

Spis treści

1	Wstęp.....	3
2	Charakterystyka problemu	6
2.1	Finanse gospodarstw domowych.....	6
2.2	Majątek gospodarstwa domowego.....	7
2.3	Budżet gospodarstwa domowego.....	8
2.4	Rachunek przepływów pieniężnych.....	10
2.5	Zarządzanie majątkiem	11
2.6	Zarządzanie budżetem i przepływami pieniężnymi	13
2.7	Rola planowania w finansach domowych.....	13
2.8	Realizacja i kontrolowanie planu finansowego	14
2.9	Aplikacje wspomagające proces zarządzania finansami domowymi	15
3	Analiza istniejących rozwiązań	16
3.1	Kryteria analizy porównawczej	16
3.2	Aplikacja Money Manager	18
3.3	Aplikacja Easy Budget.....	19
3.4	Aplikacja Wallet.....	20
3.5	Aplikacja Kontomierz.....	21
3.6	Podsumowanie	24
4	Koncepcja własnego rozwiązania	26
4.1	Koncepcja rozwiązania użytkowego.....	26
4.2	Koncepcja rozwiązania technicznego	27
5	Projekt ogólny	32
5.1	Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych.....	32
5.1.1	Wymagania funkcjonalne.....	32
5.1.2	Wymagania niefunkcjonalne.....	33
5.1.3	Diagramy przypadków użycia	34
5.1.4	Scenariusze dla wybranych przypadków użycia	35
5.2	Architektura systemu	37
5.3	Architektura mikroserwisów.....	40
5.4	Biblioteki i narzędzi użyte do budowy aplikacji klienckiej	41
5.5	Biblioteki i narzędzi wykorzystane do budowy mikrousług.....	42
5.6	Koncepcja przechowywania danych	44
5.7	Projekt interfejsu użytkownika	48

6	Implementacja systemu.....	51
6.1	Budowa mikroserwisów	51
6.2	Rozwiązania wykorzystane w mikroserwisach.....	59
6.2.1	Biblioteki reaktywne i programowanie funkcyjne	59
6.2.2	Automatyczne generowanie powtarzalnych składników kodu	60
6.2.3	Obsługa uwierzytelniania i autoryzacji.....	61
6.2.4	Reaktywna obsługa bazy danych MongoDB.....	63
6.2.5	Biblioteka wspólna dla wszystkich mikroserwisów	65
6.3	Budowa aplikacji klienckiej	65
6.4	Rozwiązania wykorzystane w aplikacji klienckiej	70
6.4.1	Reaktywne przetwarzanie danych z wykorzystaniem RxJs	71
6.4.2	Biblioteka stylów Bootstrap oraz Ng Bootstrap	71
6.5	Uwierzytelnianie i autoryzacja.....	72
6.6	Komunikacja pomiędzy komponentami systemu	72
6.7	Interfejs oferowany przez mikroserwisy	75
7	Rozwój i wdrożenie aplikacji	78
7.1	Ciągła integracja i ciągłe dostarczanie.....	78
7.2	Wdrożenie z wykorzystaniem platformy Okteto	78
7.3	Rozwój aplikacji w środowisku lokalnym.....	82
8	Przykładowy scenariusz wykorzystania systemu	83
9	Testy i weryfikacja systemu	88
9.1	Automatyczne testy jednostkowe	88
9.2	Automatyczne testy integracyjne.....	91
10	Zakończenie.....	96
11	Bibliografia.....	99
12	Spis rysunków	101
13	Spis tabel	102

1 Wstęp

Gospodarstwo domowe jest podstawową jednostką gospodarki i najstarszą instytucją ekonomiczną świata. W obliczu przemian gospodarczych oraz społecznych gospodarstwom domowym coraz trudniej jest jednak podejmować racjonalne decyzje ekonomiczne. Nie sprzyja temu ani ciągle zmieniający się asortyment i ceny produktów, ani zaawansowane socjotechniki stosowane przez firmy w celu zwiększenia wyników sprzedaży, często kosztem nieświadomych konsumentów. Nieracjonalne zachowania gospodarstw domowych, w tym nieprzemyślane zaciąganie zobowiązań kredytowych doprowadziły m.in. do kryzysu finansowego w latach 2008 – 2010.

Podejmowanie racjonalnych i korzystnych z punktu widzenia gospodarstw domowych decyzji jest zagadnieniem złożonym, wymagającym szerokiej wiedzy m.in. z zakresu zarządzania majątkiem i kapitałem, zarządzania budżetem i przepływami pieniężnymi, zarządzania oszczędnościami, zarządzania długiem czy też zarządzania ryzykiem. Zarządzanie domowymi finansami wymaga systematycznego podejścia polegającego na planowaniu działalności finansowej oraz jej organizowania, motywowania i kontrolowania. Większość tych działań można usprawnić wykorzystując specjalistyczne oprogramowanie, które niestety w przeciwnieństwie do oprogramowania wykorzystywanego w firmach, nie jest szeroko rozpowszechnione, a dostępne na rynku aplikacje często nie obejmują swoim zakresem wszystkich aspektów związanych z zarządzaniem domowymi finansami.

Celem pracy jest zaprojektowanie i implementacja aplikacji internetowej wspomagającej zarządzanie finansami domowymi. Przy czym implementacja poszczególnych funkcji aplikacji będzie przede wszystkim stanowiła tło dla pokazania możliwości jakie oferują zastosowane technologie. Aplikacja zbudowana będzie z wykorzystaniem architektury opartej o mikrousługi, zapewniającej wysoką skalowalność gwarantującą efektywną pracę przy zróżnicowanym obciążeniu. Aplikacja zbudowana będzie jako usługa (ang. Software as a Service, SaaS) dostępna dla zarejestrowanych użytkowników. W rzeczywistości aplikacja składać się będzie z kilku odrębnych aplikacji w tym w aplikacji napisanej w języku TypeScript z wykorzystaniem środowiska Angular odpowiedzialnej za wyświetlanie interfejsu użytkownika oraz kilku niezależnych

aplikacji serwerowych działających jako mikrousługi, napisanych w języku Java z wykorzystaniem środowiska Spring Boot. Poszczególne elementy systemu komunikować się będą na dwa sposoby – synchronicznie z wykorzystaniem interfejsu REST API oraz asynchronicznie z wykorzystaniem brokera wiadomości RabbitMQ. Implementacja swoim zakresem obejmować będzie podstawowe funkcje zarządzania budżetem i przepływami pieniężnymi. Oprócz tego aplikacja wzbogacona zostanie o funkcje związane z zarządzaniem majątkiem, których najczęściej nie oferują aplikacje obecnie dostępne na rynku. Będą to funkcje pozwalające kontrolować posiadane depozyty bankowe oraz lokaty terminowe. Ponadto modułowa budowa aplikacji umożliwi łatwe jej rozszerzanie o dodatkowe funkcje w przyszłości.

W rozdziale drugim przedstawiono charakterystykę zagadnienia, jakim jest zarządzanie finansami gospodarstwa domowego oraz wskazano potencjalne możliwości zastosowania oprogramowania wspomagającego ten proces.

Rozdział trzeci zawiera analizę istniejących rozwiązań dostępnych na rynku. Wybrane rozwiązania przeanalizowano w szczególności pod kątem pokrycia zagadnień związanych z zarządzaniem majątkiem oraz zarządzaniem budżetem i przepływami pieniężnymi w gospodarstwie domowym, które opisane zostały w rozdziale drugim.

W rozdziale czwartym przedstawiono koncepcję własnego rozwiązania. Rozdział ten opisuje między innymi wymagania użytkowe aplikacji, przedstawia przykładowe scenariusze wykorzystania systemu oraz propozycje rozwiązań technologicznych, w tym wstępna koncepcję architektury projektowanego systemu.

Rozdział piąty zawiera projekt ogólny aplikacji, w tym wymagania funkcjonalne i niefunkcjonalne, diagramy przypadków użycia wraz z wybranymi scenariuszami przypadków użycia, opis architektury systemu i wykorzystanych technologii, opis architektury wykorzystanej do budowy mikrousług, przegląd wykorzystanych bibliotek i narzędzi, projekt modelu danych oraz projekt interfejsu użytkownika.

W rozdziale szóstym opisano szczegóły związane z implementacją poszczególnych aplikacji wchodzących w skład systemu. Rozdział ten zawiera m.in. opisy zastosowanych rozwiązań technicznych wraz z przykładami ich implementacji, a także wybrane fragmenty dokumentacji technicznej w skład której wchodzą m.in. diagramy hierarchii klas, oraz diagramy sekwencji. Rozdział opisuje również zastosowany sposób uwierzytelniania i autoryzacji użytkowników

z wykorzystaniem protokołu OAuth 2.0 / OpenID Connect. W ramach rozdziału szóstego opisano również sposób komunikacji pomiędzy poszczególnymi elementami systemu, w tym wykorzystanie interfejsu REST API oraz brokera wiadomości RabbitMQ. W ostatnim podrozdziale rozdziału szóstego przedstawiono natomiast dokumentacje punktów końcowych interfejsu REST API, w tym sposób jej przygotowania z wykorzystaniem bibliotek do automatycznego generowania dokumentacji w formacie OpenAPI.

Rozdział ósmy przedstawia przykłady wykorzystania systemu ilustrowane rzutami poszczególnych ekranów aplikacji, odpowiadające scenariuszom przypadków użycia określonych w rozdziale piątym.

W rozdziale dziewiątym przedstawiono sposoby wykorzystane do testowania i weryfikacji poprawności działania systemu, w tym opis zaimplementowanych automatycznych testów jednostkowych oraz automatycznych testów integracyjnych napisanych w języku Groovy z wykorzystaniem bibliotek Spock oraz podejścia opartego na testach sterowanych danymi (ang. Data-Driven Testing).

W ostatnim dziesiątym rozdziale przedstawiono podsumowanie niniejszej pracy dyplomowej.

2 Charakterystyka problemu

Zaprojektowanie i zbudowanie aplikacji, której celem jest efektywne wspomaganie prowadzenia domowych finansów wymaga dobrej identyfikacji wszystkich wymagań funkcjonalnych. W tym celu w niniejszym rozdziale dokonano analizy wszystkich aspektów związanych z zarządzaniem domowymi finansami, w tym aspektów związanych z zarządzaniem majątkiem oraz zarządzaniem budżetem domowym.

Zagadnienia przedstawione w niniejszym rozdziale, w większości nieznane wcześniej autorowi, zostały opracowane na podstawie materiałów źródłowych [1] [2] [3] [4] [5].

2.1 Finanse gospodarstw domowych

Tematyka gospodarstw domowych w przeciwnieństwie np. do tematyki związanej z przedsiębiorstwami wydaje się być nieco zaniedbywana zarówno w dyscyplinach takich jak finanse czy rachunkowość, jak i w szeroko pojętej przestrzeni publicznej. Problemów można się doszukiwać już na etapie edukacji na poziomie szkoły podstawowej i średniej, gdzie liczba godzin nauczania przedmiotów związanych z finansami i przedsiębiorczością jest symboliczna.

Wydarzenia ostatnich lat udowodniły, że w zmieniającym się świecie, w którym ilość produktów, ich zmieniających się cen oraz specyfikacji jest przytaczająca, a agresywna, wykorzystująca często nieetyczne socjotechniki reklama wszechobecna, podejmowanie racjonalnych decyzji jest wyjątkowo trudne. Kryzys w latach 2008-2010 u swoich źródeł ma właśnie takie nieracjonalne, wynikające z nieświadomości decyzje konsumentów, które doprowadziły setki tysięcy gospodarstw domowych do niewypłacalności. Należy przy tym zauważyć, że instytucje finansowe, które w dużej mierze odpowiedzialne są za taką sytuację, w sposób wręcz cynyczny, wykorzystywały niewiedzę i naiwność ludzi.

Dostępne sondaże wskazują, że ok. 80% gospodarstw domowych w Polsce nie planuje i nie kontroluje swoich finansów, zarówno wydatków bieżących jak i inwestycji [1]. W takiej sytuacji konieczność propagowania wiedzy, pozwalającej na rozsądne gospodarowanie budżetem i majątkiem gospodarstw domowych wydaje się szczególnie istotna, tym bardziej że zachowania gospodarstw domowych mają oczywisty wpływ na kondycję całej gospodarki. Sposobem

na odciążenie gospodarstw domowych z konieczności ręcznego prowadzenia domowych rachunków, a także po części zwolnieniem z konieczności posiadania rozległej wiedzy w niektórych obszarach, mogą być nowoczesne aplikacje komputerowe. Aplikacje takie mogą wspomagać nie tylko podstawowe czynności związane z zarządzaniem przepływami pieniężnymi, ale mogą być także pomocne w zarządzaniu majątkiem w szerszym kontekście. Bardziej zaawansowane aplikacje mogą być również pomocne w podejmowaniu decyzji finansowych związanych z inwestycjami czy też zaciąganiem zobowiązań kredytowych.

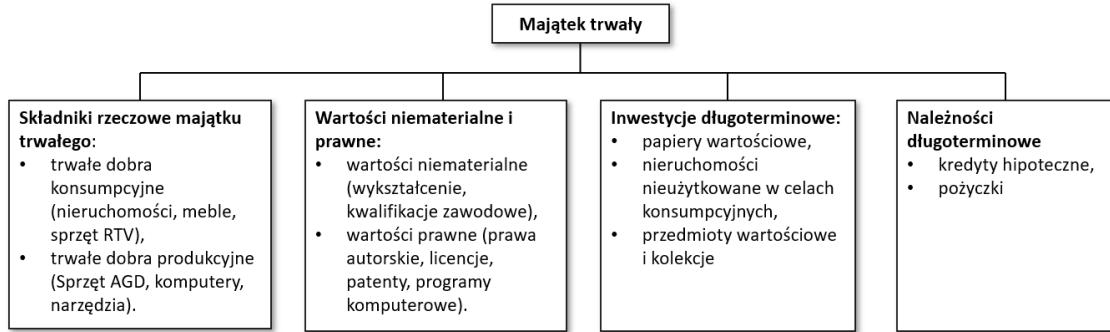
2.2 Majątek gospodarstwa domowego

Istnieje wiele podobieństw pomiędzy gospodarstwem domowym a przedsiębiorstwem, które chcąc sprawnie funkcjonować, jak każdy inny podmiot gospodarczy musi dysponować odpowiednim majątkiem. Podobnie jak w przypadku przedsiębiorstw, istotne będzie więc ustalenie wielkości struktury, a także efektywności majątku. Umożliwia to analiza zasobów majątkowych której celem jest:

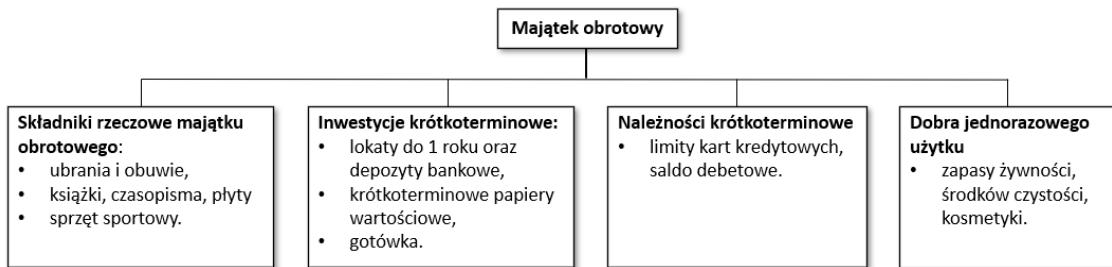
- ustalenie wielkości majątku oraz jego struktury i pochodzenia,
- monitorowanie zmian majątku w czasie
- monitorowanie efektywności wykorzystania majątku w czasie
- ocena efektywności wykorzystania majątku oraz wnioski określające sposoby kształtowania majątku w przeszłości.

Podobnie jak w przypadku przedsiębiorstw, do ustalenia aktualnego stanu majątku gospodarstwa domowego wykorzystany może zostać bilans majątkowy. Bilans taki może być bardzo pomocny w ocenie skuteczności prowadzenia gospodarstwa domowego, można np. ustalić zmiany wartości majątku w określonym czasie.

Majątek gospodarstwa domowego (nazywany również aktywami) podobnie jak w przypadku przedsiębiorstwa możemy podzielić na majątek trwały oraz majątek obrotowy. Majątek trwały skupia głównie majątek użytkowany przez dłuższy czas, najczęściej powyżej roku. Składniki majątku trwałego prezentuje Rysunek 1. Majątek obrotowy z kolei zawiera wszystkie pozostałe składniki majątku które nie weszły w skład majątku trwałego. Są to głównie dobra które ulegają szybkiemu zużyciu lub zbyciu, zwykle przed upływem jednego roku. Składniki majątku trwałego prezentuje Rysunek 2.



Rysunek 1 Składniki majątku trwałego gospodarstwa domowego [1] [5]



Rysunek 2 Składniki majątku obrotowego gospodarstwa domowego [1] [5]

Drugą stronę bilansu majątku gospodarstwa stanowią pasywa, czyli kapitał. Kapitał możemy podzielić na dwie części, tj. kapitał własny oraz kapitał obcy. Na kapitał własny składa się kapitał rodzinny (czyli kapitał pozyskany od innych członków rodziny np. w drodze spadków czy darowizn), a także własne zasoby finansowe. Z kolei kapitał obcy stanowią wszystkie zobowiązania krótko i długoterminowe, takie jak kredyty i pożyczki.

2.3 Budżet gospodarstwa domowego

W przypadku przedsiębiorstw kolejnym źródłem informacji finansowych jest rachunek przychodów i rozchodów. Jego odpowiednikiem w przypadku gospodarstw domowych jest budżet gospodarstwa domowego, który zgodnie z klasyfikacją Głównego Urzędu Statystycznego można traktować jako usystematyzowane zestawienie pieniężnych i niepieniężnych przychodów oraz rozchodów w określonym przedziale czasu [1].

Zgodnie z klasyfikacją GUS wyróżnia się następujące składniki gospodarstwa domowego:

- Przychody - są to wszystkie wartości wpływające do gospodarstwa domowego.

- Rozchody – są to wszystkie wartości przekazywane przez gospodarstwo domowe na zewnątrz [2].

W praktyce w budżecie domowym często wykorzystuje się przychody i rozchody netto, które m.in. nie uwzględniają zaliczek na podatek dochodowy od osób fizycznych, czy też składek na ubezpieczenie społeczne i zdrowotne [2]. Składniki budżetu gospodarstwa domowego netto prezentuje Tabela 1.

Tabela 1 Zestawienie budżetu gospodarstwa domowego bez uwzględnienia podatków i zaliczek na ubezpieczenie społeczne [1]

Przychody netto	Rozchody netto
<p>Dochody rozporządzalne:</p> <ul style="list-style-type: none"> • dochody z pracy najemnej; • dochody z indywidualnego gospodarstwa rolnego; • dochody z pracy na własny rachunek; • dochody z tytułu własności (odsetki, udziały w zyskach, dochód z wynajmu); • dochody ze świadczeń ubezpieczeniowych (emerytury, renty, świadczenia rehabilitacyjne, odszkodowania); • dochody ze świadczeń pozostałych (świadczenie z budżetu państwa, świadczenia z funduszy specjalnych, świadczenia w naturze od instytucji niekomercyjnych); • pozostałe dochody (odszkodowania, dary i alimenty, wygrane w grach hazardowych, inne nieuwzględnione wcześniej dochody). <p>Pozycje oszczędnościowe po stronie dochodowej:</p> <ul style="list-style-type: none"> • gotówka z poprzednich okresów, pobrane lokaty, wpływy ze sprzedaży papierów wartościowych; • pożyczki i kredyty; • wpływy ze sprzedaży majątku trwałego, • wpływy ze sprzedaży artykułów konsumpcyjnych; • wpływy ze sprzedaży zwierząt i roślin. 	<p>Wydatki:</p> <ul style="list-style-type: none"> • wydatki na towary i usługi konsumpcyjne (wydatki na żywność i napoje, odzież i obuwie, użytkowanie mieszkania i nośniki energii, wyposażenie mieszkania, zdrowie, transport, łączność, rekreację i kulturę, edukację, restauracje i hotele oraz inne towary i usługi); • pozostałe wydatki (dary przekazane innym, niektóre podatki w tym podatki od spadków i darowizn, zaliczki na podatki od dochodów oraz składki na ubezpieczenie społeczne samodzielnie opłacane przez podatnika, inne wydatki nieprzeznaczone bezpośrednio na cele konsumpcyjne); • pozycje oszczędnościowe po stronie rozchodów; • założone lokaty i wydatki na zakup papierów wartościowych oraz odłożona gotówka pieniężna; • spłacone pożyczki i kredyty; • pożyczki udzielone innym gospodarstwom domowym; • składki ubezpieczeniowe na życie • zakup, remont i naprawa majątku rzeczowego gospodarstwa domowego.

2.4 Rachunek przepływów pieniężnych

Rachunek przepływów pieniężnych przedstawia pieniężne przepływy wpływające do gospodarstwa domowego oraz z niego wypływające, a jego końcowym rezultatem są przepływy pieniężne netto, stanowiące różnice między wpływami a wydatkami pieniężnymi. Rachunek przepływów pieniężnych pozwala na ocenę płynności finansowej gospodarstwa domowego.

Przepływy pieniężne można zasadniczo podzielić na trzy rodzaje [1]:

1. Przepływy środków pieniężnych związanych z konsumpcją.
 - 1.1. Wpływy.
 - 1.1.1. Wynagrodzenia z tytułu pracy najemnej.
 - 1.1.2. Dochody z prowadzonej działalności gospodarczej.
 - 1.1.3. Świadczenia z tytułu ubezpieczeń społecznych i opieki społecznej.
 - 1.1.4. Inne wpływy z tytułu pracy oraz z działalności gospodarczej.
 - 1.2. Wydatki.
 - 1.2.1. Zakup dóbr konsumpcyjnych (dóbr materialnych i usług).
 - 1.2.2. Wydatki związane z prowadzeniem działalności gospodarczej.
 - 1.2.3. Wynagrodzenia osób wykonujących usługi na rzecz gospodarstwa domowego.
 - 1.2.4. Ubezpieczenia społeczne i zdrowotne oraz ubezpieczenia.
 - 1.2.5. Podatki i opłaty o charakterze publicznoprawnym.
 - 1.2.6. Inne wydatki konsumpcyjne.
 - 1.3. Przepływy pieniężne netto z działalności bezpośrednio podporządkowanej realizacji funkcji konsumpcyjnej (1.1 – 1.2).
2. Przepływy środków pieniężnych z działalności inwestycyjnej.
 - 2.1. Wpływy.
 - 2.1.1. Zbycie rzeczowych aktywów trwałych oraz wartości niematerialnych i prawnych.
 - 2.1.2. Zbycie inwestycji w nieruchomości oraz wartości niematerialne i prawne.
 - 2.1.3. Z aktywów finansowych, w tym szczególnie:
 - dywidendy i udziały w zyskach,
 - spłata (zwrot) udzielonych pożyczek długoterminowych,
 - uzyskane odsetki od udzielonych pożyczek długoterminowych
 - inne wpływy z aktywów finansowych.
 - 2.1.4. Inne wpływy inwestycyjne.

2.2. Wydatki.

- 2.2.1. Nabycie rzeczowych aktywów trwałych oraz wartości niematerialnych i prawnych.
- 2.2.2. Inwestycje w nieruchomości oraz wartości materialne i prawne.
- 2.2.3. Wydatki na aktywa finansowe, w tym szczególnie:
 - nabycie aktywów finansowych,
 - udzielone pożyczki długoterminowe.
- 2.2.4. Inne wydatki inwestycyjne.

2.3. Przepływy pieniężne netto z działalnością inwestycyjną (2.1 - 2.2).

3. Przepływy środków pieniężnych związanych z działalnością finansową:

3.1. Wpływy.

- 3.1.1. Nabycie kredytów i pożyczek.
- 3.1.2. Pozyskane spadki i darowizny finansowe.
- 3.1.3. Inne wpływy finansowe.

3.2. Wydatki.

- 3.2.1. Spłaty kredytów i pożyczek.
- 3.2.2. Wypłacone odsetki od kredytów i pożyczek.
- 3.2.3. Przekazane spadki i darowizny finansowe.
- 3.2.4. Inne wydatki finansowe.

3.3. Przepływy pieniężne netto z działalności finansowej (3.1 - 3.2).

W rachunku przepływów, można uwzględnić wyżej wymienione przepływy osobno lub razem. Ponadto rachunek może obejmować środki pieniężne na początku oraz na końcu okresu sprawozdawczego.

2.5 Zarządzanie majątkiem

Najważniejszą częścią majątku gospodarstwa domowego jest majątek rzeczowy. Majątek rzeczowy, może ulegać zużyciu, w związku z tym zmieniać się będzie jego wartość w czasie. Gospodarstwa domowe jednak nie prowadzą z reguły ewidencji zmian wartości majątku trwałego, zdarza się jednak, że konieczne jest określenie bieżącej wartości określonego składnika majątkowego. Przy określaniu wartości majątku trwałego bądź jego składników warto przy tym wzorować się na rachunku amortyzacyjnym stosowanym w przedsiębiorstwach. Ponadto warto zwrócić również uwagę na zmiany wartości składników majątku wynikające z inflacji, przy czym ważne tutaj będzie przyjęcie

określonego rodzaju wskaźników inflacji. Może to być ogólny indeks cen konsumpcyjnych (ang. Customer Price Index) lub indeks wzrostu cen dla konkretnej grupy dóbr trwałych [1].

Oprócz majątku trwałego, podczas ustalania wartości majątku nie można pominąć pozostałych elementów majątku (opisanych w niniejszym rozdziale powyżej) w tym inwestycji i należności długoterminowych, które są nabywane najczęściej w celu osiągnięcia korzyści ekonomicznych w przyszłości, a także wartości niematerialnych i prawnych.

Kolejnym trudnym w praktycznej realizacji zagadnieniem jest zarządzanie majątkiem obrotowym, a w szczególności wysoko rotującymi nietrwałymi dobrami materialnymi, takimi jak odzież, obuwie, chemia domowa, kosmetyki czy artykuły spożywcze. Struktura majątku obrotowego będzie najczęściej zależeć od skłonności konsumpcyjnych gospodarstwa domowego. W tym obszarze istnieje zwykle ogromne pole do optymalizacji, której efektem mogą być znaczne oszczędności.

Istotnym składnikiem szeroko rozumianych zasobów finansowych są również opisane w niniejszym rozdziale pasywa, czyli kapitał gospodarstwa domowego, na które składają się kapitał własny i kapitał obcy. Szczególnie istotne okazują się proporcje pomiędzy tymi dwoma rodzajami kapitału. Podjęcie decyzji o finansowaniu inwestycji lub zakupu składników majątku trwałego czy obrotowego będzie mieć z reguły kluczowe znaczenie. Nie można bowiem jednoznacznie stwierdzić, że finansowanie tego rodzaju zakupów ze środków obcych, będzie zawsze mniej korzystne, niż wykorzystanie do tego celu środków własnych. Okazuje się, że zaciągnięcie kredytu np. na zakup mieszkania może być bardziej opłacalne (również ze względów poza materialnych) niż np. odroczenie zakupu o kilka lat i sfinansowanie go ze środków własnych. Kształtowanie proporcji pomiędzy kapitałem własnym, a kapitałem obcym, będzie miało więc charakter decyzji strategicznych, które powinny być odpowiednio zaplanowane oraz przeanalizowane pod kątem ryzyka finansowego. Podczas takiej analizy należy wziąć pod uwagę wszystkie koszty związane z kredytami. Poza samym oprocentowaniem istotnym czynnikiem kosztotwórczym związany z oferowanymi na rynku kredytami są m.in. prowizje od udzielanych kredytów, prolongaty w spłacie zadłużenia, opłaty za zmianę waluty kredytu czy opłaty pobierane za wydawanie różnego rodzaju zaświadczeń oraz potwierdzeń związanych z zadłużeniem.

Ustalenie wartości majątku i kapitału w całości oraz ich składników, mimo iż rzadko praktykowane przez gospodarstwa domowe, mogło by być bardzo pomocne w zarządzaniu finansami gospodarstwa domowego, a w szczególności w planowaniu i zarządzaniu ryzykiem. Zastosowanie odpowiednich metod pozwala szacować również z pewnym przybliżeniem wysokość majątku w przyszłości, uwzględniając jego zużycie i prognozowaną inflację.

2.6 Zarządzanie budżetem i przepływami pieniężnymi

Zarządzanie budżetem w największym uproszczeniu sprowadza się do planowania przychodów oraz wydatków, prowadzenia rachunków (bilansu przychodów i wydatków), oraz monitorowania bieżących przychodów i wydatków tak aby uzyskać nadwyżkę finansową.

Aby gospodarstwo domowe mogło prawidłowo funkcjonować konieczne jest osiąganie takiej nadwyżki finansowej, którą można rozważyć w określonym okresie (miesiąca, kwartału czy roku). Można założyć, że w pewnych okresach nadwyżka finansowa nie wystąpi, lub też będzie ona miała ujemną wartość (np. w okresie zwiększych wydatków). Z ujemną nadwyżką finansową wiąże się ryzyko utraty płynności finansowej, które z kolei może spowodować powstanie długów oraz konieczność rezygnacji z zaspokojenia części potrzeb gospodarstwa domowego.

Planowanie budżetu domowego powinno ponadto uwzględniać nie tyle dochody nominalne, co dochody realne, tj. powinno uwzględnić ewentualne zmiany siły nabywczej gospodarstwa domowego spowodowane inflacją, nie tylko w odniesieniu do dochodów bieżących, ale przede wszystkim w odniesieniu do dochodów w przyszłości.

2.7 Rola planowania w finansach domowych

Osiągnięcie zamierzonych celów w jakimkolwiek obszarze, a w szczególności w obszarze finansowym, bez odpowiedniego planowania było by bardzo utrudnione. W praktyce związanej z zarządzaniem procesami czy projektami planowanie wymaga zwykle złożonych analiz, najczęściej jest też wieloetapowe. Istnieje również wiele narzędzi i metodyk wspomagających ten proces.

Proces planowania finansów w gospodarstwie domowym, pozwala na podejmowanie trafniejszych decyzji mających realne przełożenie na budżet domowy oraz w dłuższym okresie na majątek gospodarstwa domowego. Niestety proces ten najprawdopodobniej przez swoją złożoność, jest przez większość gospodarstw domowych zaniedbywany.

Przedmiotem planowania finansowego w gospodarstwie domowym powinno być w szczególności:

- planowanie dochodów, ich wielkości i źródeł,
- planowanie dopływu obcych środków finansowych, takich jak pożyczki czy kredyty,
- planowanie inwestycji gospodarstwa domowego,
- planowanie wydatków bieżących,
- planowanie regulacji zobowiązań z tytułu zaciągniętych pożyczek i kredytów,
- planowanie gospodarowania nadwyżką finansową oraz oszczędnościami [1].

Rzeczywistość po pandemii koronawirusa oraz sytuacja toczącej się wojny w Ukrainie, przypomniała także jak ważne jest odpowiednie zarządzanie ryzykiem, również w przypadku podmiotów takich jak gospodarstwa domowe. Niestabilność gospodarcza, rosnąca inflacja, zmieniające się stopy procentowe, to tylko kilka z niewielu czynników, które należy brać pod uwagę podczas analizy ryzyka finansowego, które powinno być uwzględniane, przed podjęciem każdej większej decyzji finansowej.

2.8 Realizacja i kontrolowanie planu finansowego

Osiągnięcie zamierzonych celów zależy najczęściej od wszystkich członków gospodarstwa domowego, a w szczególności ich aktywności zawodowej oraz konsumpcyjnej. Problemem okazać się może również sprzeczność celów poszczególnych członków gospodarstwa domowego oraz ich zmienność w czasie, powodująca konieczność wprowadzania korekt w planie finansowym. Co istotne, kluczowa w osiągnięciu zamierzonego celu okazuje się motywacja członków gospodarstwa domowego. Pomocny może być między innymi wybór odpowiednich bodźców motywacyjnych. Bodźce te mogą być pozytywne (zachęcające) bądź negatywne (zniechęcające). Ich skuteczność zależy od konkretnej sytuacji.

Ostatnim elementem procesu zarządzania jest kontrola wyników osiągniętego celu. Na proces kontroli składają się:

- identyfikacja celu sformułowanego w planie działalności,
- ustalenie stanu rzeczywistego,
- porównanie stanu planowanego ze stanem rzeczywistym oraz ustalenie niezgodności,
- wyjaśnienie przyczyn niezgodności,
- wnioski i zalecenia dla zarządzania w przyszłości wynikające z analizy zadania [1].

Kontrola wyników może odbywać się po zakończeniu danej działalności, ale może być także prowadzona na bieżąco, o ile możliwe w danym momencie jest ustalenie zarówno stanu pożdanego jak i stanu faktycznego. Monitorowanie bieżące działalności pozwoli zredukować ryzyko, ponieważ umożliwia wprowadzenie korekty natychmiast po wykryciu nadmiernego odchylenia od stanu pożdanego.

2.9 Aplikacje wspomagające proces zarządzania finansami domowymi

Opisane w niniejszym rozdziale procesy związane z zarządzaniem szeroko rozumianymi finansami domowymi, w praktyce okazują się zagadnieniami bardzo złożonymi. Podejmowanie trafnych decyzji finansowych wymaga prawidłowego zarządzania swoim majątkiem oraz bieżącym budżetem domowym. Wymaga to najczęściej prowadzenia odpowiednich ewidencji środków trwałych gospodarstwa domowego z uwzględnieniem zmian ich wartości w czasie, wymaga również prowadzenia ksiąg rachunkowych często niewiele mniej skomplikowanych niż księgi rachunkowe niewielkiej firmy. Żmudne prowadzenie ksiąg, wpisywanie ręczne każdej pozycji, jest nie tylko czasochłonne, ale wymaga odpowiedniej wiedzy oraz skrupulatności, w celu uniknięcia błędów.

Znakomita większość tych procesów może być zautomatyzowana z wykorzystaniem nowoczesnych aplikacji, w szczególności aplikacji internetowych. Aplikacje takie mogą być wykorzystane nie tylko do prowadzenia obliczeń rachunkowych, ale mogą same importować dane o transakcjach (np. bezpośrednio z serwisów bankowych) oraz agregować te dane w sposób automatyczny, eliminując przy tym błędy ludzkie.

3 Analiza istniejących rozwiązań

W niniejszym rozdziale przedstawiono analizę istniejących, podobnych aplikacji internetowych służących do wspomagania procesów zarządzania domowymi finansami. W tym celu spośród dostępnych na rynku aplikacji, wybrano kilka aplikacji o zróżnicowanych możliwościach. Oprogramowanie to zostanie poddane analizie porównawczej, uwzględniającej wybrane kryteria.

3.1 Kryteria analizy porównawczej

Z uwagi na komercyjny charakter ocenianych aplikacji, a co za tym idzie brak dostępności do kodu źródłowego oraz ograniczone informacje o architekturze poszczególnych aplikacji, skupiono się głównie na ocenie właściwości funkcjonalnych analizowanych aplikacji.

Z analizy przedstawionej w rozdziale 2 wynika, że wspomaganie samej działalności związanej z prowadzeniem rachunków, może być niewystarczające. Gospodarstwa domowe wykazują często potrzebę większego wsparcia w procesie podejmowania decyzji mających przełożenie na ich sytuację finansową. Decyzje takie wymagają często wieloczynnikowej złożonej analizy. Dla potrzeb analizy porównawczej funkcje ocenianych aplikacji podzielono na dwie grupy:

- funkcje związane z zarządzaniem majątkiem gospodarstwa domowego,
- funkcje związane z zarządzaniem budżetem i przepływami pieniężnymi.

Funkcje związane z zarządzaniem majątkiem gospodarstwa domowego, zgodnie z analizą przedstawioną w rozdziale 2, powinny umożliwiać kompleksowe zarządzanie aktywami gospodarstwa domowego, a w szczególności:

- zarządzanie majątkiem rzecznym,
- zarządzanie inwestycjami długoterminowymi,
- zarządzanie należnościami długoterminowymi,
- zarządzanie obrotowym majątkiem rzecznym,
- zarządzanie zobowiązaniami krótkoterminowymi takimi jak salda debetowe czy limity kart kredytowych.

Realizacja funkcji wspomagających zarządzanie wymienionymi wyżej składnikami majątku powinna uwzględniać takie czynności, jak planowanie, monitorowanie oraz prowadzenie operacji finansowych dotyczących poszczególnych składników majątku.

Funkcje związane z zarządzaniem budżetem domowym oraz przepływami pieniężnymi, to funkcje wspomagające prowadzenie domowych rachunków które powinny umożliwiać m.in.:

- planowanie bieżącego budżetu,
- tworzenie bilansów dochodów i wydatków,
- obsługę wydatków regularnych w tym przypomnienia o zaplanowanych wydatkach,
- przeglądanie historii dochodów i wydatków,
- tworzenie wizualizacji dochodów i wydatków w postaci wykresów graficznych łatwo zrozumiałych dla odbiorców,
- obsługę wielu kont,
- podział zgromadzonych zasobów pieniężnych na zasoby na kontach bankowych oraz zasoby gotówkowe.

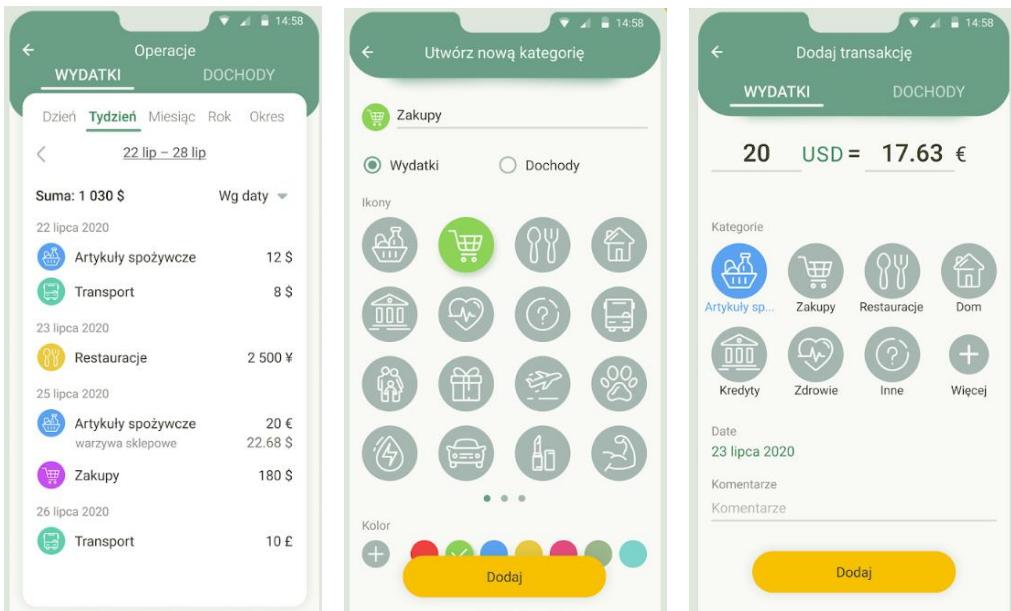
Z uwagi na złożoność tematu, który zarysowano bardziej szczegółowo w rozdziale 2, dla celów analizy porównawczej wyżej wymienione zagadnienia związane z zarządzaniem budżetem oraz zarządzaniem majątkiem i przepływami pieniężnymi przedstawiono w uproszczonej formie. Szczegółowe rozwiązania zastosowane w aplikacjach przedstawiono w opisach poszczególnych systemów.

Oprócz wyżej wymienionych merytorycznych kryteriów funkcjonalnych rozwiązania zostaną ocenione również pod względem wsparcia funkcji użytkowych takich jak:

- możliwość integracji z systemami bankowości elektronicznej i systemami płatności,
- dostępność aplikacji w wersji online (aplikacja sieciowa) lub możliwości przechowywania danych w chmurze,
- dostępność aplikacji mobilnej,
- możliwość importu danych o transakcjach bankowych z plików tekstowych (np. CSV, TSV),
- możliwość eksportu danych do pliku (CSV, TSV, EXCEL).
- możliwość importu płatności ze zdjęć faktur i paragonów (pliki graficzne i pdf)
- możliwość obsługi wielu walut,
- obsługę automatycznych przypomnień o zaległych płatnościach,
- podział na gotówkę oraz konta bankowe,
- obsługę wielu kont bankowych.

3.2 Aplikacja Money Manager

Jedną z bardziej popularnych aplikacji jest darmowa aplikacja *Money Manager*, dostępna również pod nazwą *Finanse, przychody i wydatki, planowanie budżetu* wydana przez firmę Innim Mobile Exp. Jest to aplikacja mobilna dostępna dla systemu Android oraz Apple iOS [6].



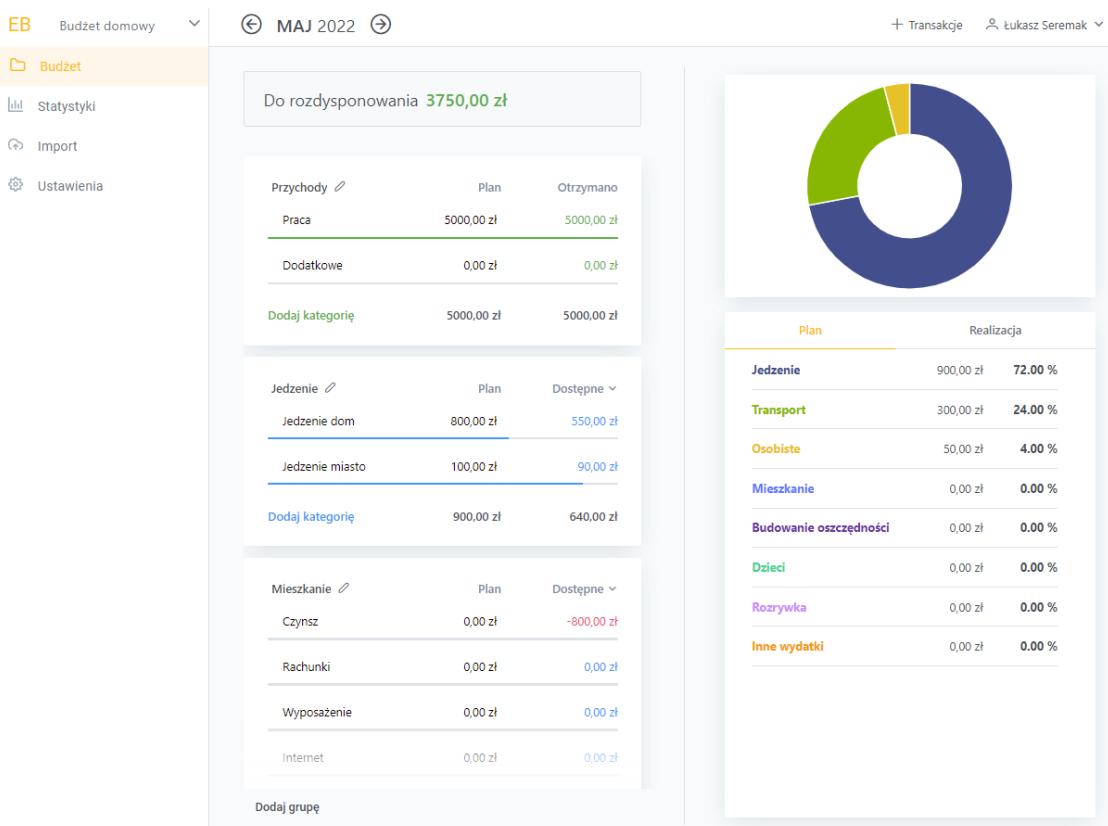
Rysunek 3 Widok ekranu wydatków Money Manager [6]

Aplikacja cechuje się prostym przejrzystym interfejsem użytkownika, ale również stosunkowo ograniczonymi możliwościami. Przykładowe ekrany aplikacji prezentuje Rysunek 3. Aplikacja umożliwia zarządzanie budżetem i przepływami pieniężnymi w podstawowym zakresie, umożliwia ręczne dodawanie dochodów oraz wydatków z podziałem na kategorie. Obsługuje także wiele walut. Pozwala przy tym także na dodawanie regularnych wydatków oraz ustawiania przypomnień. Niestety te dwie funkcje nie są ze sobą zsynchonizowane, a zatem przypomnienie o regularnej płatności musi być zdefiniowane osobno. Aplikacja oferuje także łatwy w obsłudze mechanizm filtrowania i wyświetlania dochodów i wydatków z wybranego okresu. Dane z aplikacji mogą być wyeksportowane do formatu Excel. Niestety aplikacja nie posiada funkcji pozwalających na zarządzanie oszczędnościami ani długami, nie pozwala także zdefiniować celów finansowych. Aplikacja umożliwia założenie konta, dzięki czemu dane mogą być przechowywane zarówno lokalnie na urządzeniu mobilnym oraz na serwerze firmy Innim Mobile Exp [6].

3.3 Aplikacja Easy Budget

Easy Budget z kolei jest aplikacją działającą wyłącznie w przeglądarce. Producenci dołożyli jednak starań, aby aplikacja wyświetlała się poprawnie również na urządzeniach mobilnych. Widok przykładowego ekranu aplikacji Easy Budget prezentuje Rysunek 7.

Aplikacja oferuje nowoczesny oraz prosty i przejrzysty interfejs, a jej funkcjonalność skupia się podobnie jak w przypadku aplikacji Money Manager wyłącznie na aspektach związanych z zarządzaniem domowym budżetem i przepływami pieniężnymi. Aplikacja nie zapewnia automatycznej integracji z systemami bankowymi, ale posiada możliwość importu danych z plików CSV. Obsługuje przy tym kilka formatów udostępnianych przez wybrane banki działające na terenie Polski. Aplikacja umożliwia również ręczne dodawanie transakcji (zarówno przychodów jak i wydatków).



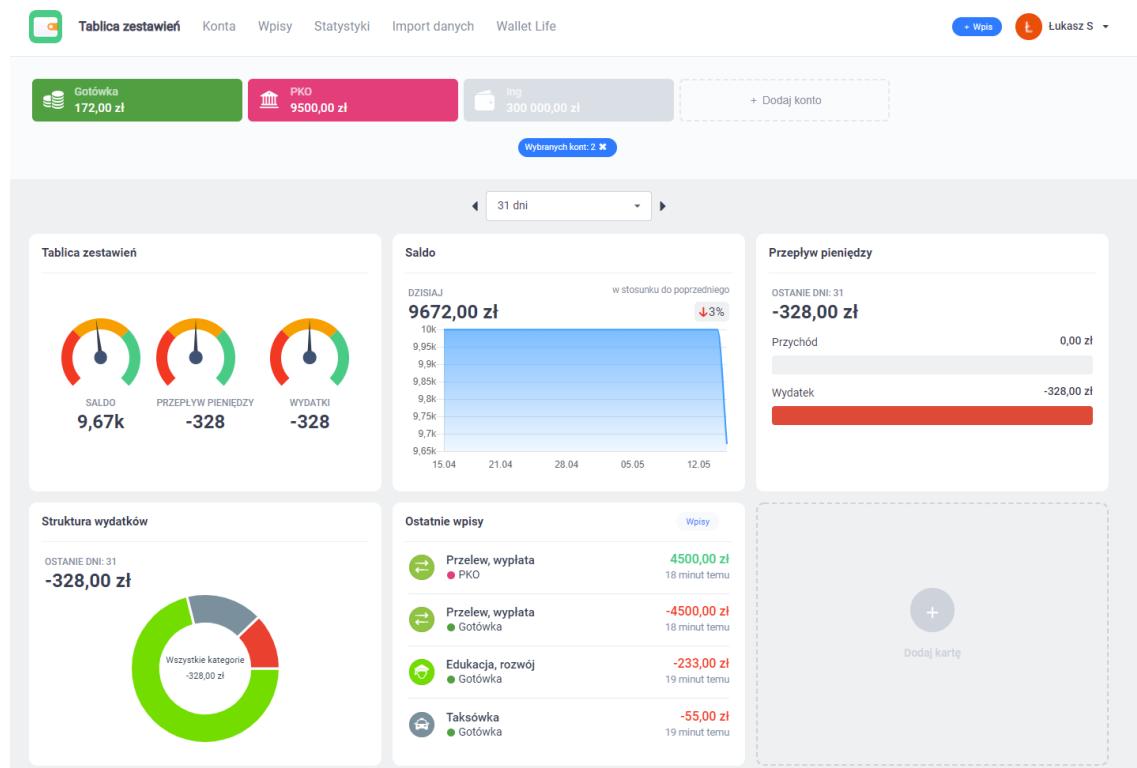
Rysunek 4 Przykładowy ekran aplikacji internetowej Easy Budget [7]

Oprócz tego aplikacja wspomaga planowanie wydatków oraz ich monitorowanie. Umożliwia przypisywanie wydatków do predefiniowanych kategorii, a także tworzenie swoich własnych kategorii. Interfejs użytkownika jest przyjazny

i pozwala na wygodne korzystanie z aplikacji. Aplikacja pozwala także na wizualizacje danych w postaci łatwo zrozumiałych dla użytkownika wykresów [7].

3.4 Aplikacja Wallet

Inną popularną, a także stosunkowo łatwą w obsłudze aplikacją jest aplikacja Wallet firmy Budget Bakers. Aplikacja posiada zarówno wersję działającą w przeglądarce internetowej, jak i wersję mobilną na urządzenia z systemem Android oraz Apple iOS. Przykładowy ekran aplikacji Wallet prezentuje Rysunek 5.



Rysunek 5 Ekran tablicy zestawień aplikacji Wallet firmy BudgetBakers [8]

Aplikacja oferuje nieco większe możliwości niż poprzednio opisane aplikacje Money Manager oraz Easy Budget. Oprócz funkcji związanych z obsługą budżetu, przepływów pieniężnych, kategoryzowania transakcji oraz planowania przyszłych wydatków, aplikacja posiada także podstawowe funkcje pozwalające zarządzać należnościami takimi jak pożyczki czy kredyty. Ponadto, aplikacja pozwala na definiowanie celów oszczędnościowych oraz monitorowanie ich realizacji. Dużą zaletą aplikacji jest funkcja pozwalająca na automatyczne pobieranie danych po

przez integrację z systemami bankowości elektronicznej. Oprócz tego aplikacja umożliwia import danych z plików CSV oraz XLS. Aplikacja obsługuje wiele kont oraz wiele walut, pozwala także na tworzenie portfeli gotówkowych.

Należy przy tym podkreślić, że aplikacja dostępna z poziomu przeglądarki internetowej ma nieco ograniczone możliwości, a część funkcji dostępna jest tylko w aplikacji mobilnej. Niemniej jednak połączenie możliwości dostępu do danych zarówno po przez przeglądarkę internetową, jak i po przez aplikację mobilną z widokami zoptymalizowanymi pod urządzenia mobilne, jest rozwiązaniem wygodnym dla użytkowników [8].

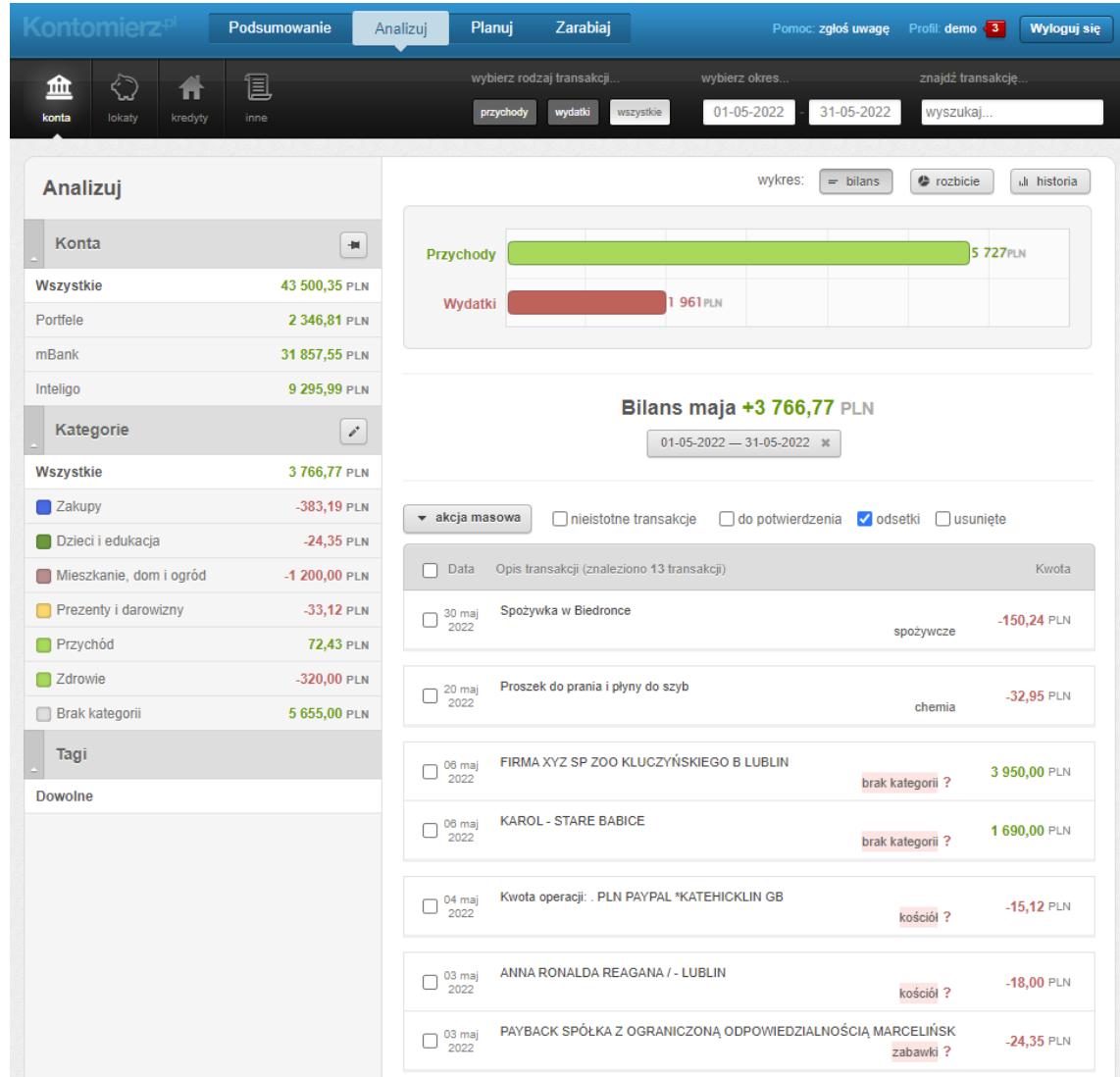
3.5 Aplikacja Kontomierz

Nieco bardziej zaawansowanym systemem niż te opisane poprzednio jest wydana przez polską firmę Fineld Sp. z o.o aplikacja o nazwie Kontomierz¹. Aplikacja ta dostępna jest nie tylko w wersji na urządzenia mobilne, ale posiada także wersję działającą w przeglądarce internetowej [9].

Niestety w parze z funkcjonalnością nie idzie przejrzystość interfejsu, przez co część funkcji może być (przynajmniej początkowo) nie zrozumiała dla użytkowników, podejmujących decyzję o wyborze aplikacji wspomagającej zarządzanie finansami domowymi. W szczególności interfejs aplikacji w przeglądarce internetowej może odbiegać nieco od dzisiejszych standardów aplikacji internetowych. Przykładowy widok ekranu aplikacji Kontomierz prezentuje Rysunek 6.

Największą zaletą aplikacji, jest integracja z systemami bankowości elektronicznej umożliwiająca automatyczne pobieranie stanu konta oraz historii transakcji z wielu kont jednocześnie. Aplikacja analizuje te dane, kategoryzuje automatycznie wydatki, a następnie przedstawia je użytkownikowi w przejrzystej, zbiorczej formie, dzięki czemu użytkownik widzi swoje przychody i wydatki z wielu kont w jednym miejscu. Oprócz automatycznego importu danych bezpośrednio z kont bankowych aplikacja umożliwia import danych z pliku CSV oraz ręczne dodawanie przychodów oraz wydatków.

¹ Przedstawiona analiza dotyczy aplikacji Kontomierz w starszej wersji, która dostępna była w momencie przygotowywania analizy. Obecnie aplikacja dostępna jest w nowszej wersji z odświeżonym interfejsem graficznym.

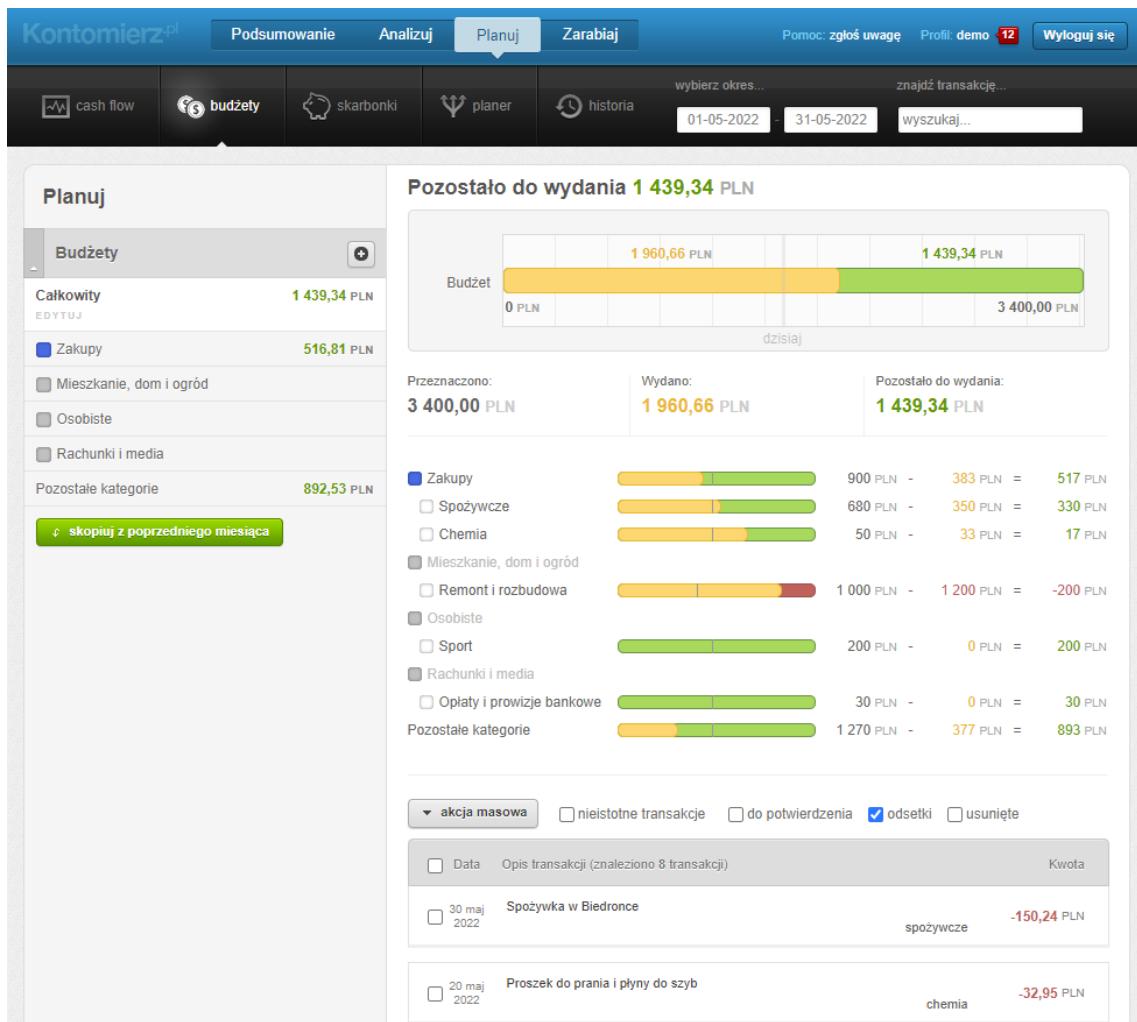


Rysunek 6 Widok ekranu „Konta” w aplikacji internetowej Kontomierz

Aplikacja Kontomierz umożliwia także zarządzanie fizyczną gotówką dzięki funkcji Portfel. Funkcja ta umożliwia automatyczne zasiliwanie portfela gotówką podczas wypłat z bankomatu. Portfel umożliwia również dodawanie gotówki w sposób ręczny.

Oprócz zarządzania bieżącymi przepływami pieniężnymi aplikacja posiada ciekawą funkcję planowania wydatków, która pozwala na przydzielenie odpowiednich kwot przeznaczonych na wydatki w ramach określonej kategorii. Na przykład możliwe jest zaplanowanie kwoty wydatków na artykuły spożywcze w danym miesiącu, a następnie monitorowanie stanu wydatków. Pomaga przy tym również przystępna w odbiorze wizualizacja wydatków za daną kategorię, co prezentuje Rysunek 7. Planować można nie tylko wydatki, ale również oszczędności, w czym pomaga funkcja skarbonki. Jest to prosta funkcja pozwalająca na obliczenie kwoty jaką należy odkładać miesięcznie,

aby po ustalonym czasie odłożyć określoną sumę. Aplikacja pozwala jednocześnie na monitorowanie stanu skarbonki, w celu zwiększenia prawdopodobieństwa osiągnięcia celu w ustalonym terminie. Aplikacja pozwala również planować stałe regularne wydatki, o których przypomina na głównym ekranie.



Rysunek 7 Ekran planowania wydatków aplikacji Kontomierz

Ponadto, aplikacja umożliwia zarządzanie takimi składnikami majątku jak lokaty oraz należności (takie jak np. kredyty hipoteczne), pozwala również na dodawanie innych bliżej nie określonych składników majątku. Niestety funkcje związane z zarządzaniem składnikami majątku są w aplikacji bardzo uproszczone i sprowadzają się w zasadzie jedynie do dodawania składników majątku trwałego oraz ich wartości, przez co pozbawione są wielu elementów koniecznych do kompleksowego zarządzania majątkiem, opisanych w rozdziale 2.

3.6 Podsumowanie

W niniejszym rozdziale przeanalizowano oraz porównano wybrane aplikacje, będące pewnego rodzaju obrazem przedstawiającym rozwiązania dostępne na rynku. Szczegółowe porównanie aplikacji sporzązone wg ustalonych w punkcie 3.1. kryteriów prezentuje Tabela 2.

Przeanalizowane aplikacje w większości przypadków skupią swoją funkcjonalność na zarządzaniu budżetem gospodarstwa domowego oraz przepływami pieniężnymi, oferując przy tym wiele ciekawych funkcji, kategoryzacji, synchronizacji czy wizualizacji danych w postaci wykresów. Wszystkie te funkcje automatyzują proces prowadzenia rachunków i są bardzo pomocne w zarządzaniu domowym budżetem. Większość przeanalizowanych aplikacji posiada również funkcje umożliwiające planowanie wydatków oraz monitorowanie realizacji przyjętego planu, co pozwala na osiągnięcie realnych oszczędności.

Niestety w większości przeanalizowanych aplikacji całkowicie pominięte są aspekty związane z zarządzaniem majątkiem, w szczególności majątkiem trwałym, które dla gospodarstw domowych wydają się być równie istotne. Należy przy tym podkreślić, że analiza dotyczyła jedynie aplikacji dostępnych na rynku polskim. Istnieją bowiem aplikacje niedostępne na rynku polskim, które oferują zdecydowanie większe możliwości. Ze względów licencyjnych ich analiza nie była jednak możliwa. Przykładem takiej aplikacji jest aplikacja Personal Capital (personalcapital.com), która posiada bardzo rozbudowany zakres funkcji związany z zarządzaniem oszczędnościami i inwestycjami [10]. Inną bardzo popularną aplikacją niedostępną oficjalnie na rynku polskim jest aplikacja Mint, która również przewyższa pod względem możliwości opisane w niniejszym rozdziale aplikacje.

Spośród przeanalizowanych aplikacji najbogatszym zakresem oferowanych funkcji cechuje się aplikacja Kontomierz. Aplikacja ta jako jedyna z przeanalizowanych aplikacji wspiera także w podstawowym zakresie zarządzanie majątkiem rzeczowym oraz inwestycjami i należnościami długoterminowymi. Niniejsza analiza wykazała jednak, że wciąż istnieje potrzeba stworzenia nowej aplikacji, która kompleksowo pokryła by wszystkie zagadnienia związane z zarządzaniem domowymi finansami, a w szczególności zagadnienia związane z zarządzaniem majątkiem, których to obecnie dostępne na rynku aplikacje nie wspierają w pełnym zakresie.

Tabela 2 Porównanie funkcji wybranych aplikacji do zarządzania finansami domowymi

	Funkcja	Money Manager	Easy Budget	Wallet	Kontomierz
Zarządzanie majątkiem	Zarządzanie majątkiem rzeczowym	Nie	Nie	Nie	podstawowe
	Zarządzanie inwestycjami długoterminowymi	Nie	Nie	Nie	podstawowe
	Zarządzanie należnościami długoterminowymi	Nie	Nie	podstawowe	podstawowe
	Zarządzanie obrotowym majątkiem rzeczowym	Nie	Nie	Nie	Nie
	Obsługa sald debetowych i limitów kart kredytowych	Nie	Nie	Tak	Nie
Zarządzanie budżetem	Planowanie budżetu	Nie	Tak	Tak	tak
	Bilans dochodów i wydatków	Tak	Tak	Tak	tak
	Kategorie dochodów i wydatków	Tak	Tak	Tak	tak
	Wydatki regularne z przypomnieniami	Tak ²	Bez przypomnień	Nie	Tak
	Przeglądanie historii dochodów i wydatków	Tak	Tak	Tak	Tak
	Wizualizacja wydatków i dochodów	Tak	Tak	Tak	Tak
	Podział na gotówkę oraz konta bankowe	Nie	Nie	Tak	Tak
	Obsługa wielu kont	Tak	Nie	Tak	Tak
Pozostałe funkcje	Integracja z systemami bankowymi	Nie	Nie	Tak	Tak
	Aplikacja online (w przeglądarce)	Nie	Tak	Tak	Tak
	Aplikacja mobilna	Tak	Nie	Tak	Tak
	Import danych z plików (np. CSV, TSV, EXCEL)	Nie	Tak	Tak	Tak
	Export danych do pliku (CSV, TSV, EXCEL)	Tak (Excel)	Nie	Nie	Tak
	Obsługa wielu walut	Tak	Nie	Tak	Tak

² Aplikacja Money Manager firmy Innim Mobile Exp posiada funkcję wydatków regularnych, a także osobną funkcję przypomnień. Nie ma jednak możliwości bezpośredniego zdefiniowania przypomnienia dla określonego wydatku regularnego.

4 Koncepcja własnego rozwiązania

W tym rozdziale przedstawiono koncepcję własnego rozwiązania, w skład której wchodzi koncepcja rozwiązania użytkowego oraz koncepcja rozwiązania technologicznego. Koncepcja rozwiązania użytkowego skupia się na wymaganiach funkcjonalnych, podczas gdy koncepcja rozwiązania technologicznego obejmuje między innymi proponowane warianty rozwiązań architektonicznych i technologicznych oraz ostateczną przyjętą koncepcję rozwiązania technologicznego.

Podkreślić należy, że realizacja aplikacji wspomagającej zarządzanie domowymi finansami, stanowiła tutaj przede wszystkim tło dla implementacji nowoczesnych rozwiązań technologicznych, w tym architektury bazującej na mikrousługach oraz asynchronicznej komunikacji opartej o zdarzenia. Z tego względu zakres funkcji samej aplikacji został ograniczony do funkcji podstawowych, pozwalających jednak pokazać zarówno możliwości jak i korzyści płynące z zastosowanych technologii.

4.1 Koncepcja rozwiązania użytkowego

Analiza przedstawiona w rozdziale drugim oraz rozdziale trzecim niniejszej pracy pozwoliła ustalić ogólny zbiór wymagań funkcjonalnych jakie powinny posiadać aplikacje wspomagające zarządzanie finansami domowymi. Ponadto wykazała ona istotne braki w funkcjonalności dostępnych na rynku aplikacji, które skupiają się głównie na funkcjach związanych ze wspomaganiem procesu zarządzania budżetem domowym, pomijając w większości przypadków funkcje związane z zarządzaniem majątkiem gospodarstwa domowego.

Aby aplikacja mogła być konkurencyjna i użyteczna dla użytkowników musi w co najmniej w podstawowym zakresie umożliwiać zarządzanie budżetem domowym oraz przepływami pieniężnymi. Będzie to zrealizowane przede wszystkim po przez możliwość ręcznego dodawania transakcji – zarówno przychodów jak i wydatków. Aplikacja będzie zawierała zestaw wstępnie zdefiniowanych kategorii do których można będzie przypisać każdą transakcję. Zestaw ten będzie można dowolnie edytować i rozszerzać. Na podstawie dodanych transakcji tworzone będą wykresy podsumowujące przychody i wydatki dla bieżącego miesiąca. Oprócz tego aplikacja będzie miała możliwość przeglądania historii wszystkich transakcji.

Efektywne zarządzanie budżetem nie jest możliwe bez jego zaplanowania, dlatego aplikacja będzie umożliwiała zaplanowanie miesięcznego budżetu po przez przypisanie odpowiednich limitów do konkretnych kategorii wydatków. Odpowiednie informacje odnośnie wykorzystania budżetu (i poszczególnych jego części przypisanych do konkretnych kategorii) będą wyświetlane w postaci tekstuowej oraz graficznej.

Z przedstawionej w rozdziale 2 charakterystyki problemu oraz przedstawionej w rozdziale 3 analizy podobnych rozwiązań wynika, że dostępne na rynku aplikacje wspomagające zarządzanie finansami domowymi skupiają się głównie na zarządzaniu przepływami pieniężnymi, prawie całkowicie pomijając kwestie związane z zarządzaniem majątkiem (aktywami). Sensowne więc wydaje się zbudowanie aplikacji wyposażonej również w taką funkcję. Kwestia zarządzania aktywami jest jednak wyjątkowo skomplikowana, wymaga bowiem kompleksowego pokrycia szeregu zagadnień takich jak m.in.:

- zażądanie majątkiem trwałym, w tym rejestr składników majątku trwałego umożliwiający ustalenie (oszacowanie) wartości majątku i poszczególnych jego składników,
- zarządzanie oszczędnościami, w tym rejestr oszczędności,
- zarządzanie długami, w tym rejestr długów umożliwiający ustalenie kosztu dłużu oraz poszczególnych składników składających się na ten koszt.

Ze względu na złożoność zagadnienia, które wykracza poza nadzędny cel pracy, jakim jest przede wszystkim analiza rozwiązań technologicznych, kwestię zarządzania majątkiem ograniczono do funkcji umożliwiających zarządzanie oszczędnościami, w tym tworzenie, edycję oraz usuwanie depozytów i lokat terminowych.

4.2 Koncepcja rozwiązania technicznego

Podczas projektowania aplikacji wspomagającej zarządzanie domowymi finansami, będącej przedmiotem niniejszej pracy dyplomowej przyjęto założenie, że będzie to aplikacja dostępna online, jako usługa sieciowa (ang. Software as a Service). Ze względu na charakter aplikacji, potencjalną możliwość jej integracji z dostawcami usług zewnętrznych (jak np. z systemami bankowości elektronicznej) oraz założeniem, że aplikacja docelowo powinna być dostępna zarówno na urządzeniach stacjonarnych jak i mobilnych, całkowicie porzucono koncepcję wykonania aplikacji w formie desktopowej.

Niniejsza praca skupia się ponadto głównie na części serwerowej aplikacji (ang. backend), gdzie założeniem nadrzędnym jest stworzenie systemu oferującego interfejs programistyczny API (ang. Application User Interface) z wykorzystaniem stylu architektonicznego REST (ang. Representational State Transfer), dzięki czemu możliwe jest stworzenie usługi w modelu określonym w języku angielskim jako headless software. Koncepcja headless software jest w ostatnim czasie bardzo popularna wśród firm produkujących oprogramowanie, ponieważ pozwala oddzielić funkcje związane z realizacją logiki poszczególnych elementów systemu od interfejsu użytkownika. Sprawia to, że aplikacja serwerowa zbudowana w modelu headless software może być wykorzystana przez innych producentów oprogramowania specjalizujących się z kolei w rozwoju warstw związanych z interfejsem użytkownika (ang. frontend). Dla realizacji niniejszej pracy dyplomowej zbudowana zostanie również uproszczona aplikacja odpowiedzialna za obsługę interfejsu użytkownika, co umożliwi prezentacje działania wszystkich funkcji aplikacji.

Pierwszą kwestią związaną z koncepcją rozwiązania technicznego będzie wybór odpowiedniej architektury systemu. Rozważyć można dwa modele tj. tradycyjny model aplikacji monolitycznej oraz nieco bardziej nowoczesny model wykorzystujący architekturę mikroserwisów (nazywanych w polskim piśmiennictwie również mikrousługami). Oba te modele mają oczywiście swoje wady i zalety, przy czym największą zaletą architektury mikroserwisowej z perspektywy projektowanej aplikacji, jest jej nieporównywanie lepsza skalowalność [11]. Porównanie wad i zalet architektury monolitycznej i architektury mikro serwisowej przedstawia Tabela 3.

Ponieważ jednym z najważniejszych założeń przyjętych podczas projektowania aplikacji jest zapewnienie odpowiedniej wydajności przy obsłudze zróżnicowanej liczby użytkowników, aplikacja powinna więc cechować się bardzo dobrą skalowalnością. Ponadto biorąc pod uwagę charakter systemu jakim jest aplikacja do wspomagania zarządzania domowymi finansami, którą można podzielić na grupy funkcji przeznaczone do realizacji powiązanych ze sobą zadań, wskazane jest, aby aplikacja miała budowę modułową, pozwalającą również na rozszerzanie jej funkcji o dodatkowe moduły w przyszłości. Biorąc pod uwagę powyższe założenia, opcją najbardziej optymalną wydaje się być architektura mikroserwisowa, zapewniająca zarówno modularność jak i wysoką skalowalność aplikacji, a przy tym dobrze nadającą się do stworzenia aplikacji w modelu headless software.

Tabela 3 Porównanie wad i zalet aplikacji monolitycznych oraz aplikacji o architekturze mikroserwisowej [11] [12]

Aplikacje monolityczne	
Zalety	Wady
<ul style="list-style-type: none"> • łatwe w rozwoju • łatwe we wdrożeniu • łatwe monitorowanie i testy end-to-end • szybka i niezawodna komunikacja pomiędzy warstwami systemu • większe bezpieczeństwo ze względu na brak komunikacji pomiędzy rozproszonymi elementami systemu • transakcyjność w rozumieniu ACID. 	<ul style="list-style-type: none"> • ograniczona skalowalność • niska odporność na błędy (jeden błąd może spowodować awarie całego systemu) • trudniejsze i kosztowniejsze utrzymanie aplikacji • niższa dostępność (po każdej aktualizacji konieczne ponowne wdrożenie).
Aplikacje mikroserwisowe	
Zalety	Wady
<ul style="list-style-type: none"> • łatwa skalowalność • większa odporność na błędy (błąd w jednym serwisie nie spowoduje błędu całej aplikacji), • łatwe w utrzymaniu (ze względu na podzielenie na mniejsze, łatwiejsze do zrozumienia części), • wysoka dostępność (ponowne wdrożenie mikroserwisu po aktualizacji dzięki konteneryzacji nie wymaga długiego czasu przestoju), • łatwe w integracji z zewnętrznymi systemami dzięki wykorzystaniu API, • możliwość rozwoju po przez dodawanie nowych mikroserwisów lub wymianę istniejących, bez wpływu na pozostałe. 	<ul style="list-style-type: none"> • bardziej skomplikowane wdrożenie • trudność w zapewnieniu integralności danych oraz transakcyjności (mikroserwisy posiadają swoje własne bazy danych) • trudniejsze zapewnienie bezpieczeństwa, spowodowane koniecznością zabezpieczenia komunikacji pomiędzy mikroserwisami • trudniejsze wprowadzanie zmian obejmujących swoim zakresem wiele usług

Systemy mikroserwisowe wspierane są również szeroko przez oprogramowanie do konteneryzacji oraz zarządzania systemami kontenerowymi, co ułatwia automatyzację procesu wdrażania takich aplikacji, a także umożliwia ich łatwe skalowanie (w tym również skalowanie automatyczne, bazujące na danych o ruchu w poszczególnych mikroserwisach). Sprawia to, że projektowany system mikroserwisowy będzie mógł być potraktowany jako

natywne rozwiązanie chmurowe (ang. Cloud Native), czyli rozwiązanie zaprojektowane do wdrożenia z wykorzystaniem systemów takich jak np. Kubernetes.

Istotną kwestią związaną z wyborem architektury mikroserwisowej jest wybór sposobu komunikacji pomiędzy poszczególnymi mikroserwisami. W podejściu bardziej klasycznym komunikacja ta odbywa się w sposób synchroniczny, najczęściej z wykorzystaniem interfejsu REST API. Komunikacja synchroniczna pomiędzy mikroserwisami ma jednak następujące wady [13]:

- Powiązania typu punkt-punkt – mikrousługi synchroniczne, opierają się na innych usługach, które z kolei używają innych usług zależnych. Powoduje to dużą liczbę wzajemnych powiązań, co z kolei utrudnia wprowadzanie zmian w przyszłości.
- Skalowanie zależne – możliwość skalowania pionowego usługi zależy od zdolności skalowania usług zależnych.
- Konieczność obsługi awarii – w przypadku kiedy usługa zależna nie działa, nie będzie działać również serwis, który z tej usługi korzysta, w takim przypadku należy zaimplementować obsługę takiej awarii.
- Konieczność zarządzania wersjami i zależnościami interfejsu API.
- Konieczność dostępu do danych powiązanych z implementacją – chociaż istnieją strategie minimalizujące, mikroserwisy i tak często będą potrzebowaly uzyskiwać dostęp do danych z innych usług.
- Trudność w testowaniu – ponieważ każda mikrousługa wymaga do działania szeregu innych usług zależnych, testy integracyjne będą przez to utrudnione.

Alternatywą dla komunikacji synchronicznej jest komunikacja asynchroniczna. Implementacja tego rodzaju komunikacji jest możliwa dzięki zastosowaniu architektury opartej o zdarzenia (ang. Event-Driven Architecture). W przypadku komunikacji asynchronicznej, mikroserwisy komunikują się po przez wysyłanie wiadomości lub zdarzeń, które z kolei są odbierane i przetwarzane przez inne mikroserwisy. Różnice pomiędzy wiadomościami oraz zdarzeniami opisano w paragrafie 5.2.

Komunikacja asynchroniczna ma szereg zalet, których część z nich wynika z braku (lub ograniczonej liczby) powiązań pomiędzy mikroserwisami. Są to m.in.:

- łatwa skalowalność pozioma,
- łatwa zmiana wymagań biznesowych,

- wysoka testowalność,
- wsparcie dla ciągłego dostarczania (ang. Continuous Delivery) [13].

Mikroserwisy wykorzystujące architekturę asynchroniczną opartą o zdarzenia, mogą być ponadto luźno pomiędzy sobą powiązane, przy pomocy danych dziedzinowych, a nie po przez interfejs API, co ułatwia zarządzanie zmianami [13].

Aplikacja będzie zbudowana z kilku mikroserwisów komunikujących się ze sobą w sposób asynchroniczny. Każdy z mikroserwisów będzie odpowiedzialny za poszczególną grupę funkcji, przewiduje się stworzenie następujących mikroserwisów:

- mikroserwis do obsługi funkcji związanych z zarządzaniem przepływami pieniężnymi,
- mikroserwis do obsługi funkcji związanych z zarządzaniem aktywami,
- mikroserwis do obsługi funkcji związanych z planowaniem.

Szczegółowe informacje na temat wykorzystanych narzędzi, języków programowania oraz architektury poszczególnych mikroserwisów, przedstawiono w rozdziale 5.

Oprócz warstwy serwerowej zbudowana zostanie osobna aplikacja odpowiedzialna za obsługę warstwy klienckiej, czyli za wyświetlanie interfejsu użytkownika w przeglądarce internetowej. Jako że, głównym celem niniejszej pracy jest zbudowanie efektywnej aplikacji chmurowej działającej w modelu headless software, aplikacja kliencka zostanie wykonana w formie uproszczonej.

Aplikacja kliencka będzie zaprojektowana jako aplikacja SPA (ang. Single Page Application), czyli aplikacja, która będzie posiadać tylko jeden plik HTML, dzięki czemu jej zawartość nigdy nie będzie przeładowywana w przeglądarce w całości, a jedynie w niezbędnej części na żądanie użytkownika. Szczegółowe informacje na temat budowy aplikacji klienckiej, wykorzystanych narzędzi oraz języków programowania zostaną przedstawione w rozdziale 5.

5 Projekt ogólny

W niniejszym rozdziale przedstawiono ogólny projekt systemu, na który składają się:

- wymagania funkcjonalne i niefunkcjonalne, oraz diagramy przypadków użycia (ang. „use case”);
- opis architektury systemu,
- opis technologii zastosowanych przy budowie systemu i poszczególnych jego elementów;
- opis zastosowanej nierelacyjnej bazy danych, a także opis struktury wraz z diagramami ilustrującymi model danych,
- projekt interfejsu użytkownika.

5.1 Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych

5.1.1 Wymagania funkcjonalne

Projektowany system będzie umożliwiał:

- tworzenie prywatnych kont przez użytkowników;
- logowanie do aplikacji przez użytkowników oraz wylogowanie się;
- dodawanie, edycję oraz usuwanie przychodów oraz wydatków;
- wyświetlanie listy przychodów i wydatków w formie tabeli, z możliwością wyszukiwania oraz sortowania transakcji;
- wyświetlanie stanu posiadanych środków pieniężnych;
- dodawanie, edycję oraz usuwanie kategorii przychodów oraz wydatków;
- przypisywanie limitu wydatków do kategorii wydatków;
- wyświetlanie listy kategorii przychodów i wydatków;
- automatyczne dodawanie domyślnych kategorii podczas pierwszego logowania użytkownika do systemu;
- dodawanie, edycję oraz usuwanie depozytów oraz lokat bankowych;
- wyświetlanie listy depozytów i lokat bankowych;
- automatyczną validację wprowadzonych przez użytkownika wartości, w tym informowanie użytkownika o braku poprawności wprowadzonych danych oraz blokadę możliwości przesłania formularza dla niepoprawnych bądź niepełnych danych;
- wyświetlenia wykresu na głównej stronie aplikacji, ilustrującego udział poszczególnych kategorii wydatków w bieżącym miesiącu;

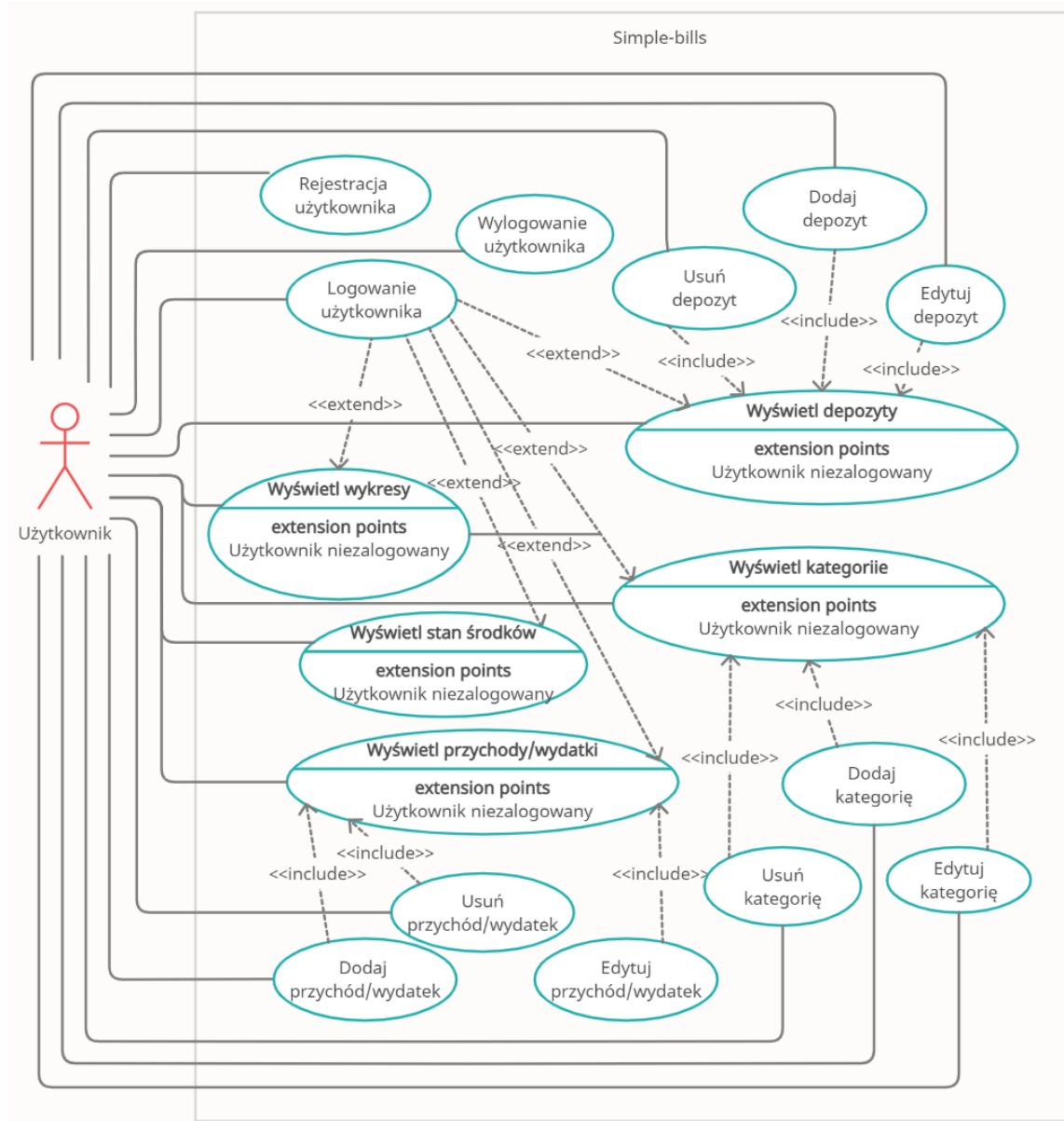
- wyświetlenie wykresu na głównej stronie aplikacji, pokazującego wykorzystanie limitów dla poszczególnych kategorii oraz wykresu pokazującego wykorzystanie całkowitego limitu wydatków.

5.1.2 Wymagania niefunkcjonalne

- Wymagania związane z interfejsem użytkownika (GUI):
 - interfejs aplikacji powinien działać jako aplikacja internetowa w przeglądarce;
 - aplikacja powinna posiadać klasyczny interfejs z menu umieszczonym w górnej części.
- Wymagania związane z dostępnością i lokalizacją:
 - język: angielski;
 - waluta: dolar amerykański;
 - obsługa przeglądarek: Chrome, Edge;
 - prawidłowe wyświetlanie się interfejsu użytkownika na komputerach stacjonarnych oraz laptopach.
- Bezpieczeństwo:
 - przesyłanie danych z przeglądarki do mikroserwisów oraz z mikroserwisów do systemów zewnętrznych zabezpieczone protokołem HTTPS;
 - uwierzytelnianie i autoryzacja z wykorzystaniem protokołu OpenID Connect oraz OAuth 2.0 i metody Authorization Code;
 - delegowanie przechowywania danych wrażliwych do zewnętrznego systemu, będącego dostawcą tożsamości np. systemu Keycloak (Open Source).
- Wymagania systemowe:
 - ładowanie poszczególnych komponentów do wyświetlenia sukcesywnie, kiedy wyświetlenie ich jest konieczne, bez konieczności przeładowywania całej zawartości strony – architektura typu Single Page Application;
 - wszystkie komponenty systemu zaprojektowane i zbudowane w sposób umożliwiający uruchomienie ich w kontenerach z wykorzystaniem systemu Docker i/lub Kubernetes.

5.1.3 Diagramy przypadków użycia

Na podstawie wymagań funkcjonalnych określonych w paragrafie 5.1.1. przygotowano diagramy przypadków użycia, które przedstawia Rysunek 8.



Rysunek 8 Diagramy przypadków użycia

5.1.4 Scenariusze dla wybranych przypadków użycia

Tabela 6 przedstawia scenariusz dla wybranego przypadku użycia „Dodaj przychód/wydatek” oraz powiązanych z nim przypadków „Logowanie użytkownika” i „Wyświetl przychody/wydatki”. Z uwagi na ograniczenia dotyczące objętości niniejszej pracy, pozostałe przypadki użycia zostały pominięte.

Tabela 4 Scenariusz dla przypadku użycia „Logowanie użytkownika”

Nazwa przypadku użycia	Logowanie użytkownika	
Cel w kontekście systemu	Użytkownik loguje się do systemu	
Warunki wstępne	Użytkownik posiada konto w systemie	
Warunek pomyślnego zakończenia	Użytkownik zalogował się do systemu	
Stan końcowy - niepowodzenie	Zwrócono informację o nieprawidłowym loginie lub haśle	
Główni Aktorzy	Użytkownik	
Aktorzy współuczestniczący	-	
Wywołanie (inicjacja) przypadku użycia	Użytkownik żąda możliwości zalogowania się do systemu	
Przypadki użycia - include	-	
Przypadki użycia - extend	-	
Główny przepływ	Krok	Czynność
	1	Użytkownik żąda możliwości zalogowania się
	2	Użytkownik wpisuje login i hasło
	3	Dane użytkownika są weryfikowane
	4	Użytkownik zostaje zalogowany do systemu
	5	Użytkownik zostaje przekierowany do strony głównej dostępnej dla zalogowanych użytkowników
Rozszerzenia głównego przepływu	Krok	Rozgałęzienie czynności
	3.1	Wynik walidacji wprowadzonego loginu i hasła jest negatywny
	3.2	Próba zalogowania zostaje odrzucona
	3.3	Wyświetlana jest informacja o nieprawidłowym loginie lub haśle

Tabela 5 Scenariusz dla przypadku użycia „Wyświetl przychody/wydatki”

Nazwa przypadku użycia	Wyświetl przychody/wydatki	
Cel w kontekście systemu	Użytkownik wyświetla listę przychodów i wydatków	
Warunki wstępne	Użytkownik posiada konto w systemie	
Warunek pomyślnego zakończenia	Lista przychodów i wydatków została wyświetlona	
Stan końcowy - niepowodzenie	Lista nie została wyświetlona / Zwrócono komunikat o błędzie systemu	
Główni Aktorzy	Użytkownik	
Aktorzy współuczestniczący	-	
Wywołanie (inicjacja) przypadku użycia	Użytkownik żąda wyświetlenia listy przychodów i wydatków	
Przypadki użycia - include	-	
Przypadki użycia - extend	Logowanie użytkownika	
Glówny przepływ	Krok	Czynność
	1	Extend::Logowanie użytkownika
	2	Użytkownik żąda wyświetlenia listy przychodów i wydatków
	3	Dane pobierane są z serwera
	4	Lista przychodów i wydatków zostaje wyświetlona
Rozszerzenia głównego przepływu	Krok	Rozgałęzienie czynności
	1.1	Próba logowania zostaje odrzucona z powodu nieprawidłowego loginu lub hasła
	2.1	Użytkownik wprowadza dodatkowe kryteria wyszukiwania
	3.1	Serwer jest niedostępny lub zwraca odpowiedź z kodem błędu
	3.2	Aplikacja wyświetla komunikat informujący o błędzie systemu

Tabela 6 Scenariusz dla przypadku użycia „Dodaj przychód/wydatek”

Nazwa przypadku użycia	Dodaj przychód/wydatek	
Cel w kontekście systemu	Użytkownik dodaje nowy przychód/wydatek	
Warunki wstępne	Użytkownik posiada konto w systemie	
Warunek pomyślnego zakończenia	Użytkownik dodał nowy przychód/wydatek	
Stan końcowy - niepowodzenie	Dodanie nowego przychodu/wydatku zostało odrzucone	
Główni Aktorzy	Użytkownik	
Aktorzy współuczestniczący	-	
Wywołanie (inicjacja) przypadku użycia	Użytkownik żąda dodania przychodu/wydatku	
Przypadki użycia - include	Wyświetl przychody/wydatki	
Przypadki użycia - extend	-	
Główny przepływ	Krok	Czynność
	1	Include:: Wyświetl przychody/wydatki
	2	Użytkownik żąda dodania przychodu/wydatku
	3	Użytkownik wypełnia formularz utworzenia nowego przychodu/wydatku
	4	Dane są walidowane po stronie przeglądarki
	5	Dane są walidowane po stronie aplikacji serwerowej
	6	Przychód/wydatek zostaje dodany i wyświetlony na liście
Rozszerzenia głównego przepływu	Krok	Rozgałęzienie czynności
	5.1	Wynik walidacji po stronie przeglądarki negatywny
	5.1	Przycisk tworzenia depozytu zablokowany
	6.1	Wynik walidacji po stronie serwera negatywny
	6.2	Aplikacja serwerowa zwraca odpowiedź z kodem 400 (Bad Request)
	6.3	Aplikacja wyświetla komunikat przyjazny dla użytkownika

5.2 Architektura systemu

Projektowany system będzie posiadał trójwarstwową strukturę. Warstwa kliencka (ang. frontend) będzie aplikacją internetową napisaną z wykorzystaniem szkieletu bibliotek (ang. Framework) Angular. Działać będzie ona w przeglądarce

internetowej i komunikować się z warstwą serwerową za pomocą interfejsu REST API.

Warstwa serwerowa (ang. backend) z kolei, zbudowana zostanie w oparciu o architekturę mikrousług i będzie składać się z trzech mikroserwisów stanowiących niezależne aplikacje, napisane z wykorzystaniem języka Java i szkieletu bibliotek Spring Boot. Mikroserwisy komunikować się będą ze sobą w dwojaki sposób:

- synchronicznie – z wykorzystaniem interfejsu REST API.
- asynchronicznie – z wykorzystaniem brokera wiadomości RabbitMQ.

Komunikacja synchroniczna zostanie wykorzystana tylko w przypadkach, kiedy dla zachowania spójności danych, określony mikroserwis będzie potrzebował uzyskać natychmiastowej odpowiedzi na wysłane zapytanie. W większości przypadków jednak komunikacja będzie się odbywać asynchronicznie, w sposób taki, że określone zdarzenie, spowoduje dodanie wiadomości do kolejki, z której to zostanie ona odebrana, a następnie przetworzona przez „zainteresowany” mikroserwis. Taki rodzaj przetwarzania danych możemy nazwać architekturą bazującą na zdarzeniach (ang. Event-Based Architecture – EDA lub Event-Driven Architecture - EDA) lub też architekturą opartą o wiadomości (Message-Driven Architecture - MDA). W tym miejscu należało by wyjaśnić różnice pomiędzy architekturą opartą o zdarzenia a architekturą opartą o wiadomości. W tym celu należało by zdefiniować pojęcie zdarzenia i pojęcie wiadomości. Można przyjąć, że zdarzenie jest to porcja danych emitowana z danego komponentu, w sposób taki, że wszystkie inne uprawnione komponenty systemu mogą takie zdarzenie odebrać i przetworzyć (skonsumować). W przypadku wiadomości, komponenty systemu mogą posiadać swój unikalny adres na które inny komponent może wysyłać wiadomości [14].

W celu obsługi wymiany zarówno zdarzeń, jak i wiadomości stosuje się brokery zdarzeń lub brokery wiadomości, będące osobnymi niezależnymi aplikacjami. Należy przy tym zauważyć, że brokera zdarzeń można wykorzystać jako brokera wiadomości, natomiast broker wiadomości nie posiada pełnej funkcjonalności aby pełnić funkcję brokera zdarzeń. W przypadku brokera wiadomości, przesłane (publikowane) przez serwis (producenta) wiadomości umieszczane są w kolejce, z której to odbierane (konsumowane) są przez inny serwis (konsumenta), a następnie po potwierdzeniu usuwane z kolejki. Broker zdarzeń z kolei, nie usuwa wiadomości po ich skonsumowaniu przez konsumenta, a przechowuje rejestr wszystkich zdarzeń, dzięki czemu każdy konsument może mieć dostęp

do pełnego zbioru zdarzeń. Wzorce wprowadzane przez brokery komunikatów są poprawnymi wzorcami dla architektury opartej o zdarzenia, jednak nie oferują one wszystkich funkcji jakich wymaga taka architektura [13].

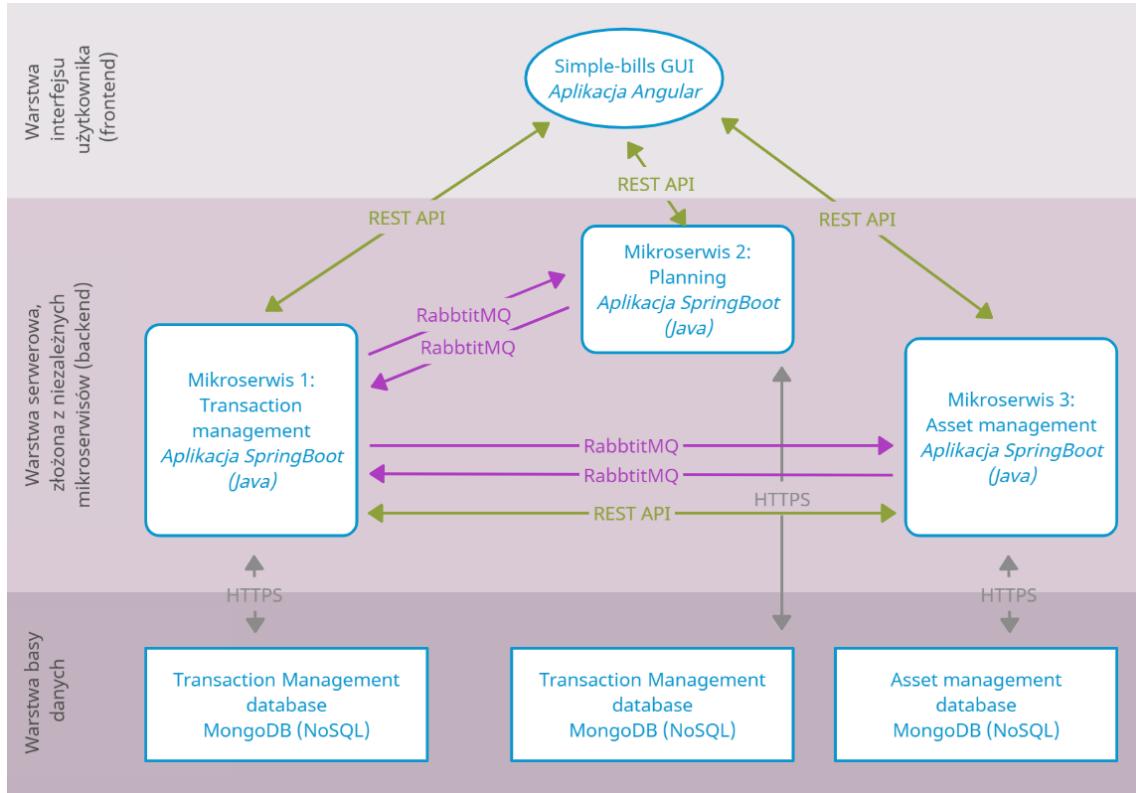
W projekcie niniejszego systemu dane wysyłane przez producentów kierowane są za pośrednictwem kolejki obsługiwanej przez brokera wiadomości RabbitMQ do konkretnych mikroserwisów, można więc przyjąć że zastosowano w tym przypadku architekturę opartą o wiadomości.

Architektura mikroserwisowa oraz architektura oparta o zdarzenia/wiadomości uzupełniają się wzajemnie przynosząc szereg korzyści takich jak:

- możliwość łatwego skalowania poziomego aplikacji po przez zwiększanie instancji danych mikroserwisów w zależności od obciążenia,
- możliwość łatwego skalowania pionowego aplikacji, po przez zwiększanie/zwiększenie zasobów aplikacji,
- efektywniejsze wykorzystanie zasobów fizycznych, dzięki podziałowi aplikacji na mniejsze części, które mogą być skalowane niezależnie, a także dzięki asynchronicznej komunikacji bez operacji blokujących, pozwalającej na bardziej równomierne obciążenie mikroserwisów.

Ostatnią warstwą systemu będzie nierelacyjna baza danych MongoDB. Każdy z serwisów posiadać będzie własną niezależną bazę danych, składającą się z kolekcji odpowiadających modelowi danych w mikroserwisach. Rysunek 9 przedstawia schemat zastosowanej architektury systemu. Na schemacie wprowadzono nazewnictwo dla poszczególnych aplikacji i mikroserwisów, gdzie:

- Simple Bills GUI – to aplikacja napisana w języku TypeScript wykorzystaniem środowiska Angular, działająca w przeglądarce, stanowiąca najwyższą warstwę systemu.
- Transaction management, Planning oraz Asset management – to nazwy mikroserwisów działających w warstwie serwerowej. Zostały one napisane w języku Java (wersja 17) z wykorzystaniem szkieletu bibliotek (ang. framework) Spring Boot.



Rysunek 9 Architektura systemu aplikacji Simple Bills

5.3 Architektura mikroserwisów

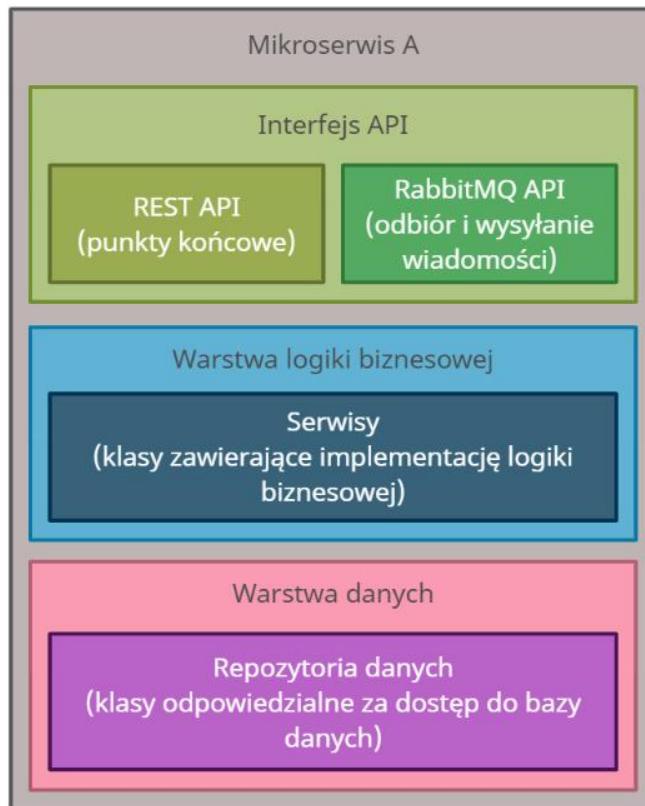
Główna część niniejszego systemu stanowi warstwa serwerowa składająca się z trzech mikroserwisów będących niezależnymi aplikacjami. Każdy z mikroserwisów posiada swoją bazę danych i może działać niezależnie. Taka autonomiczność poza kwestiami związanymi z możliwością łatwego skalowania poziomego serwisów jest największą zaletą systemów mikroserwisowych, umożliwia bowiem niezależne prace rozwojowe nad poszczególnymi mikroserwisami, w tym całkowicie niezależny proces ich wdrażania (ang. deployment), który w komercyjnych aplikacjach jest zwykle całkowicie zautomatyzowany dzięki zastosowaniu narzędzi ciągłego dostarczania i ciągłego wdrażania (ang. CI/CD – Continous integration / Continous delivery).

Wszystkie trzy mikroserwisy zbudowane są w oparciu o architekturę warstwową. Każdy mikroserwis składa się z 3 następujących warstw:

- warstwy interfejsu API, która stanowią punkty końcowe REST API (ang. endpoints) umożliwiające komunikację zewnętrznych aplikacji z mikroserwisem, a także klasy odpowiedzialne za wysyłanie i nasłuch przychodzących wiadomości asynchronicznych z wykorzystaniem brokera wiadomości RabbitMQ,

- warstwy logiki biznesowej, którą stanowią klasy określane mianem serwisów,
- warstwy danych, którą stanowią repozytoria danych, posiadające połączenie z nierelacyjną bazą danych MongoDB.

Schemat architektury warstwowej przedstawia Rysunek 10.



Rysunek 10 Architektura warstwowa zastosowana w mikroserwisach

5.4 Biblioteki i narzędzia użyte do budowy aplikacji klienckiej

Interfejs graficzny użytkownika zapewnia aplikacja internetowa Simple Bills GUI działającą w przeglądarce internetowej. Została ona napisana w języku TypeScript będącym nadzbiorem języka JavaScript. Do stworzenia aplikacji wykorzystano szkielet bibliotek (ang. framework) Angular.

W aplikacji wykorzystano również biblioteki RxJs umożliwiające tworzenie reaktywnych serwisów, a przede wszystkim reaktywne przetwarzanie zapytań REST API. Aplikacja korzysta również z zestawu stylów SCSS Bootstrap oraz bibliotek Ng Bootstrap oferujących m.in. funkcje wyświetlania tabel oraz okien modalnych zastosowanych w aplikacji. Aplikacja korzysta również

z biblioteki Ngx Charts oferującej zestaw wykresów. W celu obsługi ciasteczek w przeglądarce wykorzystano bibliotekę Ng Cookies.

Tabela 7 Lista najważniejszych bibliotek zastosowanych w aplikacji Simple Bills GUI

Nazwa biblioteki	Wersja	Opis
Angular	15.0.4	szkielet bibliotek (ang. framework)
TypeScript	4.8.4	język TypeScript
JavaScript	ES2021	język JavaScript
RxJS	7.5.0	programowanie reaktywne
Bootstrap	5.2.0	style SCSS
Ng Bootstrap	14.0	komponenty graficzne
Ngx Charts	20.1.2	zestaw wykresów graficznych
Ng Cookies	1.0.12	obsługa ciasteczek

5.5 Biblioteki i narzędzi wykorzystane do budowy mikroustug

Część serwerową aplikacji stanowią trzy mikroserwisy Transaction management, Planning oraz Asset management. Wszystkie mikroserwisy zostały napisane w języku Java z wykorzystaniem zestawu bibliotek Spring Boot. Do budowy aplikacji wykorzystano narzędzie Gradle.

Aplikacja została napisana w sposób reaktywny z wykorzystaniem bibliotek Spring WebFlux (Project Reactor). Ponadto aplikacja korzysta z bibliotek Spring Security, umożliwiających konfigurację zabezpieczeń w aplikacji oraz bibliotek wspierających protokół OAuth 2.0. Obsługa brokera wiadomości RabbitMQ została zaimplementowana również z wykorzystaniem biblioteki AMQP Spring Rabbit. Aplikacja korzysta również z bibliotek reaktywnych do obsługi bazy danych MongoDB. Inną ważną biblioteką jest biblioteka Lombok, rozszerzająca standardowe możliwości języka Java po przez wykorzystanie adnotacji do automatycznej obsługi powtarzalnych fragmentów kodu, jak np. do automatycznego tworzenia konstruktorów, getterów, setterów i innych standardowych elementów. W implementacji serwisów wykorzystano również biblioteki narzędziowe Apache Commons Lang 3.

Tabela 8 Lista najważniejszych bibliotek oraz narzędzi zastosowanych w implementacji mikroserwisów

Nazwa biblioteki / narzędzia / języka	Wersja	Opis
Gradle	7.5.1	narzędzie do budowania projektów
Java	17	język programowania
Groovy	3.0.13	język programowania
OpenJDK	17.0.2	środowisko programistyczne i uruchomieniowe
Spring Boot	2.7.5	szkielet bibliotek (ang. framework)
Spring Boot Starter Webflux	2.7.5	biblioteki reaktywne (Project Reactor)
Spring Boot Starter Security	2.7.5	konfiguracja zabezpieczeń
Spring Boot Starter Security OAuth Resource Server	2.7.5	obsługa OAuth 2.0 (Resource Server)
Spring Boot Starter Data MongoDB Reactive	2.7.5	reaktywna obsługa bazy danych MongoDB
Apache Commons Lang 3	3.12.0	obsługa ciasteczek
AMQP Spring Rabbit	2.4.7	obsługa brokera wiadomości RabbitMQ
Spock framework	2.3	biblioteki do tworzenia testów w języku Groovy
Springdoc OpenApi WebFlux	1.6.14	biblioteka do automatycznego generowania dokumentacji w formacie OpenAPI
Testcontainers	1.17.6	biblioteka do tworzenia automatycznych kontenerów testowych w środowisku Docker
Reactor Test	najnowsza	biblioteka do testowania strumieni reaktywnych
Simple Bills Commons	0.1.54	wspólna biblioteka zawierająca narzędzia oraz modele danych wykorzystywany w mikroserwisach

Oprócz standardowych bibliotek dostępnych w ramach szkieletu Spring Boot, opracowano również własną bibliotekę Simple Bills Commons, która zawiera narzędzia wykorzystywane we wszystkich trzech mikroserwisach. Biblioteka Simple Bills commons zawiera również modele danych oraz obiekty DTO (Data Transfer Object) stanowiące interfejs wymiany danych pomiędzy mikrousługami, a także wspólne narzędzia wykorzystywane w poszczególnych mikroserwisach, w tym konwertery i walidatory danych.

Oprócz bibliotek wykorzystanych do implementacji poszczególnych funkcji mikroserwisów, skorzystano również z kilku bibliotek zewnętrznych wykorzystywanych do tworzenia automatycznych testów jednostkowych oraz integracyjnych. Najważniejszą z tych bibliotek jest biblioteka Spock, która stanowi w zasadzie pełne środowisko testowe umożliwiające pisanie testów sterowanych danymi (ang. Data-Driven Tests - DDT) w języku Groovy.

Inną ważną biblioteką wykorzystywaną do tworzenia testów integracyjnych jest biblioteka Testcontainers. Umożliwia ona automatyczne uruchamianie testowych instancji m.in. bazy danych MongoDB oraz brokera wiadomości RabbitMQ w kontenerach Dockera.

5.6 Koncepcja przechowywania danych

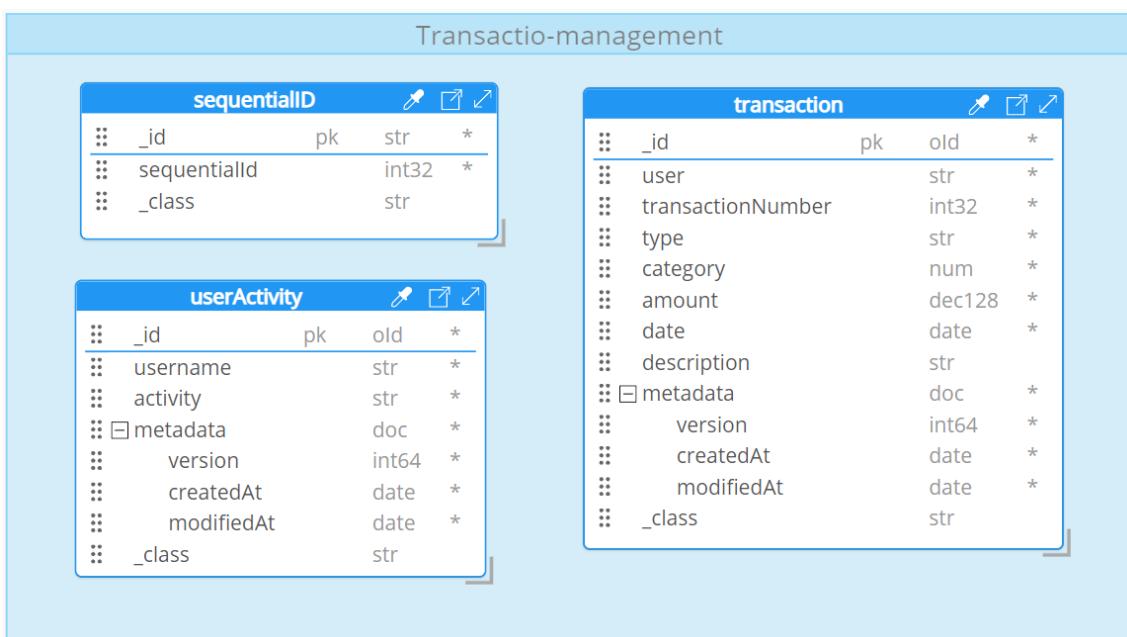
W systemach mikroserwisowych każda z mikrosług powinna działać w sposób całkowicie niezależny, a zatem powinna posiadać również osobną niezależną bazę danych. Taka architektura nie sprzyja zachowaniu spójności danych, a większości przypadków uniemożliwia realizację transakcyjności w rozumieniu zasad ACID. Relacyjne bazy danych których niewątpliwą zaletą jest właśnie możliwość utrzymania spójności danych, a także normalizacja danych, pozwalająca uniknąć zbędnej redundancji, obarczone są jednak również pewnymi wadami. Relacyjna budowa oraz istnienie związków pomiędzy tabelami sprawia, że taką bazę danych skalować można jedynie pionowo (po przez zwiększenie zasobów fizycznych takich jak pamięć RAM czy powierzchnia dyskowa do zapisu danych). Ponadto mechanizmy gwarantujące transakcyjność nie sprzyjają wydajności takich baz danych. Nierelacyjne bazy danych z kolei, pozbawione relacji i związków pomiędzy tabelami, a w wielu przypadkach również obsługi transakcji, pozwalają najczęściej na zdecydowanie szybszy dostęp do danych niż bazy relacyjne, mogą być przy tym skalowane poziomo [15].

W praktyce w systemach mikroserwisowych stosuje się zarówno bazy relacyjne jak i nierelacyjne, w zależności od potrzeb. W dobrze zaprojektowanych systemach, bardzo często będziemy mogli spotkać oba te rozwiązania jednocześnie. W takich przypadkach część mikroserwisów będzie korzystała z baz relacyjnych zapewniających spójność danych tam gdzie jest to krytyczne, z kolei w przypadku usług wymagających przetwarzania danych w czasie rzeczywistym, gdzie czas odpowiedzi systemu ma kluczowe znaczenie zastosowane zostaną nierelacyjne bazy danych, które jednak w ogólnym rozrachunku będą dominowały.

W niniejszej pracy ograniczono się do jednego rozwiązania, a mianowicie do zastosowania nierelacyjnej bazy danych MongoDB. Baza ta umożliwia przechowywanie danych w formacie BSON, który wzorowany jest kolejno na popularnym formacie JSON. Spójność danych zapewniona jest na poziomie mikroserwisów, po przez odpowiednią implementację ich obsługi. System ten

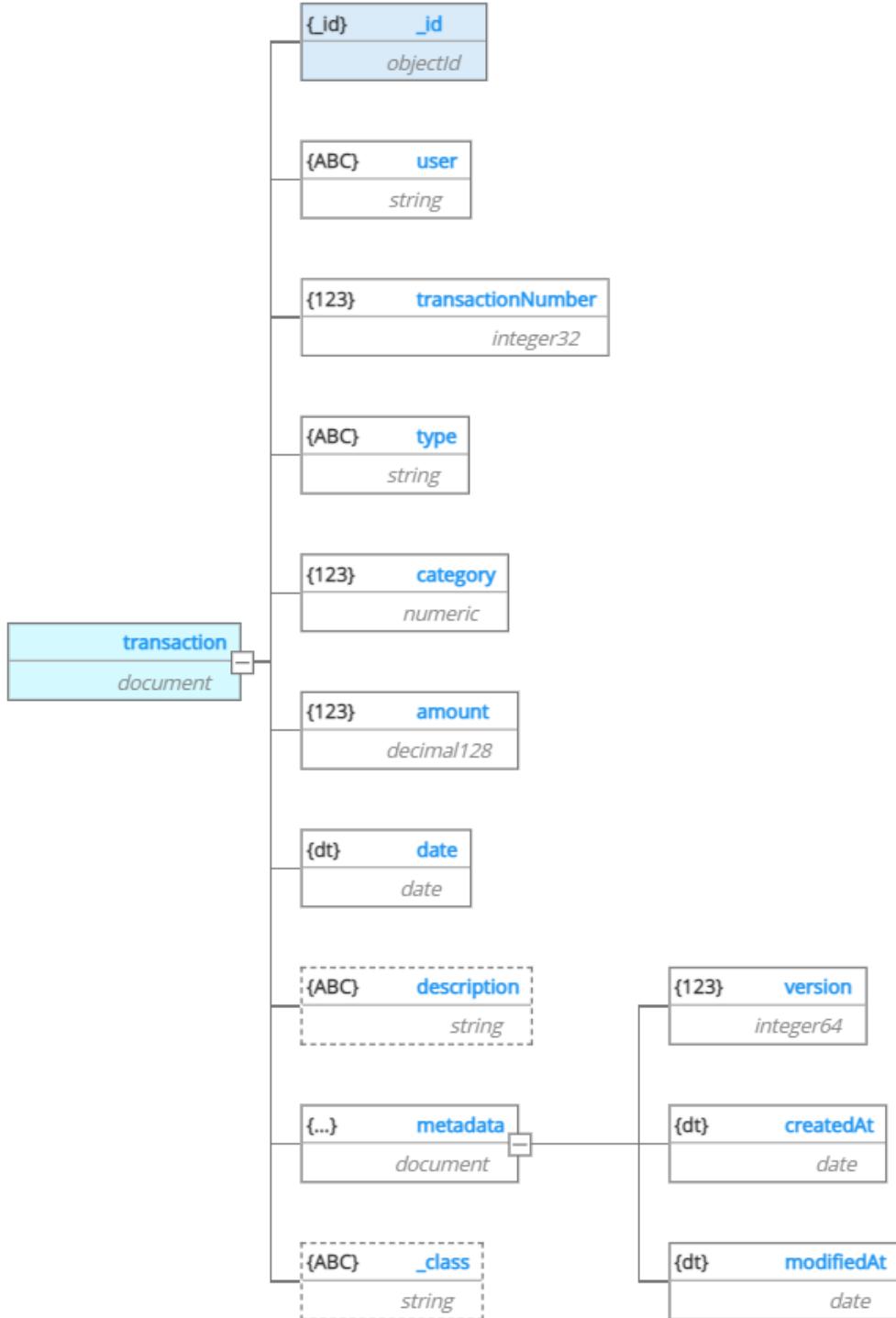
cechuje się pewną redundancją danych, co jest w większości przypadków nieuniknione podczas wykorzystania baz nierelacyjnych. Niewątpliwą zaletą takiego rozwiązania będzie jednak szybkość dostępu do danych, a także możliwość poziomego skalowania bazy danych wraz z danym mikroserwisem.

W bazie danych MongoDB dane przechowywane są w kolekcjach, które w pewnym sensie odpowiadają tabelą w bazach relacyjnych. Każda baza może posiadać wiele kolekcji. W projektowanym systemie każdy z trzech mikroserwisów będzie posiadał własną bazę danych, a w niej własne kolekcje. Mikroserwis Transaction management będzie składał się z trzech kolekcji: sequentialId, transaction oraz userActivity. Model danych mikroserwisu Transaction management przedstawia Rysunek 11.



Rysunek 11 Model bazy danych mikroserwisu Transaction management

Model danych serwisu jest stosunkowo prosty. Kolekcje transaction oraz userActivity posiadają po jednym obiekcie zagnieżdżonym, jakim jest obiekt metadata przechowujący informacje o wersji dokumentu oraz datę jego utworzenia i ostatniej modyfikacji. Na podstawie przedstawionego diagramu można wygenerować schemat poszczególnych kolekcji. Schemat taki można przedstawić również w postaci tekstowego pliku JSON (ang. JSON Schema) lub też analogiczne w formacie YML.



Rysunek 12 Schemat kolekcji transaction

Diagramy ilustrujące model danych dla mikroserwisów Planning oraz Asset management przedstawiają kolejno Rysunek 13 oraz Rysunek 14.

Planning				
balance				
#	_id	pk	str	*
#	balance		dec128	*
#	metadata		doc	*
#	version		int64	*
#	createdAt		date	*
#	modifiedAt		date	*
#	_class		str	

category				
#	_id		old	*
#	username		str	*
#	name		str	*
#	type		str	*
#	transactionType		str	*
#	metadata		doc	*
#	version		int64	*
#	createdAt		date	*
#	modifiedAt		date	*
#	_class		str	

categoryUsageLimit				
#	_id		old	*
#	username		str	*
#	categoryName		str	*
#	usage		dec128	*
#	yearMonth		str	*
#	metadata		doc	*
#	version		int64	*
#	createdAt		date	*
#	modifiedAt		date	*
#	_class		str	

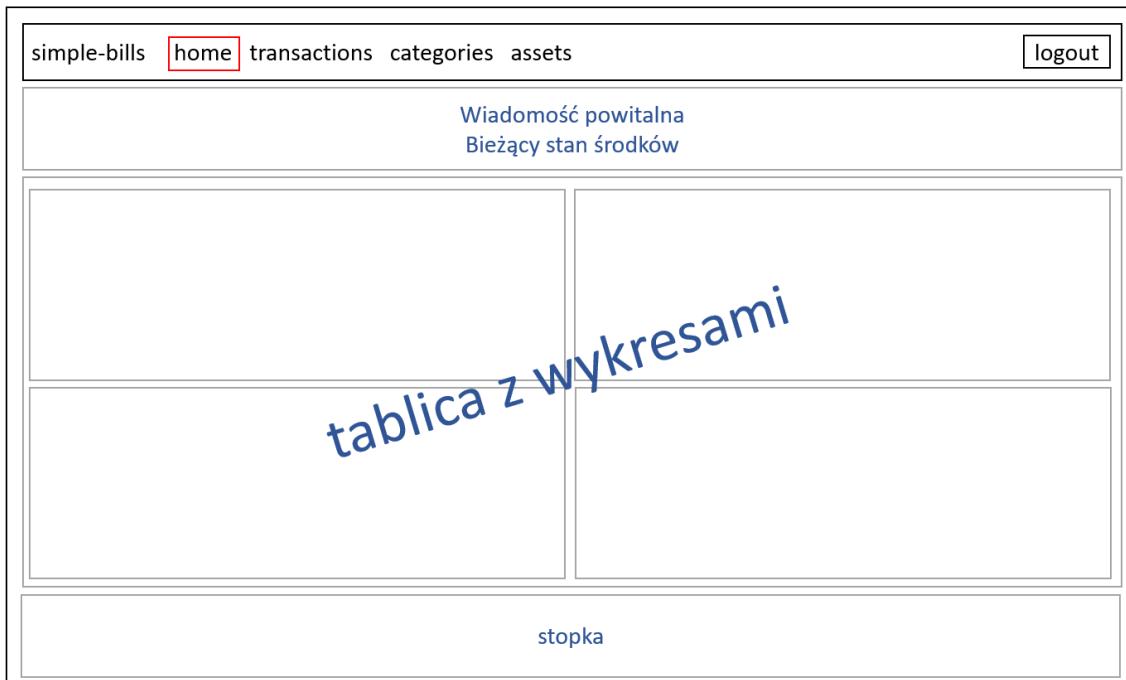
Rysunek 13 Model bazy danych mikroserwisu Planning

Asset Management				
deposit				
#	_id		old	*
#	username		str	*
#	name		str	*
#	value		dec128	
#	depositType		str	*
#	bankName		str	*
#	durationInMonths		int32	
#	annualInterestRate		dec128	
#	metadata		doc	*
#	version		int64	*
#	createdAt		date	*
#	modifiedAt		date	*
#	_class		str	

Rysunek 14 Model bazy danych mikroserwisu Asset Management

5.7 Projekt interfejsu użytkownika

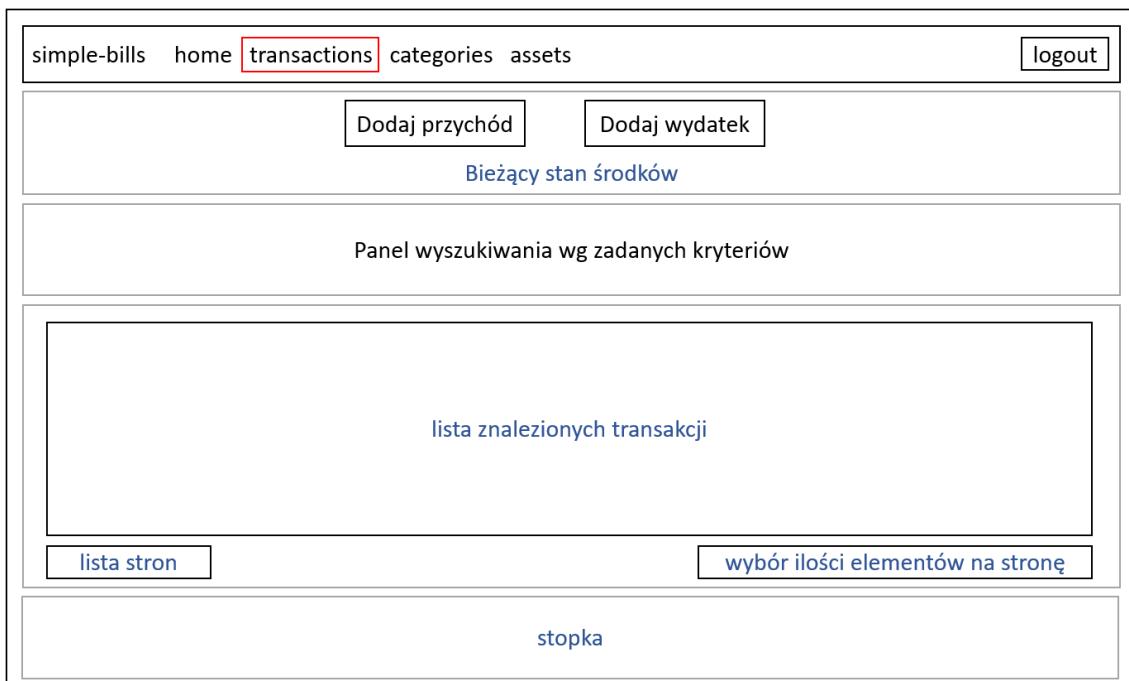
Aplikacja będzie posiadała klasyczny interfejs użytkownika z menu umieszczonym na belce w górnej części ekranu, środkiem ekranu przeznaczonym na wyświetlanie wszystkich informacji oraz stopką u dołu ekranu. Zawartość menu będzie różna dla niezalogowanego i zalogowanego użytkownika. Cały interfejs aplikacji zgodnie z wymaganiami określonymi w punkcie 5.1.2 będzie w języku angielskim. Niezalogowany użytkownik zobaczy wyłącznie zakładkę home oraz znajdujący się w prawym górnym rogu przycisk logowania. Po zalogowaniu natomiast pojawią się dodatkowe zakładki transaction, categories, assets oraz links, a przycisk logowania zamieni się na przycisk wylogowania. Projekt ekranu głównego home przedstawia Rysunek 15.



Rysunek 15 Projekt interfejsu użytkownika – strona główna

Środkowa część ekranu, służąca do wyświetlania danych w przypadku ekranu transakcji podzielona zostanie na 3 części. Część górną będzie zawierać przyciski pozwalające na dodawanie nowych transakcji (przychodów i wydatków). W części tej wyświetlany będzie także bieżący stan środków pieniężnych. Poniżej znajdować się będzie panel umożliwiający wyszukiwanie transakcji według zadanych kryteriów. Pod panelem wyszukiwania będzie znajdować się tabela zawierająca listę transakcji spełniających kryteria. Pod tabelą z kolei umieszczona

zostanie lista z wyborem stron oraz rozwijana lista pozwalająca na wybór ilości elementów na stronie. Projekt ekranu transakcji przedstawia Rysunek 16.



Rysunek 16 Projekt interfejsu użytkownika – ekran transakcji

Po kliknięciu przycisku dodawania nowych transakcji (przychodów lub wydatków) otwarte zostanie nowe okno modalne zawierające formularz oraz przyciski umożliwiające anulowanie operacji oraz przycisk potwierdzający dodanie transakcji. Przykładowe okno modalne z formularzem dodania transakcji przedstawia Rysunek 17. Pozostałe okna modalne zostały zaprojektowane w ten sam sposób – różnią się jedynie rodzajem i ilością pól formularza.

Ostatnie dwa ekranы to ekran kategorii (Rysunek 18) oraz ekran aktywów. Zostały one zaprojektowane w identyczny sposób. Część środkowa przeznaczona na informacje do wyświetlenia, w przypadku tych ekranów została podzielona na dwie części – część górną zawierającą przyciski dodawania kategorii lub dodawania aktywów oraz część dolną służącą do wyświetlenia tabeli kategorii czy też aktywów w przypadku ostatniego z ekranów.

Dodaj nowy przychód

Kategoria

Opis

Kwota

Data

Anuluj Dodaj

Rysunek 17 Projekt interfejsu użytkownika – okno modalne formularza

simple-bills home transactions categories assets [logout](#)

[Dodaj kategorie przychodów](#) [Dodaj kategorie wydatków](#)

lista kategorii

stopka

Rysunek 18 Projekt interfejsu użytkownika – ekran kategorii

6 Implementacja systemu

W niniejszym rozdziale przedstawiono opis implementacji systemu wraz z wybranymi szczegółami, w tym:

- opis budowy mikroserwisów zawierający diagramy klas,
- opis rozwiązań wykorzystanych przy budowie mikroserwisów,
- opis budowy aplikacji klienckiej zawierający diagramy klas,
- opis sposobu zarządzania tożsamością użytkowników i procesem uwierzytelniania z wykorzystaniem protokołu OpenID Connect oraz OAuth 2.0 i zewnętrznej aplikacji Keycloak.

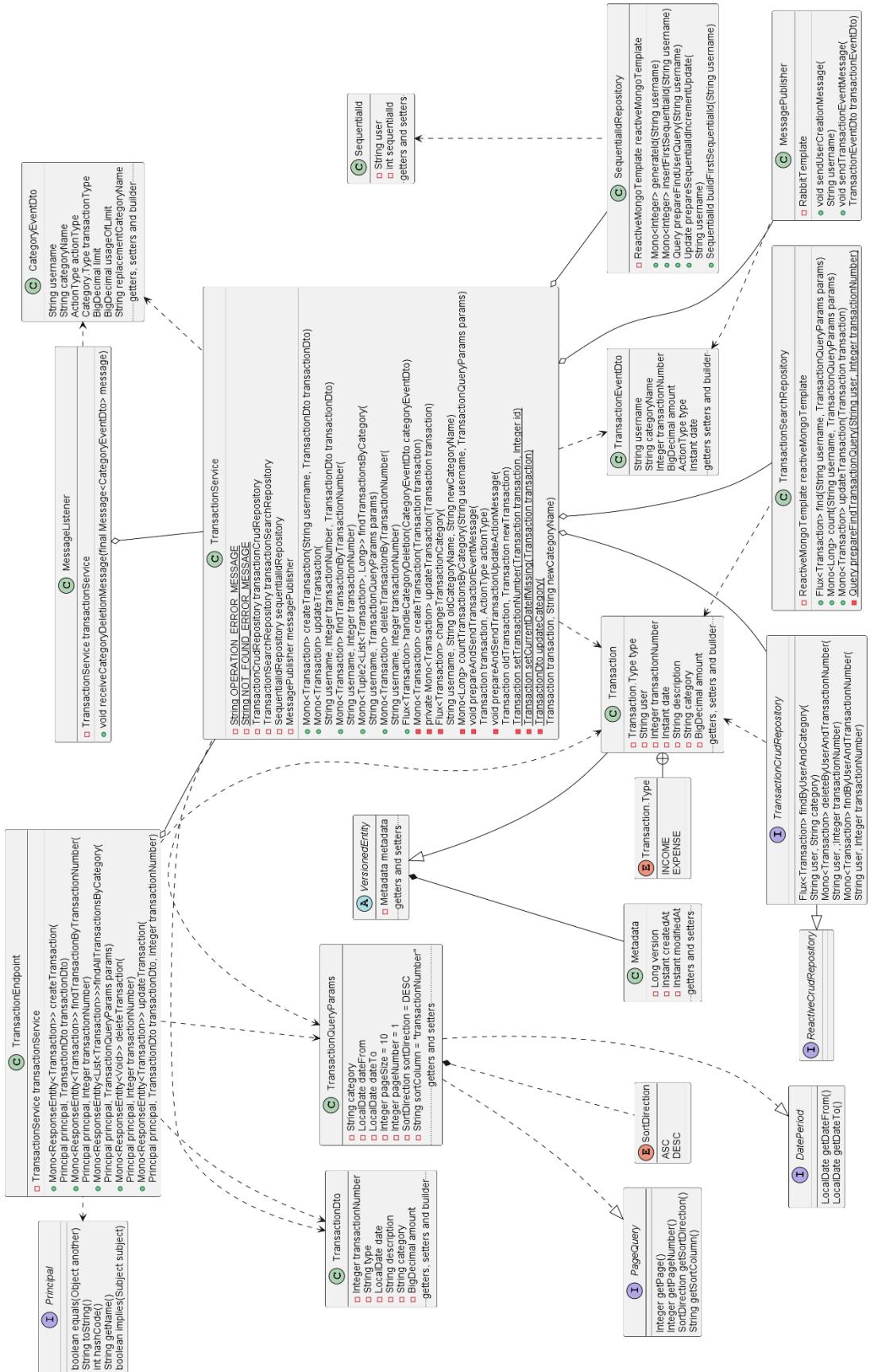
Podczas tworzenia systemu wykorzystano system kontroli wersji Git, a cały kod źródłowy mikroserwisów oraz aplikacji klienckiej umieszczony został w repozytorium na platformie Gitlab: <https://github.com/lukaszse/simple-bills>.

6.1 Budowa mikroserwisów

Wszystkie trzy mikroserwisy zostały napisane w języku Java w wersji 17 z wykorzystaniem bibliotek Spring Boot. Mikrousługi posiadają trójwarstwową strukturę, opisaną szczegółowo w paragrafie 5.3.

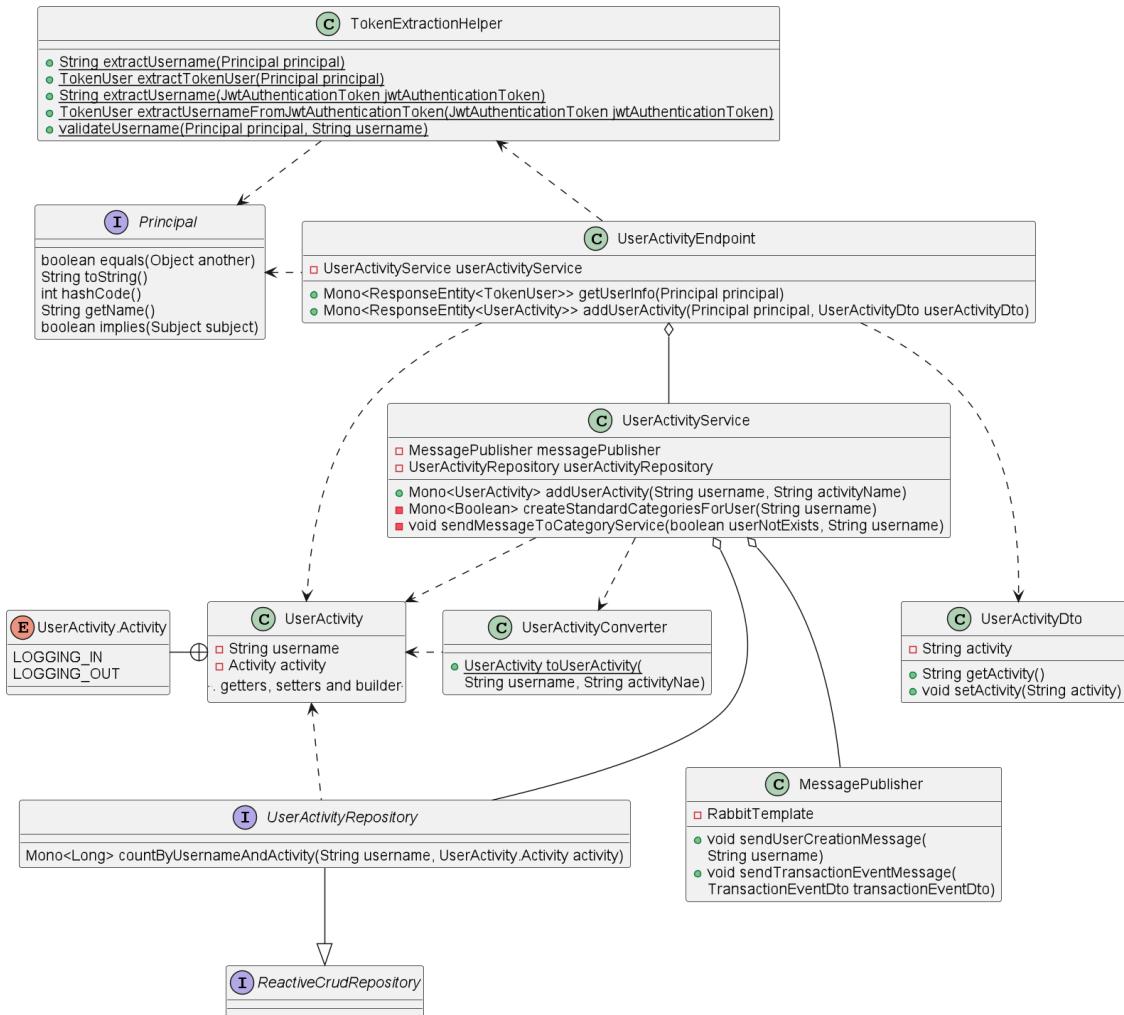
Sprint Boot jest narzędziem które umożliwia m.in. wstrzykiwanie zależności. Wykorzystuje on wzorzec projektowy „odwrócenie sterowania” (ang. Inversion of Control - IoC), dzięki czemu zapewnia kontener w którym w sposób automatyczny tworzy instancje wybranych (odpowiednio skonfigurowanych) obiektów. Pozwala to uprościć kod aplikacji po przez oddzielenie kodu odpowiedzialnego za realizację logiki biznesowej, od kodu odpowiedzialnego za tworzenie obiektów.

Rysunek 19 przedstawia diagram klas dla pionu odpowiedzialnego za obsługę transakcji (przychodów i wydatków) serwisu Transaction management. Przedstawione w niniejszym rozdziale diagramy klas zawierają większość klas odpowiedzialnych za realizację logiki biznesowej, w celu polepszenia widoczności pominięto jednak szereg składników będących elementami środowiska Spring Boot. W rzeczywistości Spring Boot tworzy szereg dodatkowych obiektów, które odpowiedzialne są m.in. za konfigurację, obsługę bazy danych czy też obsługę funkcji związanych z bezpieczeństwem.



Rysunek 19 Diagram klas mikroserwisu Transaction management odpowiedzianych za realizację obsługi transakcji

Dla polepszenia widoczności pominięto również niektóre klasy zawierające statyczne metody narzędziowe odpowiedzialne m.in. za wykonywanie operacji konwersji czy też obsługę tokenów autoryzacyjnych (jak np. metodę odpowiedzialną za ekstrakcję nazwy użytkownika z przychodzącego tokenu typu Basic lub Bearer).



Rysunek 20 Diagram klas mikroserwisu Transaction Management odpowiedzialnych za rejestracje aktywności użytkownika

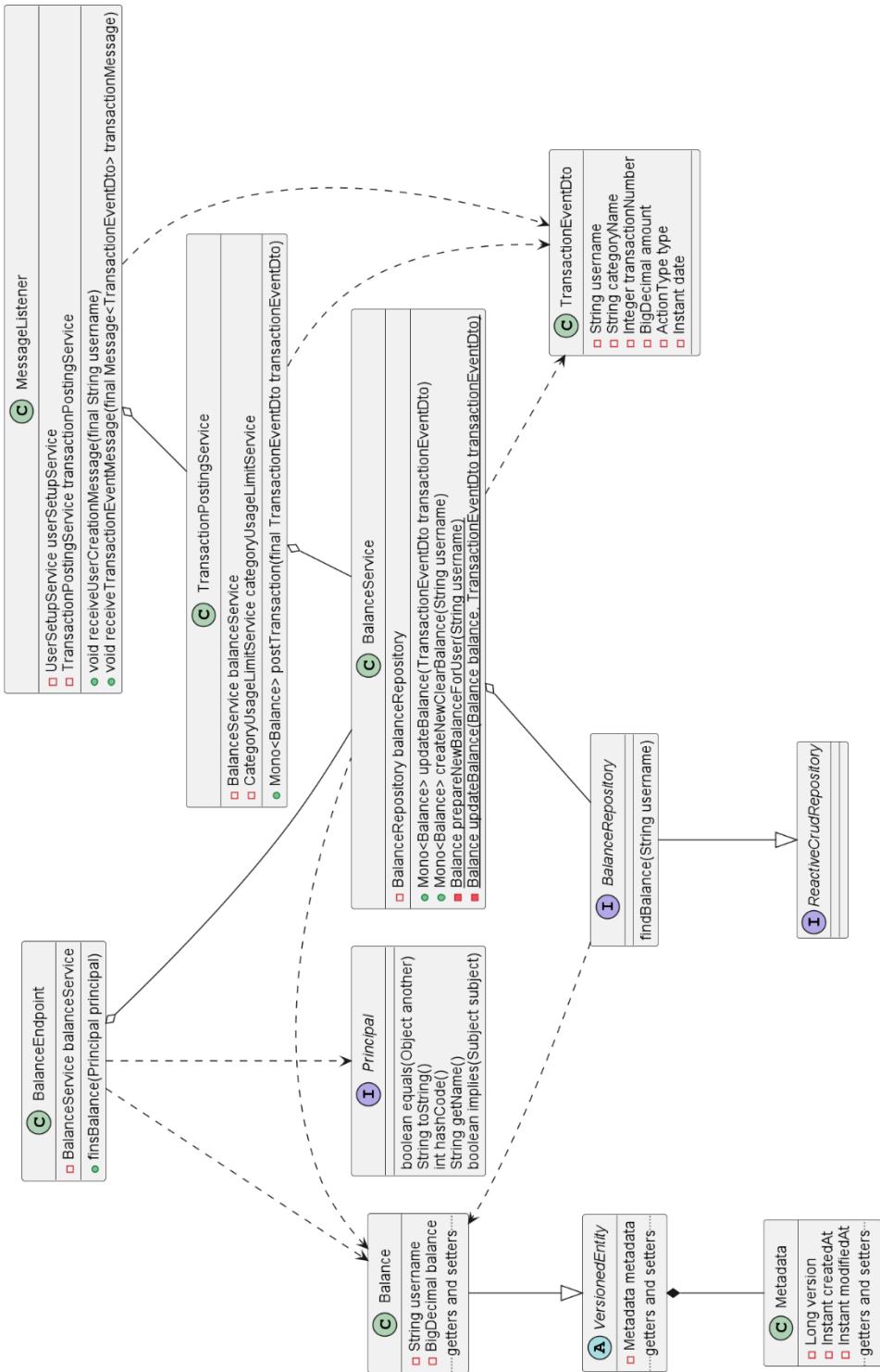
Mikroserwis Transaction management posiada jeszcze jedną odpowiedzialność jaką jest rejestracja aktywności użytkowników. Rysunek 20 przedstawia diagram klas odpowiedzialnych za realizację związanych z tym funkcji. Należy tutaj jednak wyraźnie zaznaczyć, że odpowiedzialność ta w zasadzie powinna być realizowana przez odrębny serwis. Serwis taki oprócz funkcji związanych z rejestracją aktywności użytkowników mógłby służyć do ogólnego zarządzania użytkownikami, w tym na przykład przypisywania

dostępu dla poszczególnych użytkowników do konkretnych funkcji systemu, w zależności od wykupionego abonamentu lub też do usuwania użytkowników, w tym usuwania wszystkich danych usuwanych użytkowników z baz danych wszystkich pozostałych mikroserwisów. Mikroserwis do zarządzania użytkownikami powinien ponadto posiadać swój interfejs użytkownika, np. w postaci niezależnej aplikacji działającej w przeglądarce. Dla uproszczenia systemu funkcję służącą do rejestracji aktywności użytkowników zaimplementowano w serwisie Transaction management. Funkcja ta jest niezbędna dla prawidłowego działania systemu, pozwala ona bowiem na weryfikację czy dany użytkownik istnieje już w systemie, a w przypadku nowych użytkowników umożliwia przygotowanie konfiguracji wstępnej, w tym utworzenie domyślnych kategorii wydatków dla nowego użytkownika.

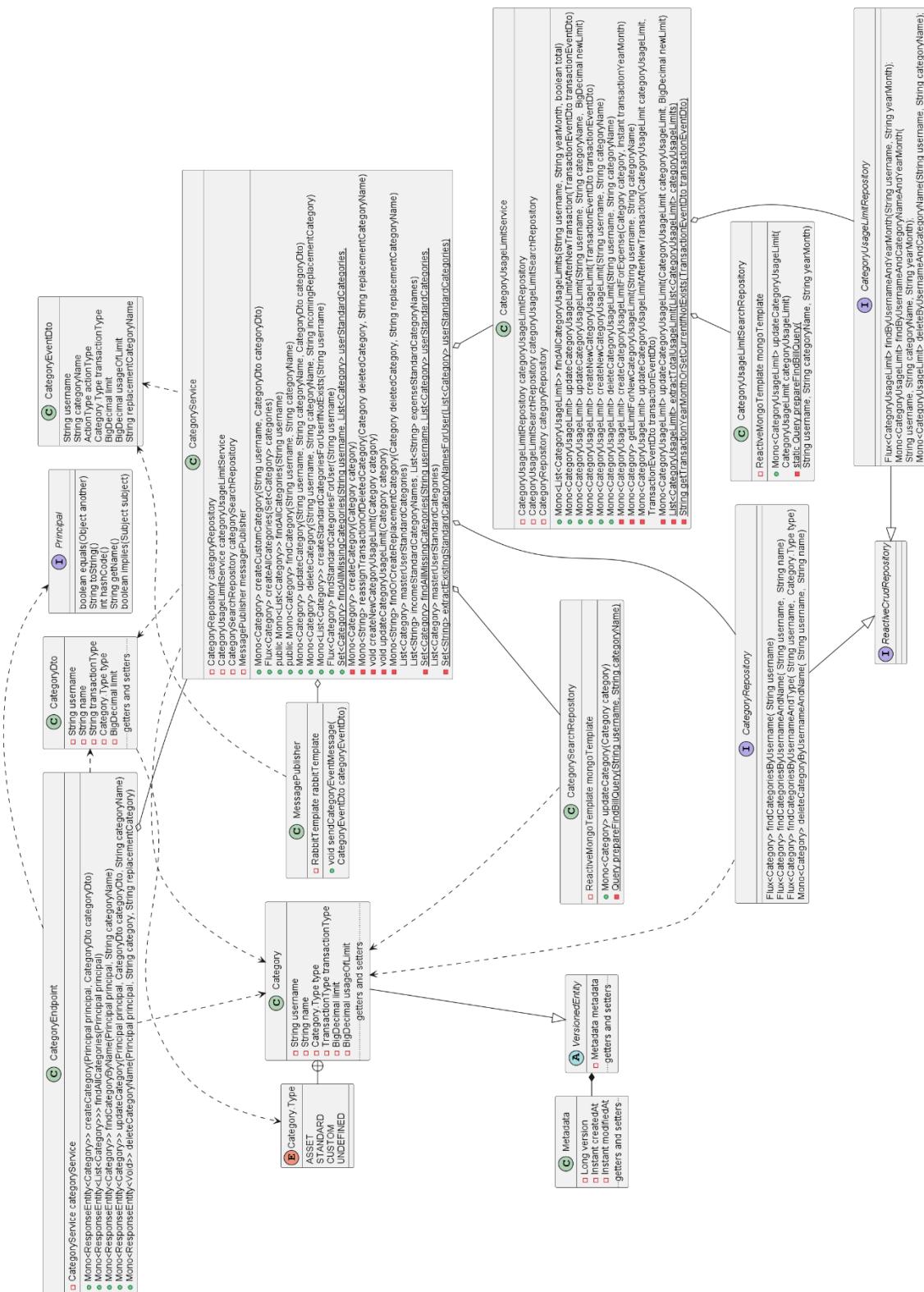
Kolejną z mikrousług jest serwis Planning odpowiedzialny za funkcje związane z planowaniem, w tym za zarządzanie kategoriami wydatków, obsługę bieżącego stanu konta oraz wykorzystania limitów dla poszczególnych kategorii. Mikroserwis ten posiada identyczną strukturę jak mikroserwis Transaction management. Kolejne trzy rysunki przedstawiają diagramy klas odpowiedzialnych za funkcje związane z obsługą stanu bieżącego środków (Rysunek 21), zarządzanie kategoriami wydatków (Rysunek 22) oraz obsługę wykorzystania limitów przypisach dla poszczególnych kategorii wydatków (Rysunek 23).

Ostatnim z mikroserwisów posiadającym identyczną strukturę jak dwa poprzednie jest serwis Asset management odpowiedzialny za obsługę funkcji związanych z tworzeniem, edycją i usuwaniem aktywów. Dla uproszczenia aplikacji skupiono się jedynie na jednym rodzaju aktywów jakim będą lokaty i depozyty bankowe. Diagram klas dla mikroserwisu Asset management przedstawia Rysunek 24.

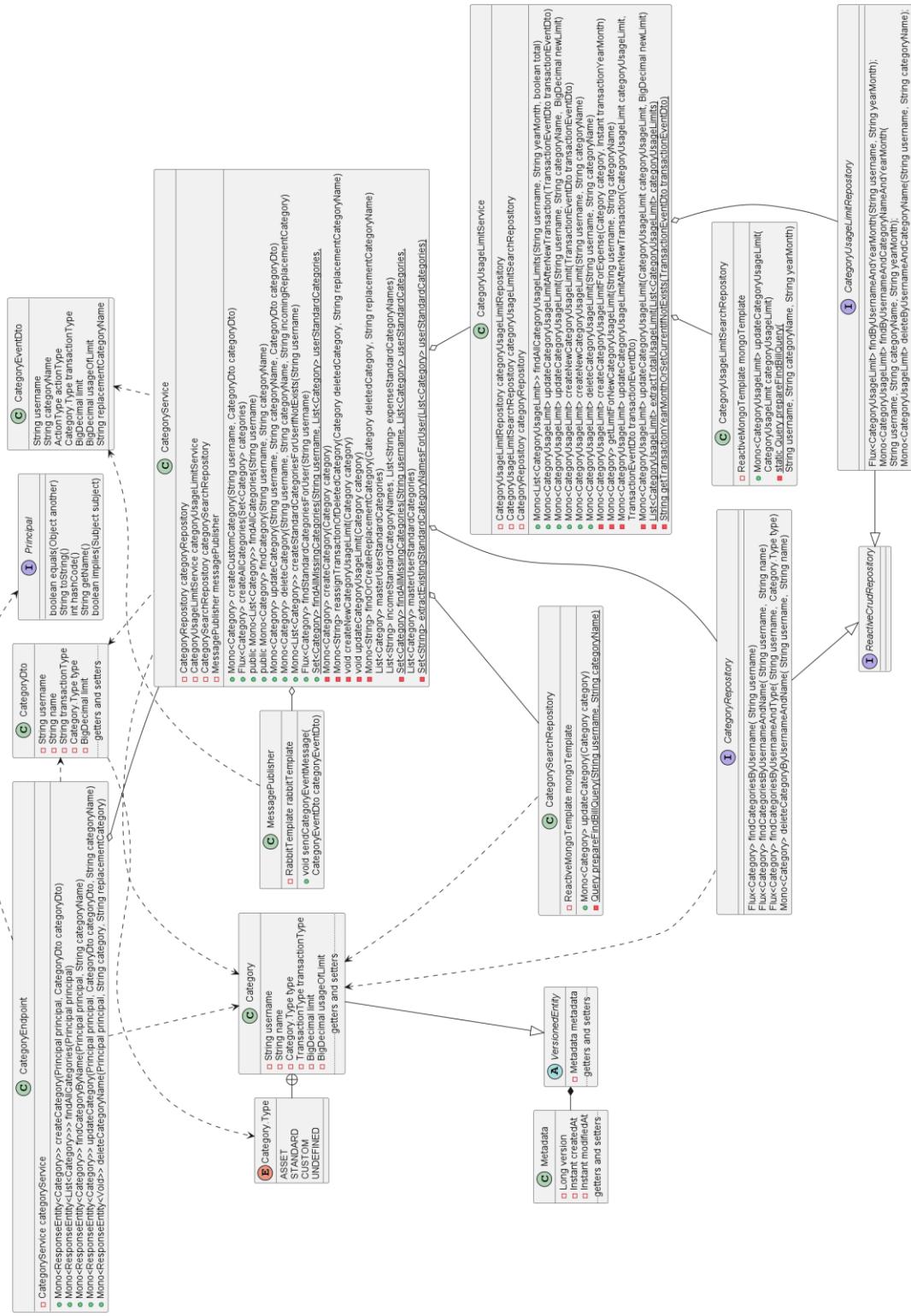
Wszystkie diagramy klas zostały przygotowane w formacie PlantUML [16] umożliwiającym tworzenie diagramów UML w postaci plików tekstowych, na podstawie których renderowane są graficzne schematy. Pliki źródłowe diagramów klas zostały umieszczone w repozytorium projektu na platformie Gitlab: <https://github.com/lukaszse/simple-bills/tree/master/diagrams/class-diagrams>.



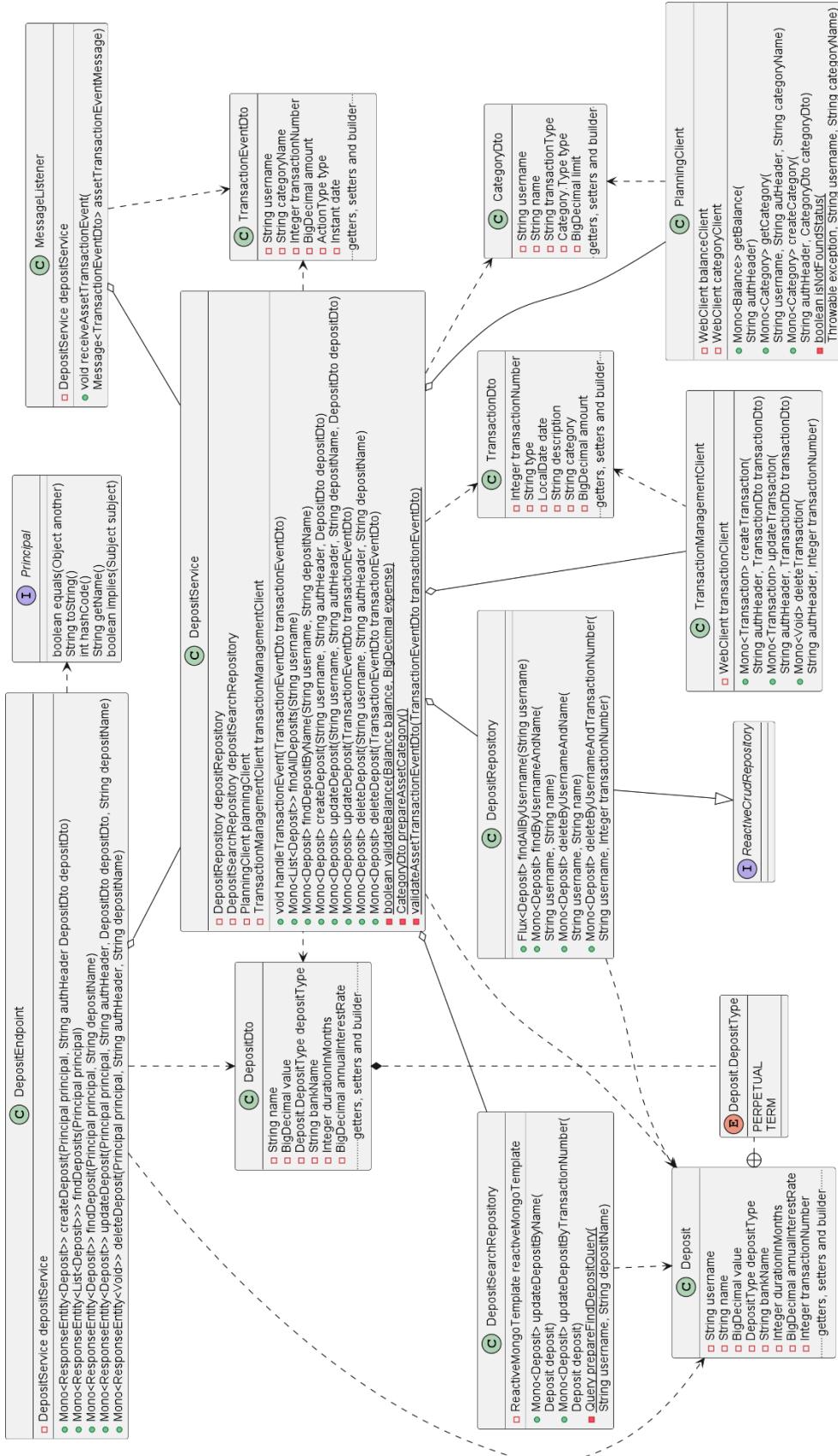
Rysunek 21 Diagram klas mikroserwisu Planning odpowiedzialnych za przetwarzanie stanu środków bieżących (balance)



Rysunek 22 Diagram klas mikroserwisu Planning odpowiedzialnych za obsługę kategorii wydatków



Rysunek 23 Diagram klas mikroserwisu Planning odpowiedzialnych za zarządzanie obsługą wykorzystania limitów wydatków dla poszczególnych kategorii



Rysunek 24 Diagram klas mikroserwisu Asset management odpowiedzialnych za obsługę depozytów

6.2 Rozwiązania wykorzystane w mikroserwisach

W mikroserwisach wykorzystano wiele technologii i rozwiązań wartych dodatkowego opisu. Poniżej przedstawiono kilka z nich.

6.2.1 Biblioteki reaktywne i programowanie funkcyjne

Biblioteki reaktywne Project Reactor dostępne w ramach pakietu Spring WebFlux umożliwiają tworzenie aplikacji działających w sposób wielowątkowy oraz nieblokujący. Reaktywne przetwarzanie danych usuwa w praktyce konieczność oczekiwania wątków na wynik obliczeń czy też oczekiwania na odpowiedzi przychodzące z zewnętrznych systemów. Jest to możliwe dzięki implementacji wzorca projektowego obserwator [17]. Biblioteki reaktywne umożliwiają również pisanie kodu w sposób funkcyjny, gdzie dane są przetwarzane w strumieniach. Reactor obsługuje dwa typy strumieni Mono, który może wyemitować 0 lub 1 element, oraz Flux który wyemitować 0 lub wiele elementów. Elementem podobnie jak w przypadku kolekcji w języku Java, może być dowolny obiekt.

Poniżej przedstawiono przykład metod napisanej z wykorzystaniem bibliotek reaktywnych, odpowiadającej za realizację przypadku użycia „Dodaj depozyt”:

```
1 public Mono<Deposit> createDeposit(final String username,
2                                     final String authHeader,
3                                     final DepositDto depositDto) {
4     return planningClient.getBalance(authHeader)
5         .filter(balance -> validateBalance(balance, depositDto.getValue()))
6         .then(planningClient.getCategory(username, authHeader,
7             Category.Type.ASSET.toString().toLowerCase())
8             .switchIfEmpty(planningClient.createCategory(authHeader,
9                 prepareAssetCategory())))
10        .then(transactionManagementClient.createTransaction(authHeader,
11            toTransactionDto(depositDto)))
12        .map(depositTransaction -> toDeposit(username,
13            depositTransaction.getTransactionNumber(), depositDto))
14        .map(VersionedEntityUtils::setMetadata)
15        .flatMap(depositRepository::save)
16        .doOnNext(createdDeposit ->
17            log.info("Deposit with name={} and username={} created.",
18                    createdDeposit.getName(),
19                    createdDeposit.getUsername()));
20 }
```

Powyższy kod w pierwszej kolejności pobiera bieżący stan konta z serwisu Planning z wykorzystaniem reaktywnego klienta HTTP planningClient (linia 4), następnie wywołuje metodę validateBalance(), która sprawdza

czy użytkownik posiada odpowiednie środki do utworzenia depozytu, a w przypadku pozytywnego rezultatu w kolejnym kroku (linia 6) pobierana jest odpowiednia kategoria lub tworzona jest nowa (linia 8) w przypadku jeśli odpowiednia kategoria została znaleziona. Metoda *then* powoduje, że wyemitowany zostanie nowy obiekt Mono, tylko w przypadku jeśli wcześniej nie wystąpił żaden błąd oraz zwrócony wcześniej w strumieniu obiekt Mono nie był pusty. Metoda *then* wykorzystywana będzie w sytuacji kiedy nie ma konieczności przekazania danych z poprzednio wywołanej w strumieniu metody. Metoda *switchIfEmpty* z kolei umożliwia wyemitowanie nowego obiektu Mono, w sytuacji kiedy strumień zwrócił pusty obiekt Mono. Następnie z wykorzystaniem klienta HTTP `transactionManagementClient` (linia 10) tworzona jest nowa transakcja. Odpowiedź klienta będzie wykorzystana do utworzenia nowego obiektu `Deposit` (linia 12), następnie obiekt ten uaktualniany jest o metadane (linia 14) po czym zapisywany jest w bazie danych (linia 15). Następnie wyświetlany jest log informujący o powodzeniu całego łańcucha operacji (linia 17). Podejście reaktywne pozwala nie tylko na efektywniejsze przetwarzanie danych, ale również dzięki programowaniu funkcyjnemu pozwala zachować czystość kodu.

6.2.2 Automatyczne generowanie powtarzalnych składników kodu

Programowanie w języku Java wiąże się z koniecznością tworzenia wielu powtarzalnych składników kodu takich jak konstruktory, gettery czy settery. Często również korzysta się z powtarzalnych składników będących np. implementacją kreacyjnych wzorców projektowych jak np. implementacja wzorca projektowego budowniczy. Z pomocą przychodzi tutaj biblioteka Lombok pozwalająca automatyzować tworzenie powtarzających się elementów kodu z wykorzystaniem adnotacji.

Przedstawiony poniżej fragment kodu, pokazuje możliwość wykorzystania biblioteki Lombok:

```
1 @Data
2 @EqualsAndHashCode(callSuper = true)
3 @Builder
4 @NoArgsConstructor
5 @AllArgsConstructor
6 @Document
7 public class Deposit extends VersionedEntity {
8     // implementacja klasy
9 }
```

Adnotacja `@Data` pozwala m.in. utworzyć automatycznie gettery i settery dla wszystkich pól klasy. Tworzy ona także standardową implementację metody `toString()` a także implementację metod `hashCode()` i `equals()`. Adnotacja `@EqualsAndHashCode` z dodatkowym parametrem `callSuper` ustawionym na `true`, pozwala wywołać metody `hashCode()` i `equals()` w metodzie bazowej. Adnotacja `@Builder` pozwala na szybką implementację wzorca projektowego budowniczy, z kolei `@NoArgsConstructor` tworzy automatycznie konstruktor bezparametry, a `@AllArgsConstructor` konstruktor dla wszystkich pól klasy [18].

6.2.3 Obsługa uwierzytelniania i autoryzacji

W systemie wykorzystano protokół OAuth 2.0 oraz autoryzację typu Authorization Code Flow, w tym uwierzytelnianie z wykorzystaniem warstwy OpenID Connect. Opis wykorzystanego rodzaju autoryzacji i uwierzytelniania, w tym aplikacji Keycloak wykorzystanej system do zarządzania tożsamością i dostępem użytkowników przedstawiono w paragrafie 6.5.

Środowisko Spring Boot automatyzuje proces autoryzacji i uwierzytelniania, pozwalając oddzielić go od logiki biznesowej. Jest to możliwe dzięki zastosowaniu dodatkowych bibliotek Spring Security.

Konfiguracja wszystkich zabezpieczeń znajduje się w dwóch miejscach: w pliku właściwości (`application.properties`) oraz w klasie `SpringSecurity` która odpowiedzialna jest za stworzenie obiektu `SecurityWebFilterChain` zawierającą konfigurację bezpieczeństwa dostępną w trakcie działania aplikacji. W praktyce plik `application.properties` może być rozszerzony o dodatkowe pliki zawierające konfigurację dla wybranego środowiska (np. oddzielną konfigurację dla środowiska lokalnego, deweloperskiego i produkcyjnego) [19]. W tym celu w każdym z mikroserwisów utworzono cztery osobne pliki – `application.properties` zawierający tylko jedną zmienną `spring.profiles.active` służącą do określenia aktywnego środowiska, oraz pliki `application-local.yml`, `application-dev.yml`, `application-test.yml` zawierające konfigurację dla poszczególnych środowisk.

Poniżej przedstawiono zawartość pliku `application-dev.yml` odpowiedzialnego za konfigurację środowiska deweloperskiego, jakim w przypadku niniejszego systemu jest opisana w paragrafie 7.2 platforma Okteto, służąca do uruchamiania aplikacji z wykorzystaniem kontenerów w systemie Kubernetes.

```

1 spring:
2   autoconfigure:
3     exclude:
4       - org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration
5       - org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
6   data:
7     mongodb:
8       database: transaction
9       uri: ${MONGO_URI}
10  security:
11    oauth2:
12      resourceserver:
13        jwt:
14          issuer-uri: ${KEYCLOAK_ISSUER_URI}
15          jwk-set-uri: ${KEYCLOAK_JWK_SET_URI}
16    rabbitmq:
17      host: rabbitmq
18      username: ${RABBITMQ_DEFAULT_USER}
19      password: ${RABBITMQ_DEFAULT_PASS}
20
21 allowed-origin: ${SIMPLE_BILLS_GUI_URL}

```

Zmienna `security.oauth2.resourceserver.jwt.issuer-uri` odnosi się do lokalizacji umożliwiającej sprawdzenie poprawności wystawcy tokenów, którym jest serwer autoryzacji (ang. Authorization server) natomiast `security.oauth2.resourceserver.jwt.jwk-set-uri` odnosi się do lokalizacji zawierającej klucz publiczny (ang. Public key), potrzebny do weryfikacji integralności tokenów. W przypadku zaimplementowanego systemu obie lokalizacje obsługiwane są przez aplikację Keycloak pełniącą funkcję serwera autoryzacji. Zmienne w formacie `${ZMIENNA_SYSTEMOWA}` pozwalają na użycie dostępnych zmiennych systemowych [19].

Fragment klasy `SpringSecurity` odpowiedzialny za konfiguracje funkcji bezpieczeństwa przedstawiono poniżej:

```

1 @Configuration
2 @EnableWebFluxSecurity
3 @RequiredArgsConstructor
4 public class SpringSecurity {
5
6   private static final String[] openApiPaths =
7     {"/api-docs", "/api-docs/*", "/*/swagger-ui/*", "/swagger-ui"};
8
9   @Value("${allowed-origin}")
10  private String allowedOrigin;
11
12  private final Environment environment;
13
14  @Bean
15  public SecurityWebFilterChain securityWebFilterChain(
16    final ServerHttpSecurity http) {
17    final ServerHttpSecurity serverHttpSecurity = http

```

```
18     .authorizeExchange()
19     .pathMatchers(openApiPaths).permitAll()
20     .anyExchange().authenticated()
21     .and()
22     .oauth2ResourceServer()
23     .bearerTokenConverter(bearerTokenConverter())
24     .jwt()
25     .and().and()
26     .cors()
27     .and();
28 return extractActiveProfileName(environment).equals("local") ?
29         serverHttpSecurity
30             .csrf()
31             .disable()
32             .build() :
33         serverHttpSecurity
34             .csrf()
35             .csrfTokenRepository(
36                 CookieServerCsrfTokenRepository.withHttpOnlyFalse())
37             .and()
38             .build();
39 }
40
41 @Bean
42 public CorsConfigurationSource corsConfiguration() {
43     final CorsConfiguration corsConfig = new CorsConfiguration();
44     corsConfig.applyPermitDefaultValues();
45     corsConfig.addAllowedMethod(HttpMethod.POST);
46     corsConfig.addAllowedMethod(HttpMethod.PUT);
47     corsConfig.addAllowedMethod(HttpMethod.PATCH);
48     corsConfig.addAllowedMethod(HttpMethod.DELETE);
49     corsConfig.setAllowedOrigins(List.of(allowedOrigin));
50     final UrlBasedCorsConfigurationSource source =
51         new UrlBasedCorsConfigurationSource();
52     source.registerCorsConfiguration("/**", corsConfig);
53     return source;
54 }
```

Jak widać z powyższego kodu konfiguracja różnić się będzie w zależności od uruchomionego środowiska. W przypadku uruchomienia aplikacji lokalnie, w celu ułatwienia prac deweloperskich wyłączona zostanie obsługa CSFR. Konfiguracja CORS tworzona jest natomiast osobno w postaci obiektu *CorsConfigurationSource*.

6.2.4 Reaktywna obsługa bazy danych MongoDB

Wszystkie mikroserwisy korzystają z bazy danych MongoDB. W celu dostępu do bazy danych zaimplementowano dwa osobne rozwiązania. Oba korzystają z bibliotek reaktywnych Reactor. Pierwszym z nich jest wykorzystanie interfejsu ReactiveCrudRepository oferujący mechanizm pozwalający na łatwe

generowanie zapytań do bazy danych. Poniższy kod przedstawia interfejs TransactionCrudRepository dziedziczący po interfejsie ReactiveCrudRepository.

```
1 @Repository
2 public interface TransactionCrudRepository
3     extends ReactiveCrudRepository<Transaction, String> {
4
5     Flux<Transaction> findByUserAndCategory(
6         final String user, final String category);
7     Mono<Transaction> deleteByUserAndTransactionNumber(
8         final String user, final Integer transactionNumber);
9     Mono<Transaction> findByUserAndTransactionNumber(
10        final String user, final Integer transactionNumber);
11 }
```

Spring Boot automatycznie przetwarza tak przygotowane interfejsy, tworząc automatyczną ich implementację, a następnie inicjuje ich instancje. *ReactiveCrudRepository* oferuje również możliwość tworzenia bardziej zaawansowanych zapytań z wykorzystaniem natywnego języka zapytań MongoDB wykorzystującego obiekty JSON [19]. W implementowanym systemie, w celu pokazania różnorodności dostępnych narzędzi zdecydowano się jednak wykorzystać inny rodzaj obsługi dostępu do bazy danych MongoDB, a mianowicie mechanizm jaki oferuje *ReactiveMongoTemplate*. Obiekt ten oferuje zestaw metod pozwalających na dostęp do bazy danych. Wspiera także tworzenie zaawansowanych zapytań pozwalających na wyszukiwanie, filtrowanie i agregację danych. Poniżej przedstawiono przykładową implementację repozytorium *TransactionSearchRepository* wykorzystującą *ReactiveMongoTemplate*

```
1 @Repository
2 @RequiredArgsConstructor
3 public class TransactionSearchRepository {
4
5     private final ReactiveMongoTemplate mongoTemplate;
6
7
8     public Flux<Transaction> find(final String username,
9             final TransactionQueryParams params) {
10        return mongoTemplate.find(
11            prepareFindByCategoryQueryPageable(username, params),
12            Transaction.class);
13    }
14
15    public Mono<Long> count(final String username,
16                           final TransactionQueryParams params) {
17
18        return mongoTemplate.count(
19            prepareFindByCategoryQuery(username, params),
```

```
20             Transaction.class);
21     }
22
23     public Mono<Transaction> updateTransaction(final Transaction transaction) {
24         transaction.setMetadata(null);
25         return mongoTemplate.findAndModify(
26             prepareFindTransactionQuery(transaction.getUser()),
27             transaction.getTransactionNumber()),
28             preparePartialUpdateQuery(transaction, Transaction.class),
29             new FindAndModifyOptions().returnNew(true),
30             Transaction.class);
31     }
32
33     private static Query prepareFindTransactionQuery(final String user,
34                                                 final Integer transactionNumber)
35     {
36         return new Query()
37             .addCriteria(Criteria.where("user").is(user))
38             .addCriteria(Criteria.where("transactionNumber")
39             .is(transactionNumber));
40     }
41 }
```

6.2.5 Biblioteka wspólna dla wszystkich mikroserwisów

W celu uniknięcia duplikowania się kodu powtarzające się fragmenty wyniesiono do metod statycznych, które z kolei umieszczono we wspólnej dla wszystkich serwisów bibliotece. Do zbudowania biblioteki wykorzystano narzędzie Gradle, a pliki biblioteczne w formacie JAR przechowywane są na platformie Repsy (<https://repsy.io>).

Przykładem takiej metody bibliotecznej jest pokazana powyżej metoda `prepareFindTransactionQuery()` użyta w `TransactionSearchRepository` do tworzenia standardowego zapytania służącego do wyszukiwania w bazie danych transakcji spełniających określone kryteria. Oprócz metod statycznych z powtarzającymi się często w różnych serwisach fragmentami kodu, biblioteka przechowuje również obiekty odpowiadające za model danych oraz obiekty służące do przesyłania danych (Data Transfer Objects). Dzięki temu biblioteka tworzy swego rodzaju wspólny interfejs aplikacyjny (API) wykorzystywany przez wszystkie mikroserwisy składające się na system Simple Bills.

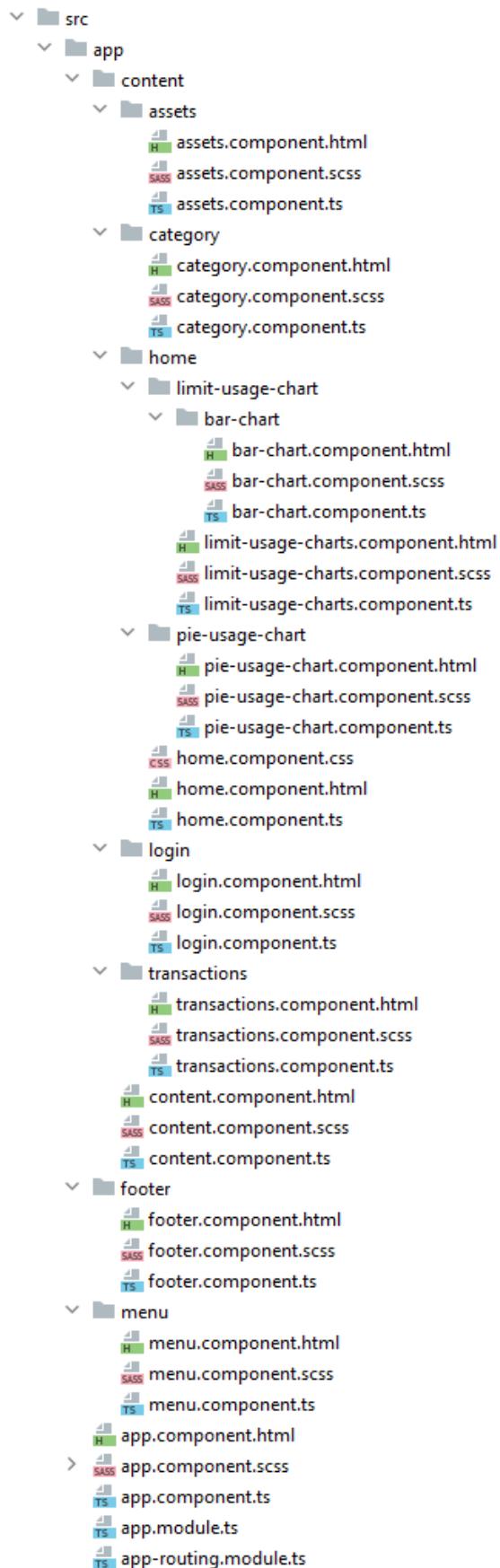
6.3 Budowa aplikacji klienckiej

Aplikację kliencką działającą w przeglądarce internetowej napisano w języku TypeScript z wykorzystaniem zestawu narzędzi i bibliotek Angular. Oferuje

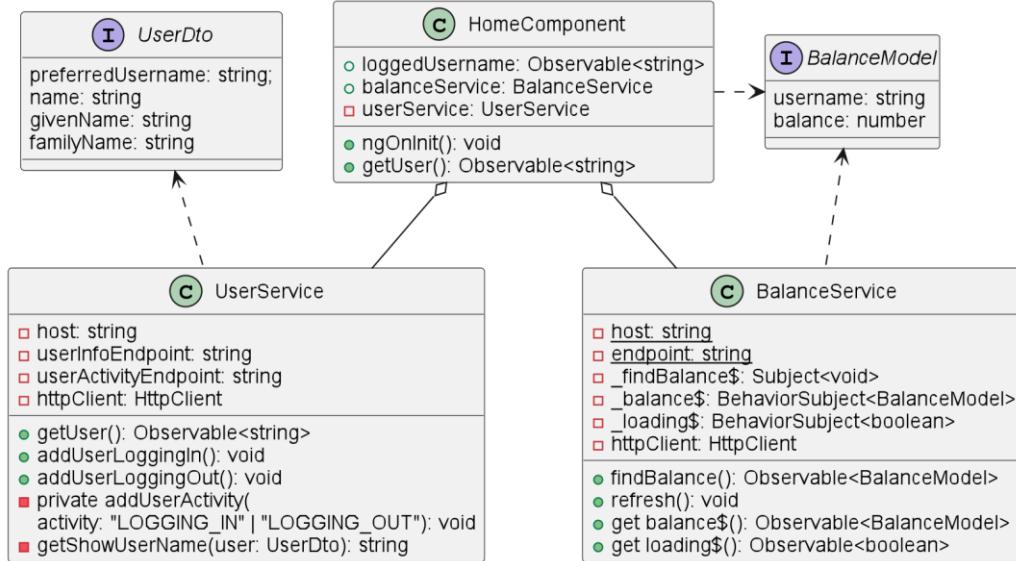
on kompletne środowisko programistyczne, pozwalające na tworzenie aplikacji wykorzystujących wzorzec aplikacji jednostronicowej (ang. Single Page Application - SPA). Podobnie jak Spring Boot, również Angular automatyzuje proces tworzenia obiektów oraz dostarcza mechanizm wstrzykiwania zależności.

Aplikacja posiada budowę charakterystyczną dla aplikacji napisanych z wykorzystaniem Angulara. Składa się z szeregu komponentów, z których każdy odpowiedzialny jest za wyświetlenie konkretnego fragmentu widoku. Każdy z komponentów składa się z pliku HTML, pliku SCSS umożliwiającego definiowanie stylów na poziomie komponentu oraz pliku z rozszerzeniem .ts będącego klasą napisaną w języku TypeScript. Plik HTML nie jest jednak statyczny, może on posiadać odwołania do metod i pól zdefiniowanych wewnątrz klasy przypisanej do danego komponentu. W aplikacji przyjęto standardową konwencję nazewnictwa stosowaną w aplikacjach napisanych z wykorzystaniem Angulara. Rzeczywistą strukturę komponentów zaimplementowaną w aplikacji przedstawia Rysunek 25. Główny komponent aplikacji czyli *app-component* zawiera 3 komponenty: *menu*, *content* i *footer*. Z kolei *content* może być dynamicznie zmieniany dzięki zastosowaniu routingu wspieranego przez biblioteki Angular. Dzięki czemu komponent *content* może w rzeczywistości wyświetlać różne widoki zależnie od wyboru użytkownika. Może to być ekran główny *home* zawierający komponenty z wykresami *pie-usage-chart* oraz *limit-usage-charts*, ekran transakcji, ekran kategorii bądź też ekran aktywów. Poszczególne widoki przełączane są za pomocą menu, stanowiącego (jak wspomniano wyżej) odrębny komponent.

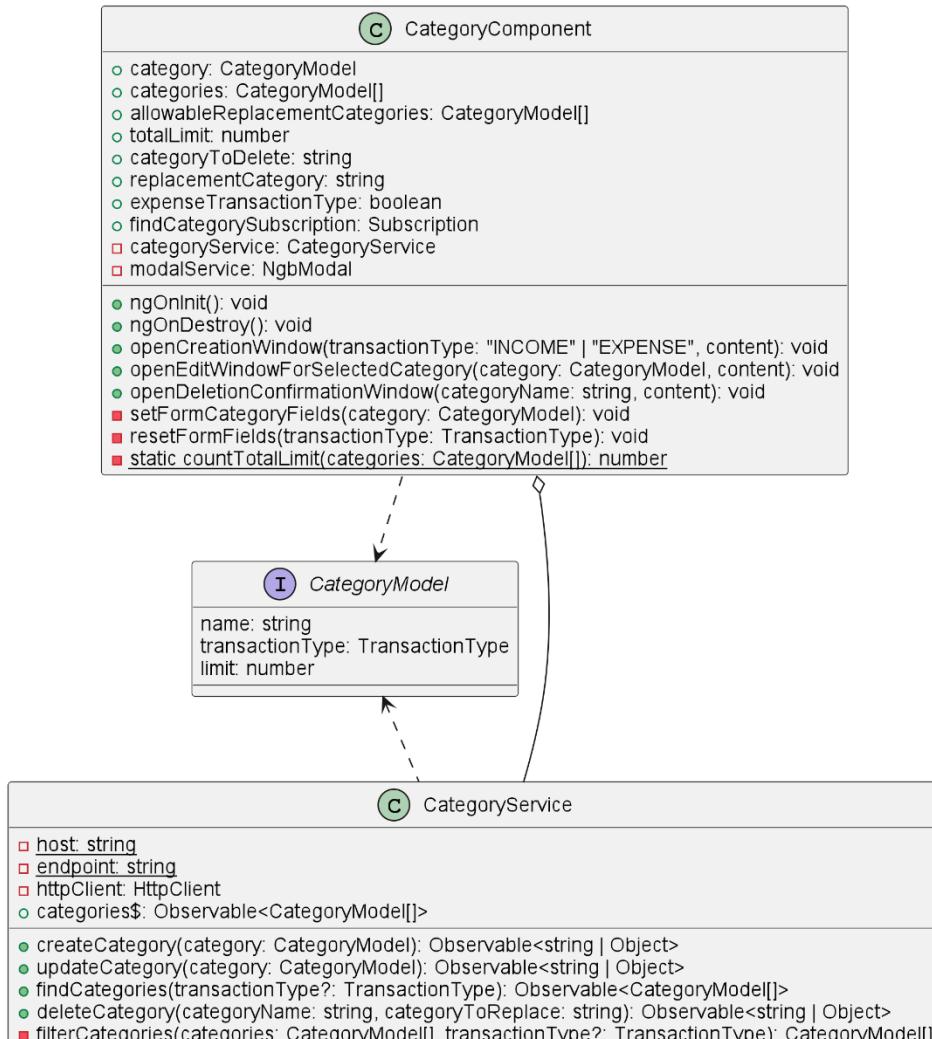
Zadaniem komponentów jest wyłącznie wyświetlanie odpowiednich informacji na ekranie, dlatego logika biznesowa aplikacji delegowana została do serwisów. Pozwala to na zachowanie zasady pojedynczej odpowiedzialności oraz uniknięcie duplikowania się kodu. W aplikacji wykorzystano kilka serwisów. Każdy z nich odpowiedzialny jest za łączność z aplikacją serwerową. Serwisy stanowią więc ogniwo pośrednie pomiędzy komponentami, a aplikacją serwerową – to właśnie w serwisach zaimplementowano reaktywne klienty HTTP umożliwiające komunikację z wykorzystaniem interfejsu REST API.



Rysunek 25 Struktura komponentów aplikacji klienckiej



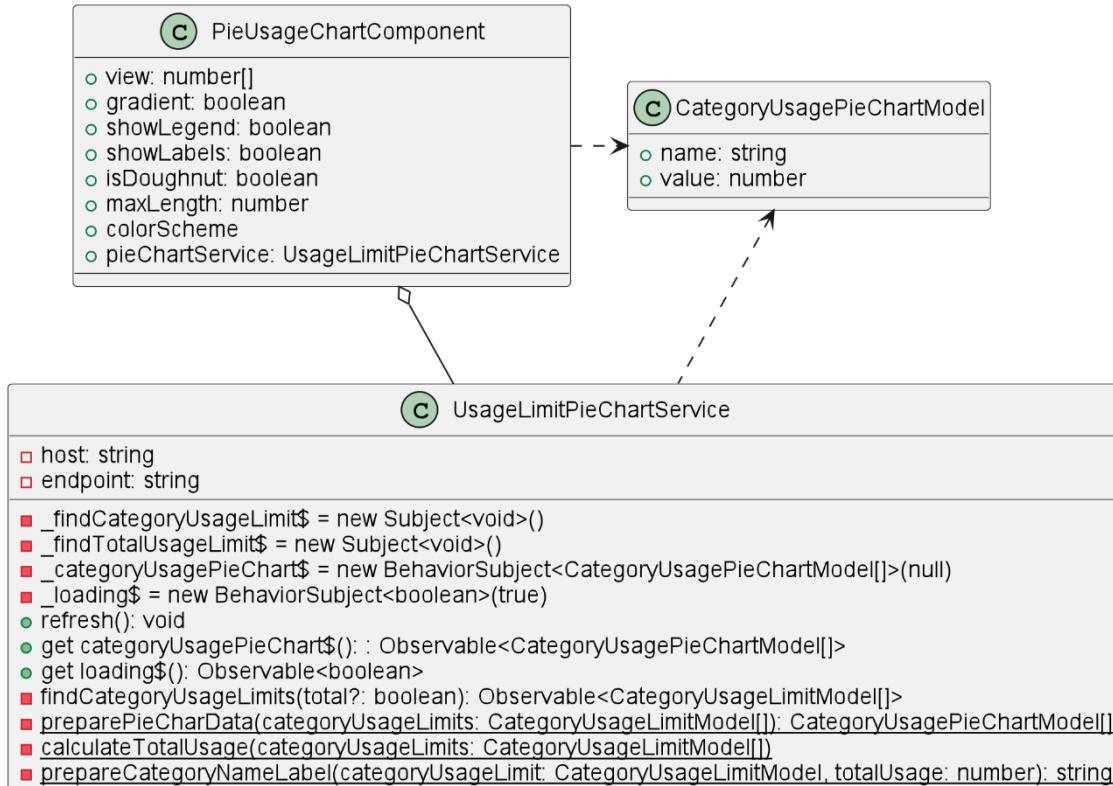
Rysunek 26 Diagram klas powiązanych z komponentem `HomeComponent`



Rysunek 27 Diagram klas powiązanych z komponentem `CategoryComponent`



Rysunek 28 Diagram klas powiązanych z komponentem TransactionComponent



Rysunek 29 Diagram klas powiązanych z komponentem *PieUsageChartComponent*

Na kolejnych rysunkach przedstawiono diagramy klas powiązanych z większością widoków dostępnych w aplikacji klienckiej. Rysunek 26 przedstawia klasy powiązane z komponentem *Home*, Rysunek 27 klasy powiązane z komponentem *Category*, Rysunek 28 klasy powiązane z komponentem *Transactions*, natomiast Rysunek 29 prezentuje klasy powiązane z komponentem *PieUsageChart*. Pominęto natomiast diagramy klas związane z komponentem *Assets*, ponieważ jest on bardzo podobny do komponentu *Transactions*, a także diagramy klas związane z komponentem *LimitUsageCharts*, który z kolei podobny jest do komponentu *PieUsageChart*.

6.4 Rozwiązania wykorzystane w aplikacji klienckiej

W aplikacji klienckiej wykorzystano wiele technologii i rozwiązań wartych dodatkowego opisu. Poniżej przedstawiono kilka z nich.

6.4.1 Reaktywne przetwarzanie danych z wykorzystaniem RxJs

Podobnie jak w przypadku aplikacji mikroserwisowych napisanych z wykorzystaniem bibliotek Spring, również aplikacja przeglądarkowa korzysta z reaktywnego przetwarzania danych. Jest to możliwe dzięki zastosowaniu reaktywnych bibliotek RxJs. Dane podobnie jak w przypadku mikroserwisów przetwarzane są reaktywnie włączając w to reaktywne klienty HTTP. Dzięki takiej implementacji dane są pobierane i przetwarzane w sposób współbieżny. Poniżej zaprezentowano fragment kodu z implementacją zapytania z wykorzystaniem reaktywnego klienta HTTP:

```
1  private findTransactions(pageSize: number,
2                           pageNumber: number,
3                           sortDirection: string,
4                           sortColumn: string,
5                           dateFrom: Date,
6                           dateTo: Date): Observable<PageableTransactionsModel> {
7
8    let url = HttpUtils.prepareUrl(TransactionSearchService.host,
9        TransactionSearchService.endpoint, pageSize, pageNumber, sortDirection,
10       sortColumn, dateFrom, dateTo);
11    return this.httpClient.get<TransactionModel[]>(url,
12      {headers: HttpUtils.prepareHeaders(), observe: 'response'})
13      .pipe(
14        tap(console.log),
15        retry({count: 3, delay: 1000}),
16        map((response) => {
17          return new PageableTransactionsModel(response.body,
18              Number(response.headers.get(HttpUtils.X_TOTAL_COUNT)));
19        }),
20        catchError(HttpUtils.handleError),
21      );
22  }
```

Pokazana wyżej metoda `findTransactions` odpowiedzialna jest za pobranie transakcji z mikroserwisu Transaction management. Obsługuje ona stronicowanie. Nie zwraca ona jednak listy obiektów bezpośrednio, a obiekt `Observable` który podobnie jak w przypadku bibliotek Reactor, działa analogicznie do obiektu `Flux`, który zwraca strumień obiektów.

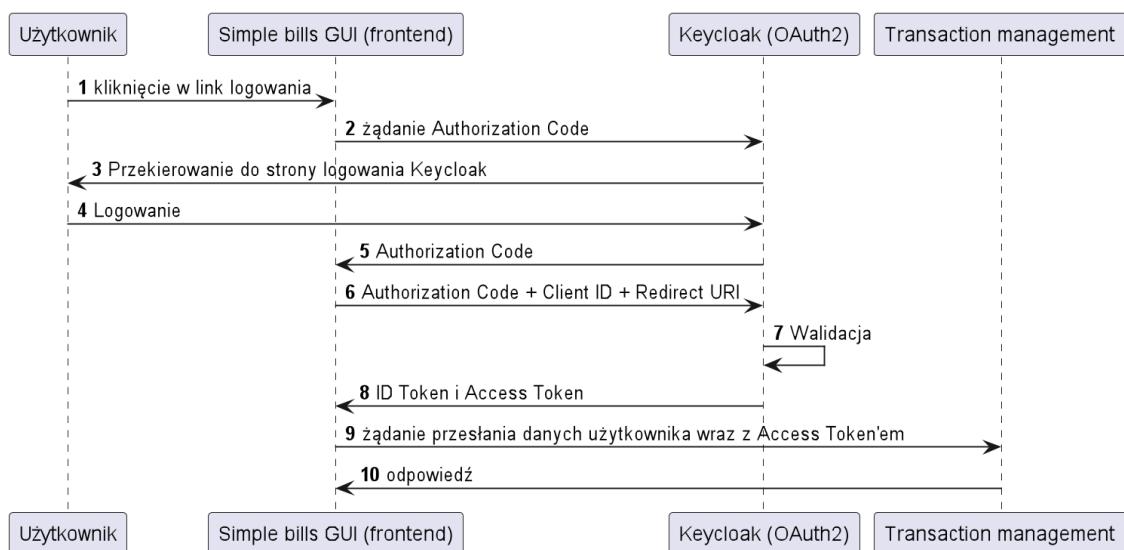
6.4.2 Biblioteka stylów Bootstrap oraz Ng Bootstrap

Aplikacja działająca w przeglądarce korzysta z zestawu bibliotek stylów SCSS Bootstrap. Zawiera ona narzędzia pomocne przy tworzeniu interfejsu graficznego użytkownika. Ponadto aplikacja wykorzystuje biblioteki Ng Bootstrap, przeznaczone dla Angulara. Pomagają one tworzyć dynamiczne elementy

graficzne takie jak listy elementów czy okna modalne wykorzystane w niniejszej aplikacji.

6.5 Uwierzytelnianie i autoryzacja

Dostęp do wszystkich funkcji aplikacji dostępny jest wyłącznie dla zalogowanych użytkowników. Dostęp do danych poszczególnych użytkowników jest chroniony dzięki zastosowaniu protokołu OAuth 2.0. Implementacja tego protokołu zrealizowana została dzięki zastosowaniu zewnętrznej aplikacji Keycloak. Umożliwia ona zarządzanie tożsamościami użytkowników oraz zapewnia uwierzytelnianie i autoryzację użytkowników korzystających z aplikacji Simple Bills. OAuth 2.0 jest protokołem umożliwiającym autoryzację. Za proces uwierzytelniania odpowiedzialny jest z kolei protokół OpenID Connect, który w praktyce rozszerza protokół OAuth 2.0 stanowiąc jego najwyższą warstwę. System wykorzystuje rodzaj autoryzacji typu Authorization Code Flow (Rysunek 30) dostępnej w ramach protokołu OAuth 2.0.



Rysunek 30 Diagram sekwencji przedstawiający przepływ Authorization Code Flow [20] [21]

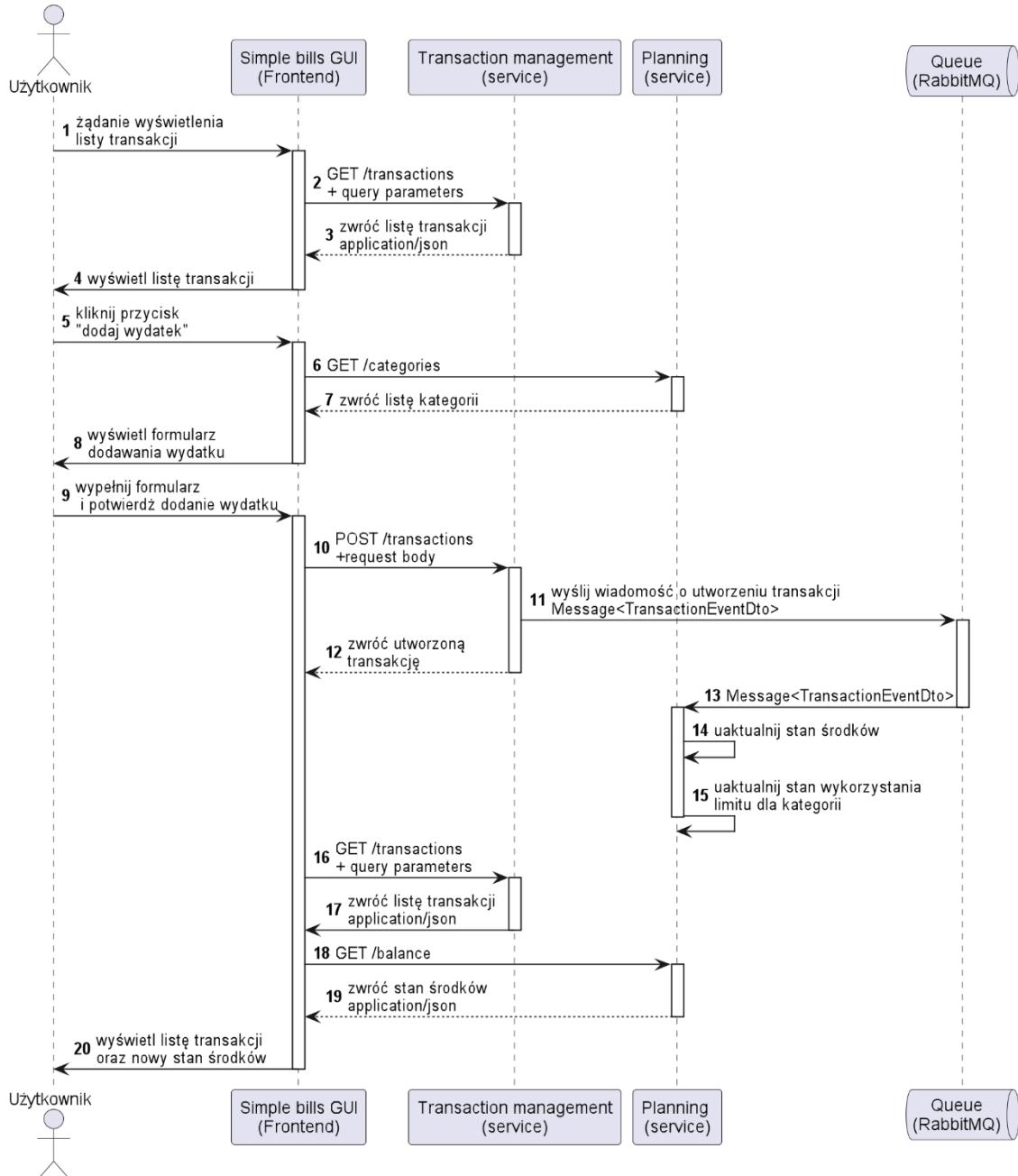
6.6 Komunikacja pomiędzy komponentami systemu

Komunikacja pomiędzy aplikacją kliencką, a mikroserwisami odbywa się w sposób synchroniczny, z wykorzystaniem interfejsu REST API. Klienty HTTP zarówno w aplikacji klienckiej jak i w mikroserwisach są zaimplementowane w sposób reaktywny. Komunikacja pomiędzy mikroserwisami odbywa się

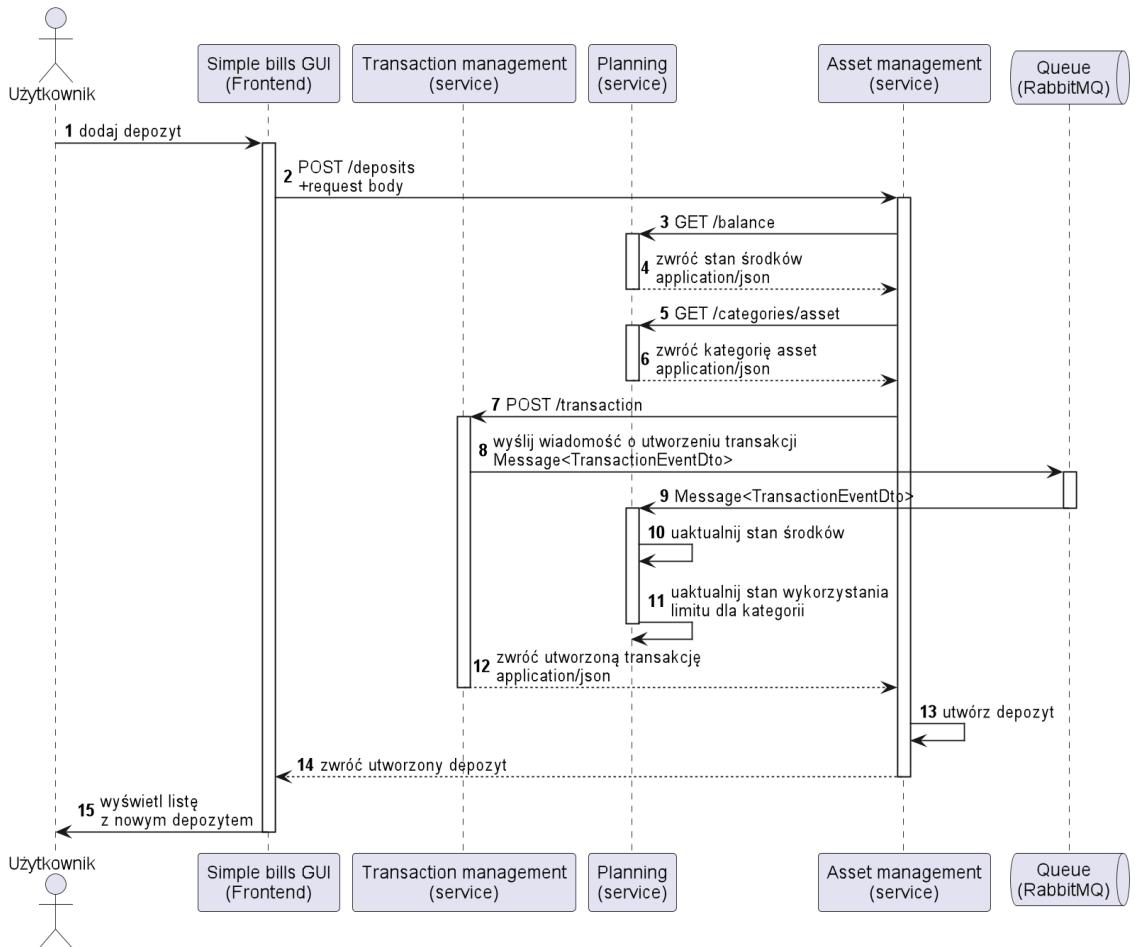
natomiast na dwa sposoby, zarówno synchronicznie jak i asynchronicznie z wykorzystaniem wiadomości i brokera RabbitMQ. Komunikacja synchroniczna w tym przypadku wykorzystywana jest tylko w sytuacjach, kiedy niezbędne jest natychmiastowe zwrócenie odpowiedzi przez serwis do którego zostało wysłane zapytanie.

W celu zilustrowania sposobu komunikacji pomiędzy elementami systemu przygotowano diagram sekwencji odpowiadający scenariuszowi przypadku użycia „Dodaj przychód/wydatek”, który prezentuje Tabela 6 (paragraf 5.1.4). Schemat ten przedstawia Rysunek 31. Alternatywne rozwiązanie gdzie komunikacja pomiędzy mikroserwisami odbywa się częściowo synchronicznie, prezentuje Rysunek 32, przygotowany na podstawie przypadku użycia „Dodaj depozyt”. W tym przypadku mikroserwis Asset management po otrzymaniu żądania utworzenia nowego depozytu, wykonuje serie synchronicznych operacji w celu pobrania bieżącego stanu środków, pobrania odpowiedniej kategorii oraz utworzenia transakcji. Serwis Transaction management w tym przypadku oprócz synchronicznej komunikacji z mikroserwisem Asset management, komunikuje się w sposób asynchroniczny z serwisem Planning, tak jak to miało miejsce w poprzednim przykładzie. W paragrafie 6.2.1 przedstawiony jest z kolei fragment kodu odpowiadający przypadkowi użycia „Dodaj depozyt”.

W obu przedstawionych schematach sekwencji pominięto kwestie uwierzytelniania, w tym komunikację z zewnętrznym serwisem Keycloak. Dla zachowania przejrzystości schematów pominięta została także logika wewnętrzna aplikacji klienckiej i mikroserwisów, w tym komunikacja z bazą danych.



Rysunek 31 Diagram sekwencji przedstawiający przepływ danych pomiędzy komponentami odpowiadający scenariuszowi „Dodaj przychód/wydatek”



Rysunek 32 Diagram sekwencji przedstawiający przepływ danych pomiędzy komponentami odpowiadający przypadkowi użycia „Dodaj depozyt”

6.7 Interfejs oferowany przez mikroserwisy

Interfejs oferowany przez mikroserwisy może stanowić odrębny pełnowartościowy produkt w ramach wspomnianego w rozdziale 4 modelu headless software. Aby taki interfejs mógł być użyteczny dla użytkowników, musi być dobrze udokumentowany. W zaimplementowanej aplikacji postanowiono wykorzystać do tego celu format OpenAPI. Aby uniknąć ręcznego tworzenia dokumentacji skorzystano z biblioteki Springdoc OpenAPI. Umożliwia ona w pełni automatyczne generowanie dokumentacji z kodu aplikacji, który można uzupełnić o dodatkowe elementy dzięki wykorzystaniu adnotacji umieszczonych nad metodami definiującymi punkty końcowe interfejsu REST API. Dla każdego z mikroserwisów tak wygenerowana dokumentacja dostępna jest w lokalizacji /api-docs w formacie JSON, oraz w lokalizacji /swagger-ui w formie graficznej

dzięki interfejsowi Swagger. Poniżej przedstawiono fragment kodu przedstawiającego punkt końcowy POST /transactions umożliwiający utworzenie nowej transakcji. Annotacje `@Operation`, `@ApiResponses`, `@Content` oraz `@Schema` umożliwiają dodanie dodatkowych opisów, które dołączone zostaną do wygenerowanej dokumentacji.

```
1 @Operation(summary = "Create transaction")
2 @ApiResponses(value = {
3     @ApiResponse(responseCode = "201", description = "Transaction created",
4         content = {@Content(mediaType = APPLICATION_JSON_VALUE,
5             schema = @Schema(implementation = Transaction.class))}),
6     @ApiResponse(responseCode = "400", description = "Bad request",
7         content = @Content),
8     @ApiResponse(responseCode = "401", description = "Unauthorized",
9         content = @Content)})
10 @PostMapping(produces = APPLICATION_JSON_VALUE, consumes = APPLICATION_JSON_VALUE)
11 public Mono<ResponseEntity<Transaction>> createTransaction(@AuthenticationPrincipal
12                                         final Principal principal,
13                                         @Valid @RequestBody
14                                         final TransactionDto tDTO)
15 {
16     final String username = TokenExtractionHelper.extractUsername(principal);
17     log.info("Received transaction creation request from user={}", username);
18     return transactionService.createTransaction(username, tDTO)
19         .doOnSuccess(createTransaction ->
20             log.info("Transaction for user={} with number={} successfully created",
21                 createTransaction.getUser(), createTransaction.getTransactionNumber()))
22         .map(transaction -> prepareCreatedResponse(TRANSACTION_URI_PATTERN,
23             String.valueOf(transaction.getTransactionNumber()), transaction));
24 }
```

Rysunek 33 przedstawia widok dokumentacji OpenAPI w wersji graficznej dla pokazanej wyżej implementacji punktu końcowego POST /transactions.

Pełną dokumentację dla poszczególnych mikroserwisów można znaleźć w następujących lokalizacjach³:

- <https://transaction-management-lukaszse.cloud.okteto.net/swagger-ui>
- <https://asset-management-lukaszse.cloud.okteto.net/swagger-ui>
- <https://planning-lukaszse.cloud.okteto.net/swagger-ui>

Tak przygotowana dokumentacja może być dodatkowo uzupełniona w razie potrzeby o informacje związane z uwierzytelnianiem.

³ Mikroserwisy działające na platformie Okteto mogą przejść w stan uśpienia po dłuższym okresie bezczynności. W takim przypadku aby aplikacja zaczęła działać ponownie, należy odświeżyć stronę po upływie kilkudziesięciu sekund.

POST /transactions Create transaction ^

Parameters

No parameters

Request body required application/json ▾

Example Value | Schema

```
{ "transactionNumber": 0, "type": "string", "date": "2022-12-28", "description": "string", "category": "string", "amount": 0 }
```

Responses

Code	Description	Links
201	Transaction created	No links
	Media type	
	application/json ▾	
	Controls Accept header.	
	Example Value Schema	
	{ "metadata": { "version": 0, "createdAt": "2022-12-28T21:36:47.638Z", "modifiedAt": "2022-12-28T21:36:47.638Z" }, "user": "string", "transactionNumber": 0, "type": "INCOME", "date": "2022-12-28T21:36:47.638Z", "description": "string", "category": "string", "amount": 0 }	
400	Bad request	No links
401	Unauthorized	No links

Rysunek 33 Fragment dokumentacji OpenAPI punktu końcowego
POST /transactions

7 Rozwój i wdrożenie aplikacji

W niniejszym rozdziale opisany został sposób realizacji procesów związanych z rozwojem oraz wdrożeniem aplikacji, w tym między innymi sposób prowadzenia prac rozwojowych w środowisku lokalnym, a także opis automatyzacji procesów związanych z wdrożeniem aplikacji z wykorzystaniem platformy Okteto oraz Docker Compose. Wdrożona i działająca na platformie Okteto aplikacja dostępna jest pod adresem: <https://simple-bills-lukaszse.cloud.okteto.net>⁴.

7.1 Ciągła integracja i ciągłe dostarczanie

Ciągła integracja (ang. Continous Integration) i ciągłe dostarczanie (ang. Continous Delivery) – CI/CD to zbiór praktyk, dzięki którym między innymi po przez automatyzacje procesów możliwe jest częstsze i szybsze dostarczanie pewnych przetestowanych zmian w kodzie. Automatyzacja możliwa jest dzięki zastosowaniu automatycznych ciągów operacji (ang. pipeline) wykorzystujących często wiele narzędzi, których celem jest zbudowanie poszczególnych aplikacji, uruchomienie automatycznych testów oraz wdrożenie aplikacji na docelowym środowisku.

7.2 Wdrożenie z wykorzystaniem platformy Okteto

W celu wdrożenia zaimplementowanego systemu postanowiono skorzystać z platformy deweloperskiej Okteto. Oferuje ona możliwość uruchamiania aplikacji internetowych w środowisku Kubernetes, przy czym jej celem nie jest zapewnienie środowiska produkcyjnego, a umożliwienie deweloperowi prowadzenia prac w środowisku możliwie zbliżonym do produkcyjnego, jednak szybszym i łatwiejszym w konfiguracji. Automatyzacja procesu wdrożenia na platformie Okteto możliwa jest m.in. po przez zastosowanie plików w formacie Docker Compose. Plik taki umożliwia konfigurację automatycznego wdrożenia wszystkich elementów systemu, w tym aplikacji klienckiej, mikroserwisów oraz pozostałych składników systemu.

Poniżej przedstawiono plik *docker-compose.yml* służący do wdrożenia na platformie Okteto, wszystkich aplikacji składających się na system Simple Bills:

⁴ Nieużywana przez dłuższy czas aplikacja przechodzi w stan uśpienia. Uśpiona aplikacja wybudza się automatycznie, jednak wymagane jest odświeżenie strony po upływie kilkudziesięciu sekund.

```
1 services:
2   simple-bills:
3     build: simple-bills-gui
4     ports:
5       - 80:80
6     restart: unless-stopped
7     depends_on:
8       - transaction-management
9       - planning
10      - asset-management
11    transaction-management:
12      build: transaction-management
13      ports:
14        - 443:8080
15      restart: unless-stopped
16      depends_on:
17        - rabbitmq
18    env_file:
19      - common_env.env
20  planning:
21    build: planning
22    ports:
23      - 443:8080
24    restart: unless-stopped
25    depends_on:
26      - rabbitmq
27    env_file:
28      - common_env.env
29  asset-management:
30    build: asset-management
31    ports:
32      - 443:8080
33    restart: unless-stopped
34    depends_on:
35      - rabbitmq
36    env_file:
37      - common_env.env
38  rabbitmq:
39    image: rabbitmq:3-management
40    ports:
41      - 5672:5672
42      - 15672:15672
43    volumes:
44      - rabbitmq_data:/var/lib/rabbitmq
45      - rabbitmq_log:/var/log/rabbitmq
46    environment:
47      - RABBITMQ_DEFAULT_USER=${RABBITMQ_USER}
48      - RABBITMQ_DEFAULT_PASS=${RABBITMQ_PASS}
49  volumes:
50    rabbitmq_data:
51      driver: local
52    rabbitmq_log:
53      driver: local
```

Docker Compose pozwala skorzystać z dostępnych w publicznym repozytorium obrazów poszczególnych aplikacji jak np. z obrazu aplikacji RabbitMQ `rabbitmq:3-management`. Umożliwia on także budowanie własnych obrazów, za pomocą dołączanych plików w formacie Dockerfile. Stworzenie własnych plików Dockerfile było konieczne do uruchomienia aplikacji klienckiej oraz wszystkich mikroserwisów. Poniższy kod przedstawia zawartość pliku Dockerfile pozwalającego zbudować obraz aplikacji klienckiej Angular z wykorzystaniem serwera Nginx:

```
1 FROM node:18-alpine3.15 AS build
2 WORKDIR /app
3 RUN npm install -g @angular/cli@15.0.4
4 COPY ./package.json .
5 RUN npm install
6 COPY . .
7 RUN ng build
8 FROM nginx as runtime
9 COPY --from=build /app/dist/simple-bills-gui /usr/share/nginx/html
```

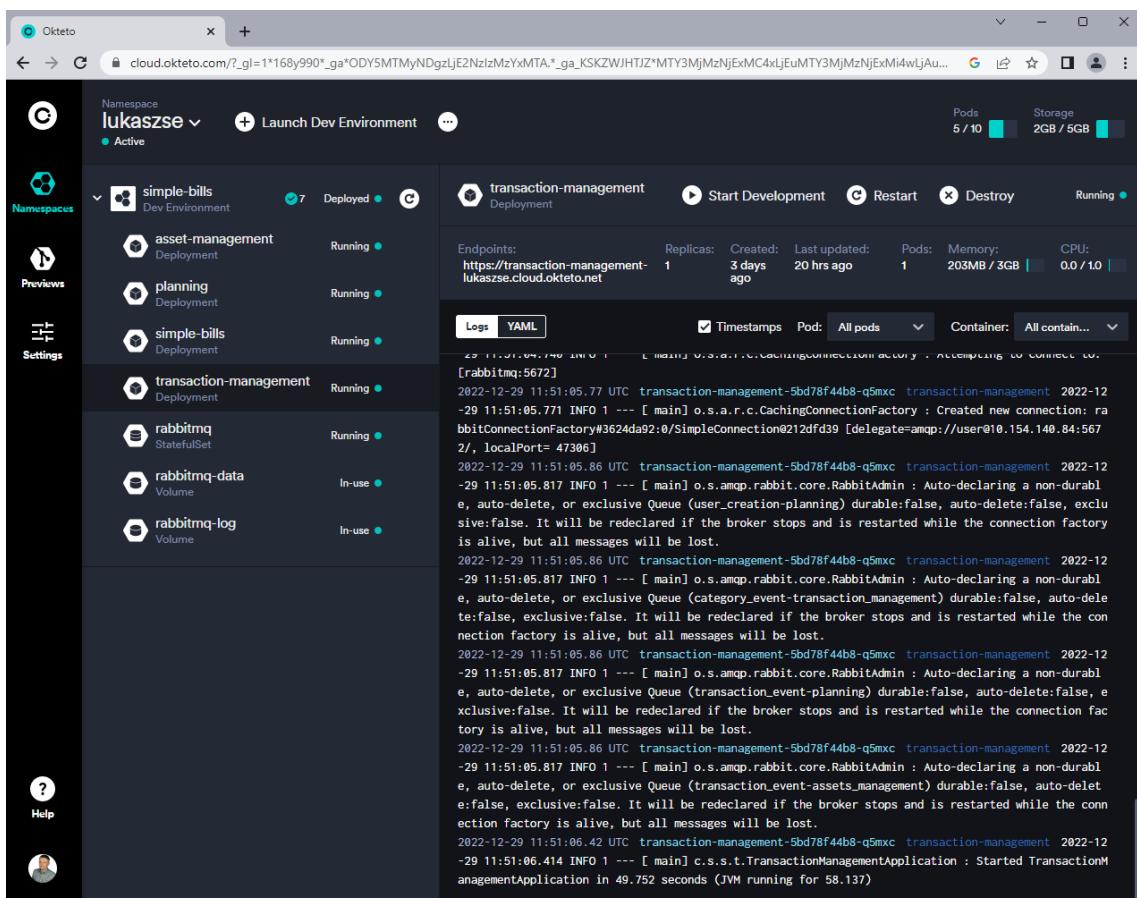
Nieco inaczej będzie wyglądał plik Dockerfile odpowiedzialny za zbudowanie mikroserwisów napisanych w Javie:

```
1 FROM gradle:7.5.1-jdk17-alpine AS build
2 COPY --chown=gradle:gradle . /home/gradle/src
3 WORKDIR /home/gradle/src
4 RUN gradle build -x test --no-daemon
5
6 FROM openjdk:17.0.2-jdk
7 RUN mkdir /app
8 COPY --from=build /home/gradle/src/build/libs/*.jar /app/spring-boot-application.jar
9 ENTRYPOINT ["java", "-jar", "/app/spring-boot-application.jar"]
```

Aby wdrożenie było możliwe, przedstawione powyżej pliki umieszczone razem z kodem aplikacji, przy czym plik `docker-compose.yml` znajduje się w głównym folderze dla całego projektu, podczas gdy pliki `Dockerfile` znajdują się w folderach poszczególnych aplikacji. Plik `docker-compose.yml` zawiera m.in. konfigurację punktów końcowych, portów, a także zmiennych systemowych, które mogą być za jego pomocą tworzone. Może on też korzystać ze zmiennych systemowych dostępnych w środowisku, dodanych za pomocą innego mechanizmu, jakim jest np. mechanizm obsługujący hasła i dane poufne udostępniony przez Okteto.

Istnieją dwie możliwości wdrożenia systemu na platformie Okteto: z wykorzystaniem dostępu po przez linie poleceń (Okteto CLI) lub też bezpośrednio z repozytorium na platformie Github. W obu przypadkach

po wywołaniu procesu wdrożenia na platformie Okteto budowane są obrazy poszczególnych aplikacji, a następnie uruchamiane są one w niezależnych kontenerach, które z kolei umieszczane są w osobnych podach. Pody są najmniejszymi niepodzielnymi jednostkami na platformie Kubernetes i zawierają najczęściej jeden kontener. Podgląd poszczególnych parametrów, w tym dostęp do logów poszczególnych aplikacji możliwy jest między innymi przy pomocy interfejsu graficznego oferowanego przez Okteto. Przykładowy widok interfejsu Okteto pokazujący uruchomione składniki systemu przedstawia Rysunek 34. Na platformie Okteto uruchomiono wszystkie składniki systemu poza aplikacją Keycloak oraz bazą danych MongoDB. Aplikacja Keycloak uruchomiona została na osobnym serwerze z systemem Linux, w przypadku bazy danych natomiast zdecydowano się skorzystać z bazy danych dostępnej w ramach usługi SaaS.



Rysunek 34 Widok interfejsu graficznego Okteto pokazujący uruchomione komponenty systemu Simple Bills

Nieodłącznym elementem procesu CI/CD jest automatyczne uruchamianie testów, których pomyślne ukończenie jest warunkiem wdrożenia

nowej wersji aplikacji na danym środowisku. Testy takie mogą być uruchamiane np. z wykorzystaniem narzędzia do automatyzacji procesu budowania projektu jakim jest Gradle. W praktyce często jednak stosuje się osobne narzędzia pozwalające na uruchamianie testów jednostkowych i integracyjnych. Ze względu na złożoność zagadnienia postanowiono pominąć uruchamianie testów jednostkowych oraz integracyjnych podczas procesu automatycznego wdrażania systemu zarówno na środowisku lokalnym, jak i na środowisku deweloperskim Okteto. Wdrożenie aplikacji na środowisku produkcyjnym wymagało by jednak również zaimplementowania mechanizmów pozwalającego testować automatycznie wszystkie aplikacje przed ich wdrożeniem. Szczegóły odnośnie testów jednostkowych i integracyjnych zastosowanych w niniejszym systemie przedstawiono w rozdziale 9.

7.3 Rozwój aplikacji w środowisku lokalnym

Rozwój systemów mikroserwisowych w środowisku lokalnym może być utrudniony z powodu konieczności uruchamiania kilku, a często kilkudziesięciu osobnych aplikacji jednocześnie. Z pomocą tutaj przychodzi również Docker oraz Docker Compose. Przy pomocy zmodyfikowanego nieco pliku *docker-compose.yml* przygotowano konfigurację pozwalającą lokalnie zbudować i uruchomić w osobnych kontenerach Dockera, wszystkie potrzebne składniki aplikacji, podobnie jak to się odbywało w przypadku Okteto. Jednak aby uniknąć konfliktów, w tym przypadku ruch każdej z aplikacji przekierowywany jest do innego portu TCP. Rozwój poszczególnych aplikacji składających się na cały system możliwy jest w ten sposób, że aplikacje działające w kontenerach Dockera mogą być niezależnie zatrzymywane, a na ich miejsce uruchamiane lokalne instancje aplikacji bezpośrednio ze zmienionego kodu przy użyciu funkcji jakie oferują środowiska programistyczne. W przypadku prac nad implementowanym systemem wykorzystano środowisko IntelliJ, służące do rozwoju aplikacji m.in. w języku Java oraz środowisko Webstorm, służące do rozwoju aplikacji napisanych w języku JavaScript oraz TypeScript.

Główny plik Docker Compose przeznaczony do wdrożenia kontenerów dla wszystkich komponentów ma nazwę *docker-compose-local.yml*. Lokalna konfiguracja korzysta z tych samych plików Dockerfile, co konfiguracja zastosowana dla platformy Okteto, za wyjątkiem aplikacji klienckiej, wymagającej osobnego pliku Dockerfile, w celu ustawienia trybu deweloperskiego.

8 Przykładowy scenariusz wykorzystania systemu

W niniejszym rozdziale przedstawiony zostanie scenariusz wykorzystania systemu odpowiadający scenariuszowi przypadku użycia, który prezentuje Tabela 6 (paragraf 5.1.4). Na scenariusz składają się następujące kroki:

1. Logowanie do systemu.

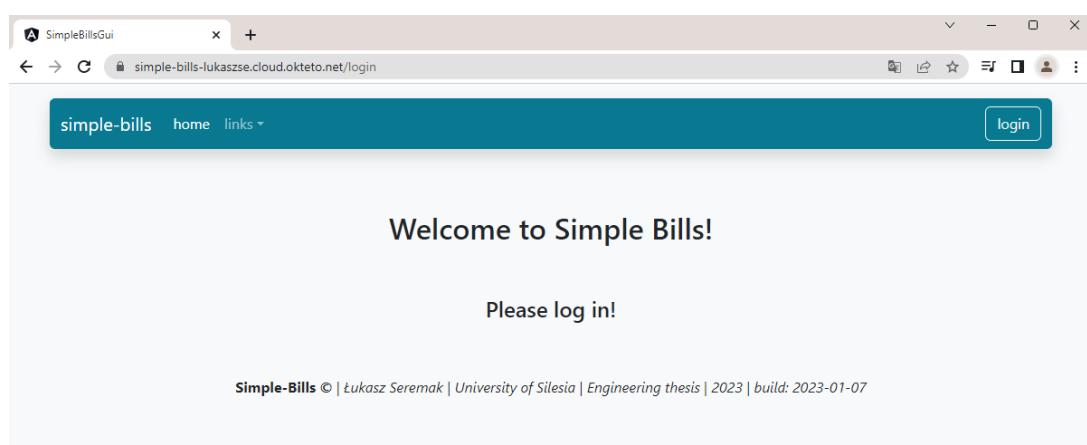
W przypadku kiedy użytkownik nie został wcześniej zalogowany lub token przechowywany w przeglądarce jako ciasteczkko stracił ważność, użytkownik musi się najpierw zalogować. Aby to zrobić po wejściu na stronę należy kliknąć na przycisk „login” znajdujący się w prawym górnym rogu ekranu. Widok dla niezalogowanego użytkownika przedstawia Rysunek 35. Po kliknięciu na przycisk „login” użytkownik zostanie przekierowany do ekranu logowania obsługiwanej przez aplikację Keycloak (Rysunek 36).

2. Przekierowanie do ekranu głównego.

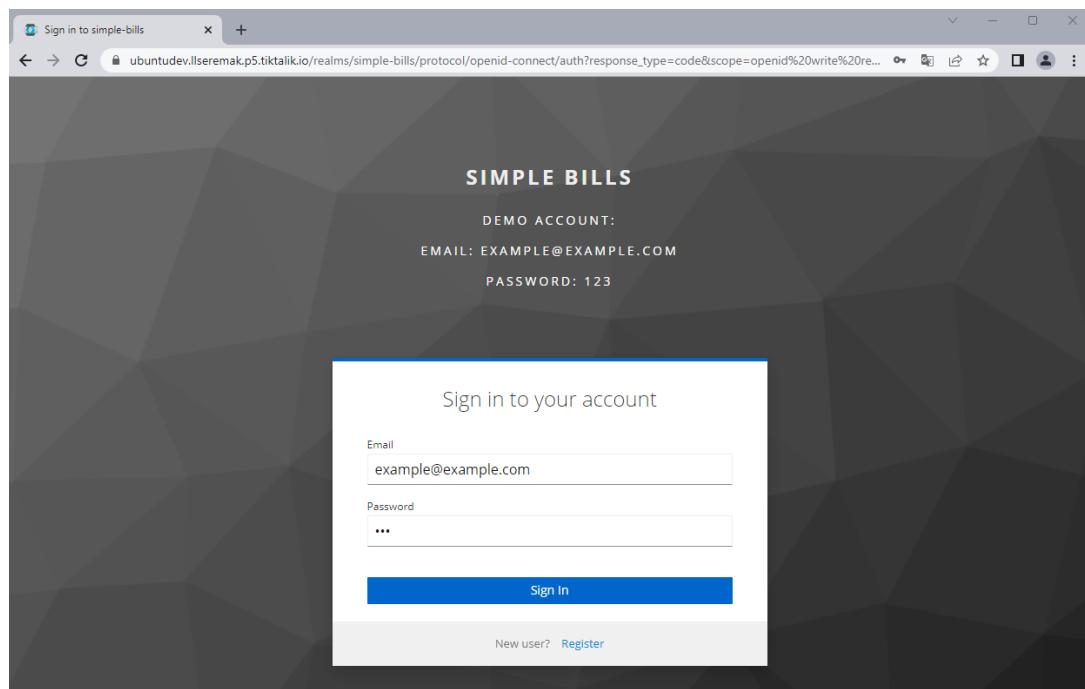
W przypadku pomyślnego zalogowania użytkownikowi wyświetlony zostanie ekran główny aplikacji (Rysunek 37).

3. Wyświetlenie listy przychodów i wydatków.

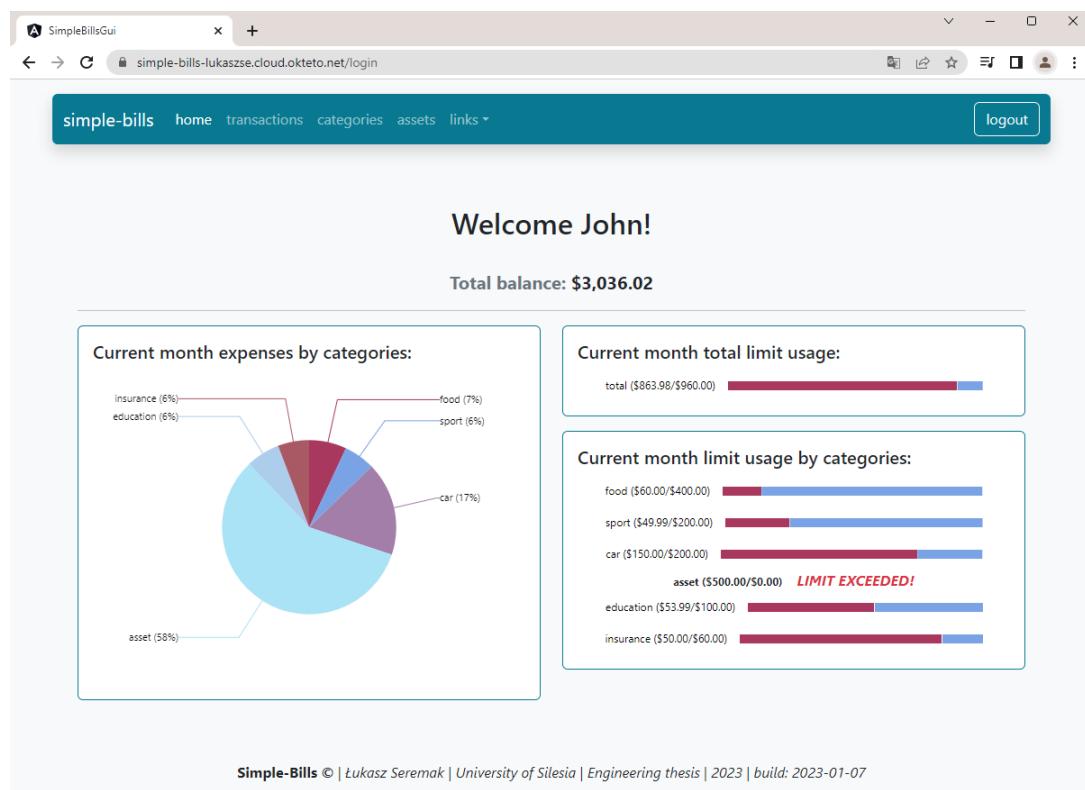
Po kliknięciu w etykietę „transactions” znajdującą się w głównym menu, użytkownikowi wyświetlony zostanie ekran zawierający listę przychodów i wydatków oraz pola umożliwiające filtrowanie listy wg zadanych kryteriów (Rysunek 38).



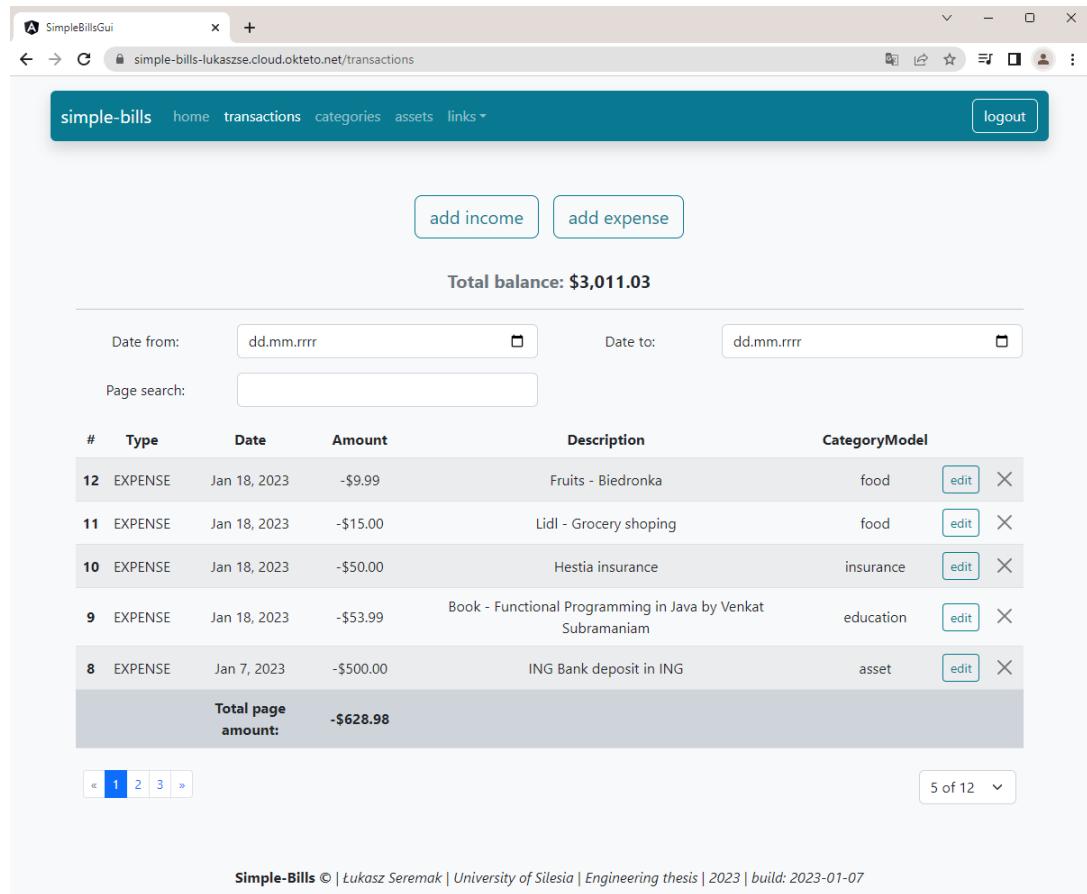
Rysunek 35 Widok aplikacji dla niezalogowanego użytkownika



Rysunek 36 Ekran logowania do aplikacji Simple Bills obsługiwany przez zewnętrzna aplikację Keycloak



Rysunek 37 Ekran główny aplikacji Simple Bills



Rysunek 38 Ekran transakcji zawierający listę przychodów i wydatków

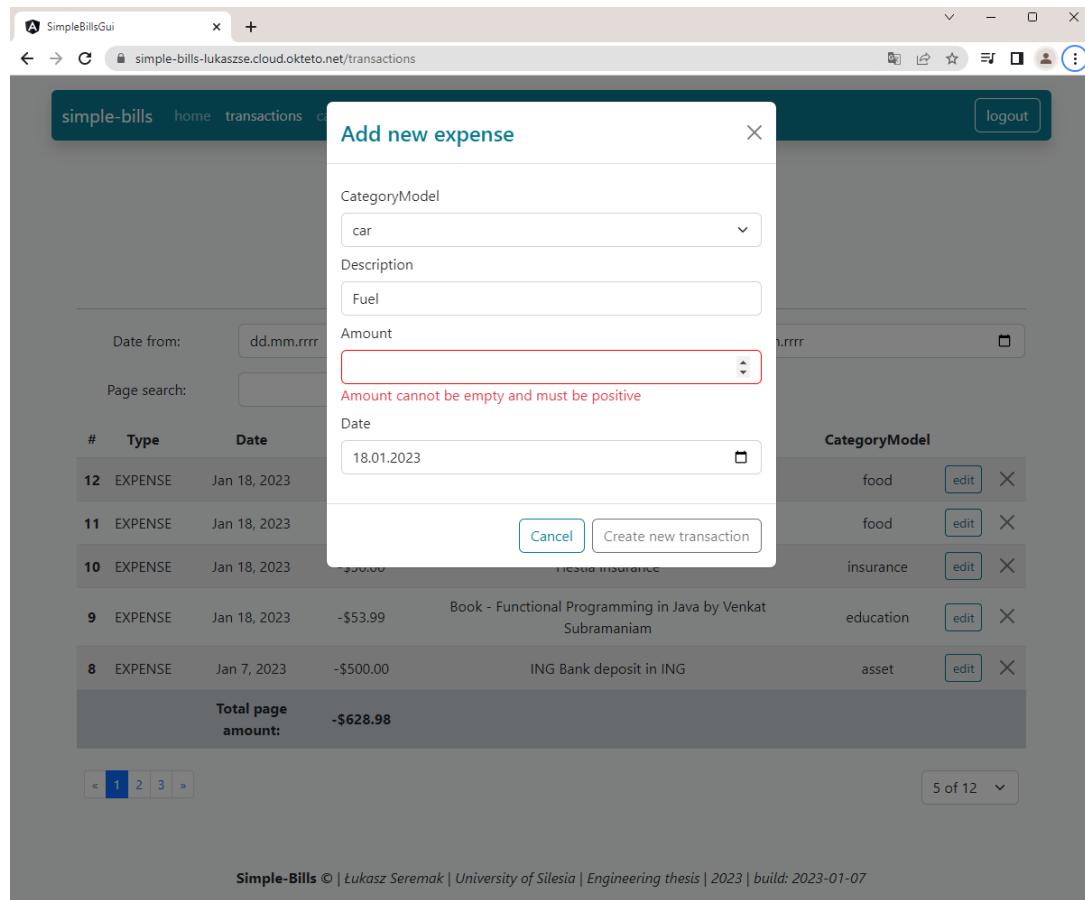
4. Dodawanie nowego wydatku.

Po kliknięciu na przycisk „add expense” wyświetlane zostanie okno modalne zawierające formularz umożliwiający wprowadzenie danych związanych z nowym wydatkiem (Rysunek 39). Przycisk dodawania nowej transakcji znajdujący się u dołu okna modalnego pozostanie zdezaktywowany do momentu prawidłowego wprowadzenia danych. Wprowadzenie błędnej wartości w danym polu lub zostawienie jej pustej spowoduje wyświetlenie się komunikatu informującego o błędzie. Po wprowadzeniu prawidłowych danych przycisk dodawania nowej transakcji zostanie aktywowany, co ilustruje Rysunek 40.

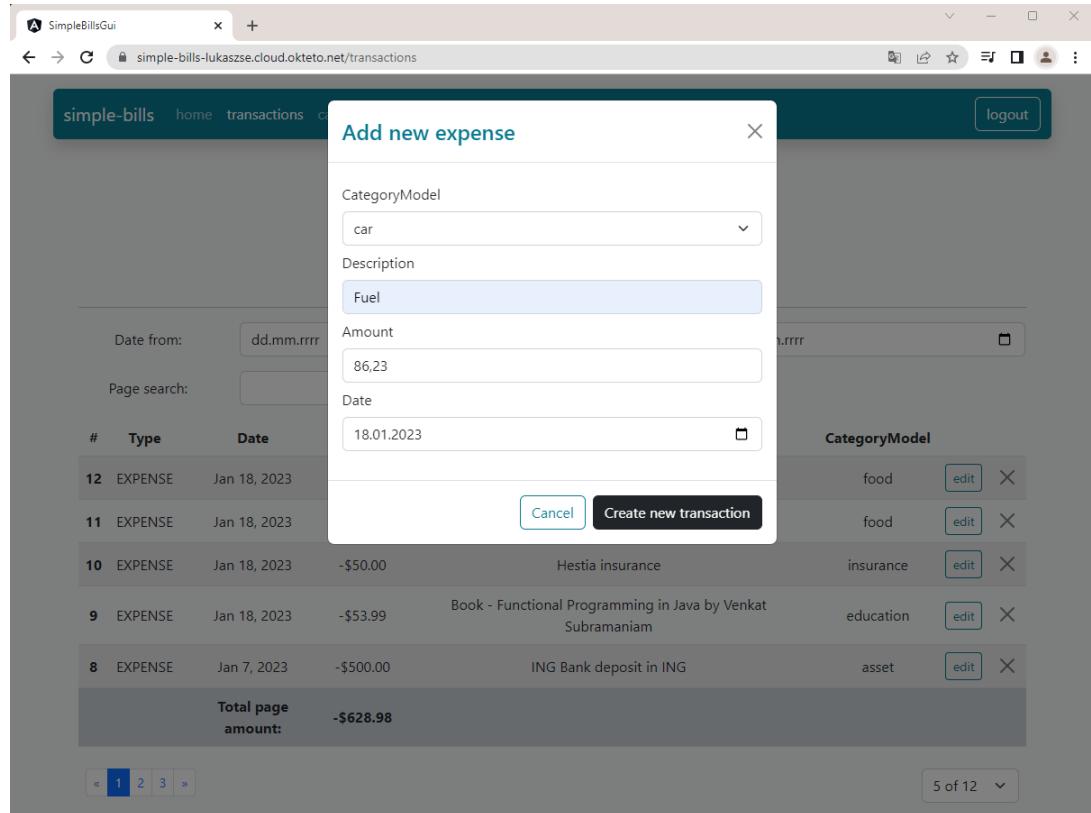
5. Wyświetlenie nowo dodanego wydatku na liście transakcji

Po prawidłowym wypełnieniu formularza i kliknięciu na przycisk przycisku „create new transaction” żądanu utworzenia nowej transakcji zostanie wysłane do mikroserwisu Transaction management, który następnie utworzy

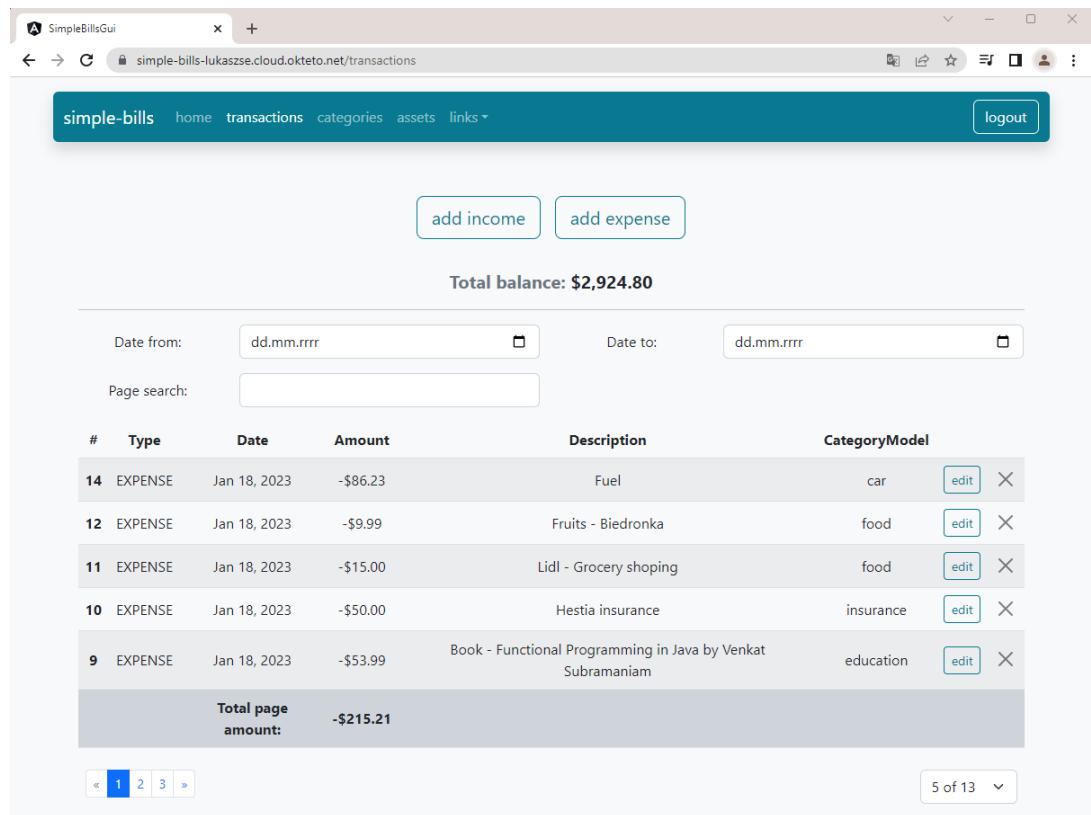
nową transakcję, po czym ekran aplikacji zostanie odświeżony a na liście przychodów i wydatków pojawi się nowo dodana transakcja (Rysunek 41). W przypadku wystąpienia błędu po stronie serwera, zwróci on odpowiedź z odpowiednim kodem błędu. Kod ten zostanie przetworzony i wyświetlony jako komunikat w przeglądarce. Z uwagi na fakt, że na ekranie transakcji wyświetlany jest również stan konta – on także zostanie zaktualizowany.



Rysunek 39 Modalne okno formularza dodawania nowej transakcji wraz z komunikatem informującym o błędzie w formularzu i zablokowanym przyciskiem tworzenia nowej transakcji



Rysunek 40 Modalne okno z prawidłowo wypełnionym formularzem



Rysunek 41 Widok listy przychodów i wydatków zawierająca nowo dodany wydatek

9 Testy i weryfikacja systemu

W niniejszym rozdziale przedstawiono wykorzystane sposoby weryfikacji działania systemu. Skupiono się przy tym przede wszystkim na testach automatycznych w tym testach jednostkowych oraz testach integracyjnych. Przykładowe testy integracyjne i jednostkowe zaimplementowane zostały w serwisie Transaction management. Przykładowe testy jednostkowe mające na celu testowanie statycznych metod zaimplementowano także w bibliotece Simple Bills Commons. W obu przypadkach do implementacji testów wykorzystano środowisko testowe Spock oferujące możliwość pisania testów w języku Groovy. Specyfika języka Groovy, który oferuje m.in. przeciążanie operatorów sprawia, że bardzo dobrze nadaje się on do pisania testów jednostkowych sterowanych danymi (ang. Data-Driven Testing). Podejście takie pozwala na sprawdzenie wielu przypadków testowych, dla różnych wariantów danych, w ramach jednej metody testowej.

9.1 Automatyczne testy jednostkowe

Przykład testu jednostkowego zaimplementowanego w serwisie Transaction management, napisanego zgodnie z podejściem DDT załączono poniżej:

```
1 @Slf4j
2 @Testcontainers
3 @SpringBootTest(classes = IntSpecConfig.class)
4 @ActiveProfiles("test")
5 class TransactionServiceSpec extends Specification {
6
7     @SpringBean
8     RabbitMQConfig rabbitMQConfig = Mock()
9
10    @SpringBean
11    RabbitMessagingTemplate rabbitMessagingTemplate = Mock()
12
13    @SpringBean
14    ReactiveMongoTemplate reactiveMongoTemplate = Mock()
15
16    @SpringBean
17    TransactionCrudRepository transactionCrudRepository = Mock()
18
19    @SpringBean
20    TransactionSearchRepository transactionSearchRepository = Mock()
21
22    @SpringBean
23    UserActivityRepository userActivityRepository = Mock()
```

```
25  @SpringBean
26  MessageListener messageListener = Mock()
27
28  @SpringBean
29  MessagePublisher messagePublisher = Mock()
30
31  @Autowired
32  TransactionService transactionService
33
34  def "should update transaction"() {
35
36      given: "prepare test data"
37      def transactionDto = new TransactionDto(
38          category: newCategory,
39          amount: newAmount,
40          type: "EXPENSE"
41      )
42
43      def existingTransaction = new Transaction(
44          user: "testUser",
45          transactionNumber: 1,
46          type: Transaction.Type.EXPENSE,
47          category: "food",
48          amount: -1 * existingAmount
49      )
50
51      def updatedTransaction = new Transaction(
52          user: "testUser",
53          transactionNumber: 1,
54          type: Transaction.Type.EXPENSE,
55          category: newCategory,
56          amount: -1 * newAmount
57      )
58
59      def transactionEventMessage = new TransactionEventDto(
60          username: "testUser",
61          transactionNumber: 1,
62          categoryName: newCategory,
63          amount: existingAmount - newAmount,
64          type: ActionType.UPDATE
65      )
66
67      and: "mock database response"
68      1 * transactionCrudRepository
69          .findByUserAndTransactionNumber("") +
70              "testUser", 1) >> Mono.just(existingTransaction)
71      1 * messagePublisher
72          .sendTransactionEventMessage(transactionEventMessage)
73      1 * transactionSearchRepository
74          .updateTransaction(updatedTransaction) >> Mono.just(updatedTransaction)
75
76      when:
77      def updatedTransactionMono =
78          transactionService.updateTransaction(
79              "testUser", 1, transactionDto)
```

```

80
81     then:
82         StepVerifier
83             .create(updatedTransactionMono)
84             .expectNextMatches(transaction ->
85                 transaction == updatedTransaction)
86             .expectComplete()
87             .verify()
88
89     where:
90     newCategory | newAmount | existingAmount
91     "food"      | 300       | 100
92     "car"       | 100       | 100
93     "travel"    | 100       | 144.23
94     "sport"     | 124.99    | 299.99
95 }
96 }
```

Test ten sprawdza poprawność funkcji uaktualniania transakcji w tym ilość wywołań poszczególnych metod, a także dzięki wykorzystaniu narzędzia jakim jest StepVerifier, poprawność reaktywnego strumienia Mono, w tym ilość wykonanych operacji oraz zakończenie strumienia. Tabela znajdująca się na dole metody zawiera dane dla czterech różnych przypadków testowych, które zostaną wykonane kolejno po sobie, za pomocą przedstawionej we fragmencie kodu logiki. Ponadto w teście pokazano sposób tworzenia protez (ang. mock) dla metod w tym przypadku odpowiedzialnych za pobieranie danych z bazy danych. Aby test się wykonał, budowany jest cały kontekst środowiska Spring i wstrzykiwane są wszystkie niezbędne zależności. Adnotacja `@SpringBean` umożliwia zastępowanie bieżących instancji klas, np. przez protezy Mock(). Zachowanie się protez definiowane jest przy pomocy operatora `>>` pozwalającego przypisać obiekt, jaki ma być zwracany przez konkretną metodę, dla określonych parametrów wywołania.

Inny przykład testu jednostkowego DDT, służącego do testowania statycznej metody znajdującej się w bibliotece Simple Bills Commons przedstawiono poniżej:

```

1 def "should uprate balance correctly" () {
2
3     given: "prepare TransactionEventDto"
4     def transactionEventDto =
5         TransactionBalanceSpecUtils
6             .prepareTransactionEventDto(transactionAmount as BigDecimal, actionType)
7
8     when: "invoke updateBalance method"
9     def balanceAfterUpdate =
10        TransactionBalanceUtils
```

```
11     .updateBalance(balBeforeUpdt as BigDecimal, transactionEventDto)
12
13     then: "should calculate balance correctly"
14     balanceAfterUpdate == expectedBalAfterUpdt
15
16     where:
17     transactionAmount | actionType | balBeforeUpdt || expectedBalAfterUpdt
18     -100              | CREATION   | 5000          || 4900
19     -99.99            | UPDATE      | 5000          || 4900.01
20     99.99             | UPDATE      | 5000          || 5099.99
21     -344.03           | DELETION    | 1000.33       || 1344.36
22 }
```

W przypadku testowania metod statycznych, nie ma konieczności tworzenia protez (ang. mock), co upraszcza tworzenie takich testów.

9.2 Automatyczne testy integracyjne

Testy integracyjne pełnią ważną rolę, pozwalając bowiem sprawdzić interakcje aplikacji z otoczeniem, w tym z bazą danych, serwisami zewnętrznymi czy też brokerami wiadomości. Wykorzystanie rzeczywistych systemów zewnętrznych w testach jednostkowych jest jednak najczęściej niemożliwe. Często spotykaną praktyką jest wykorzystywanie emulowanych komponentów np. baz danych działających w pamięci komputera. Takie rozwiązanie jednak ma wady wynikające z faktu, że emulowane komponenty mogą w praktyce zachowywać się inaczej niż ich prawdziwe odpowiedniki. Lepszym rozwiązaniem wydaje się tutaj zastosowanie tymczasowych kontenerów Dockera, w których to aplikacje zewnętrzne będą uruchamiane tylko na czas przeprowadzenia testów. Taką możliwość oferują biblioteki Testcontainers, które zastosowano w testach integracyjnych serwisu Transaction management. Wszystkie testy integracyjne zbudowano w taki sposób, że dziedziczą one po klasie bazowej zawierającej konfigurację tymczasowych kontenerów testowych z bazą danych oraz brokerem wiadomości RabbitMQ. Klasę bazową dla testu przeznaczonego do testowania punktów końcowych przedstawiono poniżej:

```
1 @Slf4j
2 @Testcontainers
3 @SpringBootTest(
4     classes = IntSpecConfig.class,
5     webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
6 @ActiveProfiles("test")
7 class EndpointIntSpec extends Specification {
8
9     @LocalServerPort
```

```
10    int port
11
12    @Container
13    static MongoDBContainer mongoDBContainer =
14        new MongoDBContainer("mongo:4.4.3")
15
16    @Container
17    static RabbitMQContainer rabbitMQContainer =
18        new RabbitMQContainer("rabbitmq:3-management")
19
20    @Autowired
21    TransactionService transactionService
22
23    @Autowired
24    SequentialIdRepository sequentialIdService
25
26    @Autowired
27    TransactionCrudRepository transactionCrudRepository
28
29    @Autowired
30    TestRabbitListener testRabbitListener
31
32    @Shared
33    def client = new RestTemplate()
34
35    def conditions = new PollingConditions(timeout: 5, initialDelay: 1)
36
37
38    @DynamicPropertySource
39    static void mongoProps(DynamicPropertyRegistry registry) {
40        mongoDBContainer.start()
41        rabbitMQContainer.start()
42        log.info("MongoDb container connection string: {}",
43            mongoDBContainer.getConnectionString())
44        registry.add("spring.data.mongodb.uri",
45            () -> mongoDBContainer.getConnectionString())
46        registry.add("spring.rabbitmq.host",
47            () -> rabbitMQContainer.getHost())
48        registry.add("spring.rabbitmq.port",
49            () -> rabbitMQContainer.getAmqpPort())
50    }
51
52    def setup() {
53        testRabbitListener.clearReceivedMessages()
54    }
55
56    def cleanupSpec() {
57        mongoDBContainer.stop()
58        rabbitMQContainer.stop()
59    }
60 }
```

Jak widać powyższy kod tworzy tymczasowe kontenery z bazą danych MongoDB oraz brokerem wiadomości RabbitMQ oraz usuwa je tuż po zakończeniu testu.

Dostarcza on także narzędzia wykorzystywane w testach takie jak m.in. testową instancję synchronicznego klienta HTTP RestTemplate. Poniżej przedstawiono konkretną metodę testową służącą do testowania danych pobieranych po przez interfejs REST API, znajdującą się w klasie *TransactionEndpointIntSpec* dziedziczącej po wyżej pokazanej klasie bazowej:

```
1
2     def 'should fetch transaction'() {
3
4         given: 'populate database with test data'
5         populateDatabase(transactionService)
6
7         and: 'prepare request to get transaction'
8         def request =
9             RequestEntity.get(
10                 TRANSACTION_URL_PATTERN.formatted(
11                     port, "/%d".formatted(transactionNumber)))
12                 .accept(MediaType.APPLICATION_JSON)
13                 .header(AUTHORIZATION_HEADER, BASIC_TOKEN).build()
14
15         when: 'should fetch bill'
16         def response = client.exchange(request, Transaction.class)
17
18         then:
19             response != null
20             response.getStatusCode() == HttpStatus.OK
21             response.getBody().getCategory() == category
22             response.getBody().getAmount() == BigDecimal.valueOf(amount)
23
24         where:
25             transactionNumber | category | amount
26             1                | SALARY    | 5000
27             2                | TRAVEL    | -1000
28             3                | CAR       | -300
29             4                | FOOD      | -200
30     }
```

Jak można zauważyć w pierwszej części metody testowej, tuż pod klauzulą given znajduje się metoda `populateDatabase()` umożliwiająca przygotowanie danych testowych i zapisanie ich w bazie danych stworzonej w tymczasowym kontenerze testowym. Kod tej metody przedstawiono poniżej:

```
1 static def populateDatabase(TransactionService transactionService) {
2     log.info("Populating database before test")
3     def transactions =
4         JsonImporter.getDataFromFile("json/transactions.json",
5             new TypeReference<List<TransactionDto>>() {})
6     log.info("Number of transaction to populate {}", transactions.size())
7     def addedTransactions = transactions.stream()
```

```

8     .map(transactionToSave ->
9         transactionService.createTransaction(TEST_USER, transactionToSave))
10    .map(transactionMono -> transactionMono.block())
11    .collect(Collectors.toList())
12    log.info("Transaction added to database: {}", addedTransactions.toString())
13 }

```

Metoda ta wczytuje dane z pliku JSON, mapuje do odpowiedniego formatu, a następnie zapisuje je w bazie danych z wykorzystaniem logiki zaimplementowanej w klasie TransactionService. Po zakończeniu testu kontener jest usuwany z całą zawartością bazy danych. W celu wczytania danych z pliku JSON wykorzystana jest generyczna metoda przedstawiona poniżej:

```

1 static <T> T getDataFromFile(String filePath, TypeReference<T> valueTypeRef) {
2     def jsonString = getStringFromFile(filePath)
3     def elements = CommonTestUtils
4         .objectMapper.readValue(jsonString, valueTypeRef) elements as T
5 }

```

W dalszej części testu wykonywane jest żądanie do punktu końcowego z wykorzystaniem klienta HTTP po czym sprawdzana jest poprawność danych zwróconych w odpowiedzi zapytania.

Oprócz wyżej opisanych testów integracyjnych dla punktów końcowych zaimplementowano dodatkowo testy sprawdzające tworzenie nowej transakcji z wykorzystaniem interfejsu REST API, w tym sprawdzające czy po utworzeniu transakcji odpowiednia wiadomość dotarła do uruchomionego w kontenerze testowym brokera wiadomości, a także testy sprawdzające poprawność usuwania transakcji z wykorzystaniem interfejsu REST API. Oprócz testów dla pozytywnej ścieżki, zaimplementowano także testy integracyjne sprawdzające kody błędów w sytuacji kiedy żądana transakcja nie została znaleziona w systemie oraz w przypadku kiedy ciało zapytania REST API zawiera błędy.

Mikroserwis Transaction management posiada również klasę służącą do odbierania przychodzących wiadomości asynchronicznych. Poniżej przedstawiono test którego zadaniem jest stworzenie testowej wiadomości, za pomocą testowego brokera RabbitMQ uruchomionego w tymczasowym kontenerze, a następnie sprawdzenie czy serwis odebrał i prawidłowo przetworzył wyslaną wiadomość testową.

```

1 def "should receive and handle Category message"() {
2
3     given: "populate database"
4     populateDatabase(transactionService)
5

```

```
6   and: "prepare test data"
7   def categoryEvent = prepareCategoryEventDto()
8   def transactionQueryParams = new TransactionQueryParams(category: WAGES)
9
10  when: "publish test message"
11  testRabbitPublisher.sendCategoryEventMessage(categoryEvent)
12
13  then:
14  conditions.eventually {
15      def transactions = transactionService
16          .findTransactionsByCategory(TEST_USER, transactionQueryParams)
17          .map { it.t1 }
18          .block()
19      assert transactions.size() == 1
20
21      def transaction = transactions.get(0)
22      assert transaction.getAmount() == 5000
23      assert transaction.getCategory() == WAGES
24      assert transaction.getDescription() == "Salary"
25  }
26 }
```

10 Zakończenie

Celem niniejszej pracy był projekt oraz implementacja aplikacji internetowej służącej do zarządzania finansami domowymi. Zgodnie z założeniami implementacja poszczególnych funkcji stanowiła przede wszystkim tło pozwalające pokazać możliwości zastosowanych technologii. Funkcje zaimplementowane w aplikacji pozwalają na zarządzanie finansami w podstawowym zakresie, w szczególności na zarządzanie bieżącymi przychodami oraz wydatkami. Użytkownik może dodawać, usuwać oraz edytować swoje przychody i wydatki, definiować limity wydatków dla poszczególnych kategorii oraz podglądać stopień ich w wykorzystania za pomocą graficznych wykresów. Ponadto aplikacja została wyposażona w przydatną, jednak rzadko spotykana w podobnych aplikacjach funkcję pozwalającą na zarządzanie depozytami i lokatami bankowymi, która w przyszłości może zostać rozszerzona o bardziej rozbudowane funkcje związane z zarządzaniem majątkiem opisane w rozdziale drugim niniejszej pracy.

Zgodnie z założeniami aplikacja została zbudowana z wykorzystaniem architektury opartej o mikrousługi. W praktyce polegało to na zaimplementowaniu niezależnej aplikacji klienckiej napisanej w języku TypeScript z wykorzystaniem środowiska Angular oraz trzech niezależnych mikroserwisów napisanych w języku Java z wykorzystaniem środowiska Spring Boot. Należy przy tym podkreślić, że zarówno aplikację kliencką, jak i mikrousługi napisano w sposób reaktywny z wykorzystaniem bibliotek RxJs oraz Spring WebFlux. W zaimplementowanym systemie komunikacja pomiędzy aplikacją działającą w przeglądarce internetowej, a mikroserwisami działającymi po stronie serwera odbywa się z wykorzystaniem interfejsu REST API. Komunikacja pomiędzy samymi mikrousługami z kolei, zgodnie z przyjętymi założeniami może odbywać się na dwa sposoby: synchronicznie po przez interfejs REST API oraz asynchronicznie z wykorzystaniem brokera wiadomości RabbitMQ.

Istotną kwestię stanowił sposób wdrożenia zaimplementowanego systemu. W tym celu skorzystano z platformy Okteto pozwalającej na uruchamianie aplikacji w środowisku Kubernetes. Platforma Okteto nie zapewnia środowiska produkcyjnego, a środowisko zoptymalizowane dla programistów, ułatwiające tworzenie aplikacji, jednak w warunkach przypominających rzeczywiste środowisko produkcyjne. W celu wdrożenia systemu składającego się z kilku

odrębnych aplikacji wykorzystano możliwości jakie oferuje Docker Compose, którego pliki konfiguracyjne wspierane są przez platformę Okteto, a także repozytorium Gitlab, na którym znajdują się wszystkie pliki źródłowe. Umożliwiło to pełną automatyzację procesu wdrożenia, bezpośrednio z repozytorium na platformie Gitlab. Ponadto zmodyfikowane pliki Docker Compose wykorzystano do uruchamiania systemu na komputerze lokalnym, co umożliwiło łatwe wprowadzanie zmian w poszczególnych elementach systemu w środowisku lokalnym.

Ważnym elementem systemu są również automatyczne testy jednostkowe oraz integracyjne zaimplementowane w jednym z mikroserwisów, a także we wspólnej dla wszystkich mikrousług bibliotece. Testy te zostały napisane w języku Groovy z wykorzystaniem środowiska Spock oraz podejścia opartego na testach sterowanych danymi (ang. Data-Driven Tests).

Zaletą tak zaimplementowanego systemu jest niewątpliwie jego skalowalność. W praktyce dzięki zastosowaniu oddzielnej dla każdego serwisu nierelacyjnej bazy danych, mikrousługi można skalować poziomo po przez uruchamianie dodatkowych instancji, w zależności od obciążenia systemu. Dzięki zastosowaniu bibliotek reaktywnych dane są przetwarzane w sposób wielowątkowy, co teoretycznie poprawia również efektywność tak napisanych aplikacji. Inną istotną kwestią jest asynchroniczna komunikacja pomiędzy mikrousługami z wykorzystaniem brokera wiadomości RabbitMQ, która również sprzyja polepszeniu wydajności działającego systemu. Poszczególne składowe systemu w większości przypadków nie muszą bowiem czekać na natychmiastową odpowiedź, co powoduje lepsze wykorzystanie zasobów przez aplikacje, a także ułatwia tworzenie aplikacji modułowych składających się z luźno powiązanych między sobą serwisów.

Zaimplementowany w ten sposób system mikroserwisowy posiada również wady. Najważniejszą z nich jest poziom skomplikowania systemu, większy niż w przypadku aplikacji monolitycznych, co przekłada się w sposób istotny na czasochłonność implementacji poszczególnych funkcji. W teorii jednak odpowiedzialność za poszczególne mikrousługi może być dedykowana osobnym niezależnym zespołem. Zdecydowanie bardziej skomplikowany jest również proces wdrożenia aplikacji wykorzystującej mikrousługi. W praktyce wymaga to zastosowania dodatkowych narzędzi umożliwiających automatyzację procesu wdrożenia np. na platformach wykorzystujących środowisko Kubernetes,

co w sposób uproszczony zrealizowano w niniejszej pracy. Inną wadą zaimplementowanego systemu jest zwiększoną trudność utrzymania spójności danych oraz brak transakcyjności w rozumieniu zasad ACID, przez co tak napisane aplikacje mniej będą nadawały się do zastosowań w których spójność i integralność danych jest kluczowa.

Ostatecznie udało się zrealizować założenia przyjęte we wstępie niniejszej pracy i zbudować system działający w oparciu o architekturę mikrousług, który w łatwy sposób może zostać wdrożony na docelowym środowisku produkcyjnym. Zaimplementowanemu systemowi do pełni funkcjonalności wprawdzie brakuje jeszcze kilku funkcji, w tym m.in. funkcji umożliwiającej zarządzanie użytkownikami, jednak modułowa budowa systemu daje możliwość łatwego jego rozszerzania, a w efekcie zbudowania rozbudowanego, skalowalnego systemu, przystosowanego do pracy pod zróżnicowanym obciążeniem.

11 Bibliografia

- [1] C. Bywalec, *Ekonomika i Finanse Gospodarstw Domowych*, Warszawa: PWN, 2012.
- [2] M. Barlik, B. Lewandowska i K. Siwiak, *Zeszyt metodologiczny. Badanie budżetów gospodarstw domowych.*, Warszawa: Główny Urząd Statystyczny, 2018.
- [3] „Encyklopedia Zarządzania,” 2022. [Online]. Available: <https://mfiles.pl/>.
- [4] R. Zajkowski, „Składowe koszty zadłużenia, a ustawa o kredycie konsumenckim,” *Rynek finansowy w erze zawierowań*, pp. 489-498, Lublin: Wydawnictwo Uniwersytetu Marii Curie-Skłodowskiej, 2009.
- [5] „Portal finansowy qmamfinanse.pl,” 2022. [Online]. Available: <https://qmamfinanse.pl/>.
- [6] „Strona aplikacji Money Manager,” Innim Mobile Exp, 2022. [Online]. Available: <https://en.innim.org/finance>.
- [7] „Strona aplikacji Easy Budget,” 2022. [Online]. Available: <https://easybudget.pl/>.
- [8] „Strona aplikacji Wallet,” 2022. [Online]. Available: <https://budgetbakers.com/>.
- [9] „Strona aplikacji Kontomierz,” Finelf sp. z o.o., 2022. [Online]. Available: <https://kontomierz.pl/>.
- [10] „Strona aplikacji Personal Capital,” 2022. [Online]. Available: <https://www.personalcapital.com/>.
- [11] K. Gos i W. Zabierowski, „The Comparison of Microservice and Monolithic Architecture,” *Research Gate*, 2020.
- [12] „Blog Transparent Data - Monolity vs. mikroserwisy – zalety i wady [porównanie],” 2020. [Online]. Available: <https://medium.com/blog-transparent-data/monolity-vs-mikroserwisy-zalety-i-wady-por%C3%B3wnanie-155e652fdb59>.
- [13] A. Bellmare, *Mikrousługi oparte na zdarzeniach. Wykorzystywanie danych w organizacji na dużą skalę.*, Gliwice: Helion S.A., 2021.
- [14] R. Kuhn, B. Hanadee i J. Allen, *Systemy reaktywne. Wzorce projektowe i ich stosowanie*, Gliwice: Helion, 2018.

- [15] P. Rzeźnik, „Baza wiedzy JPRO,” JCommerce, 2022. [Online]. Available: <https://www.jcommerce.pl/jpro/artykuly/nosql-vs-sql-bazy-danych>.
- [16] „Witryna domowa PlantUML,” 2022. [Online]. Available: <https://plantuml.com/>.
- [17] „Blog firmy Altkom Software,” [Online]. Available: <https://www.altkomsoftware.com/pl/blog/programowanie-reaktywne-z-wykorzystaniem-project-reactor/>.
- [18] „Dokumentacja Lombok,” 2022. [Online]. Available: <https://projectlombok.org/features/>.
- [19] „Dokumentacja Spring,” 2022. [Online]. Available: <https://spring.io/>.
- [20] „Strona Bealdung - kursy dla programistów,” 2023. [Online]. Available: <https://www.baeldung.com/>.
- [21] „Strona OAutho,” 2023. [Online]. Available: <https://autho.com/>.

12 Spis rysunków

Rysunek 1 Składniki majątku trwałego gospodarstwa domowego [1] [5]	8
Rysunek 2 Składniki majątku obrotowego gospodarstwa domowego [1] [5]	8
Rysunek 3 Widok ekranu wydatków Money Manager [6]	18
Rysunek 4 Przykładowy ekran aplikacji internetowej Easy Budget [7]	19
Rysunek 5 Ekran tablicy zestawień aplikacji Wallet firmy BudgetBakers [8] ...	20
Rysunek 6 Widok ekranu „Konta” w aplikacji internetowej Kontomierz	22
Rysunek 7 Ekran planowania wydatków aplikacji Kontomierz	23
Rysunek 8 Diagramy przypadków użycia.....	34
Rysunek 9 Architektura systemu aplikacji Simple Bills	40
Rysunek 10 Architektura warstwowa zastosowana w mikroserwisach	41
Rysunek 11 Model bazy danych mikroserwisu Transaction management.....	45
Rysunek 12 Schemat kolekcji transaction	46
Rysunek 13 Model bazy danych mikroserwisu Planning.....	47
Rysunek 14 Model bazy danych mikroserwisu Asset Management	47
Rysunek 15 Projekt interfejsu użytkownika – strona główna.....	48
Rysunek 16 Projekt interfejsu użytkownika – ekran transakcji	49
Rysunek 17 Projekt interfejsu użytkownika – okno modalne formularza	50
Rysunek 18 Projekt interfejsu użytkownika – ekran kategorii.....	50
Rysunek 19 Diagram klas mikroserwisu Transaction management odpowiedzianych za realizację obsługi transakcji.....	52
Rysunek 20 Diagram klas mikroserwisu Transaction Management odpowiedzialnych za rejestracje aktywności użytkownika.....	53
Rysunek 21 Diagram klas mikroserwisu Planning odpowiedzialnych za przetwarzanie stanu środków bieżących (balance).....	55
Rysunek 22 Diagram klas mikroserwisu Planning odpowiedzialnych za obsługę kategorii wydatków	56
Rysunek 23 Diagram klas mikroserwisu Planning odpowiedzialnych za zarządzanie usługą wykorzystania limitów wydatków dla poszczególnych kategorii	57
Rysunek 24 Diagram klas mikroserwisu Asset management odpowiedzialnych za obsługę depozytów	58
Rysunek 25 Struktura komponentów aplikacji klienckiej	67
Rysunek 26 Diagram klas powiązanych z komponentem HomeComponent	68

Rysunek 27 Diagram klas powiązanych z komponentem CategoryComponent	68
Rysunek 28 Diagram klas powiązanych z komponentem TransactionComponent	69
Rysunek 29 Diagram klas powiązanych z komponentem PieUsageChartComponent	70
Rysunek 30 Diagram sekwencji przedstawiający przepływ Authorization Code Flow [20] [21]	72
Rysunek 31 Diagram sekwencji przedstawiający przepływ danych pomiędzy komponentami odpowiadający scenariuszowi „Dodaj przychód/wydatek”	74
Rysunek 32 Diagram sekwencji przedstawiający przepływ danych pomiędzy komponentami odpowiadający przypadkowi użycia „Dodaj depozyt”	75
Rysunek 33 Fragment dokumentacji OpenAPI punktu końcowego POST /transactions	77
Rysunek 34 Widok interfejsu graficznego Okteto pokazujący uruchomione komponenty systemu Simple Bills	81
Rysunek 35 Widok aplikacji dla niezalogowanego użytkownika	83
Rysunek 36 Ekran logowania do aplikacji Simple Bills obsługiwany przez zewnętrzną aplikację Keycloak	84
Rysunek 37 Ekran główny aplikacji Simple Bills	84
Rysunek 38 Ekran transakcji zawierający listę przychodów i wydatków	85
Rysunek 39 Modalne okno formularza dodawania nowej transakcji wraz z komunikatem informującym o błędzie w formularzu i zablokowanym przyciskiem tworzenia nowej transakcji	86
Rysunek 40 Modalne okno z prawidłowo wypełnionym formularzem	87
Rysunek 41 Widok listy przychodów i wydatków zawierająca nowo dodany wydatek.....	87

13 Spis tabel

Tabela 1 Zestawienie budżetu gospodarstwa domowego bez uwzględnienia podatków i zaliczek na ubezpieczenie społeczne [1].....	9
Tabela 2 Porównanie funkcji wybranych aplikacji do zarządzania finansami domowymi.....	24
Tabela 3 Porównanie wad i zalet aplikacji monolitycznych oraz aplikacji o architekturze mikroserwisowej [10] [11]	29
Tabela 4 Scenariusz dla przypadku użycia „Logowanie użytkownika”	35
Tabela 5 Scenariusz dla przypadku użycia „Wyświetl przychody/wydatki”.	36
Tabela 6 Scenariusz dla przypadku użycia „Dodaj przychód/wydatek”....	37
Tabela 7 Lista najważniejszych bibliotek zastosowanych w aplikacji Simple Bills GUI	42
Tabela 8 Lista najważniejszych bibliotek oraz narzędzi zastosowanych w implementacji mikroserwisów	43