MoveFaster

Autorzy:

Krystian Koza

Jakub Michałek

Mateusz Janiczek

Data ukończenia: 19.12.2024

Version: 1.1 Beta

Wprowadzenie

Aplikacja rozwiązuje problem braku centralnego i efektywnego zarządzania rozkładami jazdy. Umożliwia szybką zmianę przez administratora miejsca, daty, ceny danego połączenia, a użytkownik może sprawdzić zaktualizowany rozkład w każdej chwili oraz kupić bilety na konkretny przejazd. Dzięki niej zarówno administratorzy, jak i pasażerowie mogą korzystać z intuicyjnego i wygodnego systemu do obsługi transportu publicznego i przewozów prywatnych.

Aplikacja zawiera możliwość tworzenia kont, logowanie, rejestracja. Po zalogowaniu istnieje możliwość zarządzania rozkładem jazdy. Administrator może dodawać, edytować, wyświetlać rozkład jazdy.

Podział pracy:

Krystian Koza - programista odpowiedzialny za tworzenie systemu rejestracji/logowania

Mateusz Janiczek - programista odpowiedzialny za tworzenie rozkładu jazdy

Jakub Michałek - project manager

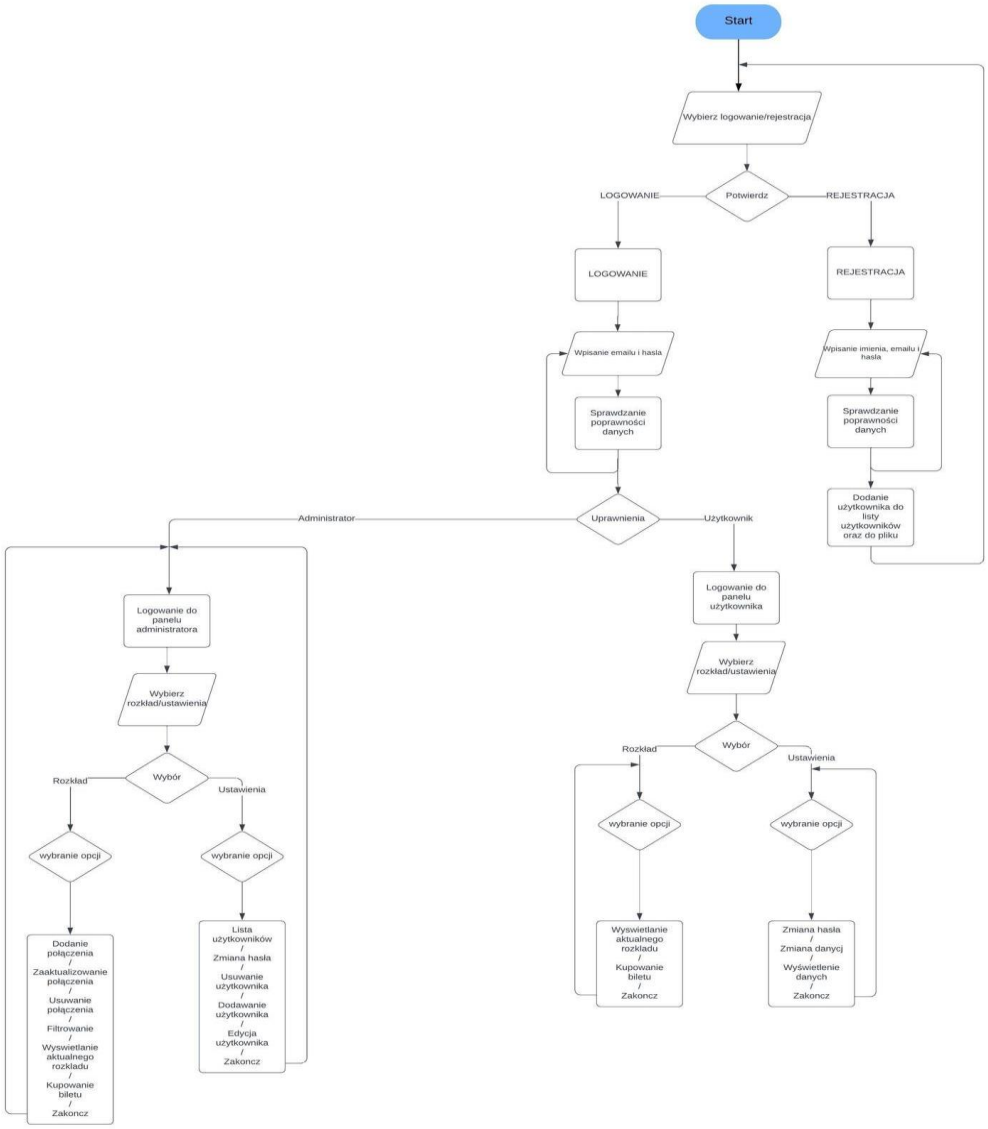
Decyzje:

Wybranie języka **C#**

Dodanie bibliotek do generowania biletu jako zdjęcia wraz z kodem QR

Szczegółowy opis aplikacji

Diagram architektury:



Funkcje aplikacji

1. Klasa User

- Zarządza danymi użytkownika, takimi jak imię, nazwisko, email, hasło i uprawnienia.
- **Najważniejsze funkcje**:
    - **zmianaHasla**: Umożliwia użytkownikowi zmianę hasła.
        - Przykład: User u = new User("Jan", "Kowalski", "jan.k@example.com", "Haslo123", "admin"); u.zmianaHasla();
    - **WyswietlDane**: Wyświetla dane użytkownika.
        - Przykład: u.WyswietlDane();
    - **SprawdzHaslo**: Sprawdza poprawność podanego hasła.
        - Przykład: bool poprawne = u.SprawdzHaslo("Haslo123");

2. Logowanie i rejestracja

- Obsługuje proces logowania i rejestracji użytkowników.
- **Funkcje**:
    - **logowanie**:
        - Przykład: Manager.logowanie();
    - **rejestracja**:
        - Przykład: Manager.rejestracja();

3. Panel użytkownika i administratora

- Zawiera interfejs dla użytkowników z różnymi uprawnieniami.
- **Funkcja główna**:
    - **Panel**: Wyświetla panel z opcjami w zależności od uprawnień użytkownika.
        - Przykład: Manager.Panel("jan.k@example.com");

4. Obsługa rozkładu jazdy

- Zarządza rozkładem jazdy, umożliwiając jego przeglądanie, edycję i zapis.
- **Funkcje**:
    - **DodajPolaczenie**: Dodaje nowe połączenie.
        - Przykład: Manager.DodajPolaczenie();
    - **UsunPolaczenie**: Usuwa wybrane połączenie.
        - Przykład: Manager.UsunPolaczenie();
    - **FiltrujPolaczenia**: Filtrowanie po mieście, cenie lub czasie dojazdu.
        - Przykład: Manager.FiltrujPolaczenia();

5. Kupowanie biletu

- Umożliwia zakup biletu na wybrane połączenie.
- **Funkcja**:
    - **KupBilet**: Realizuje zakup biletu.
        - Przykład: Manager.KupBilet();

6. Weryfikacja i zarządzanie użytkownikami

- Zarządza użytkownikami i ich danymi w systemie.
- **Funkcje**:

- **wyswietlUzytkownikow**: Wyświetla listę użytkowników.
- **usuwanieUzytkownikow**: Usuwa użytkownika z listy.
- **dodawanieUzytkownika**: Dodaje nowego użytkownika.
- **edytowanieUzytkownika**: Edytuje dane istniejącego użytkownika.

Interfejs oparty na konsoli, umożliwiający interakcję poprzez wybór opcji w menu.

Kolory konsoli (np. zielony dla sukcesu, czerwony dla błędów) poprawiają czytelność i estetykę.

Problemy i wyzwania

Problemy napotkane:

- Walidacja danych wejściowych: Należało upewnić się, że wprowadzone przez użytkownika dane są poprawne (np. format e-maila, długość hasła).
- Obsługa wyjątków: Wprowadzono mechanizmy obsługi błędów dla operacji takich jak odczyt z plików czy parsowanie danych.
- Problem z biblioteką do kodu QR - Użyta biblioteka niewspółgrała z biblioteką System.drawing.
- Problem z rysowanie tekstu na obrazie (System.drawing) - Problem z podaniem koordynatów. Koordynaty nie odwzierciedlały miejsca rysowania.

Rozwiązane wyzwania:

- Użyto wyrażeń regularnych dla walidacji e-maila i hasła.
- Stworzono mechanizmy obsługi wyjątków dla operacji wejścia/wyjścia.
- Po chwilowym researchu team postanowił za zmianą wersji biblioteki.
- Dostoswanie koordynatów do rozmiaru pliku

Kompromisy:

- Brak zewnętrznej bazy danych – wszystkie dane są przechowywane w plikach tekstowych.

Wnioski i dalszy rozwój

Wnioski:

- Udało się stworzyć aplikację zgodnie z założeniami projektowymi.
- System jest funkcjonalny, ale wymaga dalszej optymalizacji.
- Kompletny system zarządznaia użytkownikami. Możliwość logowania, rejestracji, edycji i usuwania kont. Walidacja danych.
- Generowanie wizualnych biletów. Tworzenie biletów z danymi pasażera i szczegółami podróży. Umieszczenie kodu QR z szczegółami podróży

Dalszy rozwój:

- Wdrożenie graficznego interfejsu użytkownika.
- Użycie bazy danych (np. Mysql) zamiast plików tekstowych.
- Dodanie bardziej zaawansowanego systemu filtracji i raportowania.
- Rozszerzenie i dodanie nowych funkcji dla administratora. np. możliwość sprawdzenia logów, grupowego zarządzania użytkownikami

Załączniki

- Fragmenty kodu źródłowego
- Generowanie biletu w formacie JPG:

```
private static void GenerateTicketImage(string imie, string Nazwisko, string from, string to, string godzOdjazdu, string godzinaPrzyjazdu, string cena)
{
    using (Bitmap image1 = new Bitmap("bilet.jpg", true))
    {
        Graphics graphics = Graphics.FromImage(image1);
        Brush brush = new SolidBrush(Color.Black);
        Font arial = new Font("Arial", 10, FontStyle.Regular);
        Font arialBoldItalic = new Font("Arial", 15, FontStyle.Bold | FontStyle.Italic);

        graphics.DrawString(imie, arial, brush, new Rectangle(200, 1360, 450, 100));
        graphics.DrawString(Nazwisko, arial, brush, new Rectangle(270, 1470, 450, 100));
        graphics.DrawString(cena, arial, brush, new Rectangle(670, 1410, 550, 100));
        graphics.DrawString(to, arialBoldItalic, brush, new Rectangle(85, 570, 450, 1000));
        graphics.DrawString(from, arialBoldItalic, brush, new Rectangle(110, 500, 650, 100));
        graphics.DrawString(godzinaPrzyjazdu, arialBoldItalic, brush, new Rectangle(530, 570, 450, 100));

        string qrText = $"{imie} {Nazwisko}, {cena}, Trasa: {from}, GodzinaWyjazdu: {to}, GodzinaPrzyjazdu: {godzinaPrzyjazdu}";
        using (QRCodeGenerator qrGenerator = new QRCodeGenerator())
        {
            using (QRCodeData qrCodeData = qrGenerator.CreateQrCode(qrText, QRCodeGenerator.ECCLevel.L))
            {
                using (QRCode qrCode = new QRCode(qrCodeData))
                {
                    Bitmap qrCodeImage = qrCode.GetGraphic(20);
                    graphics.DrawImage(qrCodeImage, new Rectangle(300, 1650, 350, 350));
                }
            }
        }

        string fileName = $"bilet_{DateTime.Now:ddssHH}.jpg";
        image1.Save(fileName, GetEncoderInfo("image/jpeg"), new EncoderParameters(1) { Param = { [0] = new EncoderParameter(Encoder.Quality, 25L) } });
    }
}

private static ImageCodecInfo GetEncoderInfo(string mimeType)
{
    ImageCodecInfo[] encoders = ImageCodecInfo.GetImageEncoders();
    for (int j = 0; j < encoders.Length; ++j)
    {
        if (encoders[j].MimeType == mimeType)
            return encoders[j];
    }
    return null;
}
```

Wynik:

- Wygenerowany bilet w pliku **JPG** zawierający dane pasażera i kod QR.
- Dane biletu zapisane w pliku **bilety.txt**.

Obsługa zakupu biletu:

```
static void KupBilet()
{
    Console.WriteLine("Wybierz połączenie do zakupu biletu (podaj indeks):");

    if (!int.TryParse(Console.ReadLine(), out int indeks) || indeks < 0 || indeks >= rozkladJazdy.Count)
    {
        Console.WriteLine("Nieprawidłowy indeks połączenia. Spróbuj ponownie.");
        return;
    }

    Console.WriteLine("Podaj imię pasażera:");
    string imie = Console.ReadLine();
    Console.WriteLine("Podaj nazwisko pasażera:");
    string Nazwisko = Console.ReadLine();

    string polaczenie = rozkladJazdy[indeks];
    string[] czesci = polaczenie.Split(',').Select(c => c.Trim()).ToArray();

    if (czesci.Length < 5)
    {
        Console.WriteLine("Nieprawidłowe połączenie. Sprawdź dane rozkładu.");
        return;
    }

    string bilet = $"Bilet:\nPasażer: {imie} {Nazwisko}\nPołączenie: {czesci[0]} -> {czesci[1]}\nOdjazd: {czesci[1]}\nPrzyjazd: {czesci[2]}\nCena: {czesci[4]}\n\n";
    File.AppendAllText(ticketPath, bilet);

    GenerateTicketImage(imie, Nazwisko, czesci[0], czesci[1], czesci[1], czesci[2], czesci[4]);

    Console.WriteLine("Bilet został zakupiony i zapisany (w pliku bilety.txt)");
}
```

Wejście:

- Indeks połączenia: 0
- Imię pasażera: **"Anna"**
- Nazwisko pasażera: **"Nowak"**

Wynik:

Generowanie wyniku

Rejestracja:

```
public static void rejestracja()
{
    string imie, nazwisko, email, haslo, uprawnienia;
    Console.Clear();
    do
    {
        try
        {
            Console.WriteLine("==============");
            Console.WriteLine(" REJESTRACJA ");
            Console.WriteLine("==============");

            Console.WriteLine("Podaj swoje imię: ");
            imie = Console.ReadLine()!;
            Console.WriteLine("Podaj swoje nazwisko: ");
            nazwisko = Console.ReadLine()!;
            Console.WriteLine("Podaj adres email: ");
            email = Console.ReadLine()!;

            if (CzyIstniejEmail(email))
            {
                Console.WriteLine("Błąd! Istnieje już konto z tym adresem email.");
                continue;
            }

            Console.WriteLine("Podaj hasło: ");
            haslo = Console.ReadLine()!;
            uprawnienia = Console.ReadLine()!.ToLower() == "admin123" ? "admin" : "user";

            if (!SprawdzPoprawnoscEmail(email) || !SprawdzPoprawnoscHasla(haslo))
            {
                Console.WriteLine("Błąd! Niepoprawne dane wejściowe.");
                continue;
            }

            listaUzytkownikow.Add(new User(imie, nazwisko, email, haslo, uprawnienia));
            zapisywanieUzytkownikowDoPliku("users.txt");
            Console.WriteLine("Rejestracja zakończona pomyślnie!");
            break;
        }
        catch (Exception e)
        {
            Console.WriteLine($"Błąd: {e.Message}");
        }
    } while (true);
}
```

Rejestracja użytkownika

Wejście:

- Imię: **"Jan"**
- Nazwisko: **"Kowalski"**
- Email: **"jan.kowalski@example.com"**

- Hasło: **"Haslo123"**

Nowe połaczenie:

```
static void DodajPolaczenie()
{
    Console.WriteLine("Wprowadź nowe połączenie (format: MiastoA - MiastoB, Godzina Odjazdu, Godzina Przyjazdu, Cena):");
    string nowePolaczenie = Console.ReadLine();
    string[] czesci = nowePolaczenie.Split(',');

    if (czesci.Length < 4)
    {
        Console.WriteLine("Niepoprawny format danych.");
        return;
    }

    TimeSpan czasOdjazdu = TimeSpan.Parse(czesci[1].Trim());
    TimeSpan czasPrzyjazdu = TimeSpan.Parse(czesci[2].Trim());
    if (czasPrzyjazdu < czasOdjazdu)
    {
        czasPrzyjazdu = czasPrzyjazdu.Add(new TimeSpan(24, 0, 0));
    }

    TimeSpan czasDojazdu = czasPrzyjazdu - czasOdjazdu;
    rozkladJazdy.Add($"{czesci[0]}, {czesci[1]}, {czesci[2]}, {czasDojazdu.TotalMinutes} min, {czesci[3].Trim()} PLN");
    Console.WriteLine("Połączenie dodane.");
}
```

Dodanie połączenia do rozkładu jazdy

Wejście:

- Miasto: **"Warszawa - Kraków"**
- Godzina odjazdu: **"08:00"**
- Godzina przyjazdu: **"12:00"**
- Cena biletu: **"150 PLN"**

Wynik:

- Połączenie dodane do listy **rozkladJazdy** i zapisane w pliku **rozklad_jazdy.txt**.

Filtrowanie połaczeń:

```
static void FiltrujPoCzasie()
{
    Console.WriteLine("Podaj maksymalny czas dojazdu (minuty):");
    int maxCzas = Convert.ToInt32(Console.ReadLine());

    var wyniki = rozkladJazdy.Where(p =>
    {
        string[] czesci = p.Split(',');
        if (czesci[3].Trim().EndsWith("min") && double.TryParse(czesci[3].Trim().Replace("min", "").Trim(), out double czas))
        {
            return czas <= maxCzas;
        }
        return false;
    });

    foreach (var polaczenie in wyniki)
    {
        Console.WriteLine(polaczenie);
    }
}
```

Filtracja połączeń

Scenariusz: Filtrowanie po cenie **"do 100 PLN"**.

Wejście:

- Wybór opcji filtrowania: **2**
- Maksymalna cena: **100 PLN**

Wynik:

- Wyświetlono listę połączeń, które spełniają kryterium cenowe.