Dokumentacja projektu zaliczeniowego "Vabank"

Piotr Hejmo Karol Borowski

Czerwiec 2025

Dokumentacja projektu "Vabank"
tworzonego w okresie
Maj - Czerwiec 2025
przez Piotra Hejmo i Karola Borowskiego

Zawartość dokumentacji

1 Opis Projektu

Postanowiliśmy stworzyć aplikacje symulującą aplikację bankową, którą wykonamy z wykorzystaniem technologii Qt dla aplikacji okienkowych w języku C++. Aplikacja wedle naszego zamysłu miała mieć wbudowane podstawowe funkcjalności komercyjnych aplikacji bankowych oraz być przemyślana pod kątem skalowalności.

2 Podstawowe funkcjonalności

Aplikacja posiada następujące funkcje:

- Mechanizm przesyłania przelewów
- Bankomat
- Walidacje danych wprowadzanych przez użytkownika
- Mechanizm logowania oraz zapisywania danych

Działanie poszczególnych funkcji zostanie pokazane poniżej.

3 Zastosowane technologie

Do stworzenia projektu wykorzystaliśmy technologię **Qt** dla języka C++. Cała aplikacja jest napisana w dedykowanym dla tej technologii Qt Creatorze, którego licencję uzyskaliśmy z domeny **.polsl.pl** jako ofertę firmy stojącej za Qt dla edukacji. Logika aplikacji napisana jest w C++, system zapisywania danych obsługuje baza danych stworzona w technologii **SQLite**.
Grafika, która posłużyła nam za ikonę aplikacji została *wygenerowana* w **ChatGPT firmy OpenAI**.

4 Architektura i zawartość plików

Aplikacja składa się z trzech typów plików

1. **plików .cpp**
2. **plików .h**
3. **plików zasobów**

.

4.1 Plik main.cpp

Jest to plik inicjalizujący działanie aplikacji oraz ustawiającym za pomocą ścieżki ikonę programu.

- Funkcja main

```
int main(int argc, char *argv[])
{
QApplication a(argc, argv);
```

```
QIcon appIcon(":/Ikona.png");
a.setWindowIcon(appIcon);

MainWindow w;
w.show();
return a.exec();
}
```

4.2 Plik mainwindow.cpp

W tym pliku jest największa część naszej aplikacji.

- **Konstruktor Okna**
  Jest to klasa dziedzicząca po klasie **QMainWindow**.

  ```
  MainWindow::MainWindow(QWidget *parent)
  : QMainWindow(parent)
  , ui(new Ui::MainWindow)
  ```

  Po ustawieniu **ui** jako nowe UI dziedziczące po MainWindow:

  ```
      ui->setupUi(this);
  ```

  Następuje mechanizm przyporządkujący nazwy przycisków, które nie potrzebują zapytania do bazy, ich odpowiednikom w UI:

  ```
  //tablica nazw przyciskow z glownego okna do automatycznego przypisania
  QStringList tablicaNazwPrzyciskow = {"pushButton_logowanie_do_banku",
  "pushButton_bankomat",
  "pushButton_anuluj_2",
  "pushButton_anuluj_3",
  "pushButton_zarejestruj_sie",
  "pushButton_anuluj"};

  //petla przypisujaca nazwy przyciskow do akcji w UI
  for (int i = 0; i < tablicaNazwPrzyciskow.size(); ++i) {
      QString nazwa = tablicaNazwPrzyciskow[i];
      QPushButton *przycisk = findChild<QPushButton*>(nazwa);
      if (przycisk) {
          connect(przycisk, &QPushButton::clicked, this, [=]() {
              obsluzPrzycisk(i);
          });
          qDebug() << "Połaczono przycisk:" << nazwa << "z wartościa:" << i;
      } else {
          qDebug() << "Nie znaleziono przycisku:" << nazwa;
      }
  }
  ```

Po każdej iteracji developer dostaje informację czy operacja się udała. Użytkownik nie widzi tej informacji. Następnie program przygotowuje bazę danych:

```
QString sciezka = "baza1.db";
qDebug() << "Ścieżka do bazy:" << sciezka;

DB_Connection = QSqlDatabase::addDatabase("QSQLITE");
DB_Connection.setDatabaseName(sciezka);

if(DB_Connection.open()){
    qDebug() << "Polaczenie gut";
} else {
    qDebug() << "Polaczenie not gut: " << DB_Connection.lastError().text();
}
```

Ponownie użytkownik nie widzi komunikatu dla developera czy połączenie zostało nawiązane

- ***Obsługa przycisków*** Obsługa przycisków dla usług nie wymagających SQL wygląda w następujący sposób:

```
void MainWindow::obsluzPrzycisk(int wartosc)
{
switch(wartosc)
{
case 0: //przycisk logowanie do banki clicked // przejscie do strony z logowanniem
    ui->stackedWidget->setCurrentIndex(1);
    break;
case 1: //bankomat clicked // przejscie do strony bankomatu wplaty/wypaty
    ui->stackedWidget->setCurrentIndex(3);
    break;
case 2: //anuluj_2 clicked // przejscie do wyboru startowego z panelu logowania
    ui->stackedWidget->setCurrentIndex(0);
    break;
case 3: //anuluj w bankmacie clicked // przejscie do wyboru startowego z bankomatu
    ui->stackedWidget->setCurrentIndex(0);
    break;
case 4: // zarejestruj clicked // przejscie do strony z rejestracja
    ui->stackedWidget->setCurrentIndex(2);
    break;
case 5: //anuluj_clicked // cofanie do logowania
    ui->stackedWidget->setCurrentIndex(1);
    break;
default:
    qDebug() << "Nieobsługiwany indeks:" << wartosc;
```

```
    break;
}
}
```

Wartością po której działa ten switch to indeks przycisku w podanej powyżej tabeli nazw przycisków. W przypadku default tą informację widzi tylko developer.
Z kolei przyciski wymagające SQL są opisane w następujący sposób:

- **Przycisk wpłaty środków**

```
void MainWindow::on_pushButton_wplata_clicked() //
{
bool okKwota, okId;
double kwota = ui->lineEdit_kwota_bankomat->text().toDouble(&okKwota);
kwota = round(kwota*100)/100;
int id = ui->lineEdit_numer_konta_id_bankomat->text().toInt(&okId);
if (!okKwota || !okId || kwota <= 0) {
    QMessageBox::warning(this, "Błąd", "Wprowadź poprawne dane.");
    return;
}
```

Na początku następuje pobranie z okna kwoty oraz jej poprawny zapis. Później program pobiera z sesji numer id bankomatu jako nadawcy środków.
W przypadku błędu lub niepoprawności wprowadzonych danych aplikacja zgłosi użytkownikowi błąd.
W przypadku pozytywnej weryfikacji następują poniższe operacje:

```
QSqlDatabase db = QSqlDatabase::database();
if(!db.isOpen()) db.open();

db.transaction();
QSqlQuery q;
q.prepare("UPDATE login SET balans = balans +:kwota WHERE id = :id");
q.bindValue(":kwota",kwota);
q.bindValue(":id",id);

if (q.exec()) {
    db.commit();
    QMessageBox::information(this, "Sukces", "Wpłata wykonana pomyślnie.");

} else {
    db.rollback();
    QMessageBox::critical(this, "Błąd",
    "Nie udało się wykonać wpłaty.
    Sprawdź kwote i numer konta(id).");
```

Najpierw program otwiera połączenie z bazą oraz weryfikuje jego poprawność. Następnie przygotowuje odpowiednie zapytania i sprawdza poprawność wykonania zapytania. Użytkownik zostaje powiadomiony o wyniku operacji.

- **Wypłata z bankomatu**
  W tym fragmencie omówimy działanie programu po naciśnięciu przycisku wypłaty z bankomatu.
  Na początek następuje taka sama jak w przypadku wpłaty procedura ustalania id i kwoty oraz otwarcie połączenia do bazy danych.

```
bool okKwota, okId;
double kwota = ui->lineEdit_kwota_bankomat->text().toDouble(&okKwota);
kwota = round(kwota*100)/100;
int id = ui->lineEdit_numer_konta_id_bankomat->text().toInt(&okId);
double bilans1,bilans2; // sprawdzanie czy wgl sie cos wyplacilo
if (!okKwota || !okId || kwota <= 0) {
    QMessageBox::warning(this, "Błąd", "Wprowadź poprawne dane.");
    return; //znowu void returnuje
}
QSqlDatabase db = QSqlDatabase::database();
if(!db.isOpen()) db.open();
```

  Następnie rozpoczyna się proces przygotowania odpowiednich zapytań do bazy. Na początku w zapytaniu **q2** wyciągamy stan konta przypisany do tego id.

```
db.transaction();
QSqlQuery q,q2,q3;

q2.prepare("SELECT balans FROM login where id = :id");
q2.bindValue(":id",id);
q2.exec();
q2.next();

bilans1 = q2.value("balans").toDouble();
```

  Po jego konwersji na double w zapytaniu **q** pomniejszamy jego stan o kwotę jeżeli jest mniejsza od bilansu.

```
q.prepare("UPDATE login SET balans = balans -:kwota
WHERE id = :id AND balans >= :kwota");
q.bindValue(":kwota",kwota);
q.bindValue(":id",id);
```

  W zapytaniu **q3** ponownie pobieramy bilans przypisany do tego id i sprawdzany czy jest różny od bilansu z **q2**.

```
q3.prepare("SELECT balans FROM login where id = :id");
q3.bindValue(":id",id);
q3.exec();
q3.next();
bilans2 = q3.value("balans").toDouble();
```

Jeżeli są różne to użytkownik zostaje powiadomiony o sukcesie operacji.

```
if (q.exec() && bilans1!=bilans2)
{
db.commit();
qDebug() << bilans1;
qDebug() << bilans2;
QMessageBox::information(this, "Sukces", "Wypłata wykonana pomyślnie.");

} else {
db.rollback();
QMessageBox::critical(this, "Błąd", "Nie udało się wykonać wypłaty.
Sprawdź kwote i numer konta(id).");
}
```

- **Logowanie** W celu zalogowania użytkownika do serwisu otwieramy połączenie do bazy oraz pobieramy dane z okna:

```
//otwieranie bazy i deklaracja podstawowych zmiennych
DB_Connection.open();
QSqlDatabase::database().transaction();

QString login = ui->lineEdit_login->text();

QString password = ui->
lineEdit_2_password->text();
```

Następnie przygotowujemy odpowiednie zapytanie:

```
//zapytanie do bazy w celu zalogowania
query.prepare("SELECT id FROM login WHERE
login = :login AND haslo = :haslo LIMIT 1");
query.bindValue(":login", login);
query.bindValue(":haslo", password);
```

Po czym następuje jego weryfikacja i powiadomienie użytkownika o jego wyniku. Dodatkowo deweloper może uzyskać informację o id zalogowanego użytkownika. Następnie zamykana jest baza danych i procedura jest zakończona.

```
//wykonanie zapytania i weryfikacja
if (query.exec()) {
if ( query.exec()&& query.next()) {
    idzbazy = query.value(0).toInt();
    qDebug() << "Zalogowano. ID uzytkownika:" << idzbazy;
    secondwindow *second = new secondwindow(idzbazy);
    second->show();
    this->close();
} else {
    qDebug() << "Błędny login lub hasło";
    QMessageBox::warning(this, "Błąd logowania", "Błędny login lub hasło.");
}

} else {
qDebug() << "Blad zapytania: " << query.lastError().text();
}

QSqlDatabase::database().commit();
DB_Connection.close();
```

- **Mechanizm rejestracji nowego użytkownika** Ten mechanizm jest najbardziej skomplikowany w całej aplikacji. Rozpoczyna się ustaleniem listy parametrów potrzebnych do rejestracji i połączenia ich z nazwami w UI:

```
 //lista nazw atrybutow potrzebna do rejestracji
QStringList atrybuty = {
    "login", "haslo", "imie", "nazwisko", "email",
    "numer_tel", "Ulica_i_nr", "Miasto"
};

// lista nazw atrybutow w UI
QStringList atrybutyUi = {
    "lineEdit_login_2",
    "lineEdit_haslo_2",
    "lineEdit_imie",
    "lineEdit_nazwisko",
    "lineEdit_adres_email",
    "lineEdit_numer_telefonu",
    "lineEdit_ulica",
    "lineEdit_miasto"
};
```

Następnie przygotowane jest zapytanie do bazy w celu zapisania użytkownika w bazie:

```
    QSqlQuery query;
    query.prepare("INSERT INTO login (login, haslo, imie, nazwisko, email,
    numer_tel, Miasto, Ulica_i_nr)
    VALUES
    (:login, :haslo, :imie, :nazwisko, :email, :numer_tel, :Miasto, :Ulica_i_nr)");
```

Następnie powstaje **mapa typu QMap**, której zadaniem jest przypisanie wartości do zapytania:

```
//przypisanie wartosci atrybutu -> atrybut w bazie
QMap<QString, QString> dane;
for (int i = 0; i < atrybuty.size(); ++i)
{
    QLineEdit* lineEdit = this->findChild<QLineEdit*>(atrybutyUi[i]);
    if (lineEdit)
    {
        dane[atrybuty[i]] = lineEdit->text();
    }
}
```

Kolejne procesy dotyczą weryfikacji podanych danych. Najpierw jest to weryfikacja czy dane pole nie jest puste:

```
//sprawdzanie czy wszystkie dane sa wpisane
for(int i = 0; i< atrybuty.size();i++)
{
    QString wprowadzonaWartosc = dane[atrybuty[i]];
    if(wprowadzonaWartosc.isEmpty())
    {
        QMessageBox::warning(this, "Błąd", "Wypełnij wszystkie wymagane pola!");
        return;
    }
```

Następnie w specjalnych przypadkach (imie, nazwisko) sprawdzamy poszczególne litery wprowadzonych wartości. W przypadku maila sprawdzamy czy ma znaki **@** oraz **.** oraz jego długość.

```
for (const QChar &ch : wprowadzonaWartosc)
{
if (!ch.isLetter() && ch != ' ')
{
    if(i==2)
    {
        QMessageBox::warning(this, "Błąd",
        "Wprowadz poprawne imie!");
        ui->lineEdit_imie->setFocus();
```

```
        return;
    }
    else if(i==3)
    {
        QMessageBox::warning(this, "Błąd",
        "Wprowadz poprawne nazwisko!");
        ui->lineEdit_nazwisko->setFocus();
        return;
    }
    else if(i==4)
    {
        if (!wprowadzonaWartosc.contains("@") ||
        !wprowadzonaWartosc.contains(".") ||
        wprowadzonaWartosc.length() < 5)
        {
            QMessageBox::warning(this, "Błąd",
            "Podaj poprawny adres email!");
            ui->lineEdit_adres_email->setFocus();
            return;
        }
    }
}
```

Następnie w polu dla **numer telefonu** sprawdzamy długość oraz poprawność:

```
{
    if(wprowadzonaWartosc.length() < 9)
    {
        QMessageBox::warning(this, "Błąd",
        "Numer telefonu musi mieć co najmniej 9 cyfr!");
        ui->lineEdit_numer_telefonu->setFocus();
        return;
    }
    for (const QChar &ch : wprowadzonaWartosc)
    {
        if (!ch.isDigit() && ch != ' ' && ch != '-')
        {
            QMessageBox::warning(this, "Błąd",
            "Numer telefonu może zawierać tylko cyfry, spacje i myślniki!");
            ui->lineEdit_numer_telefonu->setFocus();
            return;
        }
    }
}
```

W przypadku przejścia weryfikacji następuje przypisanie danych do zapytania:

```
//przypisanie wartosci do zapytania
for (auto it = dane.begin(); it != dane.end(); ++it)
{
    query.bindValue(":" + it.key(), it.value());
}
```

Poźniejszym etapem jest wykonanie zapytania i zamknięcia bazy. Użytkownik dostaję informację o wyniku zapytania:

```
//sprawdzenie zapytania
if (query.exec())
{QMessageBox::information(this, "Sukces",
"Utworzono konto");}
else
{QMessageBox::critical(this, "Błąd",
"Rejestracja nieudana");}

obsluzPrzycisk(0);
QSqlDatabase::database().commit();
DB_Connection.close();
```

4.3 Plik secondwindow.cpp

Jest to plik opisujący działanie okna do specjalnych operacji.

- **Klasa secondwidnow**
  Konstrutkor tej klasy wyświetla developerowi informację nt obecnego id wywoluje funkcje **daneuzytkownika(),danedoprzelewu(), wyswietlHistorie()**.

```
secondwindow::secondwindow(int idzbazy,QWidget *parent)
: QDialog(parent),
ui(new Ui::secondwindow),
m_idzbazy(idzbazy)
{
ui->setupUi(this);
qDebug() << "id przekazane to: " << m_idzbazy;

ui->tableWidget_historia->setColumnCount(4);
ui->tableWidget_historia->
setHorizontalHeaderLabels(QStringList() << "Nadawca ID" << "Odbiorca ID" <<
"Kwota" << "Data");
```

```
daneuzytkownika();
danedoprzelewu();
wyswietlHistorie();
}
```

- **daneuzytkownika()**
  Ta funkcja odpowiada za dostarczenie informacji nt obecnego stanu konta użytkownika.
  Rozpoczyna od przygotowania odpowiedniej kwerendy:

```
 //przygotowanie zapytania
QSqlQuery q;
q.prepare("SELECT imie, balans FROM login where id = :id");
q.bindValue(":id", m_idzbazy);
```

  Oraz sprawdza jego poprawne wykonanie:

```
//postepowanie w przypadku bledu zapytania
if (!q.exec()) {
    qDebug() << "Błąd SELECT:" << q.lastError().text();
    ui->label_imie ->setText("Błąd bazy");
    ui->label_balans->setText("---");
```

  A następnie wyświetla w odpowiednim formacie dane:

```
if (q.next()) {
        const QString imie  = q.value("imie").toString();
        const double  balans = q.value("balans").toDouble();

        //wyswietlenie danych w dobrym formacie
        ui->label_imie ->setText(QString("%1").arg(imie));
        ui->label_balans->setText(QString("%1 zł").arg(balans, 0, 'f', 2));
        ui->label_id->setText(QString("%1").arg(m_idzbazy));
```

  W przypadku nie zrealizowania kwerendy wyświetla się informacja o braku informacji:

```
} else {
    ui->label_imie ->setText("Nie znaleziono");
    ui->label_balans->setText("---");
}}
```

- **danedoprzelewu()**
  Funkcja odpowiada za wyświetlenie balansu w oknie wykonania przelewu:

```
//przygotowanie zapytania o balans po id
QSqlQuery q;
q.prepare("SELECT balans FROM login where id = :id");
q.bindValue(":id", m_idzbazy);

//przypadek gdy zapytanie nie zadziala
if (!q.exec()) {
    qDebug() << "Błąd SELECT:" << q.lastError().text();
}
else
{
if (q.next()) {
    const double  balans = q.value("balans").toDouble();
    ui->label_balans_2->setText(QString("%1 zł").arg(balans, 0, 'f', 2));
    ui->label_id_2->setText(QString("%1").arg(m_idzbazy));
    return;
}}
//postepowanie w przypadku gdy q.exec lub !q.next()
ui->label_balans->setText("---");
}
```

W przypadku błędnego wykonania kwerendy wyświetli się —.

- **on pushButton wykonaj przelew()**
  Funkcja odpowiada za pobranie, uwierzytelnianie danych oraz wykonanie przelewu. Etap weryfikacji kwoty i id jest taki sam jak w pliku **mainwindow.cpp**
  Mechanizm pobrania, formatowania i uwierzytelniania danych wygląda następująco:

```
{
//przygotowanie zmiennych
bool okKwota, okId;
double kwota = ui->lineEdit_przelew_kwota->text().toDouble(&okKwota);
kwota = round(kwota*100)/100;
qDebug()<<kwota;
int odbiorcaId = ui->lineEdit_przelew_nr__konta->text().toInt(&okId);


//sprawdzenie kwoty i id
if (!okKwota || !okId || kwota <= 0) {
    QMessageBox::warning(this, "Błąd", "Wprowadź poprawne dane.");
    return;
}
```

Mechanizm przesyłu danych do bazy wygląda następująco:

```
//przygotowanie zapytania
QSqlDatabase db = QSqlDatabase::database();
if(!db.isOpen()) db.open();
db.transaction();
QSqlQuery q1,q2,q3;

//przygotowanie q1
q1.prepare("UPDATE login SET balans = balans - :kwota WHERE
id = :nadawca_id AND balans >= :kwota");
//tu tez jest mechanizm wyplaty,
q1.bindValue(":kwota",kwota); //podowjne opisanie tylko sie parametry roznia
q1.bindValue(":nadawca_id", m_idzbazy);

q2.prepare("UPDATE login SET balans = balans + :kwota
WHERE id = :odbiorca_id");
q2.bindValue(":kwota",kwota);
q2.bindValue(":odbiorca_id",odbiorcaId);

q3.prepare("INSERT INTO historia (nadawca_id, odbiorca_id, kwota)
VALUES (:nadawca_id, :odbiorca_id, :kwota)");
q3.bindValue(":nadawca_id",m_idzbazy);
q3.bindValue(":odbiorca_id",odbiorcaId);
q3.bindValue(":kwota",kwota);
```

Na sam koniec przeprowadzona jest weryfikacja tego czy operacja została wykonana prawidłowo:

```
//sprawdzenie czy wykonaly sie zapytania
    if (q1.exec() && q1.numRowsAffected() == 1
    && q2.exec() && q2.numRowsAffected() == 1 && q3.exec()) {
        db.commit();
        QMessageBox::information(this, "Sukces", "Przelew wykonano pomyślnie.");
        daneuzytkownika();
        danedoprzelewu();
        wyswietlHistorie();
    } else {
        db.rollback();
        QMessageBox::critical(this, "Błąd", "Nie udało się wykonać przelewu.
        Sprawdź środki i ID odbiorcy.");
    }
```

- **wyswietlHistorie()**
  Funkcja agregująca i wyświetlająca w formie tabeli historie przelewów. Na początku jest przygotowywana i realizowana kwerenda do bazy danych:

```
//zapytanie o transakcje po id
```

```
QSqlQuery q;
q.prepare("SELECT odbiorca_id, nadawca_id, kwota, data FROM historia WHERE
odbiorca_id = :id OR nadawca_id = :id ORDER BY data DESC");
q.bindValue(":id",m_idzbazy);

//przypadek bledu
if (!q.exec()) {
    qDebug() << "Błąd historii:" << q.lastError().text();
    return;
}
```

Następnie w pętli wyświetlana jest tabela, ilość jej rzędów zależy od danych uzyskanych z bazy:

```
ui->tableWidget_historia->setRowCount(0);
    int row = 0;

//wyswietlanie danych
while (q.next()) {
    int nadawca = q.value("nadawca_id").toInt();
    int odbiorca = q.value("odbiorca_id").toInt();
    double kwota = q.value("kwota").toDouble();
    QString data = q.value("data").toString();

    //mechanizm koloru zaleznego od typu transakcji
    QColor kolor = (nadawca == m_idzbazy) ? QColor(220, 0, 0) : QColor(0, 150, 0);
    QString kwotaStr = QString("%1 zł").arg(kwota, 0, 'f', 2);

    //definicja wyswietlania danych
    QTableWidgetItem *itemNadawca = new QTableWidgetItem(QString::number(nadawca));
    QTableWidgetItem *itemOdbiorca = new QTableWidgetItem(QString::number(odbiorca));
    QTableWidgetItem *itemKwota = new QTableWidgetItem(kwotaStr);
    QTableWidgetItem *itemData = new QTableWidgetItem(data);

    itemKwota->setForeground(QBrush(kolor));

    //definicja sposobu wyswietlania
    ui->tableWidget_historia->insertRow(row);
    ui->tableWidget_historia->setItem(row, 0, itemNadawca);
    ui->tableWidget_historia->setItem(row, 1, itemOdbiorca);
    ui->tableWidget_historia->setItem(row, 2, itemKwota);
    ui->tableWidget_historia->setItem(row, 3, itemData);

    row++;
}
```

- **Funkcje specjalne** Znajdują się tu następujące 3 funkcje odpowiedzialne za reakcje na naciśnięcie przycisków: Funkcja po naciśnięciu przycisku

  1. **Wyloguj**:

     ```
     void secondwindow::on_pushButton_wyloguj_clicked()
     {
         MainWindow *main = new MainWindow();
         main->show();
         this->close();
     }
     ```

  2. Funkcja po naciśnięciu **Powrót**

     ```
     void secondwindow::on_pushButton_powrot_clicked()
     {
         ui->stackedWidget->setCurrentIndex(0);
     }
     ```

     Oraz funkcja odpowiedzialna za powrót do głównego okna konta.

  3. **wykonaj przelew**

     ```
     void secondwindow::on_pushButton_przelew_clicked()
     //przycisk by przejsc do strony przelewow
     {
         ui->stackedWidget->setCurrentIndex(1);
     }
     ```

     Funkcja ta przełącza UI do bazowego okna.

4.4 Pliki zasobów

Do tych plików zaliczamy pliki:

- plik z bazą danych: **baza1.db**
- plik z ikoną aplikacji: **Ikona.png**

Oba pliki znajdują się w głównym katalogu projektu.

5 Pierwsze uruchomienie oraz konfiguracja danych

- W celu uruchomienia projektu jako developer zalecana jest instalacja **Qt Creatora** wraz z odpowiednimi pakietami do języka C++ zalecanymi przez wydawcę oraz sklonowania repozytorium z serwisu **Github.com**.
- Dane wymagane do utworzenia konta:

  1. Login: Maksymalnie 50 znaków.

2. Imie: Max 50 znaków, same litery
3. Nazwisko: Max 50 znaków, same litery
4. Adres email, max 50znaków, wymaga użycia @ oraz .
5. Nr Telefonu: min 9 cyfr i max 11 cyfr, dopuszcza format XXX-XXX-XXX.
6. Miasto: max 50 znaków.
7. Ulica i nr domu: max 50 znaków.

- Do wpłaty,wyplaty potrzebny jest **numer id** widoczny po zalogowaniu w prawym dolnym rogu.
- Po **zarejestrowaniu** nowego użytkownika, aplikacja przełączy się na ekran logowania.
  Użytkownik powinien zalogować się danymi właśnie stworzonym kontem.

6 Pomysły na rozwój aplikacji

- Wprowadzenie mechanizmu szyfrowania danych.
  Nasz pomysł na rozwój aplikacji zawiera przede wszystkim pomysły na zwiększenie zabezpieczeń aplikacji poprzez szyfrowanie. Umożliwiłoby to wprowadzenie mechanizmu podobnego do cyfrowego podpisu. W koncepcji rozwoju pojawił się też mechanizm generatora kodów systemu BLIK.
- Kalkulator rat
  Wprowadzilibyśmy kalkulator rat na pożyczyki ze stałym oprocentowaniem. Umożliwi on użytkownikom realne i proste planowanie budżetu.
- Wprowadzenie większej skalowalności i rewitalizacja kodu.
  Mimo obecnego przygotowania optymalizacyjnego i skalowalnego wprowadzilibyśmy zestaw narzędzi dla developera ułatwiający dodawanie kwerend lub przycisków.

7 Podsumowanie

Jesteśmy zadowoleni z wyników końcowych naszego projektu. Dzięki niemu rozwineliśmy swoje umiejętności w technologiach takich jak **Qt, SQL, C++** .Nauczyliśmy się też kontroli wersji dzięki narzędziom z serwisu Github.com oraz korzystania z **Qt Creator**. Nasza dokumentacja została w całości przygotowana w **LaTeX**' ie w serwisie *overleaf.com*. Mamy nadzieję, że przyszli użytkownicy docenią naszą pracę!