

# **Wyższa Szkoła Technologii Informatycznych w Katowicach**

---

Wydział Informatyki  
Kierunek Informatyka

**Marcin Krawczyk**

Nr albumu: 05451  
Studia niestacjonarne

## **Opracowanie i implementacja aplikacji internetowej do zarządzania budżetem osobistym**

Praca dyplomowa inżynierska  
napisana pod kierunkiem  
dr inż. Romana Simińskiego  
w roku akademickim 2016/2017

Katowice 2018

## **Spis treści**

---

1	Wstęp .....	3
2	Charakterystyka/analiza problemu .....	4
3	Analiza istniejących rozwiązań .....	6
4	Koncepcja własnego rozwiązania.....	10
5	Projekt ogólny .....	14
5.1	Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych .....	14
5.1.1	Wymagania funkcjonalne .....	14
5.1.2	Wymagania niefunkcjonalne .....	14
5.1.3	Diagram przypadków użycia.....	15
5.2	Architektura systemu.....	16
5.2.1	Ogólne .....	16
5.2.2	Warstwa kliencka .....	16
5.2.3	Warstwa serwerowa.....	18
5.2.4	Przechowywanie danych.....	19
6	Projekt techniczny .....	21
6.1	Część serwerowa .....	21
6.2	Część kliencka.....	24
7	Testy i weryfikacja systemu .....	30
8	Przykładowy scenariusz wykorzystania systemu .....	31
9	Zakończenie .....	41
10	Bibliografia.....	44
11	Spis rysunków .....	45

## 1 Wstęp

Celem pracy jest zaprojektowanie oraz realizacja systemu do zarządzania przychodami i wydatkami, budżetem osobistym, która pomoże nam odpowiednio dbać o przepływ naszej gotówki. Motywacją do stworzenia tego typu aplikacji była dla mnie chęć raz na zawsze zapanowania nad swoimi wydatkami i procesem zarabiania, wydawania i oszczędzania pieniędzy. Brak w pełni satysfakcjonującego mnie narzędzia dostępnego na rynku był ostatecznym czynnikiem dzięki któremu zdecydowałem, że chcę się pochylić nad tym problemem i zaprojektować aplikację w pełni wyczerpującą moje potrzeby. Jednocześnie to, że istnieje wiele pomniejszych aplikacji do zarządzania budżetem wskazuje, że na takie narzędzie jest zapotrzebowane. Problemem jest niepełność i słaby poziom merytoryczny dostępnych aplikacji, który jest szansą na powodzenia dla mojej implementacji tego problemu.

Główną platformą aplikacji będzie responsywna strona www dostosowująca się do każdego urządzenia. W aplikacji będą zaimplementowane funkcje dodawanie wydatków i przychodów, zestawiania zależności zachodzących pomiędzy nimi na interaktywnych wykresach, bazując między innymi na kategoriach wydatków i przychodów oraz innych zdefiniowanych właściwościach. Aplikacja będzie miała wydzielony moduł do stworzenia tzw. *Budżetu Osobistego*, opisującego nasze oczekiwania, plany jak wyobrażamy sobie dany miesiąc w sensie wydatków i przychodów. W *Budżecie Osobistym* będziemy deklarować nasze oczekiwania zarobkowe a także to ile na daną kategorie wydatków chcemy przeznaczyć pieniędzy.

Praca składa się z 9 rozdziałów. W niniejszym rozdziale znajduje się krótki wstęp oraz, opis zawartości pracy. W drugim rozdziale zostaje przedstawiona analiza tematu, motywacji do podjęcia się rozwiązania problemu finansów. W rozdziale trzecim znajduje się analiza dostępnych aplikacji na rynku polskim i światowym. W czwartym rozdziale przedstawiam koncepcję mojego rozwiązania problemu. W piątym rozdziale skupiam się na ogólnym podejściu do tematu, na wymaganiach funkcjonalnych i niefunkcjonalnych aplikacji,. W rozdziale szóstym przedstawiony jest cały projekt od strony technicznej. Rozdział siódmy traktuje zaś o testowaniu aplikacji. Rozdział ósmy to przedstawienie przykładowego zastosowania systemu. W rozdziale dziewiątym znajdują się zaś podsumowanie całego projektu.

## 2 Charakterystyka/analiza problemu

Kwestia pieniędzy zawsze jest kwestią dość drażliwą. Tak naprawdę większość z nas ma wystarczającą ilość pieniędzy na wykonywanie wielu różnych działań. Po prostu w wielu przypadkach nie potrafimy z tych pieniędzy w sensowny sposób korzystać. Nasze finansowe środki znikają bardzo szybko i często nie mamy pojęcia gdzie i dlaczego wyparowały. Co się wydarzyło, że nagle z naszej pensji w połowie miesiąca nie zostaje nic, a mamy wrażenie, że nic wielkiego nie kupowaliśmy.

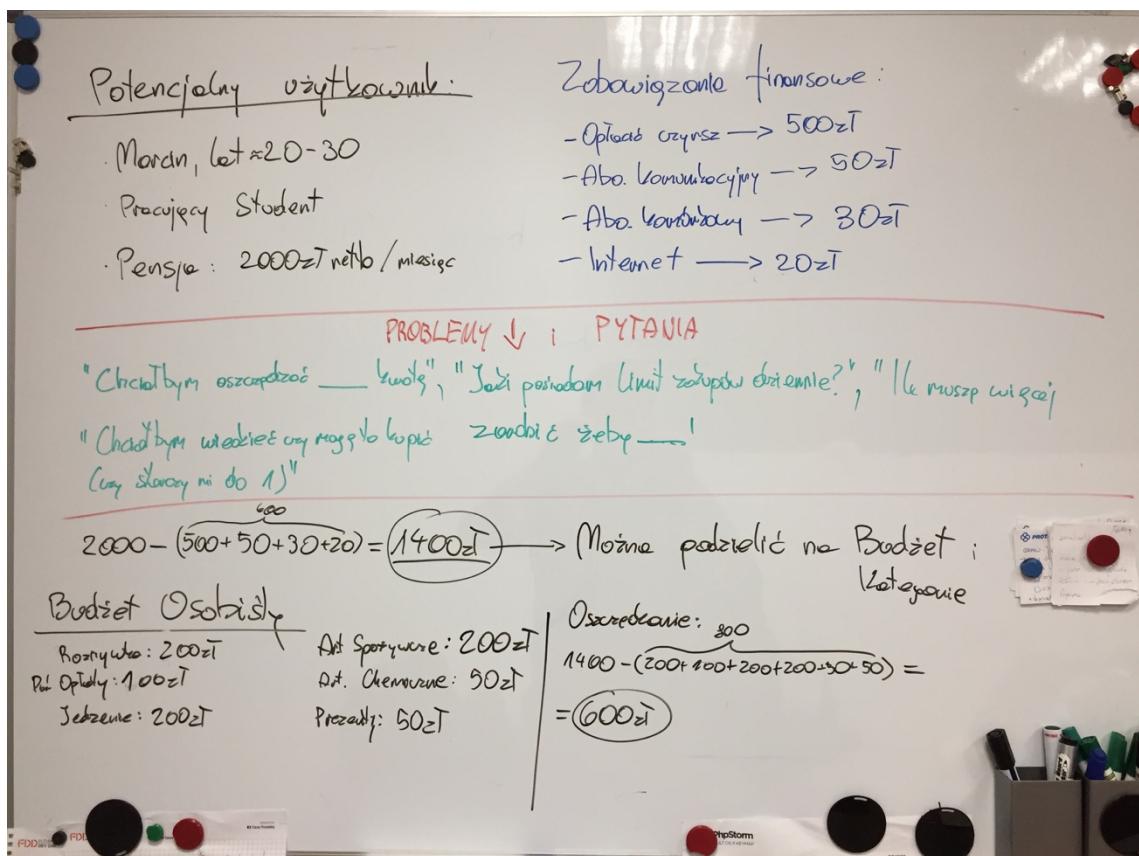
W zarządzaniu finansami różni ludzie oczekują różnych rezultatów, szukają innych profitów. Są jednostki, które potrzebują spisywać wydatki bo nie wiedzą gdzie „uciekają” ich pieniądze. Kiedy spisujemy takie rzeczy, na początku możemy nie widzieć ewidentnych plusów, jednak po miesiącu, dwóch jesteśmy w stanie wyśledzić na jakie niepotrzebne sprawy wydaliśmy pieniądze. Bardzo często okazuje się, że wydajemy duże pieniądze na bardzo nieistotne nabytki. Często widzimy także, że to małe zakupy, ale w powtarzającym się schemacie najbardziej wpływają na nasze braki finansowe.

Są także ludzie, którzy chcą by oprogramowanie automatyzowało decyzje zakupowe. Przykładowo, po szybkim spojrzeniu na telefon widzimy, czy zdefiniowany budżet na zakupy ubrań i obuwia nie został już przekroczony, czy może spokojnie możemy pozwolić sobie na jeszcze jedną parę butów w tym miesiącu.

Oszczędzanie na konkretne cele, bądź po prostu oszczędzanie, także jest celem mojej aplikacji. Kiedy mamy świadomość na co wydajemy pieniądze, kiedy ustalamy budżet i wiemy ile mamy do rozdysponowania funduszy na określone kategorie, możemy także tak przemyśleć nasz budżet, żeby zostawić trochę pieniędzy na oszczędności. Dzięki kontroli i przypomnieniom naszej aplikacji, będziemy wiedzieli dlaczego w tym miesiącu nie wydajemy pieniędzy na określone dobra.

Bardzo długo zastanawiałem się nad różnymi wariantami tego problemu. Nad tym jakie zagadnienia związane z problemem finansów mogę podjąć. Wyobraziłem sobie każdą osobę z osobna, która może używać aplikacji.

Studenta, który musi bardzo sprawnie zarządzać niewielką ilością gotówki. Biznesmana który tej gotówki ma więcej, ale także ma różnego rodzaju zobowiązania finansowe, które musi zaadresować. Poniżej rozpisałem przykładowo sylwetkę pracującego studenta i jego oczekiwania.



Rysunek 1 Zobrazowanie problemu budżetu osobistego

Dzięki mojej aplikacji, tenże student, będzie w stanie na bieżąco monitorować swoje wydatki, swoje przychody. Będzie mógł widzieć zależność pomiędzy tym ile zostało z ostatniej pensji, a tym na co wydaje posiadane pieniądze. Będzie mógł w łatwy sposób, zobaczyć na wykresach zależności pomiędzy kategoriami do których przypisuje wydatki. A także, po zdefiniowaniu budżetu, widzieć na bieżąco na ile jeszcze może sobie pozwolić w tym miesiącu w określonych kategoriach. Jego życie stanie się bardziej poukładane, i zyska potencjał do oszczędzania nawet posiadając relatywnie mało środków.

### 3 Analiza istniejących rozwiązań

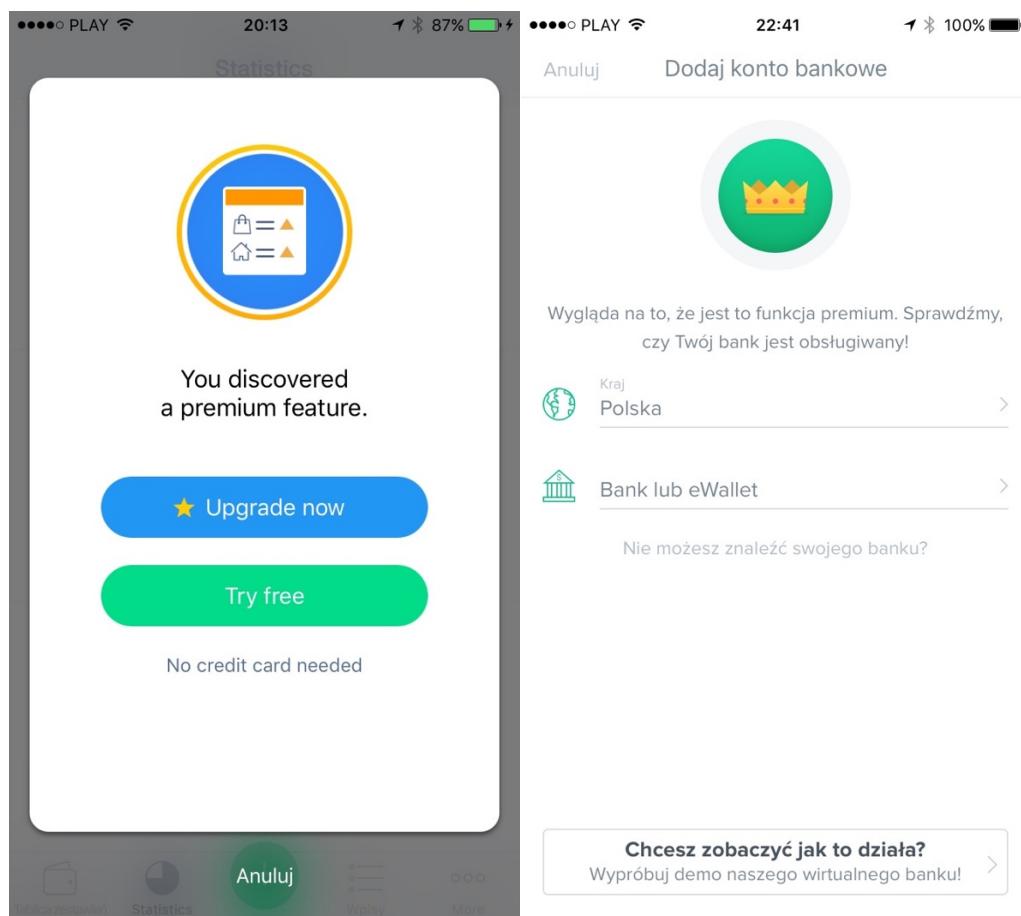
Istniejących rozwiązań jest sporo. Dla urządzeń typu desktop, na smartfony, tablety, urządzenia posiadające dostęp przez przeglądarkę. Jednak żadne z nich w pełni mnie nie usatysfakcjonowało. Każde z nich miało braki w funkcjach, które uważałem za potrzebne, a kiedy dana aplikacja posiadała funkcje A, to nie posiadała funkcji B i przeciwnie. Przykładowo, jedna aplikacja posiada funkcje tworzenia budżetu w różnym okresie czasu, ale nie posiada funkcji wyświetlania dobrze sformatowanych wykresów które są w stanie pokazać nam wiele zależności pomiędzy naszymi finansami. W innym przypadku, takie wykresy można było nawet samemu tworzyć, ale okazywało się, że budżet można ustawić tylko raz i to tylko na miesiąc, więc pewnej funkcjonalności brakowało. Bardzo często spotykałem się z aplikacjami, które rozwiązywały tylko część problemu związanego z zarządzaniem finansami osobistymi. Przykładowo dawały dostęp tylko do wpisywania wydatków, przychodów i nic poza tym.



Rysunek 2 Aplikacja Cents

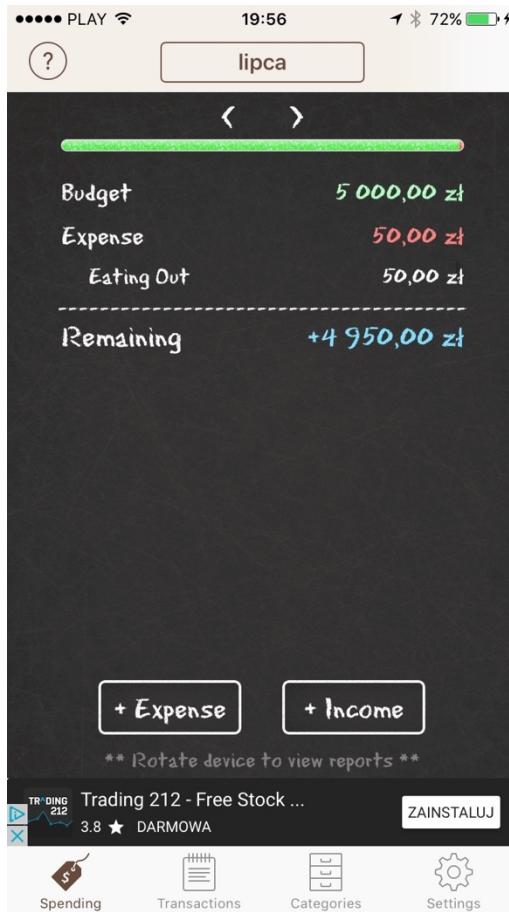
Aplikacja Cents służy tylko do wpisywania wydatków i przychodów, oprócz tego jest zablokowana do kilkunastu wpisanych wydatków, możliwość wpisania większej ilości jest dostępna w wersji płatnej PREMIUM

Wiele aplikacji które testowałem miało bardzo uproszczone funkcje w wersji darmowej. Dopiero po wykupieniu abonamentu PREMIUM, można było liczyć na funkcje synchronizowania danych o wydatkach i dochodach poprzez zalogowanie do swojego banku (która także była dostępna tylko w niektórych bankach).



Rysunek 3 Po lewej aplikacja Spendee w której dodanie konta bankowego jest dopiero dostępne w funkcji PREMIUM, w Polsce, jedynie 3 banki obsługiwane. Po prawej aplikacja Wallet, podobnie.

Duża ilość aplikacji, które analizowałem, nie może się też poszczycić zbyt profesjonalnym i zachęcającym wyglądem. Często zaprojektowanie grafik dla narzędzia do zarządzania finansami jest infantylne i dość dziecięce.



Rysunek 4 Aplikacja Cents nie wzbudza zaufania. (a aplikacja zarządzająca naszymi pieniędzmi powinna)

Podsumowaniem do tego rozdziału może być stwierdzenie, że po prostu na rynku aplikacji panuje pewnego rodzaju chaos. Jest bardzo dużo narzędzi do zarządzania własnymi finansami. Ale nie ma żadnej aplikacji, której bym na dłużej zaufał, która miała by wszystkie funkcje potrzebne. A nawet jeśli znajdzie się aplikacja która jest prawdziwym skarbem i jest kompletna to jej tzw. „User Experience” jest bardzo zły i aplikację rzucamy w kąt bo nie jesteśmy w stanie z niej szybko i przyjemnie korzystać. W dzisiejszym zabieganym, dynamicznym świecie często właśnie łatwość użytkowania i prostota przyciąga do nas klientów i daje nam wygrywającą pozycję i polecenia od zadowolonych użytkowników.

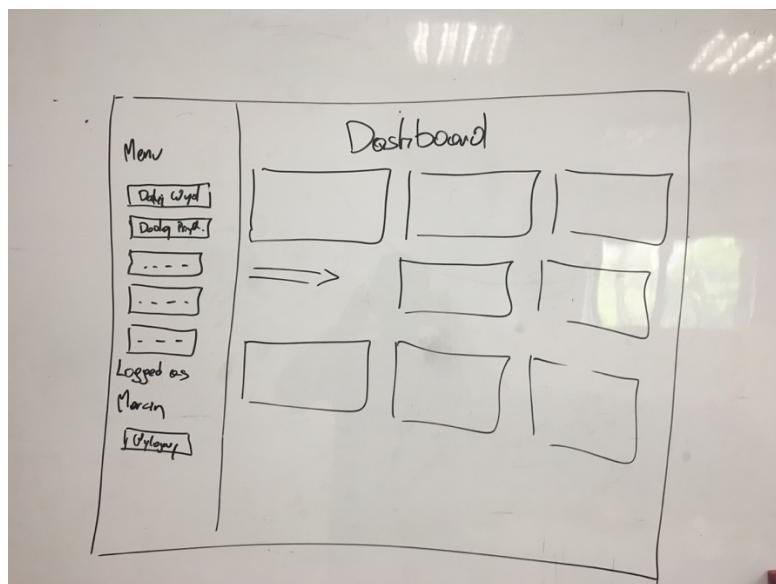
Moim głównym celem jest zaproponowanie użytkownikowi kompleksowego, aczkolwiek intuicyjnego systemu do zarządzania finansami. Dość dobrym przykładem przyjętej metodyki jest zdanie „**Easy to begin, harder to master**”. Łatwo rozpocząć, ciężej być mistrzem. Chciałbym poprowadzić użytkownika poprzez proste funkcje które dadzą mu od razu kontrolę nad swoim

bilansem finansowym powoli wprowadzając w bardziej zaawansowane narzędzia kontroli, oszczędzania, budżetowania.

## 4 Koncepcja własnego rozwiązania.

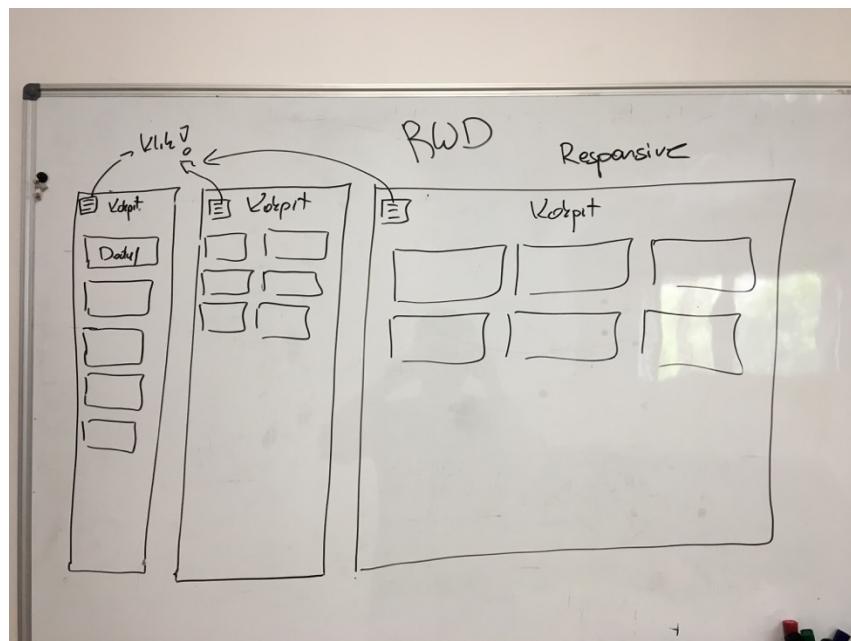
Widzimy już, że zarówno problem panowania na wydatkami i przychodami jest problemem, który wciąż jest nierozwiążany. Firmy próbują w różnoraki sposób podejść do tematu, jednak wciąż brakuje tu dobrego i prostego rozwiązania.

Moja propozycja to aplikacja **webowa**, która będzie wysoko dostępna i multiplatformowa dająca nam natychmiastowy dostęp do naszego budżetu. Postanowiłem wykorzystać potęge przeglądarek internetowych i najnowszych standardów by dotrzeć do jak największej ilości ludzi. Aplikacja **PersonalBudget** będzie bazować na interfejsie użytkownika stworzonym w technologiach takich jak **HTML, CSS, JavaScript**. To co było dla mnie istotne w trakcie projektowania i implementowania funkcji aplikacji była łatwa dostępność do poszczególnych elementów aplikacji. Wykorzystując gotowe komponenty graficzne mogłem w łatwy sposób dać użytkownikowi dostęp do rozsuwanego menu bocznego. W tymże użytkownik ma dostęp do wszystkich modułów aplikacji.



Rysunek 5 – wysuwane boczne menu

Oczywiście zależało mi także na tym, by aplikacja była używalna na każdym urządzeniu jakie może mieć dostęp do internetu i przeglądarki, więc zaprojektowałem ją tak, by zależnie od rozdzielczości urządzenia wciąż była używalna i pokazywała te informacje, które są kluczowe.

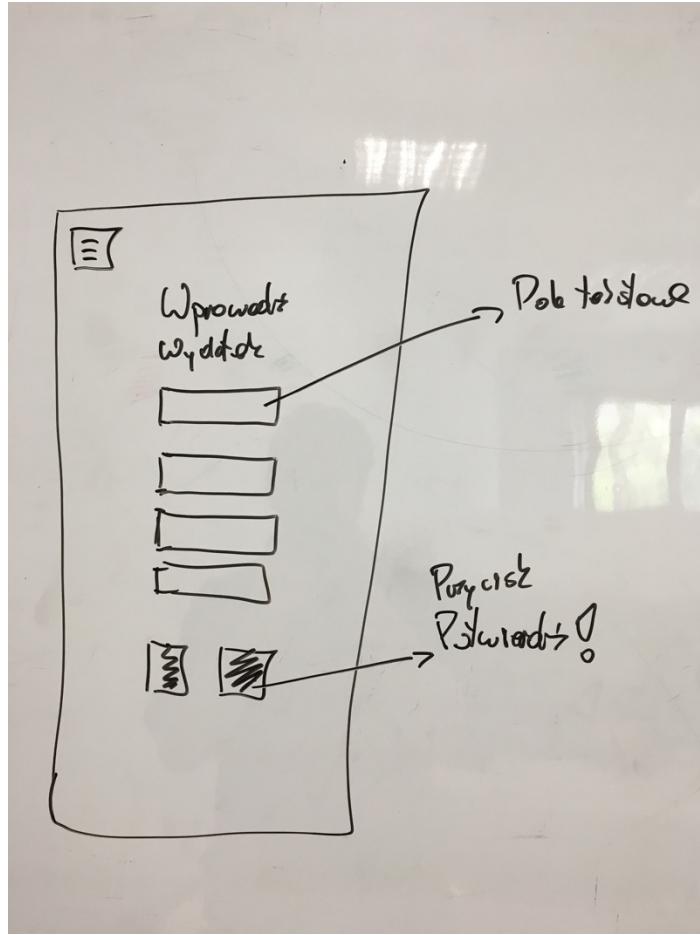


Rysunek 6 – zasady responsywności – dostępność na każdym urządzeniu

W moim rozwiązaniu skupiam się na 3 najważniejszych aspektach.

- 1. Zapisywanie**
- 2. Budżetowanie**
- 3. Analizowanie**

Pierwszy z nich, czyli **Zapisywanie** jest tak naprawdę najważniejszym z całej trójcy. Niestety, żeby móc stworzyć budżet a także analizować nasze wydatki i wynieść z tego jakąś korzyść, musimy najpierw stworzyć bazę informacji o nas, o naszych wydatkach, tak naprawdę o naszych zachowaniach. Poprzez szereg podstron umożliwiających dodawanie wydatków i przychodów jest realizowana ta funkcjonalność. W pracy inżynierskiej skupiam się jedynie na manualnym sposobie dodawania wydatków – w momencie wykonania zakupu, kierujemy się do aplikacji, wchodzimy na podstronę związaną z dodawaniem wydatku i wypełniamy wskazane pola.



Rysunek 7 – schemat dodawania wydatku/przychodu/budżetu

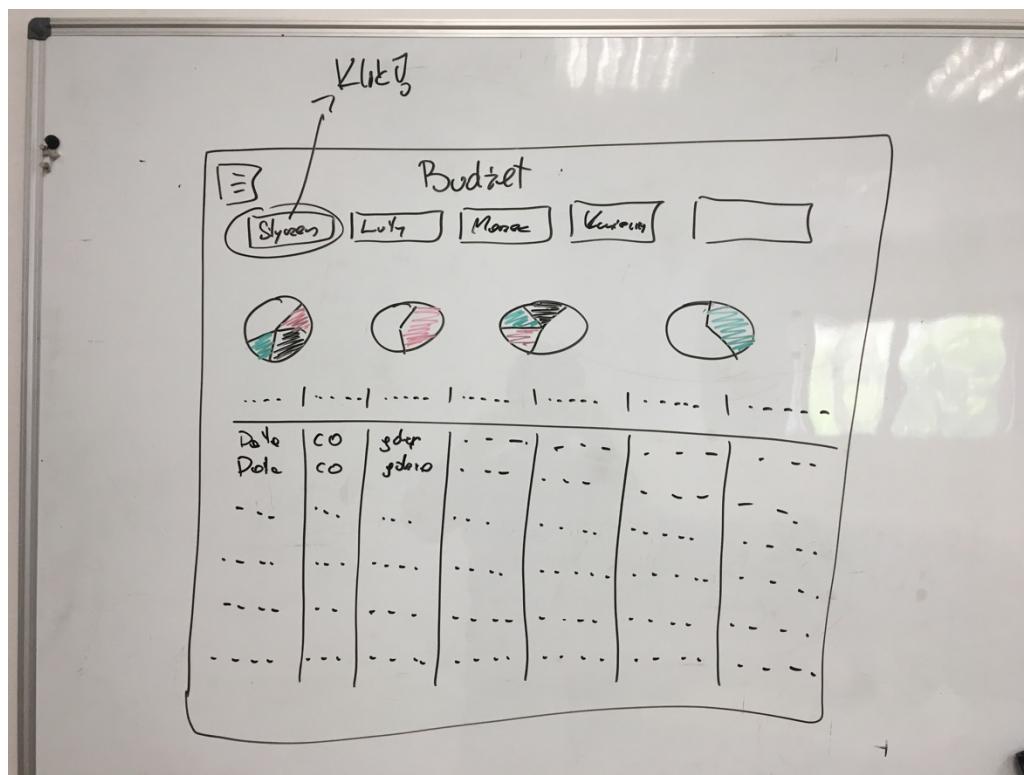
W tym miejscu, można by było pokusić się o implementacje rozwiązania pól automatycznego. Aktualne mechanizmy rozpoznawania pisma, rozpoznawania tekstu na obrazkach pozwoliły by na stworzenie modułu do aplikacji, który z sukcesem mógłby rozpoznać dużą ilość informacji z paragonu. Robiłoby się zdjęcie paragonu a aplikacja sama podjęłaby próbę dopasowania poszczególnych kawałków tekstu do pól tekstowych w podstronie z dodawaniem wydatku. Jednak niepełność rozwiązania i ograniczone zasoby czasowe nie pozwoliły mi dłużej pochyłać się nad takim rozwiązaniem.

Kolejnym z najważniejszych podpunktów jest **Budżetowanie**. W sytuacji kiedy wyrobiliśmy już sobie nawyk zapisywania wydatków i korzystamy z prostego formularza w aplikacji, możemy dodać do tego tworzenie budżetu. Postanowiłem ułatwić mechanizm budżetowania tylko do stworzenia budżetu – zdefiniowanie nazwy budżetu, zdefiniowanie daty początku okresu i końca okresu, i zdefiniowanie poszczególnych kategorii w budżecie na podstawie których będzie można zobaczyć na interaktywnych wykresach na jakie wydatki

jestemy w stanie sobie jeszcze pozwolić w tym okresie budżetowania. Po tym jak stworzymy budżet, możemy w trakcie dodawania jakiegokolwiek wydatku wybrać dodatkowo, do jakiej kategorii w jakim budżecie chcemy ten wydatek przypisać.

Co prowadzi nas do ostatniej z najważniejszych cech mojego oprogramowania czyli **Analiza**. Dzięki tak zapisywany danym o naszych dochodach i wydatkach mamy dostęp do aktualizowanych w czasie rzeczywistym wykresów którą pokazują nam zależności pomiędzy poszczególnymi rodzajami wydatków.

Na poniższym rysunku, widzimy jedno z miejsc w których wykorzystane będą te wykresy, czyli na podstronie gdzie zarządzamy naszymi budżetami. Możemy wybrać konkretny budżet i analizować dane z nim związane.



Rysunek 8 – podstrona budżetu osobistego z interaktywnymi wykresami

## 5 Projekt ogólny

### 5.1 Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych

#### 5.1.1 Wymagania funkcjonalne

1. Zarejestrowanie użytkownika.
2. Bezpieczne zalogowanie użytkownika.
3. Dodanie wydatku.
4. Edycja wydatku.
5. Wyświetlenie listy wydatków.
6. Usunięcie wydatku.
7. Dodanie przychodu.
8. Edycja przychodu.
9. Wyświetlenie listy przychodów.
10. Usunięcie przychodu.
11. Dodanie budżetu.
12. Dodanie kategorii do budżetu.
13. Zdefiniowanie kwot maksymalnych dla poszczególnych kategorii w budżetach.
14. Dodanie wydatku do budżetu.
15. Możliwość analizy wydatków na interaktywnych wykresach.
16. Obsługa błędów i stosowne informacje dla użytkownika w przypadku błędnie wypełnionych pól formularza dodawania przychodu, wydatku, budżetu.

#### 5.1.2 Wymagania niefunkcjonalne

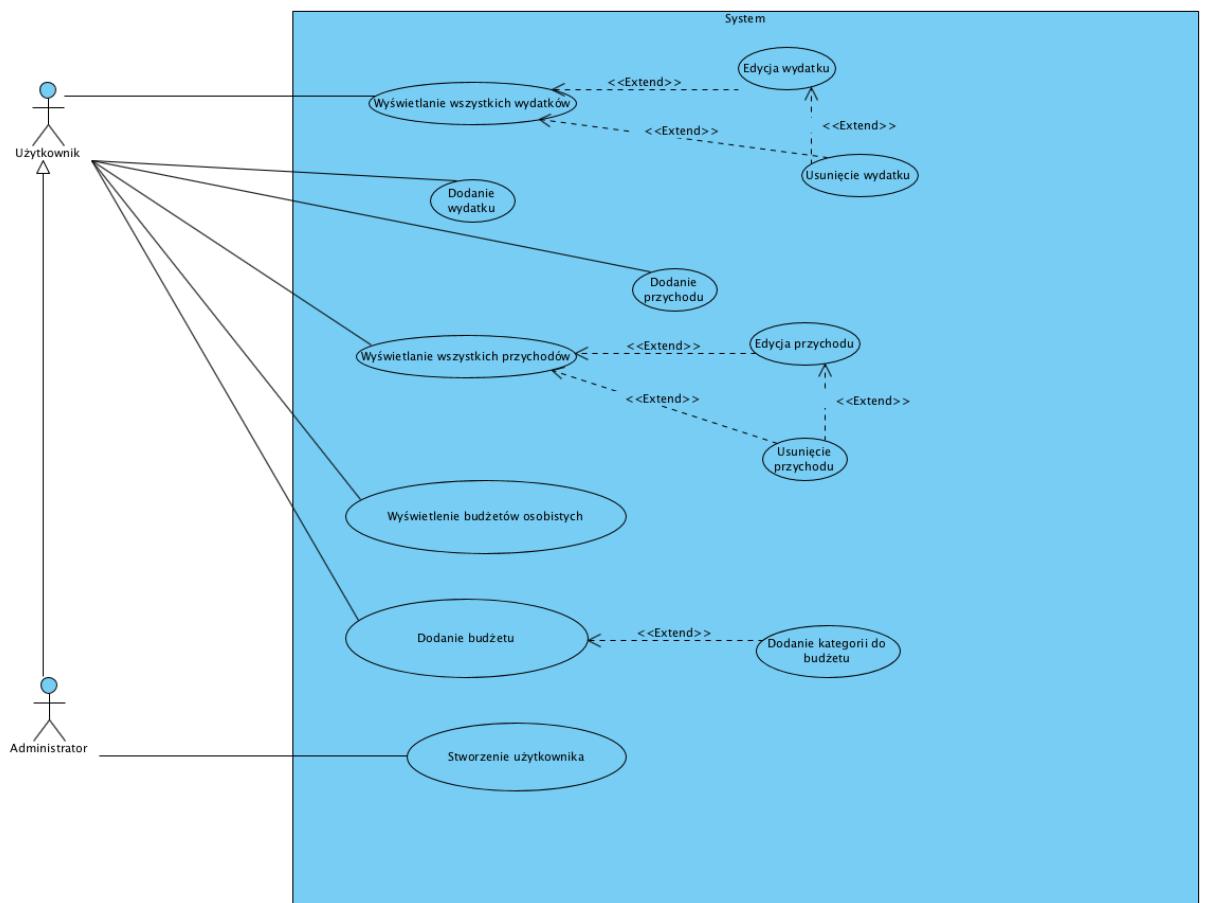
1. Bezpieczne logowanie
2. Uruchomienie i hostowanie aplikacji z wykorzystaniem technik wirtualizacji i kontenerów Docker<sup>1</sup>
3. Nierelacyjna baza danych
4. RWD – responsywny design, dopasowywający się do urządzenia które jest używane do wyświetlenia aplikacji
5. Prawidłowa dla urządzeń mobilnych wielkość przycisków i elementów interakcji

---

<sup>1</sup> <https://www.youtube.com/watch?v=cLT7eUWKZpg&t=48s>

6. Prawidłowa dla urządzeń mobilnych wydajność pod kątem wolniejszych łącz internetowych i optymalizacja pod kątem procesorów mobilnych.
7. Prawidłowa, łatwo rozszerzalna, podzielona na moduły aplikacja, zarówno w części klienckiej jak i serwerowej.
8. Aplikacja typu SPA – Single Page Application – umożliwia pracę bez przeładowania przeglądarki.
9. Aplikacja oparta o nowoczesny framework Angular.
10. Dowolny nowoczesny telefon i komputer z przeglądarką internetową będzie miał dostęp do aplikacji.

### 5.1.3 Diagram przypadków użycia



## 5.2 Architektura systemu

### 5.2.1 Ogólne

Aplikacja będzie aplikacją przeglądarkową podzieloną na warstwę kliencką – odpowiadającą za interfejs użytkownika, prawidłowe wyświetlanie go, i interakcję z warstwą serwerową poprzez zapytania RESTowe. Warstwę serwerową – odpowiadającą za prawidłowe obsługiwanie zapytań przychodzących z frontendu, odpowiednie połączenie z bazą danych MongoDB i odbieranie, wprowadzanie, edytowanie danych w niej zawartych.

Środowisko developerskie tak jak i środowisko produkcyjne, będzie w prosty sposób (dla użytkownika chcącego uruchomić je) stworzone z wykorzystaniem narzędzi wirtualizacji i kontenerów Docker. Dla każdego z „serwisu” (frontend, backend, baza danych) będzie stworzony oddzielnny kontener. Kontenery te będą ze sobą połączone specjalnymi mechanizmami Dockera (plik docker-compose.yml). Dzięki takiemu zaprojektowaniu, każdy nowy developer, czy też osoba która chciałaby obejrzeć oprogramowanie działające na lokalnej maszynie posiada prosty sposób na uruchomienie go. Wystarczy, że będzie miała zainstalowany **Docker Community Edition** i jedyne co musi zrobić, to w katalogu głównym aplikacji wykonać polecenie **<sup>2</sup>docker-compose build** a później **docker-compose up** by bezproblemowo uruchomić każdą z warstw połączonych ze sobą w prawidłowy sposób.

### 5.2.2 Warstwa kliencka

Do zrealizowania aplikacji do kontrolowania wydatków postanowiłem wykorzystać aktualne najnowsze technologie jakie można spotkać w świecie programistów webowych. Aplikacja istnieje w środowisku przeglądarkowym więc oczywistym wyborem po stronie klienckiej będzie język **JavaScript**. Do tworzenia kolejnych widoków a także strony wizualnej wykorzystałem duet **HTML + CSS**. Oczywiście, w obu przypadkach koncentruję się na korzystaniu z najnowszych ich wersji czyli HTML w wersji 5 i CSS w wersji 3.

Dodatkowym zwiększeniem możliwości języka CSS jest rozszerzenie go o tzw. Preprocesor – **SASS**. Dodaje on nam możliwość deklarowania zmiennych, tworzenia funkcji i większej ilości reużywalnych bloków kodu.<sup>3</sup> Na użycie tego preprocesora także zdecydowałem się w pracy inżynierskiej. Jednak sam czysty

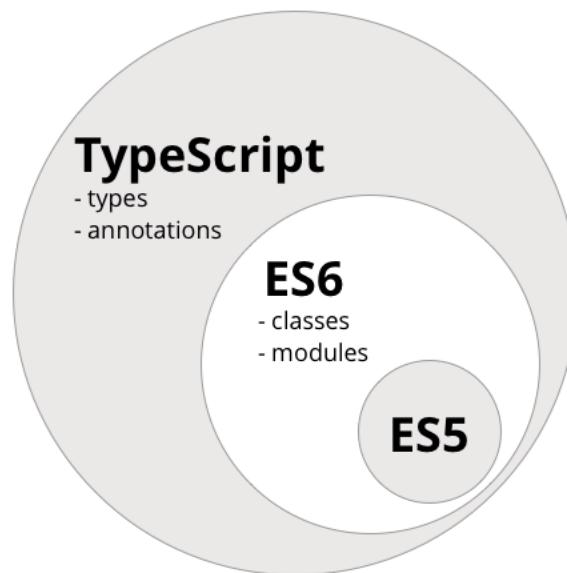
---

<sup>2</sup> <https://docs.docker.com/compose/overview/>

<sup>3</sup> <https://sass-lang.com/>

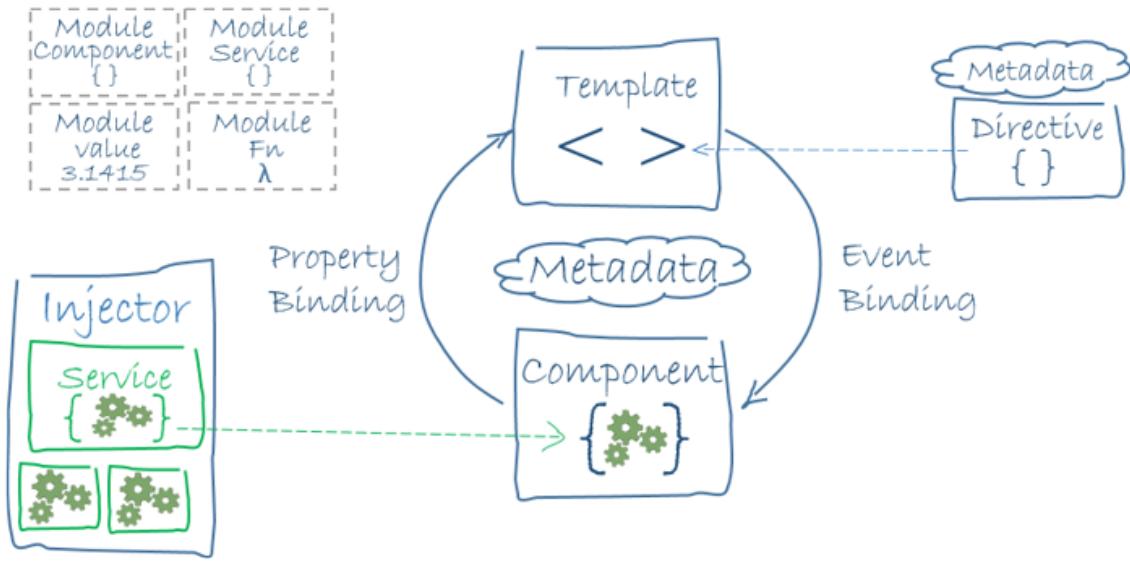
JavaScript, HTML i CSS to za mało by móc umożliwić łatwe skalowanie aplikacji i jej potencjalny przyszły rozwój w zrównoważonym środowisku. Dlatego zdecydowałem się na użycie frameworka **Angular** w wersji **2+** tworzonego głównie przez firmę **Google**. Dzięki temu rozwiązaniu jestem w stanie dzielić aplikację na logiczne części, które w trakcie pracy można bezproblemowo dopisywać. Te części to **komponenty** na które składa się: widok napisany w HTMLu (jest to HTML poszerzony o dodatkowe dyrektywy i konstrukcje dostarczane przez Angulara), wygląd komponentów definiowany jest w dołączonym pliku CSS, z kolei logika komponentu jest pisana jako eksportowana klasa z wykorzystaniem języka **TypeScript**.

TypeScript to tak naprawdę język JavaScript rozszerzony o możliwość deklarowania typów, interfejsów i innych możliwości znanych z języków takich jak C#. Wybór padł na TypeScript, ponieważ jest on domyślnie wspierany przez Angular, a sam Angular napisany jest właśnie z wykorzystaniem powyższego.



Rysunek 9 Czym jest TypeScript

Architektura aplikacji Angularowej jest przedstawiona na poniższym obrazku pobranym z oficjalnej dokumentacji frameworka.



Rysunek 10 Architektura aplikacji Angular

Oprócz wspomnianych wyżej komponentów drugim najważniejszym mechanizmem Angulara są tzw. Serwisy. Służą one do przechowywania logiki naszej aplikacji i odciążanie komponentów. W komponentach przechowujemy logikę związaną z tymi konkretnymi komponentami, zaś serwisy służą do przechowywania logiki na wyższym poziomie abstrakcji. Bardzo częstym przypadkiem użycia serwisu jest udostępnianie za jego pomocą interfejsu do wykonywania zapytań HTTP do czego między innymi ja używam serwisów w mojej aplikacji. Sam serwis to klasa udostępniająca określoną funkcjonalność która dzięki zastosowaniu dekoratora `@Injectable` jest w stanie być wstrzykiwana do jakiegokolwiek komponentu z wykorzystaniem wzorca wstrzykiwania zależności.

### 5.2.3 Warstwa serwerowa

W mojej pracy inżynierskiej oprócz części klienckiej stworzyłem także część odpowiadającą za wszelkie akcje wykonujące się po stronie serwera. W tym miejscu pozwoliłem sobie na mały eksperyment i połączylem kilka elementów by stworzyć dobrze funkcjonujący i skalowalny backend.

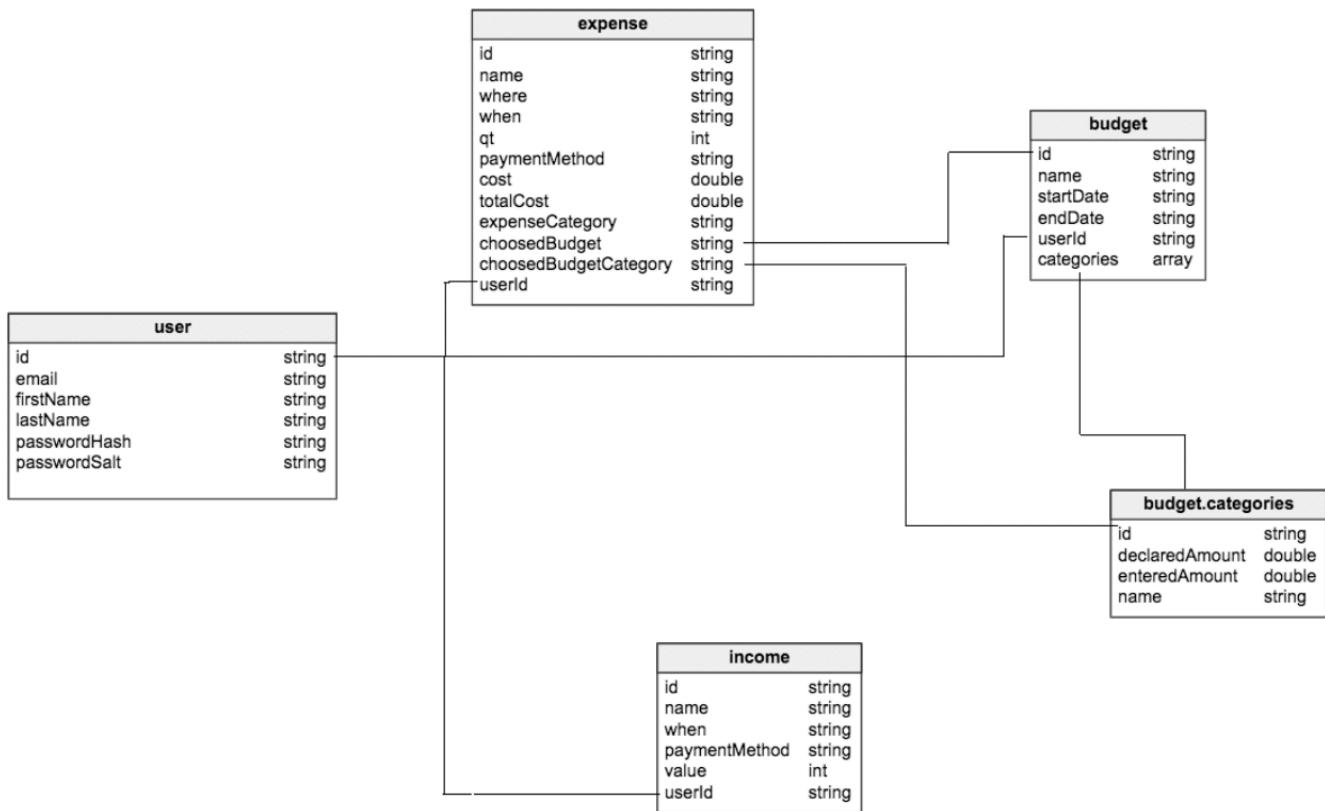
Po pierwsze użyłem frameworka **NestJS** stworzonego przez polskiego programistę **Kamila Myśliwca**, który garściącmi czerpie z wzorców przedstawionych w Angularze. Między innymi takich jak wstrzykiwanie zależności, czy też tworzenie komponentów. Oprócz tego, framework daje nam

możliwość tworzenia kontrolerów odpowiadających między innymi za wystawianie RESTowych końcówek przez które można zwracać i przekazywać dane. Jednocześnie, **NestJS** korzysta z biblioteki **Express.js**, która oparta jest na **Node.js** czyli implementacji języka **JavaScript** w środowisku serwerowym. Tutaj także pokusiłem się o ulepszenie języka jego typowaną wersją czyli skonfigurowałem wykorzystanie języka **TypeScript**.

#### 5.2.4 Przechowywanie danych

Do zapisywania i przechowywania danych wykorzystałem nierelacyjną bazę danych **MongoDB**. Powodem wyboru tej technologii była bardzo duża ilość materiałów i pomocy na temat tejże bazy danych. Swego czasu jednym z bardzo popularnych stosów technologicznych do wytwarzania oprogramowania był tak zwany MEAN stack. Rozwinięciem tego akronimu jest: **Mongo**, **Express**, **Angular**, **Node**. Postanowiłem więc wykorzystać ten stos technologiczny poszerzając go o własną konfigurację (wspomniany NestJS i wykorzystanie języka TypeScript). By nadać trochę zasad i uporządkowania do bazy danych opartej na MongoDB wykorzystałem bibliotekę ODM – (Object Data Modeling – modelowanie danych obiektowych) – **mongoose**. Zapewnia ona rygorystyczne środowisko do modelowania danych, wymuszając strukturę, przy jednoczesnym zachowaniu elastyczności.

Z powodów nierelacyjności bazy danych, poniżej przedstawiam pseudo model bazy danych, przedstawiający istniejące i pseudo powiązania pomiędzy nimi.



Rysunek 11 – model bazy danych

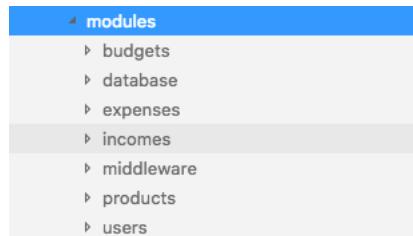
Jako komentarz do tego modelu warto nadmienić, że nie przetrzymuję oczywiście haseł użytkownika w bazie danych a jedynie sól i zahashowane posolone hasło potrzebne do późniejszej validacji.

## 6 Projekt techniczny

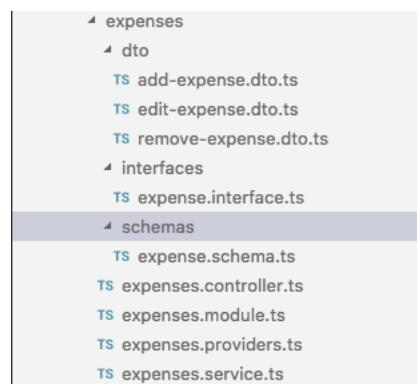
W tym rozdziale przedstawię garść informacji technicznych w jaki sposób aplikacja została podzielona i jakie są najważniejsze jej części.

### 6.1 Część serwerowa

Część serwerowa to aplikacja napisane z wykorzystaniem języka **TypeScript** w frameworku **NestJS**, wystawiająca dla warstwy klienckiej końcówki **RESTowe** na które można wysyłać zapytania **HTTP** takie jak **GET**, **POST** i inne. Dzięki zastosowanemu frameworkowi aplikacja jest podzielona na logiczne moduły, które są bezpośrednio powiązane z modelem bazy danych i dają niejako interfejs do komunikacji z nim.



Rysunek 12



Rysunek 13

Każdy z modułów składa się z pokazanej na zrzutach ekranów struktury plików, w której najważniejszymi do przeanalizowania są **\*.controller.ts**, **\*.service.ts**, **\*.schema.ts**

### \*.schema.ts

Jest to plik który definiuje schemat kolekcji w bazie danych **MongoDB**, dzięki niemu baza danych **MongoDB** staje się mniej losowa, lecz łączy w sobie zalety elastyczności i zorganizowania

```
import * as mongoose from "mongoose";

export const ExpenseSchema = new mongoose.Schema({
  name: String,
  where: String,
  when: String,
  qt: Number,
  paymentMethod: String,
  cost: Number,
  totalCost: Number,
  expenseCategory: String,
  choosedBudget: String,
  choosedBudgetCategory: String,
  userId: String,
});
```

Rysunek 14

### \*.controller.ts

Jest to plik który definiuje wszystkie końcówki i możliwe zapytania http, które możemy wykonywać. To tu odbywa się pierwsza komunikacja pomiędzy warstwą kliencką a serwerową. Dzięki temu, że framework **NestJS**, zaleca rozdzielanie logiki na wiele plików, każda końcówka w kontrolerach zwraca się do pliku **\*.service.ts** w celu wykonania konkretnych operacji dla określonej końcówki. Dzięki temu w pliku z kontrolerem nie ma chaosu i wszystkie operacje interakcji z bazą danych i inne odbywają się w pliku z serwisem.

```
@Controller("expenses")
export class ExpensesController {
  constructor(private expenseService: ExpensesService) {}

  @Post("add")
  async addExpense(@Body() addExpenseDto: AddExpenseDto) {
    return this.expenseService.addExpense(addExpenseDto);
  }

  @Put("edit")
  async editExpense(@Body() editExpenseDto: EditExpenseDto) {
    return this.expenseService.editExpense(editExpenseDto);
  }

  @Delete("remove")
  async removeExpense(@Body() removeExpenseDto: RemoveExpenseDto) {
    return this.expenseService.removeById(removeExpenseDto);
  }

  @Get(":id")
  async getExpenses(@Param() params): Promise<IExpense[]> {
    return this.expenseService.getExpenses(params.id);
  }

  @Get()
  async getAllExpenses(): Promise<IExpense[]> {
    return this.expenseService.getAllExpenses();
  }
}
```

**\*.service.ts**

Jest to plik który definiuje wszystkie działania, kalkulacje, interakcje z bazą danych dla określonego modułu (w pokazywanych przykładach do modułu zarządzającego wydatkami). Warto wspomnieć, że framework **NestJS**, jest zbudowany w oparciu o **Express.js**, i każda funkcja wykonująca operacje na bazie danych zwraca **Promise**. Dzięki temu, że **NestJS**, korzysta z najnowszych możliwości języka, możemy używać tutaj konstrukcji **async ... await** dzięki którym kod jest bardziej czytelny i poukładany.

```
@Component()
export class ExpensesService {
    constructor(
        @Inject("ExpenseModelToken") private readonly expenseModel: Model<IExpense>,
    ) {}

    async addExpense(addExpenseDto: AddExpenseDto): Promise<IExpense> {
        const addedExpense = new this.expenseModel(addExpenseDto);
        return await addedExpense.save();
    }

    async editExpense(editExpenseDto: EditExpenseDto) {
        const id = editExpenseDto._id;

        return await this.expenseModel.findByIdAndUpdate(id, editExpenseDto);
    }

    async getAllExpenses(): Promise<IExpense[]> {
        return this.expenseModel.find().exec();
    }

    async getExpenses(userId): Promise<IExpense[]> {
        return this.expenseModel.find({userId}).exec();
    }

    async removeById(removeExpenseDto: RemoveExpenseDto): Promise<IExpense> {
        return this.expenseModel.findByIdAndRemove(removeExpenseDto._id).exec();
    }
}
```

Oprócz tego warto wspomnieć tutaj o tym co widzimy w konstruktorze. Do każdego modułu, dodajemy informację o jaką kolekcję w bazie danych chodzi, z jakim modelem mamy do czynienia. To co widzimy w konstruktorze jest odzwierciedleniem tego co definiujemy w samym module, w pliku którego nie omawiam.

```
{
    provide: "ExpenseModelToken",
    useFactory: (connection: Connection) => connection.model("Expenses", ExpenseSchema),
    inject: ["DbConnectionToken"],
},
```

## 6.2 Część kliencka

Część kliencka to aplikacja napisana z wykorzystaniem frameworka **Angular**. Aplikacja angularowa składa się także z modułów w których definiujemy poszczególne **komponenty** i **serwisy**, które składają się na całą aplikację. Naczelną zasadą kompozycji takiej aplikacji jest tworzenie komponentów, które posiadają minimalną ilość logiki biznesowej i implementacja tejże w serwisach. Takie podejście także starałam się przyjąć w mojej pracy. Jest kilka ważnych i interesujących elementów aplikacji, które z chęcią przedstawię.

### Router

Skoro jest to SPA to nieodłączną częścią takiej aplikacji jest po stronie klienta router. Wykorzystywany jest tutaj domyślnie dostarczony z Angularem router. Na poniższym zrzucie ekranu widać poszczególne ścieżki na które dzięki routerowi możemy wejść.

```
const appRoutes: Routes = [
  {
    path: "home",
    component: HomeComponent,
    canActivate: [AuthGuard],
    canActivateChild: [AuthGuard],
    children: [
      { path: "dashboard", component: DashboardComponent },
      { path: "add-product", component: AddProductComponent },
      { path: "add-expense", component: AddExpenseComponent },
      { path: "add-income", component: AddIncomeComponent },
      { path: "add-budget", component: AddBudgetComponent },
      { path: "manage-products", component: ManageProductsComponent },
      { path: "manage-expenses", component: ManageExpensesComponent },
      { path: "manage-incomes", component: ManageIncomesComponent },
      { path: "manage-budgets", component: ManageBudgetsComponent },
      { path: "manage-monthly-fees", component: ManageMonthlyFeesComponent },
      { path: "chart-and-analyze", component: ChartAndAnalyzeComponent },
      { path: "", redirectTo: "/home/dashboard", pathMatch: "full" },
      { path: "**", component: DashboardComponent },
    ],
  },
  { path: "register", component: RegisterComponent },
  { path: "login", component: LoginComponent },
  { path: "", redirectTo: "/login", pathMatch: "full" },
  { path: "**", component: LoginComponent },
];

```

Rysunek 15 – angular router i dostępne ścieżki

W tym miejscu należy powiedzieć o specjalnym polu **canActivate**, które odpowiada za kontrolę uwierzytelniania użytkownika. Używa ono pliku **AuthGuard**, który ma za zadanie sprawdzić czy w **LocalStorage**, została zapisana poprawna informacja o logowaniu użytkownika. Jeśli tak, pole **canActivate** przepuszcza użytkownika do kolejnych ścieżek.

Integralną częścią routera jest możliwość wykorzystania atrybutu **routerLink**, na elemencie odsyłacza. Wykorzystuję ten mechanizm między innymi w wysuwanym menu.

```
<nav class="main-nav-container">
  <ul>
    <li><a mat-raised-button color="warn" (click)="sidenav.close()" routerLink="/home/dashboard">Dashboard</a></li>
    <li><a mat-raised-button color="primary" (click)="sidenav.close()" routerLink="/home/add-expense">Dodaj wydatek</a></li>
    <li><a mat-raised-button color="primary" (click)="sidenav.close()" routerLink="/home/add-income">Dodaj przychód</a></li>
    <li><a mat-raised-button color="primary" (click)="sidenav.close()" routerLink="/home/add-budget">Dodaj budżet</a></li>
    <li><a mat-raised-button color="accent" (click)="sidenav.close()" routerLink="/home/manage-expenses">Zarządzaj wydatkami</a></li>
    <li><a mat-raised-button color="accent" (click)="sidenav.close()" routerLink="/home/manage-incomes">Zarządzaj przychodami</a></li>
    <li><a mat-raised-button color="accent" (click)="sidenav.close()" routerLink="/home/manage-budgets">Zarządzaj budżetami</a></li>
    <li>Zalogowany jako:</li>
    <li><strong>{{ userDataRepository.userData?.firstName }} {{ userDataRepository.userData?.lastName }}</strong></li>
    <li><a mat-raised-button color="warn" (click)="logout()">Wyloguj</a></li>
  </ul>
```

Rysunek 16 – wykorzystanie atrybutu **routerLink**

### confirmation-modal.component

Jest to reużywalny komponent do potwierdzania wszelkich czynności (dodawanie, usuwanie, edycja). Stworzyłem taki komponent w celu zastosowania zasady **DRY** – dont repeat yourself. Jest to dobry przykład tego, że komponenty w angularze mogą być używane w wielu miejscach.

```
export class ConfirmationModalComponent {  
  
    constructor(  
        public dialogRef: MatDialogRef<ConfirmationModalComponent>,  
        @Inject(MAT_DIALOG_DATA) public data: any,  
        public snackBar: MatSnackBar,  
    ) {}  
  
    public cancel() {  
        this.dialogRef.close(this.data);  
    }  
  
    public openSnackBar(data) {  
        this.snackBar.openFromComponent(GenericSnackbarComponent, {  
            duration: 1000,  
            data,  
        });  
    }  
  
    public confirm() {  
        this.data.confirmed = true;  
        this.openSnackBar(this.data);  
        this.dialogRef.close(this.data);  
    }  
}
```

Rysunek 17 – reużywalny komponent

Tutaj warto także wspomnieć, że do budowania interfejsu użytkownika używam także komponentów z biblioteki **angular material**. Chociażby widoczna tutaj referencja do **MatDialogRef** i **MatSnackBar** to referencje do reużywalnych komponentów do pokazywania modala i tzw. snack bara.

## Typowanie i interfejsy

Angular korzysta z języka TypeScript, więc w aplikacji mamy zdefiniowane wiele interfejsów. Dla każdego wydatku, przychodu, budżetu zdefiniowałem oddzielne interfejsy, które pokrywają się z schematem bazy danych po stronie backendowej. Przykłady interfejsów w Angular są przedstawione na poniższych zrzutach ekranu.

```
export interface IIIncome {  
    _id: string;  
    _v: number;  
    name: string;  
    when: string;  
    paymentMethod: string;  
    value: number;  
}
```

Rysunek 18 – interfejs IIIncome

```
export interface ICategories {  
    _id: string;  
    name: string;  
    declaredAmount: number;  
    enteredAmount: number;  
}  
  
export interface IBudget {  
    _id: string;  
    name: string;  
    startDate: string;  
    endDate: string;  
    categories: ICategories[];  
}
```

Rysunek 19 – Interfejs IBudget i ICategories

## Serwisy wysyłające zapytania HTTP

Oprócz komponentów mam wiele serwisów których celem jest wykonywanie zapytań RESTowych do warstwy serwerowej. Przykład takiego serwisu na zrzucie ekranu poniżej. Jako, że angular garściąmi czerpie z wzorca projektowego **Observer**, dane zwracane z zapytania http to **Observables**< dane zwarcane >. <sup>4</sup> W takim wypadku w komponencie w którym wywołuję metodę z tegoż serwisu, muszę wykonać operację **subskrypcji** i oczekiwając na zmiany, którą mogę przypisać do konkretnej zmiennej z komponentu.

```
@Injectable()
export class ExpenseService {
  private apiUrl: string = API_URL;

  constructor(private http: HttpClient) { }

  public addExpense(expense: IExpense) {
    const headers = new HttpHeaders();
    headers.set("Content-Type", "applications/json");

    const userId = decodeJwt(localStorage.getItem("jwt"))._id;

    const expenseWithUserId = {
      ...expense,
      userId,
    };

    return this.http.post(`${this.apiUrl}expenses/add`, expenseWithUserId, { headers });
  }

  public editExpense(expense: any): Observable<any> {
    const headers = new HttpHeaders();
    headers.set("Content-Type", "applications/json");

    return this.http.put(`${this.apiUrl}expenses/edit`, expense, { headers });
  }

  public showExpenses(): Observable<IExpense[]> {
    const userId = decodeJwt(localStorage.getItem("jwt"))._id;

    return this.http.get<IExpense[]>(`${this.apiUrl}expenses/${userId}`);
  }

  public showAllExpenses(): Observable<IExpense[]> {
    return this.http.get<IExpense[]>(`${this.apiUrl}expenses`);
  }

  public removeExpense(expense: any): Observable<any> {
    return this.http.request("delete", `${this.apiUrl}expenses/remove`, { body: expense });
  }
}
```

Rysunek 20 – serwis wykonujący zapytania http

---

<sup>4</sup> <https://angular.io/guide/observables>

```
this.budgetService.showBudgets().subscribe(budgets => {
  this.allBudgets = budgets;
});
```

Rysunek 21 – przykład przypisywania wyniku zapytania w subskrypcji do zmiennej

```
this.expenseService.showExpenses().subscribe(
  expenses => {
    this.expenses = expenses;
    this.expenseCategoryChartLabels.map( category => {
      const filteredCategory = this.expenses.filter( expense => expense.expenseCategory === category);
      if (!filteredCategory) {
        const acc = 0;
      }
      const acc = filteredCategory.reduce( (prev, curr) => prev + curr.totalCost, 0);

      this.expenseCategoryChartData.push(acc);
    });
    this.isExpenseCategoryLoaded = true;

    this.paymentMethodChartLabels.map( method => {
      const filteredPayment = this.expenses.filter( expense => expense.paymentMethod === method);
      if (!filteredPayment) {
        const acc = 0;
      }
      const acc = filteredPayment.reduce( (prev, curr) => prev + curr.totalCost, 0);

      this.paymentMethodChartData.push(acc);
    });
    this.isPaymentMethodLoaded = true;
  },
);
```

Rysunek 22 – bardziej rozbudowany przykład wykonywania działań po wykonaniu zapytania

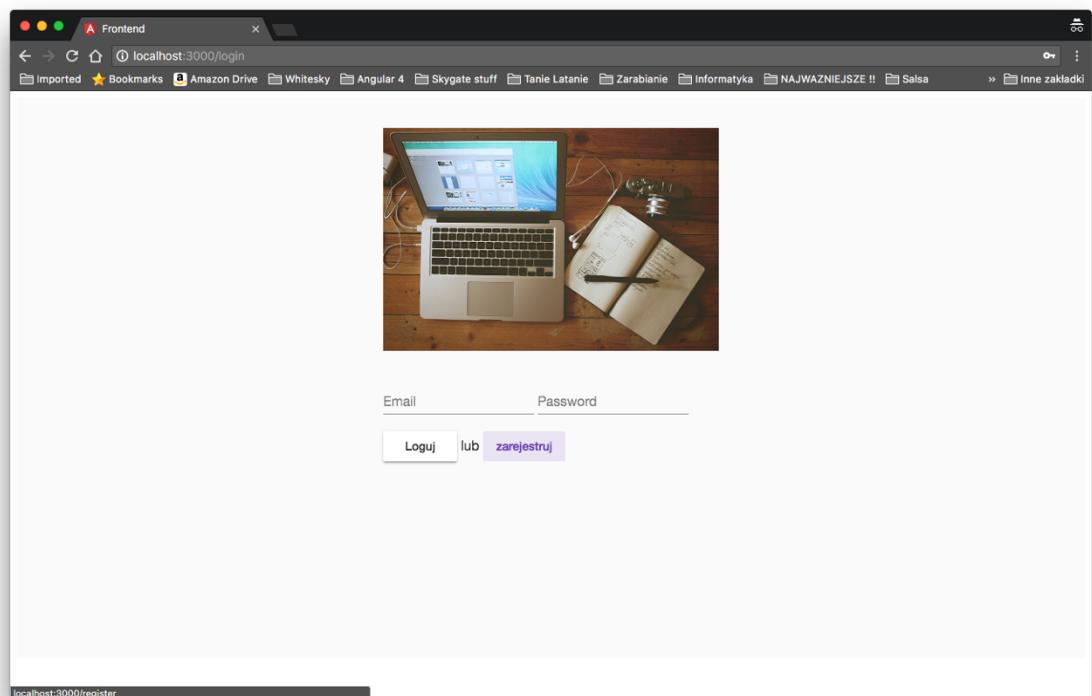
## 7 Testy i weryfikacja systemu

Testowanie funkcjonalności serwera a także klienta odbywało się przez cały proces pisania pracy. Po napisaniu każdej funkcjonalności, sprawdzałem ją pod względem poprawności działania. Końcówki RESTowe testowałem za pomocą programu **Postman**, w którym bardzo sprawnie można wykonywać proste jak i skomplikowane zapytania HTTP. Napisałem także kilka testów jednostkowych, jednak w tak dynamicznym programowaniu jakim jest tworzenie oprogramowania w jedną osobę do pracy dyplomowej okazuje się to niekoniecznie wydajnym rozwiążaniem. Najważniejsze były testy manualne, do których podchodziłem bardzo skrupulatnie, zapisując poszczególne kroki które muszę wykonać i przejść po implementacji każdej funkcjonalności.

## 8 Przykładowy scenariusz wykorzystania systemu

Główym celem aplikacji jest umiejętne wprowadzanie danych i ich analiza. I taki właśnie przykładowy sposób wykorzystania aplikacji przedstawię w tym rozdziale.

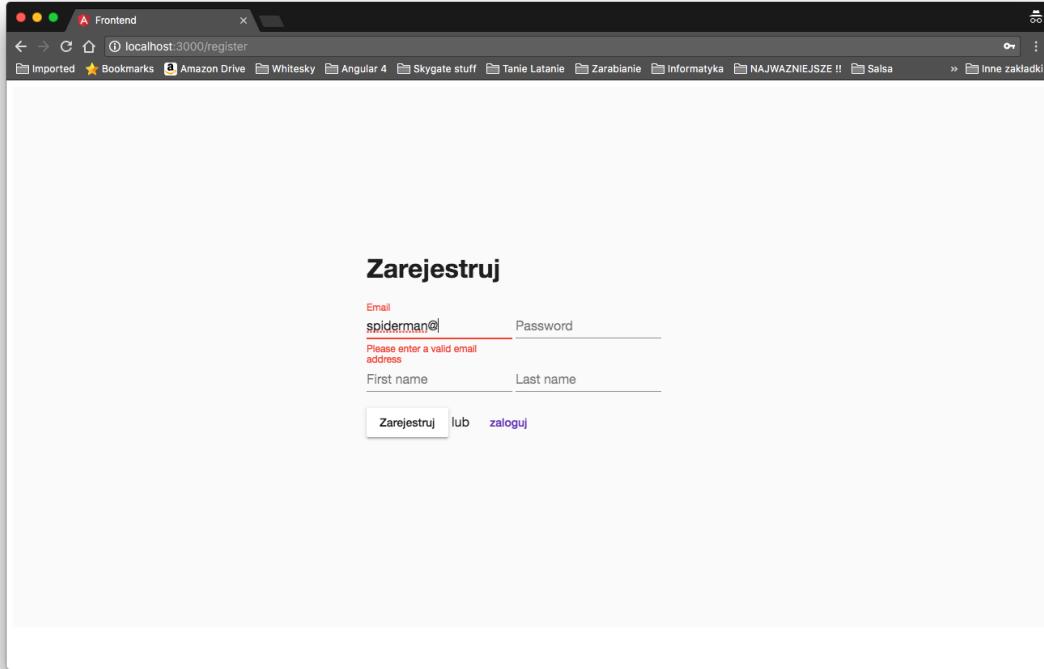
Zaczniemy od tego, że użytkownik musi założyć swoje własne konto. W tym celu ze strony głównej aplikacji przechodzi do rejestracji po kliknięciu przycisku **rejestruj.**



Rysunek 23 – ekran logowania

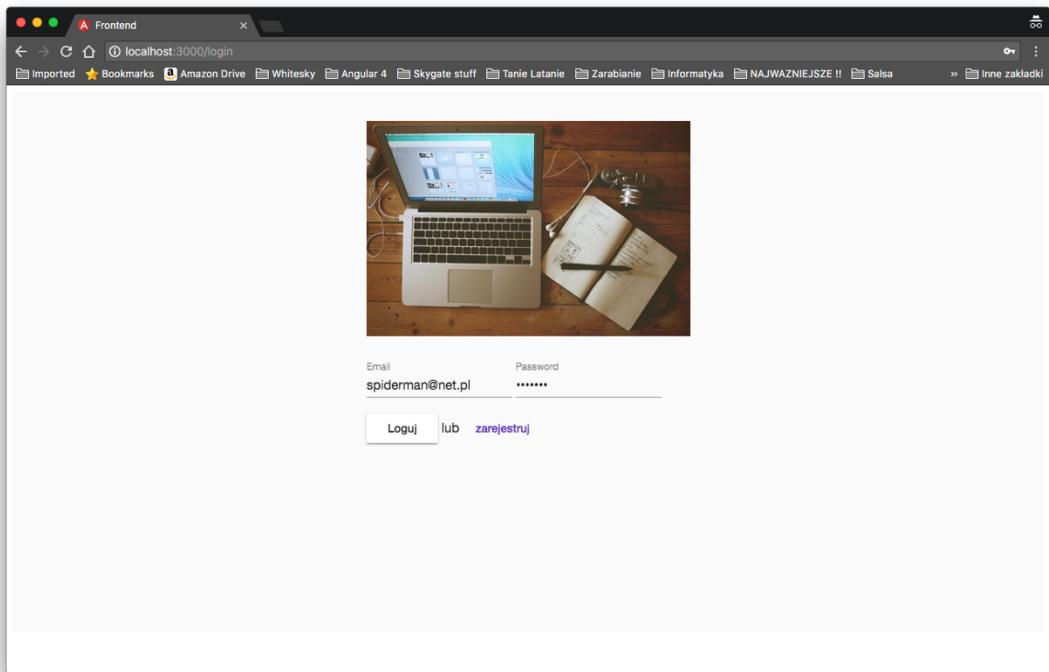
Po przejściu na stronę rejestracji, użytkownik wpisuje swoje dane takie jak email, hasło, imię i nazwisko. Przy każdej zmianie pola tekstowego system sprawdza czy wprowadzone dane są poprawne i w przypadku niepoprawności wyświetla stosowny komentarz i nie pozwala się zarejestrować póki błędy nie zostaną poprawione. Warto tutaj nadmienić, że rejestracja i logowanie odbywa się w bezpieczny sposób przy pomocy posolenia hasła i hashowania, dzięki czemu

nie przechowujemy hasła użytkownika w bazie danych, tylko jego zahashowaną i posoloną wersję.



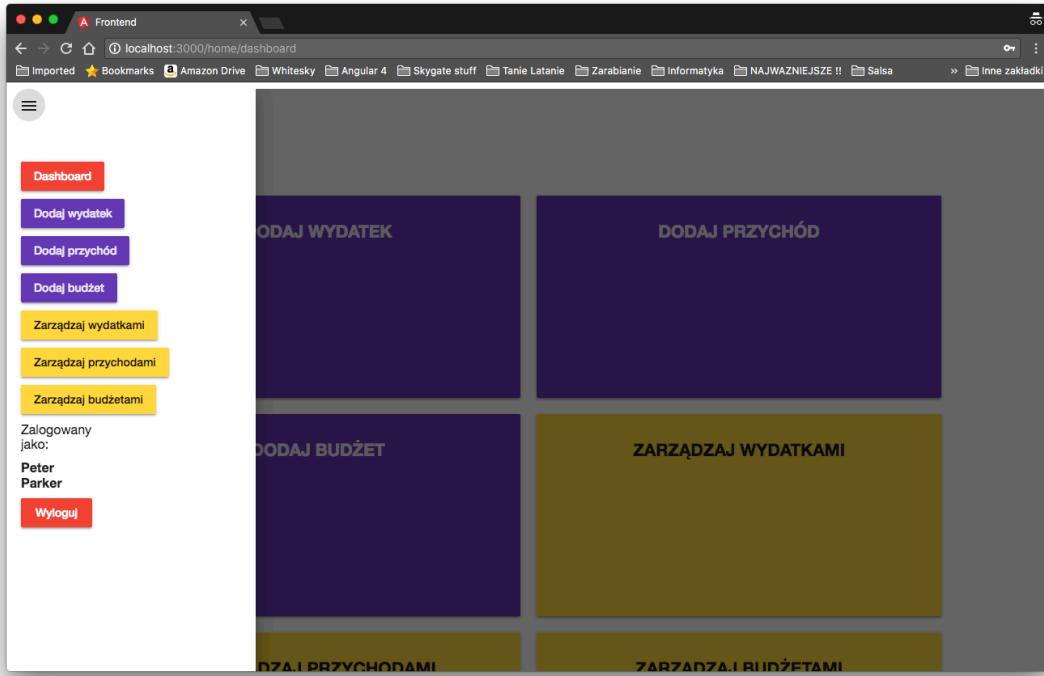
Rysunek 24 – ekran rejestracji z błędnym adresem email

Po poprawnym zarejestrowaniu przechodzimy z powrotem na stronę z logowaniem, wprowadzamy swoje dane (email i hasło) i wybieramy przycisk zaloguj (możemy także po prostu potwierdzić wpis klawiszem ENTER).



Rysunek 25 – ekran logowania z poprawnymi danymi

Po wypełnieniu tego kroku ukazuje nam się główny ekran naszej aplikacji z przyciskami które prowadzą nas do każdej z funkcji systemu. Oprócz przycisków na stronie głównej, mamy także dostęp w lewym górnym rogu do wysuwanego z lewej ścianki menu, w którym umieszczone są te same odnośniki a także informacja o aktualnie zalogowanym użytkowniku i możliwość wylogowania z aplikacji.



Rysunek 26 – ekran główny i wysuwane menu

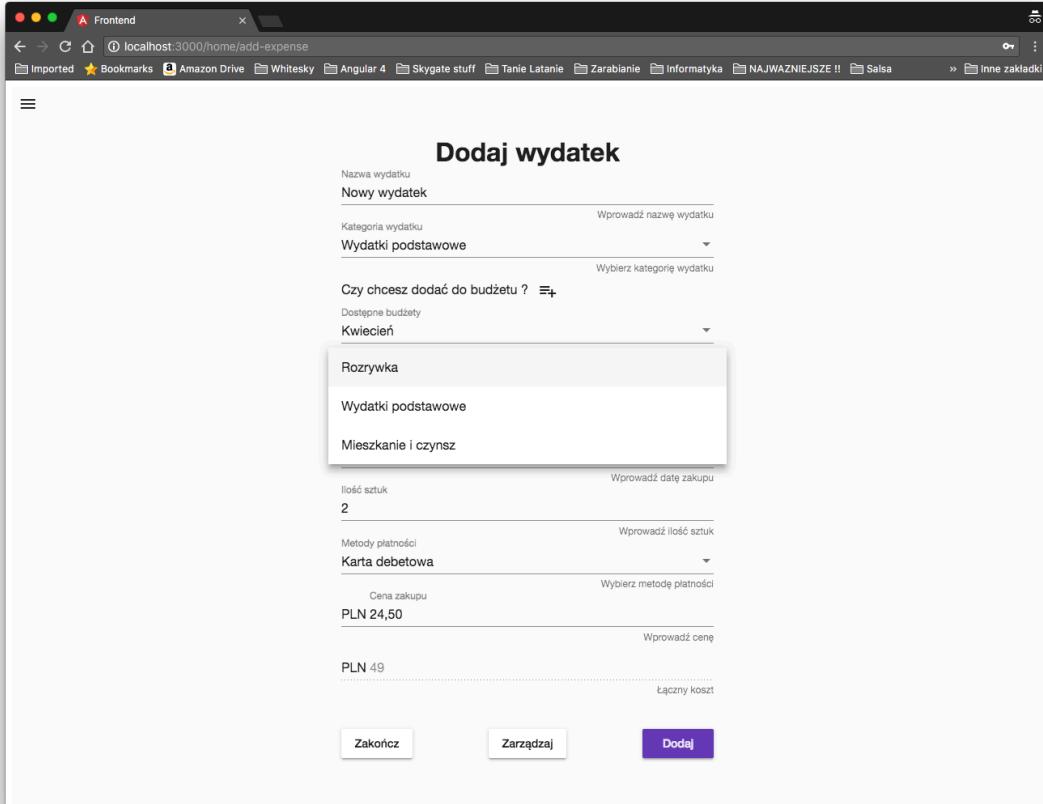
Użytkownik może teraz podjąć każde z działań, natomiast by jak najwięcej zyskać z naszej aplikacji powinniśmy w tym momencie utworzyć pierwszy budżet na najbliższy miesiąc, byśmy mogli bezproblemowo dodawać kolejne wydatki do zdefiniowanego budżetu. W oknie dodawania budżetu możemy zdefiniować dla niego nazwę, datę rozpoczęcia, datą zakończenia. Ostatnim i najważniejszym elementem tworzenia budżetu jest stworzenie kategorii powiązanych z tym budżetem, do których będziemy przypisywać wydatki. Nazwy proponowanych kategorii do na przykład **Rozrywka**, **Wydatki podstawowe**, **Mieszkanie i czynsz**. Kategorie powinny być wyznacznikiem tego, na jakie sfery naszego życia chcemy nałożyć pewnego rodzaju limit. Chociażby kategoria **Rozrywka** daje nam informację o tym, ile możemy jeszcze pozwolić sobie na rzeczy niekoniecznie potrzebne lecz umilające nam codzienność.

Rysunek 27 – ekran dodawania budżetu wypełniony informacjami

Oczywiście w każdym momencie kiedy wpisujemy informację, system sprawdza czy informację mają sens, czy wypełniliśmy wszystkie pola i daje nam o tym znać specjalną wiadomością tak jak było to w przypadku rejestracji.

W tym momencie możemy dodać nasz pierwszy wydatek. Ekran dodawania w celu wygenerowania tego samego poczucia estetyki wygląda bardzo podobnie, różni się jedynie ilością pól, które możemy wypełnić.

W momencie dodawania wydatku możemy zdefiniować dla niego takie właściwości jak: nazwa, kategoria (globalna), gdzie został wykonany, kiedy, ilość sztuk, metoda płatności, i cena. Oprócz tego możemy dodać go do budżetu który dopiero co stworzyliśmy bądź innych, co także zrobimy. Po wybraniu budżetu musimy zdefiniować także kategorie (lokalna) w budżecie do której chcemy go dopisać. Cały proces przedstawiają poniższe zrzuty ekranu.

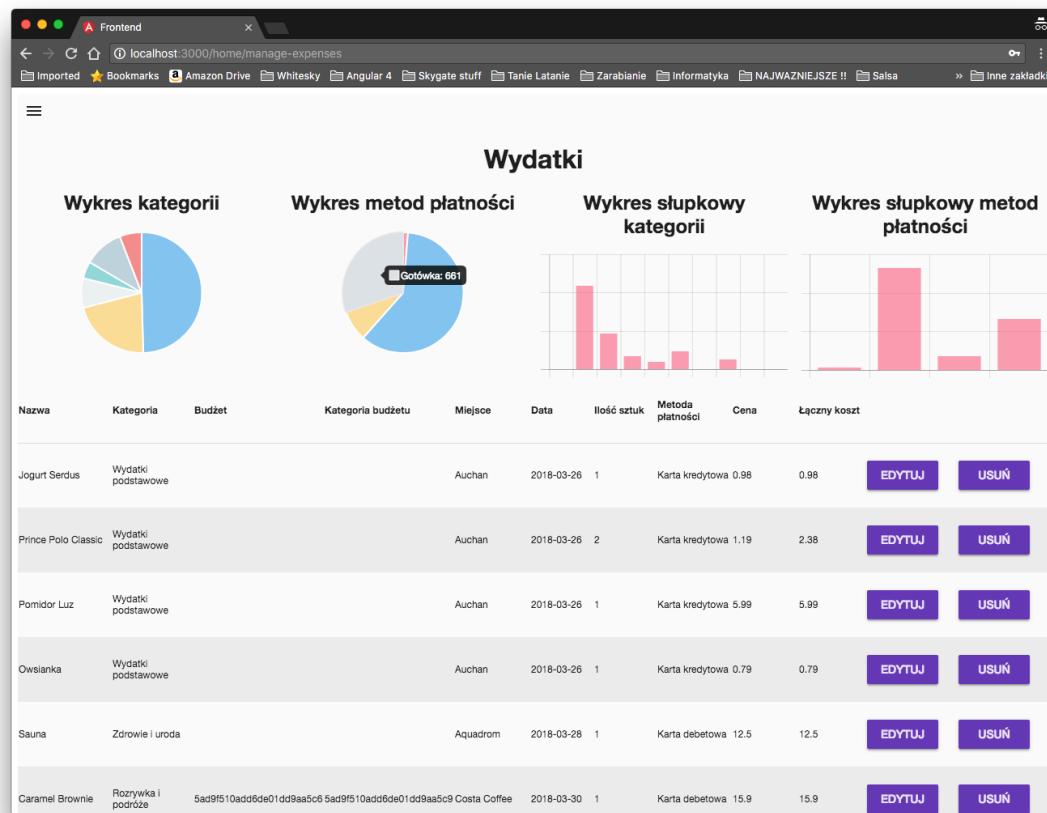


Rysunek 28

Tak wygląda prosty proces dodawania wydatku do określonego budżetu. Proces ten jest kluczową częścią dbania o swój budżet i daje nam możliwość w przyszłości monitorowania naszych pieniędzy.

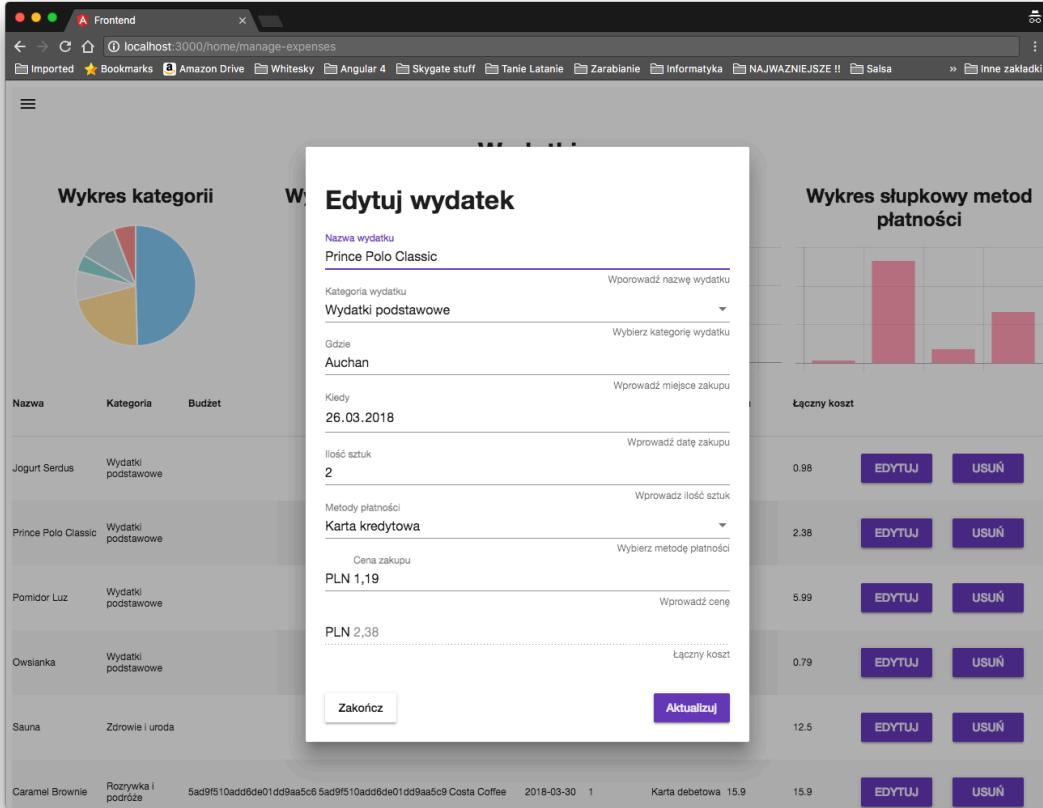
Dzięki temu, że mam przygotowane konto użytkownika który dodał już wiele informacji, pokażę teraz wygląd aplikacji i dalsze kroki po kilkunastu dniach użytkownika by w pełni przedstawić drugą najważniejszą funkcję, czyli prezentacje zależności między wprowadzonymi danymi.

Przede wszystkim, każdy użytkownik ma także dostęp do widoku „Zarządzania wydatkami” (a także zarządzanie przychodami, analogiczny do widoku który teraz pokażę).



Rysunek 29 – widok zarządzania wydatkami

Z tego miejsca użytkownik może obejrzeć rozkład swoich finansów na interaktywnych wykresach na podstawie kategorii (globalnych) a także metody płatności. W przyszłości przewidywane jest wprowadzenie pełnego filtrowania, wydatków. Wykresy są stworzone w taki sposób, by odzwierciedlać aktualnie wyświetlane w tabeli wydatki, więc przy potencjalnym filtrowaniu tabeli, wykresy automatycznie wskażą podany zakres listy wydatków. Z tego widoku można także usunąć i edytować poszczególne wydatki, co automatycznie znajdzie swoje odzwierciedlenie w wykresach.

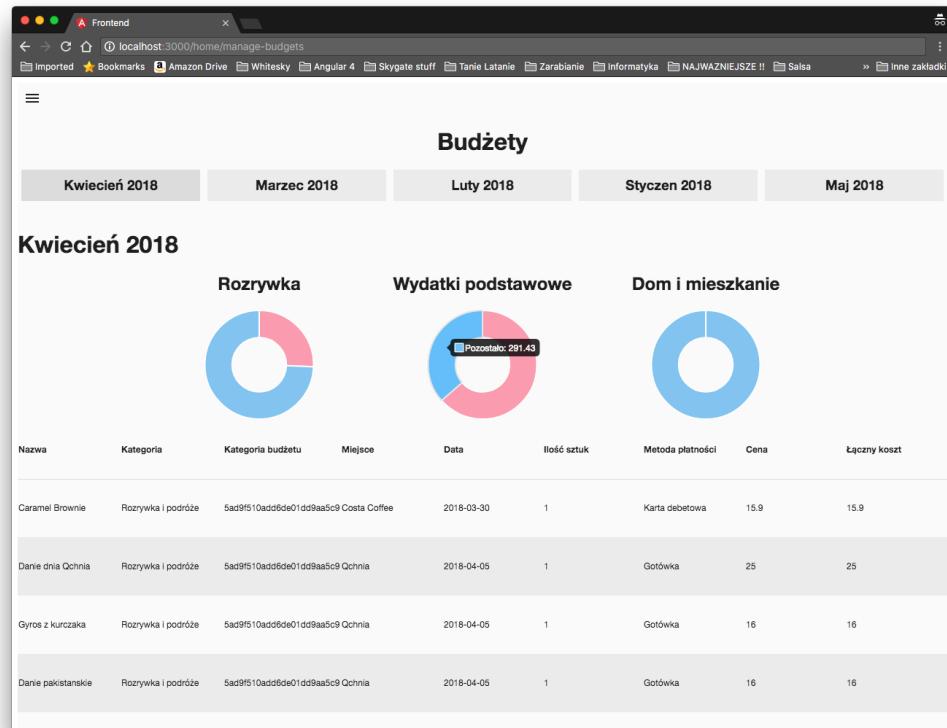


Rysunek 30 – edytowanie wydatku z poziomu zarządzania wydatkami

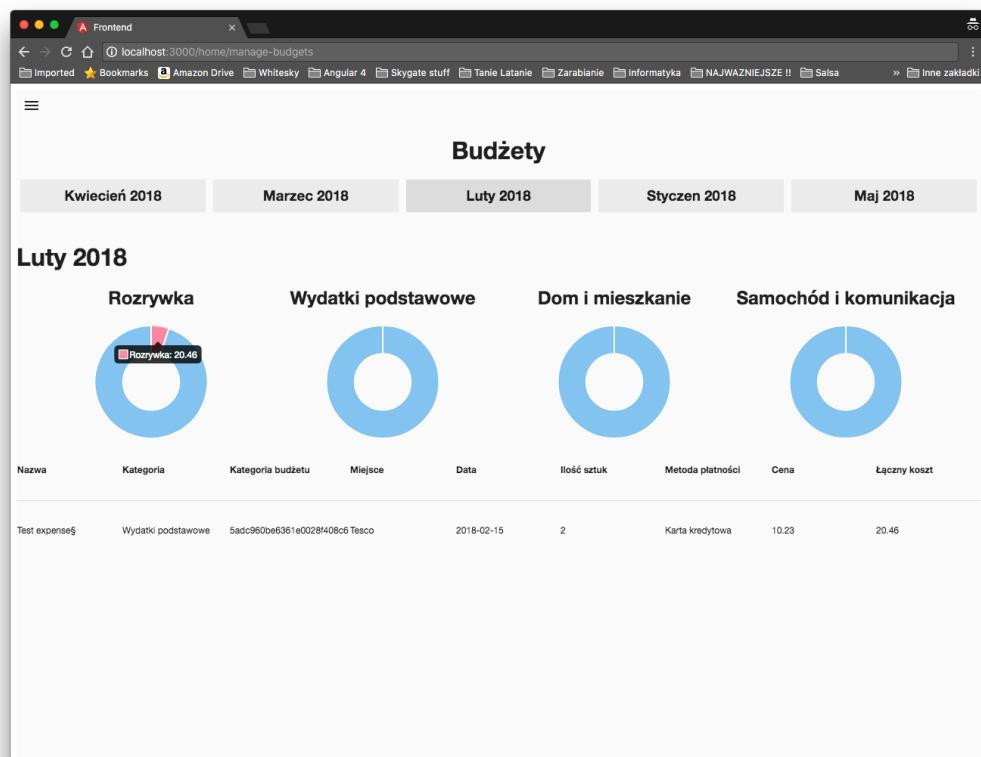
Ostatnim bardzo ważnym miejscem w którym na pewno chciałby się znaleźć użytkownik jest **Zarządzanie budżetami**.

To miejsce w którym użytkownik może obejrzeć wszystkie dane dotyczące konkretnych stworzonych przez niego budżetów w trakcie całego użytkowania aplikacji. Ma dostęp do listy z której może wybrać interesujący go budżet i dowiedzieć się jak dużo pieniędzy w tym budżecie zostało mu dla określonej kategorii.

## Opracowanie i implementacja budżetu osobistego



Rysunek 31 – zarządzanie budżetami



Rysunek 32 – zarządzanie budżetami 2

Oprócz dodawania wydatków, mamy także możliwość dodawania swoich przychodów i wyświetlania tak samo, strony z podsumowaniem z jakich kategorii użytkownik uzyskiwał przychody. W przyszłości także jest plan wprowadzenia analizy wydatków pomiędzy przychodami a wydatkami.

## 9 Zakończenie

Celem niniejszego projektu było stworzenie użytecznej aplikacji do zarządzania wydatkami i przychodami każdego człowieka. Głównymi założeniami było stworzenie systemu, w którym każdy mógłby przechowywać i analizować swoje codzienne życie finansowe. Ważnym aspektem było stworzenie przyjaznego, prostego i intuicyjnego interfejsu użytkownika do wprowadzania danych. To zawsze jest największy problem szczególnie w dzisiejszym zabieganym świecie, gdzie większość ludzi chce mieć wszystko od razu. Nie udało się zaimplementować automatycznego wprowadzania wydatków na podstawie paragonu, ale myślę, że proces który zaproponowałem, jest wystarczająco prosty, by potencjalny użytkownik, mógł szybko przyzwyczaić się do niego i dzięki regularności ulepszyć zarządzanie swoimi finansami. Forma graficzna często mówi więcej niż tysiące słów. Dlatego uważam, że interaktywne wykresy, które czytują dane już wprowadzone i wyświetlają je użytkownikowi są także bardzo dużym plusem w stworzonej aplikacji. Wizualna reprezentacja zawsze bardziej zapada w pamięć użytkownika.

Projektowanie i implementacja serwisu przysporzyła wiele problemów, które jednak w większości udało się rozwiązać. Dużo czasu musiałem poświęcić na skonfigurowanie poprawnie kontenerów Docker w celu łatwego procesu programowania. Poświęcony jednak na to czas zwrócił się w 100% kiedy musiałem skorzystać z innego komputera bez zainstalowanej bazy danych czy chociażby Node.js. Wystarczyło użyć dwóch prostych komend i środowisko było gotowe do pracy.

Także, konfiguracja i wykorzystanie frameworka **Nest.JS** nie należało do najprostszych głównie z powodu bardzo dużej świeżości rozwiązania. W momencie, w którym zaczynałem pracę inżynierską był to bardzo mało znany framework, który jednak zyskiwał bardzo szybko popularność. Liczby jednak nie kłamią i według GitHub biblioteka staje się coraz bardziej popularna i widzę że wybranie jej było dobrą decyzją. Wiedza nabыта w trakcie tworzenia aplikacji i programowania, na pewno przyda się w przyszłości.

Projekt ma duży potencjał, i to co zostało zaimplementowane to dopiero wierzchołek góry lodowej. Filtrowanie wydatków, przychodów, korelacja przychodów z wydatkami i wreszcie funkcja, która ułatwiła by codzienne wpisywanie wydatków, czyli automatyczne skanowanie paragonów to tylko kilka z możliwych usprawnień, które wprowadziły by aplikację na kolejny poziom.



## 10 Bibliografia

1. <https://www.youtube.com/watch?v=cLT7eUWKZpg&t=48s>, [dostępne: 25 kwietnia 2018]
2. <https://docs.docker.com/compose/overview/>, [dostępne: 10 kwietnia 2018].
3. <https://sass-lang.com/>, [dostępne: 10 kwietnia 2018].
4. <https://angular.io/guide/observables>, [dostępne: 11 kwietnia 2018].

## 11 Spis rysunków

Rysunek 1 Zobrazowanie problemu budżetu osobistego.....	5
Rysunek 2 Aplikacja Cents.....	6
Rysunek 3 Po lewej aplikacja Spendee w której dodanie konta bankowego jest dopiero dostępne w funkcji PREMIUM, w Polsce, jedynie 3 banki obsługiwane. Po prawej aplikacja Wallet, podobnie.....	7
Rysunek 4 Aplikacja Cents nie wzbudza zaufania. (a aplikacja zarządzająca naszymi pieniędzmi powinna) .....	8
Rysunek 5 – wysuwane boczne menu .....	10
Rysunek 6 – zasady responsywności – dostępność na każdym urządzeniu .....	11
Rysunek 7 – schemat dodawania wydatku/przychodu/budżetu.....	12
Rysunek 8 – podstrona budżetu osobistego z interaktywnymi wykresami .....	13
Rysunek 9 Czym jest TypeScript.....	17
Rysunek 10 Architektura aplikacji Angular.....	18
Rysunek 11 – model bazy danych.....	20
Rysunek 12 .....	21
Rysunek 13 .....	21
Rysunek 14 .....	22
Rysunek 15 – angular router i dostępne ścieżki .....	24
Rysunek 16 – wykorzystanie atrybutu <b>routerLink</b> .....	25
Rysunek 17 – reużywalny komponent.....	26
Rysunek 18 – interfejs IIIncome .....	26
Rysunek 19 – Interfejs IBudget i ICategories.....	27
Rysunek 20 – serwis wykonujący zapytania http.....	28
Rysunek 21 – przykład przypisywania wyniku zapytania w subskrypcji do zmiennej .....	29
Rysunek 22 – bardziej rozbudowany przykład wykonywania działań po wykonaniu zapytania .....	29
Rysunek 23 – ekran logowania.....	31
Rysunek 24 – ekran rejestracji z błędnym adresem email .....	32
Rysunek 25 – ekran logowania z poprawnymi danymi.....	33
Rysunek 26 – ekran główny i wysuwane menu.....	34
Rysunek 27 – ekran dodawania budżetu wypełniony informacjami .....	35
Rysunek 28.....	36
Rysunek 29 – widok zarządzania wydatkami.....	37

Rysunek 30 – edytowanie wydatku z poziomu zarządzania wydatkami .....	38
Rysunek 31 – zarządzanie budżetami.....	39
Rysunek 32 – zarządzanie budżetami 2 .....	39