Złożenie pracy online:
2022-03-15 17:22:04
Kod pracy:
8381/38366/CloudA

Marcin Bielawski
(nr albumu: 23575)

Praca inżynierska

Deklaratywny UI Framework dla platformy .NET

Declarative UI Framework for .NET platform

Wydział: Wydział Nauk Społecznych i Informatyki

Kierunek: Informatyka

Specjalność: programowanie aplikacji biznesowych

Promotor: dr Henryk Telega

Chciałbym złożyć serdeczne podziękowania mojemu promotorowi dr. Henrykowi Teledze za wyrozumiałość, pomoc i poświęcony mi czas podczas pisania pracy inżynierskiej.

Streszczenie

Celem pracy jest zaprojektowanie narzędzia dla platformy .NET, które umożliwi programiście tworzenie interfejsów użytkownika. Projekt zakłada stworzenie UI Frameworka, z którego wykorzystaniem programista może budować drzewa komponentów w sposób deklaratywny. Framework we własnym zakresie obsługuje procesy renderowania, umożliwia elastyczne pozycjonowanie i wymiarowanie komponentów, oraz dostarcza system stylowania dzięki któremu można w przystępny sposób kontrolować wyglądem stworzonego interfejsu. Framework powinien dodatkowo obsługiwać interakcję końcowego użytkownika z interfejsem za pomocą myszy i klawiatury. Efektem pracy jest działający UI Framework dzięki któremu programista w może prosto i szybko tworzyć dobrze wyglądające aplikacje dla platformy .NET.

Słowa kluczowe

.net, c#, ui, framework,skia,yogalayout,interfejs użytkownika

Abstract

The purpose of this project is to design a tool for the .NET platform that will enable a way for programmer to create user interfaces. The project involves the creation of a UI Framework with which a developer can build component trees by using declarative programming paradigm. The framework handles rendering processes on its own, allows flexible positioning and dimensioning of the components, and provides a styling system that provides control of the created interface's appearance in an accessible way. The framework should additionally support end-user interaction with the interface using the mouse and keyboard. The result of the work is a working UI Framework, where the programmer can easily and quickly create good-looking applications for the .NET platform.

Keywords

.net, c#, ui, framework,skia,yogalayout,user interface

SPIS TREŚCI

1 WPROWADZENIE

1.1 CEL PRACY

Celem niniejszej pracy jest opracowanie prostego w użyciu UI Frameworka(ang. User Interface Framework) dla platformy .NET pozwalającego programiście na tworzenie interfejsów użytkownika z wykorzystaniem paradygmatu deklaratywnego programowania. Narzędzie powinno w łatwy sposób umożliwić kontrolę nad formą prezentacji komponentów wykorzystując system stylowania, zapewnić prosty i skuteczny sposób elastycznego rozmieszczania i wymiarowania elementów wykorzystując koncept Flex Box Layout. Ponadto projekt powinien dostarczać podstawowe komponenty umożliwiające interakcję użytkownika za pomocą myszy i klawiatury.

1.2 ZAKRES PRACY

W rozdziale drugim przeprowadzono przegląd istniejących rozwiązań, dzięki czemu można dowiedzieć się jakie aktualnie technologie do tworzenia interfejsów istnieją na platformie .NET. Opisano cechy tych rozwiązań, co pomogło określić w jakim kierunku powinno iść dalsze projektowanie UI Frameworka. W kolejnym rozdziale opisane jest jakie technologie i dlaczego zostały wykorzystane w projekcie. W czwartym rozdziale autor skupia się na opisaniu struktury projektu. W rozdziale piątym autor opisuje implementację projektu, wytłumaczone zostały wybory zastosowanych paradygmatów oraz napotkanych problemów i ich rozwiązań w czasie projektowania. W szóstym rozdziale autor prezentuje sposób działania i wykorzystanie stworzonego UI Frameworka poprzez utworzenie prostych interfejsów użytkownika.

2 AKTUALNIE DOSTĘPNE ROZWIĄZANIA

2.1 XAMARIN.FORMS

To otwartoźródłowy UI Framework od firmy Microsoft. Służy do tworzenia aplikacji na takich platformach jak iOS, Android oraz Windows. Jest to najbardziej popularne rozwiązanie do tworzenia interfejsów w środowisku .NET. Umożliwia deklaratywne budowanie interfejsów za pomocą języka XAML(ang. Extensible Application Markup Language) – opartym na języku XML.

Xamarin.Forms ma wbudowaną obsługę wzorca projektowego MVVM(ang. Model-View-ViewModel) za pośrednictwem „data binding" – właściwość **BindingContext**, umożliwia wiązanie Widoku(View) z właściwościami w Modelu Widoku(ViewModel) .

2.2 .NET MAUI

.NET MAUI czyli z ang. *.NET Multi-platform App UI* to rozwiązanie które jest ewolucją Xamarin.Forms. Na obecną chwilę jest w wersji poglądowej, jednak zespół .NET sprawnie pracuje nad wydaniem stabilnej wersji produkcyjnej. W sferze tworzeniu interfejsów użytkownika, MAUI otrzymuje takie usprawnienia jak wsparcie dla wzorców MVU(Model-View-Update), RxUI(ReactiveUI).

2.3 FLUTTER UI

Flutter to produkt od Google pisany w języku Dart. Oznacza to że nie jest dostępny dla środowiska .NET, jednak został uwzględniony w tym zestawieniu ponieważ nasz framework jest po części nim inspirowany. W tym frameworku, każdy komponent interfejsu jest nazywany Widżetem. W przeciwieństwie do Xamarin.Forms, Flutter nie używa natywnych kontrolek przy tworzeniu interfejsów – zamiast tego Widżety renderowane są za pomocą biblioteki Skia. Rozwiązanie to pozwala na uzyskanie takiego samego wyglądu aplikacji, niezależnie na jakiej platformie została uruchomiona aplikacja. W przypadku, jeżeli programista chciałby jednak korzystać z natywnego wyglądu aplikacji, ma możliwość zastosowania takich Widżetów, które sprytnie imitują wygląd natywnych elementów – służą do tego komponenty Material(dla Androida) oraz Cupertino(dla iOS).

3 WYKORZYSTANE TECHNOLOGIE

Decyzje o wykorzystanych technologiach podjęto z myślą o następujących cechach: otwartoźródłowość, popularność, aktywność projektu, oraz skala zastosowania w innych projektach.

Wybrane technologie są tworzone i rozwijane przez największe firmy z branży IT. Są one stosowane w wielu znanych projektach jak np. Google Chrome(w przypadku Skia) czy React Native(Yoga).

3.1 SKIA

Skia to otwartoźródłowy silnik graficzny 2D napisany w języku C++, stworzony przez firmę Skia Inc., od roku 2005 rozwijany przez firmę Google, która w 2008 roku upubliczniła jego kod źródłowy. Wykorzystywany jest w takich projektach jak Google Chrome, Chrome OS, Mozilla Firefox, oraz takie UI Frameworki jak Flutter i Avalonia.

Zapewnia on interfejs programowania aplikacji (ang. application programming interface, w skrócie API) przeznaczony do rysowania takich elementów jak tekst, bitmapy, grafiki wektorowe. Proces rysowania jest możliwy na wielu platformach z wykorzystaniem procesora(ang. central processing unit, CPU) jak i karty graficznej(ang. graphics processing unit, GPU).

3.2 SKIASHARP

Jako iż nasz projekt jest przeznaczony dla platformy .NET, wykorzystana została biblioteka SkiaSharp, dzięki której mamy zapewniony dostęp do API Skia dla platformy .NET.

3.3 YOGA LAYOUT

Yoga to otwartoźródłowy silnik do wszelkich kalkulacji związanych z pozycjonowaniem elementów UI. Został stworzony przez firmę Facebook. Używają go takie UI Frameworki jak Litho, ComponentKit oraz React Native. Implementuje on Flexbox, działający na takiej samej zasadzie jak ten znany z kaskadowych arkuszy stylów(ang. Cascading Style Sheets, **CSS**). Jego główna implementacja jest napisana w języku C++, jednak tak samo jak w przypadku Skia, dostępne są rozwiązania umożliwiające użycie go w wielu środowiskach(w naszym przypadku .NET).

Użycie tej technologii znacznie przyśpiesza proces tworzenia naszego projektu zrzucając z nas ciężar tworzenia kompleksowych obliczeń do pozycjonowania elementów oraz dając pewność poprawności ich wyników.

3.4 RICHTEXTKIT

Co prawda SkiaSharp umożliwia wyświetlanie tekstu, jednak domyślnie jest ono dosyć proste. RichTextKit od toptensoftware to biblioteka dla SkiaSharp pozwalająca na bardziej zaawansowane renderowanie tekstu. Umożliwia nam stylowanie tekstów(np. czcionki, kolory, rysowanie emoji), zaznaczanie go, obliczanie rozmiarów tekstu, wyrównywanie(ang. align) i wiele innych ułatwień.

4 STRUKTURA PROJEKTU

4.1 WIDGET

To główna klasa dla wszystkich komponentów interfejsu. Oznacza to że każdy komponent w hierarchii drzewa dziedziczy po tej klasie. Zawiera on wszystkie wartości potrzebne do procesu renderowania interfejsu. Jego fundamentalne pola to:

_yogaNode

Jest to obiekt klasy YogaNode, pochodzący z biblioteki YogaLayout. Jest ono używane w procesie kalkulacji layoutu.

Style

To obiekt klasy WidgetStyle. Przechowuje on niedziedziczony styl widżetu. Jest deklarowany przez programistę przy tworzeniu widżetu.

_resolvedStyle

To prywatne pole zawierające obiekt klasy WidgetStyle. W przeciwieństwie do Style, przechowuje on swój styl, razem z dziedziczonymi wartościami. To pole jest wykorzystywane w dalszym procesie renderowania komponentu.

Children

To Lista przechowująca wszystkie dzieci tego komponentu. Jest to jeden z fundamentów w procesie tworzenia drzewa komponentów.

Parent

To właściwość przechowująca komponent, który jest rodzicem tego komponentu.

OnMouseDownEvent, OnMouseUpEvent, OnKeyDownEvent, OnKeyUpEvent, MouseMoveEvent, OnTextInputEvent

To zdarzenia wywoływane przez interakcje użytkownika, uruchamiane są za pomocą **InputEventManager**.

Ponadto zawiera metody

Initialize

Jest wywoływana przed procesem renderowania. Jest to metoda odpowiedzialna za przekazanie wartości do *_yogaNode* oraz uruchomienie procesu rozstrzygającego style(tutaj uruchomiana jest metoda **StyleResolver,Resolve**)

Render

To główna metoda odpowiedzialna za rysowanie Widżeta.

4.2 FLEXBOX

To widżet którego dzieci są ustawiane elastycznie, według zdefiniowanych wcześniej wartości.

4.3 ROW, COLUMN

To widżety działające za pomocą flexbox, z tak ustawionymi wartościami aby horyzontalne(row) i wertykalne(column) pozycjonowanie było uproszczone. Sposób ich działania jest inspirowany biblioteką Bootstrap(html, css).

4.4 TEXT

To widżet odpowiadający za wyświetlanie tekstu. Działa z wykorzystaniem biblioteki RichTextKit.

4.5 TEXTFIELD

To widżet pola tekstowego. Odpowiedzialny jest za wyświetlanie tekstu kontrolowanego przez zawierany przez siebie obiekt klasy TextFieldController.

4.6 TEXTFIELDCONTROLLER

Obsługuje zdarzenia związanie z interakcją użytkownika z polem tekstowym, co oznacza że wpływa na wartość TextField.

4.7 StyleKey

To atrybut używany przy metodach zwracających obiekt klasy WidgetStyle. Informuje on StyleResolver o tym, że ta metoda powinna być uwzględniona w procesie wyszukiwania stylu.

4.8 WidgetStyle

To klasa przechowująca wartości opisujące wygląd widżeta.

4.9 StyleResolver

To klasa zarządzająca stylami. Jest odpowiedzialna za wyszukiwanie obiektów WidgetStyle dla obiektów klasy Widget poprzez metody zawierające atrybut StyleKey.

4.10 Border, BoxShadow, Offsets, RoundBox

To klasy przeznaczona dla WidgetStyle. Przechowują wartości opisujące inne specjalne właściwości WidgetStyle.

4.11 InputEventManager

To główna klasa służąca do obsługiwania interakcji myszy oraz klawiatury. Po przechwyceniu takiego zdarzenia, przekieruje go do docelowego komponentu obsługującego to zdarzenie.

5 IMPLEMENTACJA

5.1 DEKLARATYWNOŚĆ W TWORZENIU INTERFEJSU

Jeszcze do niedawna, przy tworzeniu interfejsów użytkownika jednym z najbardziej popularnym paradygmatem było imperatywne programowanie. Proces ten polega na osiągnięciu celu krok po kroku, skupia się na tym w jaki sposób wykonać czynność. W przypadku stworzenia elementu UI, dla przykładu, chcemy utworzyć niebieski przycisk o rozmiarach 100px na 30px, z tekstem „Witaj świecie”. Do osiągnięcia tego celu musimy wykonać następujące czynności:

- Utwórz nowy obiekt przycisku
- Ustaw długość przycisku na 100px
- Ustaw szerokość przycisku na 30px
- Ustaw tekst „Witaj świecie”
- Ustaw kolor niebieski

```
var button = new Button();
button.Width = 100;
button.Height = 30;
button.Text = "Witaj świecie";
button.Color = Colors.Blue;
```

Rysunek 1: Pseudokod tworzenia przycisku

W przypadku zmiany stanu aplikacji, np. gdy chcemy zmienić tekst przycisku, musimy zrobić to manualnie - odwołać się do jego obiektu i zmienić jego wartość:

```
button.Text = "Nowy tekst";
```

Rysunek 2: Manualna zmiana tekstu przycisku

Deklaratywne programowanie UI opisuje to co chcemy widzieć przy danym stanie, bez skupiania się na tym w jaki sposób należy to zrobić – decyduje o tym framework. W takim razie, jeżeli chcemy otrzymać taki sam przycisk jak w poprzednim przykładzie, wyrażenie wyglądałoby tak:

Chcę niebieski przycisk o szerokości 100px, wysokości 30px i napisem „Witaj świecie”

```
State<string> hello = "Witaj Świecie";
Button(
    text: hello,
    color: Colors.Blue,
    width: 100,
    height: 30
)
```

Rysunek 3: Deklaratywne tworzenie przycisku

W tym momencie po zmianie wartości pola „hello”, następuje zmiana stanu co skutkuje przebudowaniem interfejsu od nowa i przerysowaniem go – zatem tekst przycisku automatycznie ulegnie zmianie.

5.2 STATYCZNE KONSTRUKTORY

Aby stworzyć nowy obiekt w języku C# należy posłużyć się operatorem *new*. Przez to że nasz framework używa paradygmatu deklaratywności, znaczna ilość kodu musi zawierać ten operator co może znacząco zmniejszyć czytelność kodu. Aby rozwiązać ten problem, posłużono się metodami statycznymi w celu utworzenia nowej instancji obiektu. Każda taka metoda zwraca odpowiedni dla siebie obiekt Widget oraz posiada argumenty odpowiednie do właściwości komponentu. Dzięki temu że jest to metoda, nie ma potrzeby używania operatora *new*.

```
new FlexBox
{
    Width = YogaValue.Percent(100),
    Height = YogaValue.Percent(100),
    Children = new[]
    {
        new FlexBox()
        {
            Children = new[]
            {
                new Text
                {
                    Data = "Witaj"
                },
                new Text
                {
                    Data = "Świecie"
                }
            }
        }
    }
};
```

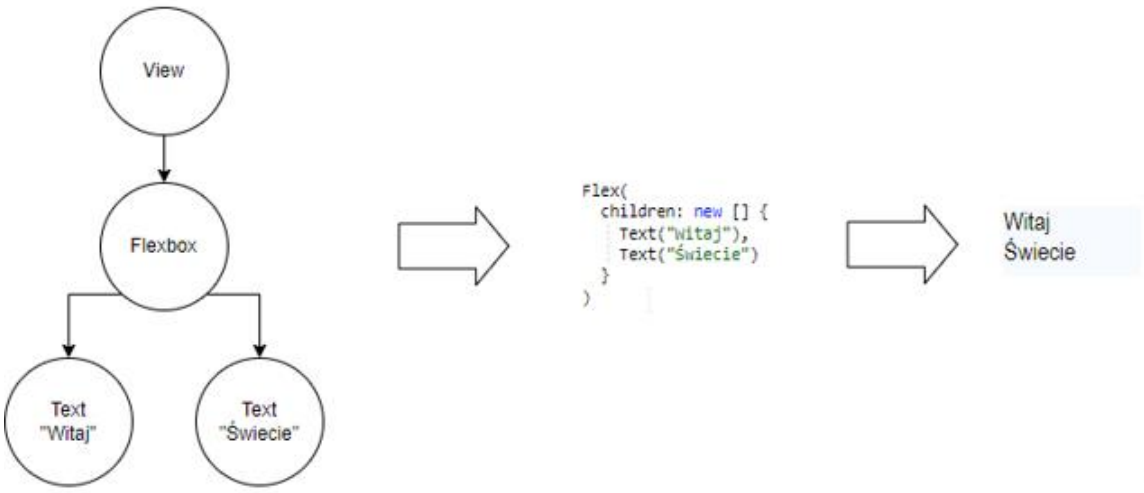
```
Flex(
    width: YogaValue.Percent(100),
    height: YogaValue.Percent(100),
    children: new[]
    {
        Flex(
            children: new[]
            {
                Text(data: "Witaj"),
                Text(data: "Świecie")
            }
        )
    }
);
```

Rysunek 4: Różnica pomiędzy inicjalizerem obiektów a użyciem statycznego kontruktora

5.3 OPIS PROCESÓW ZACHODZĄCYCH WE FRAMEWORKU

5.3.1 Utworzenie drzewa komponentów

W naszym frameworku wszystkie komponenty są obiektami klasy Widget. Każdy Widget może posiadać rodzica oraz dzieci, które również są Widgetami. W takim razie tworzenie drzewa komponentów polega na utworzeniu nowego obiektu klasy Widget, wraz z wybranymi właściwościami. Następnie można ustawić właściwość **children**, tworząc kolejne elementy.



Rysunek 5: Drzewo komponentów, jego odzwierciedlenie w kodzie oraz efekt końcowy

5.3.2 Załadowanie stylu komponentu za pomocą StyleResolver

Klasa **StyleResolver** odpowiedzialna jest za przechowywanie metod zwracających obiekt typu **WidgetStyle** oraz rozpoznawanie ich za pomocą atrybutu **StyleKey.**

```
public class TestStyleResolver: StyleResolver {

  [StyleKey("nav-title")]
  public WidgetStyle NavTitle() => new() {
    TextAlignment = TextAlignment.Center,
      FontColor = DefaultTextColor,
      FontFamily = "Roboto",
      FontWeight = 600,
      FontSize = 18,
      Margin = new Offsets(0, 25, 0, 25)
  };
}
```

Rysunek 6: Przykład definiowania stylu komponentu z kluczem "nav-title"

Każdy komponent zawiera w sobie właściwość klasy **WidgetStyle**. Tę właściwość można zdefiniować za pomocą metody **StyleResolver.FromKey.** Jako argument przyjmuje ona typ tekstowy **string**. Proces ten przypomina użycie atrybutu class w języku HTML, który po zdefiniowaniu jest odszukiwany w stylu CSS.

```
Flex(
  style: StyleResolver.FromKey("nav-title"),
  children: Text("MENU", wrap: false)
)
```

Rysunek 7: Użycie StyleResolver.FromKey przy budowaniu komponentu

Argument metody **FromKey** odpowiada wartości atrybutu **StyleKey**. Metoda ta poprzez refleksję pobiera wszystkie metody w klasie nadrzędnej zawierające ten atrybut. Następnie porównuje nazwę atrybutu z argumentem z jakim została wywołana i jeżeli pasują do siebie, wywoływana jest ta metoda. Po wywołaniu tej metody otrzymany jest obiekt klasy **WidgetStyle,** który jest zwracany.

StyleResolver obsługuje również dynamiczne klucze styli. Daje to możliwość definiowania metod z argumentami.

```
[StyleKey("m-{index}")]
public WidgetStyle Margins(string index) => new() {
  Margin = new Offsets(5 * int.Parse(index))
};
```

Rysunek 8: Używanie argumentów w kluczu stylu

Argument metody **FromKey** może być oddzielony spacją. Rozpoznane to zostanie jako kolejny klucz, dla którego również zostanie załadowany styl, który następnie jest łączony z głównym stylem za pomocą metody **Add**.

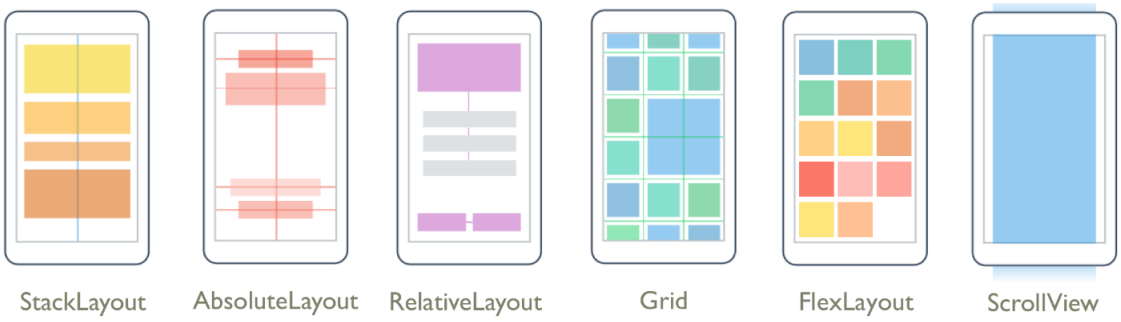
Kolejnym krokiem ładowania stylu jest dziedziczenie cech po rodzicu, podobnie jak w CSS. Dzieje się to za pomocą metody **Inherit**. Dzięki temu procesowi, wystarczy że w jednym komponencie zdefiniowany jest styl np. kolor tekstu, a wszystkie jego dzieci otrzymają tę samą

cechę. Z dziedziczenia jednak wyłączone zostały właściwości związane z pozycjonowaniem, ponieważ mogły powodować one nieoczekiwane efekty.

5.3.3 Pozycjonowanie i wymiarowanie komponentów.

Gdy drzewo komponentów jest już zbudowane, kolejnym krokiem jest przeprowadzenie kalkulacji opisujących jaki układ mają one prezentować. Najprostszym sposobem zaimplementowania tej funkcjonalności byłoby wpisywanie pozycji i rozmiarów „na sztywno" – czyli ręczne ustawienie pozycji każdego z komponentów. Jednak wtedy, okno aplikacji musiałoby posiadać stały rozmiar. Takie rozwiązanie jest przydatne gdy naszym celem jest statyczne osadzenie niektórych komponentów, jednak nie jest możliwe stworzenie bardziej złożonego interfejsu.

Mając to na uwadze, należało zdecydować w jaki sposób rozwiązać ten problem. Każda dojrzała biblioteka do tworzenia interfejsów użytkownika korzysta z systemów układów(Layout). System layoutów określa zasady, w jaki sposób jego komponenty powinny być ustawiane. Jest wiele ich rodzajów, Rys.4 przedstawia listę zaimplementowanych layoutów przez Xamarin.Forms.



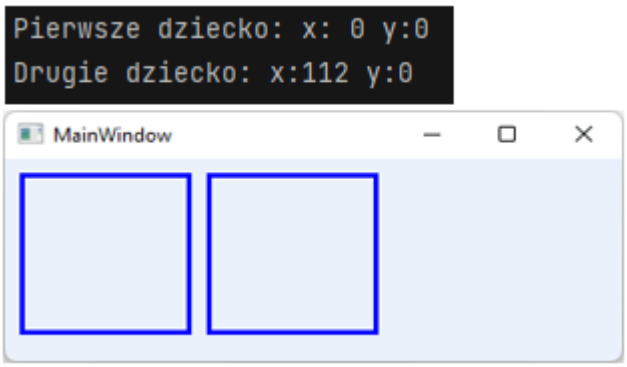
Rysunek 9: Najpopularniejsze rodzaje systemów layout na przykładzie Xamarin.Forms

Podjęto decyzję o zaimplementowaniu układu typu Flex. Jest to rozwiązanie które swoją popularność zawdzięcza CSS, na której oparte jest wiele implementacji(wzorowano na nim FlexLayout w Xamarin.Forms lub Angular Flex-Layout). Dzieci układu Flex mogą być ułożone wertykalnie jak i horyzontalnie, elastycznie się powiększać w celu wypełnienia wolnej przestrzeni lub zmniejszać aby nie pozwolić na przekroczenie granic swojego rodzica. Niestety stworzenie takiego systemu jest skomplikowane, co wiąże się z dużymi nakładami pracy. Aby uniknąć kłopotów związanych z implementacją tego systemu, została użyta biblioteka **Yoga-Layout**. Tak jak wiele podobnych rozwiązań, została zaprojektowana w oparciu o moduł CSS FlexBox Layout.

YogaLayout działa na zasadzie drzewa elementów, którego węzeł jest opisywany za pomocą klasy **YogaNode**. Zawiera w sobie właściwości opisujące w jaki sposób pozycjonować elementy, informacje o ich rozmiarach, marginesach itp., które są wykorzystywane do dostarczenia wyników obliczeń za pomocą metody **CalculateLayout**. Rys.5 przedstawia podstawowe zastosowanie tej klasy w celu ustawienia dwóch elementów obok siebie.

```
private void CalculateTest() {
  root = new YogaNode();
  var child1 = new YogaNode {
    Width = 100,
      Height = 100,
      MarginLeft = 10,
      MarginTop = 10
  };

  var child2 = new YogaNode {
    Width = 100,
      Height = 100,
      MarginLeft = 12,
      MarginTop = 10
  };

  root.FlexDirection = YogaFlexDirection.Row;
  root.AddChild(child1);
  root.AddChild(child2);
  root.CalculateLayout(500, 500);

  Console.WriteLine($"Pierwsze dziecko: x: {child1.LayoutX} y:{child1.LayoutY}");
  Console.WriteLine($"Drugie dziecko: x:{child2.LayoutX} y:{child2.LayoutY}");
}
```

Rysunek 10: Przykład wykorzystania YogaLayout w kodzie



Rysunek 11 Wynik testowej operacji oraz jego wizualizacja.

5.3.4 Wykorzystanie biblioteki YogaLayout w naszym frameworku

Mając kompletne drzewo komponentów, należy przejść do klasy **Widget**. Posiada ona pole typu **YogaNode** o nazwie **_yogaNode**. Klasa ta zawiera również metodę **Initialize()**, która jest odpowiedzialna za załadowanie styli za pomocą **StyleResolver,** a następnie przerzucenie wszystkich wartośći związanych z pozycjonowaniem z pola **_resolvedStyle** do pola **_yogaNode**.

```
public virtual void Initialize()
{
    _yogaNode = new YogaNode();
    _resolvedStyle = StyleResolver.Resolve(this);

   _yogaNode.MaxWidth = _resolvedStyle.MaxWidth ?? YogaValue.Undefined();
   _yogaNode.MaxHeight = _resolvedStyle.MaxHeight ?? YogaValue.Undefined();

   _yogaNode.Width = Width ?? YogaValue.Undefined();
   _yogaNode.Height = Height ?? YogaValue.Undefined();

   _yogaNode.AlignContent = AlignContent ?? YogaAlign.Auto;

   ...
```

Rysunek 12: Fragment metody Initialize()

Następnie na każdym węźle, poprzez rekurencję, wywoływana jest metoda **Initialize()**. Po tej operacji, w korzeniu drzewa wywoływana jest metoda **_yogaNode.CalculateLayout,** która zajmuje się niezbędnymi obliczeniami zgodnie z systemem układu Flex.

5.3.5 Przechwytywanie zdarzeń związanych z interakcją użytkownika

Wiedząc już gdzie będą znajdować się komponenty, można przejść do obsługi zdarzeń związanych z interakcją użytkownika. Ten proces jest obsługiwany przez klasę

InputEventManager. Zawiera ona pewną abstrakcję, do której dostarczane są zdarzenia z zewnątrz. Na przykładzie aplikacji desktopowej, wykorzystano zdarzenia pochodzące z obiektu klasy **System.Windows.Window**. W klasie MainWindow, zarejestrowane takie zdarzenia jak MouseMove, MouseDown, MouseUp, KeyUp, KeyDown, TextInput. Następnie przekierowywane są do odpowiednich dla nich zdarzeń w klasie **InputEventManager**.

W przypadku uruchomienia zdarzeń związanych z interakcją kursorem myszy, operacja zwana *hit test*. Polega ona na sprawdzeniu jakie elementy drzewa znajdują się pod pozycją kursora myszy. Uwzględnia się wtedy pozycję absolutną komponentu oraz jego wymiary, a następnie porównuje czy pole to zawiera pozycję kursora. Ta czynność jest obsługiwana przez metodę **HitTest**, i zwraca ona listę komponentów w kolejności zależnej od tego który element jest najbardziej na wierzchu.

```
private IEnumerable <Widget> HitTest(Point eventPoint, Func <Widget, bool> matcher) {
  var widgets = new List <Widget> ();
  var allWidgets = GetWidgets(matcher);

  foreach(var widget in allWidgets) {
    var x = (int) widget.CalculateX();
    var y = (int) widget.CalculateY();
    var width = (int) widget._yogaNode.LayoutWidth;
    var height = (int) widget._yogaNode.LayoutHeight;
    var rect = new Rectangle(x, y, width, height);

    if (rect.Contains((int) eventPoint.X, (int) eventPoint.Y)) {
      widgets.Insert(0, widget);
    }
  }

  return widgets;
}
```

Rysunek 13: Implementacja metody HitTest

Po zwróceniu listy, na pierwszym jej elemencie uruchamiane jest odpowiednie dla danej interakcji zdarzenie, jeżeli komponent posiada takie zarejestrowane(klasa **Widget** i zdarzenia myszy). Przy kliknięciu myszą na komponent, dodatkowo za każdym razem jeżeli zawiera on flagę **Focusable**, zmieniana jest flaga **Focused** na **true,** a pozostałym komponentom ta flaga dostaje wartość **false**.

Przechwytywanie zdarzeń klawiatury podlega podobnej zasadzie, jednak pomijana jest operacja hit test. Zamiast tego, przy iteracji sprawdzana jest flaga **Focused**. Jeżeli jest ona ustawiona na **true**, uruchamiane są zdarzenia klawiaturowe na tym komponencie. Zaimplementowano również obsługę przycisku Tab, który po naciśnięciu zaznacza kolejny komponent – jest to element nawigacji za pomocą klawiatury.

5.3.6 Rysowanie komponentów

Kiedy wszystkie komponenty zostały zmierzone i ustawione na swoje pozycje, nadszedł czas na wyświetlenie ich w widoku aplikacji.

Najpopularniejsze rozwiązanie dla .NET, Xamarin.Forms, cechuje się tym, że tworzy pewną warstwę interfejsu pomiędzy sobą a natywnymi kontrolkami na każdym systemie z osobna. Efekt tego jest taki, że końcowa aplikacja na każdym urządzeniu korzysta z takich kontrolek, jakie są dostępne na systemie na którym została uruchomiona.

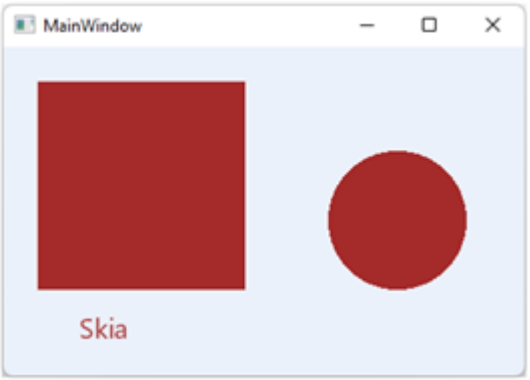
Rysunek 14: Efekt zastosowania natywnych komponentów

Takie rozwiązanie może ograniczyć kontrolę nad wyglądem aplikacji. Przykładowo, jeżeli chcemy całkowicie zmienić wygląd kontrolki, tak aby na każdym systemie wyglądała tak samo, lub zmienić jej zachowanie musimy skorzystać z niestandardowych rendererów. Wymagane jest wtedy utworzenie nowej klasy kontrolki, a potem dla każdego systemu osobno utworzyć niestandardowy renderer dla niej.

Nasz framework całkowicie pomija ten proces. Zamiast tego, wykorzystuje bibliotekę **SkiaSharp** do samodzielnego rysowania komponentów.

Jest to biblioteka która umożliwia rysowanie kształtów, cieni, bitmap i innych elementów grafiki 2D.

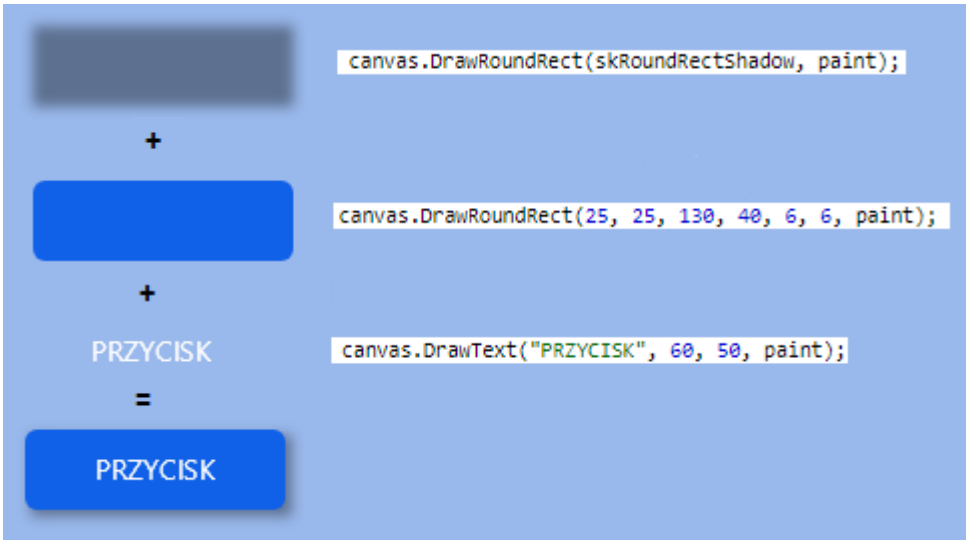
```
private void CanvasView_OnPaintSurface(object? sender, SKPaintSurfaceEventArgs e)
{
    var canvas = e.Surface.Canvas;
    canvas.Clear(SKColor.Parse("ebf1fb"));

    var paint = new SKPaint
    {
        Color = SKColors.Brown,
        TextSize = 20
    };

    canvas.DrawRect(25, 25, 150, 150, paint);
    canvas.DrawCircle(285, 125, 50,  paint);
    canvas.DrawText("Skia", 55, 210,  paint);

}
```



Rysunek 15: Podstawowe kształty rysowane przez Skia

Korzystając ze **SkiaSharp** mamy pełną kontrolę nad tym co wyświetla się w widoku aplikacji. Aby umożliwić rysowanie w oknie aplikacji wykorzystano **SKElement**, który został załadowany w domyślnym oknie aplikacji. Do obiektu tej klasy należy następnie dodać zdarzenie **PaintSurface**. Jest ono wywoływane w momencie przerysowania powierzchni(ang. Surface) – dzieje się to np. przez wywołanie metody **InvalidateVisual()** na obiekcie **SKElement**. Parametr **SKPaintSurfaceEventArgs** daje nam dostęp do wszystkich właściwości takich jak **Surface,** przez którą mamy dostęp do **Canvas** – która udostępnia metody potrzebne do rysowania kształtów.

Mając przygotowane środowisko do rysowania kształtów, można przejść do etapu rysowania komponentów. Podobnie jak w procesie pozycjonowania komponentów, następuje iteracja każdego komponentu w drzewie. Jest ona wywoływana po otrzymaniu wydarzenia **PaintSurface**. Na każdym iterowanym komponencie wywoływana jest metoda **Render**. Interpretuje ona wszystkie wartości z pola **_resolvedStyle** oraz samego komponentu, „składając" w odpowiedniej kolejności.



Rysunek 16: Proces składania trzech elementów w jeden przycisk

Każdy ze składanych elementów na rysunku powyżej to odzwierciedlenie Widżetów. Cień oraz niebieski prostokąt obrazuje Widżet kontenera(**FlexBox**) z ustawionymi wartościami tła oraz cienia. Tekst obrazuje Widżet tekstowy(**Text**) z ustawioną wartością **Data**(wartość tekstowa).

Argumenty liczbowe to pozycje i rozmiary cechujące rysowany widżet. Wartości **skRoundRectShadow**, oraz **paint** są wcześniej interpretowane z klasy WidgetStyle.

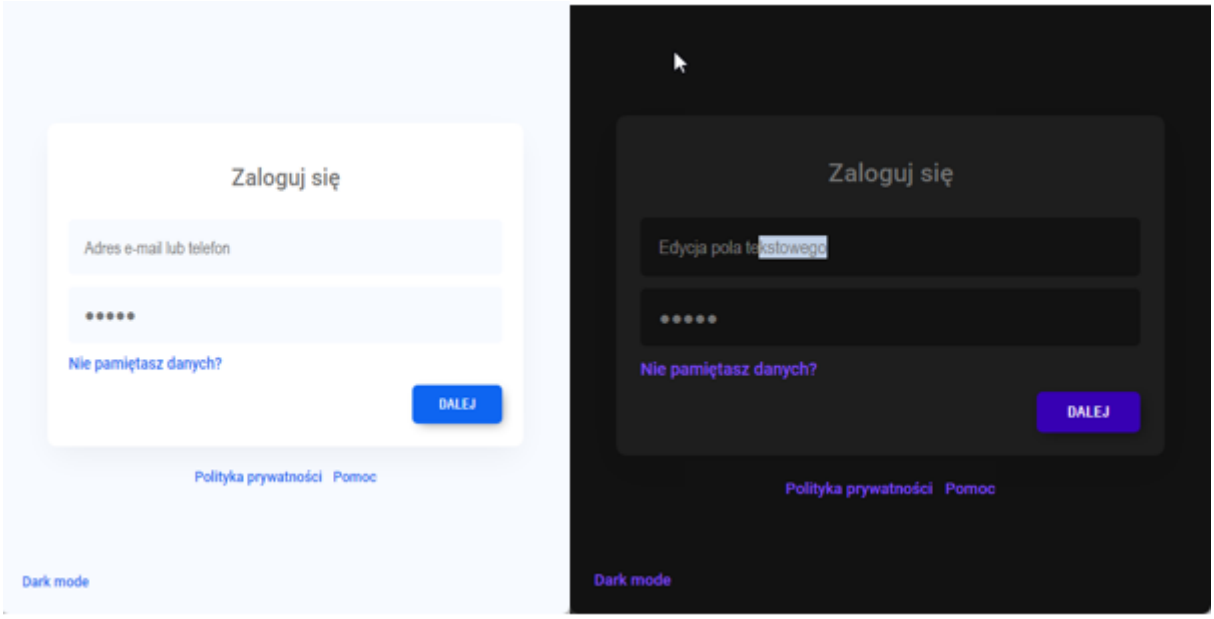
```
paint = new SKPaint {
  Color = _resolvedStyle.BoxShadow.Color ?? SKColors.Empty,
    IsAntialias = true,
    MaskFilter = SKMaskFilter.CreateBlur(SKBlurStyle.Normal, _resolvedStyle.BoxShadow.Blur ?? 0)
};
```

Rysunek 17: Fragment kodu odpowiedzialny za wczytywanie wartości dla BoxShadow poprzez WidgetStyle

6 PRAKTYCZNE WYKORZYSTANIE FRAMEWORKA

6.1 PRZYKŁADOWE WIDOKI

W tym rozdziale zaprezentowane zostaną możliwości utworzonego frameworka na przykładzie fragmentów kodu i efektów renderowania interfejsów.

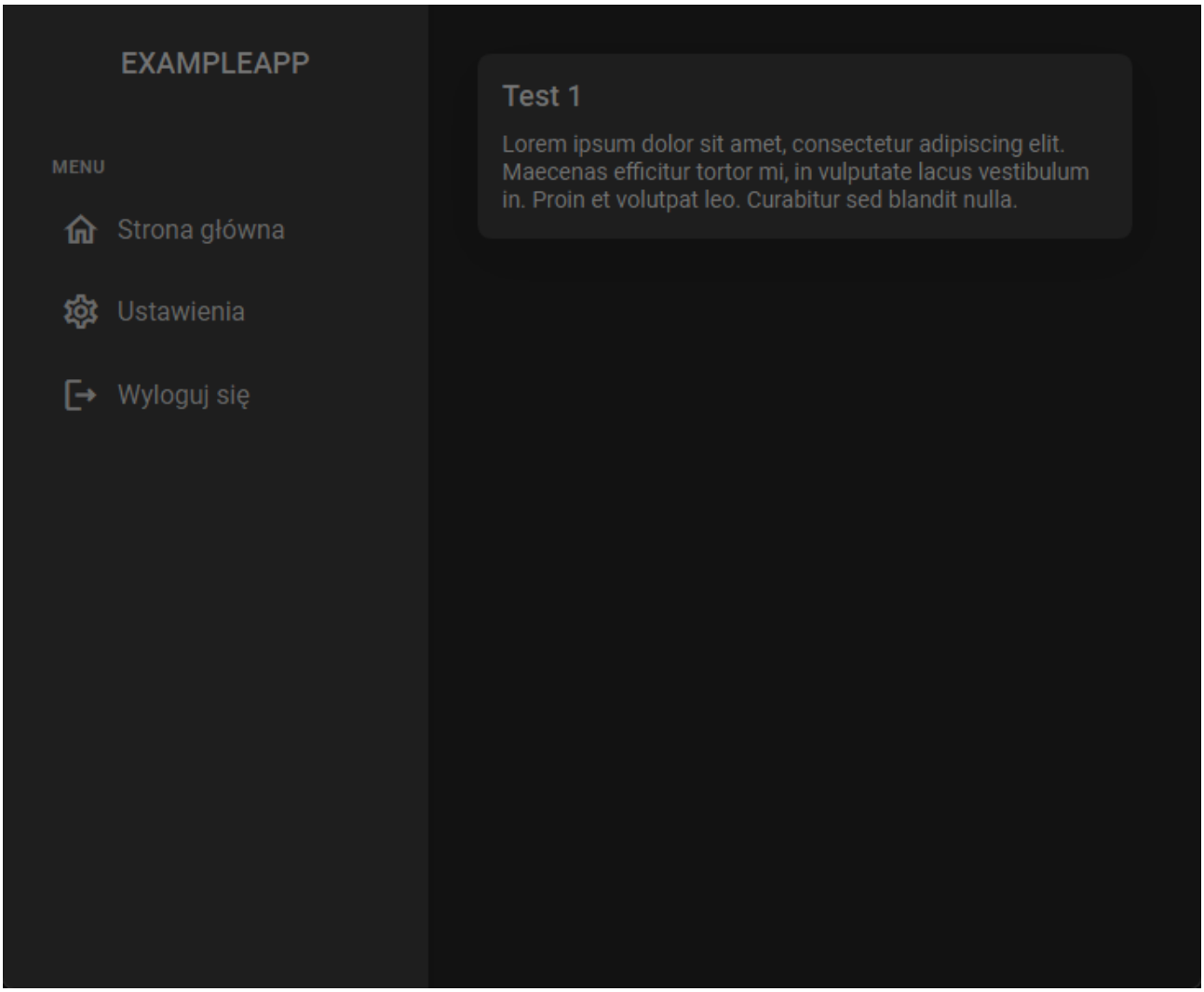


Rysunek 18: Przykładowa strona logowania wraz z trybem nocnym oraz polem tekstowym.

```
//Dark mode
public static bool DarkMode = false;

2 usages
public SKColor? BgColor => DarkMode ? SKColor.Parse("121212") : SKColor.Parse("f7faff");
2 usages
public SKColor? CardColor => DarkMode ? SKColor.Parse("1e1e1e") : SKColors.White;
1 usage
public SKColor? AccentColor => DarkMode ? SKColor.Parse("#3700b3") : SKColor.Parse("0e65f1");
1 usage
public SKColor? AccentLighterColor => DarkMode ? SKColor.Parse("#844dff") : SKColor.Parse("0e65f1");

public SKColor? DefaultTextColor = DarkMode ? SKColor.Parse("dedee3") : SKColor.Parse("#757575");
```

Rysunek 19: Definiowanie wartości oddzielnych dla trybu nocnego oraz dziennego.

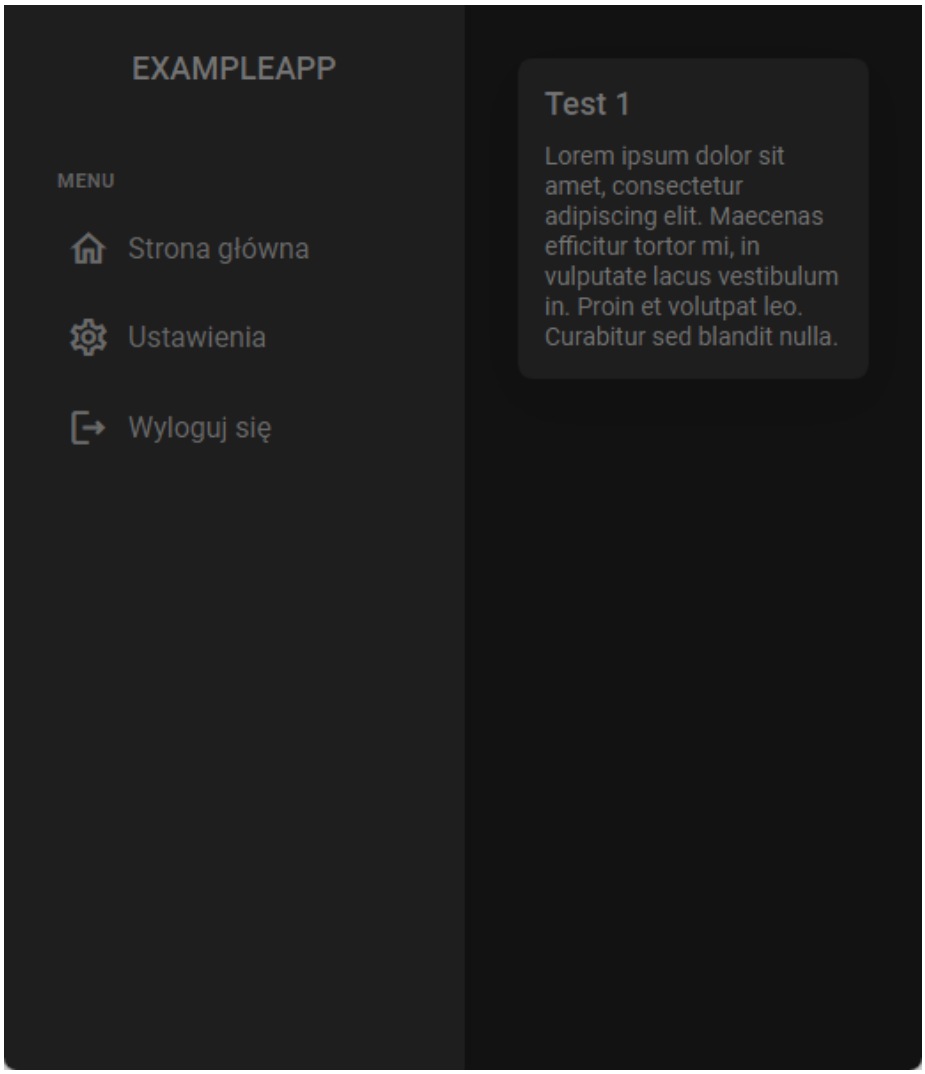
Dzięki elastyczności systemu stylowania, zmiana trybu dziennego na nocny odbywa się za pomocą jedynie kilku linii kodu. Takie rozwiązanie daje nieograniczone możliwości dostosowywania wyglądu aplikacji co jest niezbędnym elementem personalizacji interfejsu przez użytkownika.



Rysunek 20: Popularny rodzaj interfejsu, panel administracyjny

```
Flex(
    style: StyleResolver.FromKey("main-content m-6 mw:400"),
    children:
    Flex(
        style: StyleResolver.FromKey("card shadow-4"),
        children: new[]
        {
            Text("Test 1", wrap: false, style: StyleResolver.FromKey("card-title")),
            Text("Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas efficitur
                tortor mi, in vulputate lacus vestibulum in. Proin et volutpat leo. Curabitur sed
                blandit nulla. ",
                wrap: true,
                style: StyleResolver.FromKey("text-description mt-2")),
        }
    )
)
```

Rysunek 21: Kod tworzący komponent karty(z napisem „Test 1”)



Rysunek 22: Dosotosowanie się interfejsu to rozmiaru okna

7 PODSUMOWANIE

7.1 WNIOSKI

Efektem pracy jest podstawowa wersja UI Frameworka. Zaimplementowane zostały rozwiązania pozwalające na wyświetlenie oraz interakcję ze stworzonymi przez programistę komponentnami, widokami. Rozwiązane zostały problemy języka C# dotyczące składni, dzięki którym deklarowanie komponentów inspirowane Flutterem stało się możliwe.

7.2 PLANY NA ROZWÓJ PROJEKTU

W opisywanej pracy istnieje jeszcze wiele pomysłów do zrealizowania. Podstawowym celem jest stworzenie jej wieloplatformowej wersji – co dzięki architekturze projektu nie będzie dużym wyzwaniem. Następnym krokiem jest rozbudowanie projektu o dodatkowe niezbędne komponenty, oraz takie które jeszcze bardziej ułatwią projektowanie interfejsów i znacząco skrócą ilość pisanego kodu. Projekty takie jak Flutter cechują się tym, że są tworzone przez duże firmy, posiadające znaczne zasoby finansowe oraz ludzkie. Z tego względu ciężko jest dorównać im poziomem zaawansowania, jednak z drugiej strony mogą być pomocne w celu określenia kierunku w jakim nasz projekt powinien się rozwijać.

8 LITERATURA

1. https://yogalayout.com/docs/
2. https://skia.org/docs/
3. https://docs.flutter.dev/resources/architectural-overview
4. https://medium.com/surfstudio/flutter-under-the-hood-binding-2b0ea65e5314
5. https://medium.com/flutter-community/flutter-rendering-under-the-hood-2aa17c40f1f5
6. https://docs.microsoft.com/pl-pl/xamarin/xamarin-forms/user-interface/layouts/choose-layout