Wrocław, 25 marca 2025

Platformy **P**rogramistyczne `.NET` i `Java`

Laboratorium 2

Projekt aplikacji bazodanowej ***.NET***

prowadzący: *Dr inż. Radosław Idzikowski, mgr inż. Michał Jaroszczuk*

1 Cel laboratorium

Celem laboratorium jest zapoznanie się z podstawami projektowania aplikacji bazodanowych oraz korzystania z interfejsu programowania aplikacji (ang. application programming interface, `API`). W ramach zajęć należy stworzyć program bazodanowy w języku `C#`, który będzie rozbudowywany o kolejne funkcjonalności. Praca będzie oceniania na bieżąco podczas zajęć. **Ukończenie każdego etapu powinno być zgłoszone prowadzącemu w celu akceptacji i odnotowania postępów**. Sam program należy umieścić na repozytorium `github` i wysłać zaproszenie do prowadzącego. Czas na wykonanie zadania to dwa zajęcia laboratoryjne. Podczas pierwszego spotkania trzeba wykonać co najmniej jedno zadanie zdefiniowane w Rozdziale 3.

2 Zadania

W ramach zajęć należy w zespołach wykonać następujące zadania:

1. Pobranie i deseralizacja danych z zewnętrznego `API`,
2. Obsługa bazy danych i relacji pomiędzy encjami,
3. Zaawansowane GUI i obróbka danych.

Za wykonanie zadania nr 1 jest ocena dostateczna, za każde kolejne zadanie jest +1 do oceny. Na ocenę bardzo dobrą (5.0) należy wykonać wszystkie trzy zadania. Link do repozytorium należy przesłać dopiero po oddaniu i ocenieniu pracy na laboratorium.

3 Opis Zadań

W tej sekcji zostaną kolejno omówione wszystkie zadania do wykonania podczas laboratorium. Komputery w laboratorium zostały przygotowane do przeprowadzenia zajęć.

3.1 Zadanie 1

W ramach zadania należy wykonać aplikację w technologii `.NET 8.0`, która pozwoli na pobranie oraz wyświetlenie danych z wybranego `API`. Na tym etapie aplikacja może działać w konsoli, jednak zalecane jest tworzenie docelowo aplikacji okienkowej (np.: `MAUI`). Wybór odpowiedniego `API` jest dowolny i bezpośrednio nie ma wpływu na ocenę, jednak nie może być on wulgarny oraz musi zostać zaakceptowany przez prowadzącego. Ponadto istotne jest, żeby interfejs pozwalał na podawanie argumentów, ponieważ ich brak może uniemożliwić uzyskanie maksymalnej oceny za ten etap.

Z reguły `API` opisywane jest z użyciem narzędzia `Swagger`, które pozwala na czytelne przedstawienie dostępnych endpoint'ów i ich parametrów oraz na przetestowania działania serwisu. Przykład można zobaczyć pod poniższym linkiem: TimeApi Poniżej zaproponowane są trzy sprawdzone interfejsy:

1. Open Wheather Api – serwis pogodowy (miasto jako parametr),
2. Open Exchange Rates – serwis z kursem walut (data jako parametr),
3. VIES - VAT Information Exchange System – serwis zwracający dane podatnika (kraj oraz numer VAT ID jako parametry). Warto użyć poniższego linku jako endpoint'a: (dokumentacja na stronie nie jest zaktualizowana):
   `https://ec.europa.eu/taxation_customs/vies/rest-api/ms/PL/vat/5260309174`

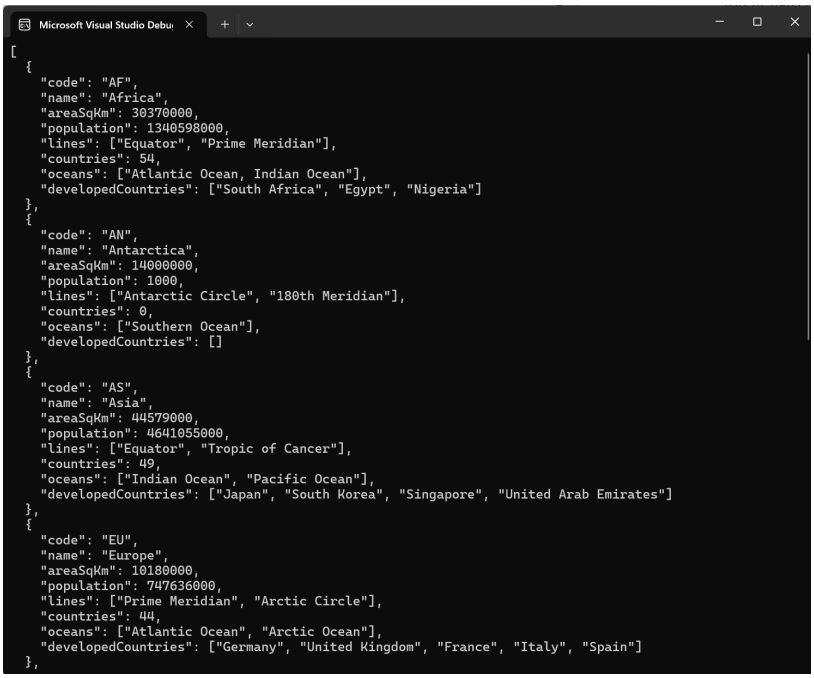
Zaproponowane serwisy posiadają wersję darmową z ograniczoną liczbą zapytań na minutę oraz limitem miesięcznym wystarczającym na potrzeby zajęć. W serwisie pogodowym ponadto będziemy ograniczeni do aktualnej pogody, ale między zajęciami można zebrać kilka próbek. W przypadku serwisu z kursem walut jesteśmy ograniczeni do waluty "USD" jako bazowej. **Mile widziane są własne propozycje!**

Podłączenie do `API`

Głównym celem zadania jest pobranie danych z `API` z poziomu programu. Przykład do uruchomienia w aplikacji konsolowej przedstawiono na Listingu 1. Należy zacząć od utworzenia obiektu klasy `HttpClient`, który jest niezbędny do komunikacji z wykorzystaniem protokołu `http`. Dla aplikacji konsolowej należy następnie wykorzystać klasę `Task`, pozwalającą zdefiniować funkcję wywołującą asynchroniczne zapytanie do `API`. Samo żądanie typu pobierz można wywołać za pomocą metody `GetStringAsync()` lub `GetAsync()`. Należy pamiętać, że wymienione metody są asynchronicznie (ponieważ zakładamy, że `API` potrzebuje pewien czas na zwrócenie odpowiedzi), w związku czym należy wymusić czekanie odpowiedź operatorem `await`, który czeka na zakończenie fragmentu kodu wykonywanego asynchronicznie. Samą funkcję o typie `Task`, po utworzeniu obiektu klasy, z której ona pochodzi, należy uruchomić w głównym programie z metodą `Wait()`, która sprawia iż funkcja będzie oczekiwać na otrzymanie odpowiedzi z `API`. Efekt wywołania programu pokazano na Rysunku 2. Łatwo zaważyć, że odpowiedź jest w formacie `json`.

```
public class APITest {
    public HttpClient client;
    public async Task GetData(){
        client = new HttpClient();
        string call = "https://dummy-json.mock.beeceptor.com/continents";
        string response = await client.GetStringAsync(call);
        Console.WriteLine(response);
    }
}
internal class Program
{
    static void Main(string[] args)
    {
    APITest t = new APITest();
    t.GetData().Wait();
    }
}
```

Listing 1: Przykład pobrania danych z API dla aplikacji konsolowej



Rysunek 2: Dane kontynentów pobrane z `API`

Warto również wiedzieć, jak dane z zewnętrznego `API` pobiera się w aplikacji okienkowej. Sam sposób jest analogiczny do przedstawionego wcześniej na Listingu 1 z tą różnicą, iż nie ma konieczności używania klasy `Task`. W momencie inicjalizacji okna aplikacji, należy podobnie jak wcześniej utworzyć obiekt klasy `HttpClient`, a następnie w metodzie wywoływanej po kliknięciu przycisku wywołać na tym obiekcie funkcję `GetStringAsync()`. Co ważne, należy pamiętać o użyciu operatorów odpowiedzialnych za asynchroniczność - metoda `Click` powinna być asynchroniczna, a sama funkcja wywołana z operatorem `await`. W przykładzie pokazanym na Listingu 3 użyto innego `API`, zwracającego listę zadań do wykonania.

```
1  namespace StudentAPI
2  {
3      public partial class FormStudent : Form
4      {
5          private HttpClient client;
6          public Form1()
7          {
8              InitializeComponent();
9              client = new HttpClient();
10         }
11         private async void buttonDownload_ClickAsync(object sender, EventArgs e)
12         {
13             string call = "https://dummy-json.mock.beeceptor.com/todos";
14             string response = await client.GetStringAsync(call);
15             textBoxResponse.Text = response;
16         }
17     }
18 }
```

Listing 3: Przykład pobrania danych z `API` dla aplikacji okienkowej

W przypadku bardziej rozbudowanych żądań, w których pojawiają się parametry, należy edytować samo zapytanie. W opisanym przypadku jest to zmienna `call`. Najprościej to zrobić poprzez sklejanie odpowiednich fragmentów typu `string`.

Deserializacja

Od `.NET 5.0` do serializacji i deserializacji danych, m. in. w formacie `json`, nie trzeba używać zewnętrznych pakietów `NuGet`. W danych pobranych z `API` pokazanym w poprzednim etapie, znajduje się **lista**, składająca się z rekordów z danymi o zadaniach do wykonania (id użytkownika, id rekordu, zadanie oraz flaga completed). Przed przystąpieniem do deseralizacji, należy stworzyć klasę `ToDo`, pokazaną na Listingu 4, która posłuży w pierwszej kolejności do zmapowania pliku, a następnie do przechowywania informacji.

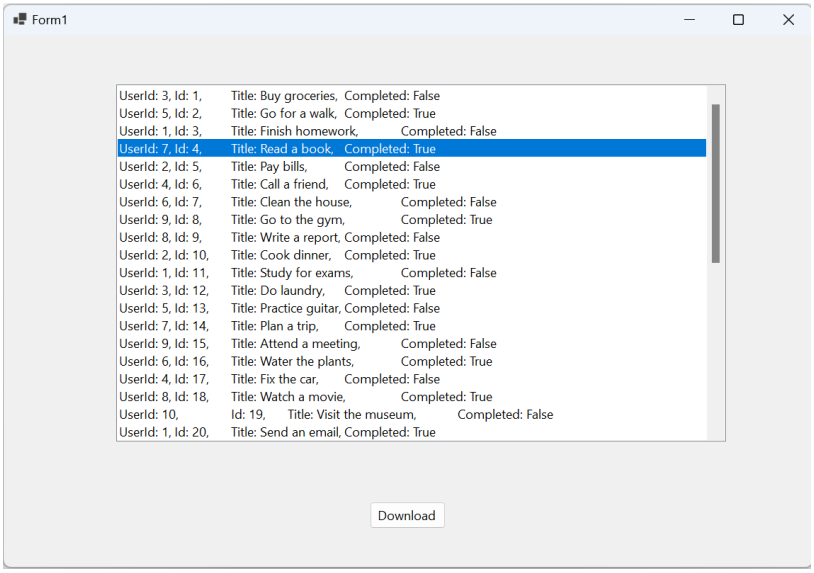
```
    internal class ToDo
    {
        public int userId { get; set; }
        public int id { get; set; }
        public string title { get; set; }
        public bool completed { get; set; }

        public override string ToString()
        {
            return $"UserId: {userId},\tId: {id},\tTitle: {title,-15},\tCompleted: {
    completed}";
        }
    }
```

Listing 4: Klasa opisująca obiekt `ToDo` pobierany z `API`

W przypadku każdej klasy warto przeciążyć metodę `ToString()` w celu łatwiejszego wyświetlania danych na dalszych etapach. Po przygotowaniu klasy do samej deseralizacji należy skorzystać z funkcji `Deserialize<typ>(string)`, która jako argument przyjmuje `string` do zmapowania oraz należy określić typ danych. W omawianym przypadku jest to lista zadań, ale nie zawsze odpowiedź z `API` będzie zwracana w takiej formie. Przykład pokazano na Listingu 5 a rezulat jego uruchomienia na Rysunku 6. Po liście w łatwy sposób można iterować i wrzucić obiekty przykładowo do `listBox` lub osobno wyświetlić pole z klasy `ToDo` (np. same nazwy zadań).

```
        private async void bDownload_Click(object sender, EventArgs e)
        {
            string call = "https://dummy-json.mock.beeceptor.com/todos";
            string response = await client.GetStringAsync(call);
            List<ToDo> toDos = JsonSerializer.Deserialize<List<ToDo>>(response);
            foreach (ToDo t in toDos)
            {
                listBox1.Items.Add(t.ToString());
            }
        }
```

Listing 5: Metoda wyświetlająca obiekty `ToDo` w aplikacji okienkowej w komponencie `ListBox`



Rysunek 6: Wynik działania metody z Listingu 5

3.2 Zadanie 2

Celem zadania jest rozbudowanie programu z zadania pierwszego o prostą bazę danych. Program musi pozwalać na podstawową obsługę bazy (dodawanie, filtrowanie czy sortowanie danych). Ponadto, aby uzyskać maksymalną ocenę za ten etap, należy pobierać dane z `API` tylko w przypadku ich braku w bazie danych (ma to na celu uniknięcie ponownego pobierania kilkukrotnie tych samych danych) oraz zaimplementować przynajmniej jedną relację pomiędzy tabelami w bazie. W zadaniu zostanie wykorzystanie mapowanie obiektowo-relacyjne (*Object-Relational Mapping*, `ORM`) z użyciem `Entity Framework`. Struktura klasy, która posłuży do przechowywania danych w bazie może się różnić od klasy użytej do deserializacji odpowiedzi z `API`.

Przygotowanie bazy danych

W pierwszej kolejności należy dodać odpowiednie pakiety do naszego projektu z wykorzystaniem menadżera paczek `NuGet`, który znajduje się w karcie `Tools` (`Narzędzia`). Podstawowym pakietem do zainstalowania jest `Microsoft.EntityFrameworkCore` w najnowszej wersji `8.0.3` (zgodność wersji jest bardzo istotna). Oprócz wybranego pakietu, powinny zainstalować się również inne wymagane pakiety. W ramach zadania wystarczy użycie bazy danych przechowywanej w pliku, więc niezbędny będzie jeszcze pakiet `Microsoft.EntityFrameworkCore.Sqlite`. Nie jest to jedyna możliwość, ponieważ `Microsoft` dostarcza również własne rozwiązanie czy wsparcie dla `PostgreSQL`. Na koniec należy jeszcze dodać narzędzia do obsługi bazy danych `Microsoft.EntityFrameworkCore.Tools`.

Dla zróżnicowania prezentowanych przykładów zaprezentowany zostanie sposób tworzenia i obsługi bazy danych, wykonany na klasie `Student`. W pierwszej kolejności należy przygotować klasę do przechowywania informacji o studentach. Można użyć tej samej klasy co do parsowania jsona z API, ale w większości przypadków klasa będzie zawierać trochę inne pola (cześć poprzednich może być niepotrzebnych, ale też mogą dojść nowe). Dodatkowo warto oznaczyć pola wymagane `required` oraz te, które mogą przyjąć wartość `null` (należy wykorzystać operator `?`). Ponadto `Entity Framework` wymaga, aby pojawiło się pole `Id` (lub `NameClassId`), które będzie kluczem głównym. W przypadku pól złożonych (reprezentowanych) przez inne klasy możemy stworzyć osobne klasy, które również są

identyfikowane po kluczu (wymaga zdefiniowania osobnej tabeli) lub od `.NET 8.0` bez klucza (nie wymaga zdefiniowania osobnej tabeli).

```
internal class Student
{
    public int Id { get; set; }
    public required string Name { get; set; }
    public required float Average { get; set; }
    public string? Specialty { get; set; }
    public override string ToString()
    {
        return $"Id: {Id},\tName: {Name,-15}\t,Average: {Average:0.00}, Specialty: {Specialty,-3}";
    }
}
```

Kolejnym krokiem jest zdefiniowanie klasy odpowiedzialnej za kontekst bazy danych. Ma ona za zadanie zarządzanie wszystkimi tabelami. Klasa nie może być publiczna. W bieżącym przykładzie wystarczy jedna tabela przechowująca dane o studentach. Następnie tworzymy konstruktor domyślny, który ma za zadanie utworzenie obiektu bazy danych. Z powodu użycia `SQLite`, trzeba jeszcze przeciążyć metodę konfiguracyjną i wskazać plik do przechowywania danych. Częstym błędem jest tworzenie bazy w innym folderze niż znajduje się uruchamiany program (Debug zamiast Release), dlatego można rozważyć podanie bezwzględnej ścieżki do pliku. Na koniec możemy jeszcze wstępnie uzupełnić bazę danych kilkoma rekordami początkowymi przy jej tworzeniu przeciążając metodę `OnModelCreating` (rozwiązanie opcjonalne). Przy dodawaniu nowych rekordów do bazy danych, należy pamiętać że klucz główny (w tym przypadku Id) nie może się powtarzać

```
internal class University : DbContext
{
    public DbSet<Student> Students { get; set; }
    public University()
    {
        Database.EnsureCreated();
    }
    protected override void OnConfiguring(DbContextOptionsBuilder options)
    {
        options.UseSqlite(@"Data Source=Univ.db");
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>().HasData(
            new Student() { Id = 1, Name = "Agnieszka", Average = 5.5f },
            new Student() { Id = 2, Name = "Bartosz", Average = 4.5f },
            new Student() { Id = 3, Name = "Czarek", Average = 5.0f }
            );
    }
}
```

Przed przystąpieniem do wykonywania operacji na bazie danych, trzeba ją w pierwszej kolejności utworzyć. W tym celu należy przejść do konsoli zarządzania pakietami (`Package Manager Console`, domyślnie powinna być uruchomiana, widoczna obok Danych wyjściowych – `Output` oraz Listy błędów `Error List`. Jeśli nie, to jest dostępna w Narzędziach – `Tools`). W pierwszej kolejności należy utworzyć pierwszy punkt migracji bazy danych np.: `Add-Migration Init`, gdzie w tym wypadku `Init` to nazwa migracji. Domyślnie możliwość tworzenia migracji powinna być włączona (jeśli nie, to włączymy je komendą `Enable-Migrations`). Na koniec wymuszamy aktualizacje bazy danych komendą `Update-Database`. W przypadku zmiany struktury bazy danych, zawsze trzeba w pierwszej kolejności utworzyć nową migrację oraz następnie zaktualizować bazę danych.

Operacje na bazie danych

Ostatnim krokiem przed rozpoczęciem wykonywania operacji na bazie danych jest utworzenie obiektu klasy kontekstowej bazy danych wewnątrz klasy głównej naszego programu. Rekordy do bazy danych można dodawać przy użyciu metody `Add` na odpowiedniej tabeli. Proszę zwrócić uwagę, że przy dodawaniu nie nadajemy numeru `Id`, będzie on uzupełniony przez `Entity Framework` automatycznie z użyciem autoinkrementacji. UWAGA! Po operacjach, które mają wpływ na dane w bazie danych (dodanie lub usunięcie rekordu) należy zapisać aktualny stan bazy danych za pomocą metody `(SaveChanges())`. Możemy rekordy z tabeli zwrócić w formie listy używając metody `ToList`, a następnie podpiąć jako źródło do kontrolki typu `listBox`, aby wyświetlić pełną listę studentów.

```
public partial class FormStudent : Form
{
    private University university;
    public FormStudent()
    {
        InitializeComponent();
        university = new University();
    }
    private void buttonAdd_Click(object sender, EventArgs e)
    {
        university.Students.Add(new Student() { Name = textBoxName.Text, Average =
    float.Parse(textBoxAverage.Text) });
        university.SaveChanges();
        listBoxStudents.DataSource = university.Students.ToList<Student>();
    }
}
```

Usunięcie wszystkich rekordów z tabeli można uzyskać z wykorzystaniem poniższej komendy.

```
uni.Students.RemoveRange(uni.Students);
uni.SaveChanges();
```

Operacje na tabelach czy kolekcjach najwygodniej wykonywać z wykorzystaniem kwerend `LINQ`. W łatwy sposób można odfiltrować studentów poniżej średniej lub posortować ich po niej. W celu usunięcia wpisu, musimy najpierw go wyszukać, a dopiero następnie usunąć. Po usunięciu należy pamiętać o zapisaniu stanu bazy danych.

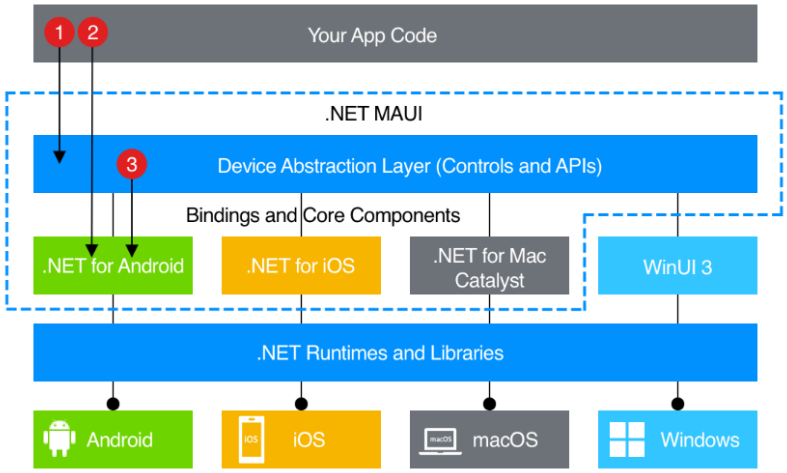
```
private void buttonGetTop_Click(object sender, EventArgs e)
{
    listBoxStudents.DataSource = university.Students.Where(s => s.Average > 4.5).
    ToList<Student>();
}

private void buttonSort_Click(object sender, EventArgs e)
{
    listBoxStudents.DataSource = university.Students.OrderBy(s => s.Average).Reverse
    ().ToList<Student>();
}
private void buttonRemove_Click(object sender, EventArgs e)
{
    var student = university.Students.First(s => s.Id == int.Parse(textBoxId.Text));
    university.Students.Remove(student);
    university.SaveChanges();
    listBoxStudents.DataSource = university.Students.ToList<Student>();
}
```

3.3 Zadanie 3

W tym zadaniu należy zaimplementować interfejs graficzny do stworzonej w poprzednich punktach bazy danych zawierającej rekordy pobrane z API. Ponieważ znajomość technologii `Windows Forms`

została sprawdzona na poprzednich zajęciach, w tym zadaniu aplikacja wytworzona będzie z użyciem interfejsu `MAUI` (ang. Multi-platform App UI). Ma on tą przewagę nad `Windows Forms`, że aplikacje tworzone z jego pomocą, można uruchamiać w systemach Android, iOS, macOS i Windows z poziomu pojedynczej udostępnionej bazy kodu. Jest to ewolucja platformy `Xamarin`, do której wsparcie zostało już wygaszone. Dzięki temu, że pisany kod wchodzi w interakcje wyłącznie z interfejsem `API` (przedstawione na Rysunku 10) do kontrolek zaczerpniętych z poszczególnych systemów, uzyskiwana jest wieloplatformowość technologii. Pojedynczy projekt zawiera następujące elementy:

- `Platforms` — znajdują się tu pliki wykorzystywane przez konkretne platformy, jak na przykład `manifest.xml` dla systemu Android,
- Resources – wszelkie zdjęcia, czcionki i inne zasoby, które będą dzielone dla wszystkich platform,
- `App.xml` — punkt wejściowy naszej aplikacji,
- `AppShell.xml` — (opcjonalnie) infrastruktura nawigacji naszej aplikacji,
- `MainPage.xml` — domyślna strona „Hello World”,
- `MauiProgram.cs` -– konfiguracja aplikacji, odpowiednik klasy Startup.cs



Rysunek 7: Diagram architektury `.NET MAUI`

Celem zadania jest stworzenie interfejsu w `MAUI`, pozwalającego na:

- wyświetlenie rekordów znajdujących się w bazie danych,
- wprowadzenie parametrów `API`, pozwalających na pobranie interesujących danych,
- ręczne dodanie rekordu do bazy danych

Wykonanie walidacji pól, użycie pól/rozwiązań innych niż zaproponowane oraz zastosowanie dobrych praktyk `UX` będą dodatkowym atutem. Aby ułatwić rozpoczęcie prac nad projektem poniżej opisana jest przykładowa konfiguracja aplikacji `MAUI` pozwalająca na wyświetlanie danych przechowywanych w kolekcji `ObservableCollection`. Podane komponenty są jedynie przykładem i dozwolone jest użycie innych oraz zastosowanie innej, aczkolwiek przemyślanej koncepcji aplikacji.

Na początek należy w wygenerowanym domyślnie pliku `MainPage.xaml` dodać pola typu `Entry`, `Button` oraz `ListView`. Ważne, aby na początku pliku dodać odwołanie do namespace'u, w którym znajdować się będą używane klasy. Można to zrobić poleceniem:

```
xmlns:local="clr-namespace:MauiApp1"
```

Parametry wszystkich typów pól dostępnych w interfejsie, są opisane na stronach `Microsoft`, w związku z czym nie będą one szczegółowo przytaczane. Przykładowa definicja komponentów pokazana jest na Listingu 8.

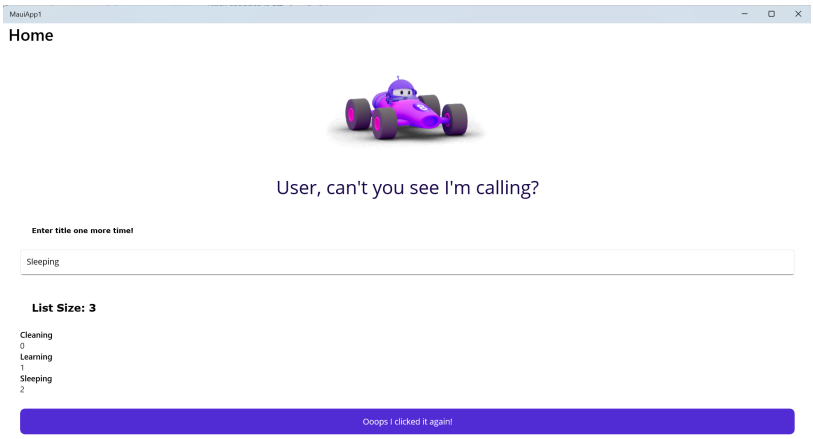
```
            <Label x:Name="myLabel"
                Text="Enter title one more time!"
                FontSize="12"
                FontFamily="Verdana"
                FontAttributes="Bold"
                TextColor="Black"
                Grid.Column="1"
                Grid.Row="2"
                Padding="20,20,0,0"/>

            <Entry x:Name="entry"
                Placeholder="I'm not that innocent..."
                TextChanged="OnEntryTextChanged"/>

            <ListView x:Name="Lista"
               Grid.Row="1"
              SeparatorColor="Black"
              SeparatorVisibility="Default"
              ItemsSource="{Binding toDos}">
                <ListView.ItemTemplate>
                    <DataTemplate x:DataType="local:ToDo1">
                        <TextCell Text="{Binding title}"
                            TextColor="Black"
                            Detail="{Binding id}" />
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>

            <Button
                x:Name="CounterBtn"
                Text="Ooops I clicked it again!"
                SemanticProperties.Hint="Counts the number of times you click"
                Clicked="OnCounterClicked"
                HorizontalOptions="Fill" />
```

Rysunek 8: Przykład definicji komponentów w pliku xaml

Po dodaniu komponentów, należy utworzyć klasę odpowiedzialną za wyświetlane obiekty. Ze względu na fakt, iż była ona już przytaczana w instrukcji, kod który ją zawiera, zostanie pominięty. Klasa może zostać utworzona w pliku `MainPage.xaml.cs`, choć zalecane jest użycie osobnego pliku klasy. We wspomnianym pliku natomiast należy zaimplementować metody odpowiedzialne za logikę działania interfejsu, czyli na przykład funkcje wywołujące się po kliknięciu przycisku czy też konstruktor aplikacji. Metody użyte w prezentowanym przykładzie pokazano na Listingu 9.

```
public partial class MainPage : ContentPage
{
    int count = 0;
    ObservableCollection<ToDo1> toDos;

    public MainPage()
    {
        InitializeComponent();
        toDos = new ObservableCollection<ToDo1>();
        Lista.ItemsSource = toDos;
        BindingContext = this;
    }

    private void OnCounterClicked(object sender, EventArgs e)
    {
        ToDo1 t = new ToDo1();
        t.id = toDos.Count();
        t.title = entry.Text;
        int size = toDos.Count() + 1;
        listSize.Text = "List Size: " + size;
        toDos.Add(t);

    }
}
```

Rysunek 9: Przykład definicji komponentów w pliku `xaml`



Rysunek 10: Uzyskany rezultat aplikacji w `MAUI`