Aplikacja do zarządzania budżetem domowym

Jakub Frydrych
Szymon Ściegienka
Daniel Prokopowicz
Wojciech Pilch
Michał Glimos

Specyfikacja wstępna systemu

Wprowadzenie

Cel i zakres projektu

Celem projektu jest opracowanie bezpiecznej, desktopowej aplikacji do zarządzania budżetem domowym, działającej w trybie offline. System ma wspierać użytkownika w kontrolowaniu jego finansów osobistych poprzez ewidencjonowanie przychodów i wydatków, tworzenie miesięcznych budżetów oraz generowanie czytelnych raportów i wykresów finansowych.

Interesariusze projektu

- Użytkownik,
- Zespół projektowy,
- Opiekun projektu,
- Testerzy.

Korzyści z wdrożenia

- Zwiększona kontrola nad wydatkami i planowanie finansów osobistych,
- Możliwość planowania budżetu domowego,
- Pełna prywatność danych, ponieważ system działa lokalnie,
- Czytelna wizualizacja danych finansowych w formie wykresów i raportów.

Zakres projektu:

- Tworzenie i zarządzanie kontami finansowymi,
- Dodawanie, edytowanie i usuwanie transakcji,
- Przypisywanie transakcji do kategorii,
- Definiowanie miesięcznych budżetów,
- Generowanie raportów i wykresów,
- Zapisywanie danych w lokalnej bazie danych,
- Możliwość zabezpieczenia aplikacji hasłem lub PIN-em.

Poza zakresem:

- synchronizacja z bankami lub chmurą,
- obsługa wielu urządzeń jednocześnie,
- płatności online i integrację z serwisami zewnętrznymi.

Analiza dziedziny:

Na rynku istnieje wiele aplikacji do zarządzania finansami, zarówno mobilnych, jak i desktopowych. Najpopularniejsze z nich to np. **HomeBank, GnuCash** czy różne aplikacje mobilne jak **Money Manager** lub **Splitwise**. Każde z tych rozwiązań ma swoje zalety i wady:

Aplikacje mobilne są wygodne, ale często wymagają połączenia z internetem i przechowują dane w chmurze, co obniża poziom prywatności. Aplikacje webowe oferują nowoczesny interfejs, ale czasem są zależne od przeglądarki. Programy desktopowe działające lokalnie są szybkie i niezależne, lecz często mają mniej nowoczesny wygląd.

Projektowana aplikacja ma łączyć zalety podejścia desktopowego (szybkość i prywatność) z wygodnym, przejrzystym interfejsem i intuicyjną obsługą. Dzięki temu użytkownik będzie miał w pełni funkcjonalne narzędzie do zarządzania finansami bez konieczności korzystania z internetu.

Opis i wizja systemu

Ogólny opis produktu:

Projektowany system to **aplikacja desktopowa** działająca w trybie **offline**, której zadaniem jest kompleksowe wspomaganie użytkownika w zarządzaniu finansami osobistymi. Po uruchomieniu aplikacji użytkownik zobaczy pulpit, na którym znajdą się podsumowania budżetu, salda kont i najnowsze transakcje. Aplikacja umożliwia rejestrowanie przychodów i wydatków, przypisywanie ich do odpowiednich kategorii, ustalanie budżetów miesięcznych oraz analizowanie struktury wydatków za pomocą raportów i wykresów.

System ma być **intuicyjny, bezpieczny i niezależny od połączenia internetowego**.
Dane użytkownika przechowywane są w lokalnej, zaszyfrowanej bazie danych, do której dostęp chroniony jest hasłem głównym. Dzięki temu aplikacja zapewnia pełną prywatność finansowych informacji użytkownika.

Z punktu widzenia użytkownika końcowego, aplikacja stanowi **osobisty dziennik finansowy** - prosty w obsłudze, ale wystarczająco rozbudowany, by umożliwić analizę i planowanie budżetu domowego w dłuższym okresie.

Aktorzy systemu:

- **Użytkownik** – korzysta z aplikacji w celu zarządzania finansami domowymi.

Wymagania Systemowe

Wymagania funkcjonalne

- **Zarządzanie kontami** - użytkownik może dodawać, edytować i usuwać konta finansowe (np. gotówka, konto bankowe, karta).
- **Rejestrowanie transakcji** - możliwość wprowadzania transakcji z informacjami o dacie, kwocie, kategorii i opisie.
- **Kategorie** - definiowanie kategorii wydatków i przychodów (np. żywność, transport, pensja).
- **Budżety miesięczne** - ustalanie limitów wydatków dla poszczególnych kategorii lub całego miesiąca.
- **Raporty i wykresy** - generowanie zestawień finansowych i wizualizacji danych w formie wykresów kołowych i słupkowych.
- **Import i eksport danych** - możliwość zapisu i odczytu danych z plików CSV oraz eksport raportów.
- **Autoryzacja użytkownika** - logowanie do aplikacji przy użyciu PIN-u lub hasła.
- **Kopie zapasowe** - automatyczne tworzenie kopii danych lokalnie.
- **Filtrowanie i wyszukiwanie** - wyszukiwanie transakcji według daty, kategorii lub opisu.
- **Działanie offline** - aplikacja ma działać bez połączenia z internetem.

Wymagania pozafunkcjonalne

- **Wydajność** - aplikacja powinna działać płynnie przy dużej liczbie danych.
- **Bezpieczeństwo** - dane są przechowywane lokalnie w zaszyfrowanym pliku; dostęp chroniony PIN-em lub hasłem.
- **Niezawodność** - system powinien automatycznie tworzyć kopię bazy danych przy zamknięciu aplikacji.
- **Użyteczność** - interfejs ma być prosty, czytelny i intuicyjny; kluczowe funkcje dostępne z poziomu głównego okna.
- **Rozszerzalność** - kod powinien być napisany w sposób modularny, umożliwiający łatwe dodawanie nowych funkcji.

Diagramy

Diagram przypadków użycia

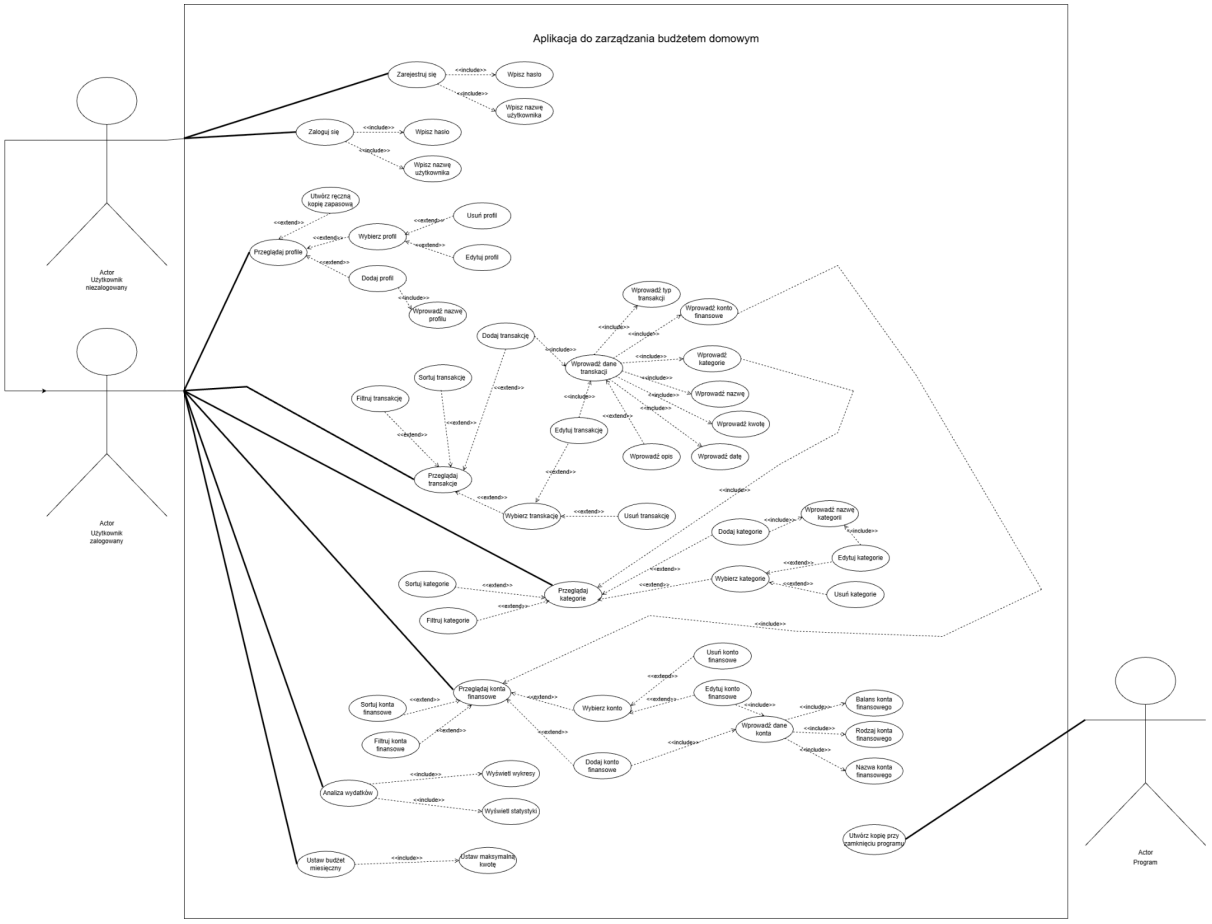
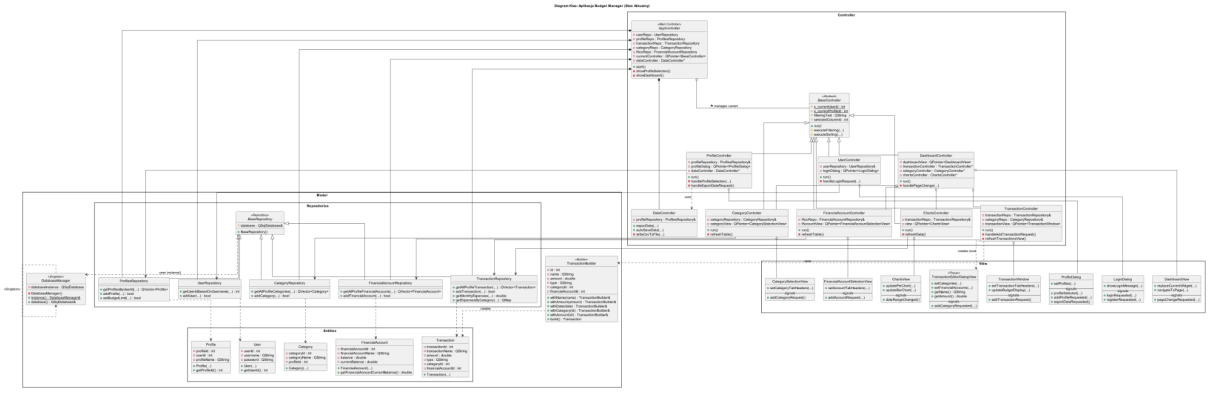
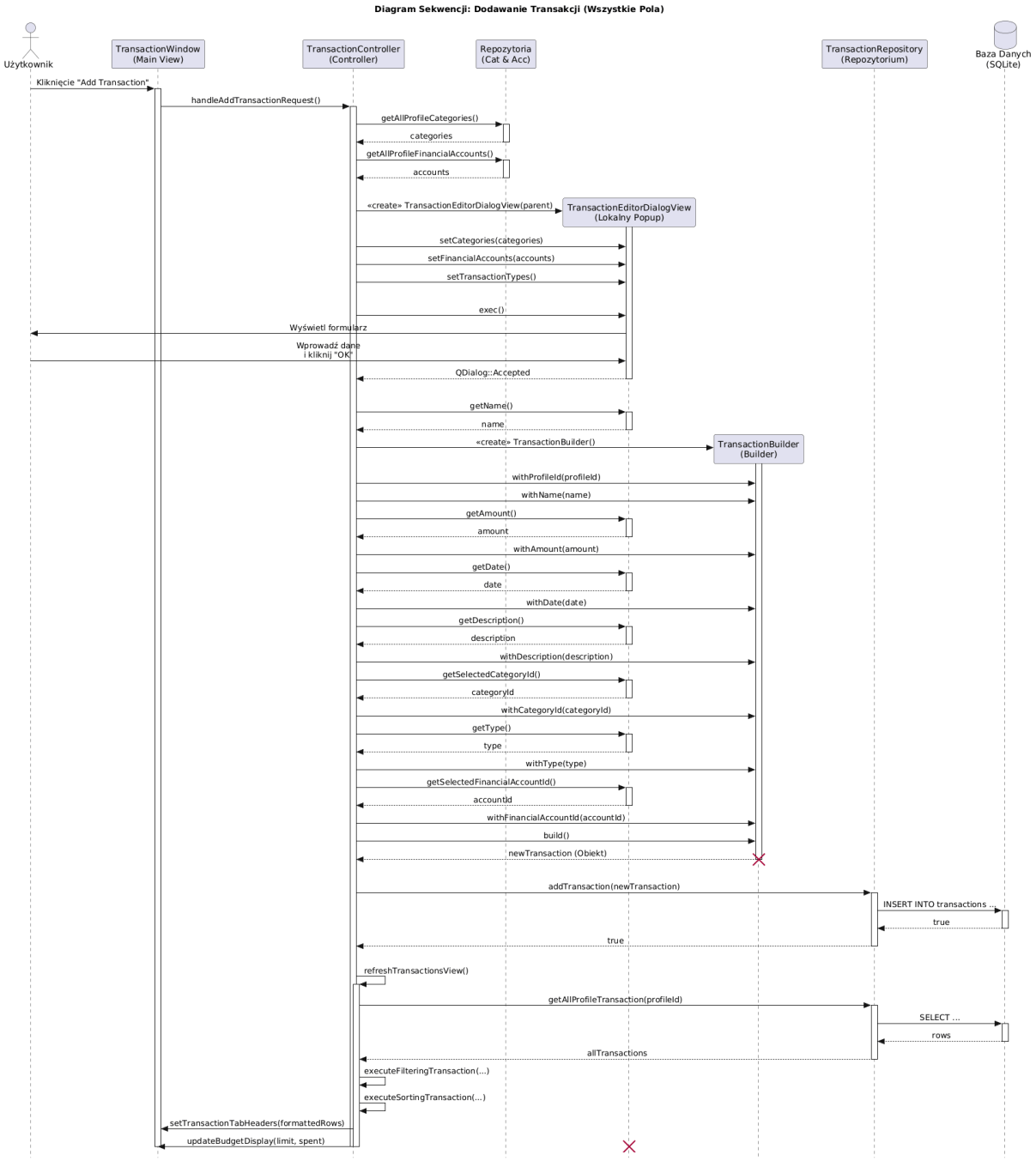


Diagram klas

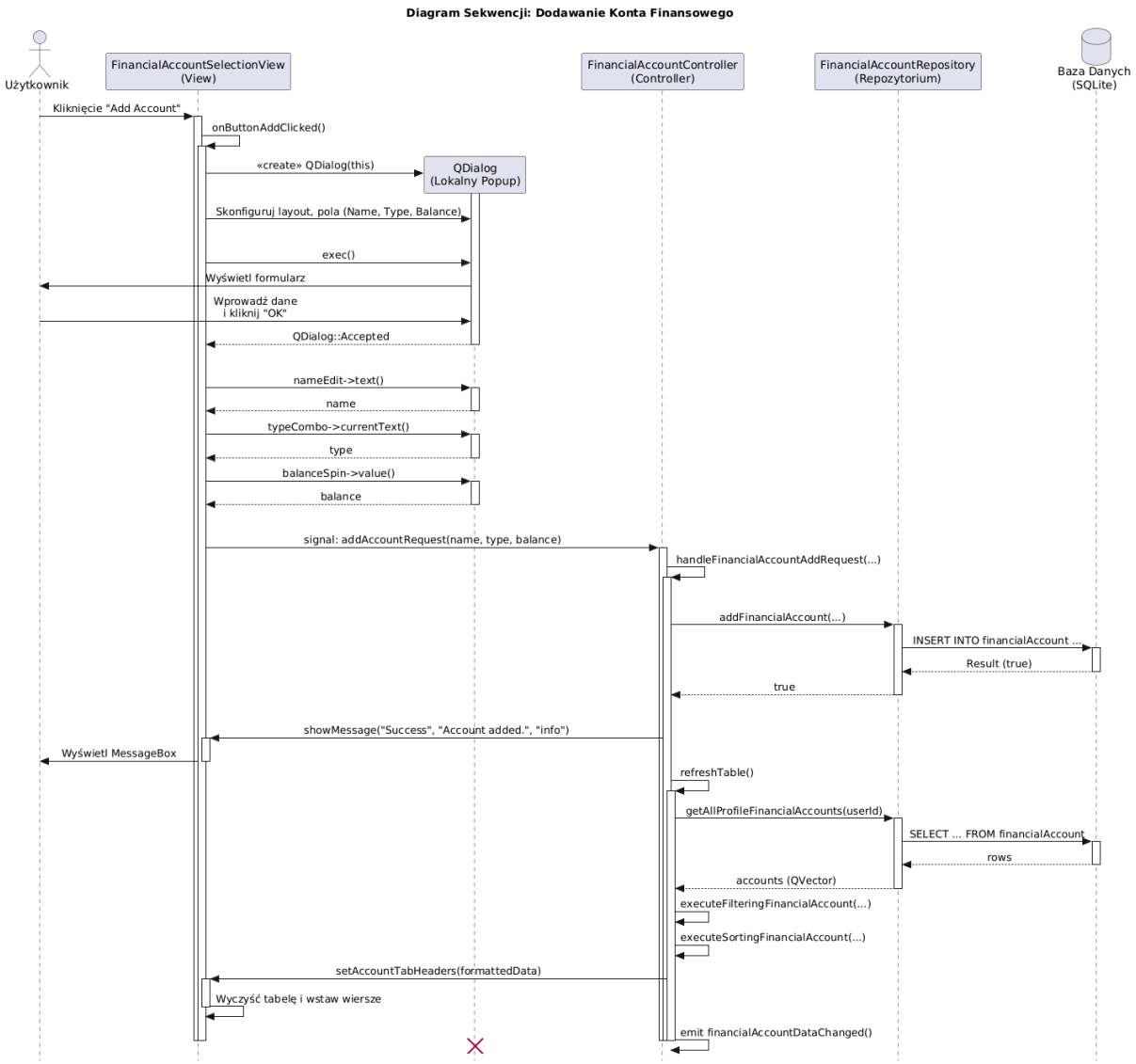


Diagramy sekwencji

a) Dodawanie transakcji



b) Dodawanie konta finansowego

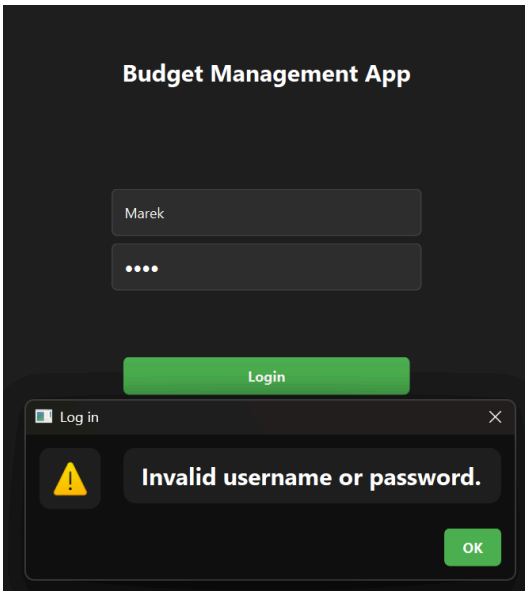


Testowanie

W projekcie przyjęto strategię manualnych testów systemowych i funkcjonalnych, koncentrujących się na weryfikacji poprawności działania interfejsu użytkownika oraz na poprawności przepływu danych między interfejsem użytkownika a bazą danych SQLite.

Uwierzytelnianie i bezpieczeństwo:
Scenariusz 1: Próba logowania na nieistniejące konto
Cel: Sprawdzenie mechanizmu weryfikacji tożsamości.
Typ Testu: Negatywny
Przebieg: Wpisanie loginu użytkownika, który nie istnieje w bazie, a następnie kliknięcie “login”.
Oczekiwany rezultat: Aplikacja blokuje dostęp i wyświetla komunikat "Invalid username or password"

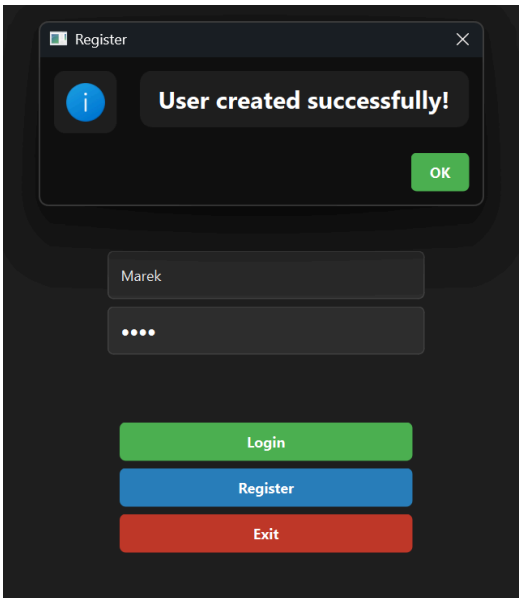
Weryfikacja:



Logika w kodzie:

```
int userId = userRepository.getUserIdBasedOnUsername(username, password);
if (userId < 0) {
    const QString header = tr("Log in");
    const QString message = tr("Invalid username or password.");
    loginDialog->showLoginMessage(header, message, "error");
    return;
}
```

Scenariusz 2: Rejestracja nowego użytkownika i hashowanie haseł
Cel: Weryfikacja poprawności zapisu nowego użytkownika z bezpiecznym przechowywaniem hasła.
Typ testu: Bezpieczeństwa.
Przebieg: Wpisanie nowej nazwy użytkownika i hasła, a następnie kliknięto "Register".
Oczekiwany rezultat: Komunikat "User created successfully". W bazie danych hasło nie jest zapisane jawnym tekstem, lecz jako hash.
Weryfikacja:



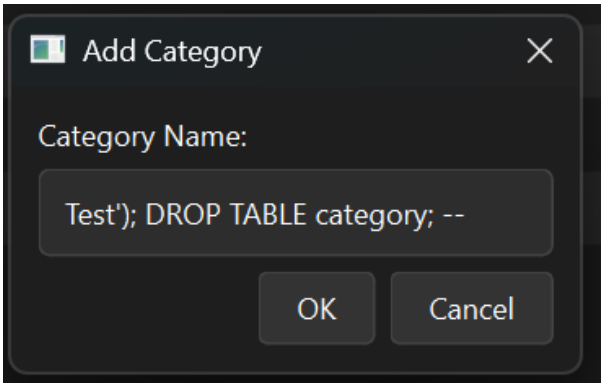
Logika w kodzie:

```
QString salt = QUuid::createUuid().toString();

QByteArray dataToHash = (password + salt).toUtf8();
QString hashedPassword = QString(QCryptographicHash::hash(dataToHash, QCryptographicHash::Sha256).toHex());
```

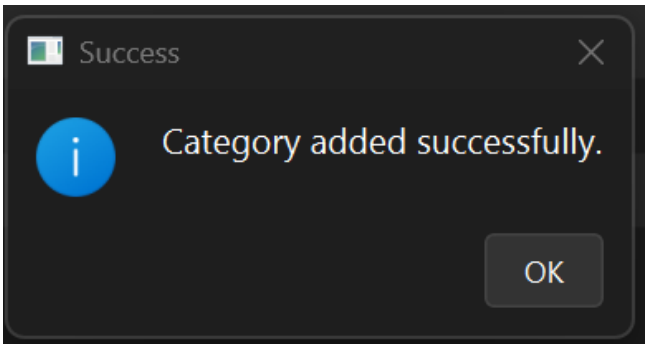
	id	username	password_hash	salt
	Filtr	Filtr	Filtr	Filtr
1	1	admin	5855a5056dca7b9d2ca98cf9f5de5dce498b...	{8939d42b-065e-4d45-9d47-6a4025986bc...
2	3	Marek	1f3a97d360fc356b3b0b2d633ba042caba2a...	{ff2577bc-af4d-4bc1-a3f6-...

Nie da się rozczytać hasła w bazie danych

Scenariusz 3: Test podatności na SQL Injection
Cel: Weryfikacja, czy wpisanie złośliwej komendy SQL w polu tekstowym spowoduje wykonanie jej przez bazę danych.
Typ testu: Bezpieczeństwa.
Przebieg testu: Uruchomiono aplikację i zalogowano się. Kliknięto w zakładkę Categories, a następnie “Add Category”. W polu nazwy nowej kategorii wpisano ciąg znaków:
Test'); DROP TABLE category; --
Kliknięto przycisk “OK”.
Oczekiwany rezultat:
Aplikacja NIE usuwa tabeli, a system traktuje wpisany ciąg znaków dosłownie i pojawia się nowa kategoria o nazwie: Test'); DROP TABLE category; --
Dowody weryfikacji:



“Test” to początek nazwy, “')” to próba zamknięcia cudzysłowu SQL i zakończenia polecenia INSERT, “DROP TABLE category” to polecenie usuwające tabelę, “--” to znak komentarza by zignorować dalszą część zapytania



Została utworzona kategoria o podanej nazwie

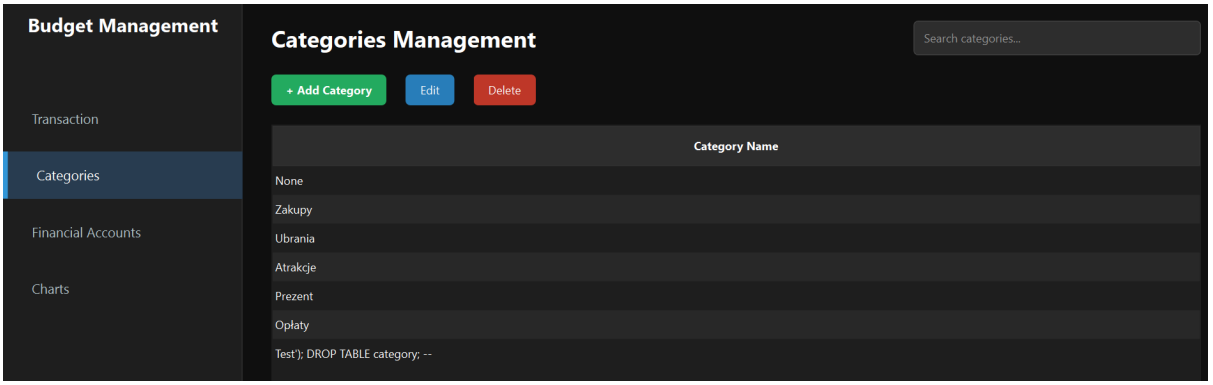


Tabela Categories nie została usunięta i na liście jest kategoria o podanej nazwie

Logika w kodzie:

```
bool CategoryRepository::addCategory(const QString& categoryName, int profileId) const
{
    QSqlQuery query(database);

    query.prepare("INSERT INTO category (category_name, profile_id) VALUES (:name, :profile_id)");
    query.bindValue(":name", categoryName);
    query.bindValue(":profile_id", profileId);

    if (!query.exec())
    {
        qDebug() << "CategoryRepository:: error: Couldn't add category to database" << query.lastError().text();
        return false;
    }

    return true;
}
```

Podczas bindowania polecenie zostaje zamienione na zwykły tekst

Zarządzanie profilami i kontami:

Scenariusz 4: Próba dodania profilu bez nazwy

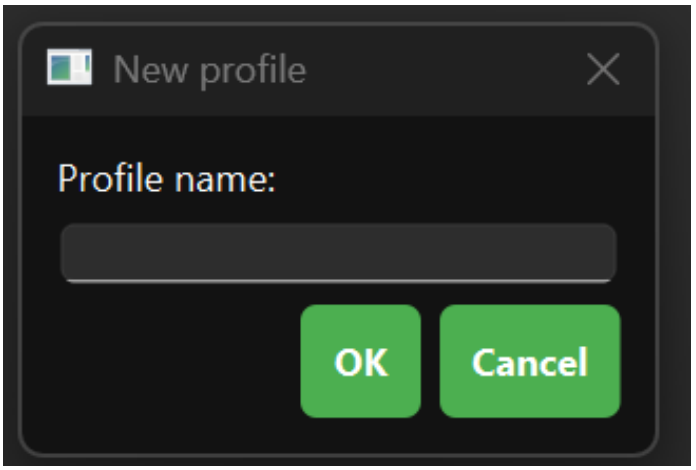
Cel: Sprawdzenie odporności na puste dane wejściowe.

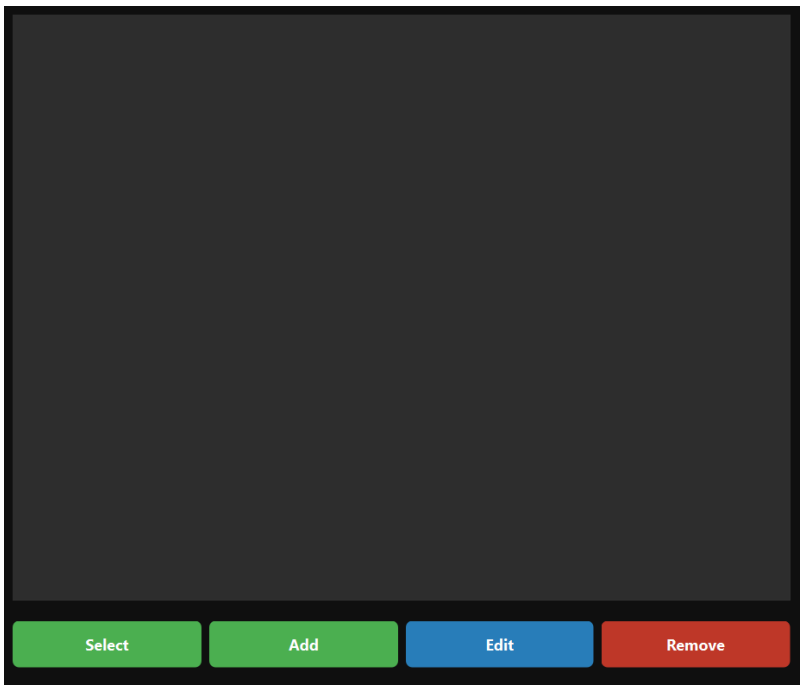
Typ testu: Walidacyjny.

Przebieg: Zalogowanie się, kliknięcie "Add Profile" i pozostawienie pola nazwy pustego i zatwierdzenie.

Oczekiwany rezultat: System nie dodaje profilu lub nie pozwala zatwierdzić pustego pola.

Dowody weryfikacji:





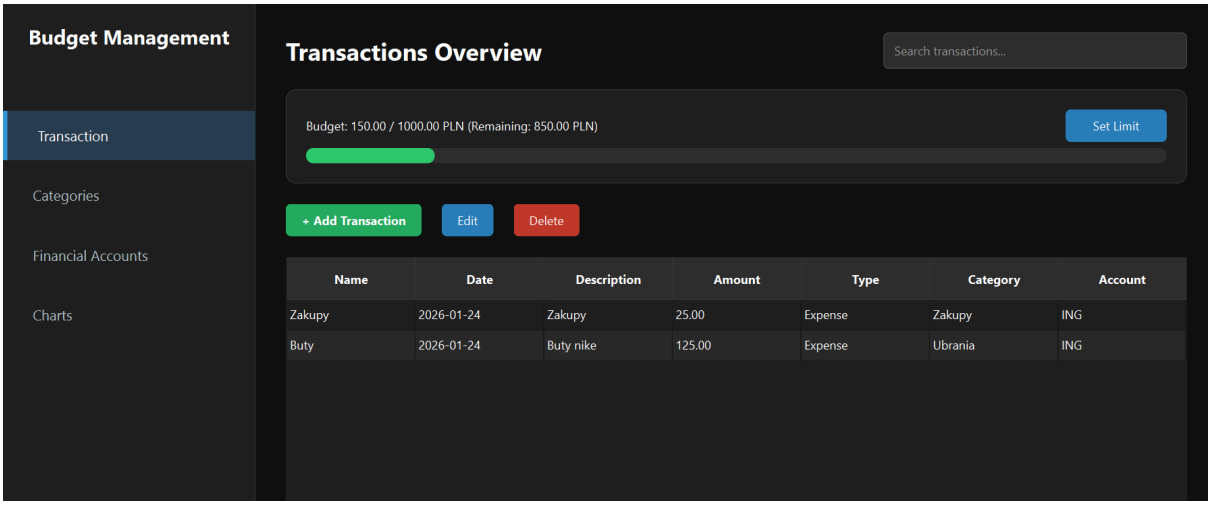
Pusta lista profili

Logika w kodzie:

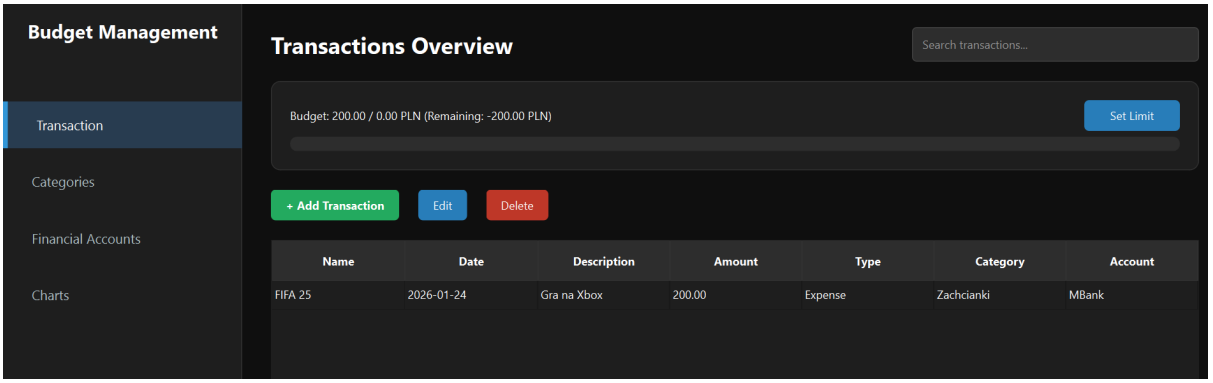
```
if (!ok || name.trimmed().isEmpty())
    return;

emit addProfileRequested(name);
```

Sygnał nie jest emitowany, jeśli nazwa jest pusta.

Scenariusz 5: Izolacja danych między profilami
Cel: Sprawdzenie, czy dane transakcyjne jednego profilu nie są widoczne na innym.
Typ testu: Integracyjny.
Przebieg: Zalogowanie na "Profil A" i sprawdzenie listy transakcji, a następnie przełączenie profilu i zalogowanie na "Profil B".
Oczekiwany rezultat: "Profil B" widzi tylko swoje transakcje i konta.
Dowody weryfikacji:



Profil A



Profil B

Logika w kodzie:

```
query.prepare(
    "SELECT id, name, date, description, amount, type, category_id, financialAccount_id, profile_id "
    "FROM transactions WHERE profile_id = :profileId"
);
query.bindValue(":profileId", profileId);
```

Zapytanie SQL zawsze filtruje po profile_id.

Logika Biznesowa:

Scenariusz 6: Integracja Transakcji z Budżetem

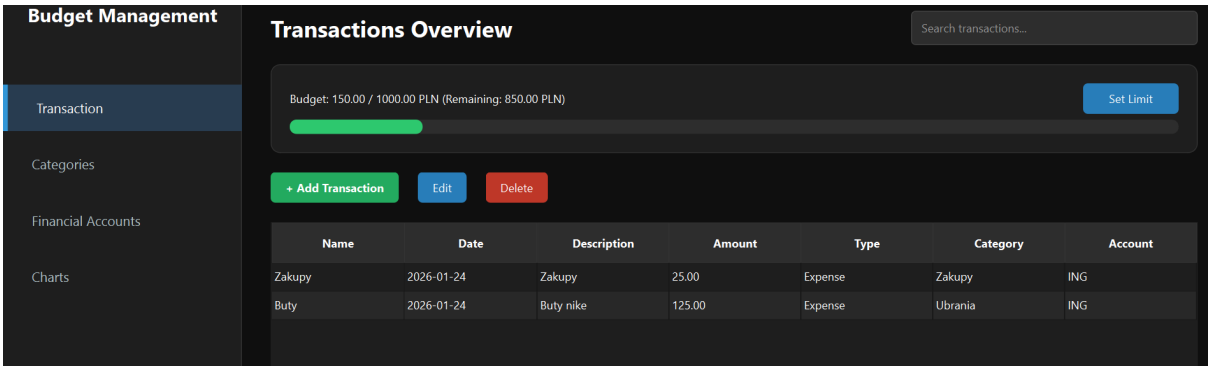
Cel: Sprawdzenie, czy dodanie wydatku aktualizuje wskaźnik zużycia budżetu.

Typ testu: Integracyjny.

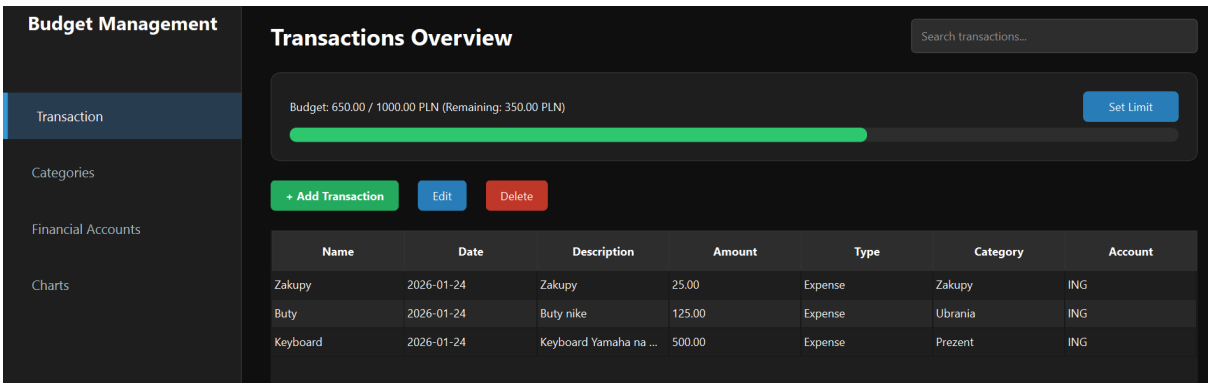
Przebieg: Sprawdzenie obecnego stanu paska budżetu, a następnie dodanie transakcji typu "Expense" na dużą kwotę.

Oczekiwany rezultat: Pasek postępu zwiększa się, a kwota "Remaining" maleje.

Dowody weryfikacji:



Przed dodaniem transakcji

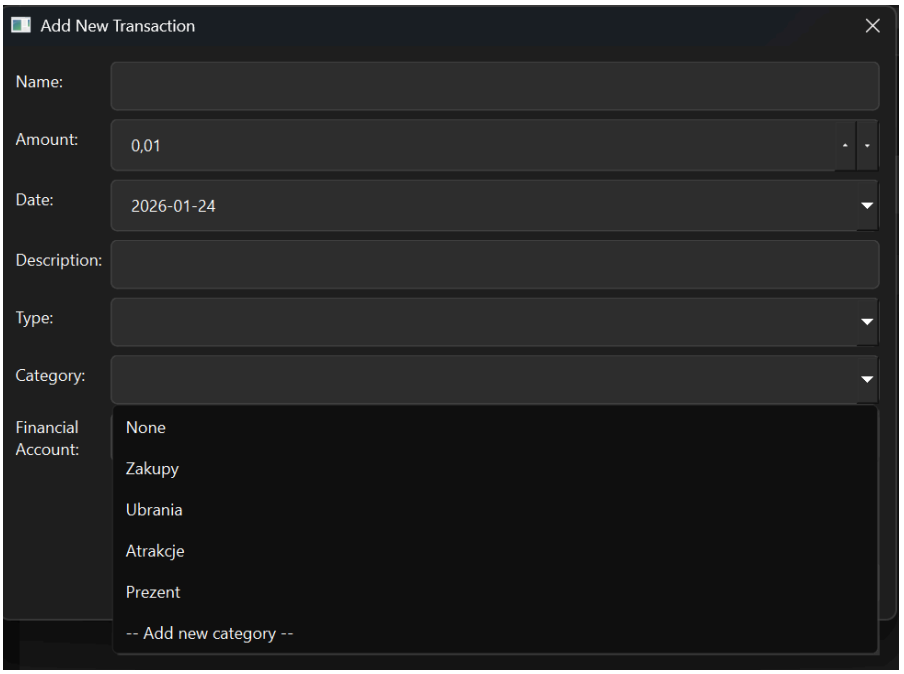


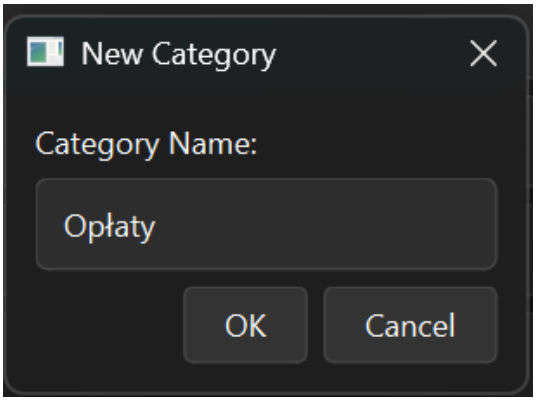
Po dodaniu transakcji

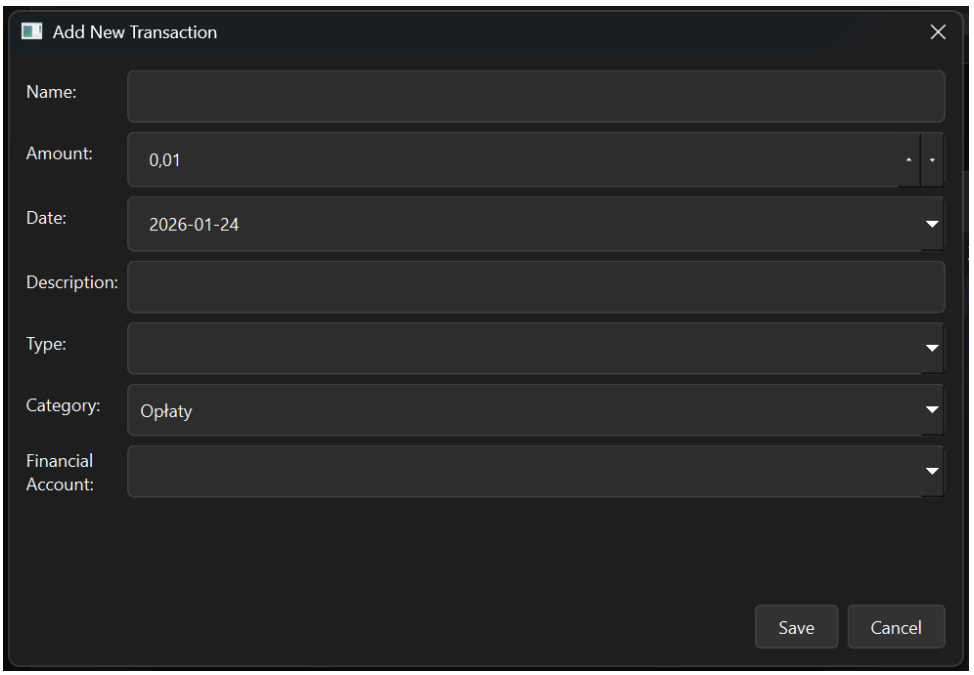
Logika w kodzie:

```
double monthlySpent = transactionRepository.getMonthlyExpenses(getProfileId(), current.month(), current.year());

transactionView->updateBudgetDisplay(budgetLimit, monthlySpent);
```

Scenariusz 7: Dodawanie kategorii podczas tworzenia transakcji
Cel: Weryfikacja funkcjonalności szybkiego dodawania.
Typ testu: Funkcjonalny.
Przebieg: Otwarcie okna dodawania transakcji, a następnie w liście rozwijanej kategorii wybranie opcji “Add new category”. Wpisanie nazwy nowej kategorii w oknie dialogowym.
Oczekiwany rezultat: Nowa kategoria zostaje dodana do bazy i automatycznie wybrana w formularzu transakcji.
Dowody weryfikacji:







Kategoria automatycznie dodana w formularzu transakcji

Logika w kodzie:

```
connect(&dialog, &TransactionEditorDialogView::addCategoryRequested, this,
    [&](const QString& name) {
        if (categoryRepository.addCategory(name, getProfileId())) {

            QVector<Category> newCats = categoryRepository.getAllProfileCategories(getProfileId());

            int newId = -1;
            for (const auto& c : newCats) if (c.getCategoryName() == name) newId = c.getCategoryId();

            dialog.refreshCategories(newCats, newId);
        }
        else {
            transactionView->showTransactionMessage("Error", "Failed to add category", "error");
        }
    });
```

Przetwarzanie i trwałość danych:

Scenariusz 8: Filtrowanie i Sortowanie Transakcji

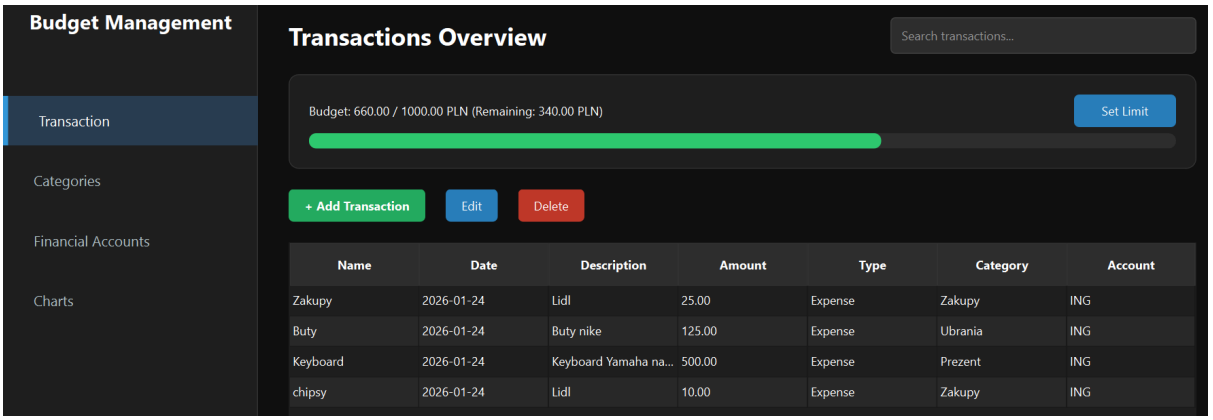
Cel: Weryfikacja działania algorytmów na danych w pamięci.

Typ testu: Funkcjonalny.

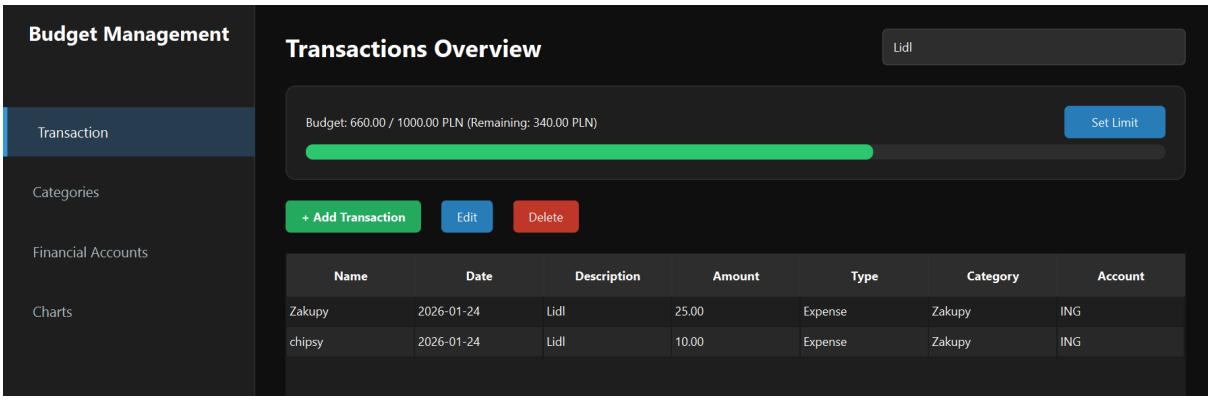
Przebieg: Wpisanie frazy (np. "Lidl") w pasku wyszukiwania. Kliknięcie nagłówka kolumny "Amount".

Oczekiwany rezultat: Lista transakcji zostaje zawężona do pasujących wyników i posortowana kwotowo.

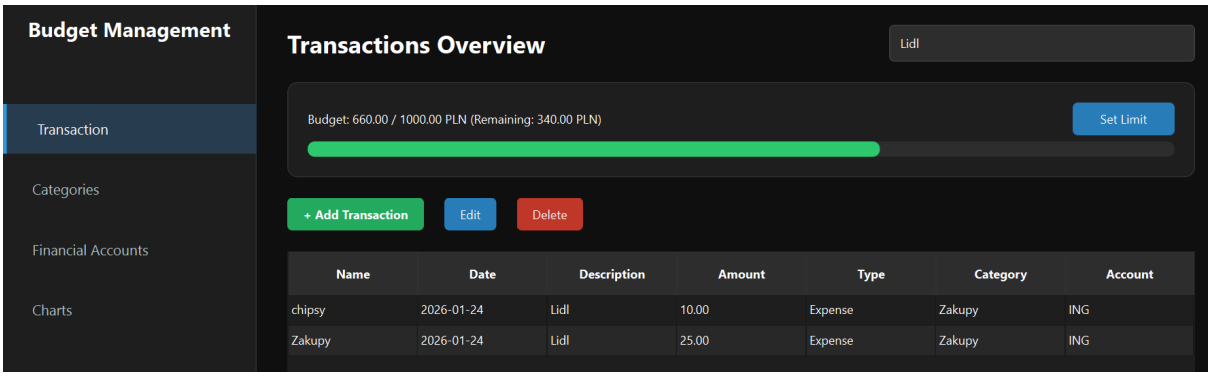
Dowody weryfikacji:



Wszystkie transakcje



Transakcje po wpisaniu frazy “Lidl”



Transakcje po kliknięciu nagłówka kolumny “Amount”

Logika w kodzie:

```cpp
template<typename T, typename P>
QVector<T> executeFiltering(const QVector<T>& allItems, P match) {
    if (getFilteringText().isEmpty()) {
        return allItems;
    }

    QVector<T> filteredItems;
    std::copy_if(allItems.begin(), allItems.end(), std::back_inserter(filteredItems), match);
    return filteredItems;
}
```

Wykorzystano algorytm do skopiowania tylko pasujących elementów

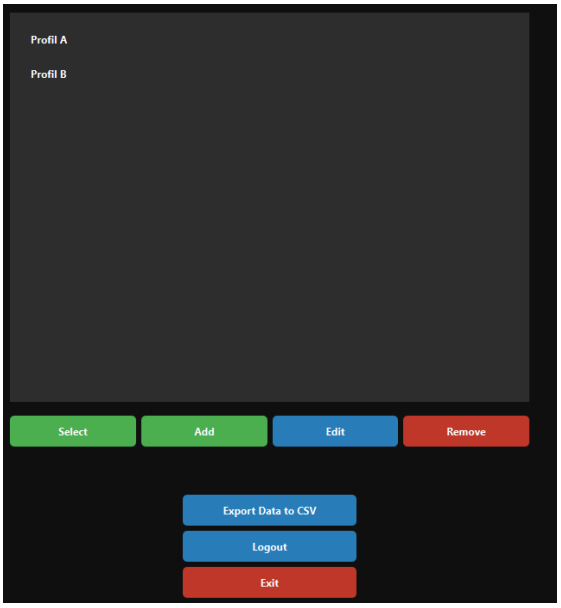
```cpp
QVector<Transaction> TransactionController::executeFilteringTransaction(const QVector<Transaction> allTransactions)
{
    return executeFiltering(allTransactions, [this](const Transaction& t) {
        QString filter = getFilteringText();

        bool nameMatches = t.getTransactionName().contains(filter, Qt::CaseInsensitive);
        bool descriptionMatches = t.getTransactionDescription().contains(filter, Qt::CaseInsensitive);
        bool typeMatches = t.getTransactionType().contains(filter, Qt::CaseInsensitive);

        bool categoryMatches = categoryRepository.getCategoryNameById(t.getCategoryId()).contains(filter, Qt::CaseInsensitive);
        bool financialAccountMatches = financialAccountRepository.getFinancialAccountNameById(t.getFinancialAccountId()).contains(filter, Qt::CaseInsensitive);

        bool dateMatches = t.getTransactionDate().toString("yyyy-MM-dd").contains(filter);

        return nameMatches || descriptionMatches || categoryMatches ||
            financialAccountMatches || typeMatches || dateMatches;
        });
}
```

Sprawdzanie dopasowania w wybranych kolumnach i zwracanie true jeśli pasuje chociaż jedno pole

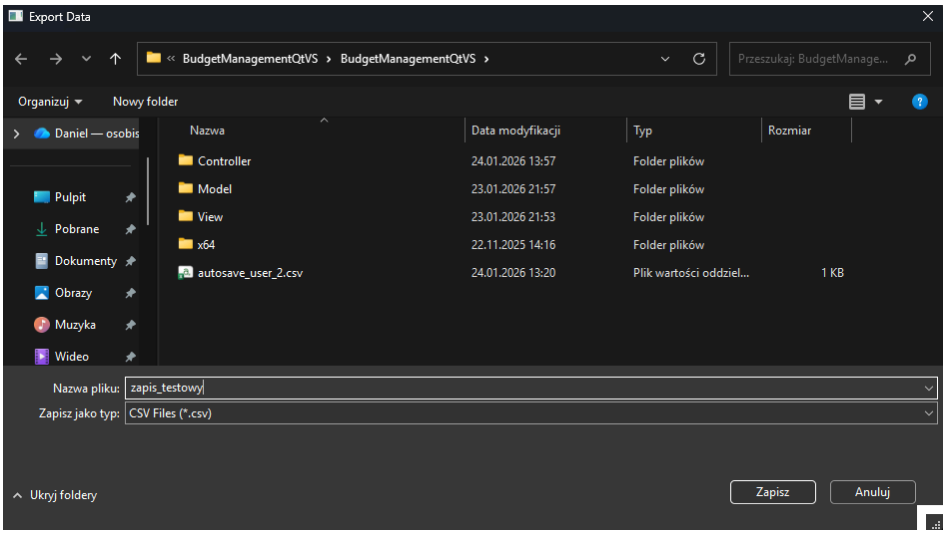
```cpp
void TransactionController::executeSortingTransaction(QVector<Transaction>& allTransactions)
{
    executeSorting(allTransactions, [this](const Transaction& a, const Transaction& b) {
        switch (getSelectedColumnId()) {
        case 1:
            return a.getTransactionName().toLower() < b.getTransactionName().toLower();
        case 2:
            return a.getTransactionDate() < b.getTransactionDate();
        case 4:
            return a.getTransactionAmount() < b.getTransactionAmount();
        case 6:
            return categoryRepository.getCategoryNameById(a.getCategoryId()).toLower() <
                categoryRepository.getCategoryNameById(b.getCategoryId()).toLower();
        case 7:
            return financialAccountRepository.getFinancialAccountNameById(a.getFinancialAccountId()).toLower() <
                financialAccountRepository.getFinancialAccountNameById(b.getFinancialAccountId()).toLower();
        default:
            return a.getTransactionId() < b.getTransactionId();
        }
        });
}
```

Implementacja sortowania w zależności od klikniętej kolumny

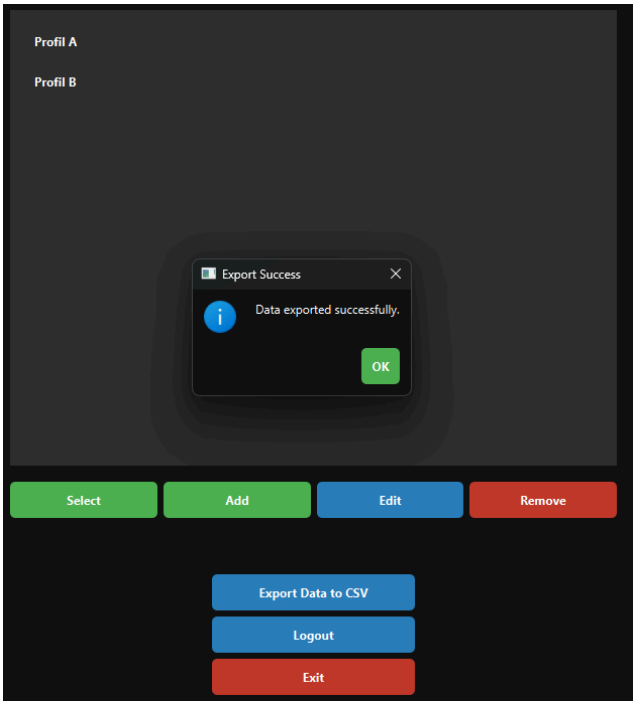
Scenariusz 9: Eksport danych do pliku CSV
Cel: Weryfikacja możliwości wyciągnięcia danych poza system.
Typ testu: Systemowy.
Przebieg: W oknie wyboru profilu kliknięcie przycisku "Export Data to CSV", a następnie wskazanie lokalizacji pliku na dysku.
Oczekiwany rezultat: Utworzenie pliku .csv, który zawiera poprawnie sformatowane dane wszystkich transakcji użytkownika.
Dowody weryfikacji:



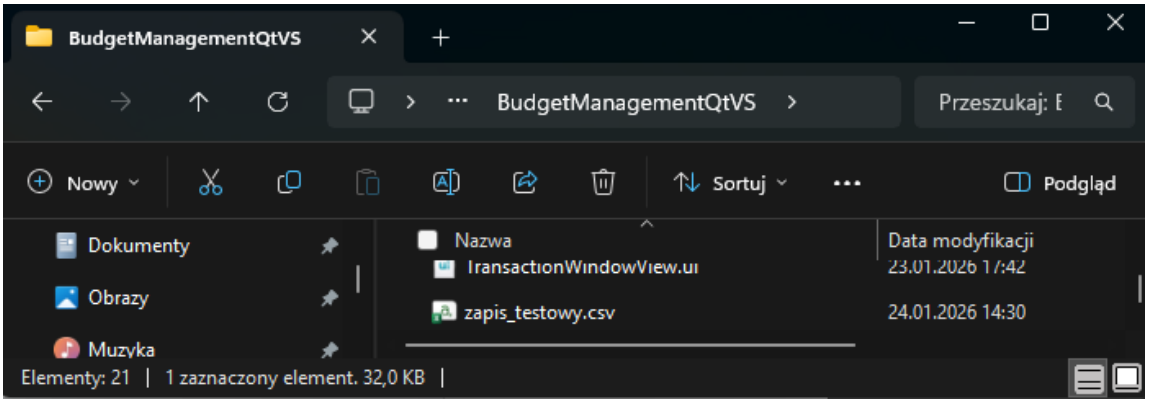
Okno wyboru profilu



Zapis pliku .csv o nazwie “zapis_testowy” w folderze projektu



Komunikat o sukcesie eksportu danych



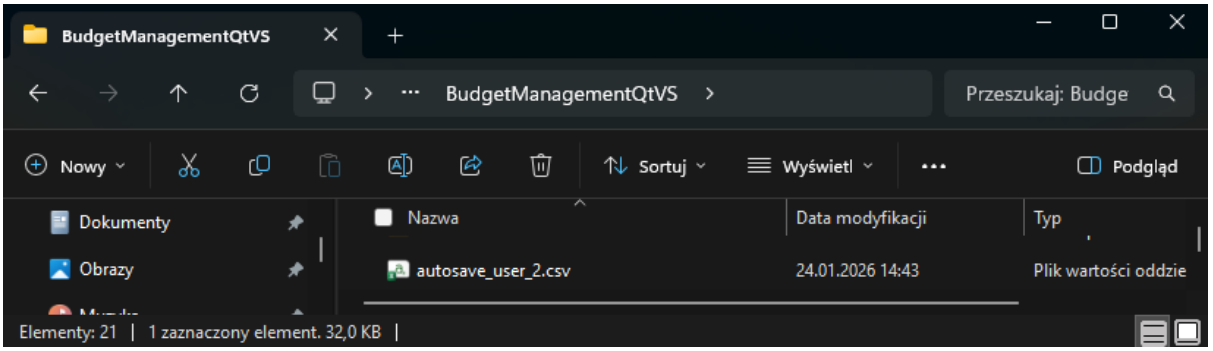
Sprawdzenie czy plik został utworzony

Logika w kodzie:

```
QTextStream out(&file);
out << "Profile,Transaction ID,Name,Date,Description,Amount,Type,Category,Account,Account Type\n";
```

Następstwem po tym jest pętla po wszystkich pętlach i transakcjach

Scenariusz 10: Auto-zapis przy zamykaniu aplikacji
Cel: Weryfikacja mechanizmu zapobiegającego utracie danych.
Typ testu: Integracyjny.
Przebieg: Zalogowanie i praca w aplikacji, następnie kliknięcie "X" (zamknięcie okna).
Oczekiwany rezultat: W folderze aplikacji pojawia się plik autosave_user_X.csv.
Dowody weryfikacji:

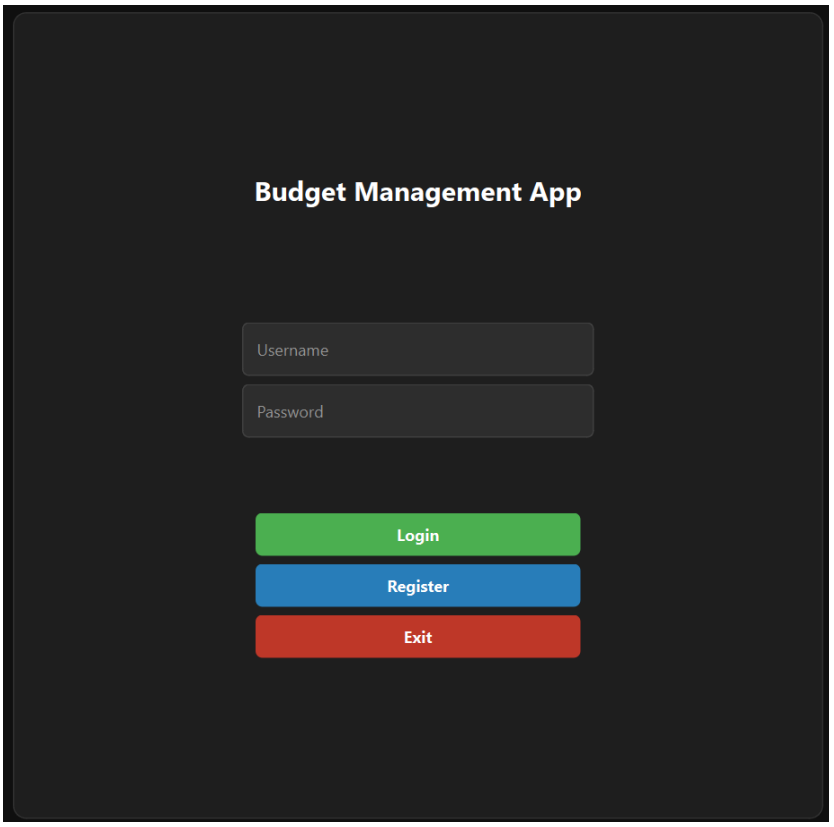


Logika w kodzie:

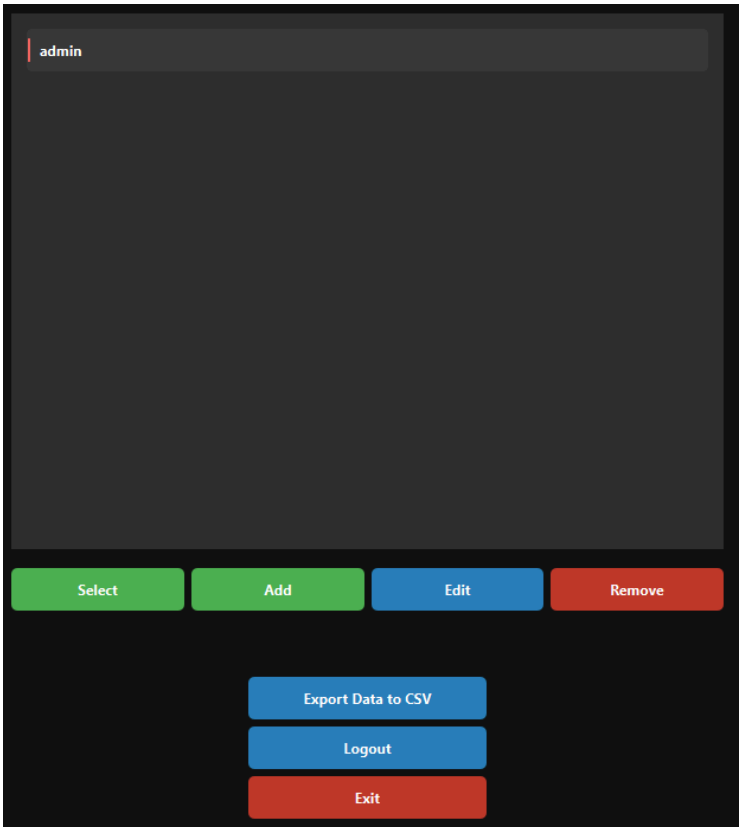
```
void AppController::onAppAboutToQuit()
{
    int currentUserId = BaseController::getUserId();

    if (currentUserId != -1) {
        if (dataController) {
            dataController->autoSaveData(currentUserId);
        }
    }
}
```

Działanie aplikacji

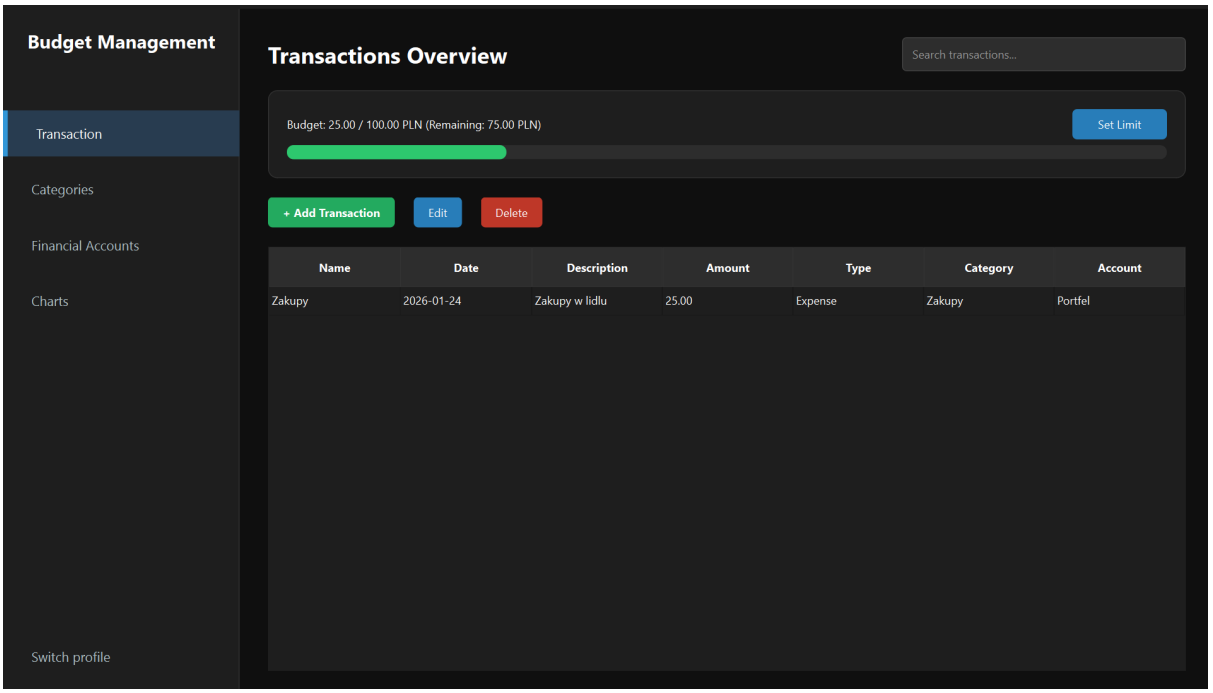
Okno logowania:



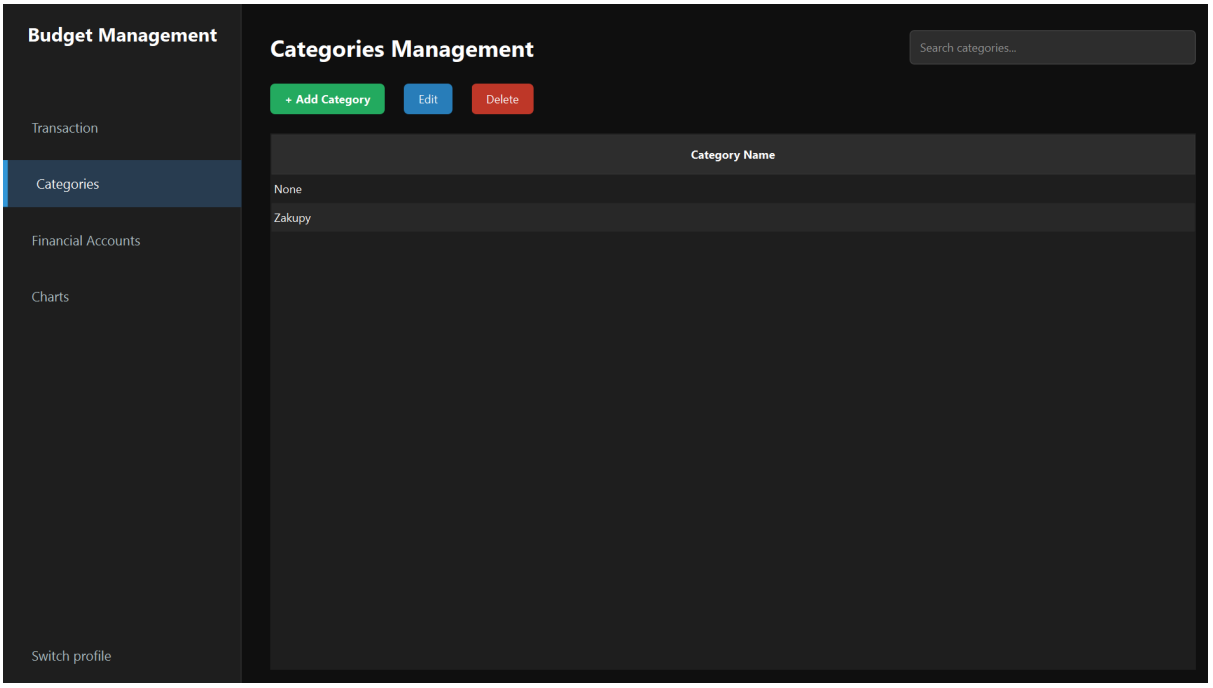
Okno wyboru użytkownika:



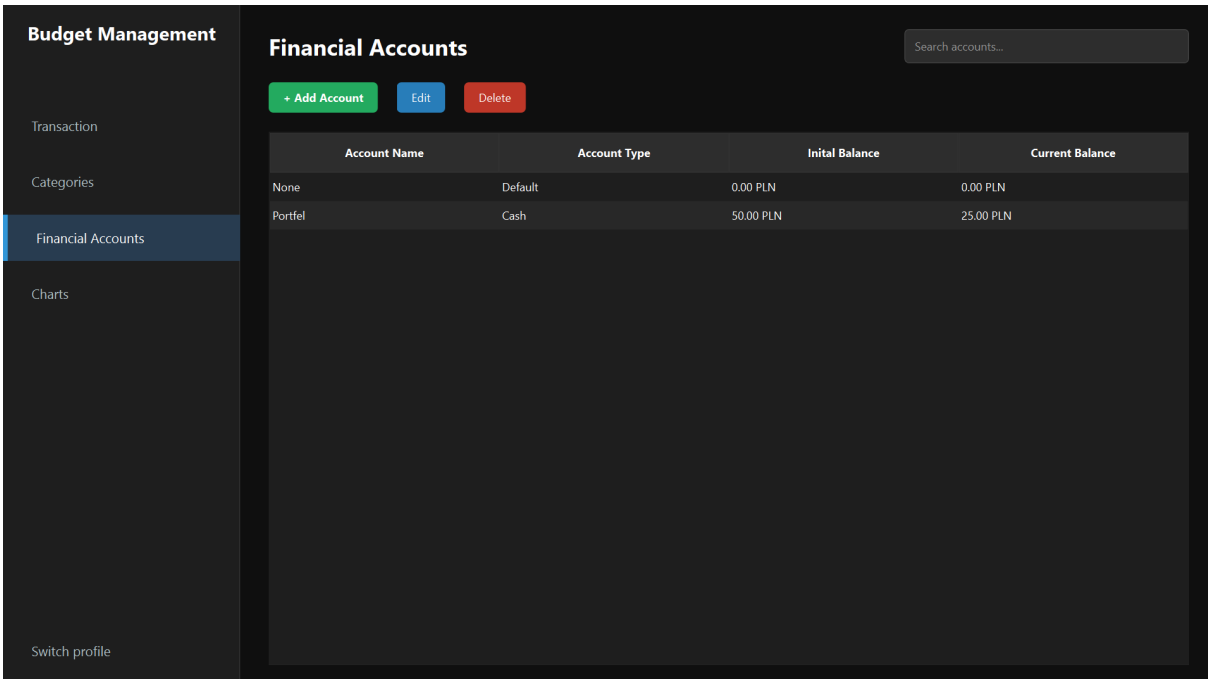
Okno transakcji i ustawienia limitu miesięcznego:



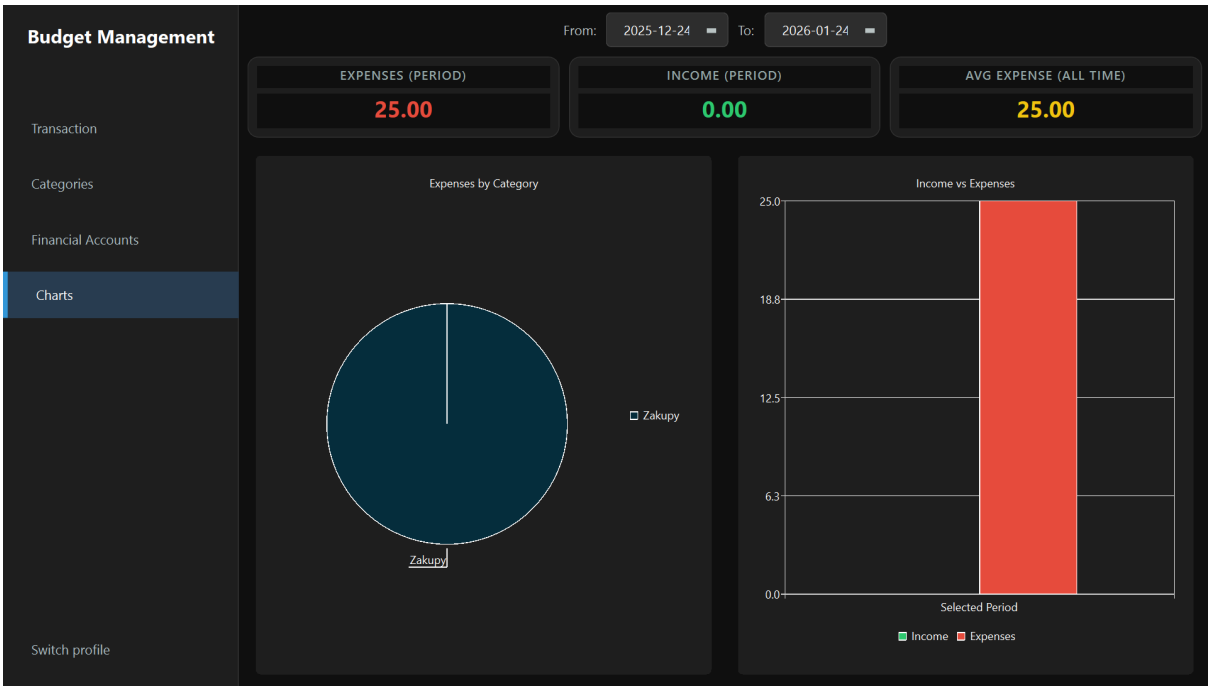
Okno kategorii:



Okno kont finansowych:



Okno statystyk:



Słownik projektu

- **Aplikacja desktopowa** – program komputerowy uruchamiany lokalnie na systemie operacyjnym, służący w tym projekcie do planowania budżetu domowego.
- **Autoryzacja** – proces weryfikacji tożsamości użytkownika podczas uruchamiania aplikacji, realizowany za pomocą hasła lub kodu PIN.
- **Baza danych** – zbiór danych finansowych użytkownika zapisywany strukturalnie w pliku lokalnym, umożliwiający ich szybkie wyszukiwanie i edycję.
- **Budżet miesięczny** – ustalona kwota, jaką użytkownik planuje przeznaczyć na wydatki w danym miesiącu (dla całości finansów lub konkretnej kategorii).
- **Eksport danych** – funkcja umożliwiająca zapisanie raportów lub historii transakcji do zewnętrznego pliku (np. CSV) w celu ich archiwizacji lub analizy w innych programach.
- **Import danych** – proces wczytywania informacji o transakcjach z zewnętrznych plików, np. wyciągów bankowych.
- **Interesariusz** – osoba lub grupa osób, która ma wpływ na projekt lub jest zainteresowana jego wynikami (np. użytkownik, zespół projektowy, opiekun).
- **Interfejs użytkownika (UI)** – wizualna część aplikacji (okna, przyciski, formularze), umożliwiająca interakcję użytkownika z systemem.
- **Kategoria** – etykieta przypisywana do transakcji w celu grupowania wydatków i przychodów (np. żywność, transport, pensja).
- **Konto** – zdefiniowane w systemie źródło środków finansowych użytkownika, np. portfel (gotówka), rachunek bankowy, karta kredytowa.
- **Kopia zapasowa** – duplikat pliku bazy danych tworzony automatycznie, służący do odzyskiwania danych w przypadku awarii sprzętu lub uszkodzenia plików.
- **MVC** – wzorzec architektoniczny użyty w projekcie, dzielący aplikację na trzy główne moduły: Model, Widok i Kontroler, co ułatwia testowanie i rozwój kodu.
- **OCR (Optyczne Rozpoznawanie Znaków)** – technologia planowana w dalszym rozwoju aplikacji, umożliwiająca automatyczne odczytywanie danych z paragonów i faktur.
- **PIN / hasło** – poufny ciąg znaków służący do zabezpieczenia dostępu do aplikacji przed osobami niepowołanymi.
- **Plik CSV** – (Comma-Separated Values) prosty format pliku tekstowego, w którym dane oddzielone są przecinkami, używany w aplikacji do wymiany danych.
- **Pulpit** – ekran startowy aplikacji (dashboard), prezentujący najważniejsze podsumowania, takie jak saldo kont, stan budżetu i ostatnie transakcje.

- **Qt** – biblioteka C++ wykorzystana do stworzenia interfejsu graficznego i logiki aplikacji.
- **Raport finansowy** – zestawienie analityczne generowane przez aplikację, prezentujące strukturę przychodów i wydatków w wybranym okresie.
- **Saldo** – aktualny stan środków finansowych, stanowiący różnicę między sumą przychodów a sumą wydatków.
- **SQLite** – lekki system zarządzania bazą danych, wykorzystany w projekcie do lokalnego przechowywania danych w pliku.
- **SQL Injection** – rodzaj ataku na aplikację polegający na wstrzyknięciu złośliwego kodu SQL.
- **Szyfrowanie** – proces kodowania danych w lokalnej bazie danych, zapewniający ich nieczytelność dla osób postronnych i gwarantujący prywatność.
- **Transakcja** – podstawowa operacja w systemie, odzwierciedlająca pojedynczy przychód lub wydatek (zawiera kwotę, datę, kategorię i opis).
- **Tryb offline** – sposób działania aplikacji nie wymagający połączenia z internetem; wszystkie dane przetwarzane i składowane są na dysku komputera.
- **Użytkownik** – aktor systemu; osoba korzystająca z aplikacji do codziennego zarządzania swoimi finansami osobistymi.
- **Wykres** – graficzna wizualizacja danych finansowych (np. kołowa lub słupkowa) ułatwiająca analizę struktury wydatków.

Wnioski

1. Podsumowanie realizacji projektu
Projekt zakończył się sukcesem, dostarczając funkcjonalną aplikację desktopową do zarządzania budżetem domowym, zgodną z założeniami specyfikacji wstępnej. Główny cel, jakim było stworzenie narzędzia działającego w trybie offline i gwarantującego pełną prywatność danych użytkownika, został osiągnięty.

2. Napotkane wyzwania techniczne i projektowe
Podczas realizacji projektu zespół napotkał szereg wyzwań, które wymagały podjęcia kluczowych decyzji architektonicznych:
Wizualizacja danych: Implementacja czytelnych i dynamicznych wykresów wymagała przetestowania kilku bibliotek graficznych. Wyzwaniem było dostosowanie ich wyglądu do interfejsu aplikacji oraz zapewnienie poprawnego skalowania przy dużej ilości danych historycznych.

Projektowanie UI w aplikacji desktopowej: Stworzenie interfejsu, który jest nowoczesny i intuicyjny, a jednocześnie zachowuje charakter aplikacji desktopowej, wymagało wielu wersji projektu graficznego.

Praca grupowa i kontrola wersji: Synchronizacja prac pięcioosobowego zespołu wymagała dyscypliny w stosowaniu systemu kontroli wersji. Rozwiązywanie konfliktów w kodzie (merge conflicts) przy pracy nad wspólnymi modułami było cenną lekcją organizacji pracy.

3. Zdobyte doświadczenia

Realizacja projektu pozwoliła członkom zespołu na zdobycie praktycznej wiedzy w następujących obszarach:

Praktyczne zastosowanie inżynierii oprogramowania: Przejście przez pełny cykl życia oprogramowania – od analizy wymagań, przez projektowanie (UML), implementację, aż po testy.

Zarządzanie bazami danych: Pogłębienie wiedzy na temat projektowania relacyjnych baz danych w kontekście aplikacji lokalnych.

Praca z bibliotekami zewnętrznymi: Integracja gotowych rozwiązań do generowania wykresów oraz obsługi formatów plików.

Kompetencje miękkie: Zespół udoskonalił umiejętności komunikacji, podziału zadań oraz terminowości.

4. Potencjalne kierunki dalszego rozwoju aplikacji

W przyszłych wersjach systemu rekomenduje się wdrożenie następujących funkcji:

Obsługa cykliczności: Dodanie modułu "Zlecenia stałe", który automatycznie dodawałby powtarzalne transakcje każdego miesiąca.

Import bankowy i OCR: Implementacja inteligentnego importu wyciągów bankowych oraz funkcji skanowania paragonów w celu automatyzacji wprowadzania danych.

Planowanie długoterminowe: Rozbudowa modułu budżetowania o cele oszczędnościowe i prognozowanie stanu konta na podstawie historycznych trendów.

Opcjonalna synchronizacja: Rozważenie wprowadzenia szyfrowanej synchronizacji chmurowej jako opcji dla użytkowników, którzy chcą mieć dostęp do danych na kilku komputerach, bez rezygnacji z prywatności.

Wersja mobilna: Stworzenie prostej aplikacji mobilnej służącej wyłącznie do szybkiego dodawania wydatków w terenie, która synchronizowałaby się z bazą główną po podłączeniu do komputera.