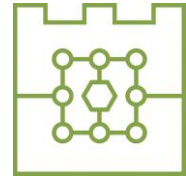




**Politechnika Krakowska
im. Tadeusza Kościuszki**

Wydział Informatyki i Telekomunikacji



Michał Lechowicz

Numer albumu: 130627

**Aplikacja internetowa do zarządzania finansami
osobistymi**

Personal finance management web application

Praca inżynierska na kierunku Informatyka

Praca wykonana pod kierunkiem:
dra inż. Lecha Jamroża

Recenzent pracy:
dr Barbara Borowik

Kraków, 2023

Spis treści

1. Cel i zakres pracy	3
2. Projekt wstępny	4
2.1. Wymagania funkcjonalne – User story	4
2.2. Wymagania niefunkcjonalne	6
2.3. Reguły biznesowe	7
2.4. Analiza możliwych rozwiązań	7
3. Architektura aplikacji	9
3.1. Diagram przypadków użycia	10
3.2. Model statyczny aplikacji – struktura bazy danych	11
3.3. Projekt graficzny	14
3.4. Diagramy interakcji	20
3.5. Dobór technologii zastosowanych do implementacji	21
4. Implementacja projektu	23
4.1. Struktura projektu	23
4.2. Implementacja serwera	24
4.3. Implementacja klienta	35
4.4. Docker	46
4.5. Przykład przepływu danych przez aplikację	48
5. Testy aplikacji	56
5.1. Testy jednostkowe	56
5.2. Testy integracyjne	65
6. Wdrożenie aplikacji	68
6.1. Continuous integration (CI) i Continuous Delivery (CD)	68

6.2. Hosting bazy danych	69
6.3. Wdrożenie aplikacji serwerowej.....	69
6.4. Wdrożenie aplikacji klienckiej	71
7. Podsumowanie.....	74
7.1. Osiągnięte cele i funkcjonalności	74
7.2. Możliwości dalszego rozwoju i wnioski z projektu	74
8. Słownik pojęć.....	75
9. Bibliografia	76
10. Spis Rysunków	79

1. Cel i zakres pracy

W obecnych czasach skuteczne zarządzanie finansami osobistymi jest bardziej istotne niż kiedykolwiek wcześniej. Kwestie związane z planowaniem budżetu, efektywnym oszczędzaniem oraz podejmowaniem mądrych decyzji finansowych stawiają przed ludźmi nowe wyzwania w niestabilnym ekonomicznie świecie. Celem tej pracy jest wyjście naprzeciw tym problemom poprzez dostarczenie narzędzia do kontroli budżetu. Dzięki temu użytkownik dowie się, czy może sobie pozwolić na dany wydatek, ustanowić cele finansowe czy też podejmować kroki mające na celu wyjście z ewentualnych długów. Wykorzystana do tego będzie aplikacja internetowa, umożliwiająca zarządzanie posiadanymi zasobami pieniężnymi, monitorowanie kierunków przepływów pieniędzy, dostęp do systemu z każdego urządzenia przez całą dobę oraz analizę finansową.

Stworzona aplikacja pomoże użytkownikom zyskać świadomość tego, na co ich pieniądze są wydawane. Będą mogli łatwo i szybko spisywać wydatki, dzielić je na kategorie i kontrahentów, by wiedzieć, jak rozkładają się procentowo oraz czy ich majątek rośnie czy maleje. W aplikacji użytkownicy będą mogli zarządzać swoimi kontami, kontrahentami, kategoriami i transakcjami. Dodatkowo duże znaczenie będzie miało zebranie i zwizualizowanie danych, aby umożliwić użytkownikom analizowanie ich nawyków finansowych. Chociaż jest coraz większa świadomość tego, że trzeba oszczędzać, ręczne spisywanie wszystkich wpływów i wydatków może być kłopotliwe. Jeśli jeszcze założymy potrzebę analizy, jej ręczne przeprowadzenie może być nieopłacalne dla dużych ilości transakcji. Przygotowana aplikacja wykona tę pracę za nas, a równocześnie będzie dostępna z każdego miejsca na świecie.

Jako nazwę projektu wybrano: Follow the money (podążaj za pieniędzmi). Jest to nawiązanie do słynnej afery Watergate i filmu o niej *All the President's Men* [6]. Fraza ta oznacza, że śledząc przepływy pieniędzy można dociec prawdy o ich pochodzeniu. W formie żartobliwej użyto jej, by zaznaczyć, że użytkownik może dowiedzieć się, gdzie znikają jego pieniądze.

By zrealizować projekt, należy zebrać i opisać wymagania, wyszczególnić przypadki użycia, wybrać technologię w celu implementacji, wykonać projekt graficzny, udostępnić aplikację użytkownikom.

2. Projekt wstępny

Niniejszy rozdział zawiera wymagania stawiane aplikacji do zarządzania budżetem. Przedstawia wymagania biznesowe względem projektu, definiuje pojęcia używane w pracy, przedstawia wymagania funkcjonalne (w formie historyjek użytkownika), wymagania niefunkcjonalne, prezentuje reguły biznesowe projektu, pokazuje diagram baz danych, przypadków użycia. Następnie projekt graficzny programu, architekturę aplikacji, diagramy interakcji oraz technologie użyte do implementacji rozwiązania założonego problemu.

2.1. Wymagania funkcjonalne – User story

Wymagania funkcjonalne to opis konkretnych działań i funkcji, jakich od systemu komputerowego lub oprogramowania oczekuje użytkownik. Określają one, co system powinien robić z punktu widzenia użytkownika, jakie akcje ma wykonywać oraz jakie warunki i ograniczenia muszą być spełnione.

User Story to zwięzły opis wymagań w projekcie, skupiający się na funkcjonalności widzianej z perspektywy użytkownika. Składa się z roli użytkownika, akcji do wykonania i celu. Jest często używany w metodykach zwinnych, aby lepiej zrozumieć potrzeby użytkowników i skupić się na dostarczaniu wartościowych funkcji.

- 1) Użytkownik może założyć profil w aplikacji.
- 2) Użytkownik może zalogować się do swojego profilu w aplikacji.
- 3) Użytkownik może się wylogować ze swojego Profilu.
- 4) Użytkownik zostanie wylogowany, po określonym czasie od zalogowania.
- 5) Użytkownik ma możliwość zobaczenia swojej Tablicy finansowej, zawierającej:
 - całkowitą kwotę ze wszystkich kont, pomniejszoną o kredyty,
 - zobaczenia różnicy pomiędzy wpływami i wydatkami z ostatnich 30 dni,
 - zobaczenia czterech najpopularniejszych kont na Tablicy finansowej,
 - wykresu wpływów i wydatków, według miesięcy,
 - czterech ostatnich aktywności, ze wszystkich kont.

- 6) Użytkownik może dodać Konto, nadając mu odpowiednią nazwę oraz typ. Konto to jest powiązane z Profilem Użytkownika.
- 7) Użytkownik może wybrać typ Konta spośród trzech dostępnych: gotówkowe, bankowe, kredytowe.
- 8) Użytkownik może zmienić nazwę i typ Konta w późniejszym czasie.
- 9) Użytkownik może usunąć Konto.
- 10) Użytkownik może utworzyć Kategorię, definiując jej nazwę.
- 11) Użytkownik może przyporządkować do Kategorii, nową Podkategorię.
- 12) Użytkownik może zmienić nazwę Kategorii lub Podkategorii.
- 13) Użytkownik może usunąć Kategorię lub Podkategorię.
- 14) Użytkownik może dodawać Płatnika, definiując jego nazwę.
- 15) Użytkownik może edytować Płatnika, zmieniając jego nazwę.
- 16) Użytkownik może usunąć Płatnika.
- 17) Użytkownik może wyświetlić wszystkie Kategorie i Podkategorie.
- 18) Użytkownik może wyświetlić wszystkich Płatników
- 19) Użytkownik może wyświetlić wszystkie Konta.
- 20) Użytkownik może wyświetlić wszystkie Płatności.
- 21) Użytkownik może wyświetlić Płatności według kont i czasu dodania.
- 22) Użytkownik może dodać nowy Transfer między kontami. Definiuje dla niego nazwę, czas i kwotę.
- 23) Użytkownik może dodać nową Transakcję. Definiuje dla niej Konto, tytuł, kwotę, Płatnika, Kategorię oraz datę.
- 24) Użytkownik może usunąć Transakcję.
- 25) Użytkownik może usunąć Transfer.
- 26) Użytkownik może edytować Transakcję, zmieniając: tytuł, datę, kwotę, Konto, Płatnika, Kategorię.

- 27) Użytkownik może edytować Transfer, zmieniając: tytuł, datę, kwotę, Konto wejściowe, Konto wyjściowe
- 28) Użytkownik może zobaczyć graficzną reprezentację danych finansowych.
- 29) Użytkownik może podsumować wpływy oraz wydatki w formie graficznej, jako wykres kołowy. Moduł będzie umożliwić selekcję względem typu (kategorie, płatnicy, konta) oraz czasu.
- 30) Użytkownik może porównać wpływy i wydatki w formie tabelarycznej. Tabela będzie umożliwiać selekcję względem typu (kategorie, płatnicy, konta) oraz czasu.

Wymienione wytyczne są kluczowe, gdyż opisują interakcje użytkownika, przetwarzanie danych, warunki graniczne i pozostałe aspekty, służące do zapewnienia poprawnie działającego systemu.

2.2. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne to aspekty działania systemu, które nie dotyczą bezpośrednio jego głównej funkcjonalności, lecz określają jego cechy jakościowe i parametry działania.

- 1) Aplikacja powinna działać na różnych wielkościach ekranu.
- 2) Aplikacja powinna działać na najpopularniejszych systemach operacyjnych w Polsce [7]: Windows, Android.
- 3) Aplikacja powinna odznaczać się wystarczającą wydajnością, aby obsłużyć ruch z wielu urządzeń jednocześnie oraz dynamicznie ładować duże ilości danych.
- 4) Aplikacja powinna zachowywać spójność danych.
- 5) Aplikacja powinna odznaczać się wysoką dostępnością. Maksymalnie 5 minut braku dostępu na dobę, w godzinach nocnych.
- 6) Dane powinny być przesyłane szyfrowanym połączeniem.

Przedstawione wymagania są istotne dla zapewnienia odpowiedniej jakości, wydajności i odbioru systemu przez użytkowników.

2.3. Reguły biznesowe

Reguły biznesowe to ustalone wytyczne i ograniczenia, które określają, jak aplikacja lub system działa oraz jak użytkownicy mogą z nich korzystać, w celu zapewnienia spójności, bezpieczeństwa i efektywności działań biznesowych.

Aplikacja ma za zadanie uproszczenie zarządzania finansami osobistymi poprzez zapisywanie wpływów i wydatków oraz generowanie raportów, pozwalających na znalezienie błędnych decyzji finansowych i wzmocnienie tych właściwych. Użytkownik powinien mieć możliwość analizy danych w formie graficznej, pokazujących wydatki oraz wpływy w wybranym przedziale czasowym, filtrowanym według odpowiednich kategorii. Język angielski został wybrany jako język interfejsu użytkownika.

- 1) Po upływie określonego czasu użytkownik powinien zostać wylogowany z aplikacji.
- 2) Użytkownik może posiadać Nielimitowaną ilość kont.
- 3) Użytkownik może posiadać Nielimitowaną ilość płatników.
- 4) Użytkownik może posiadać Nielimitowaną ilość Kategorii.
- 5) Użytkownik może przypisać Nielimonową ilość Podkategorii do danej Kategorii.
- 6) Pojedyncza Podkategoria może być przypisana tylko do jednej Kategorii.
- 7) Osoba niezalogowana nie może wykonywać działań na systemie, poza rejestracją lub logowaniem.
- 8) Próba pominięcia ekranu logowania powinna zakończyć się przekierowaniem na główną stronę aplikacji.
- 9) Użytkownik nie może wpływać na dane innego użytkownika.

Dzięki zastosowaniu tych reguł, osiąga się spójność, bezpieczeństwo i efektywność aplikacji. Przyczyniają się one do zabezpieczenia danych oraz zachowania prywatności. Dodatkowe ograniczenia dla osób niezalogowanych wspomagają integralność systemu.

2.4. Analiza możliwych rozwiązań

Wybrano aplikację internetową jako preferowaną architekturę na podstawie dokładnej analizy trzech możliwych rozwiązań: aplikacji desktopowej, mobilnej i webowej. Ostateczna

decyzja opiera się na spełnieniu stawianych wymagań niefunkcjonalnych oraz zapewnieniu najlepszego doświadczenia użytkownika. Przeważały za tym:

Uniwersalny dostęp.

Aplikacja webowa zapewnia użytkownikom dostęp do swoich danych finansowych z dowolnego urządzenia z dostępem do Internetu, bez konieczności instalowania dodatkowego oprogramowania. To umożliwia użytkownikom korzystanie z aplikacji na różnych urządzeniach, takich jak komputery, tablety i smartfony, zwiększając wygodę i dostępność.

Responsywność.

Dzięki współczesnym technologiom webowym, interfejsy aplikacji internetowej są responsywne i automatycznie dostosowują się do różnych rozmiarów ekranów. To pozwala użytkownikom na komfortowe korzystanie z aplikacji na urządzeniach o różnych wymiarach ekranu, co jest istotne w dzisiejszym zróżnicowanym ekosystemie urządzeń.

Łatwa aktualizacja.

Aktualizacje aplikacji webowej mogą być wprowadzane centralnie na serwerze, co eliminuje potrzebę ręcznej instalacji przez użytkowników. To przyspiesza wdrażanie nowych funkcji, poprawek błędów i zwiększa ogólne bezpieczeństwo aplikacji.

Dostępność i osiągalność.

Aplikacje webowe są dostępne bez konieczności instalacji z platform dystrybucji, takich jak App Store czy Google Play Store. Eliminuje to proces walidacji i publikacji, skracając czas potrzebny na udostępnienie aplikacji użytkownikom.

Ostatecznie, wybór aplikacji internetowej pozwala skupić się na dostarczeniu intuicyjnego i responsywnego interfejsu użytkownika, jednocześnie spełniając wymagania dotyczące dostępu, wydajności i bezpieczeństwa. Dzięki temu użytkownicy mogą wygodnie zarządzać swoimi finansami z różnych urządzeń, mając pewność, że aplikacja będzie aktualna i dostępna bez konieczności instalowania dodatkowego oprogramowania.

3. Architektura aplikacji

Ze względu na wybór aplikacji przeglądarkowej, zdecydowano się na zastosowanie architektury klient-serwer. Jest to optymalny wybór dla aplikacji webowej ze względu na wyraźne oddzielenie zadań między komponentami. Klient zajmuje się interfejsem użytkownika, serwer zarządza logiką biznesową i przetwarzaniem, a baza danych przechowuje dane. Taki podział umożliwia łatwą skalowalność, zapewnia bezpieczeństwo danych, pozwala na wieloplatformowość oraz umożliwia efektywne zarządzanie i analizę danych. Dodatkowo, architektura ta ułatwia aktualizacje oraz rozwój systemu.

Klient.

Jest to aplikacja lub urządzenie końcowe, które użytkownik używa do komunikacji z serwerem i korzystania z jego zasobów. Klient może być różnorodny, np. przeglądarka internetowa, aplikacja mobilna, program na komputerze stacjonarnym itp. Klient jest odpowiedzialny za interakcję z użytkownikiem, zbieranie jego żądań i prezentowanie odpowiedzi od serwera w sposób zrozumiały dla użytkownika. Klient może wysyłać zapytania do serwera w celu pobrania danych, przetworzenia ich lub wykonania określonej akcji.

Serwer.

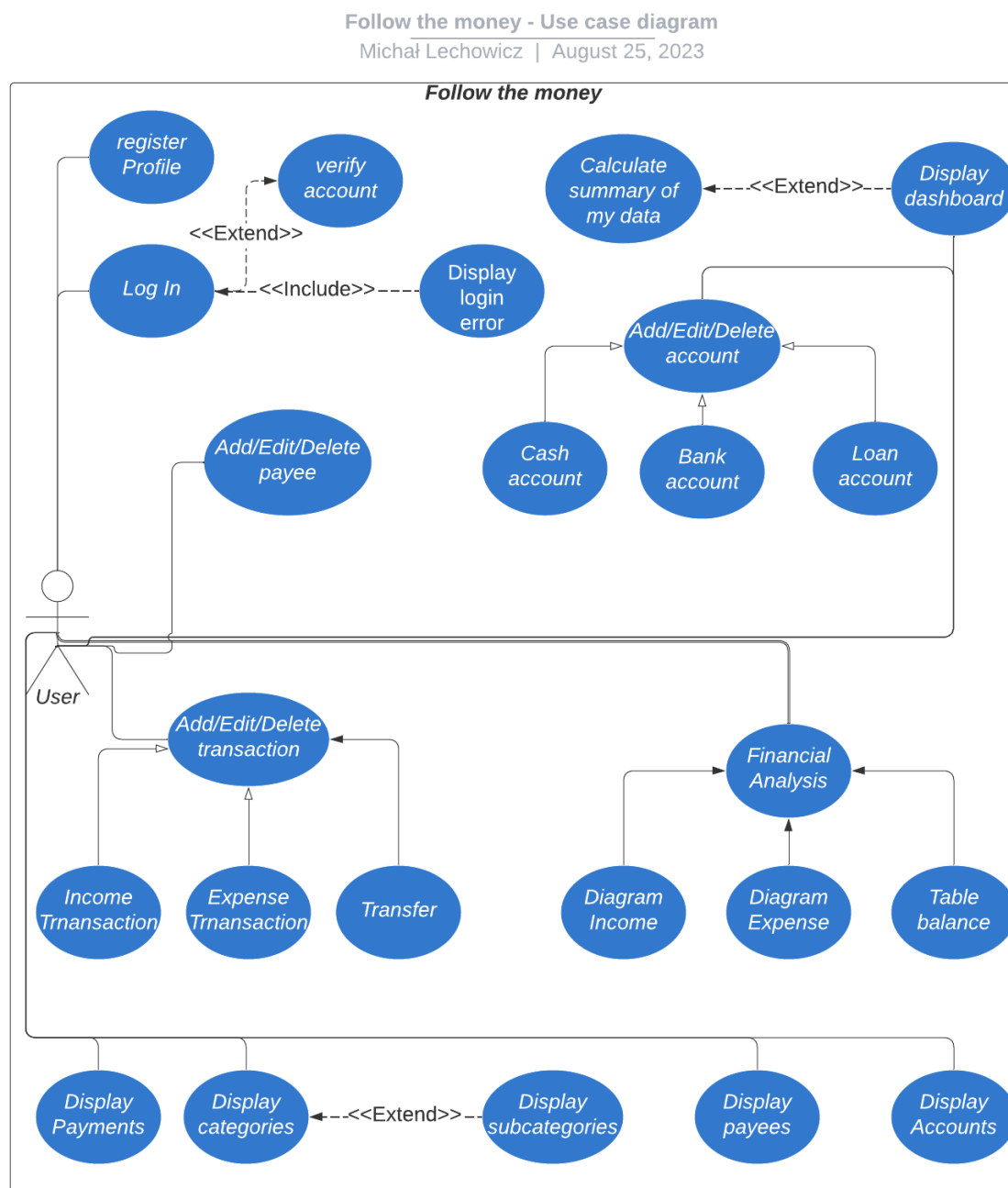
Jest to centralna część systemu, która obsługuje żądania klientów. Serwer zawiera logikę biznesową oraz dostarcza usługi, które są udostępniane klientom. To serwer odpowiada za przetwarzanie zapytań klientów, wykonywanie odpowiednich operacji oraz przesyłanie klientom odpowiedzi. Serwer może obsługiwać wielu klientów jednocześnie i monitorować ich żądania, utrzymując spójność i skuteczność działania.

Baza danych.

Jest to system przechowujący i zarządzający danymi. Baza danych jest miejscem, w którym dane są trwale przechowywane i zarządzane w sposób zorganizowany. Serwer może komunikować się z bazą danych, aby odczytywać, zapisywać, aktualizować lub usuwać dane zgodnie z żądaniami klientów.

3.1. Diagram przypadków użycia

Diagramy przypadków użycia to narzędzia używane w inżynierii oprogramowania do modelowania interakcji między użytkownikami a systemem. Skupiają się na aktorach (użytkownikach, systemach zewnętrznych), przypadkach użycia (konkretnych scenariuszach interakcji) oraz relacjach między nimi. Te diagramy ułatwiają komunikację, identyfikację wymagań, analizę zależności i stanowią podstawę dla testów systemu.

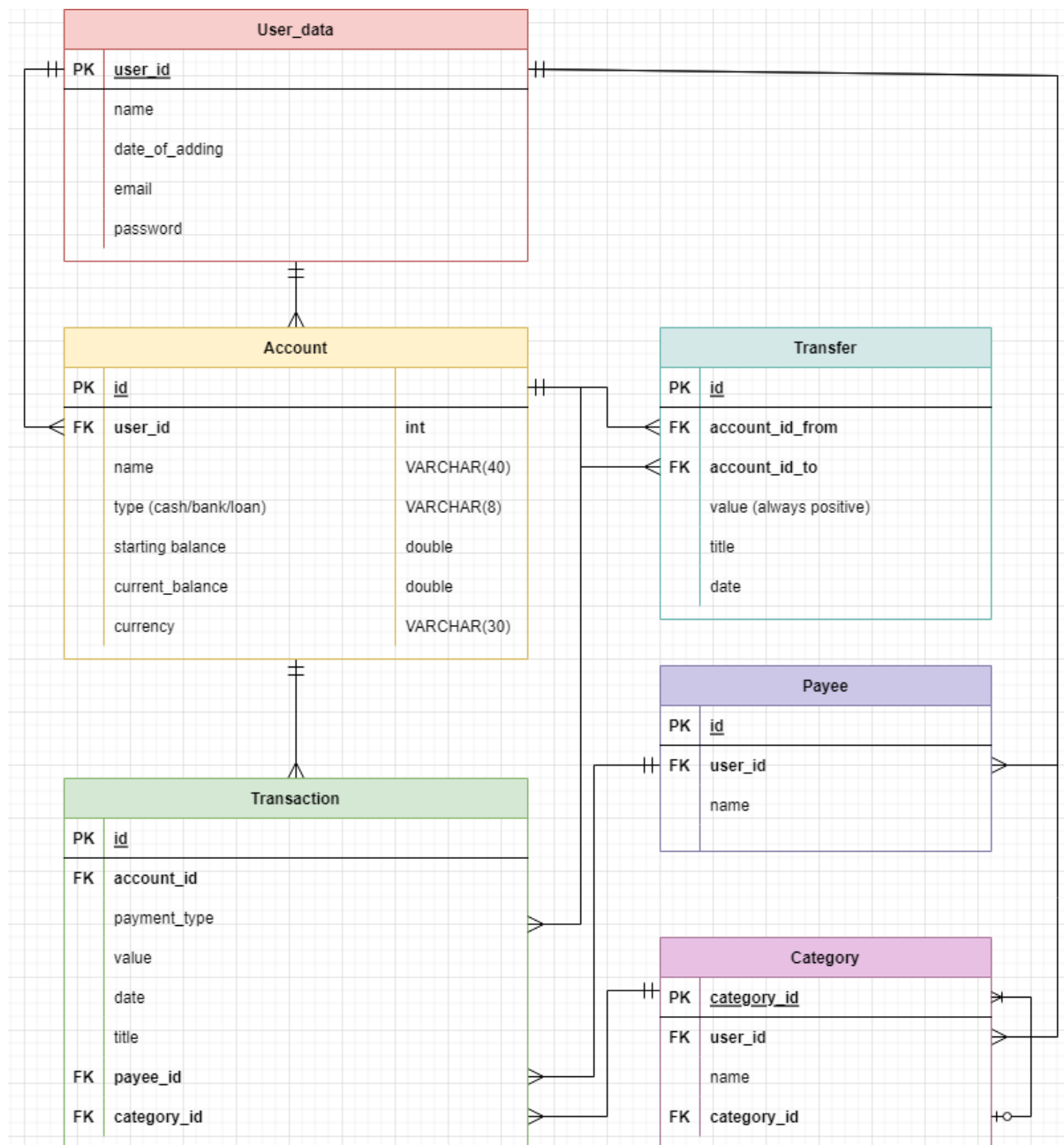


Rys. 1. Widok przypadków użycia dla Użytkownika. Opracowanie własne.

Powyżej przedstawiono diagram przypadków użycia (ang. UML use case) dla pojedynczego aktora – Użytkownika. Prezentuje on podstawowe interakcje użytkownika z systemem.

3.2. Model statyczny aplikacji – struktura bazy danych

W celu przechowywania danych zaprojektowano następującą strukturę bazodanową:



Rys. 2. Diagram encji. Opracowanie własne.

1) User_data:

- Tabela zawierająca Profile Użytkowników;
- Każdy Użytkownik posiada unikatowy *user_id*;
- Każdy Użytkownik ma przypisany unikalny *email*. Służy on do identyfikacji;
- Każdy Użytkownik posiada kolumnę *name*. Zawiera ona nazwę Użytkownika. Nie musi być unikalna;
- Każdy Użytkownik posiada pole *password*. Zawiera ono hasło służące do logowania w programie.
- Dla każdego użytkownika dodane jest pole *date_of_adding* – zawierające datę utworzenia profilu.

2) Account:

- Tabela zawierająca Konta. Każde konto jest przypisane do danego Profilu;
- Konto posiada pole: nazwę i typ;
- Konto posiada przypisaną walutę;
- Konto posiada wartość *starting_balance* – zawierającą początkowy stan konta;
- Konto posiada wartość *current_balance* – zawierającą aktualny stan konta.

3) Payee:

- Tabela zawiera Płatników. Każdy Płatnik jest przypisany do danego Profilu;
- Każdy Płatnik posiada unikatowy *id* oraz niebędącą unikatową nazwę.

4) Category:

- Tabela zawiera Kategorię. Każda Kategoria jest przypisana do danego Profilu;
- Każda Kategoria posiada unikatowy *id* oraz niebędącą unikatową nazwę.
- Kategoria zawiera klucz obcy, który można przypisać innej Kategorii. Wtedy taki wpis staje się Podkategorią.

5) Transaction:

- Tabela zawiera Transakcję. Każda Transakcja posiada unikatowy *id*;
- Każda Transakcja zawiera kwotę oraz jej typ. Typ definiuje, czy jest to wydatek czy wpływ.
- Każda Transakcja zawiera datę utworzenia oraz tytuł.
- Każda Transakcja posiada przypisane Konto, Płatnika oraz Kategorię.

6) Transfer:

- Tabela zawiera Transfery. Każdy Transfer posiada unikatowy *id*.
- Każdy Transfer posiada wartość (zawsze dodatnią), tytuł oraz datę.
- Każdy Transfer posiada informację o Koncie wypłynięcia i wpłynięcia.

Przyjęcie relacyjnej bazy danych obsługującej język zapytań SQL wydaje się odpowiednim wyborem, biorąc pod uwagę strukturę i wymagania opisanego systemu. Przemawiają za tym następujące czynniki:

Skalowalność.

W przypadku opisanego systemu finansowego, który niekoniecznie musi obsługiwać ogromne ilości danych naraz, relacyjna baza danych zapewnia odpowiednią skalowalność.

Zapewnienie spójności.

W systemach finansowych niezawodność i spójność danych są absolutnie kluczowe. W relacyjnych bazach danych operacje są atomowe, a system gwarantuje spójność danych, nawet w przypadku awarii lub błędów. To jest szczególnie istotne, gdy operujemy z danymi dotyczącymi transakcji i sald kont.

Spójny język zapytań.

Wykorzystanie języka zapytań SQL umożliwia spójne operacje na danych. Pozwala tworzyć skomplikowane zapytania, generować raporty i analizować dane.

Bezpieczeństwo.

Relacyjne bazy danych oferują mechanizmy kontroli dostępu, uwierzytelniania i audytu, które pozwalają na zarządzanie bezpieczeństwem danych. W systemie finansowym, w którym przechowywane są wrażliwe informacje, to jest niezwykle ważne.

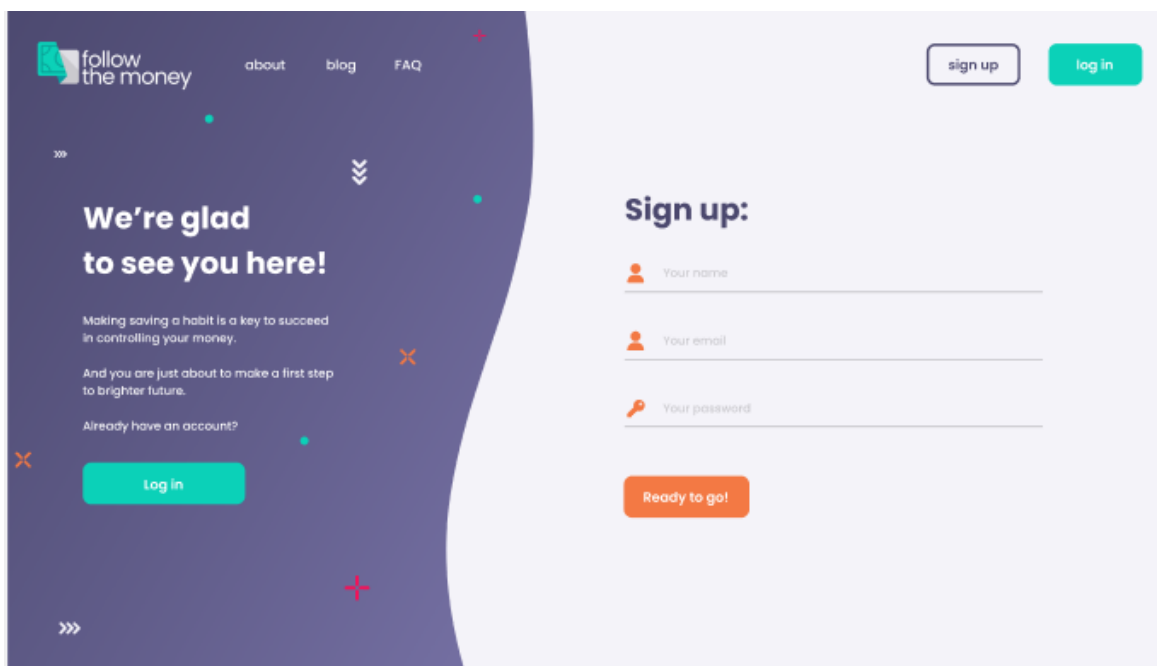
Transakcyjność.

Transakcyjność w SQL odnosi się do koncepcji gwarancji, że pewne operacje na bazie danych zostaną wykonane w sposób niepodzielny, to znaczy albo wszystkie operacje zostaną zakończone pomyślnie i zostaną zatwierdzone, albo żadna z operacji nie będzie miała wpływu na bazę danych.

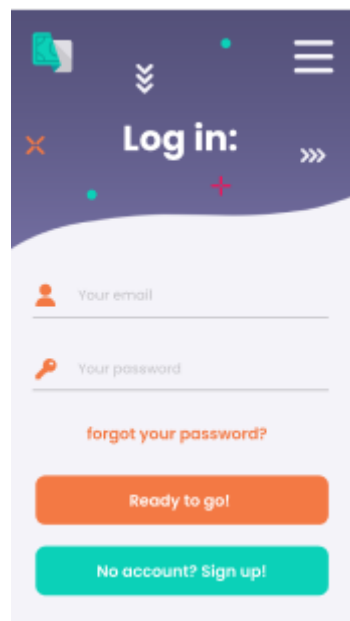
Podsumowując, wybór relacyjnej bazy danych obsługującej język zapytań SQL jest uzasadniony ze względu na kompleksową strukturę projektu, jego wymagania dotyczące spójności danych, skalowalności i bezpieczeństwa.

3.3. Projekt graficzny

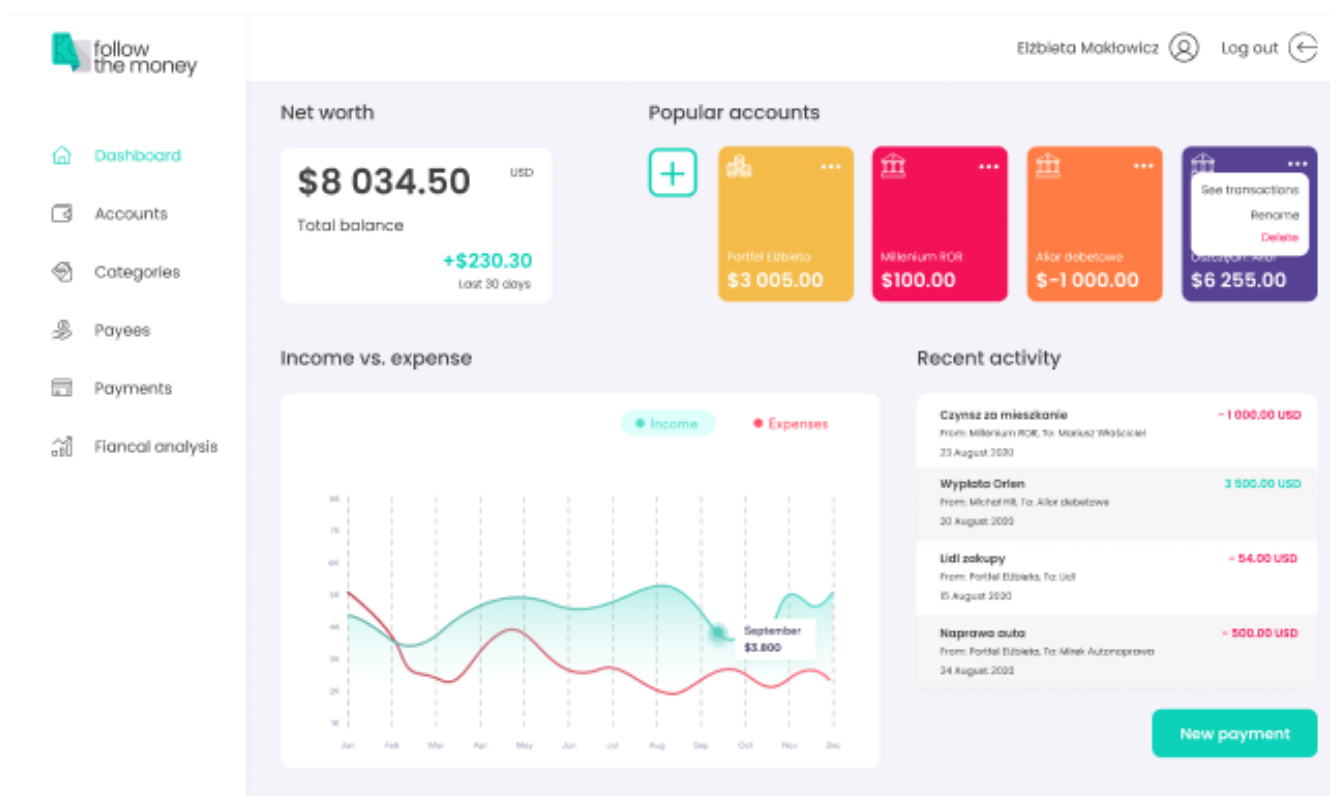
Pierwszym krokiem było zaprojektowanie projektu graficznego modułów. W tym celu użyto narzędzia Figma, ze względu na jego prostotę oraz dostępność poprzez Internet, co umożliwiło wykorzystanie go zarówno pod system Windows jak i Linux.



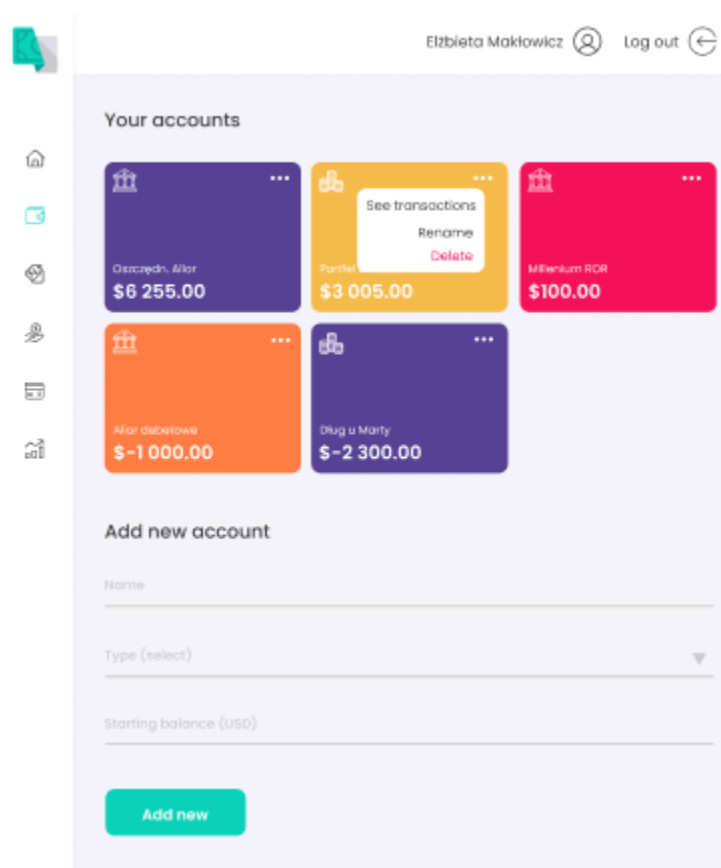
Rys. 3. Widok okna rejestracji dla komputera stacjonarnego. Opracowanie własne.



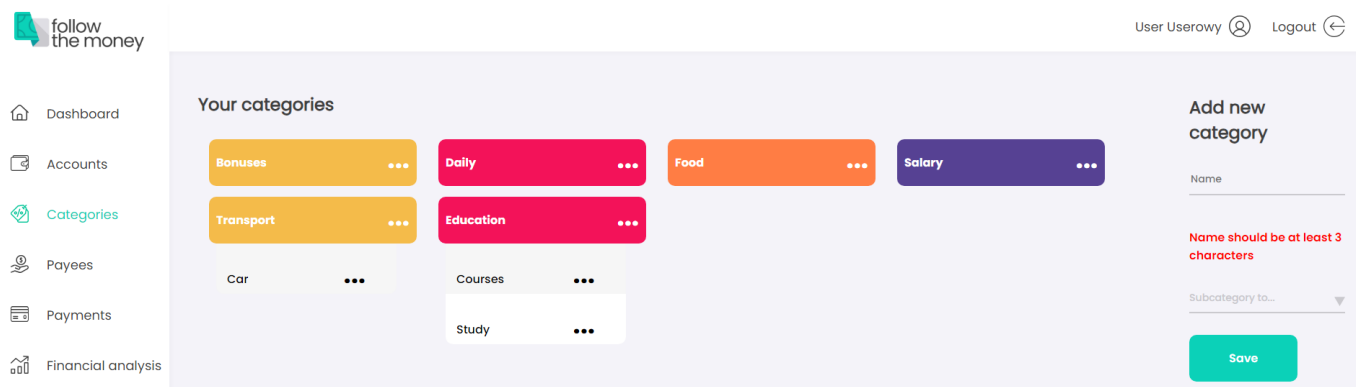
Rys. 4. Widok okna logowania dla urządzeń mobilnych. Opracowanie własne.



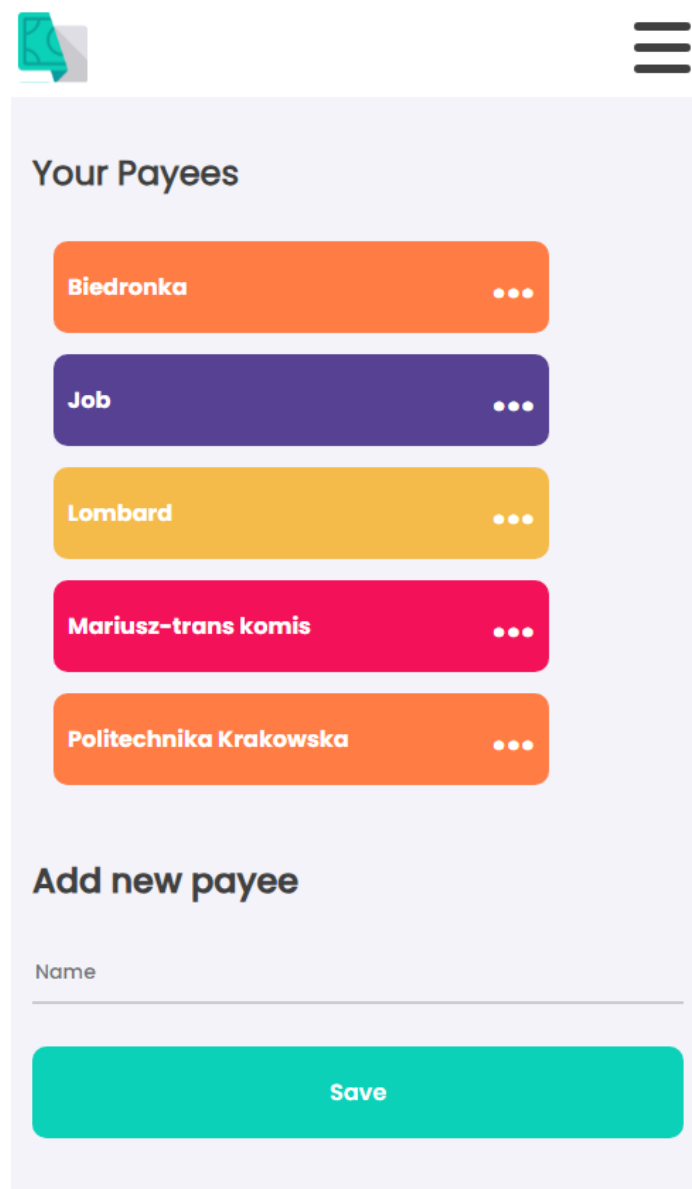
Rys. 5. Widok Tablicy finansowej dla komputerów biurowych. Opracowanie własne.



Rys. 6. Widok Kont dla tabletów. Opracowanie własne.



Rys. 7. Widok Kategorii dla komputerów stacjonarnych. Opracowanie własne.



The image shows a mobile application interface for managing payees. At the top left is a green icon of a document with a checkmark. At the top right is a hamburger menu icon consisting of three horizontal lines. The main content area has a light gray background. The title "Your Payees" is displayed in a bold, dark gray font. Below the title is a list of five payees, each represented by a colored rounded rectangle with the payee's name and three white dots on the right for more options. The payees are: "Biedronka" (orange), "Job" (purple), "Lombard" (yellow), "Mariusz-trans komis" (pink), and "Politechnika Krakowska" (orange). Below the list, the text "Add new payee" is displayed in a bold, dark gray font. Underneath this text is a text input field with the placeholder "Name". At the bottom of the form is a large teal button with the text "Save" in white.

Your Payees

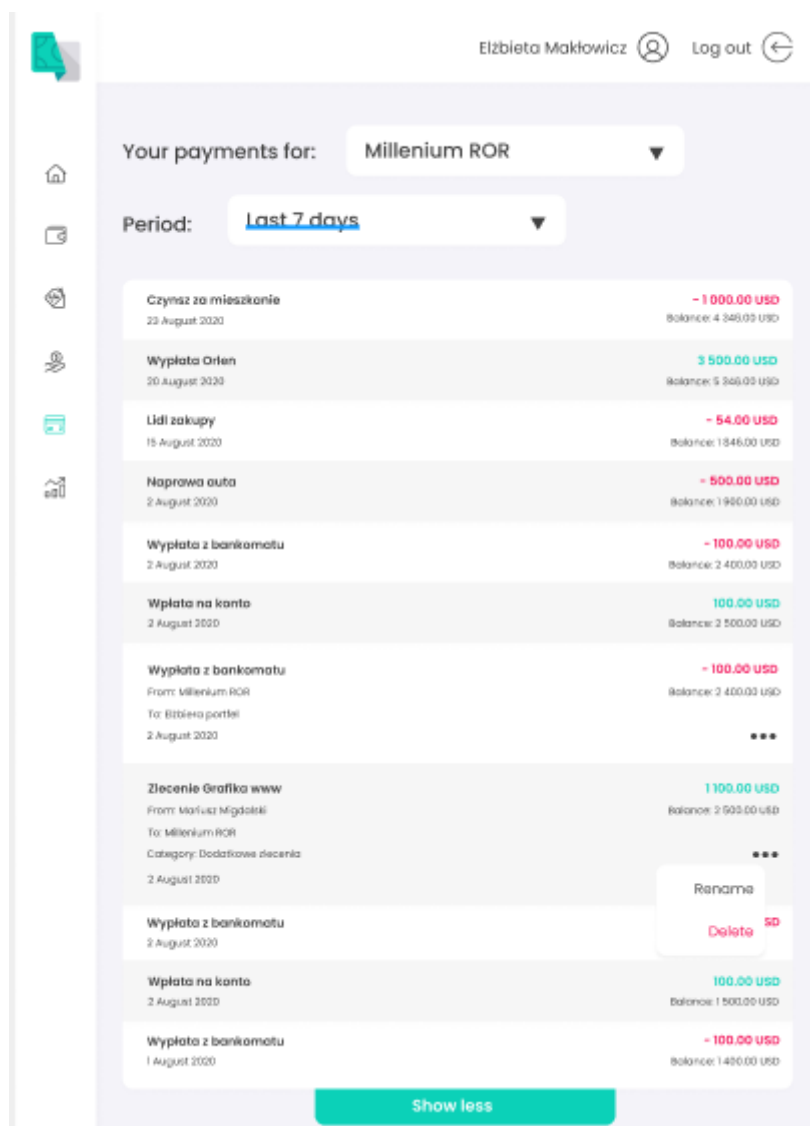
- Biedronka
- Job
- Lombard
- Mariusz-trans komis
- Politechnika Krakowska

Add new payee

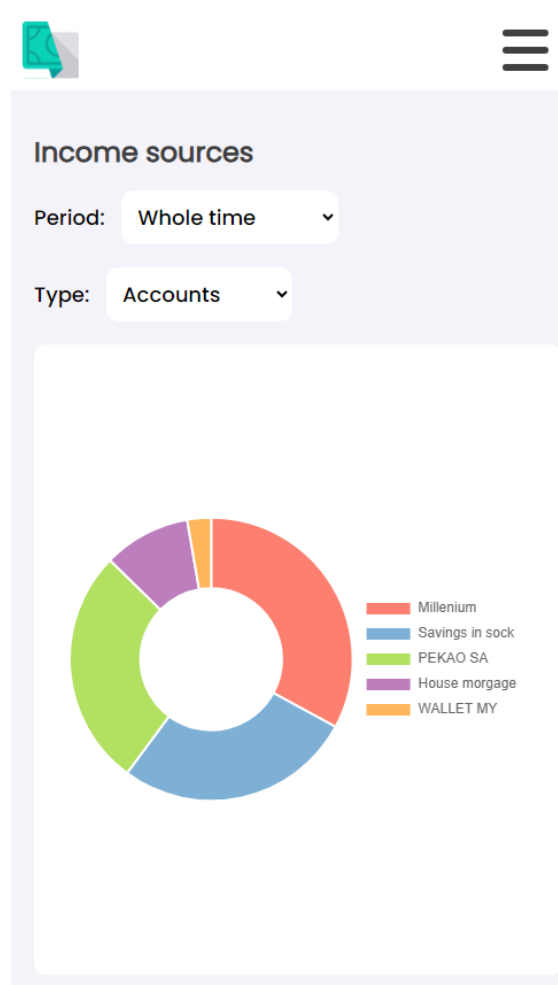
Name

Save

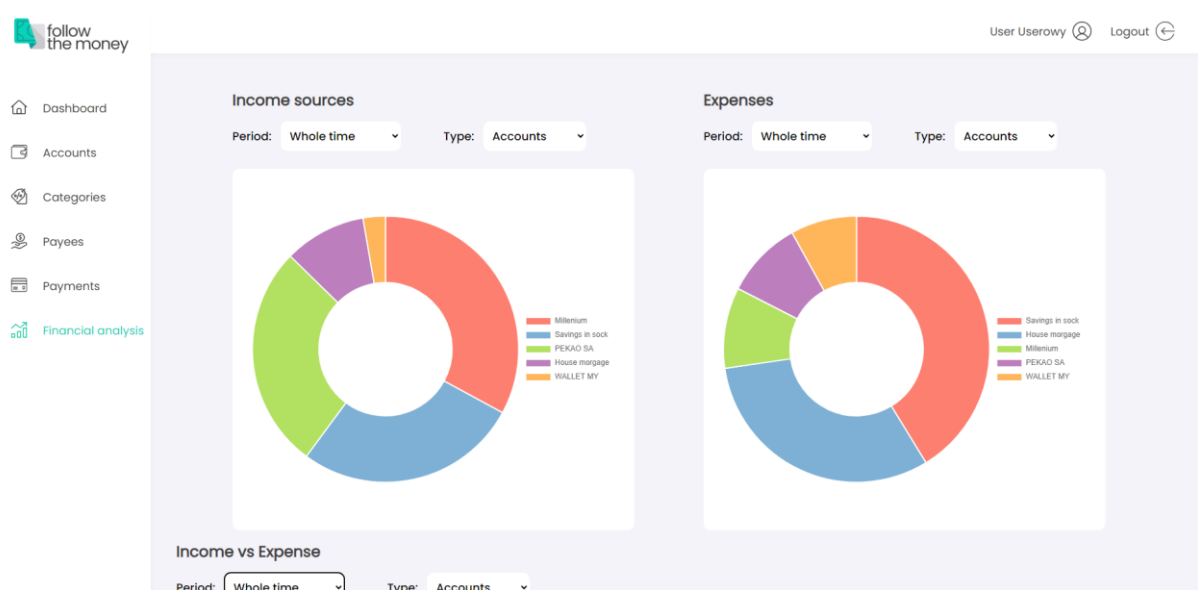
Rys. 8. Widok płatników dla urządzeń mobilnych. Opracowanie własne.



Rys. 9. Widok Płatności dla tabletów. Opracowanie własne.



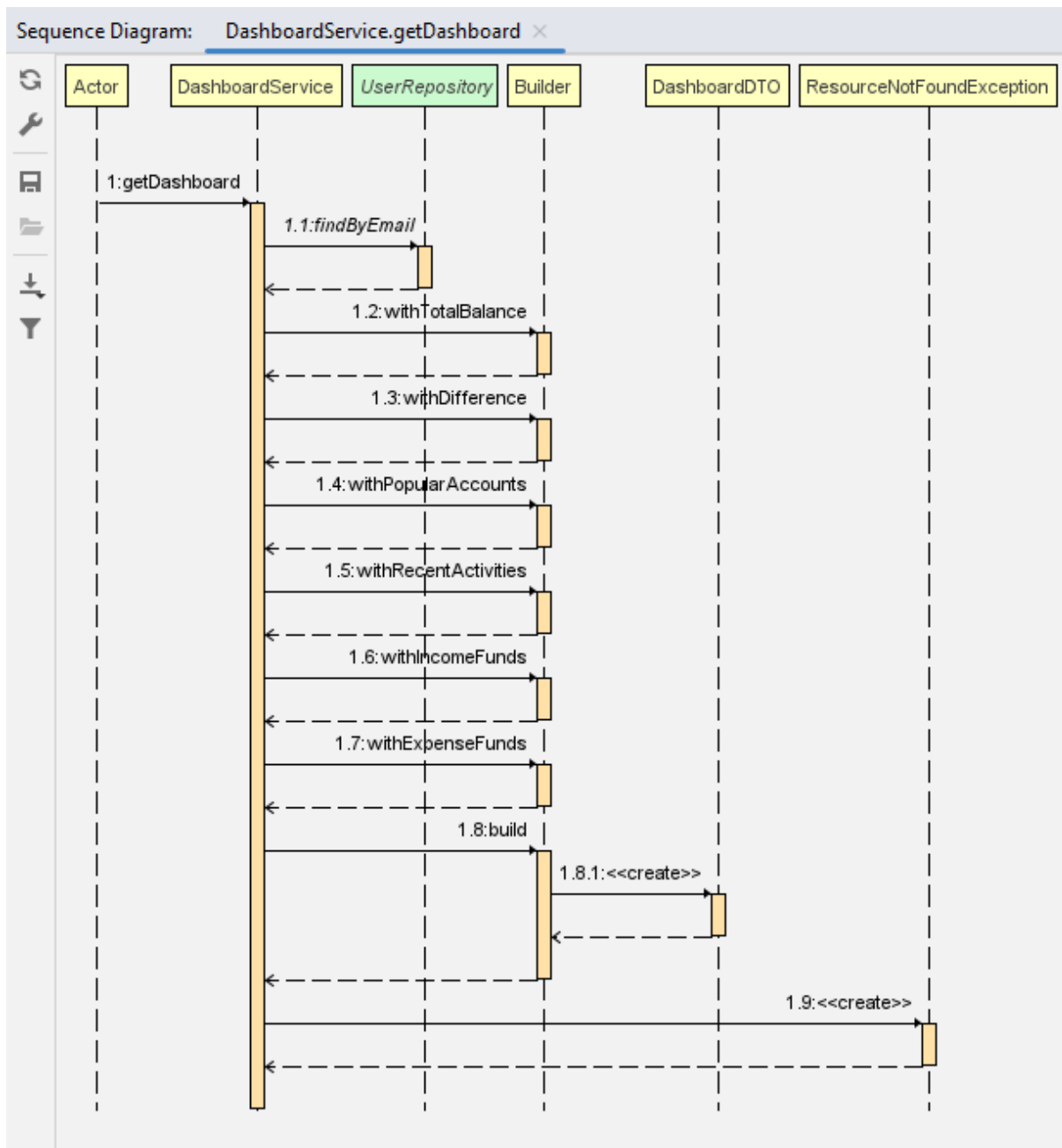
Rys. 10. Widok Analizy finansowej dla urządzeń mobilnych. Opracowanie własne.



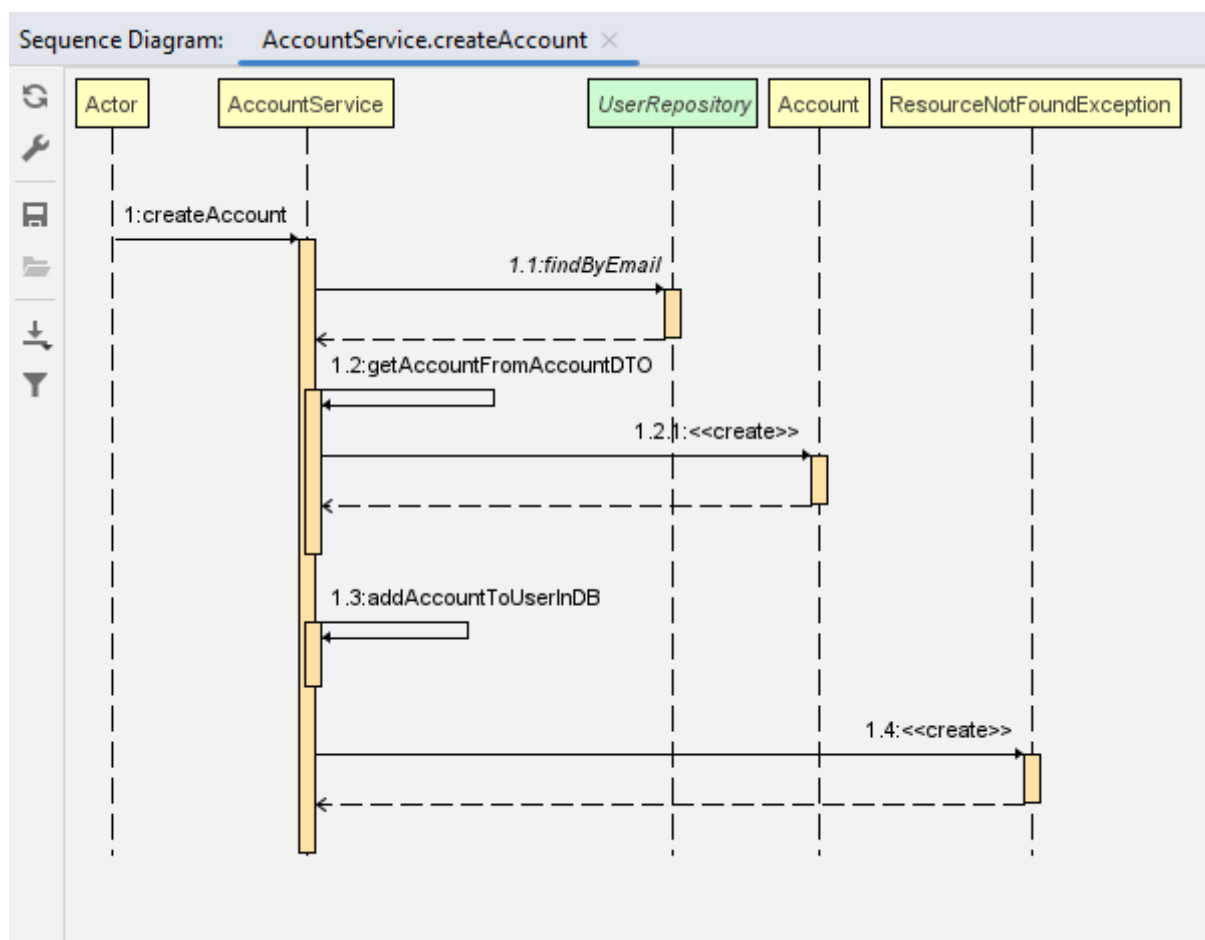
Rys. 11. Widok Analizy finansowej dla komputerów osobistych. Opracowanie własne.

3.4. Diagramy interakcji

Diagramy interakcji to rodzaj diagramów używanych w inżynierii oprogramowania do modelowania interakcji między różnymi elementami systemu lub komponentami. Są one szczególnie przydatne do opisywania interakcji między obiektami w programowaniu obiektowym oraz do modelowania procesów biznesowych w kontekście systemów informatycznych.



Rys. 12. Diagram interakcji dla generowania Tablicy finansowej. Opracowanie własne.



Rys. 13. Diagram interakcji dla tworzenia nowego Konta. Opracowanie własne.

Diagramy sekwencji to rodzaj diagramów interakcji, które przedstawiają interakcje między obiektami lub komponentami systemu w chronologicznej kolejności. Wykorzystują linie życia do reprezentowania obiektów oraz strzałki do pokazywania przepływu komunikacji. Pozwalają one wizualnie analizować, jak metody i komunikaty są wywoływane między uczestnikami, co ułatwia zrozumienie procesów interakcji i jest przydatne w projektowaniu oprogramowania.

3.5. Dobór technologii zastosowanych do implementacji

Współczesne technologie backendowe i frontendowe stanowią podstawę dla tworzenia nowoczesnych aplikacji internetowych. Zróżnicowany wybór języków i frameworków pozwala dostosować narzędzia do potrzeb projektu, oferując skalowalność, wydajność oraz bezpieczeństwo.

Do implementacji części serwerowej zdecydowano się użyć frameworka Spring Boot oraz języka Java [4]. Część kliencka zostanie napisana we frameworku Angular [2] oraz języku TypeScript. Jako bazę danych wybrano PostgreSQL [1].

Wybór Javy i Springa umożliwia podzielenie kodu aplikacji na pakiety (co ułatwia zarządzanie kodem aplikacji), a wykorzystanie Spring JPA [4] do połączenia z bazą danych ułatwia mapowanie obiektów. Jako rozwiązanie JavaScriptowe wybrano Angulara, który zapewnia łatwy podział na moduły i komponenty, a dzięki wykorzystaniu TypeScripta i statycznego typowania pomaga znaleźć błędy w kodzie jeszcze przed kompilacją projektu. Wybór PostgreSQL jest uzasadniony ze względu na jego otwartość, niskie koszty, zgodność ze standardami SQL, skalowalność, rozbudowany ekosystem narzędzi oraz aktywną społeczność.

Wykorzystując framework Spring Security [4], zapewniono zaawansowane mechanizmy uwierzytelniania i autoryzacji, które są kluczowe dla bezpieczeństwa aplikacji. Proces ten gwarantuje, że użytkownicy muszą potwierdzić swoją tożsamość przed uzyskaniem dostępu do zasobów. Wybór Spring Security zapewnia efektywne zarządzanie bezpieczeństwem aplikacji.

Ze względu na zastosowaną architekturę zdecydowano się na wybór REST API [4] do komunikacji między klientem a serwerem. Aby być zgodnym ze standardami, wybrano JWT [26] (JSON Web Token) jako model autoryzacji, gdyż gwarantuje bezstanowe połączenie. Zastosowanie JWT w aplikacji przynosi szereg korzyści w porównaniu do tradycyjnych sesji [27]. Dzięki swojej bezstanowości, JWT zawiera informacje uwierzytelniające bez potrzeby przechowywania stanu na serwerze.

4. Implementacja projektu

W niniejszym rozdziale zostanie przedstawiona implementacja projektu, włącznie ze strukturą katalogów oraz przykładem przepływu danych przez system.

4.1. Struktura projektu

Zanim przystąpiono do implementacji rozwiązania, należało przyjąć strukturę, w ramach której aplikacja ma być budowana. Zdecydowano się na zastosowanie zmodyfikowanego wzorca MVC oraz podział kodu według funkcjonalności. Projekt bazy danych opisano wyżej.

MVC(S)

Aplikacja została napisana z wykorzystaniem wzorca architektonicznego [5] MVC (Model-View-Controller). Wzorzec ten jest strukturalnym podejściem do projektowania aplikacji, które dzieli ją na trzy główne komponenty:

- Model (Model): Reprezentuje dane i logikę biznesową aplikacji. Model jest odpowiedzialny za przechowywanie informacji oraz wykonywanie operacji na tych danych. Nie jest związany bezpośrednio z interfejsem użytkownika.
- View (Widok): Odpowiada za prezentację danych użytkownikowi. Jest to warstwa odpowiedzialna za wyświetlanie informacji w formie zrozumiałej dla użytkownika. Widok otrzymuje dane z Modelu i przedstawia je Użytkownikowi.
- Controller (Kontroler): Zarządza przepływem sterowania w aplikacji. Reaguje na akcje użytkownika i aktualizuje Model oraz Widok w zależności od tych akcji. Kontroler jest pośrednikiem pomiędzy Modelem a Widokiem.

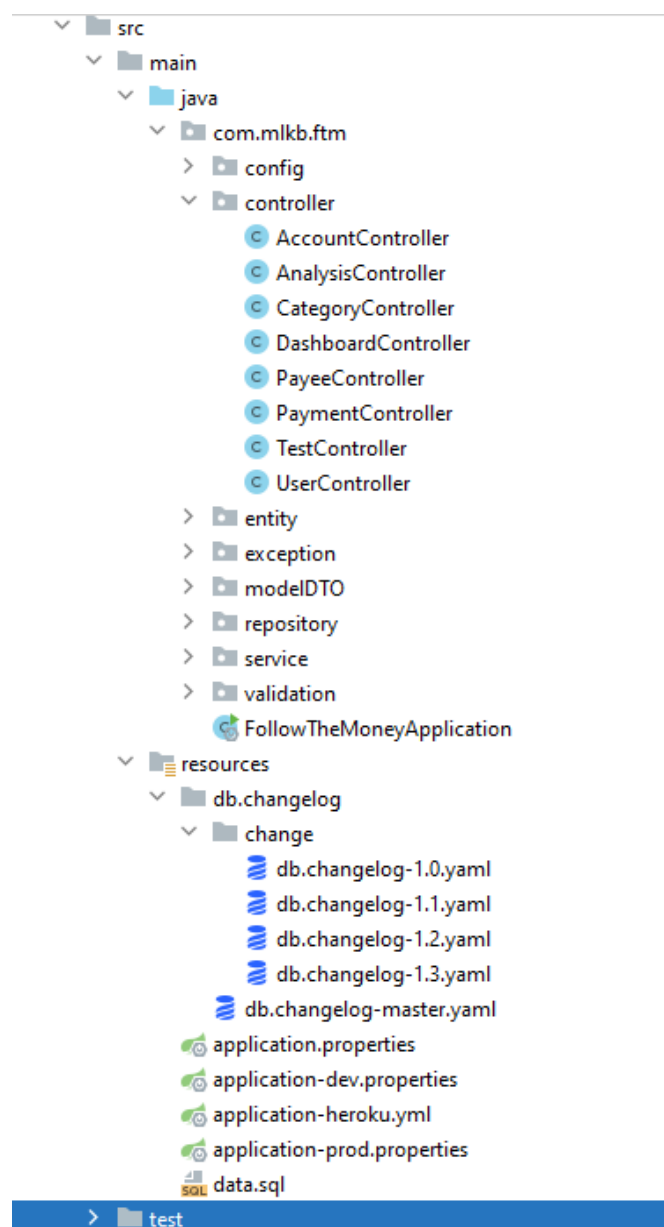
Podstawową ideą MVC jest rozdzielenie odpowiedzialności pomiędzy te trzy komponenty w celu zwiększenia modularności, skalowalności i łatwości zarządzania kodem. Dzięki temu, zmiany w jednym komponencie niekoniecznie muszą wpływać na pozostałe, co ułatwia modyfikacje, testowanie oraz rozwijanie aplikacji. Wzorzec ten jest powszechnie wykorzystywany w aplikacjach webowych, gdzie Model reprezentuje dane z bazy lub logikę biznesową, Widok odpowiada za interfejs użytkownika, a Kontroler obsługuje żądania użytkownika i koordynuje cały przepływ informacji.

Ze względu na skomplikowanie projektu, zdecydowano się na dodanie warstwy serwisowej (ang. Service). Jest to element architektury aplikacji, który działa jako pośrednik

między kontrolerami a modelami we wzorcu MVC. Głównym zadaniem serwisu jest realizacja logiki biznesowej, reguł i operacji specyficznych dla danej dziedziny problemu. Warstwa ta oddziela implementację tych operacji od kontrolerów i modeli, co przekłada się na zwiększoną modularność, czytelność oraz możliwość ponownego wykorzystania kodu.

4.2. Implementacja serwera

System został zaprojektowany według wcześniejszych założeń. Poniżej zademonstrowano fragment implementacji z ciekawszymi rozwiązaniami. Ze względu na obszerność powstałego kodu pokazano tylko niewielki fragment aplikacji.



Rys. 14. Struktura katalogów dla serwera.

Projekt serwera został zbudowany według rozszerzonego modelu MVC. Zgodnie z konwencją stosowaną w Javie, projekt został podzielony na Pakiety (pisane małą literą) [4]. W nich znajdują się klasy i interfejsy (są one zapisywane PacalCasem) [4].

Serwer – wykorzystane biblioteki.

Do budowy projektu wykorzystano Spring Boot w wersji 3.1.2. oraz Javę 17. W całym projekcie używana jest biblioteka Lombok.

Project Lombok to biblioteka programistyczna dla języka Java, która ma na celu ułatwienie tworzenia kodu poprzez automatyzację rutynowych zadań związanych z tworzeniem getterów, setterów, konstruktorów, metod *toString()*, *equals()* oraz *hashCode()*, a także obsługą adnotacji. Projekt ten dąży do zmniejszenia ilości powtarzalnego i mało istotnego kodu, co może znacznie poprawić czytelność i utrzymanie projektu. Jego wadą jest zmiana kodu, która nie jest widoczna dla programisty. Pomimo tej wady zdecydowano się go użyć, aby zwiększyć czytelność kodu.

Narzędzie do budowania projektu.

Na początku należało dokonać wyboru narzędzia do budowania projektu. Było to niezbędne, gdyż założona aplikacja miała wykorzystywać frameworki oraz biblioteki. W tradycyjnym modelu należałoby importować te zależności ręcznie, co powodowałoby, że zarówno testy jak i aktualizacja projektu, byłaby utrudniona, jeśli nie niemożliwa.

W środowisku Javy istnieją dwa konkurencyjne rozwiązania: *Maven* oraz *Gradle* [9]. Obydwa wykorzystują te same repozytoria zależności oraz posiadają zbliżone funkcje i działania. Projekt korzysta z Mavena, gdyż jest to starsze rozwiązanie, co powoduje, że ilość dostępnych materiałów jest większa niż dla Gradle.

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.2</version>
    <relativePath/>
  </parent>
  <groupId>com.mlkb</groupId>
  <artifactId>follow-the-money-server</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>follow-the-money-server</name>
  <packaging>jar</packaging>
  <description>Application for controlling budget</description>

  <properties>
    <java.version>17</java.version>
    <juniper.version>5.9.2</juniper.version>
    <liquibase.version>4.22.0</liquibase.version>
    <liquibase.propertyFile>liquibase.properties</liquibase.propertyFile>
    <testcontainers.version>1.18.0</testcontainers.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>${juniper.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

```

Rys. 15. Plik konfiguracyjny Mavena – pom.xml.

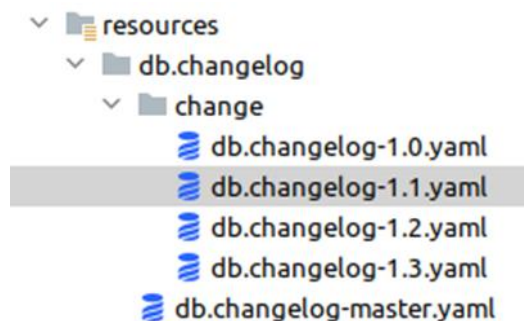
Konfiguracja Mavena zapisana jest w pliku pom.xml. W *dependencies* definiuje się zależności. Dodatkowo podaje się atrybuty projektu, jak *groupId*, *artifactId*, *version*. Można również ustawić *properties* (gdzie można zdefiniować wersję używaną dla wielu zależności).

Wersjonowanie bazy danych.

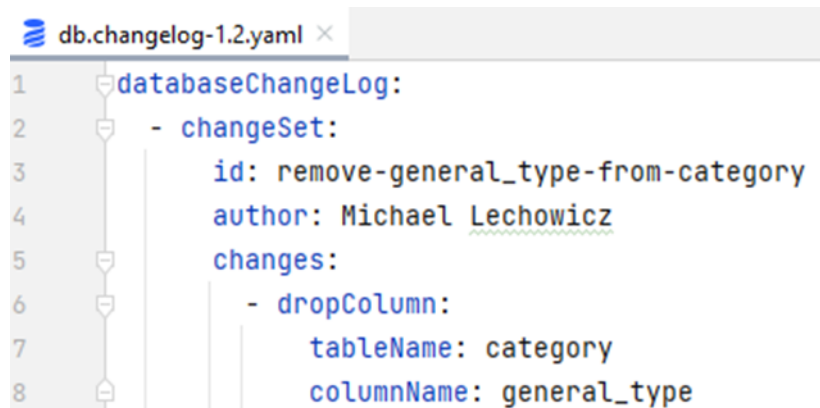
Najpierw należało dokonać wyboru narzędzia do wersjonowania. Ze względu na łatwy dostęp do dokumentacji, rozważane były dwa najpopularniejsze rozwiązania: Flyway lub Liquibase [10]. Flyway bazuje na zapisanych komendach SQL. Na jego korzyść przemawiała

prostota implementacji. Minusem jest ściśle związanie z danym dialektem SQL. Z tego powodu wybrany został Liquibase, który jest trudniejszy w implementacji w projekcie, ale gwarantuje niezaleźność od danych dialektów SQL. Jest on konfigurowany przez pliki xml, json lub yaml. W projekcie wybrano format yaml.

Po dodaniu wpisu do pliku pom.xml, należało przepisać schemat bazy danych do pliku *db.changelog-x.x.yaml*. Liquibase automatycznie śledzi zmiany w bazie danych i porównuje je z plikami *changelog*. Umożliwia to bezpieczną zmianę struktury bazy danych.



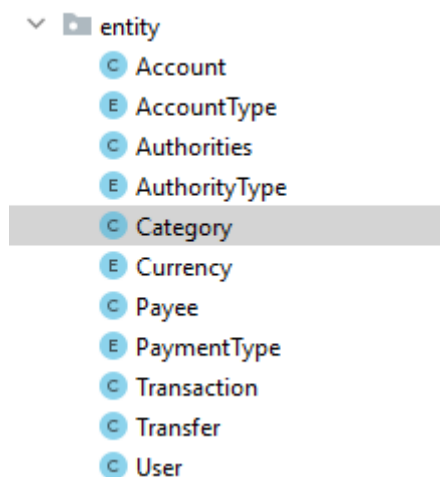
Rys. 16. Pliki z wersjami bazy danych.



Rys. 17. Plik changelog z komendą usuwającą kolumnę type z tabeli Category.

Pakiet *entity*.

Pakiet ten zawiera klasy encji [8], które służą do mapowania poszczególnych wierszy z bazy danych na pojedyncze obiekty.



Rys. 18. Pakiet entity.

Za mapowanie odpowiedzialny jest interfejs JPA (Java Persistence API). Jest to standard mapowania obiektowo-relacyjnego. Poza encjami pakiet zawiera enumy odpowiedzialne za stałe wartości, jak waluty, typ Konta czy typ Transakcji.

Jedną z jego implementacji jest Hibernate [3], który jest najpopularniejszą implementacją JPA, od 20 lat wykorzystywaną w zastosowaniach komercyjnych. Z tego powodu zdecydowano się użyć go w projekcie.

```

11  @Getter
12  @Setter
13  @NoArgsConstructor
14  @AllArgsConstructor
15  @Entity
16  @NamedEntityGraph(
17      name = "Category.subcategories",
18      attributeNodes = { @NamedAttributeNode("subcategories") }
19  )
20  public class Category {
21      @Id
22      @GeneratedValue(strategy = GenerationType.IDENTITY)
23      private Long id;
24      private String name;
25      @ManyToOne(targetEntity = Category.class, cascade = CascadeType.ALL)
26      @JoinColumn(name = "category_id")
27      private Category parentCategory;
28      private Boolean isEnabled = true;
29      @OneToMany(targetEntity = Category.class, cascade = CascadeType.ALL)
30      @JoinColumn(name = "category_id")
31      private Set<Category> subcategories;
32
33      @ManyToOne(targetEntity = User.class, cascade = CascadeType.ALL)
34      @JoinColumn(name = "user_id")
35      private User owner;
36
37      Michael Lechowicz +1
38      public Category(String name) { this.name = name; }
39
40
41      Michael Lechowicz +1
42      public Category(String name, User owner) {
43          this.name = name;
44          this.owner = owner;
45      }

```

Rys. 19. Entity Category.

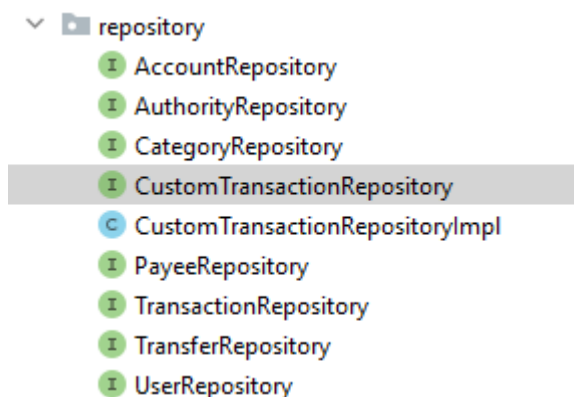
Jako przykład encji zaprezentowano *Category*. W klasie zostały wykorzystane adnotacje z Lomboka (`@Getter`, `@Setter`, `@NoArgsConstructor`, `@AllArgsConstructor`).

Adnotacja `@NamedEntityGraph` pozwala na kontrolowanie zachowania ładowania atrybutów podczas konkretnych zapytań, ale trzeba pamiętać o jego ręcznym użyciu w odpowiednich miejscach, w repozytoriach.

Wykorzystano również adnotacje `@Id`, `@OneToMany` oraz `@ManyToOne`, które mapują relację między tabelami na obiekty.

Pakiet *repository*.

Zawiera klasy i interfejsy odpowiedzialne za komunikację z bazą danych. Głównie są to interfejsy rozszerzające `JpaRepository` (będący częścią frameworku Spring) [3].



Rys. 20. Pakiet *repository*.

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Long> {
    12 usages  Michał Lechowicz
    @Query("SELECT ac " +
           "FROM User u INNER JOIN u.accounts ac WHERE ac.id = ?1 AND u.email = ?2")
    @EntityGraph(attributePaths = {"transactions"})
    Optional<Account> findByIdAndUserEmail(Long id, String email);
}
```

Rys. 21. Interfejs *AccountRepository*.

Wyjątek stanowi klasa `CustomTransactionRepositoryImpl` implementująca interfejs `CustomTransactionRepository`. Nie rozszerza ona interfejsu `JpaRepository`. Zamiast tego definiuje dwie metody zwracające mapę dla Kategorii lub Płatników. Wykorzystywane są one w Analizie finansowej.

```

public interface CustomTransactionRepository {
    2 usages 1 implementation new *
    Map<String, BigDecimal> getMapTransactionsValueForPayee(Set<Payee> payees,
                                                             PaymentType type,
                                                             Instant dateStart);

    2 usages 1 implementation Michał Lechowicz
    Map<String, BigDecimal> getMapTransactionValueForCategories(Set<Category> categories,
                                                                PaymentType type,
                                                                Instant dateStart);
}

```

Rys. 22. CustomTransactionRepository.

```

public class CustomTransactionRepositoryImpl implements CustomTransactionRepository{
    2 usages
    @PersistenceContext
    private EntityManager entityManager;

    2 usages Michał Lechowicz *
    @Override
    public Map<String, BigDecimal> getMapTransactionsValueForPayee(
        Set<Payee> payees, PaymentType type, Instant dateStart) {
        return this.entityManager.createQuery( s: """
SELECT
    SUM(t.value) AS value, p.name AS name
FROM Transaction t
INNER JOIN Payee p ON p.id = t.payee.id
WHERE t.type=:type AND p.isEnabled = true AND p IN :payees AND t.date > :date
GROUP BY p.name
""", Tuple.class) TypedQuery<Tuple>
        .setParameter( s: "type", type)
        .setParameter( s: "payees", payees)
        .setParameter( s: "date", Date.from(dateStart), TemporalType.DATE)
        .getResultStream() Stream<Tuple>
        .collect(
            Collectors.toMap(
                tuple -> (tuple.get("name")).toString(),
                tuple -> new BigDecimal((tuple.get("value")).toString())
                    .setScale( newScale: 2, RoundingMode.HALF_UP).abs()
            )
        );
}

```

Rys. 23. Implementacja interfejsu CustomTransactionRepository.

W implementacji wykorzystany jest `EntityManager`. Umożliwia on bezpośrednie połączenie do bazy danych. Dzięki zastosowaniu *JPQL* [3] (*Jakarta Persistence Query Language*), udało się zachować niezależność od dialektu PostgreSQL, co umożliwia późniejszą zmianę bazy danych w razie potrzeb. JPQL jest niezależny od konkretnego systemu zarządzania bazą danych, co oznacza, że może być używany z różnymi systemami bazodanowymi, o ile te systemy są zgodne z JPA. Metoda ta pobiera wszystkie Transakcje, do których jest przypisany dany Płatnik, filtruje je według typu Płatności i grupuje według nazw. W dalszej kolejności tworzy mapę zawierającą nazwę Płatnika oraz sumę jego Transakcji dla zadanego czasu.

W celu łatwiejszego użycia, interfejs `CustomTransactionRepository` jest implementowany przez `TransactionRepository`. Dzięki użyciu kontekstu Springa, aplikacja automatycznie wstrzykuje odpowiednią zależność. Umożliwia to wykorzystywanie wszystkich domyślnych metod z `JpaRepository`, a jednocześnie korzystanie z własnych.

```
@Repository
public interface TransactionRepository extends JpaRepository<Transaction, Long>, CustomTransactionRepository {
    2 usages  ⚡ Michael Lechowicz
    @Override
    @Modifying
    @Query("DELETE FROM Transaction t where t.id = ?1")
    void deleteById(Long aLong);

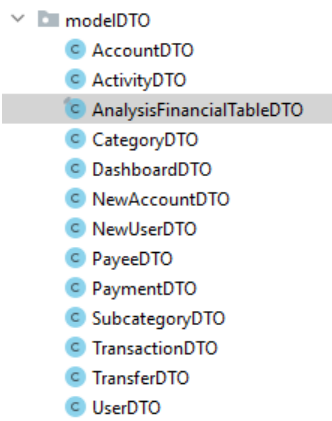
    11 usages  ⚡ Michael Lechowicz
    @Query("SELECT CASE WHEN EXISTS " +
        "( SELECT 1 " +
        "FROM User u INNER JOIN u.accounts ac INNER JOIN ac.transactions tr WHERE tr.id = ?1 " +
        "AND u.email = ?2 " +
        ") THEN true ELSE false END")
    boolean existsByTransactionIdAndUserEmail(Long Id, String email);
}
```

Rys. 24. Interfejs `TransactionRepository`.

Pakiet *modelDTO*.

Pakiet ten zawiera obiekty DTO (ang. Data Transfer Object) [11]. Są to obiekty stworzone według wzorca projektowego o tej samej nazwie. Służą do efektywnego przekazywania danych między różnymi częściami systemu lub jego komponentami. Przechowują one tylko potrzebne informacje, eliminując nadmiarowe dane. Dzięki temu unika się problemów wydajnościowych związanych z niepotrzebnym przesyłaniem informacji

oraz ułatwia skalowanie i modyfikacje aplikacji. Dodatkowo zwiększa to bezpieczeństwo, gdyż nie udostępnia się na zewnątrz wewnętrznej struktury bazy danych.



Rys. 25. Pakiet modelDTO.

```
8 @Value
9 public class AnalysisFinancialTableDTO {
10     String name;
11     BigDecimal income;
12     BigDecimal expense;
13     BigDecimal balance;
14
15     no usages
16     @Builder
17     public AnalysisFinancialTableDTO(String name, BigDecimal income, BigDecimal expense) {
18         this.name = name;
19         this.income = income;
20         this.expense = expense;
21         this.balance = income.subtract(expense);
22     }
23
24     12 usages
25     public enum AnalysisType {
26         2 usages
27         categories,
28         2 usages
29         payees,
30         accounts
31     }
32 }
```

Rys. 26. AnalysisFinancialTableDTO.

Na wyżej przedstawionym rysunku znajduje się obiekt DTO służący do demonstrowania pojedynczego wiersza tabeli podsumowania finansowego. Zawiera on nazwę wiersza, wartość wpływów, wydatków oraz typ (dla Kont, Płatników lub Kategorii).

Pakiet *service*.

Pakiet ten zawiera warstwę serwisową, która odpowiedzialna jest za operację na obiektach Entity i mapowanie ich na DTO (Data Transfer Object) [11].

```
12 @Service
13 public class PayeeService {
14     private final PayeeRepository payeeRepository;
15
16     public PayeeService(PayeeRepository payeeRepository) { this.payeeRepository = payeeRepository; }
17
18     public Set<PayeeDTO> getAllPayees(String email) {
19         return payeeRepository.getPayees(email).stream()
20             .filter(payee -> payee.getIsEnabled().equals(true))
21             .map(payee -> new PayeeDTO(payee.getId(), payee.getName()))
22             .collect(Collectors.toSet());
23     }
24
25     public void deletePayee(String email, Long id){
26         Optional<Payee> payee = payeeRepository.getPayees(email)
27             .stream()
28             .filter(findPayee -> findPayee.getId().equals(id))
29             .findFirst();
30         if (payee.isPresent()){
31             payeeRepository.setDisabled(id);
32         }
33     }
34
35     public void savePayee(PayeeDTO payee, Long userId) { payeeRepository.addPayee(payee.getName(), userId); }
36
37     public void updatePayee(String payeeName, Long payeeID) { payeeRepository.updatePayee(payeeName, payeeID); }
38 }
39
40
41
```

Rys. 27. Serwis Payee.

Klasa ta oznaczona jest adnotacją `@Service`, która dołącza ją do kontekstu Springa. Zawiera ona podstawowe operacje odczytu, zapisu, uaktualnienia oraz usunięcia. Aby zachować nazwę Płatnika w Transakcjach, zamiast tradycyjnego usunięcia z bazy danych, Płatnik oznaczany jest jako nieaktywny. Taki Płatnik nie zostanie wyświetlony na liście Płatników dla Użytkownika.

Pakiet *controller*.

Ten pakiet jest odpowiedzialny za udostępnianie punktów dostępu (ang. endpoints). Umożliwia operację na serwerze i jest napisany w zgodzie z zasadami REST API [4].

REST (czyli Representational State Transfer) jest to sposób organizowania komunikacji w sieci wraz z zestawem zasad kierujących projektowaniem aplikacji internetowych. Opracowany w 2000 roku przez Roya Fieldinga, stał się powszechnie stosowanym podejściem do tworzenia usług i interfejsów API w sieci. Opiera się na kilku podstawowych zasadach, takich jak: traktowanie wszystkiego jako zasobów (np. danych czy plików), użycie standardowych metod HTTP (np. GET, POST, PUT, DELETE) do operacji na zasobach, bezstanowość po stronie serwera pomiędzy żądaniami klienta oraz jednolity interfejs dla klienta i serwera. REST jest powszechnie używany do tworzenia interfejsów API dla aplikacji webowych, mobilnych i innych systemów rozproszonych.

```
13  @RestController
14  @RequestMapping("/api/dashboard")
15  public class DashboardController {
16      private final DashboardService dashboardService;
17      private final AccessValidator accessValidator;
18
19      public DashboardController(DashboardService dashboardService, AccessValidator accessValidator) {
20          this.dashboardService = dashboardService;
21          this.accessValidator = accessValidator;
22      }
23
24      @GetMapping("/{email}")
25      public ResponseEntity<Object> getDashboard(@PathVariable("email") String email) {
26          this.accessValidator.checkPermit(email);
27
28          try {
29              DashboardDTO dashboardDTO = dashboardService.getDashboard(email);
30              return new ResponseEntity<>(dashboardDTO, HttpStatus.OK);
31          } catch (IllegalArgumentException e) {
32              return ResponseEntity.badRequest().body(e.getMessage());
33          }
34      }
35  }
```

Rys. 28. Kontroler Tablicy Finansowej.

Prezentowany kontroler jest odpowiedzialny tylko za generowanie danych. Z tego powodu posiada tylko metodę GET. Jest on zadeklarowany jako `@RestController`. Ta adnotacja łączy w sobie `@Controller` oraz `@ResponseBody` [4]. Oznacza to, że endpoints nie zwraca strony HTML, lecz obiekt; w tym przypadku JSON.

4.3. Implementacja klienta

Do budowy projektu wykorzystano Angular 16 oraz TypeScript 4.9.5. Do rysowania wykresów wykorzystano bibliotekę chart.js w wersji 4.2.1 wraz z adapterem ng2-charts w

wersji 4.1.1, aby możliwe było użycie biblioteki w Angularze. Dodatkowo do budowy responsywnych tabel wykorzystano bibliotekę PrimeNG [25].

Ze względu na złożoność projektu zostaną zademonstrowane tylko wybrane aspekty systemu.

Struktura aplikacji klienckiej.

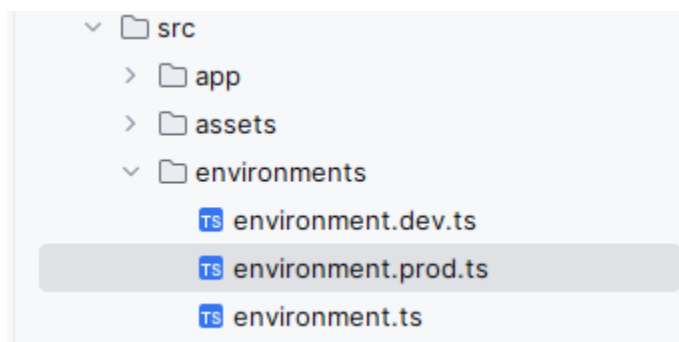
Zdecydowano się podzielić kod według funkcji, jakie ma pełnić. Osobno są przechowywane modele wykorzystywane w aplikacji klienckiej. Dla strony logowania i rejestracji zdecydowano się na stworzenie osobnego modułu. Podobnie dla właściwego portalu. Współdzielone komponenty zostały umieszczone w przeznaczonym do tego module. Warstwa serwisowa, odpowiedzialna za komunikację z serwerem, została przesunięta do osobnego modułu. Statyczne elementy (ikony, czcionki, zdjęcia) znajdują się w module assets. W ramach aplikacji klienckiej można wyodrębnić następującą strukturę:

- Folder *src* - to główny folder, w którym znajduje się większość kodu źródłowego aplikacji. Składa się z kilku kluczowych podfolderów i plików:
 - *App* - to miejsce, w którym znajduje się większość kodu źródłowego aplikacji. Zawiera komponenty, serwisy, modele danych, itp.
 - *enter-page* – zawiera w sobie stronę logowania i rejestracji. Są w nim zwarte komponenty odpowiadające funkcjonalności strony przed zalogowaniem.
 - *portal* – zawiera w sobie główną część aplikacji, dostępną po zalogowaniu.
 - *model* – przechowuje modele używane w aplikacji.
 - *service* – przechowuje serwisy, odpowiadające manipulacji na danych uzyskanych z serwera.
 - *app.module.ts* – główny moduł aplikacji, w którym rejestruje się komponenty, serwisy i inne zależności.
 - *Assets* - tutaj znajdują się pliki, takie jak: obrazy, ikony, czcionki itp., wykorzystywane w aplikacji.
 - *Environments* - zawiera konfigurację, np. zmienne środowiskowe dla różnych wdrożeń aplikacji (np. development, production).

- index.html - główny plik HTML, który zawiera punkt wejścia do aplikacji. To tu podłączane są skrypty i style.
- main.ts - plik TypeScript, który jest punktem startowym dla aplikacji. To tutaj inicjalizowane są główne moduły i komponenty.
- tsconfig.json - plik konfiguracyjny dla kompilatora TypeScript.
- tslint.json - plik konfiguracyjny dla narzędzia do statycznej analizy kodu.
- package.json i package-lock.json - pliki konfiguracyjne dla zależności i skryptów używanych w projekcie.

Podłączenie właściwego serwera.

Aby móc przełączać się między różnymi środowiskami: produkcyjnym, lokalnym, testowym, należy zmieniać adres serwera dla każdego środowiska. Adresy definiuje się w plikach w podkatalogu *environments*.



Rys. 29. Katalog *environments*.



Rys. 30. Pliki *environment.*.ts*.

Następnie należy zdefiniować nową konfigurację w pliku Angular.json.

```

39 |         "configurations": {
40 |             "production": {
41 |                 "fileReplacements": [
42 |                     {
43 |                         "replace": "src/environments/environment.ts",
44 |                         "with": "src/environments/environment.prod.ts"
45 |                     }
46 |                 ],
47 |                 "optimization": true,
48 |                 "outputHashing": "all",
49 |                 "sourceMap": false,
50 |                 "namedChunks": false,
51 |                 "extractLicenses": true,
52 |                 "vendorChunk": false,
53 |                 "buildOptimizer": true,
54 |                 "budgets": [
55 |                     {
56 |                         "type": "initial",
57 |                         "maximumWarning": "2mb",
58 |                         "maximumError": "5mb"
59 |                     },
60 |                     {
61 |                         "type": "anyComponentStyle",
62 |                         "maximumWarning": "6kb",
63 |                         "maximumError": "10kb"
64 |                     }
65 |                 ]
66 |             },
67 |             "development": {
68 |                 "fileReplacements": [
69 |                     {
70 |                         "replace": "src/environments/environment.ts",
71 |                         "with": "src/environments/environment.dev.ts"
72 |                     }
73 |                 ],

```

Rys. 31. Angular.json - podmiana plików konfiguracyjnych.

```

    getPayees(): Observable<Array<Payee>>{
        return this.http.get<Array<Payee>>({
            url: environment.apiUrl + '/api/payee/' + this.authService.getEmail(),
            options: {withCredentials: true}})
            .pipe(
                map(
                    project: data : Payee[] => {
                        const payees : Payee[] = new Array<Payee>();
                        for (const payee : Payee of data) {
                            payees.push(Payee.fromHttp(payee));
                        }
                        return payees.sort(
                            compareFn: (a : Payee , b : Payee ) => a.name.localeCompare(b.name));
                    }
                )
            );
    }
}

```

Rys. 32. Przykład wykorzystania `environment` w `PayeeService`.

Jako przykład wykorzystania `environment`, można pokazać wywołanie z serwisu. W adresie URL należałoby zmieniać domenę serwera w zależności od środowiska uruchomieniowego. Zamiast tego można odwołać się do stałej `environment.apiUrl`, którą można konfigurować w zależności od środowiska.

Aby uruchomić projekt, należy podać jako parametr odpowiednią konfigurację, co zostało zdefiniowane w pliku `package.json`.

```

4   "scripts": {
5     "ng": "ng",
6     "start": "node server.js",
7     "build": "ng build",
8     "test": "ng test",
9     "test:ci": "ng test --watch=false --browsers=ChromeHeadlessCustom",
10    "e2e": "ng e2e",
11    "postinstall": "ng build --output-path angularapp --aot --configuration production",
12    "build:prod": "ng build --configuration production",
13    --base-href /Follow-the-money-Client/
14    --deploy-url /Follow-the-money-Client/
15    --output-path=dist/Follow-the-money-Client/"
16    "build:dev": "ng build --configuration=development",
17    "lint": "ng lint"
18  },

```

Rys. 33. `Package.json` - wywołanie odpowiedniej konfiguracji.

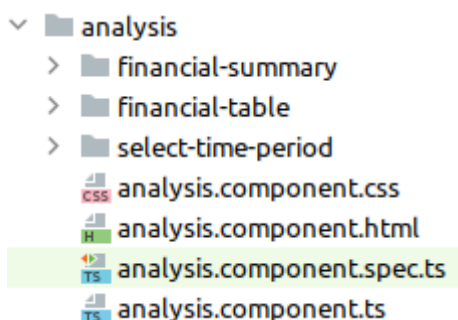
Przykład dodania modułu do Analizy finansowej.

Na początku należało dodać routing do nowego modułu w pliku `portal.module.ts`.

```
119 {  
120   path: 'analysis',  
121   component: MainPortalLayoutComponent,  
122   children: [  
123     {  
124       path: '',  
125       component: AnalysisComponent  
126     }  
127   ],  
128   canActivate : [AuthGuardService]  
129 }  
130 ];
```

Rys. 34. Dodanie routingu do pliku `portal.module.ts`.

Następnie należy przystąpić do napisania komponentu *financial-summary*, który odpowiada za generowanie diagramu. Wykorzystuje on do tego dane pobrane przez *AnalysisService* poprzez metodę *getAnalysisDataRows*. Metoda ta odwołuje się do serwera i zwraca obiekt *Observable* z listą obiektów *AnalysisTableRow*, które zawierają dane o wpływach, wydatkach, bilansie oraz nazwę Płatnika, Konta lub Kategorii.



Rys. 35. Struktura modułu do analizy danych.

Każdy nowo utworzony moduł składa się z czterech podstawowych plików: `*.spec.ts` – plik, zawierający testy komponentu; `*.css` – plik arkuszy stylów; `*.html` – plik szkieletu HTML; `*.ts` – właściwy plik klasy komponentu, kontrolujący jego stan.

```

12 export class AnalysisService {
13
14     2 usages  ⓘ Michael Lechowicz
15     constructor(private http: HttpClient,
16                  private dataService: DataService) {
17
18     5+ usages  ⓘ Michael Lechowicz *
19     getAnalysisDataRows(startDate?: string, type?: string): Observable<Array<AnalysisTableRow>> {
20         let url : string = environment.apiUrl + '/api/analysis/' + this.dataService.getEmail() + '?';
21         if (typeof startDate !== 'undefined' && startDate !== null) {
22             url += 'start=' + startDate;
23         }
24         if (type !== undefined) {
25             url += '&type=' + type;
26         }
27         return this.http.get<Array<AnalysisTableRow>>(url,
28             options: {withCredentials: true})
29             .pipe(
30                 map( project: data : AnalysisTableRow[] => {
31                     return this.extractAnalysisTableRowFromJSON(data);
32                 }),
33                 map( project: arr : AnalysisTableRow[] =>
34                     arr.sort((a: AnalysisTableRow, b: AnalysisTableRow) => a.balance - b.balance))
35             );
36     }

```

Rys. 36. Metoda `getAnalysisDataRows` w `AnalysisService`.

```

3+ usages  ⓘ Michael Lechowicz
3 updateData(eventData?: { period: number, type: string }) : void {
4     const validatedParams : {startDate: string, type: string} = this.analysisService.validateParams(eventData);
5     this.selectedType = validatedParams.type;
6
7     this.subscribeTableContent = this.analysisService
8         .getAnalysisDataRows(validatedParams.startDate, validatedParams.type).subscribe( observerOrNext: {
9             next: (res : AnalysisTableRow[] ) => this.tableData = res,
10             error: (err) => console.log('problem with getting the table rows: ', err),
11             complete: () => this.updateChart()
12         });
13 }

```

Rys. 37. Metoda `updateData` z `FinancialSummaryComponent`.

```

1 export class AnalysisTableRow {
2     name: string;
3     expense: number;
4     income: number;
5     balance: number;
6
7     1 usage  Michael Lechowicz
8     static fromHttp(dataAsJSObject): AnalysisTableRow {
9         const analysisTableRow : AnalysisTableRow = new AnalysisTableRow();
10        analysisTableRow.name = dataAsJSObject.name;
11        analysisTableRow.income = dataAsJSObject.income;
12        analysisTableRow.expense = dataAsJSObject.expense;
13        analysisTableRow.balance = dataAsJSObject.balance;
14        return analysisTableRow;
15    }
16 }

```

Rys. 38. Model *AnalysisTableRow*.

```

3 export class AnalysisBuilder {
4     private readonly _analysis: AnalysisTableRow;
5
6     constructor() {
7         this._analysis = new AnalysisTableRow();
8     }
9
10    name(name: string): AnalysisBuilder {
11        this._analysis.name = name;
12        return this;
13    }
14
15    value(valueAsString: string): AnalysisBuilder {
16        const value = Number.parseFloat(valueAsString);
17        this._analysis.expense = value;
18        this._analysis.income = value;
19        return this;
20    }
21
22    build(): AnalysisTableRow {
23        return this._analysis;
24    }
25 }

```

Rys. 39. Builder dla *AnalysisTableRow*.

Metoda *getAnalysisDataRows* wykorzystywana jest w metodzie *updateData* z *FinancialSummaryComponent* w celu pobrania danych. Metoda *updateData* przyjmuje

parametr *eventData*, który zawiera wybrane przez użytkownika *period* (ilość dni wstecz do analizy) oraz *type* (category, account, payee). Dane wprowadzone przez użytkownika są walidowane poprzez metodę *validateParams* z *AnalysisService*. Po odebraniu danych z serwera, wywoływana jest metoda *updateChart*.

```
private updateChart() : void {
    let topData : AnalysisTableRow[] = this.tableData
        .filter(this.filterZeroValue())
        .sort(this.compare());
    if (topData.length > this.maxDisplay) {
        const value : string = topData.slice(this.maxDisplay, topData.length).reduce(
            (result : number , obj : AnalysisTableRow ) => obj.expense + result , 0)
            .toFixed( fractionDigits: 2);

        topData = topData.slice(0, this.maxDisplay);
        topData.push(new AnalysisBuilder().name( name: "Other").value(value).build());
    }
    this.chartData = [{
        data: topData.map(this.mapByType()),
        backgroundColor: ["#fd7f6f", "#7eb0d5", "#b2e061", "#bd7ebe", "#ffb55a",
            "#ffee65", "#beb9db", "#fdcce5", "#8bd3c7"]
    }];
    this.chartLabels = topData.map(d : AnalysisTableRow => d.name);
}
```

Rys. 40. Metoda *updateChart*.

```

1 usage  ± Michael Lechowicz
62 private compare() {
63     if (PaymentType.INCOME == this.type) {
64         return (a, b) : number => a.income > b.income ? -1 : 1;
65     } else {
66         return (a, b) : number => a.expense > b.expense ? -1 : 1;
67     }
68 }
69
1 usage  ± Michael Lechowicz
70 private mapByType() {
71     if (PaymentType.INCOME == this.type) {
72         return d => d.income;
73     } else {
74         return d => d.expense;
75     }
76 }
77
1 usage  ± Michael Lechowicz
78 private filterZeroValue() {
79     if (PaymentType.INCOME == this.type) {
80         return d => d.income > 0;
81     } else {
82         return d => d.expense > 0;
83     }
84 }
85

```

Rys. 41. Funkcje pomocnicze dla *updateChart*.

Metoda *updateChart* aktualizuje diagram. W pierwszym kroku filtruje puste wyniki z użyciem funkcji pomocniczej *filterZeroValue*. Następnie sortuje je według funkcji pomocniczej *compare* (od największych do najmniejszych).

Dla czytelności ograniczona jest liczba wyświetlanych na diagramie danych do wartości *maxDisplay*, następnie sprawdzana jest długość otrzymanych danych. Jeśli zbiór jest większy niż *maxDisplay* (ustawione na 15), to nadmiarowe wartości są redukowane do jednego obiektu *Other*. Następnie wszystkie dane są ładowane do diagramu. Przypisywane są im kolory, dane oraz etykiety.

Po wykonaniu funkcji wyświetlany jest diagram według ustawień zdefiniowanych w polach klasy:

```

22     public chartLabels :any[] = [];
23     public chartData;
24     public chartType :string = 'doughnut';
25     public chartLegend :boolean = true;
26     public chartOptions :{scaleShowVerticalLines: boole... = {
27         scaleShowVerticalLines: false,
28         responsive: true,
29         maintainAspectRatio: false,
30         plugins: {
31             legend: {
32                 position: 'right'
33             }
34         }
35     };

```

Rys. 42. Definicja diagramu dla wpływów lub wydatków.

Jeśli dane pobrane z serwera są puste, wyświetlany jest komunikat o braku danych dla zadanego zakresu.

```

1     <h1>{{name}}</h1>
2     <app-select-time-period (updateData)="updateData($event)" />
3
4     <div class="chart">
5         <canvas baseChart
6             [datasets]="chartData"
7             [labels]="chartLabels"
8             [options]="chartOptions"
9             [legend]="chartLegend"
10            [type]="chartType"
11            *ngIf="chartData[0]['data'].length">
12        </canvas>
13        <div class="no-data" *ngIf="chartData[0]['data'].length === 0 ">
14            No Data Available
15        </div>
16    </div>
17

```

Rys. 43. Definicja komponentu dla diagramu wpływów/wydatków.

4.4. Docker

Dla obydwu części aplikacji wykorzystano narzędzie Docker [12]. Jest to platforma umożliwiająca wirtualizację na poziomie systemu operacyjnego poprzez wykorzystanie kontenerów. Docker ułatwia tworzenie, dostarczanie i uruchamianie aplikacji w izolowanych i przenośnych kontenerach. Kontenery zawierają wszystko potrzebne do uruchomienia aplikacji, co pozwala na spójne wdrażanie na różnych platformach i eliminuje problemy związane z różnicami środowiskowymi. Dzięki izolacji kontenerów aplikacje działają niezależnie, skalowanie staje się łatwiejsze, a efektywne dzielenie zasobów zapewnia większą wydajność.

Docker dla serwera.

Na potrzeby projektu utworzono plik *Dockerfile*, by móc utworzyć obraz serwera.

```
1  FROM maven:3.8-openjdk-18-slim AS MAVEN_BUILD
2
3  # copy the pom and src code to the container
4  COPY ./ ./
5
6  # package our application code
7  RUN mvn clean package
8
9  FROM openjdk:18.0.2.1-slim
10
11  COPY --from=MAVEN_BUILD /target/follow-the-money-server-0.0.1-SNAPSHOT.jar /ftm.jar
12
13  # set the startup command to execute the jar
14  CMD ["java", "-Dspring.profiles.active=prod", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/ftm.jar"]
15
```

Rys. 44. Plik *Dockerfile* dla serwera.

```

1 |version: '3.1'
2 |
3 |services:
4 |  spring-app:
5 |    container_name: ftm-server
6 |    image: ftm-server
7 |    build: ./
8 |    ports:
9 |      - "8080:8080"
10 |    environment:
11 |      FTM_DATASOURCE_URL: jdbc:postgresql://db:5432/ftm
12 |      FTM_DATASOURCE_USERNAME: admin
13 |      FTM_DATASOURCE_PASSWORD: 1234
14 |      FTM_SECRET: secretPassword
15 |      PORT: 8080
16 |    depends_on:
17 |      - db
18 |    links:
19 |      - "db:ftm_db"
20 |  db:
21 |    container_name: ftm_db
22 |    image: postgres
23 |    ports:
24 |      - "8888:5432"
25 |    environment:
26 |      - POSTGRES_PASSWORD=1234
27 |      - POSTGRES_USER=admin
28 |      - POSTGRES_DB=ftm

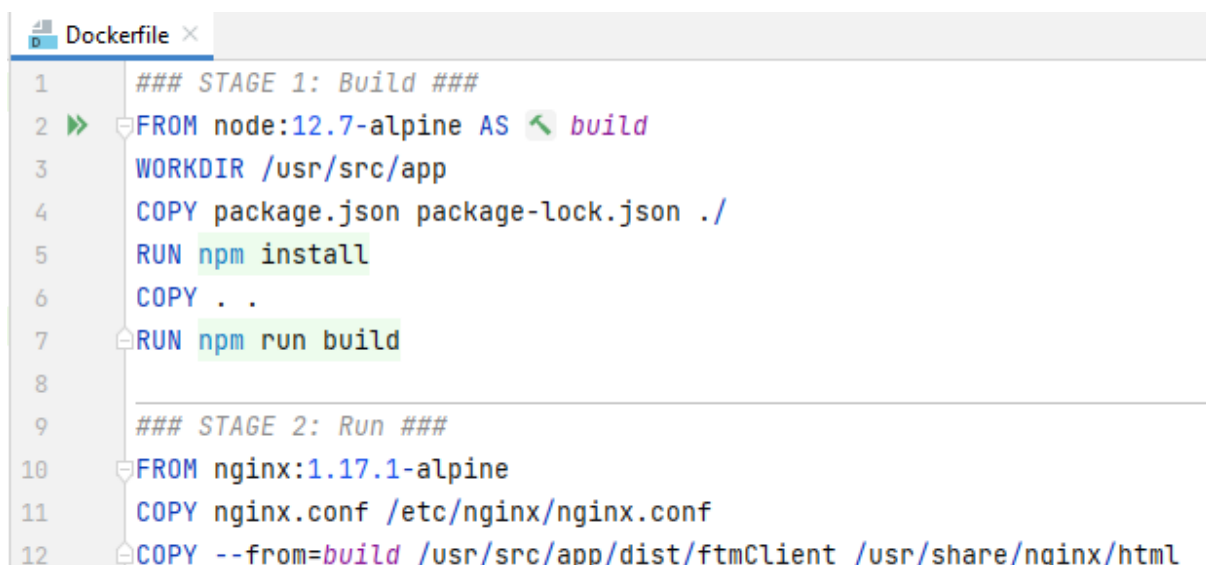
```

Rys. 45. Plik *docker-compose.yml* dla serwera.

Plik *docker-compose* umożliwia uruchomienie serwera bez potrzeby ręcznego ustawiania wszystkich potrzebnych zmiennych systemowych.

Docker dla klienta.

Na potrzeby projektu utworzono plik *Dockerfile*, by móc utworzyć obraz klienta.

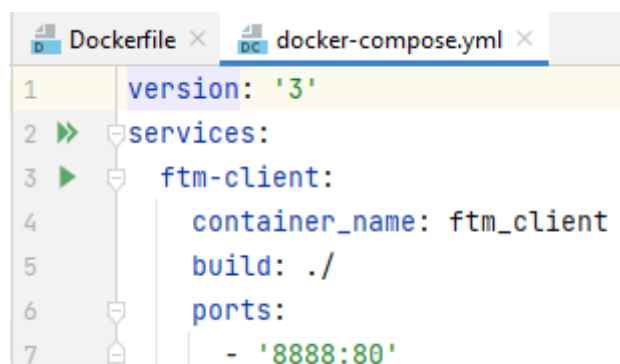


```

1  ### STAGE 1: Build ###
2  FROM node:12.7-alpine AS build
3  WORKDIR /usr/src/app
4  COPY package.json package-lock.json ./
5  RUN npm install
6  COPY . .
7  RUN npm run build
8
9  ### STAGE 2: Run ###
10 FROM nginx:1.17.1-alpine
11 COPY nginx.conf /etc/nginx/nginx.conf
12 COPY --from=build /usr/src/app/dist/ftmClient /usr/share/nginx/html

```

Rys. 46. Plik Dockerfile dla klienta.



```

1  version: '3'
2  services:
3    ftm-client:
4      container_name: ftm_client
5      build: ./
6      ports:
7        - '8888:80'

```

Rys. 47. Plik docker-compose.yml dla klienta.

Skonfigurowanie Dockera dla klienta umożliwia łatwiejsze uruchamianie aplikacji Angularowej.

4.5. Przykład przepływu danych przez aplikację

Przykład przepływu danych można zaobserwować na podstawie edycji transferu. W opisie pominięto proces logowania, wyboru płatności do edycji, ładowania danych itd. Skupiono się tylko na samym etapie edycji.

Warstwa bezpieczeństwa.

Najpierw należy przedstawić model bezpieczeństwa systemu. Rdzeniem tego modułu bezpieczeństwa jest Spring Security [4]. Jest to rozwiązanie po stronie backendu, które

zapewnia implementację zabezpieczeń w aplikacjach internetowych. Skupia się na uwierzytelnianiu (weryfikacji tożsamości użytkownika) i autoryzacji (kontrolowaniu dostępu do zasobów). Framework oferuje konfigurację opartą na kodzie Java lub XML, wykorzystuje filtry do przetwarzania żądań oraz umożliwia tworzenie dostawców uwierzytelniania i zarządzania sesją.

Pierwszym krokiem jest dodanie zależności do Spring Security. Wykonuje się to poprzez pom.xml.



```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>
<dependency>
  <groupId>com.auth0</groupId>
  <artifactId>java-jwt</artifactId>
  <version>4.3.0</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Rys. 48. Dodanie zależności Spring Security.

W następnym kroku należy utworzyć plik konfiguracyjny. Zawiera on metody potrzebne do ustawienia - które punkty dostępu mają być publicznie dostępne, a które dopiero po uwierzytelnieniu. Najważniejszą z nich jest metoda *securityFilterChain*, zwracająca obiekt o tej samej nazwie. Zawiera ona szereg ograniczeń nakładanych na ścieżki dostępu. Stosuje się je do danej metody http, URLa, dla wszystkich odwiedzających, tylko dla danej grupy użytkowników, dla nikogo. W kolejnym kroku ustawiany jest protokół bezstanowy, gdyż w aplikacji wykorzystywany będzie do uwierzytelnienia JWT token [26], zamiast sesji [27]. Następnie inicjowany jest *authenticationProvider*, w tym przypadku wykorzystuje on bazę danych PostgreSQL. Dalej ustawiany jest *authenticationFilter*. Definiuje on, które obiekty typu Filtr mają być wybrane, w zależności od tego, czy Użytkownik zalogował się poprawnie czy też nie. W razie niepoprawnego logowania zwracany jest komunikat z błędem. Jeśli

Użytkownik zalogował się poprawnie, tworzony jest JWT token. Następnie ten token jest zapisywany w ciasteczku i odsyłany klientowi. Dodatkowo tworzone jest ciasteczko *email*, zawierające adres pocztowy zalogowanego Użytkownika, który wyposażony w te informacje, może pobierać już dane z zabezpieczonych części serwera. Uprawnienia Użytkownika sprawdza *JWTAuthorizationFilter*. Testuje, czy request posiada ciasteczko z tokenem oraz czy token jest zarejestrowany w systemie. Jeśli nie, zwraca wyjątek. Za wylogowanie odpowiedzialny jest *JWTController*. Po poprawnym wylogowaniu, ciasteczka są usuwane, z kolei z kontekstu usuwany jest token.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http, HandlerMappingIntrospector introspector)
    throws Exception {
    MvcRequestMatcher.Builder mvcMatcherBuilder = new MvcRequestMatcher.Builder(introspector);

    return http.csrf(AbstractHttpConfigurer::disable).authorizeHttpRequests(auth -> auth
        .requestMatchers(mvcMatcherBuilder.pattern("/").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/login").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/register").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/logoutUser").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/swagger-ui.html").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/v2/api-docs").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/webjars/**").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/swagger-resources/**").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/swagger-ui/**").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/v3/api-docs").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.GET, "/swagger-ui/**").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.GET, "/v3/api-docs/**").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.OPTIONS, "/api/**").permitAll())
        .requestMatchers(mvcMatcherBuilder.pattern("/isLogin")).hasRole("USER")
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.GET, "/api/**")).hasRole("USER")
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.PUT, "/api/**")).hasRole("USER")
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.DELETE, "/api/**")).hasRole("USER")
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.POST, "/api/**")).hasRole("USER")
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.DELETE, "/**")).denyAll()
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.GET, "/**")).denyAll()
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.PUT, "/**")).denyAll()
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.POST, "/**")).denyAll()
        .requestMatchers(mvcMatcherBuilder.pattern(HttpMethod.OPTIONS, "/**")).permitAll()
        .anyRequest().authenticated()
        .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authenticationProvider(authenticationProvider())
        .addFilter(authenticationFilter())
        .addFilter(new JwtAuthorizationFilter(authenticationManager(this.authenticationConfiguration),
            userDetailsService(), secret, new JwtController()))
        .exceptionHandling(ex -> ex.authenticationEntryPoint(new HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED)))
        .build();
}
```

Rys. 49. Metoda *securityFilterChain*.

Część serwerowa dodatkowo posiada walidator sprawdzający, czy dany Użytkownik może uzyskać dostęp do zasobu.

```

8  @Component
9  public class AccessValidator {
10
11      38 usages  Michael Lechowicz
12      public boolean checkPermit(String email){
13          String emailFromToken = (String) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
14          if(emailFromToken == null){
15              throw new UsernameNotFoundException("You have to log in before access");
16          }
17          if(!email.equals(emailFromToken)){
18              throw new IllegalArgumentException("Access denied");
19          }
20          return true;
21      }

```

Rys. 50. AccessValidator.

Część klienta również sprawdza, czy Użytkownik ma prawo dostępu do zasobów. Wykorzystuje w tym celu dwa serwisy: *AuthService* oraz *AuthRouteGuardService*. Dodatkowo każde wywołanie do serwera musi posiadać opcję *withCredentials*, ustawioną na *true*. Wtedy załączane są pliki cookie z tokenem oraz emailem.

```

validateUser(email: string, password: string): Observable<{result: string}> {
    return this.http.post<{result: string}>({ url: environment.restUrl + '/login',
    body: {email, password}, options: {withCredentials: true}});
}

```

Rys. 51. Przykładowe wywołanie do serwera.

W chwili logowania wywoływana jest metoda *authenticate* z *AuthService*. Wysyła ona zapytanie do serwera z loginem i hasłem Użytkownika. Jeśli dane są prawidłowe, wtedy zmienia wartość *isAuthenticated* na *true*.

```

25 authenticate(name: string, password: string): void {
26   this.tryLoginEvent.emit( value: true);
27   this.dataService.validateUser(name, password).subscribe(
28     next: next : {result: string} => {
29       console.log('All cookies: ', this.cookieService.getAll());
30       this.isAuthenticated = true;
31       this.authenticationResultEvent.emit( value: true);
32       this.cookieService.set( name: 'e-mail', name);
33     },
34     error: error => {
35       this.isAuthenticated = false;
36       this.authenticationResultEvent.emit( value: false);
37       this.tryLoginEvent.emit( value: false);
38     }
39   );
40 }

```

Rys. 52. Metoda *authenticate* z *AuthService*.

AuthRouteGuardService posiada jedną metodę *canActivate*. Zwraca ona wartość *isAuthenticated* z *AuthService*. Wcześniej sprawdza, czy Użytkownik jest zautentykowany. Jeśli nie, przekierowuje Użytkownika na stronę logowania.

```

13 canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
14   if (!this.authService.isAuthenticated) {
15     this.router.navigate( commands: ['login']);
16   }
17   return this.authService.isAuthenticated;
18 }
19 }

```

Rys. 53. Metoda *canActivate* z *AuthRouteGuardService*.

Aby metoda *canActivate* działała, należy odpowiednio skonfigurować *app.module.ts*, poprzez dodanie tablicy routingu. Otwarta pozostała ścieżka do logowania, a ścieżka do portalu zabezpieczona. Próba wprowadzenia innego adresu przekieruje do strony logowania.

Edycja Transferu jako przykład działania systemu.

Na początku aplikacja wyświetla formularz z danymi Transferu. Następnie użytkownik może cofnąć edycję przyciskiem *Discard changes* lub kliknąć *Save*, aby zapisać zmiany.

Rys. 54. Okno edycji transferu.

Po kliknięciu Save, uruchamiana jest metoda `onSubmit`. Wyświetlany jest komunikat *Update a transfer...*, a następnie uruchamiana jest metoda `updateTransfer` z `TransferService`. Metoda ta zwraca obiekt `Observable`. Na nim wywoływana jest metoda `subscribe`. Jeśli edycja zakończy się poprawnie, Użytkownik zostanie przekierowany do widoku z Płatnościami, gdzie może znaleźć zedytowany Transfer. W razie niepowodzenia, Użytkownik otrzyma informację o błędzie.

```

58     onSubmit() : void {
59         this.message = 'Update a transfer...';
60         this.subscriptionPut = this.transferService
61             .updateTransfer(this.updateTransfer).subscribe( observerOrNext: {
62             next: () => this.redirectTo( uri: 'payments'),
63             error: (err) => this.message = err.message
64         });
65     }

```

Rys. 55. Metoda `onSubmit`.

```

2 usages  Michael Lechowicz
42 updateTransfer(transfer: Transfer) : Observable<void> {
43     return this.http.put<null>({
44         url: environment.restUrl + '/api/payment/transfer',
45         transfer,
46         options: {withCredentials: true}
47     });
48 }
49 }

```

Rys. 56. Metoda *updateTransfer* z *TransferService*.

Po stronie serwera request przekierowywany jest do właściwego kontrolera – odpowiada za to *ServletDispatcher* ze Springa [4]. W tym przypadku jest to *PaymentController*, a dokładnie metoda *updateTransfer*.

```

Michael Lechowicz
@PutMapping("${"/transfer}")
@ResponseStatus(HttpStatus.NO_CONTENT)
@Operation(summary = "Update transfer")
public ResponseEntity<?> updateTransfer(@RequestBody TransferDTO updateTransfer,
                                       @CookieValue(value = "e-mail", defaultValue = "none") String email)
    throws InputIncorrectException {
    accessValidator.checkPermit(email);
    paymentService.isValidUpdateTransfer(updateTransfer);
    paymentService.updateTransfer(updateTransfer, email);

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}

```

Rys. 57. Aktualizacja Transferu w Kontrolerze.

Po sprawdzeniu uprawnień oraz dokonaniu walidacji, wywoływana jest metoda *updateTransfer* z *PaymentService*. Metoda ta opatrzona jest adnotacją *@Transactional* – w razie niepowodzenia, wszystkie operacje na bazie danych zostaną cofnięte.

```

@Transactional
public void updateTransfer(TransferDTO updateTransferDTO, String email) {
    if(!this.transferRepository.existsByTransferIdAndUserEmail(updateTransferDTO.getId(), email)) {
        throw new ResourceNotFoundException(
            String.format("Transfer for id = %d does not exist", updateTransferDTO.getId()));
    }

    if(updateTransferDTO.getAccountIdTo().equals(updateTransferDTO.getAccountIdFrom())) {
        throw new IllegalArgumentException(InputValidationMessage.TRANSFER_ACCOUNTS_ID.message);
    }

    Account accountFrom = getAccountForAccountId(updateTransferDTO.getAccountIdFrom(), email);
    Account accountTo = getAccountForAccountId(updateTransferDTO.getAccountIdTo(), email);

    Transfer transfer = this.transferRepository.findById(updateTransferDTO.getId()).orElseThrow();

    transfer.setTitle(updateTransferDTO.getTitle());
    modifyCurrentBalanceInAccountsAfterUpdateTransfer(accountFrom, accountTo, transfer, updateTransferDTO);
    transfer.setValue(updateTransferDTO.getValue());
    transfer.setAccountFrom(accountFrom);
    transfer.setAccountTo(accountTo);
    transfer.setDate(updateTransferDTO.getDate());

    this.transferRepository.save(transfer);
}

```

Rys. 58. Aktualizacja Transferu w Serwisie.

```

@Repository
public interface TransferRepository extends JpaRepository<Transfer, Long> {

    2 usages  ⚙ Michael Lechowicz

    @Override
    @Modifying
    @Query("DELETE FROM Transfer t where t.id = ?1")
    void deleteById(Long aLong);

    14 usages  ⚙ Michael Lechowicz

    @Query("SELECT CASE WHEN EXISTS " +
        "( SELECT 1 " +
        "FROM User u INNER JOIN u.accounts ac INNER JOIN ac.transfersFrom tr WHERE tr.id = ?1 AND u.email = ?2 " +
        ") THEN true ELSE false END")
    boolean existsByTransferIdAndUserEmail(Long Id, String email);
}

```

Rys. 59. TransferRepository.

W ciele metody sprawdzane jest, czy dany Transfer istnieje oraz czy należy do danego Użytkownika. Jeśli nie, rzucany jest odpowiedni wyjątek. Następnie sprawdzane jest, czy nie przypisano tego samego konta jako wejściowego i wyjściowego. Później wyciąga się oba konta z bazy danych oraz Transfer. Po jego modyfikacji, dokonuje się zapisu do bazy danych.

Jeśli operacja się udała, do Użytkownika wysyłany jest kod *http 204 – No Content*. Natomiast jeśli wystąpiły błędy, Użytkownik otrzyma informację zwrotną.

5. Testy aplikacji

Ze względu na specyfikę projektu, wymagał on wielu różnych testów. Najwyższe pokrycie testami powinna mieć warstwa serwisowa serwera, gdyż błędy w tym obszarze rzutowałyby na całą aplikację. Część kliencka została przetestowana głównie pod względem właściwej budowy projektu oraz zachowania komponentów na zmiany.

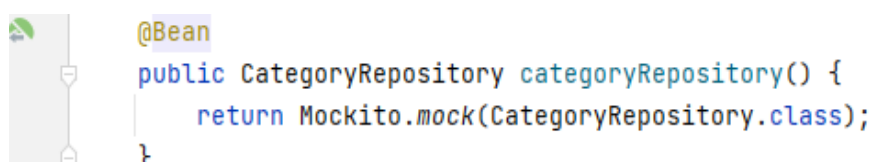
5.1. Testy jednostkowe

Objęły swym zasięgiem obie części aplikacji. Skupiają się na przetestowaniu poprawności działania jak najmniejszej części kodu w ściśle kontrolowanym i określonym scenariuszu, zakładającym izolację od innych fragmentów aplikacji [13]. W sumie napisano 83 testy jednostkowe dla serwera oraz 135 testów dla klienta. Wysoka liczba testów dla klienta, spowodowana jest tym, że test sprawdzający utworzenie komponentu jest domyślnie tworzony dla każdego z nich.

Testy serwera.

Zostały wykonane przy użyciu biblioteki JUnit 5 [14]. Aby testy zachowały swój jednostkowy charakter, wykorzystano bibliotekę Mockito [15]. Jest to popularna biblioteka do testowania jednostkowego w języku Java, umożliwiająca tworzenie tzw. Mocków, czyli obiektów, które symulują zachowanie rzeczywistych komponentów.

Ze względu na dużą ilość napisanych testów, zostaną zaprezentowane tylko niektóre z nich.



```
@Bean
public CategoryRepository categoryRepository() {
    return Mockito.mock(CategoryRepository.class);
}
```

Rys. 60. Zdefiniowanie obiektu typu mock.

```
// when
when(userRepository.existsByEmail(email)).thenReturn( t true);
when(userRepository.findByEmail(email)).thenReturn(Optional.of(user));
when(transactionRepository.findAll()).thenReturn(transactions);
when(millenium.getName()).thenReturn( t "Millenium");
when(millenium.getTransactions()).thenReturn(bankTransaction);
when(millenium.getIsEnabled()).thenReturn( t true);
when(wallet.getName()).thenReturn( t "Wallet");
when(wallet.getTransactions()).thenReturn(walletTransaction);
when(wallet.getIsEnabled()).thenReturn( t true);|
```

Rys. 61. Zdefiniowanie zachowania obiektu.

Za pomocą metody *when* możliwe jest zdefiniowanie, co dany mock powinien wywołać.

Dodatkowo utworzono specjalne klasy *Fixture*, które służą do definiowania obiektów modelowych, użytych w testach.

```
6 usages  ⚙ Michał Lechowicz
public class AnalysisFinancialTableDTOFixture {
    1 usage  ⚙ Michał Lechowicz
    public static AnalysisFinancialTableDTO bankAnalysisFinancialTableDTO() {
        return AnalysisFinancialTableDTO.builder()
            .expense(new BigDecimal( val: "55.98").setScale( newScale: 2, RoundingMode.HALF_UP))
            .income(new BigDecimal( val: "1700").setScale( newScale: 2, RoundingMode.HALF_UP))
            .name("Millenium")
            .build();
    }
}

2 usages  ⚙ Michał Lechowicz
    public static AnalysisFinancialTableDTO walletAnalysisFinancialTableDTO() {
        return AnalysisFinancialTableDTO.builder()
            .expense(new BigDecimal( val: "2527.99").setScale( newScale: 2, RoundingMode.HALF_UP))
            .income(BigDecimal.ZERO.setScale( newScale: 2, RoundingMode.HALF_UP))
            .name("Wallet")
            .build();
    }
}
```

Rys. 62. Klasa *AnalysisFinancialTableDTOFixture*.

```

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {ApplicationConfig.class})
@SpringBootTest
class AnalysisServiceTest {

    7 usages
    private AnalysisService analysisService;

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private CategoryRepository categoryRepository;

    @Autowired
    private TransactionRepository transactionRepository;

    Michał Lechowicz
    @BeforeEach
    void setUp() { analysisService = new AnalysisService(userRepository, categoryRepository, transactionRepository); }

    Michał Lechowicz
    @Test
    void should_return_set_of_AnalysisFinancialTableDTO_for_correct_credentials() {...}

    Michał Lechowicz
    @Test
    void should_return_only_enabled_accounts() {...}

    Michał Lechowicz
    @Test
    void should_return_set_of_AnalysisFinancialTableDTO_for_correct_credentials_withDateStart_param() {...}

```

Rys. 63. Testy serwisu AnalysisService.

```

@Test
void should_return_set_of_AnalysisFinancialTableDTO_for_correct_credentials() {
    // given
    final String email = "user@user.pl";
    final User user = new User();
    Account millenium = mock(Account.class);
    Account wallet = mock(Account.class);
    user.setEmail(email);
    user.setAccounts(Set.of(millenium, wallet));
    final var transactions = List.of(
        TransactionEntityFixture.buyCarTransaction(),
        TransactionEntityFixture.buyMilkTransaction()
    );
    final var walletTransaction = Set.of(TransactionEntityFixture.buyCarTransaction(),
        TransactionEntityFixture.billiardTransaction(), TransactionEntityFixture.buySugarTransaction());
    final var bankTransaction = Set.of(TransactionEntityFixture.buyMilkTransaction(),
        TransactionEntityFixture.abonamentAWSTransaction(), TransactionEntityFixture.salaryTransaction());

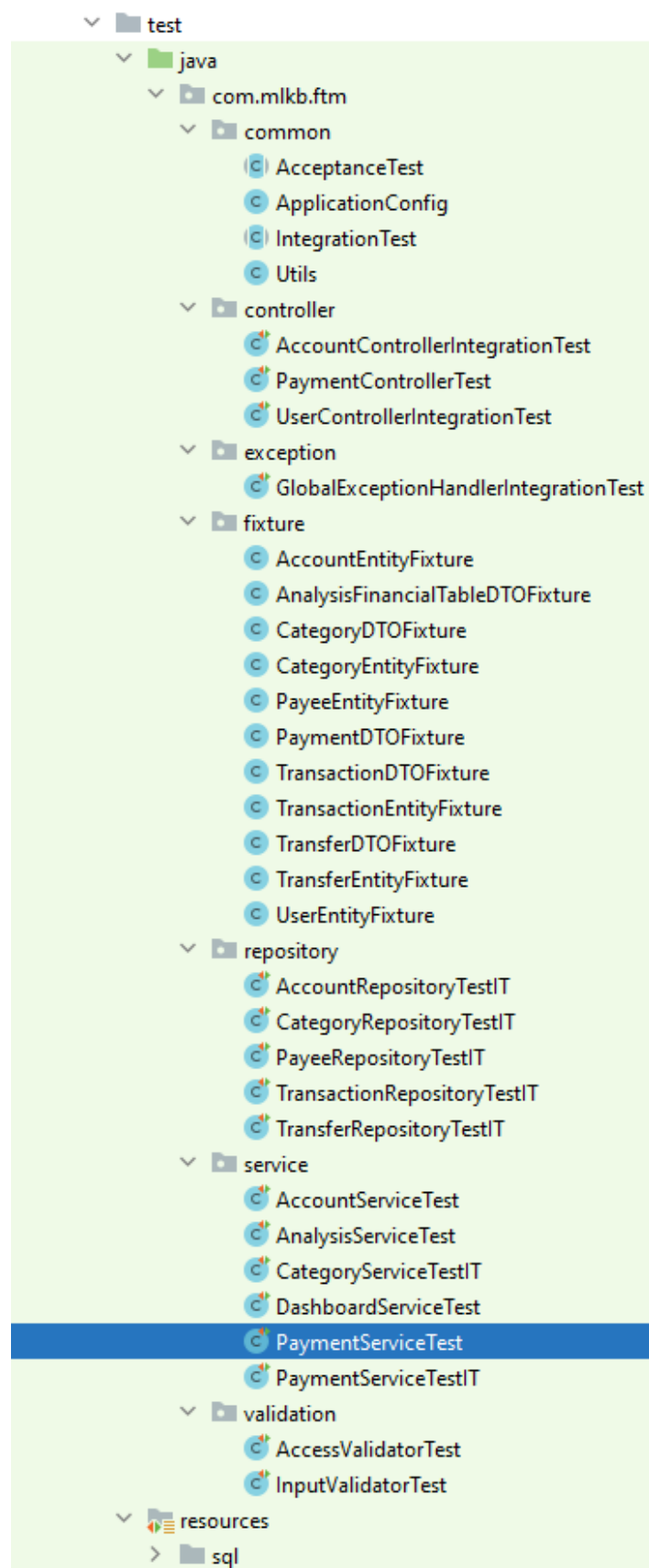
    // when
    when(userRepository.existsByEmail(email)).thenReturn(true);
    when(userRepository.findByEmail(email)).thenReturn(Optional.of(user));
    when(transactionRepository.findAll()).thenReturn(transactions);
    when(millenium.getName()).thenReturn("Millenium");
    when(millenium.getTransactions()).thenReturn(bankTransaction);
    when(millenium.getIsEnabled()).thenReturn(true);
    when(wallet.getName()).thenReturn("Wallet");
    when(wallet.getTransactions()).thenReturn(walletTransaction);
    when(wallet.getIsEnabled()).thenReturn(true);

    final var tableData = analysisService.getTableData(
        email,
        Instant.ofEpochMilli(0L),
        AnalysisFinancialTableDTO.AnalysisType.accounts
    );

    // then
    assertThat(tableData).asInstanceOf(AbstractCollectionAssert.capture of ?, Collection<...>, AnalysisFinancialTableDTO, ObjectAssert<...>)
        .hasSize(expected: 2) capture of ?
        .containsExactlyInAnyOrder(
            AnalysisFinancialTableDTOFixture.walletAnalysisFinancialTableDTO(),
            AnalysisFinancialTableDTOFixture.bankAnalysisFinancialTableDTO()
        );
}

```

Rys. 64. Test służący do wygenerowania zbioru *AnalysisFinancialTableDTO*.



Rys. 65. Struktura katalogów testowych.

Testy klienta.

Testy zostały wykonane przy użyciu Jasmine oraz Karma [16]. Jasmine to framework do testowania kodu napisanego w języku JavaScript/TypeScript, wspierający podejście BDD (Behavior-Driven Development) [17], który umożliwia tworzenie czytelnych i zrozumiałych testów opisujących zachowanie komponentów i usług w aplikacji. Z kolei Karma jest narzędziem automatyzującym testy jednostkowe, umożliwiającym uruchamianie testów w różnych przeglądarkach i środowiskach.

Jako przykład może posłużyć zbiór testów dla komponentu służącego do analizy finansowej. Najpierw należy zamockować oraz zainicjować komponent.

```
10 >> describe( description: 'FinancialSummaryComponent', specDefinitions: () : void => {
11     let component: FinancialSummaryComponent;
12     let fixture: ComponentFixture<FinancialSummaryComponent>;
13     let mockService;
14
15     beforeEach( action: async () : Promise<void> => {
16         spyDataServiceGetEmail();
17
18         mockService = jasmine.createSpyObj( baseName: 'AnalysisService',
19             methodNames: ['getAnalysisDataRows', 'validateParams']);
20         mockService.getAnalysisDataRows.and.returnValue(of( value: []))
21         mockService.validateParams.and.returnValue(
22             of( value: {startDate: '1970-01-01', type: 'accounts'}))
23         await TestBed.configureTestingModule( moduleDef: {
24             declarations: [ FinancialSummaryComponent ],
25             imports: [
26                 HttpClientTestingModule, RouterTestingModule
27             ]
28         })
29         .compileComponents();
30
31         fixture = TestBed.createComponent(FinancialSummaryComponent);
32         component = fixture.componentInstance;
33         component.chartData = [{
34             data: []
35         }];
36         fixture.detectChanges();
37     });
```

Rys. 66. Inicjalizacja komponentu.

W tym przypadku, z użyciem frameworku Jasmine, należało zamockować dwie metody z *AnalysisService*. Jedną służącą do sprawdzania poprawności parametrów i drugą do pobrania danych z serwera. Następnie w liniijkach 19-20 wskazać, co powinny zwracać te metody. Dalej następuje konfiguracja komponentu z podaniem właściwych zależności do wstrzyknięcia do testowanego modułu.

```
37 ► it( expectation: 'should create', assertion: () : void => {
38     expect(component).toBeTruthy();
39 });
40
41 ► it( expectation: 'should display h1 as name value', assertion: () : void => {
42     // given
43     const name : "Income" = 'Income';
44     // when
45     component.name = name;
46     fixture.detectChanges();
47     const displayName = fixture.debugElement.nativeElement.querySelector('h1').innerHTML;
48     // then
49     expect(displayName).toBe(name);
50 });
51
52 ► it( expectation: 'should display none when name is not set', assertion: () : void => {
53     // given
54     const name : "" = '';
55     // when
56     const displayName = fixture.debugElement.nativeElement.querySelector('h1').innerHTML;
57     // then
58     expect(displayName).toBe(name);
59 });
60
61 ► it( expectation: 'should display No data message if summary is empty', assertion: () : void => {
62     // given
63     const message : "No Data Available" = 'No Data Available';
64     // when
65     fixture.detectChanges();
66     const divElement : DebugElement[] = fixture.debugElement.queryAll(By.css( selector: '.no-data'));
67     const displayName = divElement[0].nativeElement.innerHTML;
68     // then
69     expect(displayName).toContain(message);
70 });
71 });
72
```

Rys. 67. Testy komponentu *FinancialSummaryComponent*.

W pierwszym teście wykonywane jest sprawdzenie, czy komponent został poprawnie zainicjowany. Dwie pozostałe sprawdzają, czy komponent posiada poprawnie przypisaną nazwę oraz czy wyświetla komunikat o braku danych dla pustego zbioru.

Wykonano również testy dla klasy *AnalysisBuilder*. Dla buildera zdecydowano się na przetestowanie, czy tworzy on obiekt *Other* zgodnie z wymaganiami, który powinien posiadać nazwę oraz tę samą wartość dla wydatków i wpływów. Wartość *balance* nie powinna być wyliczana.

```
3 describe( description: 'AnalysisBuilder', specDefinitions: () : void => {
4   it( expectation: 'should build AnalysisTableRow with name and value', assertion: () : void => {
5     // given
6     const name : "Example" = 'Example';
7     const value : "12" = '12';
8     const valueAsNumber : 12 = 12;
9
10    // when
11    const analysisRow : AnalysisTableRow = new AnalysisBuilder()
12      .name(name)
13      .value(value)
14      .build();
15
16    // then
17    expect(analysisRow.name).toBe(name);
18    expect(analysisRow.income).toBe(valueAsNumber);
19    expect(analysisRow.expense).toBe(valueAsNumber);
20    expect(analysisRow.balance).toBeUndefined();
21  });
22 } );
23
```

Rys. 68. Test *AnalysisBuilder*.

Dla *AnalysisService* zdecydowano się na przetestowanie działania funkcji *validateParams*. Powinna ona zwrócić zwalidowane argumenty, które posłużą do zapytania, które zostanie wysłane do serwera.


```

6  ► describe( description: 'AnalysisService', specDefinitions: () : void => {
7    let service: AnalysisService;
8    const startingTime : "2022-05-16" = '2022-05-16';
9
10   beforeEach( action: () : void => {
11     jasmine.clock().install();
12     jasmine.clock().mockDate(new Date(startingTime));
13
14     TestBed.configureTestingModule( moduleDef: {
15       imports: [
16         HttpClientTestingModule
17       ]
18     });
19     service = TestBed.inject(AnalysisService);
20   });
21
22   afterEach( action: () : void => {
23     jasmine.clock().uninstall();
24   });
25
26  ► it( expectation: 'should be created', assertion: () : void => {
27    expect(service).toBeTruthy();
28  });

```

Rys. 69. Inicjalizacja serwisu w celu testów.

```

31 ► it( expectation: 'should return correct params for correct input', assertion: () : void => {
32     // given
33     const input : {period: number, type: string} = {period: 365, type: 'payees'};
34     const expectedOutput : {startDate: string, type: string} = {startDate: '2021-05-16', type: 'payees'};
35     // when
36     const output : {startDate: string, type: string} = service.validateParams(input);
37     // then
38     expect(output).toEqual(expectedOutput);
39 });
40
41 ► it( expectation: 'should return default date for incorrect input', assertion: () : void => {
42     // given
43     const input : {period: number, type: string} = {period: -1, type: 'payees'};
44     const expectedOutput : {startDate: string, type: string} = {startDate: '1970-01-01', type: 'payees'};
45     // when
46     const output : {startDate: string, type: string} = service.validateParams(input);
47     // then
48     expect(output).toEqual(expectedOutput);
49 });
50
51 ► it( expectation: 'should return default type for incorrect input', assertion: () : void => {
52     // given
53     const input : {period: number, type: any} = {period: 365, type: undefined};
54     const expectedOutput : {startDate: string, type: string} = {startDate: '2021-05-16', type: 'accounts'};
55     // when
56     const output : {startDate: string, type: string} = service.validateParams(input);
57     // then
58     expect(output).toEqual(expectedOutput);
59 });
60
61 ► it( expectation: 'should return default params for undefined', assertion: () : void => {
62     // given
63     const input = undefined;
64     const expectedOutput : {startDate: string, type: string} = {startDate: '1970-01-01', type: 'accounts'};
65     // when
66     const output : {startDate: string, type: string} = service.validateParams(input);
67     // then
68     expect(output).toEqual(expectedOutput);
69 });
70 });

```

Rys. 70. Testy serwisu.

5.2. Testy integracyjne

Zostały przeprowadzone w części serwerowej. Objęły swym zasięgiem część serwisową w celu sprawdzenia, czy metody wykonują zmiany na bazie danych. Poza tym wykonano testy repozytoriów w celu sprawdzenia, czy napisane metody działają poprawnie.

Zostały zrealizowane za pomocą frameworka Testcontainers. Służy on do tworzenia instancji bazy danych, przeznaczonych do testów. Dzięki operacji na obrazach Dockera,

możliwe stało się zbudowanie tymczasowej bazy danych. Dzięki temu można manipulować danymi w tabelach SQL w trakcie testu.

```

@SpringBootTest
@Testcontainers
@DirtiesContext
@Profile("integration")
@WithMockUser
public abstract class IntegrationTest {
    39 usages
    @Container
    protected static final PostgreSQLContainer<?> container = new PostgreSQLContainer<>("postgres:14.6");

    Michael Lechowicz
    @DynamicPropertySource
    static void registerDatabaseProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", container::getJdbcUrl);
        registry.add("spring.datasource.username", container::getUsername);
        registry.add("spring.datasource.password", container::getPassword);
    }
}
```

Rys. 71. Klasa konfiguracyjna Testcontainers.

```

@Test
void try_delete_select_subcategory_in_database_for_wrong_user_email() {
    // given
    Long subcategoryIdToRemove = 8L;
    Long categoryId = 3L;
    CategoryService categoryService =
        new CategoryService(this.userRepository, this.categoryRepository, this.inputValidator);
    String email = "no_exists";
    //when
    ResourceNotFoundException thrown = Assertions.assertThrows(ResourceNotFoundException.class, () ->
        categoryService.deleteSubcategory(email, categoryId, subcategoryIdToRemove));
    // then
    assertEquals(String.format("Couldn't find a user with email: %s", email), thrown.getMessage());
}
}
```

Rys. 72. Test integracyjny w CategoryService.

```

class CategoryRepositoryTestIT extends IntegrationTest {

    @Autowired
    private CategoryRepository categoryRepository;

    // Michał Lechowicz
    @Test
    void should_get_all_category_without_subcategories_for_user_id() {
        // given
        Long userId = 1L;
        // when
        Set<Category> allCategories = this.categoryRepository.findAllByOwnerId(userId);
        //then
        assertThat(allCategories)
            .hasSize( expected: 5);

        var unexpectedSubcategory = getUnexpectedSubcategory(allCategories);
        assertThat(unexpectedSubcategory).isEmpty();

        var possibleCategoryForOtherUser = getUnexpectedCategoryFromOtherUser(allCategories, userId);
        assertThat(possibleCategoryForOtherUser).isEmpty();
    }

    1 usage  Michał Lechowicz
    private Optional<Category> getUnexpectedSubcategory(Set<Category> categories) {
        return categories.stream()
            .filter(c -> c.getParentCategory() != null)
            .findAny();
    }

    1 usage  Michał Lechowicz
    private Optional<Category> getUnexpectedCategoryFromOtherUser(Set<Category> categories, Long userId) {
        return categories.stream()
            .filter(c -> !Objects.equals(c.getOwner().getId(), userId))
            .findAny();
    }
}

```

Rys. 73. Testy integracyjne CategoryRepository.

6. Wdrożenie aplikacji

Pierwotnie cała aplikacja była wdrożona poprzez serwis Heroku. Reprezentuje on model platform as a service (PaaS). Dzięki temu to sam usługodawca, zarządzał przydziałem zasobów, co pozwalało znacznie zaoszczędzić czas. Niestety, w między czasie Heroku przeszło na model płatny, co znacznie zwiększyłoby koszty utrzymania [18]. Z tego powodu próbowano różnych platform jak Railway czy bit.io. Niestety, te serwisy również zakończyły swoje bezpłatne plany [19], a bit.io wręcz zakończył działalność [20].

Z tych powodów zdecydowano się wykupić dostęp do VPS [21] Mikrus [22]. Jest to nisko płatny (kilkadziesiąt złotych na rok) serwer, który można wykorzystać, np. do hostowania swojej strony internetowej lub bazy danych. W miarę potrzeb można zwiększyć jego zasoby, za niewielką opłatą. W dalszej części opisano już aktualnie używane rozwiązania.

6.1. Continuous integration (CI) i Continuous Delivery (CD)

Do wdrożenia aplikacji wykorzystywane jest GitHub Actions [23]. Jest to narzędzie CI/CD [24], dostarczane przez platformę GitHub. CI/CD to zestaw praktyk w rozwoju oprogramowania, które obejmuje Continuous Integration (CI) - regularne łączenie kodu z automatycznymi testami oraz Continuous Delivery (CD) - automatyczne dostarczanie zmian w kodzie do środowiska produkcyjnego po pomyślnym przejściu testów jakości. Dodatkowym krokiem jest Continuous Deployment (CD) - automatyczne wdrażanie zweryfikowanego kodu do produkcji. To pozwala na szybkie wykrywanie błędów, częste dostarczanie aktualizacji i zwiększenie efektywności procesu deweloperskiego.

W ramach procesu wykonywane są okresowe kopie zapasowe bazy danych, przeprowadzane są testy jednostkowe, integracyjne oraz wykonywane jest wdrożenie na serwer testowy (dla gałęzi innych niż dev) lub produkcyjny (dla gałęzi dev). Konfiguracja pipeline, czyli serii zautomatyzowanych procesów dostarczania oprogramowania, jest przeprowadzana w plikach `.yaml`, zapisywanych w repozytorium. Przechowywane one są w specjalnym katalogu `/.github/workflows/pipeline_name.yaml`. Dodatkowo wykorzystano rozszerzenia Github Actions do kopiowania pliku `.jar` na serwer Mikrus oraz restartu zaktualizowanej wersji oprogramowania.

6.2. Hosting bazy danych

Baza danych jest przechowywana na serwerze Mikrus. Na tym serwerze istnieje też współdzielona baza danych. Niestety, ruch między tą bazą danych a serwerem nie jest w pełni szyfrowany, co może grozić wyciekiem danych. Z tego powodu wymagało to doinstalowania PostgreSQL na serwerze, ustawienia portów oraz ich otwarcia na ruch.

```
name: FTM-DB-Backup

on:
  pull_request:
    branches:
      - main

jobs:
  backup:
    runs-on: ubuntu-latest
    env:
      PGPASSWORD: ${ secrets.PG_PASSWORD }}

    steps:
      - name: Checkout changes
        uses: actions/checkout@v3

      - name: Create backup
        id: create-backup
        run: |
          DATE=$(date +"%Y-%m-%d_%H:%M:%S")
          pg_dump ${ secrets.PG_DB_URI }} > ftm_${DATE}.sql
          echo "FILENAME=ftm_${DATE}.sql" >> "$GITHUB_OUTPUT"

      - name: Sent backup to mikr.us
        uses: garygrossgarten/github-action-scp@release
        with:
          host: ${ secrets.HOST }}
          username: ${ secrets.USERNAME }}
          privateKey: ${ secrets.KEY_ED25519 }}
          port: ${ secrets.PORT }}
          local: ${ steps.create-backup.outputs.FILENAME }}
          remote: "${ secrets.PG_SERVER_PATH }}/${ steps.create-backup.outputs.FILENAME }}"

      - name: Remove old backups
        uses: appleboy/ssh-action@master
        with:
          host: ${ secrets.HOST }}
          username: ${ secrets.USERNAME }}
          key: ${ secrets.KEY_ED25519 }}
          port: ${ secrets.PORT }}
          script: (cd "${ secrets.PG_SERVER_PATH }}" && sh ${ secrets.PG_REMOVE_OLD_BACKUPS_SCRIPT }}
```

Rys. 74. Pipeline backupu bazy danych.

6.3. Wdrożenie aplikacji serwerowej

Aplikacja serwerowa pracuje jako usługa w ramach serwera Mikrus. Na serwerze zainstalowana jest wersja Ubuntu 20.04.5.

```

1  name: FTM-server CI/CD
2
3  on:
4    push:
5    pull_request:
6      branches:
7        - dev
8
9  jobs:
10   build:
11     runs-on: ubuntu-latest
12
13     steps:
14       - uses: actions/checkout@v2
15       - name: Set up JDK 17
16         uses: actions/setup-java@v2
17         with:
18           java-version: '17'
19           distribution: 'adopt'
20
21       - name: Build with Maven
22         run: mvn --batch-mode --update-snapshots verify
23
24       - name: Deploy jar to mikr.us dev
25         if: github.event_name == 'pull_request'
26         uses: garygrossgarten/github-action-scp@release
27         with:
28           host: ${ secrets.HOST }
29           username: ${ secrets.USERNAME }
30           privateKey: ${ secrets.KEY_ED25519 }
31           port: ${ secrets.PORT }
32           local: target/follow-the-money-server-0.0.1-SNAPSHOT.jar
33           remote: ftmServerJar/ftm-server.jar
34
35       - name: Run server on mikr.us dev
36         if: github.event_name == 'pull_request'
37         uses: appleboy/ssh-action@master
38         with:
39           host: ${ secrets.HOST }
40           username: ${ secrets.USERNAME }
41           key: ${ secrets.KEY_ED25519 }
42           port: ${ secrets.PORT }
43           script: sh run-ftm-server.sh

```

Rys. 75. Pipeline dla serwera część 1.

```

45 - name: Deploy jar to mikr.us prod
46   if: github.ref == 'refs/heads/dev'
47   uses: garygrossgarten/github-action-scp@release
48   with:
49     host: ${ secrets.HOST_j155 }
50     username: ${ secrets.USERNAME_j155 }
51     privateKey: ${ secrets.KEY_ED25519_j155 }
52     port: ${ secrets.PORT_j155 }
53     local: target/follow-the-money-server-0.0.1-SNAPSHOT.jar
54     remote: ftmServerJar/ftm-server.jar
55
56 - name: Run server on mikr.us prod
57   if: github.ref == 'refs/heads/dev'
58   uses: appleboy/ssh-action@master
59   with:
60     host: ${ secrets.HOST_j155 }
61     username: ${ secrets.USERNAME_j155 }
62     key: ${ secrets.KEY_ED25519_j155 }
63     port: ${ secrets.PORT_j155 }
64     script: sh run-ftm-server.sh
65

```

Rys. 76. Pipeline dla serwera część 2.

W ramach pipeline wykonywana jest budowa nowej paczki jar z serwerem (krok *Build with Maven*). Następnie, jeśli pipeline został uruchomiony wraz z pull requestem, to spakowana paczka uruchamiana jest na serwerze testowym. Jeśli pipeline uruchomiony został z powodu zaktualizowania gałęzi *dev*, to wtedy paczka uruchamiana jest na serwerze produkcyjnym.

6.4. Wdrożenie aplikacji klienckiej

Aplikacja kliencka wdrażana jest w GitHub Pages dla wersji produkcyjnej oraz na platformie Vercel dla wersji rozwojowej.

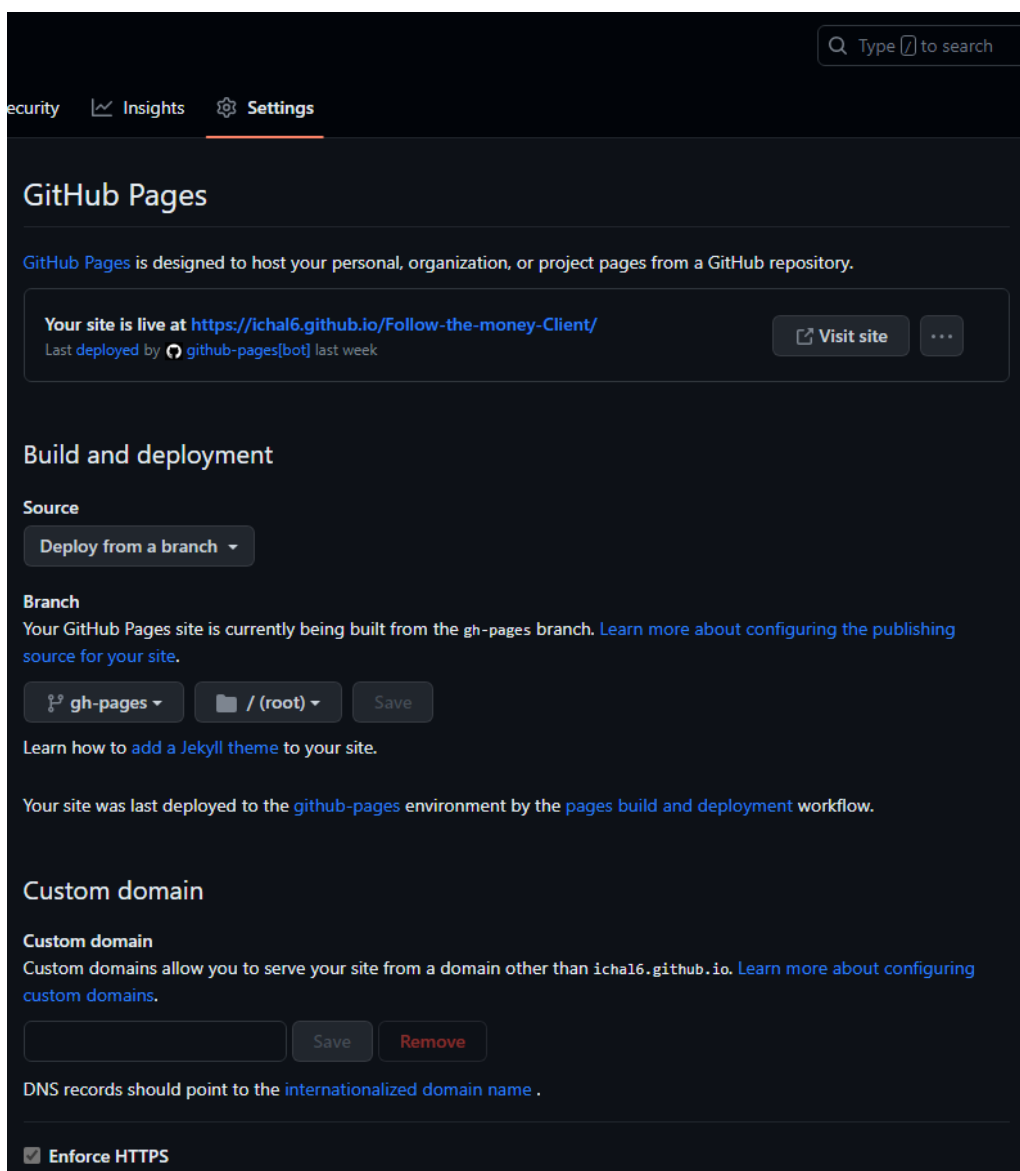
Dla platformy Vercel wystarczy założyć projekt, połączyć konto GitHub z platformą Vercel oraz wybrać właściwą konfigurację serwera backendowego.

Dla Github Pages konfiguracja była bardziej skomplikowana. Ze względu na charakterystykę SPA [2], aplikacji Angularowej wymagane było przekierowanie całego ruchu

na plik *index*. Wykonano to poprzez zastąpienie strony 404 (niewłaściwy adres), stroną *index.html*. Dzięki temu aplikacja zaczęła prawidłowo działać (krok *Create 404.html*). Następnie strona jest kopiowana do gałęzi *gh-pages*, która przechowuje już skompilowaną gotową stronę.

```
46     - name: Build
47       if: github.event_name != 'pull_request'
48       run: |
49         npm run build:prod
50
51     - name: Build dev
52       if: github.event_name == 'pull_request'
53       run: |
54         npm run build:dev
55
56     - name: Lint
57       run: |
58         npm run lint
59
60     - name: Test
61       run: |
62         npm run test:ci
63
64     - name: Create 404.html
65       if: github.ref == 'refs/heads/dev'
66       run: cp dist/Follow-the-money-Client/index.html dist/Follow-the-money-Client/404.html
67
68     - name: Deploy to GitHub Pages
69       if: github.ref == 'refs/heads/dev'
70       uses: crazy-max/ghaction-github-pages@v3
71       with:
72         target_branch: gh-pages
73         build_dir: dist/Follow-the-money-Client/
74       env:
75         GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
76
```

Rys. 77. Fragment pipeline dla klienta.



Rys. 78. Konfiguracja GitHub Pages.

Niniejszy projekt zapewnia łatwą i intuicyjną obsługę osobistych finansów. Umożliwia wygodną realizację celów. Ponadto oferuje przejrzyste diagramy oraz tabele finansowe, które umożliwiają bezpieczne zarządzanie pieniędzmi. Niezaprzeczalną zaletą aplikacji jest także bezproblemowy dostęp poprzez sieć oraz nieskomplikowana i intuicyjna obsługa.

7. Podsumowanie

Celem niniejszej pracy było zbudowanie aplikacji do śledzenia finansów Użytkownika, która pozwala na analizę oraz kontrolę wydatków. Przyczynia się do zwiększonej świadomości finansowej wśród Użytkowników.

Wszystkie założenia dotyczące implementacji zostały zrealizowane, a przypadki użycia wraz z przypisanymi im scenariuszami znajdują odzwierciedlenie w finalnej aplikacji. Przeszła ona testy pod kątem responsywności na różnych rozdzielczościach oraz z różnymi przeglądarkami.

7.1. Osiągnięte cele i funkcjonalności

Zrealizowano podstawowe założone funkcjonalności. Użytkownik może dokonać rejestracji, i następnie logowania. Po poprawnym zalogowaniu ma możliwość dodawania oraz edycji swoich Kategorii, Płatników oraz Kont. Ma możliwość dodawania Transakcji oraz Transferów. Dzięki modułowi Analizy finansowej oraz Tablicy finansowej może analizować swoje wydatki.

W aplikacji wykorzystano rozwiązania umożliwiające bezpieczne utrzymanie i rozwój kodu, takie jak testy jednostkowe oraz integracyjne, system kontroli wersji, wersjonowanie bazy danych czy odizolowane środowiska testowe.

7.2. Możliwości dalszego rozwoju i wnioski z projektu

Przyszła rozbudowa aplikacji może skupić się na: dodaniu opcji przypomnienia hasła, rozbudowy Tablicy finansowej lub wyświetlaniu powiadomień w aplikacji.

Efekt końcowy uznano za satysfakcjonujący, a zdobyte w trakcie tworzenia oprogramowania doświadczenie jako bardzo cenne. Dzięki wysiłkowi włożonemu w pisanie aplikacji zdobyto wiedzę związaną z rozwojem oprogramowania, wykorzystywaniem nowoczesnych technologii oraz wdrożeniem aplikacji.

8. Słownik pojęć

Poszczególne wpisy w słowniku zostały napisane w języku polskim, a ich tłumaczenie na angielski w nawiasach. Jeśli dany wpis występuje w pracy został on zapisany dużą literą.

Użytkownik (User) – osoba korzystająca z aplikacji do zarządzania finansami *Follow the money*;

Profil (Profile) - część serwisu zawierająca informacje o Użytkowniku;

Konto (Account) – miejsce, gdzie przechowywane są pieniądze Użytkownika;

Konto gotówkowe (Cash) – konto typu gotówkowego (np. portfel, sejf);

Konto bankowe (Bank) - konto typu elektronicznego (np. konto w banku);

Konto kredytowe (Loan) – konto typu kredytowego (np. kredyt w banku, odroczone płaćność, pożyczka);

Kategoria (Category) – rodzaj wydatku bądź wpływu. Służy do przypisywania do odpowiedniej grupy;

Podkategoria (Subcategory) – uszczegóławia rodzaj wydatku. Jedna Kategoria może mieć wiele Podkategorii;

Płatnik (Payee) – osoba lub firma, do której Użytkownik przelewa pieniądze;

Płaćność (Payment) – pojedynczy przesył środków, jakiego dokonuje Użytkownik. Posiada zdefiniowany tytuł, datę oraz kwotę.

Transakcja (Transaction) – podtyp dla Płaćności. Wydatek bądź przychód. Zawiera Konto, Płatnika oraz Kategorię.

Transfer (Transfer) – podtyp dla Płaćności. Służy do przesyłu środków pomiędzy Kontami Użytkownika. Zawiera Konto źródłowe, Konto docelowe.

Tablica finansowa (Dashboard) – strona startowa z podsumowaniem Profilu użytkownika. Zawiera sumaryczne informacje o wydatkach i wpływach, sumę pieniędzy na Kontach, ostatnie Transakcje itp.

9. Bibliografia

Dostęp do źródeł internetowych, sprawdzony dnia 19.08.2023 r.

- [1] Zdzisław Dybikowski. „PostgreSQL. Wydanie II”. Wydawnictwo Helion, 2012.
- [2] Adam Freeman. „Angular. Profesjonalne techniki programowania. Wydanie IV. Wydawnictwo Helion, 2021.
- [3] Christian Bauer, Gavin King, Gary Gregory. „Java Persistence. Programowanie aplikacji bazodanowych w Hibernate. Wydanie II”. Wydawnictwo Helion, 2016.
- [4] Craig Walls. „Spring w akcji. Wydanie V”. Wydawnictwo Helion, 2019.
- [5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. „Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku. Wydanie drugie”. WNT, 2008.
- [6] Artykuł dotyczący hasła *Follow the money* w serwisie Wikipedia:
https://en.wikipedia.org/wiki/Follow_the_money
- [7] Ranking popularności systemów operacyjnych w Polsce, wykonany przez Gemius:
<https://ranking.gemius.com/pl/ranking/systems/>
- [8] StormIT. „Encja”. Artykuł internetowy: <https://stormit.pl/encyclopedia/encja/>
- [9] Przemysław Bykowski. „Maven i Gradle – porównanie narzędzi i ich wydajności”
Artykuł internetowy: <https://bykowski.pl/maven-i-gradle-porownanie-narzedzi-i-ich-wydajnosci/>
- [10] Bartłomiej Żyliński. „Database Migration tools: Flyway vs Liquibase”. Artykuł internetowy: <https://dzone.com/articles/flyway-vs-liquibase>
- [11] Paweł Ćwik. “Obiekt domenowy, DTO, DAO”. Artykuł internetowy:
<https://clockworkjava.pl/2020/12/obiekt-domenowy-dto-dao/>
- [12] Oracle. „Co to jest Docker?”. Artykuł internetowy:
<https://www.oracle.com/pl/cloud/cloud-native/container-registry/what-is-docker/>
- [13] Maciej Aniserowicz. „Mega piguła wiedzy o testach jednostkowych”. Artykuł internetowy: <https://devstyle.pl/2020/06/25/mega-pigula-wiedzy-o-testach-jednostkowych/>

- [14] Marcin Pietraszek „Testy jednostkowe z JUnit 5”. Artykuł internetowy: <https://www.samouczekprogramisty.pl/testy-jednostkowe-z-junit5/>
- [15] Piotr Pelczar. „Mockito w pigułce! Poznaj dobre praktyki i przeczytaj czego nie mockować”. Artykuł internetowy: <https://softwareskill.pl/mockito-w-pigulce>
- [16] Asim Hussain. „Jasmine & Karma”. Artykuł internetowy: <https://codecraft.tv/courses/angular/unit-testing/jasmine-and-karma/>
- [17] Klaudia Szczepara. „What is Behavior Driven Development (BDD) and how to implement it?”. Artykuł internetowy: <https://www.euvic.com/behavior-driven-development-implementation-benefits/>
- [18] Jerry Ng. „Say Goodbye to Heroku Free Tier — Here Are 4 Alternatives”. Artykuł internetowy: <https://betterprogramming.pub/say-goodbye-to-heroku-free-tier-here-are-the-4-alternatives-8d82bb10b495>
- [19] Jake Cooper. „Introducing the \$5 Plan”. Artykuł internetowy: <https://blog.railway.app/p/introducing-trial-hobby-pro-plans>
- [20] Adam Flechter. „What’s Next For bit.io”. Artykuł internetowy: <https://blog.bit.io/whats-next-for-bit-io-joining-databricks-ace9a40bce0d>
- [21] Rafał Stępniewski. „Co to jest serwer VPS?”. Artykuł internetowy: <https://www.politykabezpieczenstwa.pl/pl/a/co-to-jest-serwer-vps>
- [22] Dominik Szczepaniak. „Mikrus – czyli co nieco o zapleczu bloga”. Artykuł internetowy: <https://devszczepaniak.pl/mikrus-czyli-co-nieco-o-zapleczu-bloga/>
- [23] Karol Ciemborowicz. „GitHub Actions – co to jest i dlaczego powinieneś z tego korzystać?”. Artykuł internetowy: <https://geek.justjoin.it/github-actions-co-to-jest-i-dlaczego-powinienes-z-tego-korzystac/>
- [24] Marcin Mroczkowski. „CI/CD jako proces niezbędny w dzisiejszym IT”. Artykuł internetowy: <https://javamaster.it/ci-cd-jako-proces-niezbledny-w-dzisiejszym-it/>
- [25] Chad Michel. „Introduction to PrimeNG”. Artykuł internetowy: <https://dontpaniclabs.com/blog/post/2021/06/15/introduction-to-primeng/>

[26] Angelika Siczek. „Co to jest i jak działa JSON Web Tokens (JWT)?”. Artykuł internetowy: <https://global4net.com/blog/ecommerce/co-to-jest-i-jak-dziala-json-web-tokens-jwt/>

[27] Bartosz Dąbek. „Jak działa mechanizm sesji?”. Artykuł internetowy: <https://www.bdabek.pl/jak-dziala-mechanizm-sesji/>

10. Spis Rysunków

Rys. 1. Widok przypadków użycia dla Użytkownika. Opracowanie własne.	10
Rys. 2. Diagram encji. Opracowanie własne.	11
Rys. 3. Widok okna rejestracji dla komputera stacjonarnego. Opracowanie własne.....	14
Rys. 4. Widok okna logowania dla urządzeń mobilnych. Opracowanie własne.	15
Rys. 5. Widok Tablicy finansowej dla komputerów biurowych. Opracowanie własne.....	15
Rys. 6. Widok Kont dla tabletów. Opracowanie własne.	16
Rys. 7. Widok Kategorii dla komputerów stacjonarnych. Opracowanie własne.	16
Rys. 8. Widok płatników dla urządzeń mobilnych. Opracowanie własne.	17
Rys. 9. Widok Płatności dla tabletów. Opracowanie własne.....	18
Rys. 10. Widok Analizy finansowej dla urządzeń mobilnych. Opracowanie własne.	19
Rys. 11. Widok Analizy finansowej dla komputerów osobistych. Opracowanie własne.	19
Rys. 12. Diagram interakcji dla generowania Tablicy finansowej. Opracowanie własne.	20
Rys. 13. Diagram interakcji dla tworzenia nowego Konta. Opracowanie własne.	21
Rys. 14. Struktura katalogów dla serwera.	24
Rys. 15. Plik konfiguracyjny Mavena – pom.xml.....	26
Rys. 16. Pliki z wersjami bazy danych.	27
Rys. 17. Plik changelog z komendą usuwającą kolumnę type z tabeli Category.	27
Rys. 18. Pakiet entity.....	28
Rys. 19. Entity Category.....	29
Rys. 20. Pakiet repository.	30
Rys. 21. Interfejs AccountRepository.....	30
Rys. 22. CustomTransactionRepository.	31
Rys. 23. Implementacja interfejsu CustomTransactionRepository.....	31

Rys. 24. Interfejs TransactionRepository.	32
Rys. 25. Pakiet modelDTO.	33
Rys. 26. AnalysisFinancialTableDTO.....	33
Rys. 27. Serwis Payee.....	34
Rys. 28. Kontroler Tablicy Finansowej.....	35
Rys. 29. Katalog environments.	37
Rys. 30. Pliki environment.*.ts.	37
Rys. 31. Angular.json - podmiana plików konfiguracyjnych.	38
Rys. 32. Przykład wykorzystania environment w PayeeService.	39
Rys. 33. Package.json - wywołanie odpowiedniej konfiguracji.	39
Rys. 34. Dodanie routingu do pliku portal.module.ts.....	40
Rys. 35. Struktura modułu do analizy danych.	40
Rys. 36. Metoda getAnalysisDataRows w AnalysisService.....	41
Rys. 37. Metoda updateData z FinancialSummaryComponent.	41
Rys. 38. Model AnalysisTableRow.....	42
Rys. 39. Builder dla AnalysisTableRow.....	42
Rys. 40. Metoda updateChart.....	43
Rys. 41. Funkcje pomocnicze dla updateChart.....	44
Rys. 42. Definicja diagramu dla wpływów lub wydatków.....	45
Rys. 43. Definicja komponentu dla diagramu wpływów/wydatków.	45
Rys. 44. Plik Dockerfile dla serwera.	46
Rys. 45. Plik docker-compose.yaml dla serwera.	47
Rys. 46. Plik Dockerfile dla klienta.....	48
Rys. 47. Plik docker-compose.yml dla klienta.....	48

Rys. 48. Dodanie zależności Spring Security.	49
Rys. 49. Metoda securityFilterChain.	50
Rys. 50. AccessValidator.	51
Rys. 51. Przykładowe wywołanie do serwera.	51
Rys. 52. Metoda authenticate z AuthService.	52
Rys. 53. Metoda canActivate z AuthRouteGuardService.	52
Rys. 54. Okno edycji transferu.	53
Rys. 55. Metoda onSubmit.	53
Rys. 56. Metoda updateTransfer z TransferService.	54
Rys. 57. Aktualizacja Transferu w Kontrolerze.	54
Rys. 58. Aktualizacja Transferu w Serwisie.	55
Rys. 59. TransferRepository.	55
Rys. 60. Zdefiniowanie obiektu typu mock.	56
Rys. 61. Zdefiniowanie zachowania obiektu.	57
Rys. 62. Klasa AnalysisFinancialTableDTOFixture.	57
Rys. 63. Testy serwisu AnalysisService.	58
Rys. 64. Test służący do wygenerowania zbioru AnlysisFinancialTableDTO.	59
Rys. 65. Struktura katalogów testowych.	60
Rys. 66. Inicjalizacja komponentu.	61
Rys. 67. Testy komponentu FinancialSummaryComponent.	62
Rys. 68. Test AnalysisBuilder.	63
Rys. 69. Inicjalizacja serwisu w celu testów.	64
Rys. 70. Testy serwisu.	65
Rys. 71. Klasa konfiguracyjna Testcontainers.	66

Rys. 72. Test integracyjny w CategoryService.	66
Rys. 73. Testy integracyjne CategoryRepository.	67
Rys. 74. Pipeline backupu bazy danych.	69
Rys. 75. Pipeline dla serwera część 1.	70
Rys. 76. Pipeline dla serwera część 2.	71
Rys. 77. Fragment pipeline dla klienta.	72
Rys. 78. Konfiguracja GitHub Pages.	73