Kierunek: ***Informatyka***

2024/2025

Kamil Dereń

PRACA INŻYNIERSKA

Projekt i implementacja systemu do rozliczania wydatków grupowych

Promotor pracy:
mgr inż. Tomasz Gądek

Tarnów, *2025*

Spis treści

1. Wstęp

Praca dyplomowa dotyczy problematyki rozliczania wydatków grupowych, koncentrując się na opracowaniu efektywnego rozwiązania ułatwiającego organizację i kontrolę wspólnych finansów. Głównym przedmiotem pracy jest aplikacja desktopowa oraz serwer, które mają wspierać użytkowników w sprawnym zarządzaniu tego rodzaju wydatkami. W kolejnych rozdziałach przedstawione zostaną zastosowane technologie, przykłady implementacji oraz finalny wygląd aplikacji.

1.1 Motywacja

Współczesne społeczeństwo coraz częściej staje przed wyzwaniem sprawnego i przejrzystego rozliczania wspólnych wydatków. Dotyczy to zarówno codziennych sytuacji, takich jak wspólne zakupy, jak i bardziej złożonych przedsięwzięć, na przykład współdzielenia mieszkania czy organizacji wyjazdów. W takich przypadkach pojawia się potrzeba sprawiedliwego podziału kosztów pomiędzy uczestników, co, choć na pierwszy rzut oka wydaje się proste, stwarza liczne trudności związane z błędami obliczeniowymi, brakiem przejrzystości czy problemami z historią wydatków.

Tradycyjne metody, obejmujące zapisywanie wydatków w kalkulatorze lub arkuszach kalkulacyjnych, często okazują się niewystarczające przy większej liczbie uczestników. Co więcej, manualne podziały kosztów stają się jeszcze bardziej skomplikowane, gdy wydatki muszą być dzielone w sposób nierównomierny. Tego typu trudności są szczególnie odczuwalne w przypadku działań długoterminowych, takich jak organizacja wspólnych przestrzeni do życia, wyjazdy integracyjne czy realizacja większych przedsięwzięć.

Motywacją do opracowania tej aplikacji była potrzeba uproszczenia i zautomatyzowania tego procesu. Dzięki automatycznemu podziałowi kosztów, aplikacja pozwala użytkownikom łatwo zarządzać wspólnymi wydatkami, zapewniając pełną przejrzystość oraz dostęp do historii rozliczeń. Zastosowanie zautomatyzowanych obliczeń oraz możliwość nierównego podziału kosztów pozwala uniknąć błędów, oszczędzając tym samym czas użytkowników.

W odpowiedzi na rosnącą potrzebę uproszczenia zarządzania finansami w grupach, aplikacja desktopowa stała się idealnym rozwiązaniem. Dzięki niej, użytkownicy mogą skupić się na innych aspektach wspólnych działań, nie martwiąc się o szczegóły związane z rozliczeniami. Opracowanie tego narzędzia odpowiada na wyzwania współczesnego życia, przyczyniając się do wygody i efektywności zarządzania wspólnymi wydatkami.

1.2 Cel Pracy

Celem niniejszej pracy jest zaprojektowanie systemu umożliwiającego rozdzielanie wydatków w obrębie grupy. Kluczową funkcjonalnością aplikacji jest algorytm sprawiedliwego podziału kosztów, który na podstawie wprowadzonych danych oblicza udział poszczególnych osób w kosztach danego produktu, zapewniając uczciwe rozliczenie wydatków. System pozwala na zakładanie kont użytkowników, dodawanie rachunków, produktów oraz osób, a także przypisywanie ich do poszczególnych produktów. Dodatkowo aplikacja oferuje moduł statystyk, generujący czytelne wykresy na podstawie zgromadzonych danych, oraz funkcje zarządzania pożyczkami i kontem użytkownika. Aby zapewnić niezawodność działania, system został wzbogacony o testy jednostkowe i integracyjne, które przyczyniły się do poprawy jakości użytkowania oraz eliminacji potencjalnych błędów w oprogramowaniu.

1.3 Zakres pracy

Zakres pracy inżynierskiej obejmował zaprojektowanie i implementację kompletnego systemu, składającego się z aplikacji desktopowej oraz serwera. Architektura projektu była starannie opracowana zgodnie z dobrymi praktykami inżynierii oprogramowania. Aplikację stworzono z wykorzystaniem platformy *.NET* w środowisku *Blazor*, co zapewniło spójność stylistyczną oraz umożliwiło implementację serwera i aplikacji w jednolitym środowisku technologicznym. Dane użytkowników były przechowywane w relacyjnej bazie danych *PostgreSQL*, udostępnionej w ramach toku studiów na Akademii Tarnowskiej. Moduł serwera został przetestowany za pomocą testów jednostkowych i integracyjnych.

2. Analiza biznesowa

Niniejszy rozdział zawiera przybliżenie głównych możliwości systemu, uwzględniając zarówno założenia funkcjonalne, jak i niefunkcjonalne odnoszących się do systemu rozliczania wydatków grupowych.

2.1. Wymagania funkcjonalne

System, który został zaprojektowany z myślą o wygodzie użytkowników, koncentruje się na umożliwieniu łatwego i przejrzystego rozliczania kosztów w ramach złożonych rachunków, co obejmuje zarówno podział wydatków pomiędzy osoby, jak i prezentację szczegółowych statystyk dotyczących ich finansowych udziałów. Jego funkcjonalności obejmują szereg narzędzi ułatwiających zarządzanie, w tym możliwość tworzenia konta użytkownika, logowania do systemu oraz intuicyjnego zarządzania swoim profilem, co pozwala na zmianę takich danych jak nazwa użytkownika, adres e-mail czy hasło. Dodatkowo system umożliwia dodawanie osób do utworzonego konta, którymi użytkownik będzie zarządzać i nakładać na nie koszta.

Istotną częścią funkcjonalności jest także możliwość tworzenia rachunków, które można szczegółowo opisać, podając nazwę, datę realizacji oraz opcjonalnie wartość rabatu wyrażoną w procentach. Każdy rachunek umożliwia dodawanie produktów, wraz z określeniem ich ceny, stopnia podzielności, liczby sztuk, rabatu, dodatkowej opłaty wynikającej z produktu, a następnie przypisanie produktów do odpowiednich osób. Całość uzupełnia rozbudowany moduł analityczny, który pozwala na monitorowanie wydatków wszystkich zaangażowanych osób za pomocą szczegółowych statystyk oraz przejrzystych wykresów, wspierając świadome zarządzanie wspólnymi finansami.

2.1.1. Algorytm rozdzielania cen produktu na części

Algorytm, który ma równo rozdzielać koszty pomiędzy osoby, powinien być dopracowany i uwzględniać różne czynniki wpływające na jego poprawne działanie. Z matematycznego punktu widzenia, przy dzieleniu produktów nie da się uniknąć sytuacji, w której wartość produktu nie będzie w pełni zdatna do równego podziału kosztów. Aby uniknąć takiej sytuacji, opracowany został mechanizm wyrównania.

Mechanizm wyrównania polega na przechowywaniu przez produkt kwotę wyrównania, będącej różnicą pomiędzy całkowitą ceną produktu a sumą części kosztów przypisanych do tego produktu. Koszt przypisania oblicza się w prosty sposób: cenę produktu mnoży

się przez liczbę przypisań, a wynik dzieli przez całkowitą liczbę przypisań, zaokrąglając zawsze w dół do dwóch miejsc po przecinku. Jedno z przypisań do produktu musi być oznaczone jako wyrównujące – to ono ponosi koszty różnicy wynikającej z wyrównania.

Każda osoba gromadzi sumę przypisanych jej kosztów wyrównania. Dzięki temu użytkownik lub system może monitorować, kto powinien ponieść dodatkowy koszt wyrównania przy kolejnych zakupach, co zapewnia sprawiedliwość w dłuższej perspektywie.

W rezultacie, w sytuacjach, gdy nie można równo podzielić kosztu produktu pomiędzy osoby, mechanizm wyrównania umożliwia optymalne rozdzielenie kosztów, eliminując problemy z niezgodnościami czy rozbieżnościami w sumach. Jednocześnie wskazuje osobę, która powinna ponieść koszt ewentualnego wyrównania.

Podstawowe definicje i notacje

Zdefiniowano następujące oznaczenia:

- P oznacza zbiór produktów, gdzie każdy produkt $p \in P$ charakteryzuje:
  price(p) - całkowita cena produktu,

  maxQuantity(p) - maksymalna liczba części, na którą produkt może zostać podzielony,

  compensationPrice(p) - kwota do wyrównania dla produktu (inicjalizowana jako price(p)).

- C oznacza zbiór osób, gdzie każda osoba $c \in C$ charakteryzuje:

  compensationAmount(c): suma wartości wyrównań, które poniosła osoba c,

  totalAmount(c): całkowita kwota nakładana na osobę.

- A oznacza zbiór przypisań, $a \in A$, gdzie każde przypisanie odnosi się do konkretnej osoby i produktu. Dla przypisania a:

  quantity(a): wartość liczby części produktu,

  partOfPrice(a): cena przypisanej części,

  compensation(a): wartość logiczna wskazująca, czy przypisanie obejmuje wyrównanie,

  productId(a): identyfikator produktu powiązanego z przypisaniem a,

  personId(a): identyfikator osoby powiązanej z przypisaniem a.

Operacje algorytmu:

1. **Podział ceny na części**

Cena części produktu partOfPrice(a) dla przypisania a obliczana jest na podstawie liczbie części przypisanej do osoby i maksymalnej liczbie podzielności dla produktu (wzór 2.1).

$$partOfPrice(a) = \lfloor \frac{quantity(a)}{maxQuantity(p)} \times price(p) \times 100 \rfloor \div 100 \quad (2.1)$$

gdzie:

- p = productId(a): Produkt powiązany z przypisaniem a,
- $\lfloor x \rfloor$: Funkcja zaokrąglająca w dół wartość x do najbliższej liczby całkowitej.

2. **Aktualizacja pozostałej kwoty wyrównania**

Po obliczeniu części ceny, obliczana zostaje kwota pozostała do wyrównania dla produktu p (wzór 2.2).

$$compensationPrice(p) = price(p) - \sum_{a \in A_p} partOfPrice(a) \quad (2.2)$$

gdzie:

- compensationPrice(p): Kwota do wyrównania dla produktu p po uwzględnieniu przypisanych części,
- price(p): Całkowita cena produktu p,
- $A_p = \{ a \in A : productId(a) = p \}$ — zbiór przypisań dla produktu p,
- partOfPrice(a) — cena przypisanej części dla przypisania a,
- $\sum_{A_p}$: Suma wartości partOfPrice(a) dla wszystkich przypisań A_p.

Wartość **compensationPrice(*p*)** oznacza kwotę, którą jeszcze należy wyrównać dla produktu p, biorąc pod uwagę już przypisane części ceny.

3. Nałożenie wartości wyrównania na osobę

1. Znajdź osobę z najmniejszym compensationAmount(c)

Wartość wyrównania jest nakładana na osobę, która ma najmniejszą wartość compensationAmount(c) spośród osób powiązanych z przypisaniami dla danego produktu p (wzór 2.3).

$$c_{min} = min\{ compensationAmount(c) \mid c \in C_p \} \quad (2.3)$$

gdzie:

- $C_p = \{ personId(a) : a \in A_p \}$ — jest zbiorem osób powiązanych z przypisaniami A_p,
- c_{min}: Osoba z najmniejszą wartością compensationAmount(c) w zbiorze C_p,
- min: Operacja wyboru minimalnej wartości.

2. Ustaw flagę compensation(a) w przypisaniu (wzór 2.4).

Dla przypisania a, gdzie $personId(a) = c_{min}$.

$$compensation(a) = true \quad (2.4)$$

3. Zaktualizuj wartość wyrównania dla osoby (wzór 2.5).

$$compensationAmount(c_{min}) \leftarrow compensationAmount(c_{min}) + compensationPrice(p) \quad (2.5)$$

gdzie:

- ←: Oznacza operację aktualizacji wartości.

4. Dodanie części ceny do wszystkich uczestników związanych z produktem

Dla każdego $c \in C_p$ zostaje dodana część ceny produktu (wzór 2.6).

$$totalAmount(c) \leftarrow totalAmount(c) + partOfPrice(a) \quad (2.6)$$

gdzie:

- totalAmount(c) oznacza całkowitą kwotę przypisaną do osoby c,
- partOfPrice(a) to koszt przypisanej części produktu, który dodawany jest do wszystkich uczestników.

5. **Aktualizacja nieuregulowanej kwoty u osoby która ponosi wyrównanie**

Jeśli przypis osoby z produktem posiada flagę $compensation(a) = true$, dodana zostaje wartość $compensationPrice(p)$ do całkowitych kosztów osoby c_{min} (wzór 2.7).

$$totalAmount(c_{min}) \leftarrow totalAmount(c_{min}) + compensationPrice(p) \quad (2.7)$$

Kroki algorytmu:

1. Podział produktu na mniejsze części.
2. Aktualizacja wartości wyrównania w produkcie.
3. Znalezienie osoby z najmniejszą wartością wyrównania przypisaną do konta.
4. Nałożenie opłaty wyrównawczej na osobę związaną z produktem.
5. Rozdzielenie kosztu przypisanej części na wszystkich uczestników.
6. Aktualizacja nieuregulowanej kwoty u osoby która ponosi wyrównanie.

2.1.2. Algorytm obliczania ceny końcowej produktu

Algorytm obliczania ceny końcowej produktu uwzględnia kilka kluczowych elementów, takich jak cena pierwotna, liczba zakupionych sztuk, dodatkowa opłata oraz ewentualny rabat. Celem algorytmu jest precyzyjne obliczenie końcowej ceny, którą użytkownik zapłaci za produkt, uwzględniając wszystkie zmienne.

Podstawowe definicje i notacje:

Zdefiniowano następujące oznaczenia:

- price - cena pierwotna produktu,
- basePrice - cena pierwotna pomnożona przez liczbę sztuk,
- additionalPrice - dodatkowa opłata związana z produktem,
- totalPrice - cena końcowa,
- discountPercentage - rabat procentowy,
- discountAmount - kwota rabatu,
- priceAfterDiscount - cena produktu po procesie rabatowania,
- purchasedQuantity - ilość zakupionych sztuk.

Operacje algorytmu:

1. **Obliczenie ceny bazowej**

Cena bazowa produktu jest obliczana przez pomnożenie ceny pierwotnej przez liczbę zakupionych sztuk (wzór 2.8).

$$basePrice = price \times purchasedQuantity \quad (2.8)$$

2. **Obliczenie wartości rabatu**

Jeśli rabat jest stosowany, jego wartość jest obliczana jako procent od ceny bazowej (wzór 2.9).

$$discountAmount = basePrice \times (\frac{discountPercentage}{100}) \quad (2.9)$$

3. **Obliczanie ceny po rabacie**

Cena po rabacie to cena bazowa po odjęciu wartości rabatu (wzór 2.10).

$$priceAfterDiscount = basePrice - discountAmount \quad (2.10)$$

4. **Obliczanie ceny końcowej**

Cena końcowa produktu obliczana jest przez dodanie dodatkowej opłaty do ceny po rabacie (wzór 2.11).

$$totalPrice = priceAfterDiscount + additionalPrice \quad (2.11)$$

5. **Zaokrąglenie ceny końcowej**

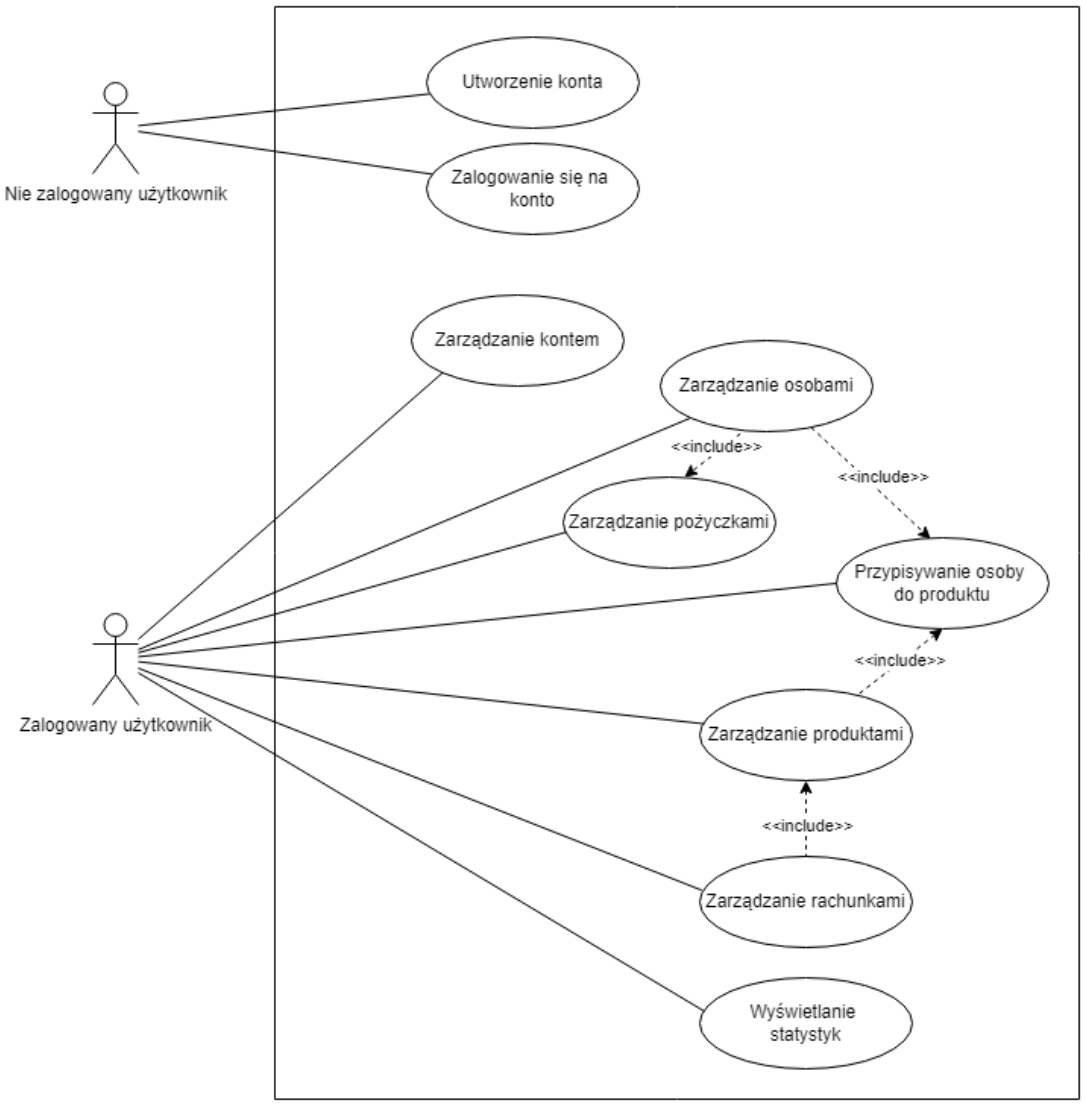
Aby wynik był precyzyjny i odpowiedni do wyświetlenia użytkownikowi cena końcowa jest zaokrąglana do dwóch miejsc po przecinku (wzór 2.12).

$$totalPrice = Math.Round(totalPrice, 2) \quad (2.12)$$

2.1.3. Diagram przypadków użycia

Diagram przypadków użycia (*Use Case Diagram*) należy do zestawu diagramów systemu *UML* (*Unified Modeling Language*). *UML* to półformalny język modelowania, który służy do wizualizacji systemów informatycznych i innych systemów złożonych. Diagram przypadków użycia pozwala na przedstawienie interakcji między aktorami a systemem, co znacząco upraszcza proces projektowania [17].

Diagram przypadków użycia dla systemu *DivvyUp* (rys. 2.1) przedstawia działania podejmowane przez dwóch aktorów: „Niezalogowanego użytkownika" oraz „*Zalogowanego użytkownika*". Diagram obrazuje relacje między przypadkami użycia, np. aby możliwe było zarządzanie produktami, użytkownik musi najpierw utworzyć rachunek, co jest realizowane w ramach przypadku użycia dotyczącego zarządzania rachunkami. Diagram w przejrzysty sposób przedstawia wzajemne powiązania i zależności pomiędzy funkcjonalnościami systemu, umożliwiając łatwiejsze zrozumienie logiki działania systemu.



Rys. 2.1. Diagram przypadków użycia dla systemu DivvyUp.

2.2. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne to aspekty, które nie odnoszą się bezpośrednio do konkretnych funkcjonalności systemu, lecz mają wpływ na całokształt jego działania oraz odbiór przez użytkownika. Ich realizacja wpływa na jakość korzystania z aplikacji, bezpieczeństwo oraz przyszłe możliwości rozwoju.

2.2.1. Bezpieczeństwo

Priorytetem w aplikacjach, które umożliwiają użytkownikom tworzenie kont, jest spełnienie powszechnych norm bezpieczeństwa. Taki zabieg sprawia, że system jest mniej narażony na ataki oraz wykorzystywanie jego podatności. Opracowany system zapewnia to bezpieczeństwo poprzez system tokenów JWT (*JSON Web Token*). Jest to popularna metoda autoryzacji, pozwalająca na bezpieczne uwierzytelnienie użytkowników bez konieczności przechowywania stanu sesji na serwerze.

2.2.2. Intuicyjny interfejs użytkownika

Interfejs jest formą komunikacji pomiędzy aplikacją a użytkownikiem. Im bardziej czytelny i przemyślany będzie jego układ, tym korzystanie z aplikacji będzie bardziej intuicyjny dla użytkownika. Operacje wykonywane przez użytkownika powinny być płynne, a aplikacja nie powinna wzbudzać żadnych podejrzeń, np. z powodu nieoczekiwanych zachowań.

Każdy element interfejsu powinien być rozmieszczony w logiczny sposób – nawigacja, główna zawartość oraz stopka muszą znaleźć się na swoich miejscach. Użytkownik powinien móc korzystać z aplikacji intuicyjnie, nawet bez wcześniejszej znajomości systemu. Ważne jest także, aby działania w aplikacji były spójne, co poprawia ogólne doświadczenie użytkownika.

2.2.3. Wydajność i optymalizacja

Aplikacja została zoptymalizowana pod kątem platformy, na której będzie działać, aby zapewnić maksymalną wydajność, co oznacza płynne działanie, szybkie reakcje na działania użytkownika oraz przetwarzanie wyłącznie tych danych, które są w danej chwili potrzebne. Kluczowe dla utrzymania odpowiedniego poziomu wydajności jest skrócenie czasów ładowania oraz skuteczne zarządzanie zasobami aplikacji aby zminimalizować zużycie pamięci i mocy obliczeniowej, co przekłada się na lepsze doświadczenie użytkownika.

2.2.4. Zarządzanie strukturą projektu i jakością kodu

Poprawne zarządzanie kodem oraz jego struktura są kluczowe dla ułatwienia rozwoju i rozbudowy aplikacji, dlatego projekt powinien być tworzony w zgodzie z popularnymi standardami programistycznymi, co pozwala na zachowanie przejrzystości oraz łatwość wprowadzania przyszłych modyfikacji. Dobrze zorganizowane pliki i katalogi, a także pisanie kodu w sposób umożliwiający elastyczne korzystanie z istniejących rozwiązań, znacząco wspierają efektywność pracy nad projektem.

Ważne jest zachowanie zasad takich jak oddzielenie logiki biznesowej od elementów wizualnych, rozdzielenie aplikacji na niezależne moduły, czytelna dokumentacja oraz odpowiednie komentowanie bardziej złożonych fragmentów kodu, a także korzystanie z systemu kontroli wersji, który ułatwia śledzenie zmian wraz z możliwością zarządzania poprzednimi wersjami systemu.

3. Opis wykorzystanych technologii

W tym rozdziale zaprezentowano zestawienie technologii, wykorzystanych przy realizacji projektu, mających na celu zapewnienie integralności pomiędzy interfejsem graficznym a warstwą odpowiedzialną za przechowywanie i przetwarzanie danych.

3.1. C#

Język programowania *C#* stanowi podstawę aplikacji. Jest to obiektowy język programowania opracowany przez firmę Microsoft, który zadebiutował w 2002 roku jako część platformy *.NET* [1]. C# to język ogólnego przeznaczenia z bezpiecznym systemem typów, którego jednym z głównych założeń jest umożliwienie szybkiego i efektywnego tworzenia programów. Implementuje szeroką gamę technik programowania obiektowego, takich jak hermetyzacja, dziedziczenie i polimorfizm, co czyni go nowoczesnym i elastycznym narzędziem. Charakteryzuje się prostą, przejrzystą składnią oraz wsparciem dla programowania obiektowego, funkcyjnego i asynchronicznego, dzięki czemu jest wszechstronnym językiem, doskonale nadającym się do pracy w różnych środowiskach programistycznych [2].

3.2. Blazor ASP.NET Core

Blazor to innowacyjny framework umożliwiający tworzenie interaktywnych aplikacji webowych z użyciem C#. Jego nazwa składa się z mutacji dwóch słów: *Browser* i *Razor*, co w wolnym tłumaczeniu oznacza "przeglądarka" i "brzytwa", nazwa *Razor* odnosi się do silnika generującego widoki HTML w technologii *.NET*. W efekcie *Blazor* umożliwia renderowanie widoków *Razor* bezpośrednio po stronie klienta, eliminuje to konieczność przetwarzania ich na serwerze przed wysłaniem ich do przeglądarki. Taki mechanizm sprawia że możliwe jest tworzenie dynamicznych i responsywnych aplikacjach jednostronicowych SPA (*Single Page Application*) przy użyciu C# po stronie serwera oraz klienta [3]. Najczęściej Blazor jest wykorzystywany do tworzenia aplikacji webowych oferujących szybki dostęp do funkcji, jednocześnie zachowując wysoką responsywność interfejsu. Framework ten posiada również szeroką bibliotekę komponentów o różnorodnym zastosowaniu od zwykłych przycisków po złożone siatki danych i wykresy, co usprawnia i przyspiesza proces tworzenia aplikacji.

3.3. HTTP (Hypertext Transfer Protocol)

HTTP (*Hypertext Transfer Protocol*) to protokół komunikacyjny używany w sieci internetowej, oparty na protokole *TCP/IP*. Umożliwia wymianę informacji między klientem a serwerem. Najczęstszym zastosowaniem tej technologii jest przesyłanie zasobów poprzez definiowanie zasad wysyłania żądań oraz odbioru odpowiedzi [4].

Żądanie *HTTP* zawiera metodę określającą rodzaj żądania, adres *URL* (*Uniform Resource Locator*) wskazujący lokalizację zasobu, do którego jest skierowane, oraz nagłówki zawierające dodatkowe informacje przesyłane wraz z żądaniem. Protokół wspiera metody które są odpowiedzialny za szczególne modyfikacje danych.

Tabela 3.1. Metody żądań HTTP

Metoda	**Zachowanie**
POST	Wysyłanie danych
PUT	Dodawanie danych
PATCH	Częściowa aktualizacja danych
GET	Pobieranie danych
DELETE	Usunięcie danych

Odpowiedź *HTTP* zawiera kod stanu, który informuje o wyniku wysłanego żądania, treść, czyli dane zwracane w odpowiedzi, oraz nagłówek, który zawiera dodatkowe informacje dotyczące odpowiedzi. Kody odpowiedzi zawierające informacje o rezultacie żądania.

Tabela 3.2. Kody odpowiedzi HTTP

Wartość kodu odpowiedzi	**Znaczenie**
1xx	Kody informacyjne
2xx	Kody powodzenia
3xx	Kody przekierowań
4xx	Kody błędów klienta
5xx	Kody błędów serwera HTTP

3.4. JSON

JSON (*JavaScript Object Notation*) to lekki format tekstowy służący do przechowywania i wymiany danych między serwerem a klientem. Zyskał popularność dzięki swojej prostocie i elastyczności, co sprawiło, że jest szeroko wykorzystywany w aplikacjach internetowych. *JSON* charakteryzuje się łatwością odczytu przez człowieka oraz prostotą interpretacji przez maszyny. Struktura *JSON* opiera się na dwóch elementach: obiektach i tablicach. Obiekt zawiera pary klucz-wartość, przy czym wartością w *JSON* mogą być liczby, łańcuchy znaków, wartości logiczne, a także obiekty. Tablica to uporządkowana lista wartości, które mogą być różnego typu [5].

3.5. Entity Framework Core

Entity Framework Core to zaawansowany *ORM* (*Object-Relational Mapping*), który umożliwia komunikację z bazą danych w sposób obiektowy. Dzięki *EF Core*, zarządzanie danymi staje się bardzo intuicyjne, co pozwala na szybsze tworzenie z zachowaniem prostszym utrzymaniem aplikacji, a także ułatwia implementację skomplikowanych operacji na danych [6].

3.6. .NET MAUI (Multi-platform App UI)

.NET MAUI to framework, który umożliwia tworzenie aplikacji na wiele platform przy użyciu jednego kodu źródłowego. Jest to rozwinięcie platformy *Xamarin*, dostosowane współczesnych potrzeb tworzenia aplikacji mobilnych i desktopowych. Dzięki *.NET MAUI*, deweloperzy mogą tworzyć aplikacje, które działają zarówno na systemach Android, iOS, jak i na komputerach z systemem Windows oraz macOS, co znacząco przyspiesza proces rozwoju i obniża koszty związane z utrzymaniem kilku oddzielnych aplikacji dla różnych platform [7].

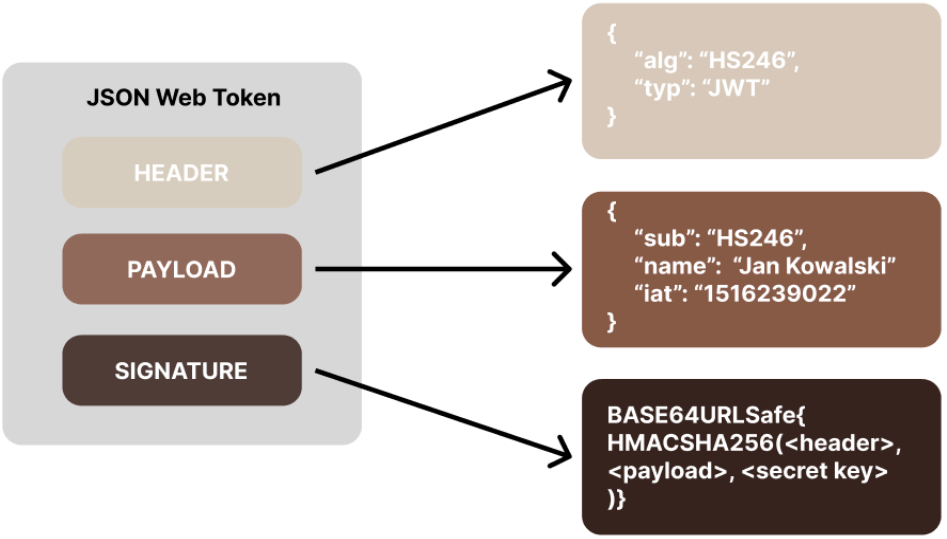
3.7. Radzen Blazor UI

Radzen Blazor UI (*Radzen Blazor User Interface*) to biblioteka komponentów interfejsu użytkownika, umożliwiająca szybkie tworzenie estetycznych i funkcjonalnych interfejsów dla aplikacji. Biblioteka komponentów oferuje szeroką gamę gotowych elementów *UI*, takich jak tabele, formularze, przyciski, wykresy, nawigacje, a także bardziej zaawansowane elementy jak kalendarze czy edytory tekstu, które można łatwo dostosować do specyficznych wymagań aplikacji. Zapewniając jednocześnie responsywność i atrakcyjność wizualną aplikacji [8].

3.8. JSON Web Token

JSON Web Token to standard tokenów wykorzystywany do autoryzacji w aplikacji. Umożliwia bezpieczne przesyłanie informacji pomiędzy użytkownikami a serwerem, a także zapewnia, że dane o sesji są przechowywane w sposób bezpieczny i nie wymagają dodatkowego stanu na serwerze [9].

Token składa się z trzech głównych części (rys. 3.1): nagłówka (*header*), który zawiera metadane dotyczące algorytmu szyfrowania, ładunku (*payload*), który przechowuje dane, oraz podpisu (*signature*), który służy do weryfikacji integralności tokenu.



Rys. 3.1. Struktura i składniki JSON Web Token.

3.9. PostgreSQL

PostgreSQL jest to zaawansowany system zarządzania relacyjnymi bazami danych *RDBMS* (*Relational Database Management System*). Umożliwia przechowywanie i efektywne zarządzanie dużymi zbiorami danych. Charakteryzuje się wysoką skalowalnością, elastycznością i pełnym wsparciem dla transakcji, zapewniając integralność danych oraz ich bezpieczeństwo. Dzięki zastosowaniu zaawansowanych mechanizmów, takich jak *ACID* (Atomowość, Spójność, Izolacja, Trwałość). *PostgreSQL* gwarantuje, że operacje na danych są przeprowadzane w sposób niezawodny i spójny. W projekcie *PostgreSQL* pełni rolę centralnej bazy danych, zapewniając solidne przechowywanie informacji, łatwe rozszerzanie aplikacji oraz efektywne zarządzanie danymi użytkowników [10].

3.10. xUnit

xUnit jest popularnym frameworkiem do testów jednostkowych oraz integracyjnych, skierowanym dla technologii *.NET*. Wspiera on automatyczne testowanie aplikacji, poprawiając jej jakość i stabilność. Umożliwia tworzenie testów jednostkowych za pomocą atrybutu *[Fact]*, a dla testów z wieloma danymi – *[Theory]*, co pozwala na przeprowadzenie testów dla różnych scenariuszy. *xUnit* automatycznie zarządza zasobami testów, zwalniając je po każdym teście, co minimalizuje zależności i zapewnia niezależność testów. W projekcie *xUnit* będzie wykorzystywany do weryfikacji poprawności kluczowych funkcji aplikacji, zapewniając zgodność z założeniami biznesowymi [11].

3.11. Git

Git to rozproszony system kontroli wersji, który przechowuje pełną historię projektu lokalnie, umożliwiając pracę nad kodem zarówno w trybie lokalnych, jak i synchronizację z centralnym repozytorium na serwerze. Dzięki temu programiści mogą bezpiecznie i efektywnie współpracować w zespołach, wprowadzając zmiany w różnych częściach aplikacji jednocześnie. *Git* pozwala na odwołanie się do wcześniejszych wersji projektu i przywrócenie ich w razie potrzeby, co zapewnia elastyczność i zabezpiecza przed utratą danych. *Git* wykorzystuje mechanizm gałęzi, który umożliwia tworzenie odizolowanych środowisk do eksperymentowania z nowymi funkcjonalnościami lub poprawek, bez ryzyka destabilizowania głównej wersji projektu. Stabilną wersję aplikacji można utrzymywać na gałęzi produkcyjnej, podczas gdy prace deweloperskie mogą toczyć się na osobnych gałęziach, które później są łączone (tzw. *merge*) z główną gałęzią projektu [12].

3.12. Swagger

Swagger to popularne narzędzie służące do dokumentacji, testowania oraz zarządzania interfejsami *API*. Dzięki integracji z projektem, *Swagger* generuje interaktywną dokumentację *API*, która pozwala na łatwe przeglądanie dostępnych punktów końcowych, metod *HTTP*, parametrów wejściowych oraz formatów odpowiedzi. *Swagger* umożliwia testowanie *API* bezpośrednio z poziomu dokumentacji, co pozwala na szybkie sprawdzenie działania poszczególnych punktów końcowych oraz diagnozowanie ewentualnych problemów w komunikacji z serwerem [13].

4. Architektura systemu

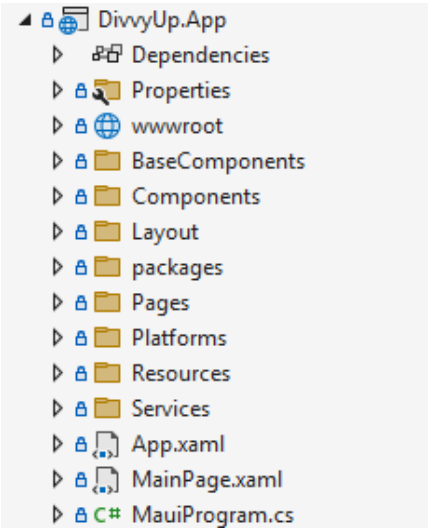
System *DivvyUp* składa się z czterech projektów, które wspólnie tworzą spójną architekturę aplikacji. Każdy z nich pełni określoną funkcję, co umożliwia klarowną organizację kodu, wyraźne rozdzielenie odpowiedzialności oraz ułatwienie w rozwoju projektu. Solucja całego systemu zawierająca pomniejsze projekty (rys. 4.1).

Rys. 4.1. Architektura systemu DivvyUp.

4.1. DivvyUp.App - klient

DivvyUp.App to klient (rys. 4.2), który zapewnia użytkownikowi graficzny interfejs do interakcji z systemem. Korzystając z tej aplikacji, użytkownik może bezpośrednio manipulować logiką aplikacji poprzez przeglądanie i zarządzanie danymi.

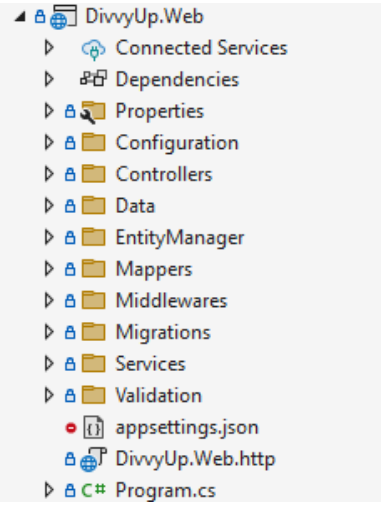
Projekt składa się głównie z komponentów, które tworzą spójną całość i są umiejscowione na stronach aplikacji. Logika działania znajduje się w poszczególnych komponentach, jak i w serwisach. Serwisy te dzielą się na te związane bezpośrednio z klientem oraz na serwisy *HTTP* umożliwiające komunikację z serwerem. Projekt został opracowana z wykorzystaniem platformy *.NET MAUI*.



Rys. 4.2. Architektura projektu DivvyUp.App.

4.2. DivvyUp.Web - aplikacja serwerowa

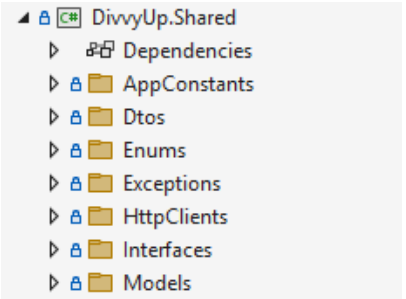
DivvyUp.Web pełni rolę warstwy serwerowej w systemie *DivvyUp* (rys. 4.3). Zawiera logikę biznesową aplikacji serwerowej, która w połączeniu z komunikacją z bazą danych umożliwia manipulowanie danymi w określony sposób przez użytkownika. Projekt ten charakteryzuje się bogatą architekturą, składającą się z maperów, middlewarów, autoryzacji, a także serwisów odpowiedzialnych za walidację danych, logikę kontrolerów, na które kierowane są zapytania, oraz serwisów odpowiedzialnych za odświeżanie danych po wykonanej transakcji. Dzięki temu każdy obszar jest kompletny i realizuje powierzone zadanie ze szczególną starannością.



Rys. 4.3. Architektura projektu DivvyUp.Web.

4.3. DivvyUp.Shared - warstwa współdzielona

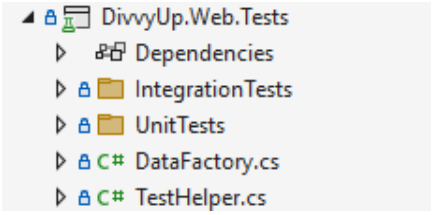
DivvyUp.Shared to projekt, którego celem jest współdzielenie danych i elementów między różnymi projektami w systemie *DivvyUp* (rys. 4.4). Projekt ten upraszcza operowanie danymi, ponieważ umożliwia centralne zarządzanie zmianami związanymi ze stałymi elementami, takimi jak modele bazy danych, modele transferu *DTO*, enumy, obsługa zmodyfikowanego klienta *HTTP* oraz dedykowany typ wyjątku.



Rys. 4.4. Architektura projektu DivvyUp.Shared.

4.4. DivvyUp.Web.Tests - testy warstwy serwerowej

DivvyUp.Web.Tests to segment projektu, który koncentruje się na testowaniu oprogramowania związanego z serwerem (rys. 4.5). Testy przeprowadzane w tym projekcie mają na celu zwiększenie niezawodności kodu oraz jego weryfikację w różnych scenariuszach. Składa się on z testów jednostkowych, które sprawdzają pojedyncze zachowania poszczególnych funkcji, a także testów integracyjnych, które weryfikują, jak system zachowuje się po wykonaniu określonej akcji i czy wszystkie rezultaty są zgodne z oczekiwaniami.

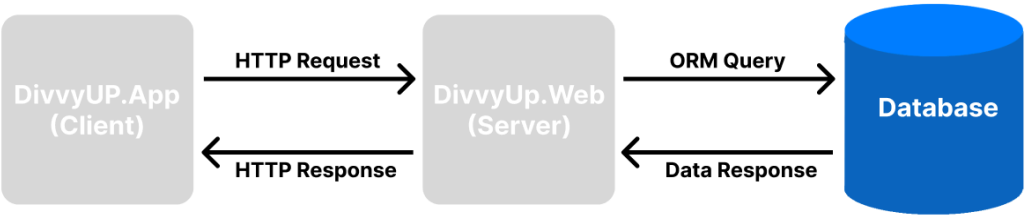


Rys. 4.5. Architektura projektu DivvyUp.Web.Tests.

4.5. Przepływ danych pomiędzy modułami systemu

W systemie *DivvyUp* komunikacja pomiędzy aplikacją kliencką a serwerową odbywa się za pośrednictwem protokołu *HTTP* (rys. 4.6). Proces rozpoczyna się, gdy klient wysyła żądanie *HTTP* do serwera, zawierające informacje potrzebne do realizacji konkretnej operacji. Serwer, po odebraniu żądania, przetwarza je i w razie potrzeby komunikuje się z bazą danych, korzystając z zapytań *ORM* (*Object-Relational Mapping*). Zapytania te umożliwiają zapis lub odczyt danych w bazie. Następnie baza danych zwraca odpowiedź zawierającą wymagane informacje. Otrzymane dane są przez serwer przetwarzane formułowane w odpowiedź *HTTP*, która jest następnie przesyłana z powrotem do klienta.

Wszystkie dane wymieniane między klientem a serwerem są przesyłane w formacie *JSON*, co zapewnia lekkość i czytelność przesyłanych informacji. Taki uporządkowany proces komunikacji umożliwia płynne i niezawodne przekazywanie danych między użytkownikiem (klientem) a bazą danych, co jest kluczowe dla prawidłowego działania systemu.



Rys. 4.6. Przepływ danych pomiędzy modułami systemu.

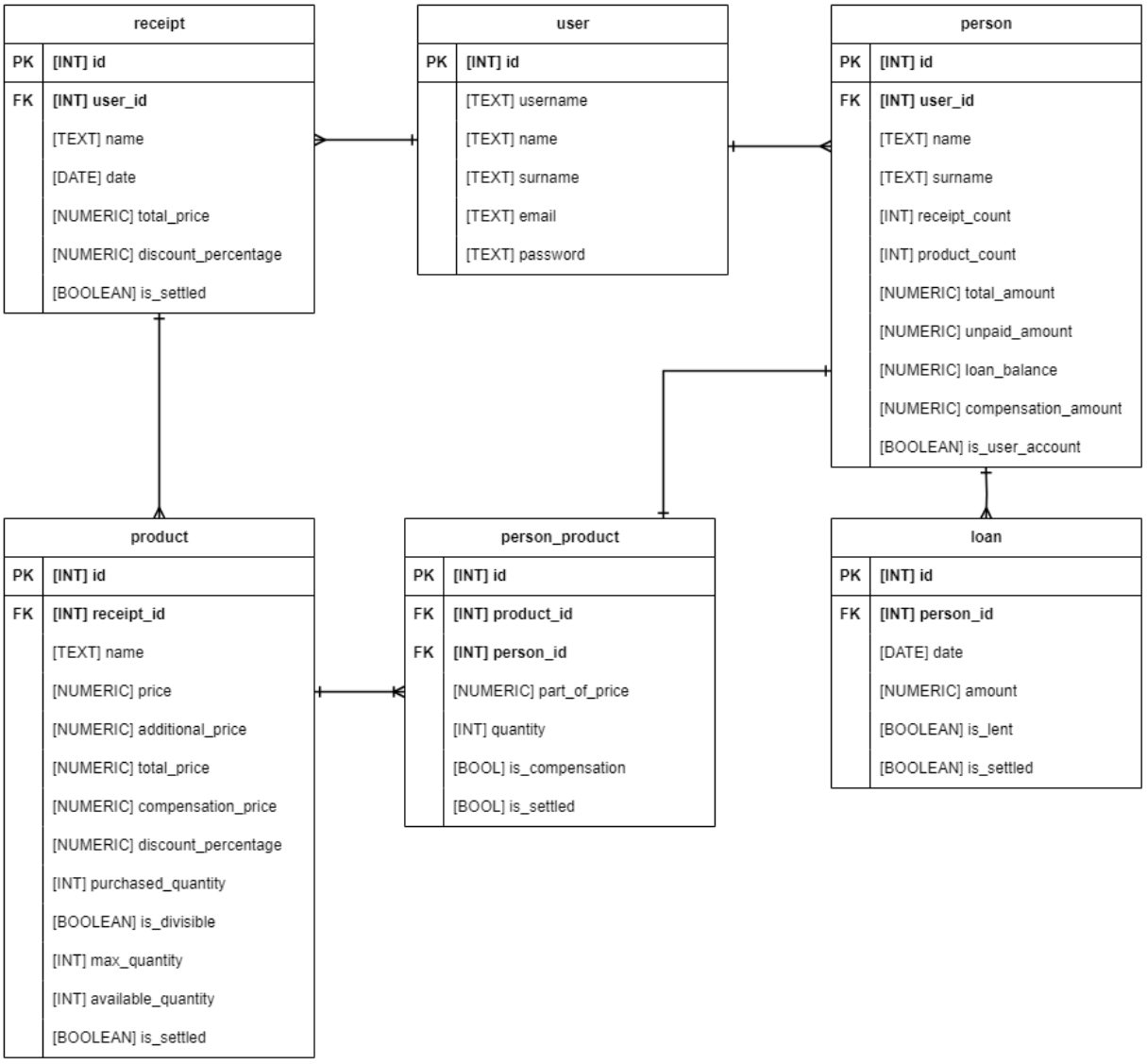
4.6. Diagram ERD

Diagram ERD (*Entity-Relationship Diagram*) to graficzne przedstawienie relacji pomiędzy tabelami w bazie danych. W przypadku systemu DivvyUp (rys. 4.7) diagram składa się z sześciu tabel, z których każda zawiera unikalny zestaw danych.

Tabela użytkowników zawiera dane o użytkowniku m. in. nazwę, e-mail a także zaszyfrowane hasło.

Tabele powiązane z procesem rozliczania zawierają kolumne *is_settled*, które wskazuje, czy dany rachunek, produkt, pożyczka lub przypisanie osoby do produktu zostało już uregulowane.

Największą tabelą jest tabela produktów, która zawiera informacje o przypisaniu do rachunku, nazwie, cenie, możliwości podziału, kwocie wyrównania, dodatkowych opłatach, rabacie i cenie końcowej.



Rys. 4.7. Diagram ERD dla systemu DivvyUp.

5. Rozwiązania implementacyjne

W niniejszym rozdziale skupiono się na szczegółach technicznych oraz na sposobie implementacji poszczególnych komponentów systemu *DivvyUp*. Omówione zostaną kluczowe mechanizmy, podejścia oraz rozwiązania zastosowane w systemie.

5.1. Implementacja aplikacji użytkownika

Aplikacja użytkownika przedstawia klienta aplikacji oraz warstwę wizualną projektu. Technologie i rozwiązania implementacyjne będą miały wpływ na manewrowanie po systemie użytkownika, dlatego powinny być niezawodne oraz wykonane ze szczególną starannością, aby spełniać wymagania użytkowników. W tym rozdziale przedstawiono różne fragmenty kodu, które składają się na kluczowe mechanizmy klienta aplikacji.

5.1.1. Konfiguracja aplikacji w pliku MauiProgram.cs

Plik *MauiProgram.cs* odpowiada za konfigurację i inicjalizację technologii *.NET MAUI* (rys. 5.1). W nim definiowane są podstawowe ustawienia aplikacji, rejestrowane usługi wraz z integrowaniem komponentów zewnętrznych, między innymi biblioteki i frameworki.

Metoda *CreateMauiApp* tworzy i zwraca główny obiekt *MauiApp*, jest on konieczny do uruchomienia aplikacji. Konfiguracja obejmuje rejestrację usług przy użyciu wzorca *Dependency Injection*, który umożliwia wstrzykiwanie zależności do komponentów aplikacji.

W ramach tej rejestracji dodawane są usługi odpowiedzialne za komunikację z serwerem oraz komponenty interfejsu użytkownika umożliwiające interakcję z aplikacją. Usługi serwera zarządzają wymianą danych z serwerem, podczas gdy usługi GUI odpowiadają za zarządzanie elementami wizualnymi oraz interaktywnością aplikacji.

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();

        builder.Services.RegisterHttpClient();
        builder.Services.RegisterAppServices();

        #if DEBUG
        builder.Services.AddBlazorWebViewDeveloperTools();
        builder.Logging.AddDebug();
        #endif

        return builder.Build();
    }
}
```

Rys. 5.1. Fragment pliku MauiProgram.cs.

5.1.2. Rejestracja klienta HTTP w kliencie

Rejestracja *HTTP* klienta (rys. 5.2) która jest wywoływana w CreateMauiApp obejmuje konfigurację niestandardowego klienta *DHttpClient* do kontenera usług. Wykorzystuje *IHttpClientFactory* która tworzy instancje *HttpClient* z ustawionym adresem bazowym na "*http://localhost:5000/*". Na końcu jest wywoływana metoda *AddHttpClient*(), która dodaje domyślną konfigurację klienta *HTTP*.

```
public static class HttpClientRegistration
{
    public static void RegisterHttpClient(this IServiceCollection services)
    {
        services.AddSingleton<DHttpClient>(sp =>
        {
            var httpClientFactory = sp.GetRequiredService<IHttpClientFactory>();
            var httpClient = httpClientFactory.CreateClient();
            httpClient.BaseAddress = new Uri("http://localhost:5000");
            return new DHttpClient(httpClient);
        });
        services.AddHttpClient();
    }
}
```

Rys. 5.2. Fragment rejestracji klienta HTTP.

5.1.3. Zarządzanie sesją użytkownika

Aplikacja, która posiada mechanizm autoryzacji użytkownika, powinna również zachowywać sesję logowania podczas ponownego uruchomienia. Logika odpowiadająca za zapamiętywanie sesji jest realizowana poprzez metodę *SetUser* w głównym widoku aplikacji przy wykorzystaniu serwisu *UserStateProvider* (rys. 5.3). Kod odczytuje token z serwisu, zapisuje jego zawartość do zmiennej, a następnie usuwa token z serwisu. Następnie sprawdza, czy token jest pusty lub czy jest nadal ważny. Jeżeli token jest ważny, zostanie ponownie ustawiony.

```
private async Task SetUser()
{
    var token = await UserStateProvider.GetTokenAsync();
    await UserStateProvider.ClearTokenAsync();

    if (!string.IsNullOrEmpty(token))
    {
        try
        {
            bool isValid = await UserService.ValidToken(token);
            if (isValid)
            {
                await UserStateProvider.SetTokenAsync(token);
            }
        }
        catch (Exception ex)
        {
            System.Diagnostics.Debug.Print(ex.Message);
        }
    }
    StateHasChanged();
}
```

Rys. 5.3. Mechanizm ustawiania stanu użytkownika podczas uruchomienia aplikacji.

Serwis *UserStateProvider*, który odpowiada za przechowywanie stanu użytkownika wraz z tokenem, wykorzystuje mechanizm *LocalStorage*. Dzięki temu token jest przechowywany niezależnie od aktualnego stanu aplikacji, także po jej zamknięciu (rys. 5.4).

```
private const string UserTokenKey = "UserToken";
public async Task<string?> GetTokenAsync()
{
    return await _localStorageService.GetItemAsync<string>(UserTokenKey);
}
```

Rys. 5.4. Fragment serwisu UserStateProvider - pobieranie tokenu.

Proces zapisu tokena (rys. 5.5) przebiega następująco: najpierw nowy token zostaje zapisany w pamięci, a następnie ustawiony jako nagłówek autoryzacji w kliencie *HTTP*. Dodatkowo zmieniana jest wartość flagi *AfterValidation* na "*true*". Flaga ta jest wykorzystywana w innych funkcjach i wskazuje, że proces pierwszej walidacji został zakończony. Na końcu system jest powiadamiany o zmianie stanu użytkownika.

```
public async Task SetTokenAsync(string token)
{
    await _localStorageService.SetItemAsync(UserTokenKey, token);
    UpdateHttpClientHeader(token);
    AfterValidation = true;
    NotifyUserStateChanged();
}
```

Rys. 5.5. Fragment serwisu UserStateProvider - zapis tokenu.

5.1.4. Serwis HTTP aplikacji klienta

Klient nawiązuje połączenie z serwerem, wykorzystując mechanizm żądań kierowanych do kontrolerów serwera. Proces ten realizowany jest przez serwisy *HTTP* dostosowane do konkretnych zastosowań. Przykładem takiego serwisu jest moduł odpowiedzialny za obsługę rachunków (rys. 5.6). Serwis *ReceiptHttpService* implementuje interfejs *IReceiptService*, co oznacza, że musi dostarczyć implementację wszystkich metod zadeklarowanych w tym interfejsie. Dzięki temu zapewniona jest zgodność parametrów oraz nazw funkcji z mechanizmami serwera, co umożliwia prawidłową wymianę danych.

Serwis ten posiada dwa obiekty inicjalizowane podczas uruchamiania. Pierwszym z nich jest *_dHttpClient* — spersonalizowany klient *HTTP* z nagłówkiem autoryzacyjnym ustawianym na etapie startu aplikacji. Drugim obiektem jest *_logger*, który umożliwia rejestrowanie nieprawidłowości napotkanych podczas wysyłania żądań lub odbierania odpowiedzi z serwera.

```
public class ReceiptHttpService : IReceiptService
{
    [Inject]
    private DHttpClient _dHttpClient { get; set; }
    private readonly ILogger<ReceiptHttpService> _logger;

    public ReceiptHttpService(DHttpClient dHttpClient, ILogger<ReceiptHttpService> logger)
    {
        _dHttpClient = dHttpClient;
        _logger = logger;
    }
```

Rys. 5.6. Serwis HTTP do obsługi rachunków - konstruktor i zależności.

Serwis wysyła żądania *HTTP* na adres *URL*, który zawiera informacje o tym, do jakiego kontrolera ma się odwołać oraz jaki jest typ żądania. Przykładem jest edycja rachunku, która jest wysyłana na adres *URL* z klasy *ApiRoute*, przechowującej wszystkie dostępne ścieżki *URL* dla żądań. Adres *URL* jest manipulowany poprzez podstawienie identyfikatora rachunku w miejsce odpowiedniego znaku, a następnie żądanie jest wysyłane na serwer z odpowiednią zawartością (rys. 5.7).

Jeśli serwer zgłosi wyjątek, zostanie on przechwycony przez blok *catch*, a następnie ponownie rzucony i przekazany do aplikacji. W zależności od tego, czy jest to wyjątek typu *DException* czy inny rodzaj *Exception*, aplikacja obsłuży go w odpowiedni sposób. Wyjątki typu *DException* będą wyświetlane w panelu notyfikacji, natomiast inne wyjątki zostaną przechwycone przez *logger*, który wyświetli komunikat w konsoli.

```
// ApiRoute.cs
public const string ARG_RECEIPT = "{receiptId}";
public const string EDIT = $"{RECEIPT_ROUTE}/edit/{ARG_RECEIPT}";

// ReceiptHttpService.cs
public async Task Edit(AddEditReceiptDto receipt, int receiptId)
{
    try
    {
        var url = ApiRoute.RECEIPT_ROUTES.EDIT
            .Replace(ApiRoute.ARG_RECEIPT, receiptId.ToString());
        var response = await _dHttpClient.PutAsync(url, receipt);
        await EnsureCorrectResponse(response, "Błąd w czasie edycji rachunku");
    }
    catch (DException ex)
    {
        _logger.LogError(ex, ex.Message);
        throw;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Błąd w czasie edycji rachunku: {Message}", ex.Message);
        throw;
    }
}
```

Rys. 5.7. Metoda serwisu HTTP odpowiedzialna za edytujące rachunku.

5.1.5. Główny widok aplikacji

MainLayout jest to domyślny układ (*layout*) aplikacji *Blazor*, definiuje on wspólny wygląd wraz z strukturą dla wszystkich stron w aplikacji. Pełni on rolę szablonów, które pozwalają zarządzać elementami powtarzalnymi takie jak nawigacyjne menu, stopkę oraz cały układ aplikacji. Struktura komponentu składa się z nagłówka, który zawiera stałe elementy (rys. 5.8). Nawigacja zawiera odnośniki do kluczowych stron. Część główna to element **@Body**, w którym Blazor wstawia zawartość stron na podstawie trasy.

W widoku dostępu do odpowiednich stron aplikacji wykorzystano ponownie serwis *UserStateProvider* który steruje wyświetlaniem elementów na panelu nawigacyjnym, w zależności w jakim stanie znajduje się obecnie użytkownik oraz czy jest zweryfikowany przez aplikację.

```
private bool IsLogged = false;
private async void OnUserStateChanged()
{
    IsLogged = await UserStateProvider.IsLoggedInAsync();
    StateHasChanged();
}
<RadzenSidebar Responsive="true" Style="width: max-content">
        <RadzenPanelMenuItem Text="Rachunki" Path="@AppPath.RECEIPT" Visible="@(IsLogged)" />
</RadzenSidebar>
<RadzenBody>
    @Body
</RadzenBody>
```

Rys. 5.8. Fragment zawartości pliku MainLayout.razor.

5.1.6. Strona produktów

Blazor umożliwia programistom tworzenie stron jako komponentów typu *.razor*. Każda taka strona posiada dekorator **@page**, który definiuje ścieżkę *url*, pod którą dostępny będzie widok. Na stronie produktów (rys. 5.9) w skład ścieżki wchodzi parametr zawierający identyfikator rachunku. Strona produktów składa się z komponentu *ProductsGrid*, który wyświetla widok siatki produktów w obrębie danego rachunku, przekazując również komponentowi identyfikator pobrany ze ścieżki.

```
@page "/receipt/{id:int}/products"
@using DivvyUp_App.Components.Product

<ProductsGrid ReceiptId="@id"/>

@code {
    [Parameter]
    public int id { get; set; }
}
```

Rys. 5.9. Zawartość pliku ProductPage.razor.cs.

5.2. Implementacja aplikacji serwerowej

Aplikacja serwerowa jest łącznikiem pomiędzy klientem a bazą danych, wykonując poszczególne zadania które są wysyłane przez aplikacje. Serwisy i mechanizmy powinny być wykonane w najbardziej optymalny i dopracowany sposób aby zredukować niepotrzebny czas wykonywania zapytań. Ważnym aspektem jest zabezpieczenie całego modułu serwera, począwszy od ustawienia odpowiedniej autoryzacji dla zapytań, co jest kluczowe dla ochrony przed nieautoryzowanym dostępem do zasobów. Dodatkowo, serwer musi być przygotowany na wystąpienie w przypadku błędów, takich jak brak zasobu lub niepowodzenie w wykonaniu akcji. Takie podejście zapewni integralność i bezpieczeństwo aplikacji.

5.2.1 Opis klasy DivvyUpDbContext

DivvyUpDbContext jest klasą kontekstu bazy danych (rys. 5.10), dziedzicząca po klasie *DbContext* (część *Entity Framework*). Umożliwia ona interakcję z bazą danych z poziomu aplikacji w sposób obiektowy. Klasa ta odpowiada za mapowanie modelu aplikacji do struktury bazy danych, co jest niezbędne do wykonywania operacji *CRUD* (*Create, Read, Update, Delete*) na tabelach.

```
public class DivvyUpDBContext : DbContext
{
    public DivvyUpDBContext(DbContextOptions<DivvyUpDBContext> options) : base(options)
    {

    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasDefaultSchema("divvyup");
        modelBuilder.Entity<User>().ToTable("user");
        modelBuilder.Entity<Person>().ToTable("person");
        modelBuilder.Entity<Receipt>().ToTable("receipt");
        modelBuilder.Entity<Loan>().ToTable("loan");
        modelBuilder.Entity<Product>().ToTable("product");
        modelBuilder.Entity<PersonProduct>().ToTable("person_product");
        base.OnModelCreating(modelBuilder);
    }

    public DbSet<User> Users { get; set; }
    public DbSet<Person> Persons { get; set; }
    public DbSet<Receipt> Receipts { get; set; }
    public DbSet<Loan> Loans { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<PersonProduct> PersonProducts { get; set; }
}
```

Rys. 5.10. Struktury klasy DivvyUpDbContext.

5.2.2. Konfiguracja uruchomieniowa aplikacji Web API

Plik *Program.cs* w warstwie serwerowej aplikacji *DivvyUp.Web* odpowiada za konfigurację i uruchomienie aplikacji *ASP.NET Core*. Zawiera ustawienia dotyczące rejestracji usług, konfiguracji bazy danych, middleware oraz mechanizmów autentykacji i autoryzacji (rys. 5.11).

Proces rozpoczyna się od inicjalizacji aplikacji webowej, a następnie do kontenera *DI* (*Dependency Injection*) dodawane są różne usługi, takie jak serwisy, mapery, konfiguracja *Swaggera* oraz ustawienia autentykacji. W zależności od trybu uruchomienia aplikacji, konfigurowana jest baza danych: może to być połączenie z bazą *PostgreSQL* lub użycie bazy danych w pamięci.

Następnie aplikacja jest budowana, w ramach czego dodawane są middleware, w tym odpowiedzialne za obsługę wyjątków po stronie serwera. Dla trybu deweloperskiego włączany jest interfejs graficzny *Swaggera*. Na koniec konfigurowane są mechanizmy autentykacji i autoryzacji, mapowane są kontrolery, a serwer zostaje uruchomiony.

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(new WebApplicationOptions
        {
            // Do wygenerowania pliku exe
            //EnvironmentName = "Development"
        });
        builder.Services.AddServices();
        builder.Services.AddMapper();
        builder.Services.AddSwaggerGenConfiguration();
        builder.Services.AddAuthenticationServices(builder.Configuration);

        if (builder.Environment.IsEnvironment("Testing"))
        {
            builder.Services.AddDbContext<DivvyUpDBContext>(options =>
                options.UseInMemoryDatabase("TestDatabase"));
        }
        else
        {
            builder.Services.AddDbContext<DivvyUpDBContext>(options =>
                options.UseNpgsql(builder.Configuration.GetConnectionString("PostgreSqlConnection")));
        }

        var app = builder.Build();
        app.UseMiddleware<ExceptionMiddleware>();

        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        app.UseAuthentication();
        app.UseAuthorization();
        app.MapControllers();
        app.Run();
    }
}
```

Rys. 5.11. Konfiguracja uruchomieniowa warstwy serwerowej pliku Program.cs.

5.2.3. Prezentacja przykładowego kontrollera wraz z serwisową logiką

Kontroler *ReceiptController* służy do realizacji operacji związanych z modułem rachunków. Dostęp do niego mają wyłącznie zalogowani użytkownicy, ponieważ cała klasa kontrolera jest opatrzona adnotacją (*Authorize*), co wymaga ważnego tokenu *JWT* do wykonania żądania na ten kontroler (rys. 5.12).

```
[Authorize]
[ApiController]
[Route(ApiRoute.RECEIPT_ROUTES.RECEIPT_ROUTE)]
public class ReceiptController : ControllerBase
```

Rys. 5.12. Deklaracja klasy ReceiptController obsługującej zarządzanie paragonami.

Każda metoda w kontrolerze składa się z określonego typu żądania *HTTP* (np. *GET, POST*), ścieżki, parametrów, oraz wywołania odpowiedniej metody w serwisie, która zwraca oczekiwany wynik. Przykładem może być metoda kontrolera *GetReceipt*, która służy do pobierania pojedynczego rachunku na podstawie jego identyfikatora (rys. 5.13).

```
[HttpGet(ApiRoute.RECEIPT_ROUTES.RECEIPT)]
public async Task<ActionResult<ReceiptDto>> GetReceipt([FromRoute] int receiptId)
{
    var receipt = await _receiptService.GetReceipt(receiptId);
    return Ok(receipt);
}
```

Rys. 5.13. Implementacja metody GetReceipt obsługującej żądanie GET dla szczegółów paragonu.

Serwis, do którego odwołuje się kontroler, zawiera zależności oraz obiekty ustawione podczas inicjalizacji (rys. 5.14). Jednym z kluczowych elementów serwisu jest *_dbContext*, który umożliwia interakcję z bazą danych. Kolejnym ważnym składnikiem jest *_mapper*, odpowiedzialny za mapowanie modeli z bazy danych na obiekty *DTO* (*Data Transfer Object*), które są zrozumiałe przez aplikację. Serwis korzysta również z *_validatora*, który odpowiada za walidację danych wejściowych, oraz *_entityManagerService*, realizuje między innymi aktualizację danych po wykonaniu danej transakcji.

```
public class ReceiptService : IReceiptService
{
    private readonly DivvyUpDBContext _dbContext;
    private readonly EntityManagementService _managementService;
    private readonly DValidator _validator;
    private readonly IMapper _mapper;
```

Rys. 5.14. Implementacja klasy ReceiptService obsługującej logikę biznesową dla paragonów.

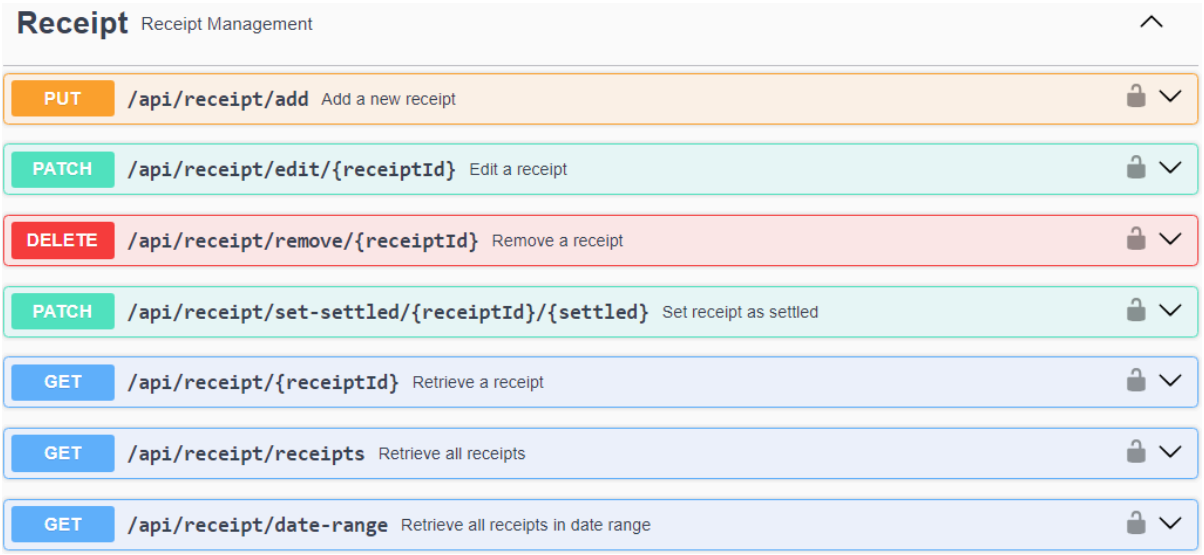
Metoda *GetReceipt* w implementacji serwisu warstwy serwerowej (rys. 5.15) przedstawia funkcję służącą do pobierania rachunku na podstawie jego identyfikatora. Na początku walidowane są dane wejściowe, a następnie pobierany jest użytkownik na podstawie tokenu z nagłówka żądania. Kolejny krok to weryfikacja, czy rachunek istnieje i należy do użytkownika, a następnie mapowanie na obiekt *DTO* i zwrócenie rezultatu do kontrolera.

```
public async Task<ReceiptDto> GetReceipt(int receiptId)
{
    _validator.IsNull(receiptId, "Brak identyfikatora rachunku");
    var user = await _managementService.GetUser();
    var receipt = await _managementService.GetReceipt(user, receiptId);
    var receiptDto = _mapper.Map<ReceiptDto>(receipt);
    return receiptDto;
}
```

Rys. 5.15. Implementacja metody GetReceipt w klasie ReceiptService.

5.2.4. Interfejs Swagger dla modułu rachunków

Poprzez poprawnie zaimplementowane kontrolery, interfejs *Swagger* generuje graficzne odnośniki, które umożliwiają wysyłanie żądań *HTTP*, upraszczając proces testowania zapytań *API*. Przykładowo, interfejs *Swagger* modułu rachunków zawiera metody odpowiedzialne za tworzenie, edycję, usunięcie oraz wyświetlanie rachunków (rys. 5.16). Dzięki swojej przejrzystości i prostocie, stanowi wygodne narzędzie do interakcji z aplikacją serwera.



Rys. 5.16. Swagger przedstawiający listę dostępnych operacji API związanych z paragonami.

5.2.5. Middleware obsługujący wyjątki rzucane przez serwis Web API

Middleware jest to oprogramowanie działające pomiędzy żądaniami a odpowiedzią aplikacji serwerowej. Umożliwia on przeprowadzenie wstępnych operacji na żądaniach, zanim aplikacja udzieli odpowiedzi.

ExceptionMiddleware jest oprogramowaniem pośredniczącym, odpowiadającym za obsługę wyjątków. Jego celem jest wyłapanie błędów podczas obsługi żądania *HTTP* i wygenerowanie odpowiedniej odpowiedzi dla użytkownika (rys. 5.17). Mechanizm ma zapewnić bezpieczną i jednolitą obsługę błędów po stronie serwera. Poprzez to działanie użytkownicy dostają szczegółowe informacje o kodzie błędów, a programiści mogą łatwo rozszerzać logikę obsługi wyjątków.

```
public class ExceptionMiddleware : IMiddleware
{
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        try
        {
            await next(context);
        }
        catch (DException ex)
        {
            await HandleDExceptionAsync(context, ex);
        }
        catch (Exception ex)
        {
            await HandleExceptionAsync(context, ex);
        }
    }

    private Task HandleDExceptionAsync(HttpContext context, DException ex)
    {
        context.Response.ContentType = "text/plain";
        context.Response.StatusCode = (int)ex.Status;
        return context.Response.WriteAsync(ex.Message);
    }

    private Task HandleExceptionAsync(HttpContext context, Exception ex)
    {
        context.Response.ContentType = "text/plain";
        context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;
        return context.Response.WriteAsync("An unexpected error occurred");
    }
}
```

Rys. 5.17. Implementacja klasy ExceptionMiddleware.

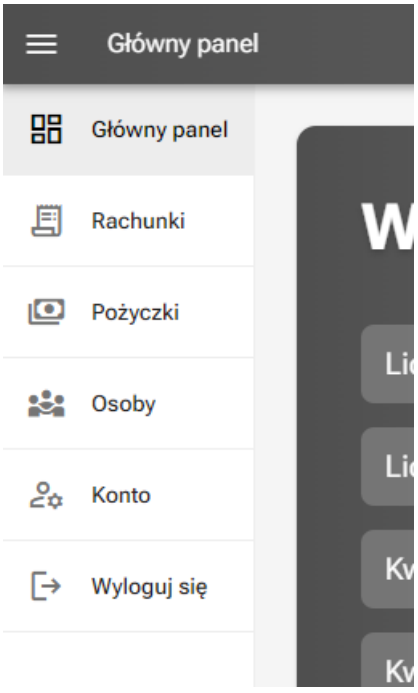
6. Interfejs użytkownika

Interfejs użytkownika (*UI*) stanowi kluczową warstwę systemu, która bezpośrednio łączy użytkownika z aplikacją, umożliwiając korzystanie z jej funkcjonalności. Na tym elemencie położono szczególny nacisk, gdyż wpływa on w znacznym stopniu na ogólne doświadczenia użytkownika (*UX*), jego komfort pracy z systemem, a także na efektywność i intuicyjność obsługi. Dobry interfejs użytkownika nie tylko ułatwia korzystanie z aplikacji, ale również buduje pozytywny wizerunek systemu w oczach użytkownika.

W niniejszym rozdziale przedstawiono szczegółowe opisy interfejsu graficznego aplikacji. Uwzględniono zarówno projekt oraz układ poszczególnych ekranów, jak i opisy ich funkcjonalności, które zostały zaprojektowane z myślą o maksymalnym uproszczeniu i ułatwieniu interakcji.

6.1. Nawigacja po aplikacji

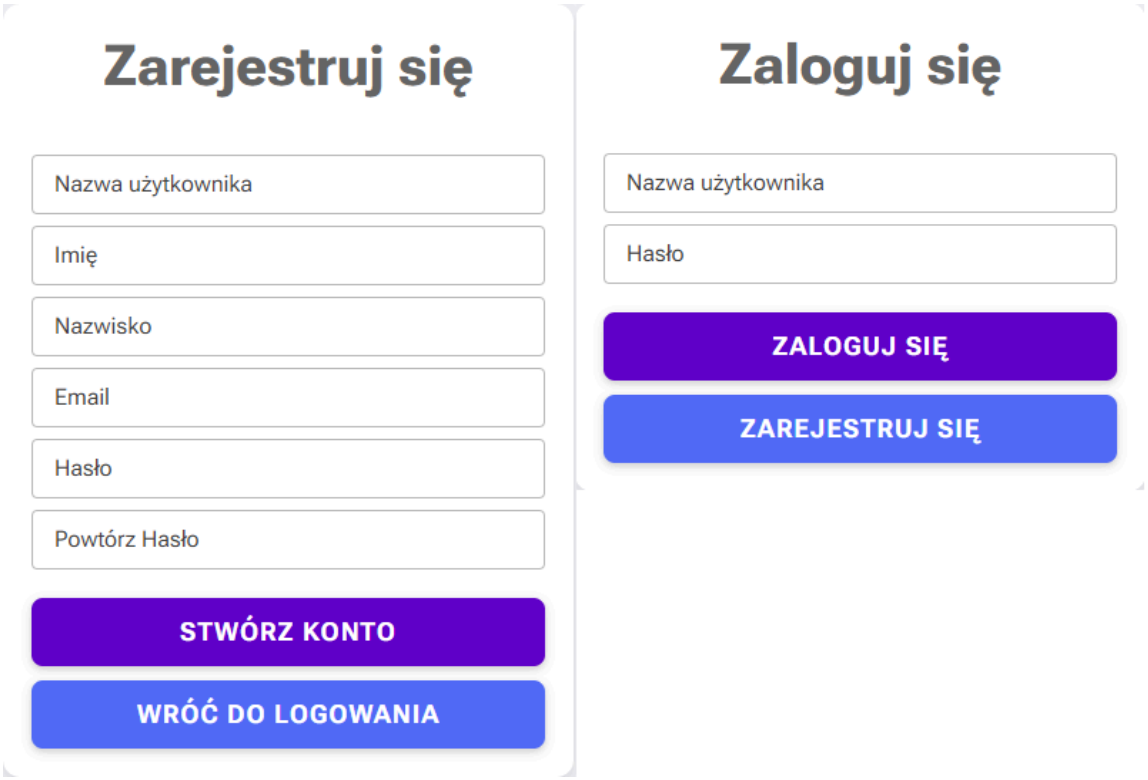
Nawigacja po aplikacji odbywa się za pośrednictwem panelu bocznego (rys. 6.1), którego zawartość dostosowuje się w zależności od stanu zalogowania użytkownika. Panel wyświetla nazwy stron, do których możliwe jest przejście, a w nagłówku widnieje nazwa bieżącej strony. Zapewnia to płynną nawigację po systemie oraz intuicyjną przejrzystość interfejsu.



Rys. 6.1. Panel nawigacyjny.

6.2. Formularze logowania oraz rejestracji

Aplikacja wymaga aktywnego konta użytkownika do działania. Ekran rejestracji umożliwia utworzenie konta, a następnie przejście na stronę logowania aby użytkownik mógł zweryfikować swoją tożsamość (rys. 6.2).



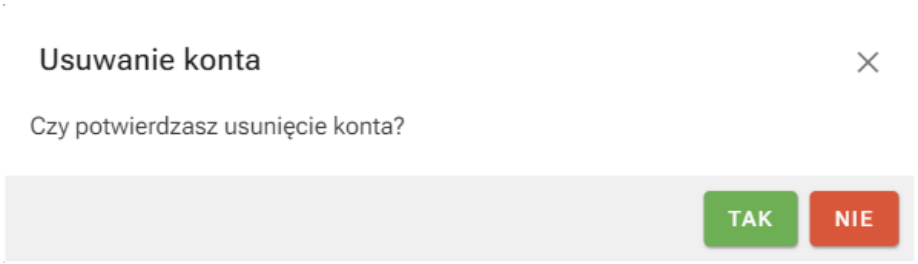
Rys. 6.2. Formularze rejestracji oraz logowania.

Formularze zostały wyposażone w mechanizmy walidacji, które weryfikują poprawność wypełnienia pól. W przypadku pola hasła wymagane jest, aby spełniało ono określone kryteria: musi zawierać od 8 do 15 znaków, w tym co najmniej jedną dużą literę, jeden znak specjalny oraz jedną cyfrę. Jeśli wprowadzone dane nie spełniają tych wymagań, odpowiednia informacja o błędzie zostanie wyświetlona pod polem formularza (rys. 6.3).

Rys. 6.3. Walidacja danych w formularzu rejestracji.

6.3. Dialog potwierdzania operacji

Każda operacja usunięcia poprzedzona jest wyświetleniem okna dialogowego (rys. 6.4), które wymaga potwierdzenia. Mechanizm ten zapewnia ochronę przed przypadkowym wykonaniem tej czynności.

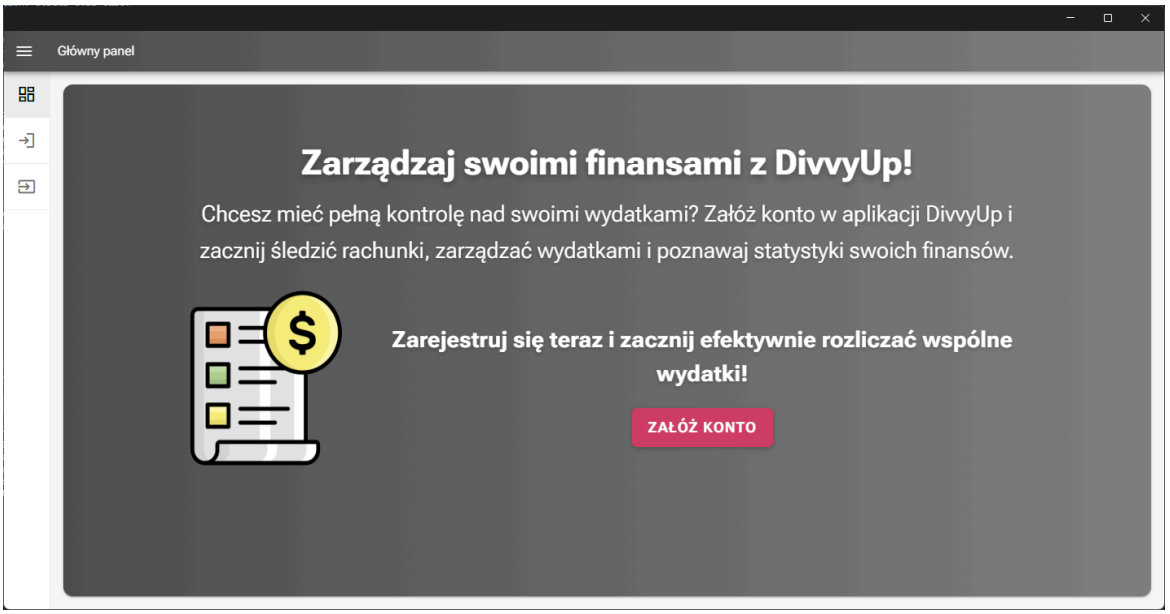


Rys. 6.4. Dialog potwierdzenia wykonania operacji.

6.4. Widok panelu głównego (niezalogowany użytkownik)

Widok panelu głównego dla niezalogowanego użytkownika został zaprojektowany z myślą o zachęceniu nowych użytkowników do rejestracji oraz informowaniu ich o konieczności utworzenia konta, aby uzyskać dostęp do pełnej funkcjonalności aplikacji (rys. 6.5).

Logo aplikacji pochodzi z platformy *Flaticon* i jest udostępniane na licencji "*Free for personal and commercial purpose with attribution*". Zgodnie z warunkami licencji, autor grafiki został wskazany w materiałach aplikacji oraz dokumentacji [16].



Rys. 6.5. Ekran panelu głównego (niezalogowany użytkownik).

6.5. Widok panelu głównego (zalogowany użytkownik)

Widok panelu głównego dla zalogowanego użytkownika dostarcza najważniejszych danych w przejrzystej formie, umożliwiając szybki wgląd w aktywność i finanse (rys. 6.6). Na górze ekranu widoczna jest nazwa użytkownika oraz podsumowanie podstawowych informacji, takich jak liczba rachunków i produktów przypisanych do konta.

Jednym z ciekawszych elementów widoku jest statystyka rzetelności osób, która prezentowana jest w formie procentowej. Użytkownik może zobaczyć, w jakim stopniu osoby biorące udział w jego rozliczeniach wywiązują się ze swoich zobowiązań. Dzięki temu łatwo ocenić, które kontakty są najbardziej niezawodne.

W centralnej części panelu znajdują się wykresy podsumowujące kwotę łączną i rozliczoną, a także lista najdroższych produktów w bieżącym miesiącu. Sekcja trendów wydatków przedstawia dane o wydatkach na przestrzeni tygodnia i roku, umożliwiając monitorowanie finansowych nawyków. Całość została zaprojektowana tak, aby zapewnić intuicyjną i przyjazną obsługę.

Rys. 6.6. Ekran panelu głównego (zalogowany użytkownik).

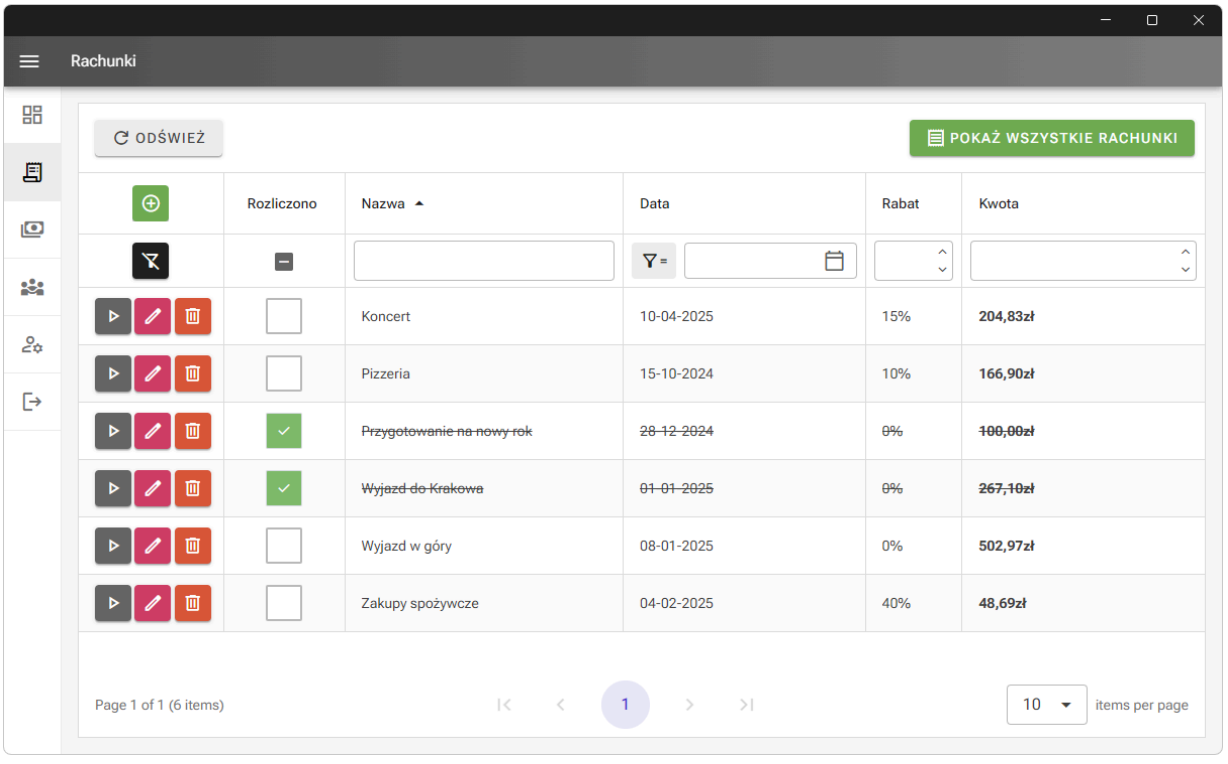
6.6. Strona listy rachunków

Interfejs listy rachunków pozwala użytkownikowi na zarządzanie swoimi rachunkami. Na ekranie wyświetlana jest lista wszystkich zapisanych rachunków z podstawowymi informacjami, takimi jak kwota, wysokość rabatu, data wystawienia i status rozliczenia (rys. 6.7).

W górnej części ekranu znajdują się opcje dodawania nowych rachunków. Edycja istniejących rachunków odbywa się poprzez naciśnięcie przycisku w odpowiednim wierszu. Użytkownik może również ustawić zakres dat wyświetlanych rachunków, co ułatwia przeglądanie informacji w odpowiednich okresach czasowych.

Ustawienie wartości rabatu wpływa na domyślnie ustawienie rabatowania dla wszystkich przyszłych dowanych produktów w obrębie danego rachunku.

Całość interfejsu jest zaprojektowana w sposób przejrzysty i intuicyjny, zapewniając łatwą nawigację po liście rachunków i szybki dostęp do najważniejszych funkcji.



Rys. 6.7. Ekran listy rachunków użytkownika.

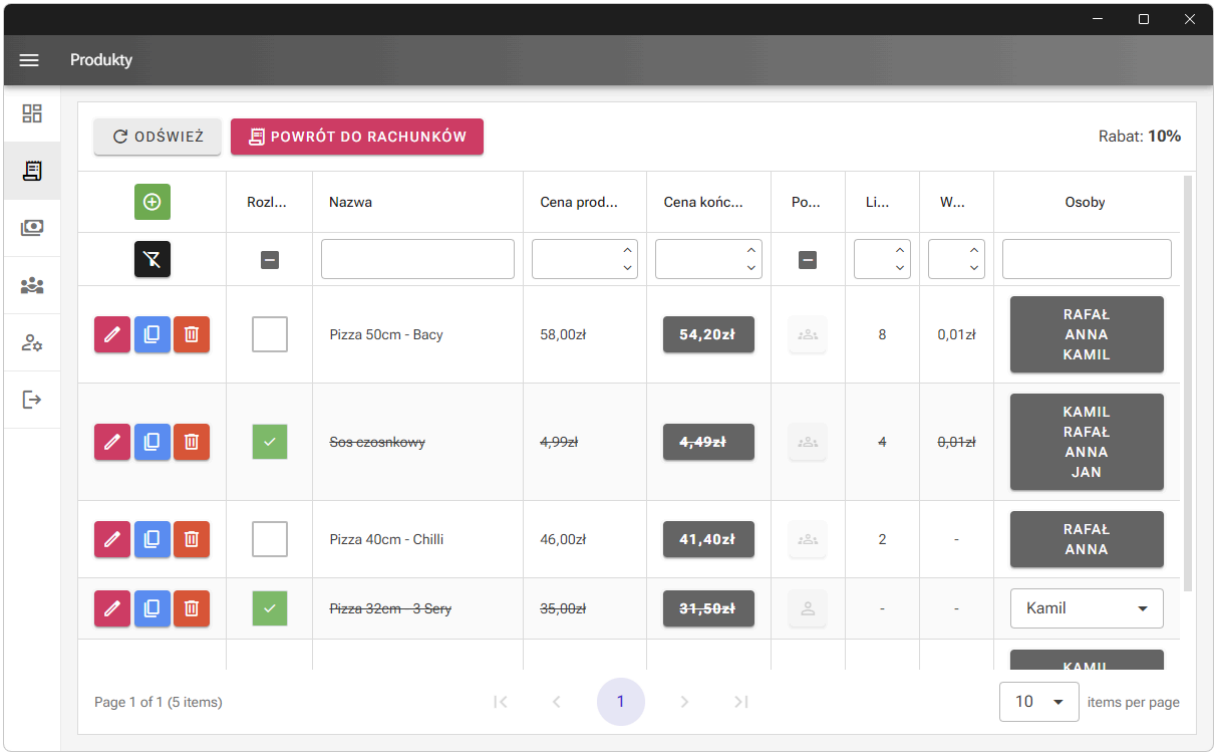
6.7. Strona listy produktów

Interfejs listy produktów pozwala na łatwe zarządzanie produktami w obrębie danego rachunku (rys. 6.8). Użytkownik może dodać nowy produkt, nadając mu nazwę, ustalając cenę oraz decydując, czy produkt jest podzielny. W przypadku produktów podzielnych, użytkownik może określić, na ile części produkt ma być rozdzielony. Program daje możliwość skopiowania produktu z bieżącej listy duplikując go co uprości proces dodawania bliźniaczych produktów w obrębie jednego rachunku.

Cena końcowa produktu to jego bazowa wartość, uwzględniająca rabat, przemnożenie przez liczbę sztuk oraz dodanie dodatkowej opłaty. Po kliknięciu na tę cenę otworzy się okno umożliwiające zarządzanie szczegółami produktu.

W widoku produktów użytkownik może również zarządzać stanem rozliczenia produktu, decydując, czy został on w pełni rozliczony, czy wymaga jeszcze uregulowania. Dodatkowo w tym widoku można przypisać osoby do produktu, wskazując, kto bierze udział w rozliczeniu, a także sprawdzić skład grupy przypisanej do produktu.

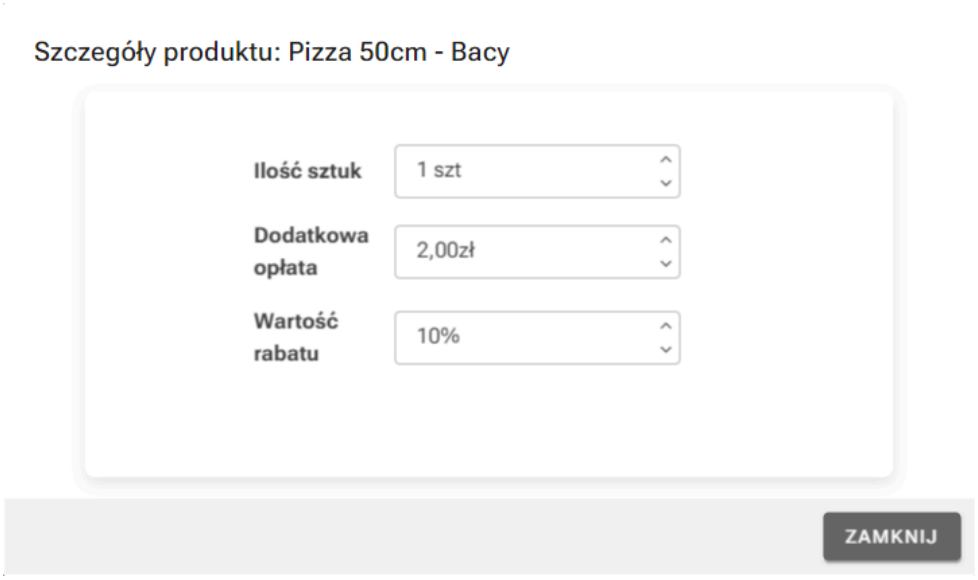
Na ekranie widoczna jest także sekcja „*Wyrównanie*”, która wskazuje, ile jeszcze pozostało do uregulowania, jeśli nie wszystkie części produktu zostały przypisane. Dzięki temu użytkownik zyskuje pełen wgląd w proces podziału kosztów i ich rozliczeń.



Rys. 6.8. Ekran listy produktów przypisanych do rachunku.

6.8. Szczegóły produktu

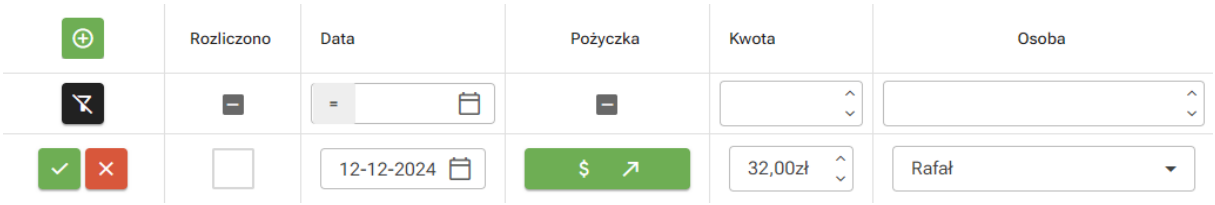
Szczegóły produktu zawierają informacje dodatkowe informacje o produkcie (rys. 6.9), umożliwiają nałożenie procentowej wartości rabatowania, wpisania ilości sztuk a także nałożenie dodatkowej opłaty która będzie dodana do ceny końcowej.



Rys. 6.9. Okno zawierające szczegóły produktu.

6.9. Modyfikacja danych w siatce danych

Modyfikacja danych w siatce danych odbywa się poprzez edycję poszczególnych wierszy wybranego wiersza lub nowo utworzonego wpisu (rys. 6.10). Wiersz przełącza się w tryb edycji a po naniesieniu zmian możemy je zapisać klikając na przycisk zapisu który znajduje się w pierwszej kolumnie wybranego wiersza.



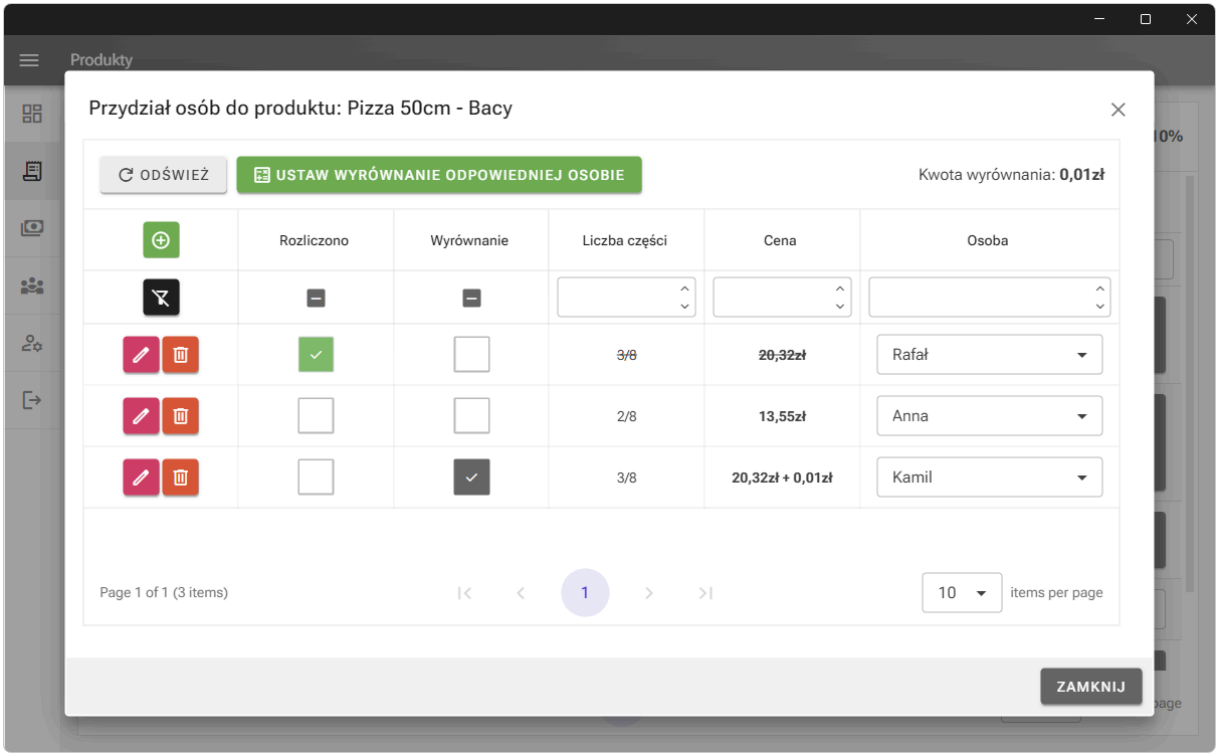
Rys. 6.10. Modyfikacja wiersza w siatce danych.

6.10. Okno przydziału produktu do osób

Jeśli produkt można podzielić na mniejsze części, możliwe jest otwarcie okna przypisów do poszczególnych osób (rys. 6.11). Interfejs ten pozwala użytkownikowi na określenie wartości podziału między osoby, umożliwiając manipulowanie udziałem każdej z nich w danym produkcie. Siatka danych zawiera informacje o stanie rozliczenia, cenie, przypisanej osobie oraz statusie wyrównania, który wskazuje, kto poniesie koszt wyrównania za dany przedmiot. Dodatkowo prezentuje udział każdej osoby w produkcie w odniesieniu do maksymalnej liczby jednostek, na jakie można go podzielić. Pozostałość wyrównania wyświetlana jest w prawym górnym rogu.

Wyrównanie można ustawić na dwa sposoby. W pierwszym użytkownik samodzielnie decyduje, kto poniesie koszt wyrównania, wybierając odpowiedni wiersz. W drugim, po naciśnięciu przycisku „*USTAW WYRÓWNANIE ODPOWIEDNIEJ OSOBIE*”, system analizuje listę osób i wybiera tę, która ma najmniejszą wartość wyrównania ze wszystkich zakupów, przypisując ją jako wyrównującą.

Takie rozwiązanie pozwala na sprawiedliwe rozdzielenie kosztów i właściwe przypisanie wyrównania.



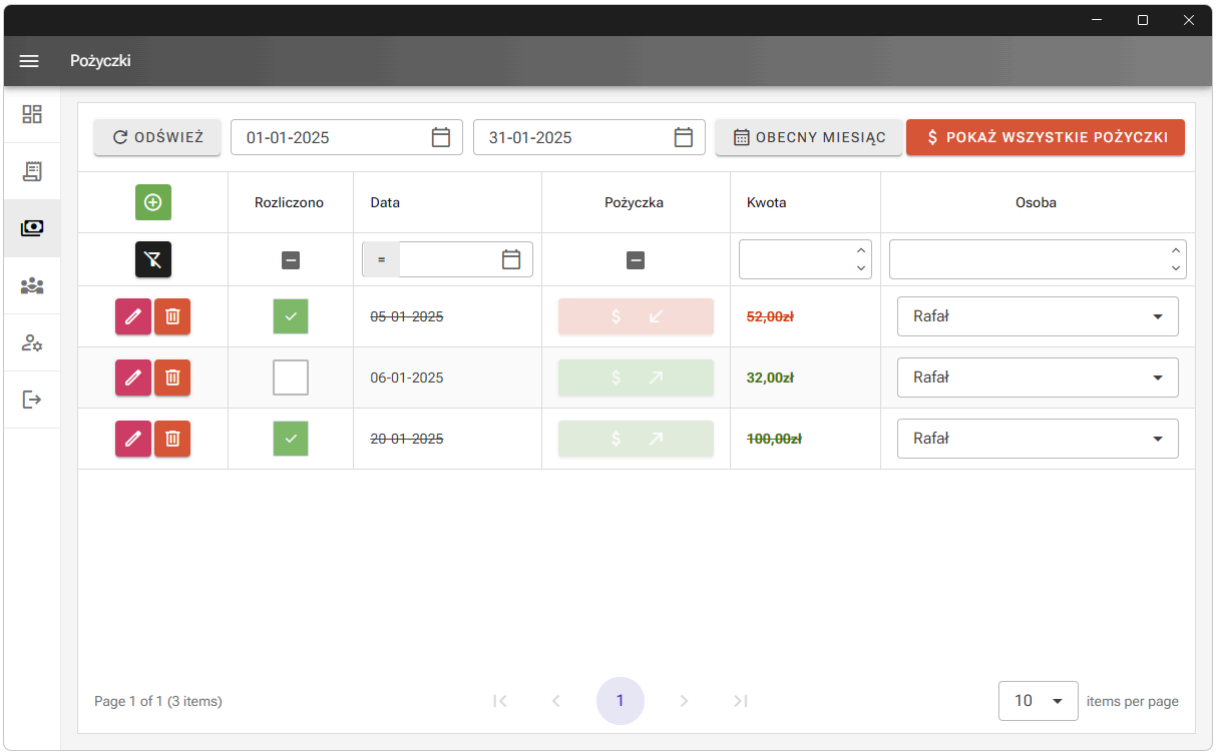
Rys. 6.11. Okno przydziału produktu do osób.

6.11. Strona listy pożyczek

Aplikacja umożliwia udzielanie oraz zaciąganie pożyczek od wybranych osób, co znacząco upraszcza proces zarządzania pożyczkami oraz prowadzenie historii związanych z takimi transakcjami. Każdy zapis w systemie zawiera szczegółowe informacje, takie jak data pożyczki, status (czy została już zwrócona), typ transakcji (pożyczka udzielona przez użytkownika lub pożyczka zaciągnięta od innej osoby), kwota oraz przypisana osoba.

Transakcja, w której użytkownik zaciąga pożyczkę od wybranej osoby, jest oznaczona kolorem czerwonym, co wskazuje, że użytkownik posiada dług wobec tej osoby i musi go uregulować. Z kolei sytuacja, w której użytkownik pożycza kwotę innej osobie, jest oznaczona kolorem zielonym, co sygnalizuje, że oczekuje na zwrot tej kwoty od wskazanej osoby.

Dostępna lista transakcji umożliwia filtrowanie wpisów według zakresu dat lub wyświetlanie wszystkich zapisanych wpisów (rys. 6.12).



Rys. 6.12. Ekran listy pożyczek.

6.12. Strona listy osób

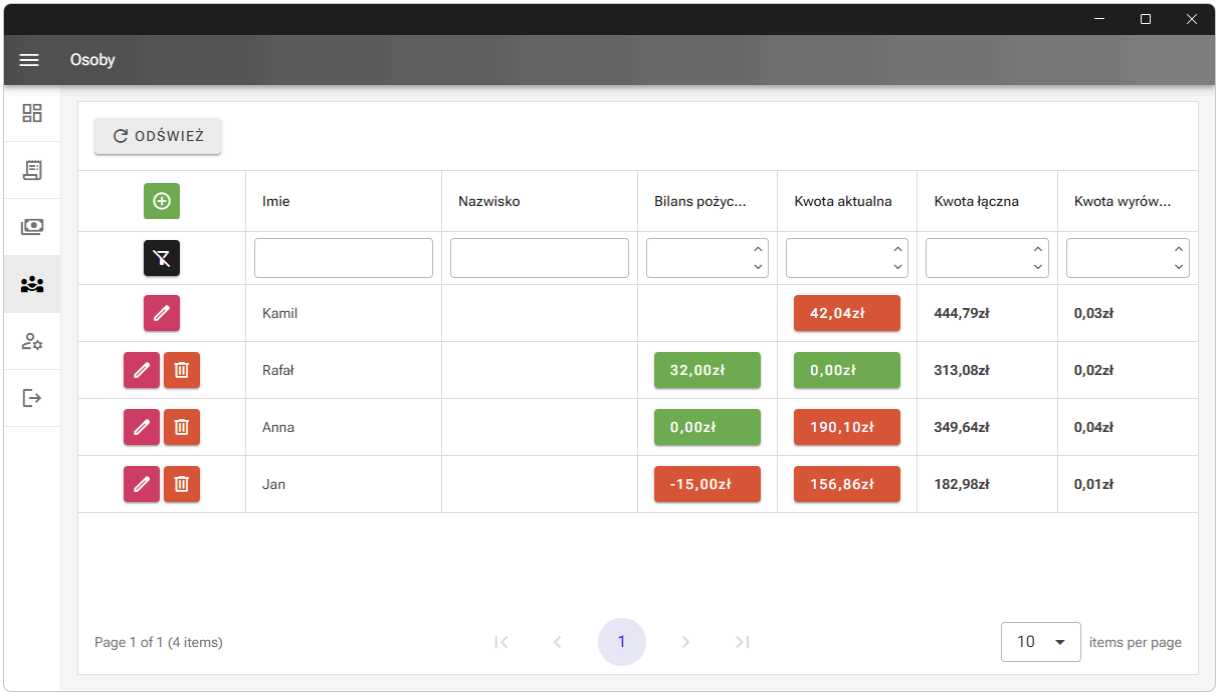
W każdej chwili użytkownik może śledzić bilanse osób utworzonych przez siebie. Taką opcję umożliwia strona osób (rys. 6.13). Użytkownik ma możliwość dodawania, edytowania osób oraz wyświetlania ich bilansu pożyczkowego, który jest liczony z perspektywy użytkownika.

Gdy bilans jest ujemny, oznacza to, że użytkownik pożyczył pieniądze od osoby. Natomiast gdy osoba pożyczyła pieniądze użytkownikowi, bilans jest dodatni. Po kliknięciu na bilans pożyczkowy wybranej osoby, wyświetlana jest lista pożyczek związanych między tą osobą a użytkownikiem.

Kolejne statystyki ukazują kwotę aktualną, czyli sumę kwot przypisanych do produktów, które nie zostały opłacone. Po naciśnięciu na kwotę aktualną, program przenosi użytkownika do listy wszystkich przypisań do produktów z którymi była związana wybrana osoba, umożliwia to łatwy wgląd do wszystkich produktów wybranej osoby.

Kwota łączna to suma wszystkich kwot za przypisane produkty do których osoba była przypisana.

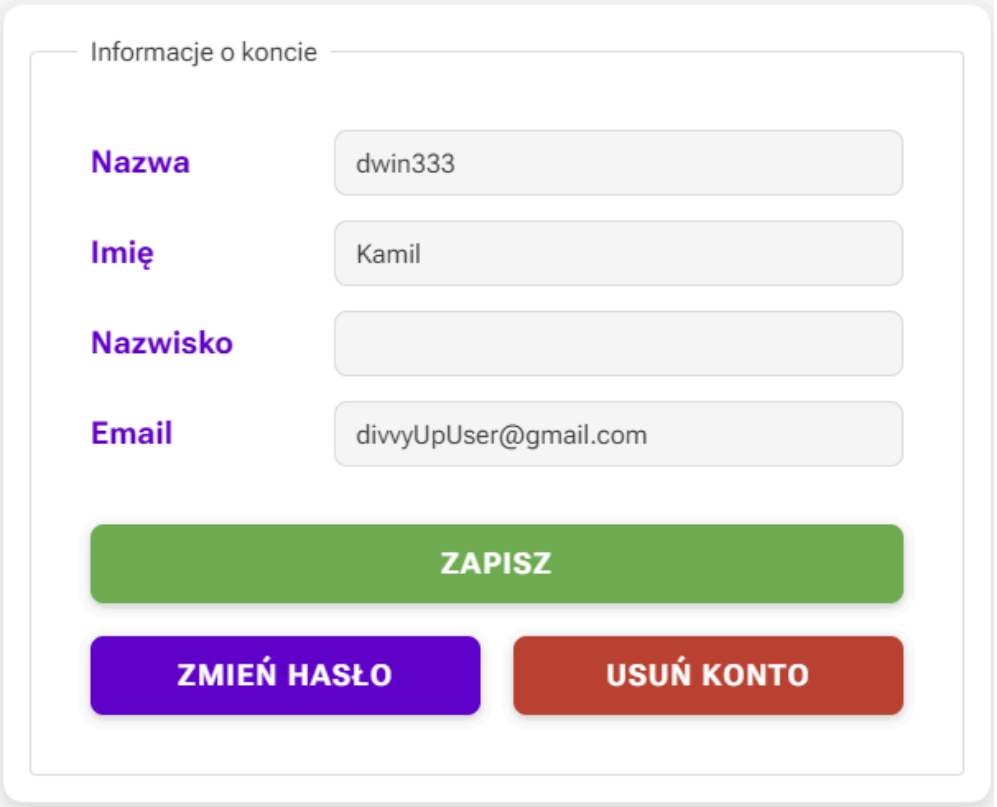
Kwota wyrównania jest sumowana z kwoty wyrównań wszystkich produktów w których osoba poniosła wyrównanie, daje ona informacje jaką ta osoba miała wkład i ile poniosła jej łączna kwota wyrównań co upraszcza proces wybierania odpowiedniej osoby podczas rozbijania produktu na części.



Rys. 6.13. Ekran listy osób.

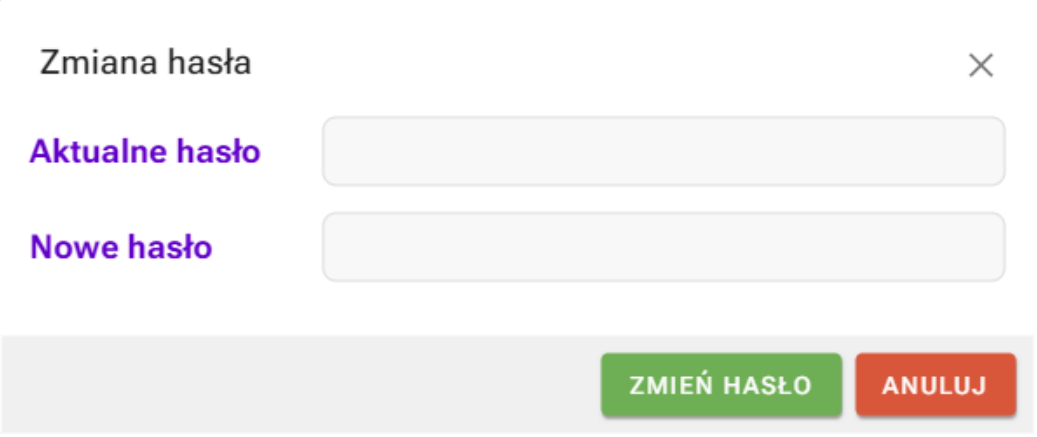
6.13. Interfejs zarządzania kontem

Strona zarządzania kontem użytkownika wyświetla formularz (rys. 6.14), który umożliwia edytowanie danych konta. Użytkownik może zmienić swoją nazwę, imię, nazwisko, adres e-mail. W razie potrzeby może również usunąć konto po potwierdzeniu tej czynności, co wiąże się z usunięciem wszystkich danych powiązanych z kontem.



Rys. 6.14. Formularz zarządzania kontem.

Dostępna jest także opcja zmiany hasła (rys. 6.15). Aby to zrobić, użytkownik musi podać aktualne hasło oraz wprowadzić nowe hasło, które będzie przypisane do konta.



Rys. 6.15. Formularz zmiany hasła.

7. Testy

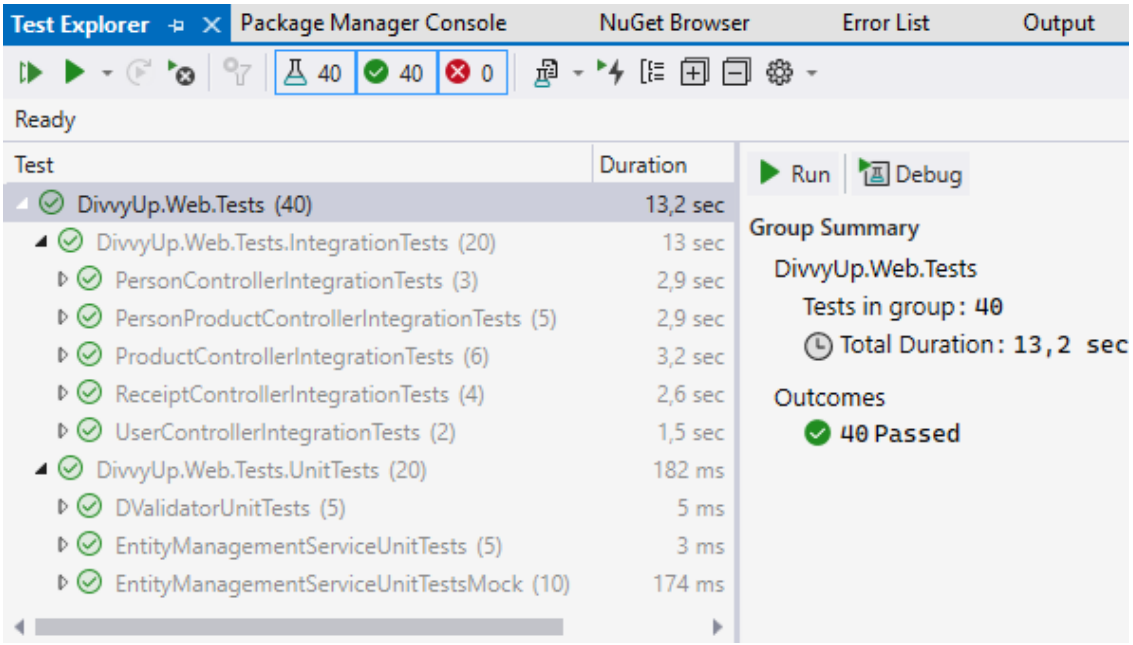
Rozdział ten poświęcony będzie przybliżeniu tematu testowania środowiska w obszarze aplikacji serwerowej. Testowanie kodu to proces czasochłonny, jednak przynosi liczne korzyści, które wynikają z zastosowania tej praktyki podczas projektowania aplikacji. Takie podejście pozwala na testowanie aplikacji w różnych scenariuszach, co może pomóc w zabezpieczeniu jej przed różnymi podatnościami, a także umożliwia weryfikację słuszności wybranych rozwiązań oraz ich wpływu na działanie aplikacji. Testy zaimplementowane w systemie, które znajdują się w projekcie *DivvyUp.Web.Test*, mają na celu poprawę niezawodności całego systemu oraz weryfikację, czy po wprowadzeniu zmian lub rozbudowie poszczególnych obszarów aplikacji i modułów, każdy z nich nadal pełni swoją funkcję prawidłowo.

Narzędziami wykorzystywanymi do testów są m.in. framework *xUnit*, który umożliwia przeprowadzanie testów jednostkowych oraz testów integracyjnych. Jest to popularny framework w *.NET*, po który programiści często sięgają podczas tworzenia oprogramowania. Wykorzystuje on adnotacje *[Fact]*, *[Theory]* oraz metody *Assert.Equal*, *Assert.NotNull*, *Assert.ThrowsAsync*, które pozwalają na łatwe definiowanie testów i weryfikację wyników [11].

Kolejnym wartym uwagi narzędziem jest *Microsoft.AspNetCore.Mvc.Testing*. Dzięki niemu możliwe jest uruchomienie aplikacji *ASP.NET* Core w środowisku testowym, co pozwala na wysyłanie zapytań *HTTP* podczas testowania kontrolerów lub innych części systemu w kontekście rzeczywistych scenariuszy. Mechanizm ten jest najczęściej wykorzystywany w testach integracyjnych, gdzie testuje się interakcje pomiędzy komponentami, symulując rzeczywiste warunki użytkowania.

Do tworzenia testów wykorzystano również własnoręcznie napisane klasy pomocnicze. Pierwszą z nich jest *DataFactory.cs*, która służy do tworzenia obiektów testowych i umożliwia generowanie danych testowych w kontrolowany oraz wygodny sposób. Drugą klasą jest *TestHelper.cs*, odpowiadająca za różne operacje wspierające proces testowania. Obejmuje ona między innymi czyszczenie testowej bazy danych, rejestrację testowego użytkownika, generowanie dla niego tokenu oraz tworzenie elastycznych żądań *HTTP* z danymi dostosowanymi do potrzeb testu, w tym zawierających odpowiednią treść oraz token autoryzacyjny. Dzięki tym klasom proces pisania i utrzymywania testów staje się bardziej przejrzysty i efektywny.

Projekt testowy *DivvyUp.Web.Tests* obejmował łącznie 40 testów. Składał się z 20 testów jednostkowych oraz 20 testów integracyjnych. Testy skupiały się na weryfikacji działania warstwy serwerowej, wraz z sprawdzeniem najważniejszych mechanik, zachowanie kontrolerów, sprawdzenie serwisów walidacyjnych oraz tych pomocniczych które odgrywają ważną rolę w systemie. Wszystkie testy przeszły pomyślnie a cały proces zakończył się po łącznym czasie 13,2 sekund (rys. 7.1).



Rys. 7.1. Testy przeprowadzone projekcie DivvyUp.Web.Tests.

7.1. Wzorce wykorzystywane w testach

Wzorzec *AAA* (*Arrange-Act-Assert*) jest popularnym schematem stosowanym w testowaniu oprogramowania. Opiera się on na określonej strukturze, która obejmuje trzy kroki w procesie testowania. Pierwszy z nich, **Arrange**, polega na przygotowaniu danych do testu, ustawieniu wartości zmiennych lub obiektów. Następnie wykonywany jest **Act**, czyli proces poddawany testowaniu, który obejmuje wywołanie funkcji lub modułu, mającego na celu zrealizowanie określonego działania. Ostatnim etapem jest **Assert**, czyli sprawdzenie poprawności otrzymanych wyników poprzez porównanie ich z zakładanymi wartościami, co pozwala zweryfikować poprawność działania testowanego kodu [15].

7.2. Testy jednostkowe

Testy jednostkowe charakteryzują się prostotą i służą do sprawdzenia działania poszczególnych funkcji oraz oceny, jak aplikacja zachowa się w różnych scenariuszach. Pozwalają one zweryfikować, czy dana funkcja lub algorytm działa zgodnie z zaplanowanym założeniem, a aplikacja zachowuje się w sposób oczekiwany w różnych warunkach [14].

W przypadku prawidłowego podziału ceny produktu na poszczególne części, które mają zostać pokryte przez osoby, kluczowe jest zapewnienie niezawodności tej funkcjonalności oraz jej odpowiednie przetestowanie. Testowanie wykorzystuje atrybut *[Fact]*, który wskazuje, że dana metoda jest testowa (rys. 7.2). Proces testowania rozpoczyna się od przygotowania danych, takich jak maksymalna podzielność produktu, wartość części przypisania oraz całkowita cena produktu. Następnie wywoływana jest funkcja serwisu odpowiedzialna za obliczanie ceny, po czym weryfikowane jest, czy wynikowa cena zgadza się z założoną wartością.

```
[Fact]
public async Task CalculatePartOfPrice_ShouldReturnCorrectResult()
{
    // Arrange
    int quantity = 2;
    int maxQuantity = 10;
    decimal price = 100m;

    // Act
    var result = await _service.CalculatePartOfPrice(quantity, maxQuantity, price);

    // Assert
    Assert.Equal(20.00m, result);
}
```

Rys. 7.2. Test jednostkowy - podział ceny produktu.

Podczas tworzenia produktu, który uwzględnia wiele zmiennych wpływających na obliczenie finalnej ceny, testy muszą być przeprowadzone w sposób różnorodny, aby sprawdzić podatności, na które narażona jest metoda. Metoda obliczania ceny końcowej została przetestowana za pomocą czterech różnych testów, z których każdy koncentrował się na innym aspekcie, takim jak liczba sztuk, rabaty, dodatkowe koszty oraz weryfikacja działania wszystkich mechanizmów jednocześnie. Test kompleksowy (rys. 7.3), obejmujący wszystkie manipulacje w produkcie mające wpływ na obliczanie ceny końcowej, został poprawnie zaimplementowany i zwrócił oczekiwany wynik.

```
[Fact]
public async Task CalculateTotalPrice_ShouldCalculateCorrectly()
{
    // Arrange
    decimal price = 10.00m;
    int purchasedQuantity = 2;
    decimal additionalPrice = 0.50m;
    int discountPercentage = 10;


    // Act
    var result = _service.CalculateTotalPrice(price, purchasedQuantity, additionalPrice, discountPercentage);

    // Assert
    Assert.Equal(18.50m, result);
}
```

Rys. 7.3. Test jednostkowy - obliczanie ceny końcowej.

W niektórych testach jednostkowych zastosowano mechanizm *mock*, który umożliwił odseparowanie testowanego kodu od rzeczywistej bazy danych. Dzięki temu testy mogły być realizowane w pełni kontrolowanym środowisku, co znacząco przyspieszyło proces ich przeprowadzania oraz wyeliminowało konieczność operowania na rzeczywistych danych zapisanych w bazie [18].

Projekt serwera musi również zapewniać weryfikację, czy użytkownik ma uprawnienia do dostępu do określonych danych. Przeprowadzono test, który sprawdzał sytuację, w której użytkownik, niebędący właścicielem rachunku, próbuje go wyświetlić.

W ramach testu przygotowano dane dla dwóch użytkowników oraz rachunku należącego do użytkownika o identyfikatorze 1. Następnie, przy użyciu mocka, zasymulowano bazę danych, w której znajdowały się te obiekty.

Podczas próby wywołania metody *GetReceipt* z danymi użytkownika, który nie jest właścicielem rachunku, sprawdzono, czy serwer poprawnie odmawia dostępu. Oczekiwano, że w takim przypadku zostanie rzucony wyjątek o statusie "*Unauthorized*" (co oznacza brak autoryzacji) oraz zwrócony komunikat: „*Brak dostępu do rachunku: 1*” (rys. 7.4).

```
[Fact]
public async Task GetReceipt_ShouldThrowUnauthorizedException_WhenUserDoesNotOwnReceipt()
{
   // Arrange
   var user1 = new User { Id = 1 };
   var user2 = new User { Id = 2 };

   var receipt = new Receipt { Id = 1, UserId = user1.Id, TotalPrice = 100.00m };

   var users = new List<User> { user1, user2 };
   var receipts = new List<Receipt> { receipt };

   var usersMock = users.ToMockDbSet();
   var receiptsMock = receipts.ToMockDbSet();

   _dbContextMock.Setup(db => db.Users).Returns(usersMock.Object);
   _dbContextMock.Setup(db => db.Receipts).Returns(receiptsMock.Object);

   // Act & Assert
   var exception = await Assert.ThrowsAsync<DException>(() =>
   _service.GetReceipt(user2, receipt.Id));

   Assert.Equal(HttpStatusCode.Unauthorized, exception.Status);
   Assert.Equal($"Brak dostępu do rachunku: {receipt.Id}", exception.Message);
}
```

Rys. 7.4. Test jednostkowy - dostęp dla zasobu przez nieuprawnioną osobę.

Przykładową funkcjonalnością wartą przetestowania jest mechanizm odpowiedzialny za zwrócenie osoby z najmniejszą wartością wyrównania, wybieranej spośród listy osób przypisanych do produktu. Proces testowania obejmuje przygotowanie danych, wykonanie akcji za pomocą funkcji serwisu oraz weryfikację, czy zwrócono właściwą osobę (rys. 7.5).

```
[Fact]
public async Task GetPersonWithLowestCompensation_ShouldReturnPersonWithLowestCompensation()
{
    // Arrange
    int productId = 1;

    var persons = new List<Person>
    {
        new Person { Id = 1, CompensationAmount = 10.00m },
        new Person { Id = 2, CompensationAmount = 5.00m },
        new Person { Id = 3, CompensationAmount = 7.00m }
    };

    var personProducts = new List<PersonProduct>
    {
        new PersonProduct { Id = 1, PersonId = 1, ProductId = productId, Person = persons[0] },
        new PersonProduct { Id = 2, PersonId = 2, ProductId = productId, Person = persons[1] },
        new PersonProduct { Id = 3, PersonId = 3, ProductId = productId, Person = persons[2] }
    };

    _dbContextMock.Setup(db => db.Persons).Returns(persons.ToMockDbSet().Object);
    _dbContextMock.Setup(db => db.PersonProducts).Returns(personProducts.ToMockDbSet().Object);

    // Act
    var result = await _service.GetPersonWithLowestCompensation(productId);

    // Assert
    Assert.NotNull(result);
    Assert.Equal(2, result.PersonId);
}
```

Rys. 7.5. Test jednostkowy - pobranie osoby z najmniejszą wartością wyrównania.

Serwer wyposażono w mechanizm walidatora (*DValidator*), którego zadaniem jest weryfikacja poprawności danych wejściowych przesyłanych wraz z żądaniami. Walidator odgrywa kluczową rolę, ponieważ jest wykorzystywany w wielu miejscach w implementacji serwera. W tym przypadku test skupia się na sprawdzeniu, czy mechanizm poprawnie waliduje zakres dat, w szczególności czy data początkowa jest wcześniejsza niż data końcowa (rys. 7.6).

```
[Fact]
public void IsCorrectDataRange_WhenValueIsMinus_ShouldThrowException()
{
    // Arrange
    DateOnly dateFrom = new DateOnly(2024, 12, 24);
    DateOnly dateTo = new DateOnly(2024, 12, 7);

    // Act
    var exception = Assert.Throws<DException>(() =>
    {
        _validator.IsCorrectDataRange(dateFrom, dateTo);
    });

    //Assert
    Assert.Equal(HttpStatusCode.BadRequest, exception.Status);
    Assert.Equal("Zakres dat jest źle ustawiony", exception.Message);
}
```

Rys. 7.6. Test jednostkowy - weryfikacji działania walidatora zakresu dat.

7.3. Testy integracyjne

Testy integracyjne są bardziej złożone niż jednostkowe. Ich celem jest testowanie całych modułów, połączeń między komponentami oraz sprawdzanie mechanizmów w szerszym kontekście. Testy integracyjne weryfikują stan aplikacji w środowisku zbliżonym do rzeczywistego, ponieważ uwzględniają więcej czynników, które mogą ujawnić potencjalne nieprawidłowości [14].

Testowanie różnorodnych scenariuszy obejmuje między innymi zmianę wartości rabatu dla całego rachunku, nawet w sytuacji, gdy rachunek zawiera już produkty. System powinien być przygotowany na takie modyfikacje i automatycznie nanosić zmiany na wszystkie powiązane produkty. Ten proces jest weryfikowany w ramach testu poświęconego edycji rachunku ze zmianą rabatu.

Przeprowadzony test miał na celu modyfikację wartości rabatu na rachunku, a następnie sprawdzenie, czy zarówno rachunek, jak i znajdujące się w nim produkty zostały zaktualizowane zgodnie z wprowadzonymi zmianami. Test weryfikujący zachowanie systemu po edycji rabatu zakończył się zgodnie z oczekiwaniami i zwrócił poprawne wyniki (rys. 7.7).

```
[Fact]
public async Task EditReceipt_ChangeDiscountPercentage_WithValid_ShouldSucceed()
{
    // Arrange
    var editReceipt = new AddEditReceiptDto()
    {
        Name = "TestReceipt",
        DiscountPercentage = 15
    };
    var url = ApiRoute.RECEIPT_ROUTES.EDIT
        .Replace(ApiRoute.ARG_RECEIPT, _receiptTest.Id.ToString());
    var requestMessage = _testHelper.CreateRequestWithToken(url, _userToken, HttpMethod.Patch, editReceipt);

    // Act
    var editResponse = await _client.SendAsync(requestMessage);

    // Assert
    editResponse.EnsureSuccessStatusCode();
    using (var scope = _factory.Services.CreateScope())
    {
        var dbContext = scope.ServiceProvider.GetRequiredService<DivvyUpDBContext>();
        var receipt = dbContext.Receipts
            .Where(p => p.Id == _receiptTest.Id)
            .FirstOrDefault();

        var product = dbContext.Products
            .Where(p => p.Id == _productTest.Id)
            .FirstOrDefault();

        Assert.NotNull(receipt);
        Assert.NotNull(product);
        Assert.Equal(editReceipt.DiscountPercentage, receipt.DiscountPercentage);
        Assert.Equal(editReceipt.DiscountPercentage, product.DiscountPercentage);
    }
}
```

Rys. 7.7. Test integracyjny - testowanie zmiany wartości rabatu w rachunku.

Po zastosowaniu adnotacji *[Theory]* możliwe jest użycie *[InlineData]*, która umożliwia przekazywanie różnych argumentów w trakcie testu. Produkt może być dzielony na mniejsze części, a jego maksymalna liczba podziałów jest określana w momencie jego tworzenia (rys. 7.8). W ramach testu jako parametry przekazywane są zarówno dozwolona liczba podziałów, jak i wartość przekraczająca maksymalną liczbę podziałów zdefiniowaną dla produktu. Mechanizm powinien prawidłowo zareagować, informując użytkownika o niemożliwości wykonania operacji w przypadku przekroczenia limitu, a w przypadku wartości mieszczącej się w dopuszczalnym zakresie zakończyć działanie powodzeniem.

```
[Theory]
[InlineData(1, true)]
[InlineData(3, false)]
public async Task AddPersonProduct_QuantityValidation_ShouldBehaveAsExpected(int quantity, bool shouldSucceed)
{
    // Arrange
    var personId = quantity == 1 ? _personTest.Id : _personTest2.Id;
    var addRequest = new AddEditPersonProductDto { PersonId = personId, Quantity = quantity };
    var url = ApiRoute.PERSON_PRODUCT_ROUTES.ADD.Replace(ApiRoute.ARG_PRODUCT, _productTest.Id.ToString());
    var request = _testHelper.CreateRequestWithToken(url, _userToken, HttpMethod.Post, addRequest);

    // Act
    var response = await _client.SendAsync(request);

    // Assert
    if (shouldSucceed)
    {
        response.EnsureSuccessStatusCode();
    }
    else
    {
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        var responseContent = await response.Content.ReadAsStringAsync();
        Assert.Contains("Przekroczono maksymalną ilość produktu", responseContent);
    }
}
```

Rys. 7.8. Test integracyjny - testowanie walidacji liczby części w powiązaniu osoby a produktu.

Proste, ale niezwykle istotne testy są równie potrzebne. Przykładem takiego testu jest weryfikacja, czy produkt oznaczony jako niepodzielny ma przypisaną wartość maksymalnej podzielności inną niż domyślna wartość 1. Próba dodania takiego produktu powinna skutkować wyświetleniem komunikatu informującego o niemożliwości wykonania operacji (rys. 7.9).

```
[Fact]
public async Task AddProduct_WithInvalidMaxQuantity_ShouldReturnBadRequest()
{
    // Arrange
    var product = new AddEditProductDto() { Divisible = false, MaxQuantity = 2 };
    var url = ApiRoute.PRODUCT_ROUTES.ADD.Replace(ApiRoute.ARG_RECEIPT, _receiptTest.Id.ToString());
    var requestMessage = _testHelper.CreateRequestWithToken(url, _userToken, HttpMethod.Post, product);

    // Act
    var productResponse = await _client.SendAsync(requestMessage);

    // Assert
    Assert.Equal(HttpStatusCode.BadRequest, productResponse.StatusCode);
    var responseContent = await productResponse.Content.ReadAsStringAsync();
    Assert.Contains("Maksymalna ilość musi być równa 1 gdy produkt jest niepodzielny", responseContent);
}
```

Rys. 7.9. Test integracyjny - testowanie poprawności dodawanego produkt.

Przygotowano również warianty testów walidacyjnych, które sprawdzają zachowanie serwera w różnych sytuacjach. Jednym z takich przypadków jest próba zwrócenia przez użytkownika rachunku, który nie istnieje. W takiej sytuacji serwer powinien zwrócić błąd typu „*Nie znaleziono*” wraz z komunikatem „*Rachunek nie został znaleziony*” (rys. 7.10).

```
[Fact]
public async Task GetReceipt_WithValid_ShouldReturnNotFound()
{
    // Arrange
    int receiptId = -1;
    var url = ApiRoute.RECEIPT_ROUTES.RECEIPT
        .Replace(ApiRoute.ARG_RECEIPT, receiptId.ToString());
    var requestMessage = _testHelper.CreateRequestWithToken(url, _userToken, HttpMethod.Get);

    // Act
    var getReceiptResponse = await _client.SendAsync(requestMessage);

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, getReceiptResponse.StatusCode);
    var responseContent = await getReceiptResponse.Content.ReadAsStringAsync();
    Assert.Contains("Rachunek nie znaleziony", responseContent);
}
```

Rys. 7.10. Test integracyjny - testowanie pobranie nieistniejącego rachunku.

Proces zmiany stanu rozliczenia rachunku powinien automatycznie powodować zmianę statusu przypisanych do niego produktów na „*rozliczony*”. Test przedstawia scenariusz, w którym status rozliczenia jest modyfikowany poprzez wysłanie żądania *HTTP* przez klienta (rys. 7.11). W kolejnym kroku wyniki są weryfikowane poprzez sprawdzenie, czy stan rachunku oraz produktów został prawidłowo zaktualizowany w bazie danych.

```
[Fact]
public async Task SetSettledInReceipt_WithValid_ShouldSucceed()
{
    // Arrange
    bool settled = true;
    var url = ApiRoute.RECEIPT_ROUTES.SET_SETTLED
        .Replace(ApiRoute.ARG_RECEIPT, _receiptTest.Id.ToString())
        .Replace(ApiRoute.ARG_SETTLED, settled.ToString());
    var requestMessage = _testHelper.CreateRequestWithToken(url, _userToken, HttpMethod.Put);

    // Act
    var setSettledResponse = await _client.SendAsync(requestMessage);

    // Assert
    setSettledResponse.EnsureSuccessStatusCode();
    using (var scope = _factory.Services.CreateScope())
    {
        var dbContext = scope.ServiceProvider.GetRequiredService<DivvyUpDBContext>();
        var receipt = dbContext.Receipts
            .Where(p => p.Id == _receiptTest.Id)
            .FirstOrDefault();

        var product = dbContext.Products
            .Where(p => p.Id == _productTest.Id)
            .FirstOrDefault();

        Assert.NotNull(receipt);
        Assert.NotNull(product);
        Assert.Equal(true, receipt.Settled);
        Assert.Equal(true, product.Settled);
    }
}
```

Rys. 7.11. Test integracyjny - testowanie zmiany stanu rozliczenia rachunku.

8. Podsumowanie i wnioski

Opracowana praca dyplomowa miała na celu zaprojektowanie systemu, który posłuży jako wszechstronne narzędzie do rozliczania kosztów grupowych. Wszystkie wymagania funkcjonalne i niefunkcjonalne zostały zrealizowane. Opracowany system oferuje następujące funkcjonalności:

- Aplikacja posiada system autoryzacji użytkowników, co zapewnia integralność i bezpieczeństwo danych.
- Użytkownik może dodawać osoby, rachunki oraz produkty.
- Na produkty możliwe jest nałożenie procentowego rabatu, dodania dodatkowej opłaty oraz sprecyzowania ilości sztuk.
- Użytkownik zarządza przypisaniem wybranych osób do konkretnych produktów oraz określeniem ich udziału w podziale.
- System umożliwia kontrolowanie i aktualizację stanu rozliczenia dla poszczególnych elementów, takich jak produkty, rachunki, przypisy i pożyczki.
- Użytkownik może nakładać pożyczki na osoby przypisane do konta.
- System umożliwia modyfikację danych konta użytkownika.
- Dostępne są statystyki, które ułatwiają analizę rozliczeń i wydatków.

Projekt charakteryzuje się szerokimi możliwościami rozbudowy. W przyszłości system może zostać rozszerzony o dodatkowe funkcjonalności, takie jak mechanizm komunikacji między użytkownikami, dodawanie znajomych czy udostępnianie list zakupów. Dzięki elastycznej architekturze aplikacja może być również zaadaptowana do działania na różnych platformach, w tym na urządzeniach mobilnych. Obecna struktura projektu opiera się na modularnych, odizolowanych komponentach, co ułatwia jego modyfikację oraz integrację z innymi systemami.

Finalna wersja projektu spełnia wszystkie założone cele. Opracowany system umożliwia efektywne rozdzielenie kosztów pomiędzy osobami przypisanymi do konta użytkownika. Aplikacja została dokładnie przetestowana za pomocą różnorodnych testów oprogramowania, które potwierdziły jej niezawodność i stabilność działania.

Wykonana aplikacja znacząco upraszcza proces podziału kosztów, co jest szczególnie przydatne w sytuacjach takich jak wspólne wyjazdy, zakupy spożywcze czy zamówienia z restauracji. Dodatkowo właściciel konta ma pełny dostęp do statystyk i historii zakupów, co pozwala na łatwe monitorowanie wydatków i zarządzanie finansami.

9. Bibliografia

[1] “Historia Języka C#.” *Microsoft Learn*, https://learn.microsoft.com/pl-pl/dotnet/csharp/whats-new/csharp-version-history. Accessed 12 Dec. 2024.

[2] Price, Mark J. *C# 10 and . NET 6 - Modern Cross-Platform Development - Sixth Edition: Build Apps, Websites, and Services with ASP. NET Core 6, Blazor, and EF Core 6 Using Visual Studio 2022 and Visual Studio Code*. Packt Publishing, 2021.

[3] “Blazor University - What Is Blazor?” *Blazor University*, https://blazor-university.com/overview/what-is-blazor.

Accessed 28 Nov. 2024.

[4] Gourley, David, et al. *HTTP: The Definitive Guide: The Definitive Guide*. “O’Reilly Media, Inc.,” 2002.

[5] “Working with JSON - Learn Web Development.” *MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON.

Accessed 28 Nov. 2024.

[6] Smith, Jon. *Entity Framework Core in Action*. Simon and Schuster, 2018.

[7] “Co to Jest .NET MAUI? - .NET MAUI.” *Microsoft Learn*, https://learn.microsoft.com/pl-pl/dotnet/maui/what-is-maui?view=net-maui-8.0.

Accessed 28 Nov. 2024.

[8] Prager, Mike. “Rapid Application Development for the Web.” *Radzen.Com*, https://www.radzen.com/.

Accessed 28 Nov. 2024.

[9] “JWT.IO - JSON Web Tokens Introduction.” *Auth0*,

https://jwt.io/introduction. Accessed 28 Nov. 2024

[10] Obe, Regina O., et al. *PostgreSQL: Up and Running*. “O’Reilly Media, Inc.,” 2012.

[11] “Unit Testing C# Code in .NET Using Dotnet Test and xUnit - .NET.” *Microsoft Learn*, https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test. Accessed 28 Nov. 2024.

[12] “Co to Jest Usługa Git? - Azure DevOps.” *Microsoft Learn*, https://learn.microsoft.com/pl-pl/devops/develop/git/what-is-git. Accessed 12 Dec. 2024.

[13] *IBM Integration Bus 10.0.0*. https://www.ibm.com/docs/pl/integration-bus/10.0?topic=apis-swagger. Accessed 28 Nov. 2024.

[14] “Unit Testing vs. Integration Testing.” *TestDevLab Blog*, https://www.testdevlab.com/blog/unit-testing-vs-integration-testing. Accessed 12 Dec. 2024.

[15] “The Arrange, Act, and Assert (AAA) Pattern in Unit Test Automation.” *Semaphore*, 2 Oct. 2024, https://semaphoreci.com/blog/aaa-pattern-test-automation. . Accessed 12 Dec. 2024.

[16] *Flaticon*, https://www.flaticon.com/. Accessed 5 Jan. 2025.

[17] *What Is Unified Modeling Language (UML)?* https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml Accessed 19 Jan. 2025.

[18] Riordan, Grant. “What Is Mocking? Mocking in .NET Explained With Examples.” https://www.freecodecamp.org/news/explore-mocking-in-net/. Accessed 22 Jan. 2025.

10. Spis tabel

11. Spis wzorów

12. Spis rysunków