierwsze z pól są polami służącymi do uzupełnienia tekstu za pomocą klawiatury systemowej bądź wyboru jednego z kilku elementów z prostej listy. Ostatnia z komórek posiada przycisk przenoszący użytkownika do ekranu wyboru kategorii, jak na rysunku 22. W prawym górnym rogu ekranu znajduje się przycisk, służący do wyczyszczenia wszystkich pól wyszukiwarki. Usunięcie dowolnego wypełnionego

pola również jest możliwe – wystarczy wybrać szary przycisk znajdujący się obok wypełnionego pola, jak na rysunku 21.

U dołu ekranu znajduje się przycisk wyszukiwania. Kliknięcie go, uruchamia wyszukiwanie, co jest oznajmione poprzez kręcące się kółeczko wewnątrz przycisku.

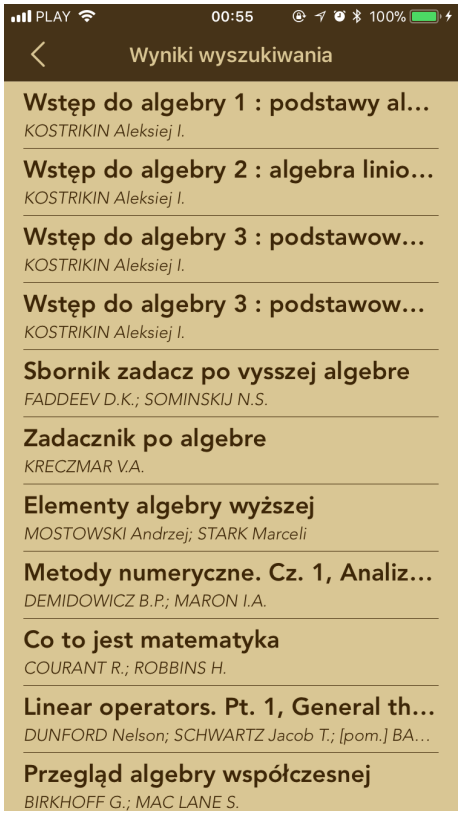
2.5.2. Wybór kategorii

Ekran kategorii, jest listą elementów posortowanych w sekcjach. Niektóre z sekcji są rozwijalne. Rozwinięcie sekcji pozwala użytkownikowi przejrzeć wszystkie podkategorie dla wybranej kategorii głównej. Użytkownik może na tym ekranie dokonać wyboru zakresu poszukiwanej przez niego książki. W przypadku wyboru kategorii głównej, zaznaczane są automatycznie wszystkie jej podkategorie. W prawym górnym rogu znajduje się przycisk z ikoną kosza. Pozwala on wyczyścić wybrane kategorie i zacząć wybór od nowa.



Rysunek 22: Ekran wyboru kategorii

W lewym górnym rogu umiejscowiony jest przycisk zapisu wybranych kategorii. Jego wybór przenosi użytkownika z powrotem na ekran wyszukiwania. Po powrocie,



Rysunek 23: Ekran wyników wyszukiwania

Rysunek 24: Pusty ekran wyników

przycisk do wyboru kategorii wyświetla ich konkretną ilość jaka została wybrana przez użytkownika. Zostało to przedstawione na rysunku 21.

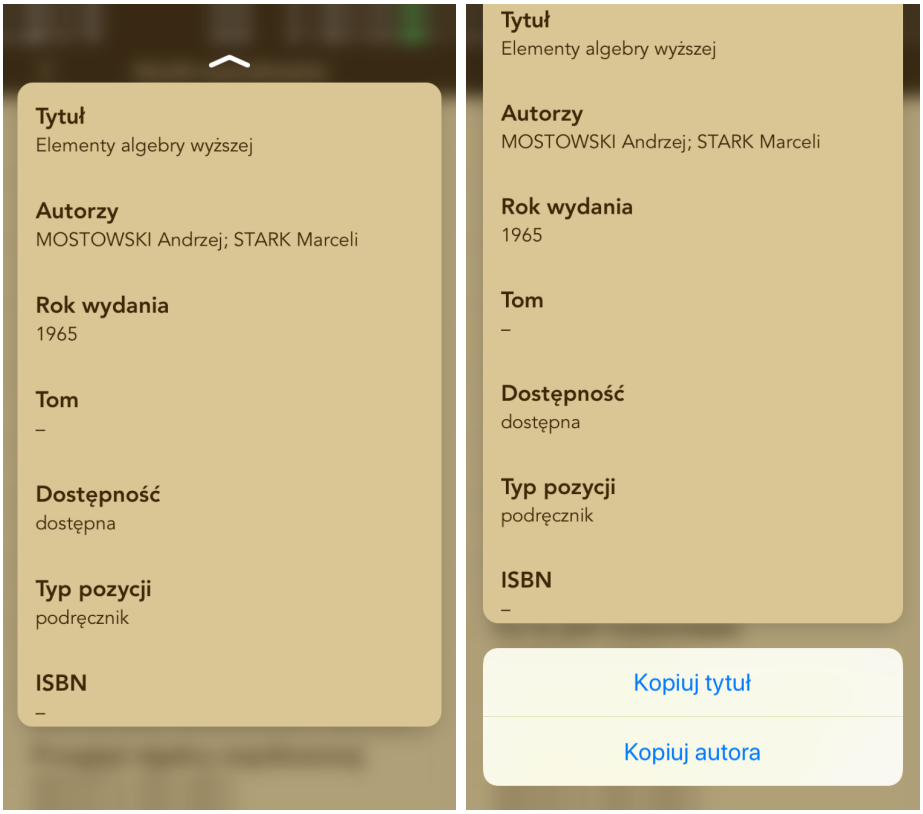
2.5.3. Wyniki wyszukiwania

Na ekranie wyszukiwania, wybór przycisku odpowiadającego za szukanie, przekierowuje użytkownika do ekranu Wyników wyszukiwania. W przypadku nie znalezienia żadnej pozycji pasującej do zapytania, wyświetlany jest ekran znajdujący się na rysunku 24. Ekran z rysunku numer 23, przedstawia listę z poprawnie znalezionymi pozycjami. Składa się on z wierszy przedstawiających tytuły oraz autorów wyszukanych pozycji. Ilość wyszukanych książek jest równa 20 lub mniejsza. W przypadku, gdy jest ona maksymalna, użytkownik może przejść do dołu listy w celu doładowania i wyświetlenia kolejnych rekordów na widoku.

Użytkownik w celu przejścia dalej ma do wyboru dwie akcje:

1. Mocniejsze dociśnięcie ekranu w miejscu interesującej go pozycji.

2. Wybór interesującego go wiersza.



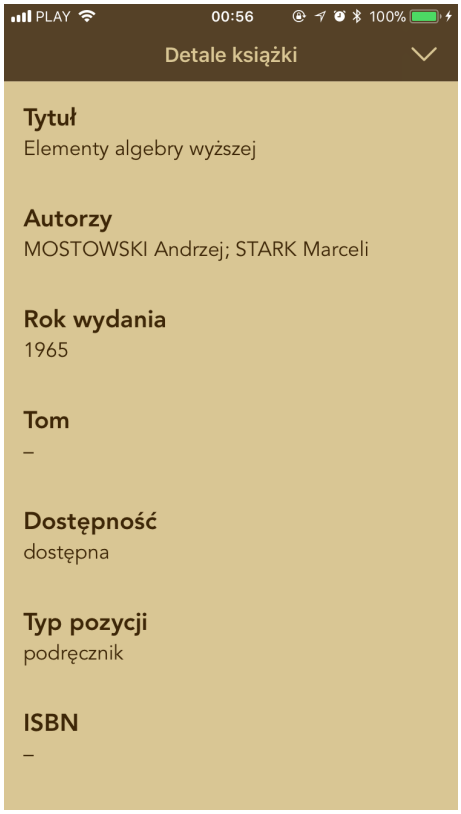
Rysunek 25: Ekrany obsługujące akcje *3D Touch*

Pierwsza opcja jest dostępna jedynie dla telefonów z opcją *3D Touch* wbudowaną w ekran. W przypadku opcji 1., użytkownik otrzymuje na ekranie podgląd szczegółów książki, który przedstawiono na rysunku 25. Przesunięcie go ku górze, spowoduje wyświetlenie się u dołu dodatkowych opcji. Będą to opcje umożliwiające skopiowanie tytułu bądź autora do schowka. Przeciągnięcie widoku podglądu w dół, spowoduje jego ukrycie. W przypadku jego jeszcze mocniejszego dociśnięcia, użytkownik zostanie przekierowany do ekranu. Analogiczna sytuacja ma miejsce podczas wyboru opcji numer 2., czyli zwykłego wyboru wiersza z książką.

W celu powrotu i rozpoczęcia szukania książki od początku, użytkownik może wybrać przycisk w lewym górnym rogu. Kliknięcie go, spowoduje powrót do ekranu wyszukiwania.

2.5.4. Szczegóły książki

Jest to ekran przedstawiony na rysunku 26. Jest on prezentowany użytkownikowi od spodu, po wyborze jednej z komórek opisujących książkę. Przedstawia prostą przesuwaną listę elementów dotyczących wybranej książki. Przycisk w prawej górnej części ekranu pozwala schować widok i wrócić do poprzedniego ekranu wszystkich wyszukanych pozycji.



Rysunek 26: Szczegóły książki

3. Zakończenie

Tworząc aplikację serwerową w tak małej grupie dla całego wydziału, mogliśmy znacznie poszerzyć swoją wiedzę o programowaniu, ale zarazem rozwinąć się w zakresie pracy w zespole. Dodatkowo w celu lepszego porozumienia się z wieloma osobami, z którymi pracowaliśmy i dyskutowaliśmy podczas wytwarzania tej aplikacji, musieliśmy poszerzyć swoje zdolności interpersonalne. Bez ich pomocy byłoby nam ciężko dopracować wiele spraw związanych z dostosowaniem obsługi, a także wyglądu aplikacji dla klienta.

Cały projekt aplikacji mobilnych jak i aplikacji webowej został omówiony i wielokrotnie prezentowany oraz testowany z Panią mgr inż. Martyną Maciaszczyk, która była osobą zatwierdzającą wszystkie postanowienia oraz zmiany w widoku wszystkich aplikacji. Proces ten przebiegał stosunkowo długo, jednak był bardzo pouczający.

To właśnie interfejs graficzny był jednym z problemów wytworzenia takiego systemu. Bardzo trudne jest dostosowanie do siebie wszystkich aplikacji pod względem wyglądu. Strona internetowa zawsze będzie nieco inna od aplikacji mobilnej, ponieważ istnieje wiele różnic dotyczących założeń działania, obsługi, czy też przyzwyczajeń i wymagań użytkownika od danego systemu. Z tego powodu aplikacja internetowa dla administratora różni się od aplikacji mobilnych przeznaczonych przede wszystkim dla studentów. Największym problemem było dostosowanie do siebie obu aplikacji mobilnych na systemy Android i iOS. Każdą z nich pisała inna osoba, więc założenia dotyczące projektu musiały być dokładnie ustalone, a każda późniejsza edycja ponosiła za sobą modyfikacje na obu platformach, co przekładało się na dwukrotnie zwiększony czas potrzebny do realizacji danej zmiany. Dodatkowo każda z platform ma inne natywne narzędzia używane do tworzenia aplikacji. Systemy różnią się od siebie również wyglądem, co przyzwyczaja użytkowników do swoich rozwiązań. Z tego powodu aplikacje na dwa różne systemy nigdy nie mogą być identyczne jeśli chcą być intuicyjne dla ich grupy docelowej odbiorców.

Aplikacja serwerowa dla biblioteki została napisana w sposób łatwy do rozszerzenia. Jej wygodne i proste w obsłudze możliwości dodawania nowych książek do bazy, mogą być wykorzystywane w przyszłości w celu powiększania ilości zbiorów dostęp-

nych w bibliotece. Dodatkowo, warto byłoby dodać możliwość rezerwacji książki online, bezpośrednio z poziomu aplikacji mobilnej. Ciekawą opcją na dalszą przyszłość (z powodu dużego nakładu pracy), byłaby możliwość wypożyczania danej pozycji książki w postaci pliku o rozszerzeniu .pdf. Pozwoliłoby to nie tylko na zmniejszenie kolejek w bibliotece, ale także na możliwość dostępu do wszystkich książek z urządzenia przenośnego bez potrzeby noszenia dużej ich ilości przy sobie. Co więcej, ilość książek w tym wypadku byłaby nieograniczona. Każdy ze studentów miałby swój wirtualny egzemplarz. Niestety byłoby to możliwe do osiągnięcia jedynie przy dużym nakładzie pracy z powodu dużej ilości skanów do wykonania a także stworzenia nowej usługi pozwalającej na ściąganie i zapisywanie plików na urządzeniu przenośnym.

Celem naszego projektu było utworzenie systemu dla Biblioteki Wydziału Matematyki Stosowanej na Politechnice Śląskiej w Gliwicach. Cel ten udało nam się osiągnąć zgodnie ze wszystkimi założeniami jakie zostały przed nami postawione. Mamy nadzieję, że w przyszłości uda się wdrożyć nasz system do użytku, i że posłuży on gronu odbiorców w taki sposób, w jaki byśmy chcieli, aby służył nam, gdy faktycznie go potrzebowaliśmy.

Literatura

[1] http://www.bn.org.pl/aktualnosci/1338-czytelnictwo-polakow-2016-%E2%80%93-raport-biblioteki-narodowej.html [dostęp: 10 lutego 2018]

[2] Robert C. Martin. *Czysty kod. Podręcznik dobrego programisty*. Wydawnictwo Helion, 2010, ISBN 978-83-283-1401-6.

[3] The Swift Programming Language (Swift 4.0.3)

[4] Using Swift with Cocoa and Objective-C (Swift 4.0.3)

[5] https://developer.apple.com/support/app-store/ [dostęp: 10 lutego 2018]

[6] https://developer.apple.com/documentation/ [dostęp: 10 lutego 2018]

[7] https://github.com/mac-cain13/R.swift [dostęp: 10 lutego 2018]

[8] https://github.com/JonasGessner/JGProgressHUD

[9] https://github.com/luispadron/UIEmptyState [dostęp: 10 lutego 2018]

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley. *The Java® Language Specification. Java SE 8 Edition*. Oracle America, 2015.

[11] https://developer.android.com/studio/releases/index.html [dostęp: 10 lutego 2018]

[12] https://developer.android.com/about/dashboards/index.html [dostęp: 10 lutego 2018]

[13] https://fabric.io/kits/android/crashlytics [dostęp: 10 lutego 2018]

[14] http://jakewharton.github.io/butterknife/ [dostęp: 10 lutego 2018]

[15] S. Madej, *Przybornik Pragmatycznego Programisty Androida*. Wydanie Pierwsze, 2015.

[16] P. Mainkar, *Expert Android Programming*. Packt Publishing, 2017, ISBN 9781786468956.

[17] https://github.com/codepath/android_guides/wiki/Dependency-Injection-with-Dagger-2 [dostęp: 10 lutego 2018]

[18] http://square.github.io/retrofit/ [dostęp: 10 lutego 2018]

[19] https://github.com/ReactiveX/RxJava [dostęp: 10 lutego 2018]

[20] https://developer.android.com/reference/android/support/constraint/ConstraintLayout.html [dostęp: 10 lutego 2018]

[21] https://developer.android.com/reference/android/support/design/widget/TextInputEditText.html [dostęp: 10 lutego 2018]