

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI  
I TECHNIK INFORMACYJNYCH



Instytut Automatyki i Informatyki Stosowanej

# Praca dyplomowa inżynierska

na kierunku Informatyka  
w specjalności Systemy Informacyjno-Decyzyjne

Śledzenie pozycji na rynku akcji amerykańskich  
z wykorzystaniem aplikacji mobilnej oraz usługi Backend as a Service

Łukasz Świtaj  
Numer albumu 283777

promotor  
dr inż. Andrzej Ratkowski

WARSZAWA 2025



## **Śledzenie pozycji na rynku akcji amerykańskich z wykorzystaniem aplikacji mobilnej oraz usługi Backend as a Service**

**Streszczenie.** Celem pracy było stworzenie aplikacji mobilnej na urządzenia z systemem Android adresującej potrzebę monitorowania pozycji giełdowych z amerykańskiego rynku akcji. Dzięki zapamiętaniu przez aplikację parametrów ceny zakupu czy ilości zakupionych aktywów oraz bieżącemu połączeniu z API zapewniającym dane giełdowe, możliwe jest monitorowanie zysku oraz aktualnej ceny waloru.

Użycie usługi Backend as a Service umożliwiło realizację takich operacji jak tworzenie konta, logowanie czy przechowywanie danych użytkownika na serwerze. Dzięki zastosowaniu tego rozwiązania dane o pozycjach użytkownika dostępne są z każdego urządzenia posiadającego aplikację i mającego dostęp do internetu.

Aplikacja natywna Android została zrealizowana przy użyciu języka programowania Kotlin (logika) oraz języka znaczników XML (interfejs użytkownika). Użyta platforma Backend as a Service to Back4app bazująca na Parse - rozwiązaniu typu open-source. Z kolei wykorzystane API do pozyskiwania informacji z giełd to Finnhub.

**Słowa kluczowe:** Android, Kotlin, aplikacja mobilna, Backend as a Service, Back4app, Finnhub, giełda

## **Mobile app for stock positions monitoring using Backend as a Service platform**

**Abstract.** The main project goal was to develop a native Android solution that allows its users to have a constant overview on their positions on the US stocks market. It is possible to monitor profit rate of the position as well as the current price. It can be realised thanks to storing such parameters as buy price and quantity by the app, and connection with stocks exchange data provider API.

The integration with a Backend as a Service solution allows users to make operations such as creating an account, logging into it, and storing their data in the remote database. Thanks to it the data about user's positions is available for one on every Android mobile device that has the network access.

The native Android app was developed using Kotlin programming language (logic) and markup language - XML (user interface). The Backend as a Service platform used in the project - Back4app bases on Parse which is an open-source solution. Finally, the Finnhub API has been used to fetch the data from stocks exchanges.

**Keywords:** Android, Kotlin, mobile application, Backend as a Service, Back4app, Finnhub, stocks, stock exchange



Politechnika Warszawska

załącznik nr 3 do zarządzenia  
nr 28 /2016 Rektora PW

Warszawa, 28.01.2016  
miejscowość i data

Łukasz Świątaj  
imię i nazwisko studenta

283777  
numer albumu

informatyka  
kierunek studiów

### OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płyście kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

Łukasz Świątaj  
czytelny podpis studenta



# Spis treści

<b>1. Wstęp</b>	9
1.1. Portfel inwestycyjny i zarządzanie ryzykiem	9
1.2. Dywersyfikacja i inwestycje	9
1.3. Istniejące rozwiązania	10
1.4. Wstępne założenia pracy	11
<b>2. Założenia oraz wybór narzędzi</b>	12
2.1. Założenia funkcjonalne	12
2.2. Wybór technologii	12
2.2.1. Język programowania	12
2.2.2. Platforma Backend as a Service	13
2.3. Wybór API giełdowego	14
<b>3. Implementacja i wyzwania</b>	16
3.1. Zaimplementowane rozwiązanie	16
3.2. Architektura rozwiązania	17
3.3. Network - API giełdowe i Retrofit	17
3.3.1. Wyzwanie - zmiana API	19
3.4. Back4App - Backend as a Service	19
3.4.1. Rejestracja i logowanie - Parse Users	20
3.4.2. Baza danych - Parse Objects	22
3.4.3. Struktury baz danych	25
3.5. Dostęp do danych przyjazny użytkownikowi - Coroutines	26
3.6. Komunikacja pomiędzy warstwami w MVVM - LiveData	27
<b>4. Weryfikacja rozwiązania</b>	30
4.1. Code review i branching model	30
4.2. Testowanie	30
<b>5. Podsumowanie</b>	32
5.1. Wnioski	32
5.2. Perspektywy rozwoju	32
<b>Bibliografia</b>	35
<b>Wykaz symboli i skrótów</b>	36
<b>Spis rysunków</b>	37





# 1. Wstęp

## 1.1. Portfel inwestycyjny i zarządzanie ryzykiem

W celu prawidłowego zdefiniowania pojęcia portfela inwestycyjnego należy rozpocząć od definicji ryzyka. Można je określić jako "wskaźnik stanu lub zdarzenia, które może prowadzić do strat. Jest ono proporcjonalne do prawdopodobieństwa wystąpienia tego zdarzenia i do wielkości strat, które może spowodować." [1]

Portfel inwestycyjny z kolei to "zbiór finansowych lub rzeczowych aktywów inwestora, które stanowią dla niego formę lokowania majątku. Wśród składników portfela inwestycyjnego znaleźć się mogą: gotówka, bankowe lokaty terminowe oraz lokaty z funduszem, akcje, obligacje, jednostki uczestnictwa w funduszach inwestycyjnych, nieruchomości, złoto, antyki, dzieła sztuki, kamienie szlachetne itp. Składniki te odznaczają się zróżnicowanym poziomem ryzyka inwestycyjnego, różną zyskownością, ale też różnym poziomem ich płynności (zbywalności). Stąd też w praktyce inwestorzy – w zależności od swoich możliwości oraz preferencji finansowych – tworzą mniej lub bardziej różnorodne portfele inwestycyjne. Bardzo jednak często specjaliści od inwestycji zalecają dywersyfikację (różnicowanie składu aktywów) portfela inwestycyjnego, które znacząco zmniejsza ryzyko inwestycyjne." [2]

Wspomniana dywersyfikacja jest niczym innym jak sposobem na zarządzanie ryzykiem w odniesieniu do inwestycji. Etapami, które obejmuje zarządzanie ryzykiem ogólnie, są:

1. "Określenie celu zarządzania ryzykiem – celem zarządzania ryzykiem może być minimalizacja prawdopodobieństwa wystąpienia danego zdarzenia lub ograniczenie negatywnych skutków zajścia takiego zdarzenia." [3] W przypadku inwestycji będzie to odpowiedni balans między ochroną kapitału a osiągnięciem zysku.
2. "Identyfikacja ryzyk" [3] związanych z ulokowaniem kapitału w dane aktywo.
3. "Ocena ryzyk – polega na oszacowaniu potencjalnych skutków zajścia zdarzenia oraz prawdopodobieństwa wystąpienia." [3]
4. "Wybór metody zarządzania ryzykiem – do najpopularniejszych metod należą: przeciwdziałanie ryzyku, przenoszenie ryzyka na inne podmioty, akceptacja ryzyka, wycofanie się z działań oraz monitorowanie ryzyka." [3] Najpopularniejszymi metodami zarządzania ryzykiem w ujęciu portfela inwestycyjnego są:
  - a) monitorowanie ryzyka, np. sytuacji ekonomicznej na danym rynku
  - b) przeciwdziałanie ryzyku poprzez równoważenie (eng. rebalancing) portfela
  - c) wycofanie się w przypadku niesprzyjających okoliczności (np. *stop loss*) lub osiągnięcia zakładanego zysku (np. *take profit*).

## 1.2. Dywersyfikacja i inwestycje

Każda z wymienionych wcześniej inwestycji ma swoje zalety - gotówka pozwala zachować określoną kwotę i ogranicza ryzyka związane z obsługą pieniędzy przez bank.

Nieruchomość może skutecznie chronić przed inflacją i nawet jeśli będzie ona stała pusta przez rok czy dwa jej wartość drastycznie nie spadnie - w perspektywie kilku/kilkunastu lat inwestycja powinna się opłacić. Akcje, gdy inwestuje się mądrze potrafią przynieść zysk wyższy niż realna stopa inflacji oraz stanowić dodatek do bieżących przychodów. Z kolei inwestycje alternatywne przynoszą spore korzyści finansowe osobom znającym ich rynek.

O ile nie da się znaleźć złotego środka i doradzić każdemu inwestycji, która będzie dla niego najlepsza, jedno jest pewne: dywersyfikacja to klucz do ochrony kapitału. Każde z wymienionych we wstępie ryzyk może się wydarzyć, jednakże im większa ilość zastosowanych metod oszczędzania (lub inwestowania) tym mniejsza szansa na utratę znaczącej części kapitału.

Można wyszczególnić wiele poziomów dywersyfikacji:

1. Rodzaj aktywów. Przykładowo, zamiast inwestować jedynie w akcje przeznaczać pieniądze zarówno na akcje, kruszce oraz gotówkę.
2. Ilość aktywów należących do określonego rodzaju. Przykładem dla kruszców może być inwestowanie nie w jeden kruszec, np. złoto, ale również w srebro i platynę.
3. Miejsca przechowywania poszczególnych aktywów. Akcje pewnej firmy można posiadać na dwóch kontach u niezależnych maklerów, a gotówkę gromadzić w postaci fizycznej oraz w niezależnych bankach.

Skuteczna dywersyfikacja pozwala znacznie zminimalizować ryzyko. Niestety rodzi ona kolejny problem, którym jest trudność monitorowania wszystkich aktywów. Przy posiadaniu portfela złożonego z gotówki, akcji, nieruchomości i kruszców trudno określić ile faktycznie pieniędzy się posiada. Ograniczenie ryzyka nie powinno odbijać się negatywnie na wiedzy o stanie finansowym.

### 1.3. Istniejące rozwiązania

Do tej pory zostało rozwiniętych sporo rozwiązań mobilnych pozwalających monitorować pozycje takie jak akcje, instrumenty ETF, kruszce, waluty czy kryptowaluty. Aplikacje zweryfikowane podczas przygotowań do napisania omawianej pracy inżynierskiej można podzielić na trzy rodzaje:

1. Aplikacje do śledzenia bieżącej ceny aktywów i czytania informacji typu "news" na ich temat. Przykład: Yahoo Finance. Gromadzą one bieżące informacje i przedstawiają prosty wykres ceny w czasie. Niestety, nie umożliwiają tworzenia własnego portfolio.
2. Aplikacje do przeprowadzania analizy technicznej. Przykład: Trading View. Bardzo dobra aplikacja adresująca potrzebę wglądu w wykres świecowy aktywa niemal każdego rodzaju oraz przeprowadzenia skrupulatnej analizy wykresu z rozbudowanymi narzędziami do rysowania linii trendu, obszarów czasowych i cenowych oraz wieloma wskaźnikami. Ponownie, brak tworzenia portfolio.
3. Aplikacje typu portfolio. Przykład: Investfolio. Jest to aplikacja, która umożliwia stworzenie własnego portfolio oraz wgląd w prosty wykres ceny. Brak w niej opcji

posiadania dwóch oddzielnych pozycji na to samo aktywo, tj. po dodaniu dwóch pozycji z akcjami firmy X sumują się one do jednej. Nie pozwala to na posiadanie dwóch różnych pozycji na akcjach tej samej firmy na dwóch rachunkach maklerskich. Również gotówka jest niestety "nierozbijalna" - co nie daje opcji przedstawienia oddzielnie tej, która (przykładowo) jest w domu, banku nr 1 oraz banku nr 2.

Można zauważyć powyżej, iż wymienione rozwiązania nie umożliwiają łatwego śledzenia pozycji na tym samym walorze przechowywanym w dwóch różnych miejscach.

#### **1.4. Wstępne założenia pracy**

Aplikacje wspomniane w poprzednim podrozdziale rozwijane były przez lata przez zespoły deweloperskie aby dojść do formy, w której są teraz - profesjonalny i rozbudowany interfejs użytkownika, platforma webowa, mnogość oferowanych funkcji oraz szeroki wachlarz aktywów. Przez ograniczone zasoby przy rozwoju pracy inżynierskiej podjęta została decyzja o zaadresowaniu głównego braku wymienionych rozwiązań - dodawanie kilku odrębnych pozycji na tym samym walorze (ograniczone do akcji na rynku amerykańskim).

## 2. Założenia oraz wybór narzędzi

### 2.1. Założenia funkcjonalne

Celem pracy było "stworzenie aplikacji mobilnej umożliwiającej łatwe monitorowanie finansów na rynku akcji amerykańskich". Aby osiągnąć te cele zdefiniowane zostały następujące wymagania funkcjonalne:

1. Dodawanie nowych pozycji z amerykańskiego rynku akcji.
2. Informowanie o cenie kupna, ilości, zysku oraz aktualnej cenie aktywów.
3. Tworzenie konta użytkownika.
4. Logowanie.
5. Dostęp do zapisanych pozycji z dowolnego urządzenia z systemem Android.

Aby zarządzanie wymaganiami i funkcjami oraz śledzenie postępów było możliwie łatwe, a nowe pomysły na usprawnienia zostały zapamiętane w projekcie zostało użyte oprogramowanie Jira.

### 2.2. Wybór technologii

#### 2.2.1. Język programowania

Od ponad 4 lat faworytem w rozwijaniu aplikacji natywnych na systemy Android jest język Kotlin. Wersja 1.0 opublikowana została oficjalnie 15. lutego 2016 roku przez firmę JetBrains. Utrzymywany przez JetBrains oraz społeczność język został w 2017 roku ogłoszony przez Google rekomendowanym dla platformy Android. Może być on wykorzystywany również do rozwoju aplikacji webowych zarówno po stronie interfejsu użytkownika (*frontend*) jak i logiki (*backend*).

W porównaniu do Javy, której (w przypadku rozwiązań mobilnych) stał się następcą jest bardziej zwięzły co ułatwia programiście napisanie tych samych funkcjonalności przy mniejszej liczbie linii kodu. Inną jego zaletą jest eliminacja błędów odwołania (*null-pointer safety*). Skutkiem tego jest konieczność świadomej deklaracji programisty, czy obsługiwane zmienne, mogą/nie mogą mieć wartość *null*, co z kolei wymusza obsługę takich wartości i nie dopuszcza do nieokreślonych zachowań wywołanych niezadeklarowanymi zmiennymi. [4]

Dowodem na przemyślane rozwinięcie języka Kotlin jest to, jak bardzo jest on ceniony przez deweloperów. W ankiecie dla stackoverflow.com z 2020 roku znalazł się on na 4. miejscu w kategorii "Most Loved Languages" z wynikiem 62,9%. [5]

Alternatywami dostępnymi na rynku są: wspomniana wcześniej Java umożliwiająca rozwój aplikacji natywnych oraz rozwiązania multi-platformowe, np. React Native (JavaScript), Xamarin (C#) czy Flutter (Dart).

### 2.2.2. Platforma Backend as a Service

Niektóre z założeń funkcjonalnych nie mogły zostać zaadresowane z poziomu jedynie aplikacji mobilnej. Wymagały one strony serwerowej, która umożliwiłaby implementację następujących z nich:

1. Tworzenie konta użytkownika.
2. Logowanie.
3. Dostęp do zapisanych pozycji z dowolnego urządzenia z systemem Android.

Tradycyjne rozwiązanie zakładałoby stworzenie autorskiego *backendu*. Jednakże, jako że praca skupia się na aspekcie mobilnym zdecydowano o skorzystaniu z nowoczesnej platformy typu *Backend as a Service (BaaS)*. Dzięki tego typu rozwiązaniom do rozwoju aplikacji nie jest potrzebny już cały zespół deweloperski składający się zarówno z backendowców i programistów mobilnych, gdyż implementują one najpotrzebniejsze usługi typu backend po swojej stronie.

Jednym z najpopularniejszych i najbardziej rozbudowanych serwisów tego typu jest Firebase - narzędzie tworzone przez Google. Zawiera ono szereg narzędzi wspomagających utrzymanie oraz rozwój aplikacji. Są to między innymi: analityka (np. *Events*), monitorowanie zdarzeń produkcyjnych (np. *Crashlytics*), komunikacja z użytkownikami (np. *Cloud Messaging* oraz *In-app Messaging*), zdalne konfiguracje (*Remote Config*) czy nawet wspomaganie testowania (*Test Lab*).

Na potrzebę niniejszej pracy konieczne było wybranie rozwiązania *BaaS*, które adresuje wspomniane wymagania funkcjonalne poprzez spełnienie następujących wymagań technicznych:

1. Tworzenie użytkowników.
2. Zarządzanie sesją użytkownika.
3. Relacyjna baza danych.

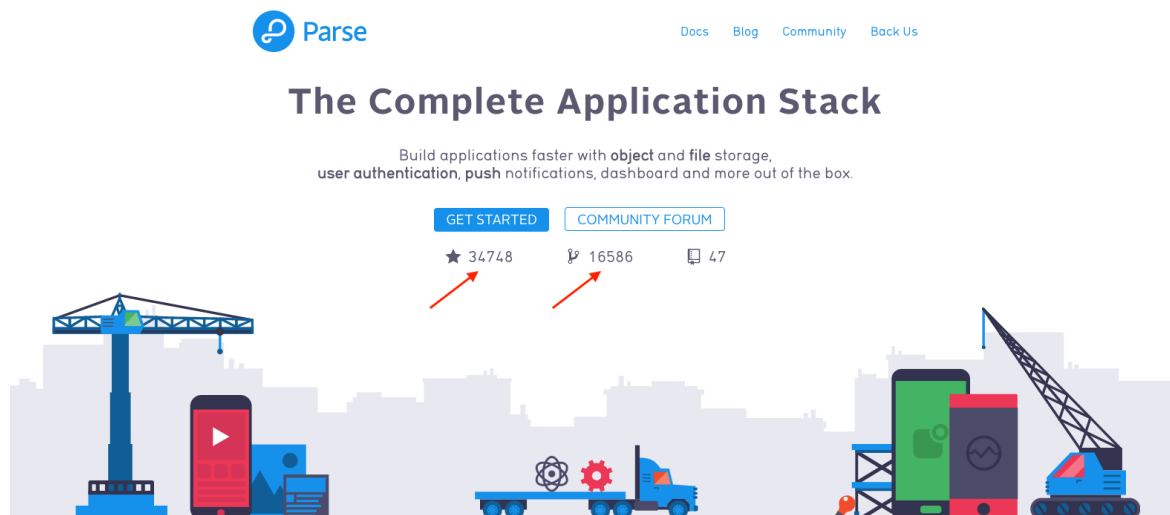
Po porównaniu rozwiązań typu *BaaS* Firebase został odrzucony ze względu na bazę danych typu *NoSQL*, zaś wybrana została platforma *Back4App*. Bazuje ona na dobrze udokumentowanym rozwiązaniu - *Parse*.

*Parse* jest rozwiązaniem typu *open-source* aktywnie rozwijanym oraz lubianym przez społeczność. Świadczą o tym statystyki z portalu *GitHub*, które można znaleźć na głównej stronie *Parse* (Rysunek 2.1) - prawie 35 tysięcy oznaczeń gwiazdką oraz 16,5 tysiąca *forków* (stan na dzień 17.01.2021).

Niestety aby korzystać z narzędzi oferowanych przez *Parse* takich jak: relacyjne bazy danych, powiadomienia typu *push* czy obsługa użytkowników (w tym rejestracja, weryfikacja adresu e-mail, resetowanie hasła oraz zarządzanie sesją i dostępem) konieczne jest hostowanie usługi na własnej infrastrukturze.

Z pomocą przychodzi *Back4App*, czyli "Parse w chmurze". Dzięki niemu, użytkownik może zaoszczędzić czas i pieniądze potrzebne na wdrożenie oraz aktualizacje *Parse* na

## 2. Założenia oraz wybór narzędzi



Rysunek 2.1. Strona *parseplatform.org* przedstawiająca zbiorcze statystyki z portalu *github.com*

The image shows the Back4App pricing page. It features a table of plans categorized into 'SHARED' and 'DEDICATED'. The 'SHARED' plans include FREE, \$5, \$25, \$50, and \$100. The 'DEDICATED' plans include SILVER (\$250), GOLD (\$400), and PLATINUM (\$1000). Each plan lists its usage limits for requests, database, data transfer, and file storage. A red arrow points to the 'FREE' plan in the 'SHARED' category.

	SHARED					DEDICATED		
	FREE	\$5	\$25	\$50	\$100	\$250	\$400	\$1000
	No Credit Card Required	Per app per month, billed monthly	Per app per month, billed monthly	Per app per month, billed monthly	Per app per month, billed monthly	Per month, billed monthly	Per month, billed monthly	Per month, billed monthly
	SIGN UP	BUY NOW	BUY NOW	BUY NOW	BUY NOW	BUY NOW	BUY NOW	BUY NOW
USAGE LIMITS								
Requests/Month	10k	50k	500k	2M	5M-\$0.02/k req	Unlimited	Unlimited	Unlimited
Database	250 MB	1 GB	2 GB	3 GB	4 GB + \$15/GB	40GB	80GB	200GB
Data Transfer	1 GB	50 GB	250 GB	500 GB	1 TB + \$0.1/GB	1 TB	2 TB	5 TB
File Storage	1 GB	10 GB	50 GB	150 GB	250 GB + \$0.1/GB	500 GB	1 TB	3 TB
Apps per Subscription	1	1	1	1	1	Up to 2	Up to 4	Up to 10

Rysunek 2.2. Strona *back4app.com/compare-all-plans* przedstawiająca plany cenowe rozwiązania Back4App

własnej infrastrukturze. To, co wyróżnia Back4App to przyjazny cennik pozwalający na napisanie pracy inżynierskiej bez konieczności zakupu wersji płatnej (Rysunek 2.2).

### 2.3. Wybór API giełdowego

W celu zapewnienia użytkownikowi aktualnych informacji na temat pozycji, które dodał do swojego portfela konieczne było zintegrowanie aplikacji z API dostarczającym takie informacje. Dostawcy danych giełdowych zostali porównani głównie ze względu na popularność, jakość dokumentacji, dostępność darmowych planów oraz informacji o różnych rodzajach aktywów, które mogą zapewnić.

Początkowy wybór padł na Alpha Vantage - dobrze udokumentowane API, które w darmowej wersji okazało się mieć dwa poważne ograniczenia:

1. Limit zapytań - 5 zapytań na minutę i 500 zapytań na miesiąc. Limit ten nie pozwala na wyświetlenie użytkownikowi aktualnych cen wszystkich pozycji w akceptowalnym

### Stock Candles

Get candlestick data for stocks going back 25 years for US stocks.

Real-time stock prices for international markets are supported for Enterprise clients via our partner's feed. [Contact Us](#) to learn more.

Method: [GET](#)

Free Tier: 1 year of historical data and new updates

#### Examples:

[/stock/candle?symbol=AAPL&resolution=1&from=1605543327&to=1605629727](#)

[/stock/candle?symbol=IBM&resolution=0&from=1572651390&to=1575243390](#)

#### Arguments:

**symbol** [REQUIRED](#)

Symbol.

**resolution** [REQUIRED](#)

Supported resolution includes `1`, `5`, `15`, `30`, `60`, `D`, `W`, `M`. Some timeframes might not be available depending on the exchange.

**from** [REQUIRED](#)

UNIX timestamp. Interval initial value.

**to** [REQUIRED](#)

UNIX timestamp. Interval end value.

**format** [optional](#)

By default, `format=json`. Strings `json` and `csv` are accepted.

#### Response Attributes:

**o**  
List of open prices for returned candles.

**h**  
List of high prices for returned candles.

**l**  
List of low prices for returned candles.

**c**  
List of close prices for returned candles.

**v**  
List of volume data for returned candles.

**t**  
List of timestamp for returned candles.

**s**  
Status of the response. This field can either be `ok` or `no_data`.

Sample code [cURL](#)

```
1 curl "https://finnhub.io/api/v1/stock/candle?symbol=AAPL&resolution=1&from=1605543327&to=1605629727"
2
```

Sample response

```
1 {
2   "c": [
3     217.68,
4     221.03,
5     219.89
6   ],
7   "h": [
8     222.49,
9     221.5,
10    220.94
11   ],
12   "l": [
13     217.19,
14     217.1402,
15     218.83
16   ],
17   "o": [
18     221.03,
19     218.55,
20     220
21   ],
22   "s": "ok",
23   "t": [
24     1569297600,
25     1569384000,
26     1569470400
27   ],
28   "v": [
29     33463820,
30     24018876,
31     20730608
32   ]
33 }
```



**Rysunek 2.3.** Dokumentacja endpointu *Candles* Finnhub API

czasie. Zostało to uznane za ograniczenie krytyczne dla realizacji omawianej pracy inżynierskiej.

2. Limit dostępnych giełd - Alpha Vantage nie posiada danych z Warszawskiej Giełdy Papierów Wartościowych (GPW). Ograniczenie to zostało również uznane za krytyczne w przypadku gdyby prace nad rozwojem aplikacji były kontynuowane poza pracą inżynierską.

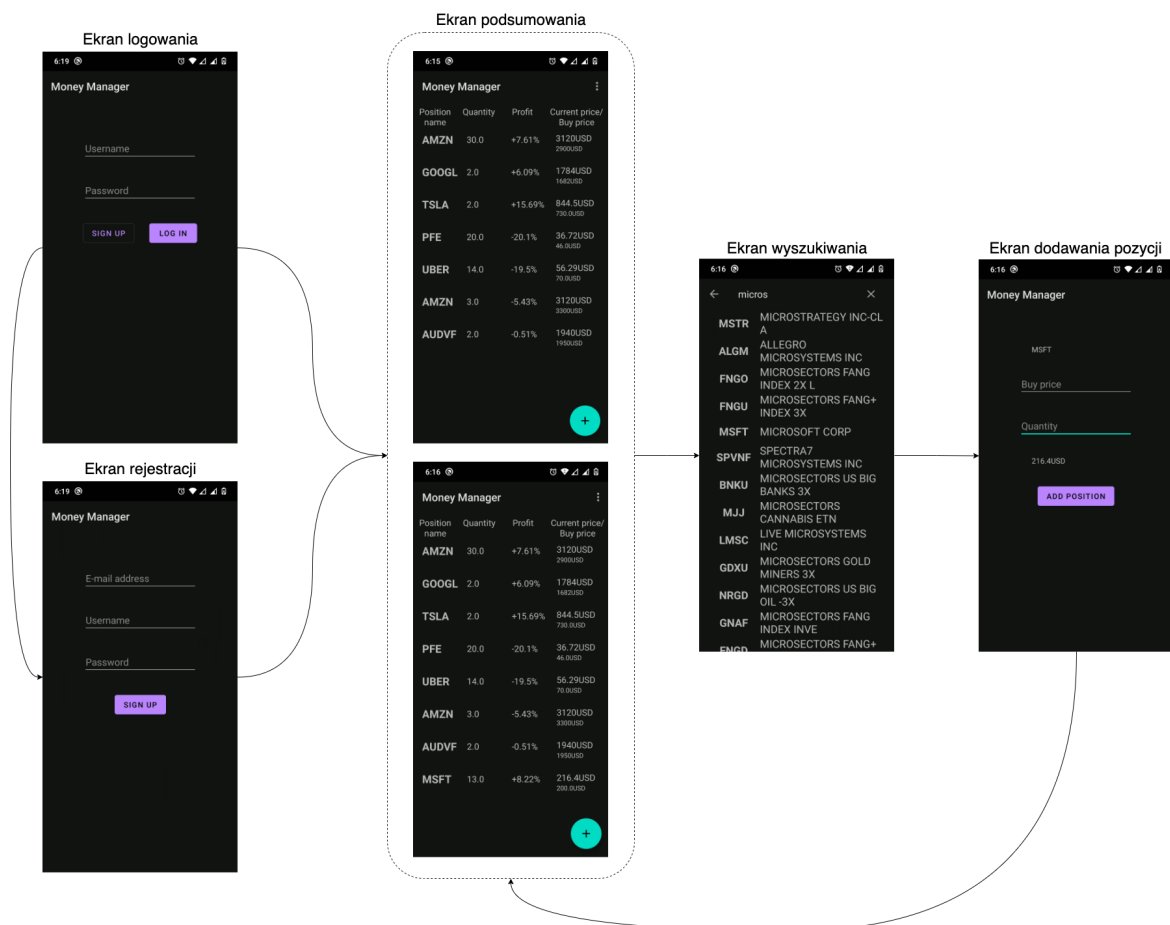
Po wykluczeniu Alpha Vantage jako API zapewniającego informacje giełdowe padł wybór na Finnhub. Nieco mniej popularne, jednakże bogatsze rozwiązanie dostarczające informacje z ponad 60 giełd z całego świata. Posiada dobrze udokumentowane API, oraz wyższe (dla wersji darmowej) limity - 60 zapytań na minutę. Podobnie jak Alpha Vantage poza danymi giełdowymi zapewnia dane z rynku Forex czy kryptowalut. Posiada wygodne RESTowe API, które jest dobrze udokumentowane (Rysunek 2.3). Przebieg zmiany API został dokładniej opisany w rozdziale 3.

### 3. Implementacja i wyzwania

#### 3.1. Zaimplementowane rozwiązanie

Finalna aplikacja realizuje kolejne kroki (Rysunek 3.1):

1. Alternatywnie
  - a) Rejestracja użytkownika
  - b) Logowanie użytkownika
2. Przejście do ekranu podsumowania, na którym wylistowane są wszystkie pozycje użytkownika.
  - a) Jeśli użytkownik nie posiada żadnych pozycji w lokalnej bazie danych są one pobierane ze zdalnej bazy danych.
  - b) Możliwość wylogowania się.
3. Wyszukanie symbolu do dodania z amerykańskiego rynku akcji.
4. Dodanie parametrów pozycji takich jak cena zakupu oraz ilość.
5. Dodanie pozycji i wyświetlenie na ekranie podsumowania.



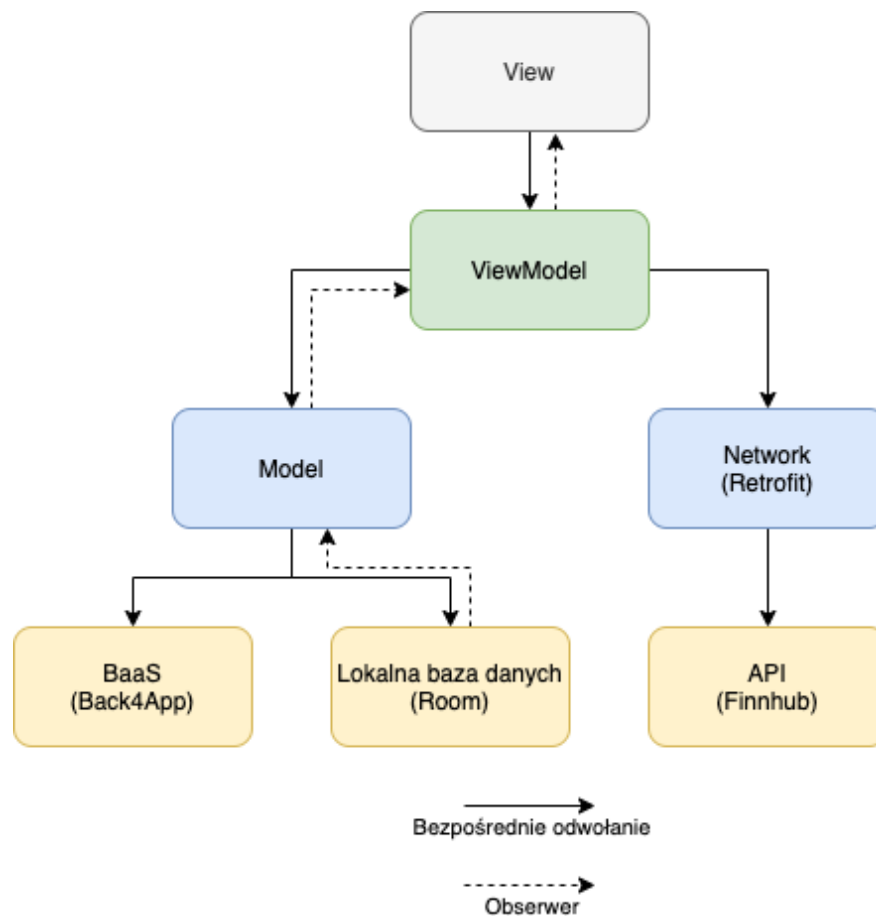
Rysunek 3.1. Rozkład ekranów oraz nawigacja między nimi



### 3.2. Architektura rozwiązania

Jako, że do pobrania najnowszych informacji o pozycjach giełdowych potrzebny jest dostęp do internetu główne funkcjonalności aplikacji takie jak: logowanie i rejestracja użytkownika, czy wyszukiwanie i dodawanie nowych symboli giełdowych dostępne są jedynie w trybie online. Bez dostępu do internetu możliwe jest jedynie sprawdzenie stanu portfela sprzed ostatniej synchronizacji.

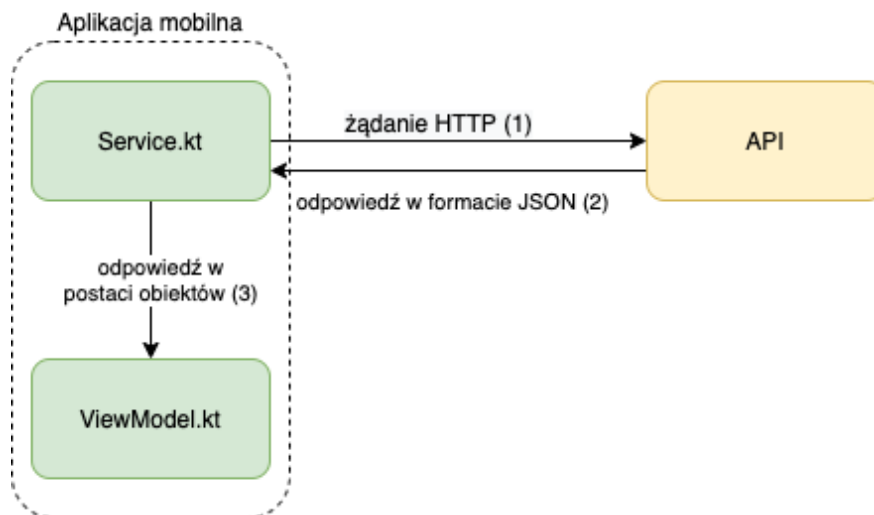
Stworzona na potrzeby pracy inżynierskiej architektura przedstawiona na Rysunku 3.2 zakłada brak przestojów (*downtime*) po stronie dostawcy API oraz BaaS. Użyty został oficjalnie rekomendowany przez Google wzorzec architektoniczny Model-View-ViewModel (MVVM).



Rysunek 3.2. Architektura rozwiązania

### 3.3. Network - API giełdowe i Retrofit

13. maja 2013 roku firma Square opublikowała *Retrofit* [6] - bibliotekę open-source, która umożliwia mapowanie odpowiedzi RESTowych API na obiekty Javy (oraz Kotlin, który działa na maszynie wirtualnej Javy). Jest to implementacja koncepcji *Data Transfer Objects (DTO)*. Wprowadza ona warstwę abstrakcji dla programisty, co jest kolejnym czynnikiem przyspieszającym pracę.



**Rysunek 3.3.** Schemat komunikacji z API

```
@GET( value: "stock/symbol?exchange=US&token=XXX")
suspend fun getSymbolsFromExchange(): List<Symbol>
```

**Rysunek 3.4.** Żądanie GET do API

Komunikację z API można przedstawić w formie schematu jak na Rysunku 3.3. Kolejne kroki, to:

1. Wysłanie żądania HTTP (Rysunek 3.4). Można zauważyć, że już w tym miejscu wartość zwracana przez żądanie jest określona jako lista obiektów klasy *Symbol*.
2. Przechwycenie odpowiedzi w formacie JSON (niewidoczne dla programisty, Rysunek 3.5)
3. Zmapowanie przy użyciu biblioteki *Moshi* (odpowiadającej za format JSON) oraz *DTO* odpowiedzi JSON na obiekty (Rysunek 3.6)

```
[
  {
    "currency": "USD",
    "description": "BROOGE HOLDINGS LTD",
    "displaySymbol": "BROGW",
    "figi": "BBG00R4Z6LH7",
    "mic": "XNCM",
    "symbol": "BROGW",
    "type": "Equity WRT"
  },
  {...},
  ...
]
```

**Rysunek 3.5.** Przykład odpowiedzi API w formacie JSON

```
data class Symbol(
    val description: String,
    val displaySymbol: String,
    val symbol: String,
    val type: String,
    val currency: String
) : Parcelable
```

Rysunek 3.6. Przykładowy *DTO*

```
{
  "bestMatches": [
    {
      "1. symbol": "TESO",
      "2. name": "Tesco Corporation USA",
      "3. type": "Equity",
      "4. region": "United States",
      "5. marketOpen": "09:30",
      "6. marketClose": "16:00",
      "7. timezone": "UTC-05",
      "8. currency": "USD",
      "9. matchScore": "0.8889"
    },
    { ... },
    ...
  ]
}
```

Rysunek 3.7. Przykładowa odpowiedź API Alpha Vantage oraz odpowiedniki pól w nowym modelu

### 3.3.1. Wyzwanie - zmiana API

Jak wspomniane zostało w rozdziale 2. przez początkowy wybór niewłaściwego API konieczna była zmiana na inne. Dzięki zaprezentowanej koncepcji *DTO* było to zadanie o niedużym stopniu złożoności dzięki względnej zgodności formatów odpowiedzi.

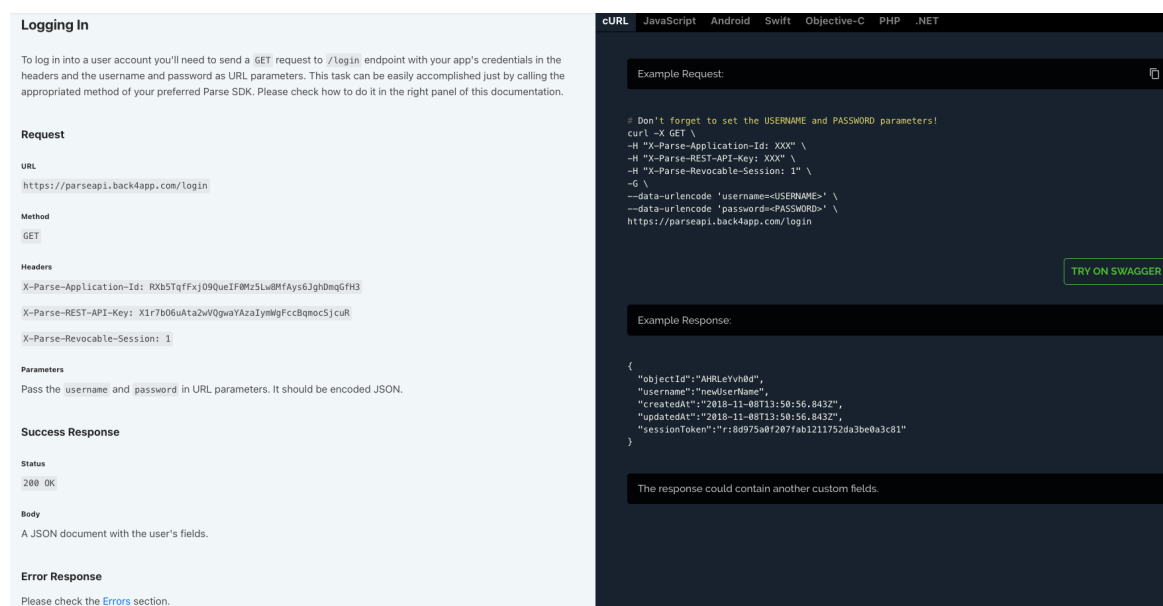
Przy realizacji projektu dodatkowym uproszczeniem było użycie zaledwie kilku pól (*symbol* oraz *description*) w aplikacji na ówczesnym etapie. Pozwoliło to zastąpić stary obiekt *Symbol* nowym bez utraty istotnych danych. Rysunek 3.7 przedstawia przykładową odpowiedź API Alpha Vantage, które użyte zostało w projekcie jako pierwsze oraz odpowiedniki pól w nowym modelu.

Dzięki zastosowaniu *DTO* oraz oddzieleniu modelu od logiki przy użyciu architektury *MVVM* operacje takie jak zmiana API mogą być niewidoczne dla wyższych warstw aplikacji.

## 3.4. Back4App - Backend as a Service

Przed rozpoczęciem użycia nowego, zewnętrznego narzędzia konieczne jest zapoznanie się z jego dokumentacją. Back4App jest rozwiązaniem, które dobrze opisuje kroki potrzebne do zintegrowania oraz używania go. Sam przebieg integracji jest prosty i wymaga dodania dodatkowej zależności w pliku *build.gradle*, zainicjowania Parse SDK w *MainActivity.kt* oraz dodania uprawnień sieciowych w pliku *AndroidManifest.xml*.

### 3. Implementacja i wyzwania



**Logging In**

To log in into a user account you'll need to send a `GET` request to `/login` endpoint with your app's credentials in the headers and the username and password as URL parameters. This task can be easily accomplished just by calling the appropriated method of your preferred Parse SDK. Please check how to do it in the right panel of this documentation.

**Request**

**URL**

`https://parseapi.back4app.com/login`

**Method**

`GET`

**Headers**

`X-Parse-Application-Id: Rxb5TqFxfj09QueIF0Wz5Lw8fAys6JghBmq6fh3`

`X-Parse-REST-API-Key: X1r7b06uA2wV0gwaY2aIynWgFcc8qmc5JcuR`

`X-Parse-Revocable-Session: 1`

**Parameters**

Pass the `username` and `password` in URL parameters. It should be encoded JSON.

**Success Response**

**Status**

`200 OK`

**Body**

A JSON document with the user's fields.

**Error Response**

Please check the [Errors](#) section.

**cURL** JavaScript Android Swift Objective-C PHP .NET

**Example Request:**

```
# Don't forget to set the USERNAME and PASSWORD parameters!
curl -X GET \
-H "X-Parse-Application-Id: XXX" \
-H "X-Parse-REST-API-Key: XXX" \
-H "X-Parse-Revocable-Session: 1" \
-G \
--data-urlencode 'username=USERNAME' \
--data-urlencode 'password=PASSWORD' \
https://parseapi.back4app.com/login
```

**Example Response:**

```
{
  "objectId": "AHRLeYvH0d",
  "username": "newUserName",
  "createdAt": "2018-11-08T13:56:56.843Z",
  "updatedAt": "2018-11-08T13:56:56.843Z",
  "sessionToken": "r:8d975a0f207fab1211752da3be0a3c81"
}
```

The response could contain another custom fields.

**Rysunek 3.8.** Opis funkcji Parse wraz z przykładem komendy cURL z dokumentacji Back4App

Parse SDK to warstwa abstrakcji nad RESTowym API, które wykorzystywane jest pod spodem. Przed użyciem każdej funkcji istnieje opcja łatwego przetestowania jej za pomocą gotowego *cURLa* (Rysunek 3.8) czy *Swaggera*. Ułatwia to przygotowanie się do prawidłowej obsługi danej funkcji przed przystąpieniem do implementacji. Back4App zapewnia również przykłady użycia dla najpopularniejszych platform/języków programowania, w tym dla Androida (Rysunek 3.9).

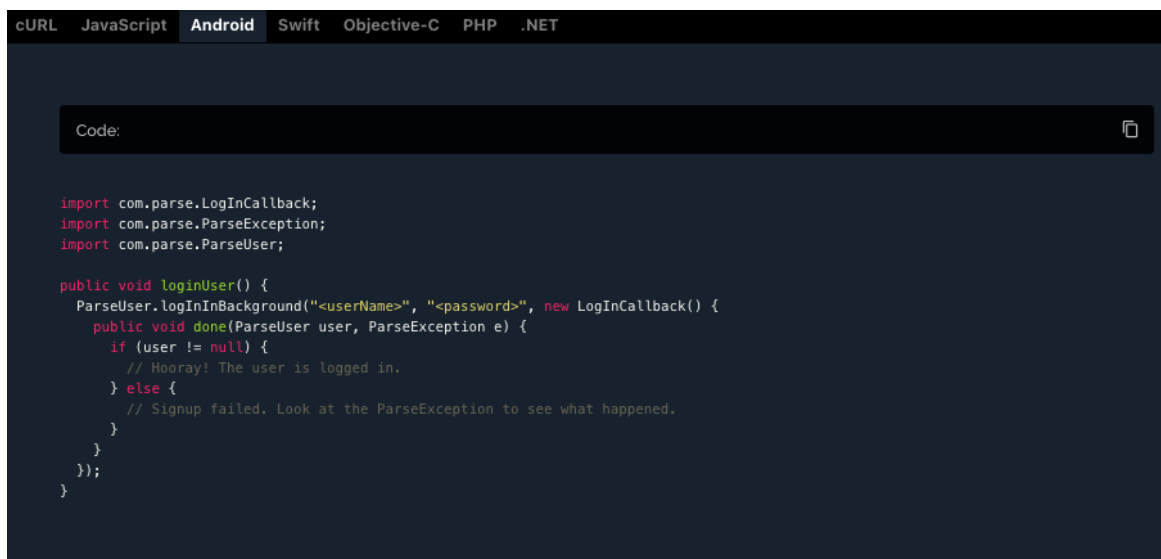
Bardziej szczegółowe opisy funkcji oraz więcej przykładów można znaleźć w dokumentacji Parse ([docs.parse-platform.org/](https://docs.parse-platform.org/)). Można z nich przykładowo dowiedzieć się, iż większość funkcji występuje w formie synchronicznej, blokującej główny wątek oraz w formie *InBackground* - asynchronicznej. Drugi wariant powoduje wywołanie operacji w tle bez konieczności blokowania głównego wątku.

#### 3.4.1. Rejestracja i logowanie - Parse Users

Parse ułatwia zarządzanie użytkownikami dzięki klasom *Session* oraz *User*, które udostępnia. Wraz z dobrymi praktykami hasło użytkownika nie jest widoczne w konsoli. *User* jest podklasą *Parse Object*, czyli działa podobnie jak każdy obiekt w Parse z kilkoma drobnymi wyjątkami (jak na przykład wspomniane ukryte pole "hasło" w konsoli).

Do utworzenia obiektu *ParseUser* wymagane jest podanie adresu e-mail, nazwy użytkownika oraz hasła (użyta metoda HTTP - POST). Odpowiedzią na żądanie jest *sessionToken*, który wygasa po roku (możliwe nadpisanie domyślnej konfiguracji).

Dostępna opcją dla osób, które chcą umożliwić swoim użytkownikom dostęp do aplikacji bez konieczności rejestracji jest anonimowy dostęp. Pozwala to na stworzenie sesji użytkownika wygasającej po 5 dniach, podczas których potencjalnie zainteresowana osoba może zweryfikować czy warto założyć konto. Innym przykładem użycia może być



**Rysunek 3.9.** Przykład implementacji funkcji dla systemu Android z dokumentacji Back4App

tu aplikacja nie wymagająca logowania użytkowników. Opcja ta nie została użyta w pracy inżynierskiej.

Dzięki oferowanym przez Parse narzędziom implementacja rejestracji użytkownika przebiega w sposób pokazany na Rysunku 3.10, co można opisać w następujących krokach:

1. Po naciśnięciu przycisku rejestracji odczytaj i sformatuj dane podane przez użytkownika (nazwy użytkowników oraz ich e-maile są przechowywane przy użyciu małych liter).
2. Przeprowadź walidację wprowadzonych danych (właściwy format e-maila, nazwa użytkownika zawierająca min. 5 znaków oraz hasło składające się z min 8 znaków w tym małych liter, wielkich liter, cyfry oraz znaku specjalnego). W przypadku niepowodzenia wyświetl *SnackBar* z treścią błędu.
3. Utwórz konto użytkownika i przenieś go na główny ekran aplikacji widoczny jedynie dla użytkowników z aktywną sesją.

Po zintegrowaniu Parse SDK z każdego miejsca w aplikacji można odwołać się do klasy *ParseUser* i wywołać metodę *getCurrentUser()* zwracającą aktualnego użytkownika. Ułatwia to weryfikację, czy w danym momencie użytkownik powinien móc zobaczyć dany ekran (przykład - Rysunek 3.11). Jednocześnie, łatwym jest zrealizowanie wylogowania z aplikacji. Aby usunąć sesję danego użytkownika wystarczy wywołać metodę *logout()* na klasie *ParseUser*.

```
fun onSignUpButtonClicked() {
    var signUpReady = false
    val username = usernameFormatter(username.value)
    val password = password.value
    val email = emailFormatter(email.value)

    when {
        !emailValidator(email) -> _errorMessage.value = emailValidationError
        !usernameValidator(username) -> _errorMessage.value = usernameValidationError
        !passwordValidator(password) -> _errorMessage.value = passwordValidationError
        else -> signUpReady = true
    }

    if (signUpReady) {
        val user = ParseUser()
        user.username = username
        user.email = email
        user.setPassword(password)
        user.signUpInBackground { e ->
            if (e == null) {
                _navigateToSummary.value = true
            } else {
                _errorMessage.value = parseErrorFormatter(e)
            }
        }
    }
}
```

Rysunek 3.10. Kod rejestracji użytkownika

```
init {
    if (ParseUser.getCurrentUser() != null) {
        viewModelScope.launch { this: CoroutineScope
            refreshPositions()
            updatePrices()
        }
    } else {
        _errorMessage.value = AUTH_ERROR_MESSAGE
        _navigateToLogin.value = true
    }
}
```

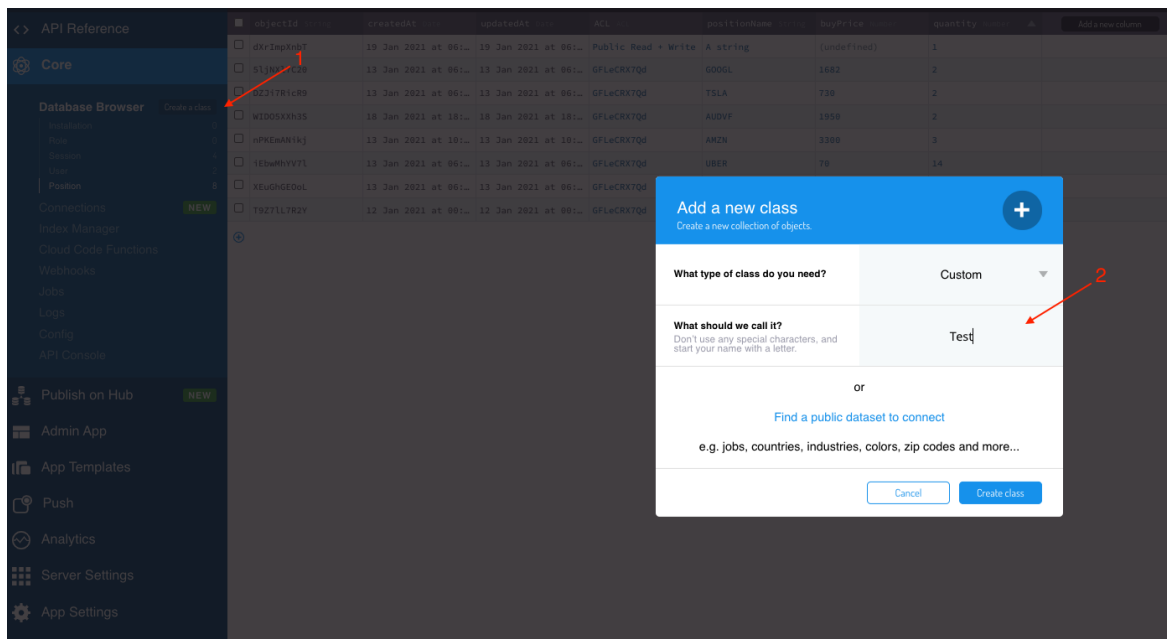
Rysunek 3.11. Weryfikacja aktualnego użytkownika

#### 3.4.2. Baza danych - Parse Objects

Parse każdy obiekt traktuje jako JSON, na którym można wywołać metody:

1. `put("key", value)`, gdy tworzy się obiekty i chce się je wysłać do zdalnej bazy danych.
2. `getX("key")`, gdy odczytuje się obiekty z serwera, gdzie X to typ zmiennej, który chcemy odczytać.

Tworzenie obiektów Parse w zdalnej bazie danych nie przebiega jak w większości projektów, gdzie pewien zespół współtworzy zarówno *frontend* jak i *backend*. W tradycyjnym



**Rysunek 3.12.** Tworzenie pustej klasy w konsoli Parse

podejściu zespół taki uzgadnia strukturę danych, *frontend* obliguje się do wysłania danych we właściwym formacie, zaś *backend* waliduje poprawność przesyłanych danych.

W rozwiązaniu Back4App domyślną bazą danych jest MongoDB (istnieje możliwość zmiany na PostgreSQL), a dodawanie do niego nowych encji działa na dwa sposoby - opisany już wcześniej tradycyjny oraz specyficzny dla tego rozwiązania.

Kolejne kroki rozwiązania "tradycyjnego" implementowanego przez Parse są intuicyjne:

1. Stworzenie nowej klasy oraz wybór jej nazwy (unikalność nazw w zakresie tej samej bazy). (Rysunek 3.12)
2. Po naciśnięciu "Create class" powstaje po stronie serwera nowy *endpoint*, do którego można wysyłać zapytania. Ponadto, z poziomu Back4App automatycznie generowana jest sekcja dokumentacji poświęcona właśnie tej encji bazy danych (Rysunek 3.13).
3. Dopiero po stworzeniu nowej encji możliwe jest zdefiniowanie kolumn, ich typu oraz rodzaju (obowiązkowe/opcjonalne). (Rysunek 3.14) Wszystkie liczby rozpoznawane są tu jako typ *Number* bez podziału na *Int*, *Long*, *Float*, *Double*, etc.
4. Po zdefiniowaniu odpowiednich kolumn są one widoczne w konsoli, zaś dokumentacja oraz Swagger są automatycznie uaktualniane. Warto dodać, że Parse dla każdego obiektu tworzy domyślne pola, takie jak: *objectId*, *createdAt*, *updatedAt* oraz *ACL* (Rysunek 3.15).
5. Dodatkową rzeczą, którą można łatwo wykonać z poziomu konsoli jest nałożenie indeksów na określone kolumny.

To, co wyróżnia Parse i może być zarówno jego zaletą jak i wadą to specyficzna dla Parse metoda na tworzenie nowych encji oraz ich kolumn:

### 3. Implementacja i wyzwania

#### Test Class

Test is a custom class that was created and is specific for Money Manager. Please use the following documentation to learn how to perform CRUD (create, read, update and delete) operations to this specific class. A new endpoint was automatically generated at the address below to which you can send your requests:

<https://parseapi.back4app.com/classes/Test>

The following fields are supported by this class' schema and can be used in the operations:

Name	Type	Example
------	------	---------

#### Creating Objects

To create a new object of the Test class, you'll need to send a POST request to the Test class' endpoint with your app's credentials in the headers and the object's data in the body. You can include as many key-value pairs of the supported fields as you want. This task can be easily accomplished just by calling the appropriated method of your preferred Parse SDK. Please check how to do it in the right panel of this documentation.

#### Request

URL

<https://parseapi.back4app.com/classes/Test>

Method

POST

Headers

X-Parse-Application-Id: RXb5TqfFxfj09QueIF0Mz5Lw8MFAys6jghDmq6fh3

X-Parse-REST-API-Key: X1r7b06uAta2wV0gwaYAzaIymHgFccBqnoc5JcuR

Content-Type: application/json

Body

A JSON document with the key-value pairs that represent your object's data according to the supported fields.

cURL JavaScript **Android** Swift Objective-C PHP .NET

Example JSON:

```
{
}
```

Code:

```
import java.util.Date;
import javax.json.JsonArray;
import javax.json.JsonObject;

import com.parse.ParseException;
import com.parse.ParseFile;
import com.parse.ParseObject;
import com.parse.ParseQuery;
import com.parse.ParseRelation;
import com.parse.SaveCallback;

public void createObject() {
    ParseObject entity = new ParseObject("Test");

    // Saves the new object.
    // Notice that the SaveCallback is totally optional!
    entity.saveInBackground(new SaveCallback() {
        @Override
        public void done(ParseException e) {
            // Here you can handle errors, if thrown. Otherwise, "e" should be null
        }
    });
}
```

Rysunek 3.13. Dokumentacja pustej klasy w Back4App

### Add a new column

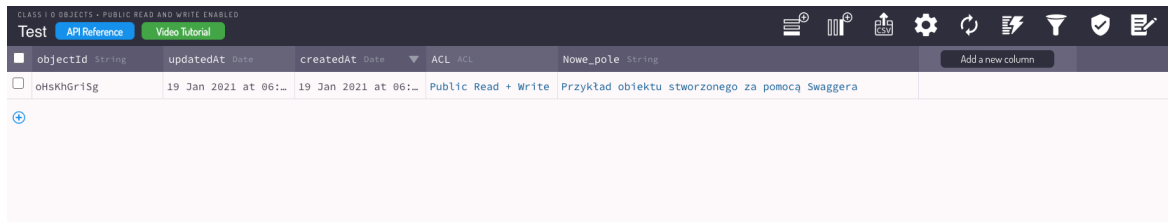
Store another type of data in this class.

What type of data do you want to store?	String
What should we call it? Don't use any special characters, and start your name with a letter.	Give it a good name...
What is the default value? If no value is specified for this column, it will be filled with its default value.	Set a default value here
Is it a required field? When true this field must be filled when a new object is created.	No <input checked="" type="radio"/> Yes

Never mind, don't.Add column

Rysunek 3.14. Dodawanie nowej kolumny w Parse





objectId	string	updatedAt	date	createdAt	date	ACL	Nowe_pole	string
oHskHGrISg		19 Jan 2021 at 06:...		19 Jan 2021 at 06:...		Public Read + Write	Przykład obiektu stworzonego za pomocą Swaggera	

**Rysunek 3.15.** Widok stworzonego obiektu w konsoli Parse

1. Wysłanie zapytania używając identyfikatora oraz klucza aplikacji do nieistniejącego endpointu z dowolnymi polami typu "klucz":wartość.
2. Stworzenie nowej klasy w Parse z typami pól zdeterminowanymi przez otrzymane dane. Wszystkie kolumny są oznaczone jako opcjonalne.
3. W celu dodania kolejnych kolumn - ponowne wysłanie zapytania do tego samego endpointu z dodanymi nowymi polami.

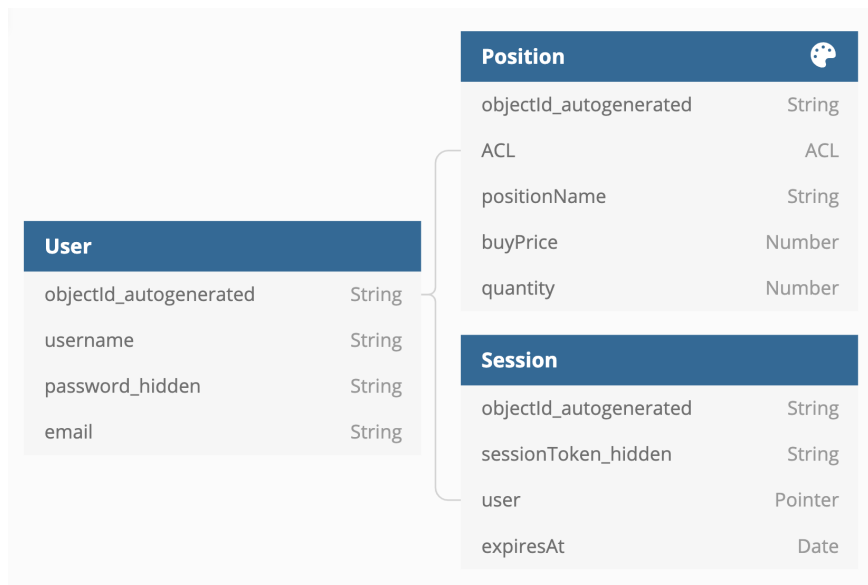
Przedstawione wyżej "specyficzne" rozwiązanie może wiązać się ze zbyt wieloma polami przy nierozważnym stosowaniu. W celu zapobiegania temu Parse umożliwia również opcję zakazującą tworzenia przez użytkowników nowych kolumn co częściowo adresuje luki tego rozwiązania.

### 3.4.3. Struktury baz danych

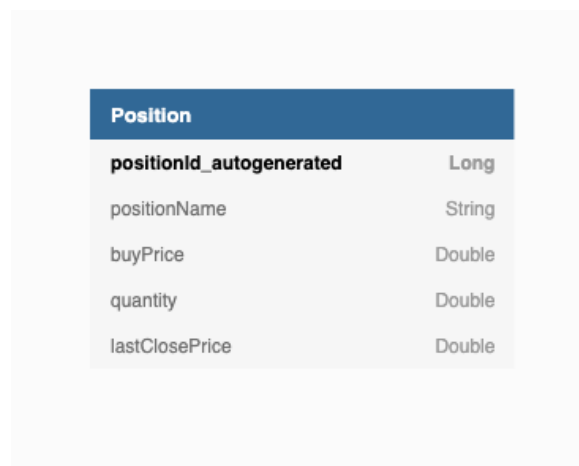
W centrum rozwiązania stoi użytkownik, który może dodawać pozycje giełdowe z rynku amerykańskiego i mieć do nich dostęp z wielu urządzeń. Uproszczona struktura zdalnej bazy danych (z pominięciem większości pól generowanych automatycznie przez Parse) odpowiada Rysunkowi 3.16. Jak łatwo zauważyć, backend nie posiada wiedzy o aktualnej cenie danego aktywa ze względu na założenie, że w każdym momencie można pobrać dane z API giełdowego, z którego dane o ostatniej cenie będą zawsze bardziej aktualne niż te przechowane na backendzie.

Z kolei baza danych po stronie klienta sprowadza się do jednej prostej tabeli reprezentującej pozycje, które posiada. (Rysunek 3.17) To, co ułatwia pracę to enigmatyczne pole *ACL* widoczne w schemacie bazy danych serwera. Jest to akronim od *Access Control List*. Wspomniane pole pozwala łatwo definiować dostęp do danych, co zostało użyte w aplikacji:

1. Przy zapisywaniu przez aplikację pozycji na backendzie dodanie linii *parsePosition.acl = ParseACL(ParseUser.getCurrentUser())* sprawia, że tylko użytkownik, który zapisał tę informację może ją odczytać.
2. Poprzez zdefiniowanie *ACL* przy zapisywaniu pozycji w zdalnej bazie pozyskanie wszystkich pozycji nie wymaga skomplikowanych zapytań, a jedynie wywołanie prostego *ParseQuery.getQuery("Position")*. Dzięki temu mechanizmowi użytkownik odczyta tylko dane, do których jest uprawniony, czyli publicznych (dane publiczne



**Rysunek 3.16.** Uproszczony schemat serwerowej bazy danych stworzony za pomocą dbdiagram.io



**Rysunek 3.17.** Schemat bazy danych klienta stworzony za pomocą dbdiagram.io

zostały pominięte jako nieistotne w trakcie implementacji) oraz zapisanych przez siebie samego.

#### 3.5. Dostęp do danych przyjazny użytkownikowi - Coroutines

Dane oraz czas związany z ich odczytem powoduje nierzadko problemy z wygodnym użytkowaniem aplikacji. Nierozważne zarządzanie komunikacją na linii aplikacja - baza danych/API może powodować zablokowanie głównego wątku przez co staje się ona nieużywalna przez jakiś czas i nie reaguje na akcje użytkownika.

Z pomocą przychodzi Kotlin z oznaczeniem *suspend* dla funkcji, które potencjalnie mogą zablokować główny wątek. Następuje konieczność wywoływania ich w tzw. *Coroutines*, które można przetłumaczyć jako "współprogramy", czyli wątki. Dla każdego z nich

```

private fun getAllSymbols() {
    viewModelScope.launch { this: CoroutineScope
        try {
            val result = FinnhubApi.finnhub.getSymbolsFromExchange()
            //symbols without description or with non-letters won't be shown
            allSymbols = result
                .filter { it.description.isNotEmpty() }
                .filter { !it.symbol.contains(regex = Regex( pattern: """"=+|\^+|#+|-+""")) }
            _searchableQueryResponse.value = allSymbols
        } catch (e: Exception) {
            if (e.message!!.contains( other: "resolve host")) {
                _errorMessage.postValue(NO_INTERNET_MESSAGE)
            } else {
                _errorMessage.postValue(e.message)
            }
        }
    }
}

```

**Rysunek 3.18.** Implementacja coroutine dla viewModelScope dla ekranu wyszukiwania

wymagane jest wywołanie w pewnym *scope*, który jest określeniem cyklu życia wątku, czyli w jakich okolicznościach dany "współprogram" powinien działać.

W przypadku omawianego projektu najczęściej stosowane są dwa *scopes*:

1. *ViewModelScope* - operacja w tle wykonuje się dopóki Fragment skojarzony z danym ViewModelem nie zniknie z ekranu. Przykładem takiej operacji jest pobranie listy symboli giełdowych które użytkownik może wyszukiwać. (Rysunek 3.18) Dane są pobierane z API póki użytkownik znajduje się na ekranie wyszukiwania. Po jego opuszczeniu operacja ta może przestać być wykonana, gdyż poza tym ekranem nie potrzebuje listy dostępnych symboli.
2. *Dispatcher.IO* - operacja wykonywana w kontekście bazy danych. Nawet gdy Fragment zniknie z ekranu operacja nie powinna zostać przerwana. Przykładem może być zapisanie nowej ceny dla pozycji w bazie 3.19.

### 3.6. Komunikacja pomiędzy warstwami w MVVM - LiveData

Obserwacją mogącą wynikać z poprzedniego podrozdziału może być sposób działania metody *updatePrices()*, która aktualizuje bazę danych, a (przynajmniej na pozór) nie aktualizuje ekranu użytkownika.

Operacje takie są możliwe dzięki zastosowaniu *LiveData* oraz *data bindingu*. Pozwalają one na stworzenie obiektów przetrzymujących dane, które są świadome cyklu życia, w którym są osadzone. Zmiany wprowadzane w *LiveData* są natychmiastowo wyświetlane na ekranie użytkownika.

Aby, przedstawiony w funkcji *updatePrices()* przykład mógł działać poprawnie, musi wydarzyć się kilka kroków:

1. Stworzenie zapytania w bazie danych Room (baza danych budowana na SQLite, która

```
private suspend fun updatePrices() {
    withContext(Dispatchers.IO) { this: CoroutineScope
        val allPositions = database.getAllPositionNames()

        allPositions.forEach { positionName ->
            try {
                database.updatePrice(positionName, getLastClosePrice(positionName))
            } catch (e: Exception) {
                if (e.message!!.contains( other: "resolve host")) {
                    _errorMessage.postValue(NO_INTERNET_MESSAGE)
                } else {
                    _errorMessage.postValue(e.message)
                }
            }
        }
    }
}
```

**Rysunek 3.19.** Implementacja coroutine dla Dispatcher.IO przy operacji na bazie danych

```
@Query( value: "SELECT * from positions ORDER BY positionId ASC")
fun getAllPositions(): LiveData<List<Position>>
```

**Rysunek 3.20.** Zapytanie SQL zwracające *LiveData*

wspiera *LiveData*) zwracającego *LiveData*. (Rysunek 3.20) Room przy każdej zmianie w bazie wywoła zapytanie ponownie, zwróci nowy wynik oraz uaktualni zawartość *LiveData*, które jest jego wynikiem.

2. Stworzenie stałej *LiveData* (*LiveData* - kontener na dane nie będzie zmieniany, ale już jego zawartość tak), która przechowywać będzie odpowiedź bazy na zapytanie SQL. Zostanie ona uaktualniona momentalnie po zmianie w bazie. (Rysunek 3.21)
3. Skomunikowanie widoku w pliku XML bezpośrednio z viewModelem, przez co pośrednio również z bazą. (Rysunek 3.22)
4. Dodatkowym krokiem w tym przypadku jest napisanie adaptera umożliwiającego wyświetlanie danych w formie listy, jednakże nie jest on częścią komunikacji przy użyciu *LiveData*, dlatego krok ten zostanie pominięty.

Przebieg komunikacji z użyciem *LiveData* w całym widoku podsumowania oraz z wyszczególnioną częścią odpowiadającą za metodę *updatePrices()* widoczny jest na rysunku 3.23.

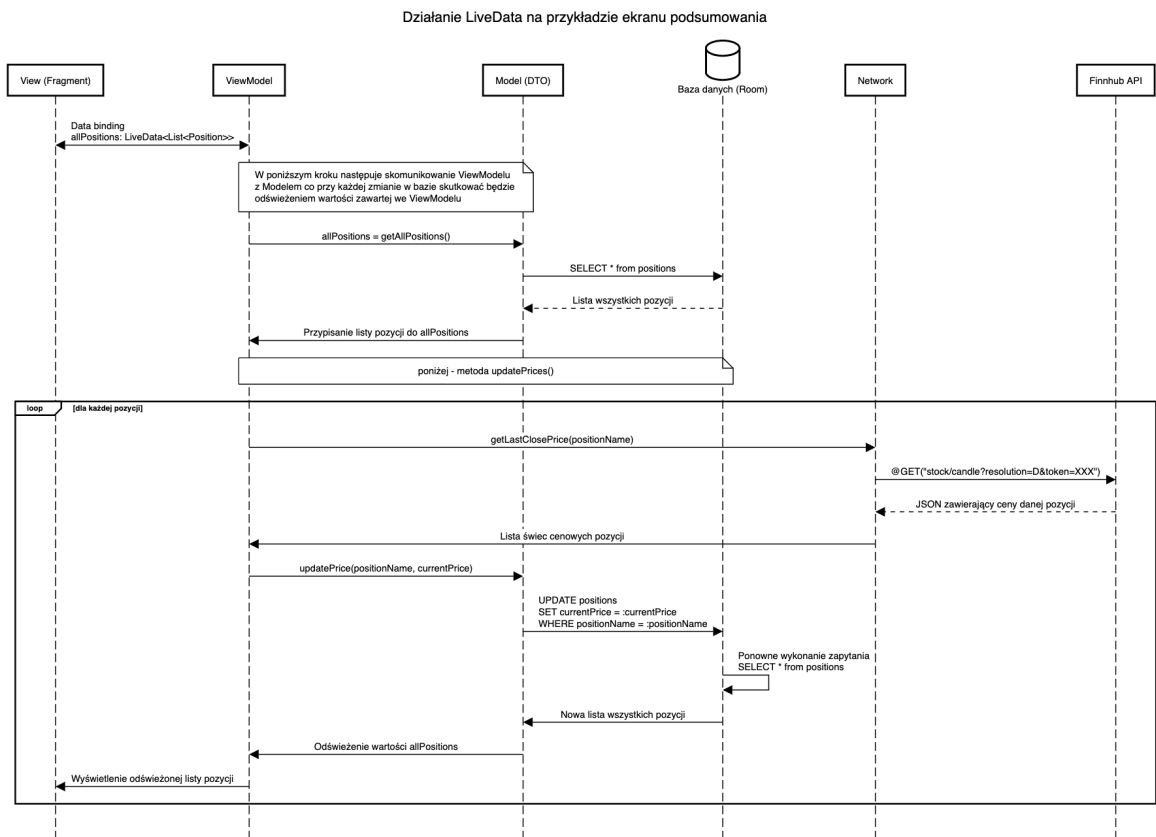
```
class SummaryViewModel(
    val database: PositionsDatabaseDao
) : ViewModel() {
    val allPositions: LiveData<List<Position>> = database.getAllPositions()
```

**Rysunek 3.21.** Inicjalizacja *LiveData* w *viewModelu*

```

<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/summary_list"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentStart="true"
    android:paddingStart="@dimen/padding_small"
    android:paddingEnd="@dimen/padding_small"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/summary_header"
    app:summaryData="@{viewModel.allPositions}"
    tools:listitem="@layout/list_summary_item" />

```

Rysunek 3.22. Referencja do *LiveData* we Fragmencie

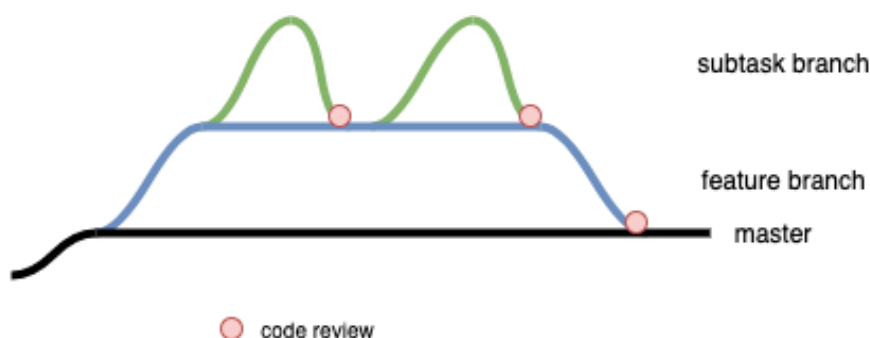
Rysunek 3.23. Diagram sekwencyjny opisujący mechanizm LiveData zastosowany w aplikacji

## 4. Weryfikacja rozwiązania

### 4.1. Code review i branching model

Praktyka, jaką jest inspekcja kodu (*code review*) może być stosowana również w jednoosobowych projektach. Dzięki spojrzeniu na zaimplementowany kod całościowo, a nie tylko z perspektywy linii kodu, którą aktualnie się rozwija łatwiej nabrać dystansu do czytanej treści i pomyśleć o potencjalnych błędach, nieścisłościach czy usprawnieniach.

W celu wprowadzenia tej praktyki efektywnie w omawianej pracy ustalony został *branching model* widoczny na Rysunku 4.1. Umożliwiał on weryfikowanie implementowanych funkcji na poziomie mniejszych, funkcjonalnych fragmentów kodu (*subtask branch code review*) oraz całych funkcji (*feature branch code review*). Podczas każdego z nich spisywane były komentarze, które pomogły utrzymać kod bardziej czytelnym, niż byłoby to w przypadku braku tej praktyki.



**Rysunek 4.1.** *Branching model* zastosowany w omawianym projekcie. Autorski rysunek z użyciem narzędzia draw.io

### 4.2. Testowanie

Dzięki zastosowanemu *branching modelowi* możliwe stało się wdrożenie regularnego testowania aplikacji wraz z każdą propagacją kodu na wyższy poziom (*subtask branch* -> *feature branch* -> *master*). Dzięki tej praktyce gałąź *master* pozostawała w stanie gotowości produkcyjnej.

Przykładowe scenariusze testowe wykonywane przy propagacji kodu na wyższą gałąź (testowane były obszary dotknięte zmianami), to m.in.:

1. Dodanie nowej pozycji do portfolio.
2. Wylogowanie użytkownika - sprawdzenie czy został on wylogowany prawidłowo, czy jest możliwe ponowne zalogowanie bez podawania loginu i hasła.
3. Logowanie użytkownika - pobranie pozycji z bazy zdalnej.
4. Testy ekranu podsumowania, wyszukiwania oraz logowania w trybie offline - sprawdzenie czy aplikacja nie ulega awarii (*crash*).

Do testów posłużył emulator w Android Studio - Pixel 2 (Android 9) oraz fizyczne urządzenie mobilne - Motorola moto g7 (Android 10).

## 5. Podsumowanie

### 5.1. Wnioski

W ramach omawianej pracy inżynierskiej powstała aplikacja mająca służyć osobom, które inwestują w akcje rynku amerykańskiego w więcej niż jednym domu maklerskim realizując dzięki temu dywersyfikację.

Rozwiązanie jest natywne dla urządzeń z systemem Android. Powstało z wykorzystaniem najnowszych wytycznych *Google*, takich jak:

1. Architektura *MVVM*
2. Język Kotlin
3. Live Data
4. Retrofit
5. *Material Guidelines* (nie wspomniane w pracy praktyki oraz komponenty do tworzenia przyjaznego użytkownikowi interfejsu)

W ramach pracy zostało zintegrowane API giełdowe Alpha Vantage, które dzięki skutecznemu oddzieleniu modelu od pozostałych warstw aplikacji pozwoliło stosunkowo łatwo zostać zmienione na Finnhub API. Główne funkcje API wykorzystane w pracy to: listowanie dostępnych symboli oraz odczytywanie aktualnej ceny dla pozycji.

Usługa *Backend as a Service* zintegrowana w ramach aplikacji to Back4App korzystająca z otwartego rozwiązania Parse. Zastosowanie nowoczesnych rozwiązań typu *BaaS* przyspiesza rozwój aplikacji mobilnych i nie wymaga dodatkowych członków zespołu odpowiedzialnych za backend. Jest to dobre rozwiązanie dla nieskomplikowanych aplikacji oraz aplikacji, które są na wczesnym etapie rozwoju. Główne funkcje Back4App wykorzystane w pracy to: logowanie i rejestracja użytkownika, zdalna baza danych.

Pierwotne założenia projektowe obejmowały pełne skupienie się na lokalnie działającej aplikacji mobilnej obsługującej wiele giełd oraz rodzajów aktywów. Po spriorytetyzowaniu integracji z usługą typu *BaaS* nie udało się dojść do założonych wcześniej rezultatów. Skutkiem czego jest zmniejszona funkcjonalność użytkowa - obsługa jedynie amerykańskiego rynku akcji. Z kolei zaletą podjętej zmiany projektowej jest działająca implementacja aplikacji mobilnej oraz części serwerowej umożliwiającej dalszy rozwój rozwiązania mogącego obsługiwać wielu użytkowników.

### 5.2. Perspektywy rozwoju

Dalszy rozwój aplikacji ma na celu rozbudować ją, aby mogła stać się pełno-funkcjonalną aplikacją wspomagającą pozyskiwanie informacji o zdywersyfikowanym portfelu użytkownika.

W krótkim oraz średnim horyzoncie czasowym plany obejmują:

1. Obsługę innych giełd niż amerykańska (m.in. GPW, DAX (Niemcy) oraz LSE (Wielka Brytania)).



2. Obsługę wielu walut.
3. Wyszukiwanie i dodawanie aktywów takich jak waluty (Forex) oraz kryptowaluty.
4. Dodawanie pozycji statycznych, np. gotówka.
5. Grupowanie aktywów i układanie z nich planu dywersyfikacji przez użytkownika.

W średnim oraz długim horyzoncie czasowym:

1. Wykresy obrazujące kondycję portfela w czasie.
2. Wyświetlanie wykresów świecowych.
3. Dodanie klienta webowego.
4. Dodanie klienta na system iOS.

Po osiągnięciu gotowości produkcyjnej pod względem funkcjonalności i konkurencyjności niewykluczone jest z punktu widzenia biznesowego nawiązanie współpracy z popularnymi blogami/youtuberami opowiadającym się za wolnością finansową oraz dywersyfikacją majątku, np. *Finanse Bardzo Osobiste (FBO)*.



## Bibliografia

- [1] “Ryzyko”, Dostęp zdalny (26.01.2021): <https://pl.wikipedia.org/wiki/Ryzyko>.
- [2] “Portfel inwestycyjny”, Dostęp zdalny (26.01.2021): <https://www.nntfi.pl/slownik/portfel-inwestycyjny>.
- [3] B. Dubiel, “Standardy zarządzania ryzykiem w jednostkach samorządu terytorialnego”, *Zeszyty Naukowe Uniwersytetu Szczecińskiego. Finanse, Rynki Finansowe, Ubezpieczenia*, nr. 74, s. 481–491, 2015, Dostęp zdalny (21.06.2019): <http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.ekon-element-000171385321>.
- [4] “Kotlin (język programowania)”, Dostęp zdalny (17.01.2021): [https://pl.wikipedia.org/wiki/Kotlin\\_\(j%C4%99zyk\\_programowania\)](https://pl.wikipedia.org/wiki/Kotlin_(j%C4%99zyk_programowania)).
- [5] “2020 Developer Survey”, Dostęp zdalny (17.01.2021): <https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted>.
- [6] *Changelog biblioteki Retrofit*, Dostęp zdalny (17.01.2021): <https://github.com/square/retrofit/blob/master/CHANGELOG.md>, 2021.

## Wykaz symboli i skrótów

**ACL** – ang. *Access Control List*

**BaaS** – ang. *Backend as a Service*

**DTO** – ang. *Data Transfer Object*

**EiTI** – Wydział Elektroniki i Technik Informatycznych

**GPW** – Giełda Papierów Wartościowych w Warszawie

**MVVM** – ang. *Model-View-ViewModel*

**PW** – Politechnika Warszawska

## Spis rysunków

2.1	Strona <i>parseplatform.org</i> przedstawiająca zbiorcze statystyki z portalu <i>github.com</i> . . . . .	14
2.2	Strona <i>back4app.com/compare-all-plans</i> przedstawiająca plany cenowe rozwiązania Back4App . . . . .	14
2.3	Dokumentacja endpointu <i>Candles</i> Finnhub API . . . . .	15
3.1	Rozkład ekranów oraz nawigacja między nimi . . . . .	16
3.2	Architektura rozwiązania . . . . .	17
3.3	Schemat komunikacji z API . . . . .	18
3.4	Żądanie GET do API . . . . .	18
3.5	Przykład odpowiedzi API w formacie JSON . . . . .	18
3.6	Przykładowy <i>DTO</i> . . . . .	19
3.7	Przykładowa odpowiedź API Alpha Vantage oraz odpowiedniki pól w nowym modelu . . . . .	19
3.8	Opis funkcji Parse wraz z przykładem komendy cURL z dokumentacji Back4App	20
3.9	Przykład implementacji funkcji dla systemu Android z dokumentacji Back4App	21
3.10	Kod rejestracji użytkownika . . . . .	22
3.11	Weryfikacja aktualnego użytkownika . . . . .	22
3.12	Tworzenie pustej klasy w konsoli Parse . . . . .	23
3.13	Dokumentacja pustej klasy w Back4App . . . . .	24
3.14	Dodawanie nowej kolumny w Parse . . . . .	24
3.15	Widok stworzonego obiektu w konsoli Parse . . . . .	25
3.16	Uproszczony schemat serwerowej bazy danych stworzony za pomocą <i>dbdiagram.io</i> . . . . .	26
3.17	Schemat bazy danych klienta stworzony za pomocą <i>dbdiagram.io</i> . . . . .	26
3.18	Implementacja coroutine dla <i>viewModelScope</i> dla ekranu wyszukiwania . . .	27
3.19	Implementacja coroutine dla <i>Dispatcher.IO</i> przy operacji na bazie danych . .	28
3.20	Zapytanie SQL zwracające <i>LiveData</i> . . . . .	28
3.21	Inicjalizacja <i>LiveData</i> w <i>viewModel</i> . . . . .	28

3.22 Referencja do <i>LiveData</i> we Fragmentcie . . . . .	29
3.23 Diagram sekwencyjny opisujący mechanizm LiveData zastosowany w aplikacji	29
4.1 <i>Branching model</i> zastosowany w omawianym projekcie. Autorski rysunek z użyciem narzędzia draw.io . . . . .	30