Uniwersytet Mikołaja Kopernika
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Instytut Fizyki UMK w Toruniu

Dawid Czaja
nr albumu: 267509

Praca magisterska
na kierunku Informatyka Stosowana

Programowanie aplikacji mobilnych z użyciem Xamarin i MvvmCross

Opiekun pracy dyplomowej
dr hab. Jacek Matulewski
Wydział Fizyki, Astronomii i Informatyki Stosowanej UMK

Toruń 2018

Pracę przyjmuję i akceptuję	Potwierdzam złożenie pracy dyplomowej
.....	.....
data i podpis opiekuna pracy	*data i podpis pracownika dziekanatu*

Za inspirację, opiekę merytoryczną i pomoc w nadaniu właściwego kształtu mojej pracy dziękuję promotorowi dr.hab. Jackowi Matulewskiemu.

Spis treści

1. Wstęp

1.1. Wprowadzenie

Od kilku lat można zaobserwować nieprzerwany dynamiczny wzrost liczby użytkowników urządzeń mobilnych. Najpopularniejsze z nich to smartfony i tablety. Urządzenia mobilne umożliwiają nie tylko połączenia telefoniczne, ale również rozrywkę, natychmiastowy kontakt ze znajomymi, odbieranie poczty oraz wiele innych przydatnych rzeczy. Wraz z rosnącą liczbą użytkowników, wzrasta również zapotrzebowanie na nowe aplikacje, których przygotowanie na poszczególne platformy mobilne pochłania sporo czasu. Z tego powodu warto zainteresować się oferowaną przez Microsoft technologią Xamarin, dzięki której można zredukować czas potrzebny do stworzenia aplikacji na różne platformy.

1.2. Cel pracy

Moim zadaniem było przygotowanie tzw. samouczka (tutorialu), w którym na przykładzie trzech aplikacji miałem pokazać, w jaki sposób można wykorzystać technologię Xamarin oraz framework MvvmCross do przyspieszenia pracy podczas tworzenia aplikacji na platformy Android, iOS oraz Universal Windows Platform (UWP). W szczególności skupiłem się na omówieniu następujących zagadnień:

- tworzenie części wspólnej w projektach multiplatformowych,
- wykorzystanie gotowych zasobów dostarczających podobne funkcjonalności na poszczególnych platformach mobilnych,
- tworzenie i wstrzykiwanie obiektów implementujących funkcjonalności na poszczególnych platformach,
- tworzenie, zarządzanie i dostęp do podstawowych operacji na bazie danych typu SQLite na poszczególnych platformach.

Wszystkie te zagadnienia, zostały w tej pracy dokładnie opisane i zilustrowane działającymi projektami. W niniejszej pracy przedstawiłem również wzorzec architektoniczny MVVM, na którym bazuje framework MvvmCross. Opisałem również technologię Xamarin, za pomocą której zostały zbudowane aplikacje.

2. Wykorzystywane technologie

2.1. Do czego służy Xamarin ?

Xamarin to darmowy „ekosystem” technologii pozwalający na tworzenie wieloplatformowych aplikacji mobilnych, w których językiem programowania jest C#. Aplikacje tworzone za pomocą Xamarin są zbudowane ze standardowych natywnych elementów, jakie udostępniają nam poszczególne platformy programistyczne. W szczególności wykorzystywane są do budowy graficznego interfejsu użytkownika. Oznacza to, że nasze aplikacje mobilne wyglądają jak standardowe aplikacje tworzone w natywnych technologiach, a dodatkowo również tak się zachowują.[1]

Warto zaznaczyć, że aplikacje Xamarin mają dostęp do pełnego zakresu funkcji udostępnianych przez platformę systemową i samo urządzenie, włączając w to funkcje specyficzne dla poszczególnych platform np. iBeacons, który jest wykorzystywany w iOS do komunikacji z różnymi sensorami oraz menedżer sensorów, który do tego samego służy w systemie Android.[2]

Korzystając z Xamarin, jesteśmy w stanie napisać wszystko to, co możemy zrobić w języku Java w przypadku projektu dla systemu Android lub w języku Objective-C wykorzystywanego do tworzenia aplikacji dla platformy iOS.[1]

Jest tak, ponieważ do kompilacji projektów dla poszczególnych platform używane są ich natywne kompilatory. W przypadku projektu przeznaczonego na urządzenia z systemem Android, kompilator kompiluje projekt do języka IL (od ang. *Intermediate Language*), a następnie przy uruchamianiu aplikacji wykorzystywana jest technologia JIT (od ang. *Just-in-time*), czyli ponowna kompilacja do języka maszynowego. [3] W przypadku projektów na platformę iOS, kompilator kompiluje aplikacje w trybie AOT (od ang. *Ahead of time*). W wyniku powstaje natywny kod dla procesorów ARM, który jest uruchamiany bezpośrednio na iPhone’ach.[3]

Dzięki tym zabiegom, Xamarin jest transparentny dla docelowych systemów mobilnych, tworząc natywne, w pełni funkcjonalne aplikacje.

2.2. Zalety języka C# w tworzeniu aplikacji mobilnych

Pierwszą zaletą technologii Xamarin jest jeden język, w jakim możemy tworzyć wszystkie wersje aplikacji, czyli C#. To nowoczesny, stale rozwijany język, którego składnia ułatwia pisanie aplikacji mobilnych. Poniżej omówię dwa przykłady, które przemawiają za tą tezą.

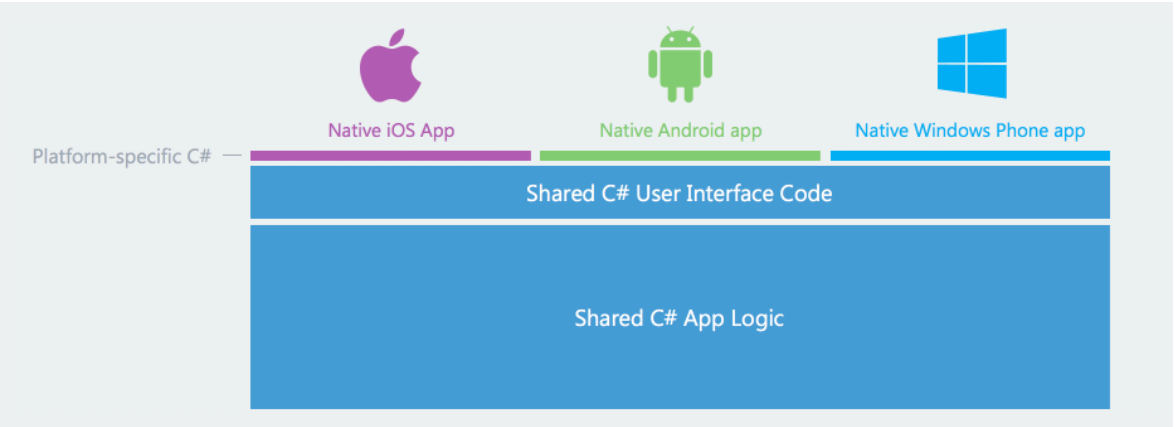
C# ułatwia programowanie asynchroniczne, dzięki czemu łatwiejsze jest zapewnienie stałej responsywności aplikacji. Czynności wymagające więcej czasu można wykonać w osobnych wątkach (asynchronicznie), synchronizując interfejs dopiero po ich zakończeniu. Dzięki temu użytkownik nie jest zmuszony do czekania na odblokowanie interfejsu w czasie, gdy np. aplikacja zapisuje dane do bazy danych[1]. W porównaniu z językami Objective-C, Swift i Javą, w C# asynchroniczne wywołania metod są prostsze i nie wymagają obsługi funkcji zwrotnych (ang. *callbacks*). [4]

Kolejnym elementem składni, który świadczy o przewadze C# nad np. Javą, są wyrażenia lambda. Co prawda w języku Objective-C istnieją wyrażenia lambda, ale są bardziej skomplikowane w użyciu. Natomiast w języku C# ich składnia jest bardziej czytelna, co ułatwia efektywne programowanie funkcjonalne. [4]

Kluczowe jest również to, że rozwiązania na wszystkie platformy możemy tworzyć w jednym języku programowania – C#. Dla osób znających ten język, ważne będzie również to, że do przygotowania części wspólnej, która w większych aplikacjach jest zdecydowanie największą częścią projektu, korzystamy z dobrze im znanych klas i narzędzi ze środowiska .NET, W ten sposób dostęp do bazy danych wspólne modele obiektów biznesowych czy samą logikę przetwarzania takich modeli przygotowują w dobrze znanym sobie środowisku. Takie podejście zapewnia nam współdzielenie zasobów średnio na poziomie 75%. [4]

[1] Zob. projekt Notatnik w rozdziale 4, w którym wykorzystano to podejście.

2.3 Typy projektów w Xamarin



Rys.1 Schemat projektowy stworzonych natywnych aplikacji
(źródło: *https://www.visualstudio.com/pl/xamarin/*)

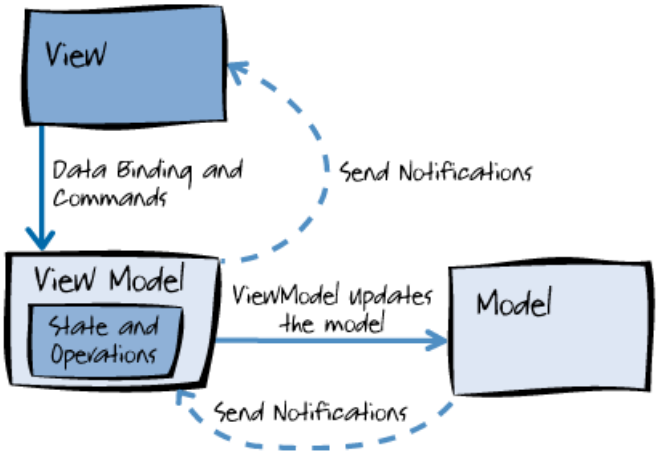
Ekosystem technologii Xamarin oprócz tworzenia natywnych aplikacji, umożliwia również przygotowanie projektu *Xamarin.Forms*, w którym warstwa widoku jest wspólna dla wszystkich platform. Dzięki temu możemy niemal w 100% współdzielić cały projekt na wszystkich trzech platformach. Inaczej niż w przypadku osobnych projektów widoków *Xamarin.Android*, *Xamarin.iOS*, dzięki którym jesteśmy w stanie dostarczyć użytkownikom natywne graficzne interfejsy użytkownika i charakterystyczne dla poszczególnych platform zachowanie aplikacji. W przypadku *Xamarin.Forms*, dostępne są kontrolki, które są jednakowe na wszystkich trzech platformach i nie możemy ich w żadnej sposób zmienić lub stworzyć widoków charakterystycznych dla poszczególnych platform. [5]

Dodatkowo tworząc ososbne projekty *Xamarin.Android* oraz *Xamarin.iOS*, niejako zdobywamy wiedzę na temat implementacji różnych funkcji na platformach docelowych. Pomimo użycia w tych projektach języka C#, klasy należą do poszczególnych platform i w przypadku ewentualnych problemów, można szukać rozwiązania na forach przeznaczonych dla technologii natywnych.

Inaczej jest w przypadku Xamarin.Forms. API tej technologii znacznie różni się od natywnych, więc może być trudniej znaleźć rozwiązanie problemu występującego w tworzonej aplikacji.

2.4. Wzorzec MVVM

Wzorzec architektoniczny MVVM (od ang. *Model–View–View Model*) umożliwia oddzielenie warstwy prezentacji od warstwy logiki biznesowej aplikacji. Dzięki temu można w łatwy sposób stworzyć aplikację która jest w większym zakresie testowalna, prosta w rozbudowie i bardzo ułatwia ponowne wykorzystanie kodu w kolejnych projektach. [6]



Rys.2 Wzorzec projektowy Mvvm i jego podstawowe składniki
(źródło: *https://android.jlelse.eu/exciting-secrets-about-mvvm-that-nobody-tells-you-a95548ea684b*)

Poszczególne moduły tego wzorca, a w zasadzie warstwy, to:

- **Model** jest obejmuje model danych wraz z korzystającą z nich logiką biznesową i walidacyjną. Przykłady obiektów modelu obejmują repozytoria, za pomocą których pobieramy dane z baz danych, obiekty biznesowe, obiekty przesyłania danych (DTO) oraz automatycznie wygenerowane obiekty proxy.[6]
- **View Model** (model widoku) jest pośrednikiem pomiędzy widokiem a modelem. Zazwyczaj model widoku współdziała z modelem (tworzy instancję modelu i przechowuje jego referencję), wywołując metody w klasach modelu. Model widoku jest również odpowiedzialny za dostarczanie danych z modelu w formie, którą widok może łatwiej wykorzystać. Zapewnia również implementacje poleceń, które użytkownik aplikacji inicjuje w widoku np. kliknięciem przycisku. Tak zainicjowane polecenie może wówczas wywołać metodę modelu.[6]
- **View** (widok) jest odpowiedzialny za określenie struktury, układu i wyglądu tego, co użytkownik widzi na ekranie. Idealnie jest, jeżeli widok jest zdefiniowany wyłącznie za pomocą języka znaczników specyficznego dla danej platformy np.

XAML w UWP, AXML w Androidzie lub XIB w iOS. Nie ma bowiem wówczas tzw. *code-behind*, który nie może być objęty testami jednostkowymi. Korzystanie z code-behind jest sprzeczne z ideą MVVM, która zakłada że warstwa widoku pozbawiona jest jakiejkolwiek logiki biznesowej.[6]

Niewątpliwą zaletą tego wzorca jest podział na trzy warstwy z charakterystycznym dla nich uporządkowaniem zależności między poszczególnymi warstwami. Dzięki temu uniezależniamy logikę biznesową od danych wyświetlanych w warstwie widoku, a jednocześnie od sposobu przechowywania danych. Dzięki zastosowaniu wzorca MVVM możemy łatwiej zrównoleglić pracę zespołu programistycznego: jedna osoba może zająć się układaniem kontrolek w graficznym interfejsie użytkownika, podczas gdy inna będzie zajmować się przygotowywaniem kodu logiki biznesowej. Podział na trzy odrębne warstwy wspomaga również proces tworzenia testów jednostkowych rozdzielając poszczególne testowane obszary[2].[7]

Oczywiście MVVM, jak każdy wzorzec projektowy, nie jest pozbawiony wad. Jedną z nich jest mnogość klas, które należy stworzyć, nawet w przypadku pojedynczego widoku. Dodatkowo stosowanie poleceń (także wzorzec projektowy) zamiast mechanizmu zdarzeń do obsługi akcji inicjowanych przez użytkowników również zwiększa ilość kodu, co dodakowo komplikuje projekt.[7]

2.5. MvvmCross

MvvmCross to framework opracowany specjalnie dla Xamarin i platform mobilnych. Pozwala tworzyć aplikacje dla poszczególnych platform mobilnych współdzieląc warstwy modelu i modelu widoku (tzw. część współdzielona projektu). Jedynie widoku, osobne dla każdej platformy, umieszczane są w osobnych projektach *Xamarin.iOS, Xamarin.Android, Xamarin.Mac,* UWP i WPF. [8]
Najważniejsze cechy, jakie oferuje ten framework:

- wsparcie dla wzorca architektonicznego MVVM, [9]
- system nawigacyjny między poszczególnymi widokami w danej aplikacji, [9]
- wiązanie danych zgodnie z regułami MVVM, [9]
- wsparcie dla techniki odwrócenia sterowania (ang. *inversion of control*) i mechanizm wstrzykiwania zależności (ang. *dependency injection*) [9],

- wiele ujednoliconych wtyczek dla zwykłych funkcji, oferujących gotowe implementacje funkcji specyficznych dla danej platformy. Jedną z wykorzystanych w tej pracy wtyczek jest Messenger. Ta wtyczka umożliwia przekazywanie wiadomości pomiędzy modelami widoku [9],
- mechanizm do testów jednostkowych. [9]

Sam framework jest wykorzystywany przez ponad 5000 firm na całym świecie. Dodatkowo jest ciągle rozwijany przez ponad dwustu ludzi na publicznym repozytorium znajdującym się na serwerze GitHub. [8]

2.6. SQLite

SQLite to biblioteka zapewniająca prosty i lekki system zarządzania relacyjnymi bazami danych. Słowo Lite (z ang. lekki) w nazwie oznacza niewielką „wagę", jeśli chodzi o wymagane zasoby, ale również nakład pracy potrzebny na konfigurację i administrację bazą danych. SQLite to samodzielny, bezserwerowy, nie wymagający konfiguracji, transakcyjny system bazodanowy, który do działania nie wymaga architektury typu klient-serwer. [10]

SQLite jest samowystarczalny, co oznacza, że wymaga minimalnego wsparcia ze strony systemu operacyjnego lub dodatkowych bibliotek. Dzięki temu SQLite może być używany w każdym środowisku, w szczególności na urządzeniach mobilnych, takich jak iPhone czy telefony z systemem Android.[11]

SQLite jest przygotowany przy użyciu języka ANSI-C. Kod źródłowy jest dostępny jako jeden duży plik o nazwie *sqlite3.c* i z plikiem nagłówkowym *sqlite3.h*. Stąd jego obecność na niemal wszystkich platformach. Aby móc korzystać z SQLite należy dodać te pliki do swojego projektu i je skompilować. [11] W przypadku platformy .NET można jednak znaleźć gotowe biblioteki, także wśród pakietów NuGet. Przykładem jest *SQLite.Net-PCL*, który został wykorzystany w tek pracy podczas tworzenia aplikacji *Notatnik*. Dostarcza ona gotowe do użycia funkcje, bez potrzeby ręcznej kompilacji plików źródłowych biblioteki SQLite.

[2] W Xamarin oprócz logiki aplikacji (backendu) można również testować widoki za pomocą biblioteki Xamarin UI Tests

3. Wprowadzenie do tworzenia aplikacji

W tym rozdziale został opisany proces tworzenia projektów z wykorzystaniem technologii Xamarin i frameworka MvvmCross na platformy Android, Universal Windows Platform (UWP) oraz iOS. W przypadku każdej z nich opisuję proces wstępnego przygotowywania i konfigurowania projektów, do czego będę się wielokrotnie odwoływał w kolejnym rozdziale.

3.1 Wymagane narzędzia do rozpoczęcia pracy

Chcąc tworzyć aplikacje z wykorzystaniem Xamarin dla urządzeń mobilnych z system Android, Windows 10 oraz iOS, musimy zaopatrzyć się w następujące narzędzia:

- komputer z systemem Windows 10 – bez tego systemu nie będziemy w stanie testować aplikacji dla platformy Universal Windows Platform,
- Visual Studio 2017 – to zintegrowane środowisko developerskie (IDE od ang. *Integrated Development Environment*) służące do tworzenia aplikacji m.in. z użyciem technologii Xamarin (projekty typu Universal Windows Platform, Xamarin.Android oraz Xamarin.iOS),
- komputer z systemem macOS High Sierra w wersji 10.13 (stan z dnia 29.03.2018) – bez niego, nie będziemy w stanie testować aplikacji tworzonych na platformę iOS, które mogą być tworzone w środowisku Windows, lecz do uruchomienia wymagają połączenia z maszyną pracującą pod systemem macOS
- Xcode – środowisko IDE, umożliwiające budowanie projektu i debugowanie wszystkich typów projektów programistycznych dla systemu OS X. Zawarty w nim *interface builder* to graficzny edytor służący do projektowania graficznego interfejsu użytkownika. Bez tego środowiska nie bylibyśmy w stanie tworzyć widoków dla projektów Xamarin.iOS oraz debugować aplikacji napisanych na tę platformę.
- Visual Studio Mac – środowisko IDE, obsługujące technologię Xamarin na komputerach z systemem macOS.

3.2 Omówienie struktury projektu

Projekt oparty na technologiach Xamarin oraz MvvmCross składa się z kilku projektów. Przede wszystkim zawiera projekt tzw. części wspólnej, która przeznaczona jest dla wszystkich platform i zawiera wspólne warstwy modelów i modelów widoku. Towarzyszą mu projekty przeznaczone na poszczególne platformy mobilne, dla których chcemy przygotować wersje projektowanej aplikacji. Wszystkie te projekty umieszczane są w jednym rozwiązaniu (solucji) stworzonej za pomocą Visual Studio 2017.

Na początku chciałbym omówić projekt części wspólnej. Możemy go przygotować na dwa sposoby: jako projekt współdzielony (ang. *shared project*) oraz jako przenośną bibliotekę klas (ang. *portable class library*, PCL).

Projekt współdzielony został wprowadzony wraz z platformą Windows 8.1 i jest dostępny od Visual Studio 2013 Update 2. Umożliwia współdzielenie zarówno źródeł, jak i zasobów różnego typu na różnych platformach. Referencję do tej biblioteki możemy umieszczać w innych projektach, tak samo jak w przypadku standardowych bibliotek. Co ważne, jest on kompilowany w trakcie kompilacji każdego projektu, który posiada do niego referencję, natomiast nie może zostać skompilowany oddzielnie, ponieważ formalnie nie jest projektem. [12]

W ramach tego projekt możemy wykonywać kompilacje warunkowe tzn. możemy w kodzie umieścić instrukcję warunkowe, które będą skompilowane w zależności od rodzaju projektu, który odwołuje się do projektu współdzielonego. [14]

Chcąc przygotować osobne implementacje jakiejś funkcjonalności dla każdej z platform oddzielnie (np. dostarczyć implementacji zapewniającej połączenie i zapis danych w bazie SQLite), możemy wykorzystać mechanizm klas lustrzanych (ang. *class mirroring*) lub klas oraz metod częściowych (ang. *partial*). [14]

Pierwszy mechanizm polega na tworzeniu dla poszczególnych platform osobnych klas o takiej samej nazwie i w takiej samej przestrzeni nazw i odwoływanie się do nich w części wspólnej. Podobnie działa drugi sposób: możemy utworzyć klasę w części wspólnej z metodą wirtualną (modyfikator `virtual` w sygnaturze metody) a następnie w klasach potomnych w projektach dla poszczególnych platform nadpisywać tą metodę z logiką specyficzną dla danej platformy. [14]

Niewątpliwą zaletą projektów współdzielonych jest możliwość współdzielenia różnych plików nie ograniczając się tylko do klas. Jednak w dłuższej perspektywie, korzystanie z mechanizmów dostarczanych w ramach *shared project* prowadzi do

bałaganu w kodzie, przez co ciężko taki projekt dalej rozwijać. Kolejną wadą jest fakt, że tego typu projekty nie mogą być testowane za pomocą testów jednostkowych, ponieważ nie mogą być oddzielnie skompilowane. [14]

W praktyce podejście typu *shared project* jest obecnie bardzo rzadko stosowane, dlatego w tej pracy skupię się na alternatywnym mechanizmie tj. *portable class library* (PCL). PCL to typ projektu będącego biblioteką klas, który może zostać wykorzystany w różnego typu projektach .NET bez ponownej kompilacji. [13]

Po utworzeniu projektu biblioteki PCL możemy wskazać wersję platformy .NET, z jakim może współpracować biblioteka oraz inne platformy, a tym samym rodzaje projektów, w których może zostać wykorzystana. Wraz ze wzrostem liczby platform wspieranych przez bibliotekę, ograniczamy zakres funkcjonalności, jaka może być w niej wykorzystana. Dla przykładu nie będzie wsparcia dla funkcjonalności „Data Annotations" w przypadku wybrania wsparcia platformy typu Windows Phone SilverLight – dana funkcjonalność musi być bowiem wspierana przez wszystkie wybrane platformy. Tabela z rysunku 3 prezentuje zależności między wspieranymi projektami a dostarczanymi funkcjonalnościami. Szczegółowe informacje na ten temat można znaleźć w oficjalnej dokumentacji pod adresem : *https://docs.microsoft.com/pl-pl/dotnet/standard/cross-platform/cross-platform-development-with-the-portable-class-library*.

Feature	.NET Framework	.NET Framework	.NET Framework	Windows Store	Windows Store	Windows Phone Store	Windows Phone Silverlight	Windows Phone Silverlight
	4	**4.0.3**	**4.5**	**8**	**8.1**	**8.1**	**7.5**	**8**
Core libraries	✓	✓	✓	✓	✓	✓	✓	✓
Async support	❶	❶	✓	✓	✓	✓	❶	❶
Compression			✓	✓	✓	✓		❷
Data annotations		✓	✓	✓	✓			
Dynamic keyword	✓	✓	✓	✓	✓			
HTTPClient	❷	❷	✓	✓	✓	✓	❷	❷
IQueryable	✓	✓	✓	✓	✓	✓	✓	✓

Rys.3 Częściowa tabela prezentująca zależności w dostarczanych funkcjach dla wybranych platform (źródło: *https://docs.microsoft.com/pl-pl/dotnet/standard/cross-platform/cross-platform-development-with-the-portable-class-libraryjpg*)

W celu przygotowania kodu tylko dla jednej z platform można albo stworzyć własne zdarzenia, albo stworzyć własne implementacje funkcjonalności na daną platformę, dostarczając ją jednocześnie do projektu części wspólnej.

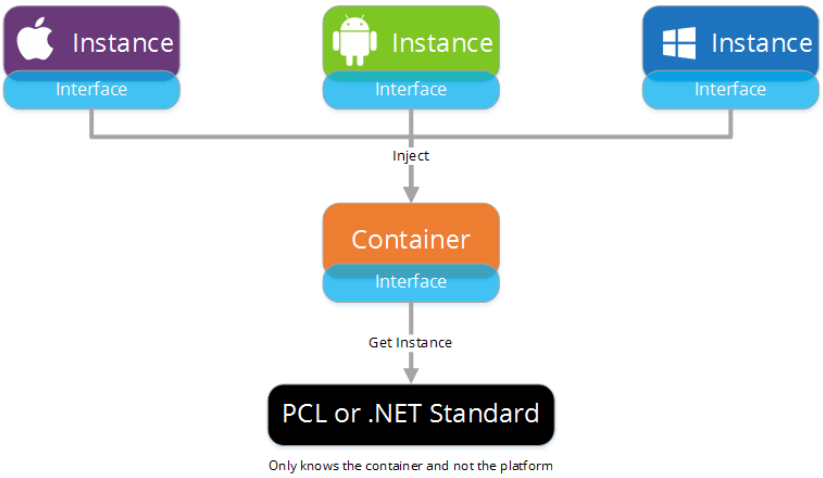
Pierwszy sposób polega na zdefiniowaniu w projekcie dla danej platformy zdarzenia, a następnie inicjowanie tego zdarzenia z klasy znajdującej się w części wspólnej.

Drugi mechanizm polega na tworzeniu abstrakcji w projekcie wspólnym (odpowiednich interfejsów deklarujących wymagane metody), a następnie implementowanie ich w projektach na specyficzne platformy. Stworzone implementacje możemy dostarczyć do części wspólnej wykorzystując mechanizmy o nazwie lokalizator usług (ang. *service locator*) oraz wstrzykiwanie zależności (ang. *dependency injection*), których obsługę dostarcza framework MvvmCross.

Lokalizator usług mapuje abstrakcje do jednego statycznego kontenera w ramach całej aplikacji. W razie potrzeby użycia danego zasobu wystarczy się do niego odwołać w dowolnym miejscu w programie przez statyczny obiekt. Dzięki lokalizatorowi, nie musimy za każdym razem tworzyć na nowo obiektów, które raz utworzone znajdują się ciągle w lokalizatorze. Niestety, w takim przypadku tracimy możliwość samodzielnego zarządzania czasem życia obiektów, które raz zarejestrowane w kontenerze, pozostają w pamięci aż do zakończenia działania całej aplikacji. Naruszamy również jedną z zasad SOLID (mnemonik zaproponowany przez Roberta C. Martina, opisujący pięć podstawowych zasad programowania obiektowego). Nie stosujemy się bowiem do zasady SRP (ang. *single responsibility principle*), która zakłada że każda klasa powinna mieć wyłącznie jedną odpowiedzialność. [16]

Drugi mechanizm to wstrzykiwanie zależności (ang. *dependency injection*, DI) Jego głównym założeniem jest przeniesienie tworzenia obiektów oraz wiązania ich między sobą poza kod aplikacji. Obiekty tworzy i wiąże osobna biblioteka nazywana kontener DI. W celu powiązania obiektów kontener posługuje się konfiguracją[3], która określa jak obiekty powinny być powiązane. [17]

[3] Do konfiguracji służy jedna klasa. Przy wykorzystaniu MvvmCrossa jest to klasa App umieszczona w części wspólnej. Przykład konfiguracji pokazano w rozdziale 3.



Rys.4 Schemat tworzenia i rejestrowania zależności za pomocą technologii Xamarin (źródło: *https://xamarinhelp.com/xamarin-forms-dependency-injection*)

Korzystając z Xamarin i MvvmCross, w projekcie części wspólnej tworzymy interfejs (przykład został pokazany na listingu 1). Implementującą go klasę tworzymy w projekcie pod daną platformę. Tam też rejestrujemy go w kontenerze DI.

Listing 1. Przykładowy interfejs do implementacji na poszczególnych platformach

```
public interface IDeviceInfo
{
    string GetUniqueIdentifier();
}
```

Listing 2. Przykładowa implementacja interfejsu `IDeviceInfo` na platformie iOS

```
public class IosDeviceInfo: IDeviceInfo
{
    public string GetUniqueIdentifier() => "IOS"
}
```

Listing 3. Przykładowa implementacja interfejsu `IDeviceInfo` na platformie UWP

```
public class UwpDeviceInfo: IDeviceInfo
{
    public string GetUniqueIdentifier() => "UWP"
}
```

Implementacje na poszczególne platformy są rejestrowane w kontenerze DI, co w przypadku frameworka MvvmCross sprowadza się do przeciążenia odpowiedniej metody w klasie odpowiadającej za konfigurację aplikacji. Dokładny proces rejestracji został omówiony w rozdziale 4.

Po zarejestrowaniu, framework MvvmCross automatycznie wstrzyknie odpowiednie implementacje interfejsów w wybrane miejsca w programie. Najpopularniejszą formą wstrzykiwania zależności jest wstrzykiwanie ich przez

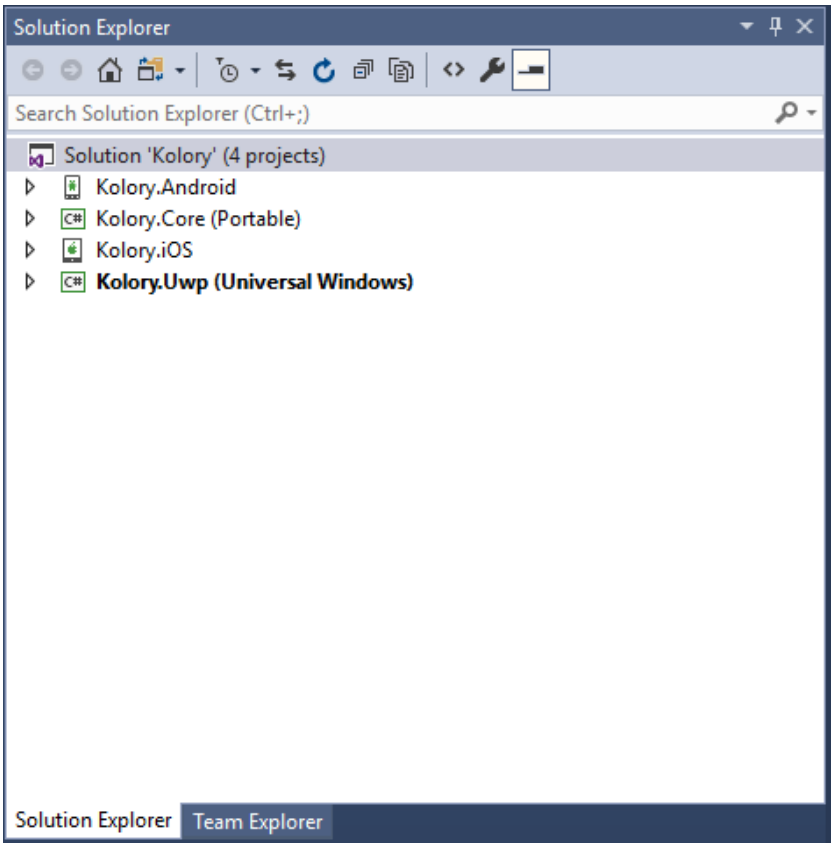
konstruktor klasy. Oznacza to, że w momencie przejścia do danego widoku, związany z nim model widoku automatycznie otrzyma nowy obiekt implementujący dany interfejs dostarczony przez kontener DI. [17]

Zaletą mechanizmu wstrzykiwania zależności jest to, że zamiast tworzyć ręcznie obiekty, oczekujemy ich dostarczenia „z zewnątrz" (obiekty są automatycznie rozpoznawane po implementowanym przez nie interfejsie). Dzięki takiemu podejściu w łatwy sposób można zmieniać implementację danej funkcjonalności tworząc nową klasę i rejestrując ją w kontenerze DI. Takie podejście wspiera również tworzenie testów jednostkowych, gdzie możemy zamiast konkretnych implementacji posłużyć się ich imitacjami (ang. *mock*), które będą zachowywać się w dokładnie określony sposób. [17]

Po wybraniu odpowiedniego typu projektu dla części wspólnej, tworzymy projekty na docelowe platformy i w każdym z nich umieszczamy odwołania do projektu części wspólnej. Dobrą praktyką jest stosowanie następującej konwencji nazw projektów:

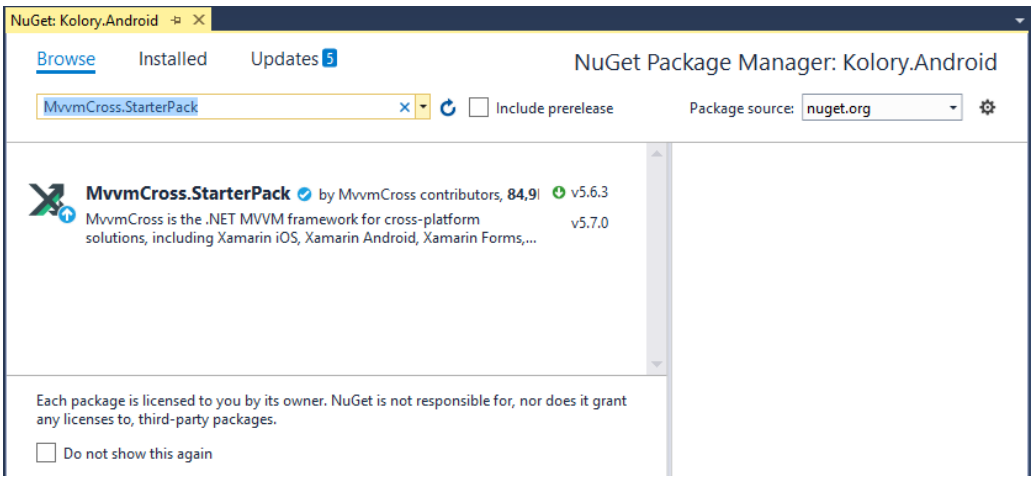
- nazwa całego rozwiązania odnośni się do nazwy aplikacji na poszczególnych platformach np. `Kolory`,
- do nazwania projektu części wspólnej wykorzystujemy nazwę rozwiązania oraz po kropce dodajemy przypisek `Core` (z ang. rdzeń) np. `Kolory.Core`,
- w projektach przeznaczonych na poszczególne platformy dodajemy po kropce identyfikator platformy np. `Kolory.Android`.

Stosując taką konwencję nazw, w jasny sposób identyfikujemy przeznaczenie poszczególnych projektów znajdujących się w rozwiązaniu. Przykład został pokazany na rysunku 5.



Rys.5 Struktura podziału projektów i ich nazw w ramach jednej solucji

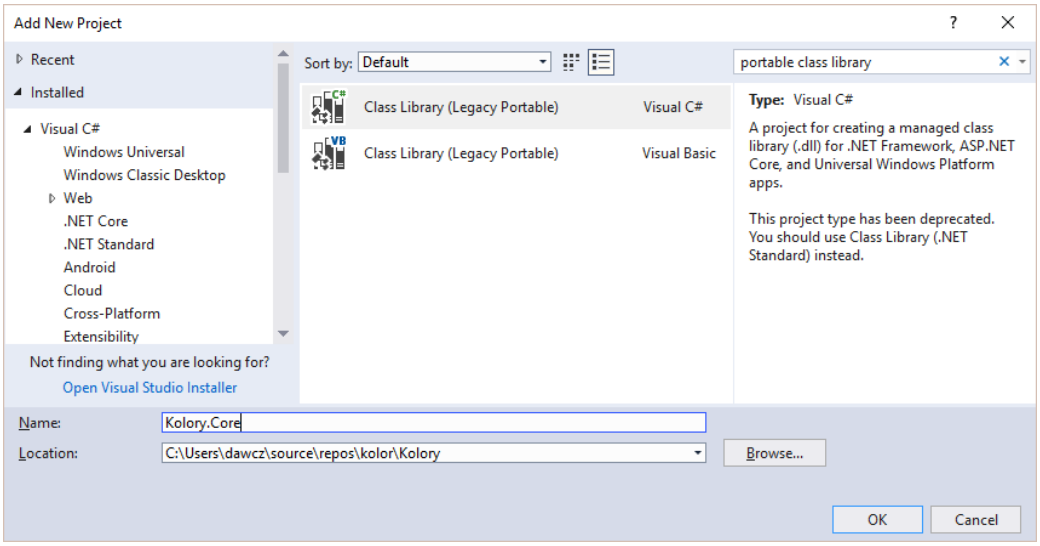
W każdym projekcie opisanym w tej pracy będziemy instalować pakiet startowy MvvmCross. Należy go zainstalować w każdym z projektów tj. zarówno w części wspólnej, jak i w projektach przeznaczonych na poszczególne platformy. Aby ten pakiet zainstalować przechodzimy do menadżera paczek NuGet, klikając prawym przyciskiem myszy na wybrany projekt, a następnie wyszukujemy w nim paczkę o nazwie *MvvmCross StarterPack* (rysunek 6).



Rys.6 Instalowanie startowego pakietu, frameworka MvvmCross

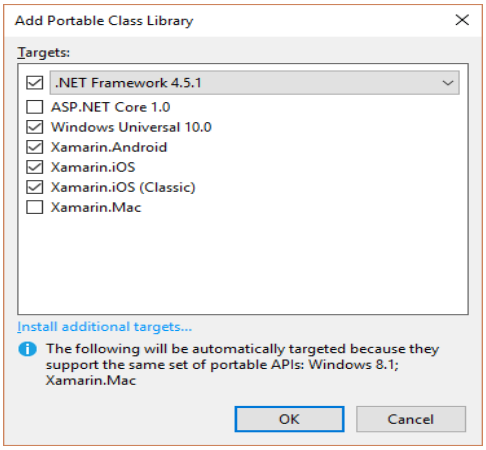
3.3 Tworzenie części wspólnej projektu

Pracę nad projektem pierwszej aplikacji rozpoczynamy od utworzenia projektu części wspólnej. Jest to projekt typu `PCL`, który nazwiemy `Kolory.Core`, a całą solucję `Kolory`. Ten przykładowy projekt przekształcimy potem w pierwszą aplikację opisaną w rozdziale 4.



Rys.7 Tworzenie projektu części wspólnej

Po wybraniu projektu typu *Class Library (Legacy Portable)* i nazwaniu go zgodnie z omówioną wyżej konwencją (rysunek 7) zostaniemy poproszeni o wybranie platform docelowych (ang. *target*) dla biblioteki. Target będzie określał z jakimi platformami będzie współpracowała ta biblioteka. Należy skonfigurować zakres wspieranych projektów w taki sposób, jaki został zaprezentowany na rysunku 8.



Rys.8 Wybór wspieranych rodzajów projektów

Na rysunku 8 widać że automatycznie dodawane są te typy projektów i platform, dla których i tak spełnione są warunki. W naszym przypadku oprócz zaznaczonych przez nas typów projektów, będziemy mogli wykorzystać tę bibliotekę także w projektach dla Windows 8.1 i w projektach *Xamarin.Mac*.

Po pomyślnym utworzeniu projektu możemy usunąć plik *Class1.cs*, który nie będzie nam potrzebn. Następnie przechodzimy do menadżera paczek NuGet i instalujemy pakiet *MvvmCross.StarterPack* (rysunek 6). Po jego pomyślnym zainstalowaniu, w projekcie pojawią się dwa pliki z klasami: *App.cs* oraz *MainViewModel.cs*.

Klasa `App` z pliku *App.cs* reprezentuje całą aplikację, będąc rodzajem jej klasy „konfiguracyjnej". Zapewnia m.in. inicjalizację podstawowej logiki biznesowej aplikacji i jej modeli widoku. Klasa ta zapewnia konfigurację następujących mechanizmów: [18]

- zasad wstrzykiwania zależności (DI) –możemy dla przykładu zdefiniować za pomocą metody `CreatableTypes` typy interfejsów, które będą automatycznie wstrzykiwane przez kontener DI (listing 4), [18]
- klasy `ViewModelLocator`, czyli sposobu wyszukiwania lub tworzenia modeli widoku (typ `ViewModels`) podczas wyświetlania widoków, [18]
- `IMvxAppStart` określający który model widoku będzie wykorzystany do wyświetlenia startowego widoku aplikacji (listing 4) [18]

Listing 4. Zawartość klasy `App`

```
public class App : MvvmCross.Core.ViewModels.MvxApplication
{
      public override void Initialize()
      {
            CreatableTypes()
                  .EndingWith("Service")
                  .AsInterfaces()
                  .RegisterAsLazySingleton();
            RegisterNavigationServiceAppStart<MainViewModel>();
      }
}
```

Wprowadźmy w tej klasie jedną zmianę: usuńmy wywołanie metody `RegisterAppStart`, a zamiast niej wywołajmy metodę `RegisterNavigationServiceAppStart` (listing 4).

Drugą klasą dodaną do naszego projektu jest `MainViewModel` (plik *MainViewMode.cs*). Jest to przykładowa klasa modelu widoku, która zostanie związana z widokami dla poszczególnymi platform. Dodatkowo ten model widoku zostanie domyślnie wczytany zaraz po uruchomieniu platformy, zgodnie z konfiguracją z metody `App.Initializate` (plik *App.cs*). Oznacza to że związany z nim widok, będzie pierwszym widokiem, jaki zobaczy użytkownik aplikacji.

Listing 5. Zawartość klasy `MainViewModel`

```
public class MainViewModel : MvxViewModel
    {
        public MainViewModel()
        {
        }

        public override Task Initialize() => base.Initialize();

        public IMvxCommand ResetTextCommand => new MvxCommand(ResetText);

        private void ResetText()
        {
            Text = "Hello MvvmCross";
        }

        private string _text = "Hello MvvmCross";
        public string Text
        {
            get { return _text; }
            set { SetProperty(ref _text, value); }
        }
    }
```

Jak widać na listingu 5, klasa `MainViewModel` dziedziczy po `MvxViewModel`, która jest konieczna do zapewnienia prawidłowego wiązania modeli widoku z widokami tworzonymi dla poszczególnych platform. [19]

W modelu widoku jest metoda o nazwie `Initialize`, która zwraca obiekt zadania (typu `Task`). Dzięki temu można oznaczyć ją jako asynchroniczną i użyć w niej operatora `await`. W tej metodzie należy wykonać wszystkie operacje ładowania, które mogą zająć więcej czasu. [19]

Warto również zwrócić na zdefiniowane w tej klasie polecenie przechowywane w referencji typu `IMvxCommand`. W MvvmCross dostępna jest gotowa implementacja polecenia – klasa MvxCommand, analogiczna do znanej z WPF klasy `RelayCommand`. Przechowuje ona i wykonuje operacji przekazane w argumencie konstruktora tej klasy. W naszym przypadku polecenie o nazwie `ResetTextCommand` jest wykorzystywane do

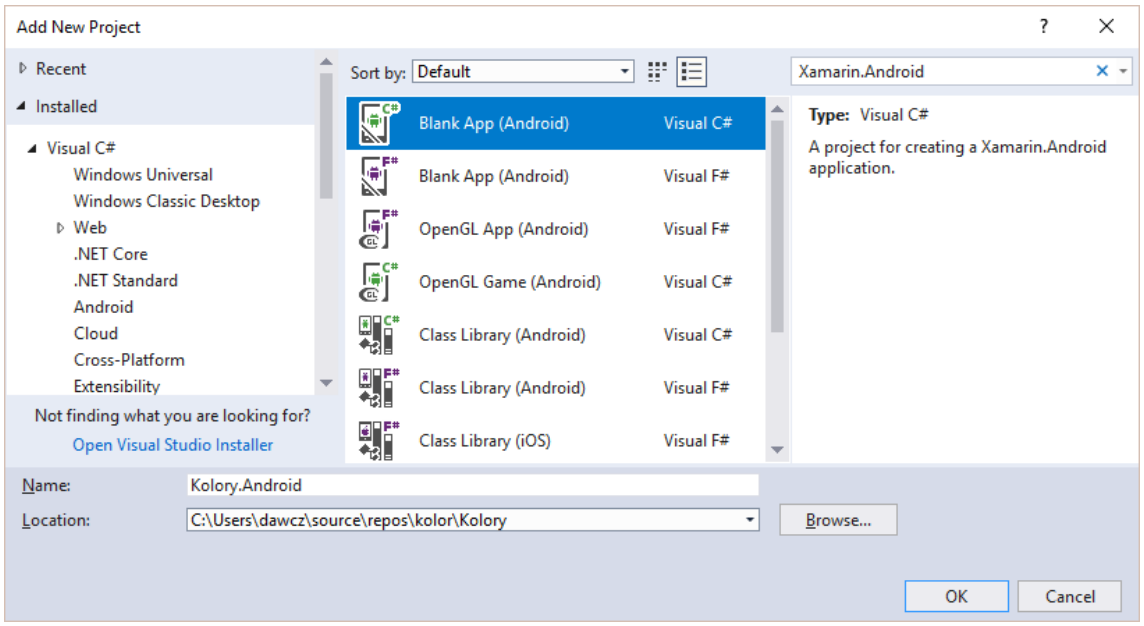
wywołania metody, która jest odpowiedzialna za wyczyszczenie wpisanego przez użytkownika tekstu. [19]

W klasie `MainViewModel` zdefiniowana jest także właściwość typu `string`, która będzie w naszym przypadku odpowiedzialna za przechowywanie tekstu wprowadzanego przez użytkownika. Klasa bazowa `MvxViewModel`, a właściwie jej klasa bazowa `MvxNavigatingObject` implementuje interfejs `INotifyPropertyChanged` i w konsekwencji ma zdefiniowaną metodę `SetProperty` powiadamiającą widok o konieczności odświeżenia kontrolek. [23]

Po tych przygotowaniach, biblioteka części wspólnej jest już gotowa do użycia w projektach na platformy `Xamarin.Android`, UWP oraz `Xamarin.iOS`.

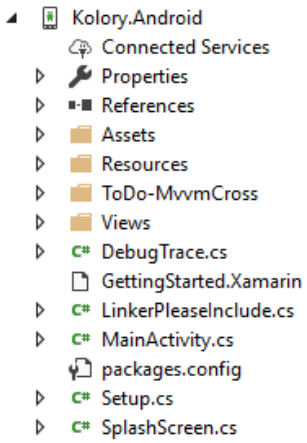
3.4 Projekt dla urządzeń z systemem Android

Dalszą pracę zaczynamy od dołożenia kolejnego projektu do stworzonego przez nas rozwiązania (solucji). W tym celu klikamy na nią prawym przyciskiem myszy w podoknie *Solution Explorer*, w menu kontekstowym przechodzimy do podmenu *Add* i wybieramy polecenie *New Project*. Pojawi się okno, w którym wyszukujemy projekt typu *Xamarin.Android* i tworzymy go nadając mu nazwę `Kolory.Android` (rysunek 9).



Rys.9 Tworzenie projektu na Androida

Po dodaniu do rozwiązania nowego projektu, instalujemy w nim pakiet *MvvmCross.StarterPack* (rysunek 2). Następnie dodajemy referencję do projektu części wspólnej (por. opis w podrozdziale 3.1). Po wykonaniu tych czynności, projekt na platformę Android powinien wyglądać tak, jak na rysunku 6.



Rys.10 Schemat projektu Xamarin.Android po zainstalowaniu frameworka MvvmCross

Po instalacji frameworka MvvmCross, pojawił się w projekcie folder *ToDo*, w którym zostały określone akcje, jakie należy wykonać żeby poprawnie skonfigurować projekt i przygotować go do współpracy z modelami widoku z części wspólnej.

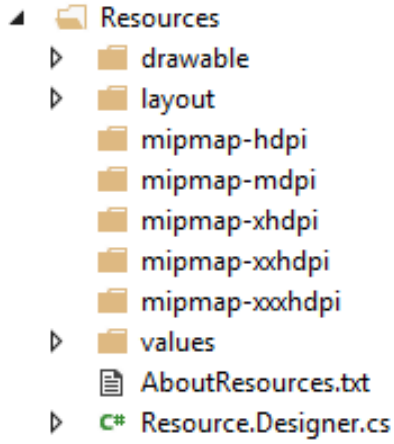
Dodatkowo pojawił się również folder *Views* (z ang. widoki), w którym umieszczane mają być klasy dziedziczące po klasie `MvxActivity`. Wszystkie te klasy, zastępują standardowe aktywności, które są swego rodzaju kontrolerami w platformie Android. Oryginalne aktywności łączą bowiem definicję widoku określaną w plikach `axml` i obsługę działań (zdarzeń) wykonywanych przez użytkowników za pomocą kontrolek z widoku. [20] My jednak nie będziemy korzystać ze standardowych aktywności. Dlatego usuwamy domyślnie umieszczony w projekcie plik *MainActivity.cs*.

Listing 6. Zawartość klasy `MainView`

```
[Activity(Label = "View for MainViewModel")]
public class MainView : MvxActivity
{
      protected override void OnCreate(Bundle bundle)
      {
            base.OnCreate(bundle);
            SetContentView(Resource.Layout.MainView);
      }
}
```

Na listingu 6, została przedstawiona klasa `MainView`. W atrybucie znajdującym się przed klasą, zdefiniowany jest tytuł widoku. W samej klasie znajduje się nadpisana metoda `OnCreate`. W niej oprócz wywołania metody `OnCreate` z klasy bazowej, wiążemy bieżącą klasę z interesującym nas widokiem. W tym przypadku jest to widok określony w pliku *MainView.axml*, który znajduje się w folderze `resources`, w pod folderze `layout`.

Tworząc klasy widoku w projektach dla systemu Android, należy zdawać sobie sprawę z jednej bardzo ważnej rzeczy. Należy wiedzieć, że aby związać utworzone w części wspólnej modele widoków z widokami musimy zachować spójną konwencję nazw tzn. jeżeli chcemy związać model widoku o nazwie `MainViewModel`, musimy utworzyć klasę `MainView`.Automatyczne wiązanie modelu widoku z widokiem wskazanym w metodzie `SetContentView` odbywa się na podstawie nazw obu klas. Bez tego, program nie będzie w stanie poprawnie obsługiwać modeli widoku. [19]



Rys.11 Struktura katalogu `Resources`

W projekcie dla systemu Android znajduje się folder *Resources*, w którym znajdują się następujące podfoldery przeznaczone na różne rodzaje zasobów:

- *drawable* – umieszczamy tutaj wszystkie pliki graficzne typu PNG lub JPG, z których będziemy chcieli korzystać w aplikacji; tam również umieszczane są pliki XML, które definiują obiekty wektorowe i kształty lub obrazy narysowane w kodzie; [20]
- *layout* - ten katalog zawiera pliki XML, w których zdefiniowany jest wygląd graficznego interfejsu użytkownika aplikacji; to właśnie tutaj umieszczamy wszystkie pliki widoku, które będziemy wiązać z modelami widoku utworzonymi w części wspólnej; [20]

- *mipmap* – zawartość tego podfolderu jest podobna do tego z *drawable*, z tą różnicą że katalog ten powinien być tylko używany do przechowywania ikon przeznaczonych do wyświetlania na ekranie telefonu (ang. *homescreen*), a nie w samej aplikacji [20]
- *values* - może zawierać wiele plików XML składających się z prostych wartości, które są używane w aplikacji; przykładem takiego pliku jest *strings.xml*, w którym zdefiniowane są łańcuchy używanej w etykietach kontrolek aplikacji; innym przykładem jest *styles.xml*, w którym przechowywane są ustawienia stylów powtarzających się w wielu miejscach aplikacji [20]

Na rysunku 11 widzimy że niektóre foldery są powielone, ale w nazwach pojawiły się przyrostki np. „hdpi" lub „xhdpi", które oznaczają różne rozdzielczości ekranu. Aby uniknąć obniżenia jakości użytych grafik, rysunki stworzone dla aplikacji powinny być przygotowane dla różnych gęstości pikseli i umieszczone w odpowiednich folderach. System Android pobierze właściwe rysunki dla rozdzielczości ekranu urządzeniu, na którym uruchamiana jest aplikacja. [20] Szczegółowe informacje na ten temat można znaleźć w oficjalnej dokumentacji, która znajduje się pod adresem: *https://docs.microsoft.com/en-us/xamarin/android/app-fundamentals/resources-in-android/?tabs=vswin*.

W katalogu *Resources* znajduje się również klasa `Resource.Designer`. Nie powinna ona być edytowana, ponieważ jej kod jest automatycznie tworzony na podstawie wprowadzanych przez nas zmian m.in. w pliku XML tworzącym warstwę widoku.

W katalogu *Resources* musimy dokonać dwóch zmian. Pierwsza polega na usunięciu pliku *Main.axml* z podkatalogu *layout* . Plik ten opisywał widok aktywności `MainActivity`. Następnie musimy dodać podfolder o nazwie *mipmap* i umieścić w nim ikonę, która zostanie wykorzystana jako ikona aplikacji. Przykładowa ikona do wykorzystania w aplikacji została umieszczona pod tym adresem: *http://fizyka.umk.pl/~jacek/dydaktyka/mobilne/xamarin/zasoby/Icon.png*

Folderem projektu, w którym możemy umieszczać zasoby inne niż pliki graficzne jest folder *Assets*. Mogą to być np. niestandardowe czcionki, które potem możemy wykorzystać w aplikacji.[22] Domyślnie w tym folderze umieszczany jest przykładowy

plik *AboutAssets.txt*, który zawiera kod pokazujący, w jaki sposób można wykorzystać czcionki w aplikacji.

Listing 7. Zawartość klasy `SplashScreen`

```
[Activity(
    Label = "Kolory.Android"
    , MainLauncher = true
    , Icon = "@mipmap/icon"
    , Theme = "@style/Theme.Splash"
    , NoHistory = true
    , ScreenOrientation = ScreenOrientation.Portrait)]
public class SplashScreen : MvxSplashScreenActivity
{
    public SplashScreen()
        : base(Resource.Layout.SplashScreen)
    {
    }
}
```

Oprócz klas widoku, w folderze głównym projektu dla systemu Android znajduje się również plik z klasą `SplashScreen` (listing 7). Odpowiada ona za załadowanie początkowego ekranu, który jest wyświetlany przez krótki okres po uruchomieniu aplikacji. Klasa ta ozdobiona jest atrybutem, który zawiera kilka parametrów wskazujących m.in. ikonę, która ma zostać użyta w aplikacji lub czy ekran początkowy (ang. *splash screen*) ma być traktowany jako normalna aktywność. Domyślnie ta wartość ustawiana jest na `false`, więc użytkownik nie będzie wstanie cofnąć się do tego widoku używając sprzętowego klawisza w swoim smartfonie.

W katalogu *Properties*, znajdują się dwa pliki: *AssemblyInfo.cs* i *AndroidManifest.xml*. Pierwszy z nich zawarty jest we wszystkich projektach .NET i zawiera informacje o projekcie (m.in. jego wersję), które można ustalić w ustawieniach projektu. Drugi jest ciekawszy. To tzw. plik manifestu używanego w systemie Android m.in. do wskazania obsługiwanych wersji tego systemu oraz wymaganych uprawnień aplikacji (np. zezwolenie na używanie czujników lub dostępu do kontaktów).Kluczową klasą, która pojawia się we wszystkich projektach jest klasa `Setup` (plik *Setup.cs*). Klasa ta jest utworzona w projektach na wszystkie platformy mobilne, na jakie będziemy tworzyć aplikacje z użyciem frameworka MvvmCross. [18]

Listing 8. Zawartość klasy `Setup`

```
public class Setup : MvxAndroidSetup
{
      public Setup(Context applicationContext) : base(applicationContext)
      {
      }

      protected override IMvxApplication CreateApp() => new Core.App();

      protected override IMvxTrace CreateDebugTrace()=> new DebugTrace();
}
```

Jej kod w przypadku platformy Xamarin.Android jest widoczny na listingu 8. Jak widać, w klasie `Setup` zostały nadpisane dwie metody: `CreateApp` oraz `CreateDebugTrace`. Pierwsza z nich korzysta z klasy `App` zdefiniowanej w części wspólnej, dzięki czemu „wie" w jaki sposób przygotować aplikacje do pierwszego uruchomienia. Natomiast druga metoda, zapewnia dodatkowe komunikaty w konsoli dotyczące akcji podejmowanych przez użytkowników aplikacji. Jest to przydatne np. w trybie debugowania na fizycznym urządzeniu, ponieważ dostarcza dokładniejszych informacji na temat ewentualnych błędów zarejestrowanych podczas działania aplikacji. [18] Więcej informacji na temat klasy `Setup`, przedstawiona zostanie w rozdziale 4.

Następną czynnością przygotowującą projekt będzie związanie utworzonych własności i poleceń w klasie modelu widoku `MainViewModel` z kontrolkami w widoku. Na platformie `Xamarin.Android` mamy możliwość ustawienia wiązania do modeli widoków na dwa sposoby: w pliku `axml` lub w klasach wiążących dany widok z modelem widoku (w tym przykładnie tą klasą jest `MainView`). Na listingu 9 widoczne jest przykładowe wiązanie z modelem widoku w pliku `axml`.

Listing 9. Zawartość pliku `MainView.axml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:local="http://schemas.android.com/apk/res-auto"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
<EditText
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:textSize="40dp"
      local:MvxBind="Text Text" />
<Button
      android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:text="Reset"
        android:textSize="40dp"
        local:MvxBind="Click ResetTextCommand" />
</LinearLayout>
```

Żeby poprawnie wykonać wiązanie, należy do elementu XML opisującego wygląd danej kontrolki dodać dodatkowy atrybut o nazwie `local:MvxBind`. W najprostszym przypadku przypisywana mu wartość składa się z dwóch części. W pierwszej wskazujemy akcję użytkownika wykonywaną na kontrolce. Dla przykładu `Click` w przypadku kontrolki `Button` oznacza związanie z zdarzeniem kliknięcia przycisku, a `Text` w przypadku kontrolki `EditText` odnosi się do wartości atrybutu przechowującego aktualnie wpisany tekst. Po spacji pododajemy natomiast nazwę własności lub polecenia zdefiniowanego w modelu widoku. Dzięki wywołaniu metody `SetContentView` w klasie związanej z modelem widoku (np. `MainView`), MvvmCross autoamtycznie rozpoznaje z jakim widokiem ma zostać związany dany model widoku.

W projekcie `Xamarin.Android` domyślnym trybem wiązania jest tryb dwukierunkowy. Możemy to jednak zmienić dodając po spacji i przecinku kierunek wiązania np. `local:MvxBind="Text UserName, Mode=OneWay"`. Poniżej znajdują się wartości, z jakich można w tym kontekście skorzystać:

- `One-Way` - tryb ten przekazuje wartości tylko z modelu widoku do widoku. Za każdym razem, gdy zmienia się wartość związanej własności z modelu widoku, widok jest powiadamiany i otrzymuje najnowszą wartość tej własności. Ten tryb wiązania jest przydatny, gdy na widoku chcemy pokazać dane przychodzące ze źródła dynamicznego np. z jakiegoś sensora. [23]
- `One-Way-To-Source` – ten rodzaj wiązania danych przekazuje wartości tylko z widoku do modelu widoku. Po zmianie właściwości widoku, odpowiednia własność modelu widoku zostanie zaktualizowana. Taki rodzaj wiązania może okazać się przydatny podczas zbierania nowych danych od użytkownika np. gdy użytkownik wypełnia pusty formularz. [23]
- `Two-Way` - ten tryb wiązania przekazuje wartości w obu kierunkach. Zmiany właściwości widoku i modelu widoku są monitorowane i jeśli którakolwiek z nich ulegnie zmianie, druga zostanie zaktualizowana. Takie wiązanie danych jest przydatne podczas edycji wpisów w istniejącym formularzu i jest jednym z najczęściej stosowanych rodzajów wiązania danych. [23]

- `One-Time` - ten rodzaj wiązania przekazuje wartości z modeli widoku do widoku, podobnie jak w przypadku trybu `One-Way`, jednak w tym przypadku wiązanie jest aktywne tylko raz, podczas pierwszego zestawienia wiązania danych, a następnie zmiany przestają być śledzone. Ten rodzaj wiązania nie jest zbyt często używany, ale może być przydatny w przypadku pól, które można konfigurować, ale które później już się nie zmieniają. [23]

Drugim sposobem wiązania modeli widoku z kontrolkami w warstwie widoku jest mechanizm „elastycznego API" (ang. *Fluent API*). Obsługuje on te same tryby wiązania, jakie możemy zadeklarować w kodzie AXML, jednak sam sposób związania jest zupełnie inny. W tym podejściu dla każdego widoku tworzymy tzw. zestaw wiązania danych (ang. *binding set*), w którym wskazujemy model widoku, jaki chcemy związać z widokiem. Następnie odwołując się do kontrolek znajdujących się w plikach opisujących widok, poprzez nadanie im unikatowych nazw (atrybuty `id`), możemy określić jaka własność kontrolki ma zostać związana z własnością modelu widoku. Określamy również kierunek, w jakim wiązanie ma przekazywać dane. [23] Po zadeklarowaniu wszystkich wiązań, musimy wywołać metodę `Apply` na rzecz całego zbioru. Bez jej wywołania wiązania nie zostaną utworzone. Przykładowe wiązanie danych korzystające z mechanizmu *Fluent API* zostało pokazane na listingu 10. W tym przykładzie najpierw tworzymy zestaw, który zwiąże `MainViewModel` z widokiem reprezentowanym przez klasę `MainView`. Następnie kontrolka o identyfikatorze `cardLabel`, a dokładniej jej atrybut `Text`, został związany z własnością modelu widoku o nazwie `UserName` w trybie `OneWay`. Na samym końcu została wywołana metoda aktywująca zadeklarowany zbiór wiązań danych. [23]

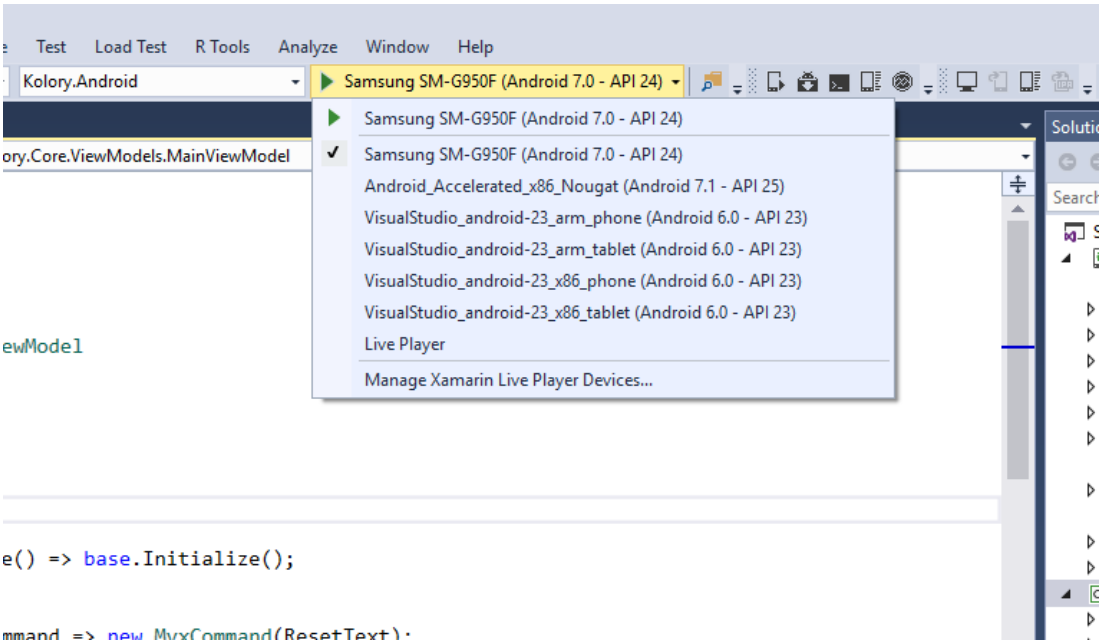
Listing 10. Przykład wiązania danych z wykorzystaniem *fluentApi*

```
var set = this.CreateBindingSet<MyView, MyViewModel>();
set.Bind(cardLabel)
    .For(v => v.Text)
    .To(vm => vm.UserName)
    .OneWay();
set.Apply();
```

W projekcie dla systemów Android oraz Universal Windows Platform, będziemy korzystać z wiązań danych definiowanych w plikach XML opisujących widok. Natomiast

Fluent Api wykorzystamy w projektach tworzonych na platformę iOS, w których jest to jedyny dostępny sposób.

Właściwie projekt jest już gotowy do uruchomienia. Możemy to zrobić na dwa sposoby: albo korzystając z emulatorów dostarczanych wraz ze środowiskiem Visual Studio albo wykorzystując podłączone do komputera rzeczywiste urządzenie z systemem Android.



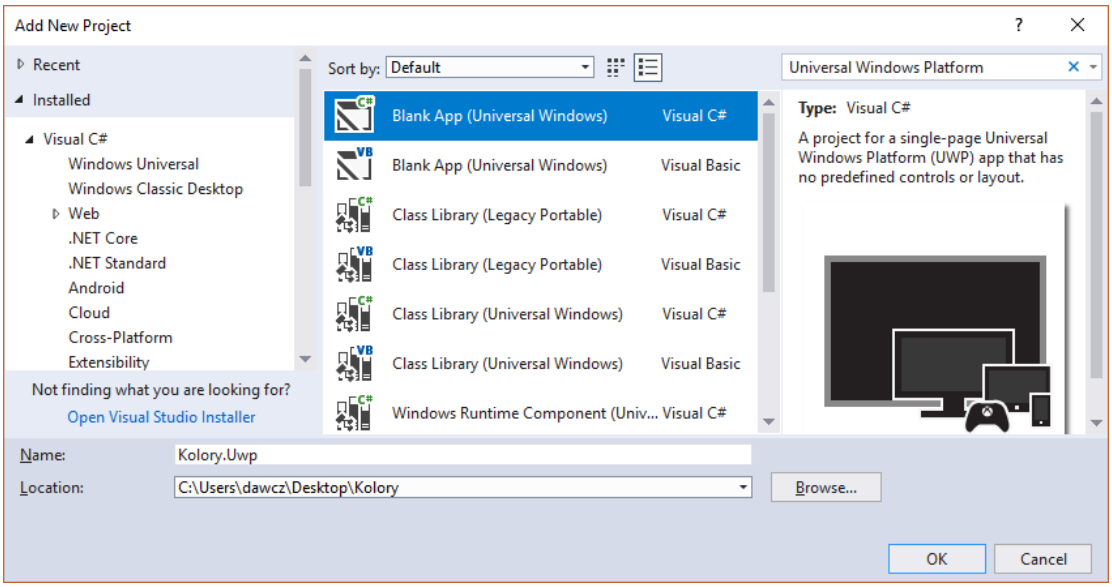
Rys.12 Miejscem w którym wybieramy urządzenia do debugowania

Obie opcje sprowadzają się do wyboru odpowiedniego urządzenia z rozwijanej listy znajdującej się przy nazwie aktualnie wybranego projektu (rysunek 12).

Jeżeli chcemy debugować aplikację uruchamianą na urządzeniu, po podłączeniu go przewodem USB do komputera, należy w ustawieniach telefonu uruchomić tryb deweloperski. W nowszych wersjach systemu oznacza to, że musimy wielokrotnie klikać w pozycję z numerem kompilacji systemu Android (pozycja *numer wersji*) widocznej w ustawieniach smartfona (w zakładce *informacje o telefonie*). Po odblokowaniu trybu dewelopera, przechodzimy do zakładki *Tryb Developera* i zezwalamy na *debugowanie USB*. Po włączeniu tej opcji telefon staje się widoczny z poziomu Visual Studio i jesteśmy w stanie debugować aplikacje bezpośrednio na nim.

3.5 Projekt dla platformy Universal Windows Platform

Przygotowanie widoku dla platformy UWP zaczynamy standardowo od dodania nowego projektu do istniejącej solucji *Kolory*. Tym razem będzie to projekt typu Universal Windows Platform (rysunek 13). Zgodnie z wyżej opisaną konwercją, nadamy mu nazwę *Kolory.Uwp*.



Rys.13 Tworzenie projektu na Uniwersal Windows Platform

Następnie dodajemy w nim referencję do projektu części wspólnej oraz zainstalujemy pakiet NuGet *MvvmCross.StarterPack*. Jednak po jej zainstalowaniu, w projekcie praktycznie nie nastąpią żadne zmiany. W przypadku platformy UWP, niestety będziemy musieli trochę bardziej się napracować przy konfiguracji frameworka MvvmCross, niż w przypadku projektów dla systemów Android i iOS.

W porównaniu z platformą Android, w projekcie dla UWP wszystkie zasoby wykorzystywane w projekcie, bez względu na to czy są to pliki graficzne czy inne, znajdują się w jednym folderze o nazwie *Assets*. Tam znajduje się m.in. domyślna ikona projektu oraz obraz, który jest wyświetlany w trakcie pierwszego uruchomienia aplikacji (ang. *aplash screen*).[24] Podobnie jak w przypadku platformy Android, w UWP także możliwe jest dopasowanie zasobów np. do rozmiaru ekranu i rozdzielczości urządzenia. W tym przypadku nie umieszczamy jednak zasobów w oddzielnych folderach, a stosujemy przyrostki w nazwach samych zasobów. np. *obraz-100.jpg*, *obraz-125.jpg* itp.[24] Dzięki temu zapewnimy optymalne wyświetlanie naszej aplikacji na różnych urządzeniach. Szczegółowe informacje na ten temat znajdują się w dokumentacji pod tym adresem:

https://docs.microsoft.com/en-us/windows/uwp/design/shell/tiles-and-notifications/app-assets.

Zmiany w projekcie rozpocznijmy od dodania klasy `Setup` (listing 11). I w tym przypadku nadpisujemy metody `CreateApp` i `GetDefaultLogProviderType`. Jednak tym razem blokujemy możliwość śledzenie błędów.

Listing 11. Zawartość klasy `Setup w projekcie UWP`

```
public class Setup : MvxWindowsSetup
{
      public Setup(Frame rootFrame) : base(rootFrame)
      {
      }

      protected override IMvxApplication CreateApp() => new Core.App();

      protected override MvxLogProviderType GetDefaultLogProviderType()
      => MvxLogProviderType.None;
}
```

Następnie modyfikujemy kod istniejącej klasy `App` z pliku *App.xaml.cs*, w którym musimy podmienić domyślną obsługę nawigacji. W tym celu modyfikujemy metodę `OnLaunched` zgodnie ze wzorem z listingu 12.

Listing 12. Zawartość podmienionej metody `OnLaunched` w klasie `App`

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
      Frame rootFrame = Window.Current.Content as Frame;
      if (rootFrame == null)
      {
            rootFrame = new Frame();
            rootFrame.NavigationFailed += OnNavigationFailed;
            Window.Current.Content = rootFrame;
      }

      if (e.PrelaunchActivated == false)
      {
            if (rootFrame.Content == null)
            {
                  var setup = new Setup(rootFrame);
                  setup.Initialize();
                  var start = MvvmCross.Platform.Mvx
                  .Resolve<MvvmCross.Core.ViewModels.IMvxAppStart>();
                  start.Start();
            }
            Window.Current.Activate();
      }
}
```

W następnym kroku przygotowujemy dodatkową klasę bazową dla tworzonych w projekcie dla UWP stron (ang. *pages*). Każda nasza klasa związana z widokiem, będzie dziedziczyć po klasie `BasePage` widocznej na listingu 13. Umieśćmy ją w nowym folderze o nazwie *Pages*, w pliku *BasePage.cs*. Tworząc dodatkową klasę dla wszystkich stron z projektu, zapewniamy sobie łatwy sposób konfigurowania i dodawania elementów na wszystkich stronach bez potrzeby powielania kodu.

Listing 13. Zawartość klasy `BasePage`, po której dziedziczą pozostałe strony w projekcie UWP

```
public class BasePage : MvxWindowsPage
{
      public BasePage()
      {
      }
}
```

Po utworzeniu projektu znajduje się w nim tylko jedna strona z klasą `MainPage`. Przenieśmy ją do utworzonego przed chwilą folderu *Pages*. Następnie zmieńmy jej kodzgodnie ze wzorem z listingu 14.

Listing 14. Zawartość klasy `MainPage` po wprowadzonych zmianach

```
namespace Kolory.Uwp.Pages
{
    public sealed partial class MainPage : BasePage
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Przejdźmy do edycji pliku w pliku *MainPage.xaml*. Należy zawarty w nim kod XAML przygotować do wiązania z modelem widoku. W tym celu skorzystamy ze standardowych mechanizmów funkcjonujących w platformie UWP, czyli umieszczania wiązania bezpośrednio w kodzie XAML. Zacznijmy od zmiany nazwy głównego elementu pliku (korzenia) z `Page` na `local:BasePage`. Nowa nazwa odnosi się do klasy po której dziedziczy klasa związana z danym widokiem. Dodatkowo w atrybucie `x:Class` musimy wskazać klasę *code-behind* tj. klasę `Kolory.Uwp.Pages.MainPage` z pliku *MainPage.xaml.cs*. Należy też zwrócić uwagę na atrybut `xmlns:local`, który definiuje

alias do przestrzeni nazw, w jakiej znajduje się związaną ze stroną klasa. Powinien to być przestrzeń `Kolory.Uwp.Pages`.

Listing 15. Zawartość kodu `XAML` związanego z klasą `MainPage` po wprowadzonych zmianach

```
<local:BasePage
    x:Class="Kolory.Uwp.Pages.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Kolory.Uwp.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
        <StackPanel>
            <TextBox Text="{Binding Text, Mode=TwoWay}" />
            <Button HorizontalAlignment="Center"
                    Command="{Binding ResetTextCommand}">
                Reset Button
            </Button>
        </StackPanel>
    </Grid>
</local:BasePage>
```

Następnie na stronie tworzymy znacznik siatki (`Grid`). Jako koloru tła użyjemy pędzla z zasobów aplikacji o nazwie `ApplicationPageBackgroundThemeBrush`, który jest kolorem z domyślnie ciemnego tematu kolorystycznego. Wewnątrz siatki umieszczamy pojemnik `StackPanel` organizującego ułożenie kontrolek. Umieszczone wewnątrz niego kontrolki będą układane jedna pod drugą (można też wymusić ustawienie ich poziomie). W tym pojemniku umieścimy kontrolki typu `TextBox` oraz `Button`.

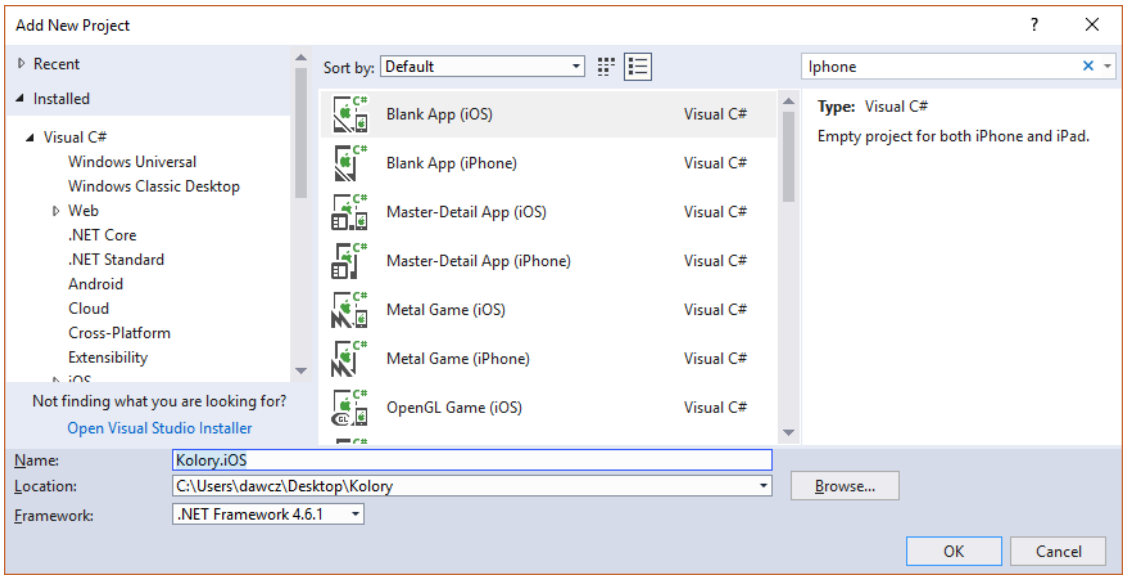
Pierwsza, zostaje związana z własnością `Text` znajdującą się w modelu widoku (w trybie dwu kierunkowym). Przycisk natomiast zostaje związany z poleceniem `ResetTextCommand`, korzystając z atrybutu `Command`. Kompletny kod XAML po zmianach został pokazany na listingu 15.

Osoby programujące w UWP bez frameworka MvvmCross zwrócą na pewno uwagę na brak elementu `local:BasePage.DataContext`. Nie jest on potrzebny ponieważ dzięki frameworkowi model widoku zostanie automatycznie związany z odpowiadającemu mu widokiem. Dzieję się tak dzięki temu, że framework automatycznie

dopasowuje po nazwie modele widoku z modelami, które dziedziczą bazowy typ `MvxWindowsPage`.[4]

3.4 Tworzenie części przeznaczonej na iOS

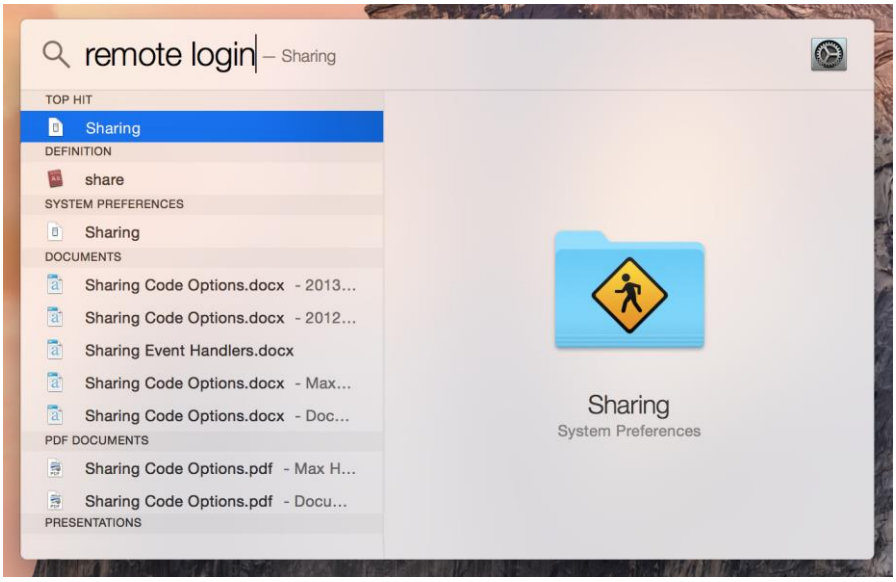
Tworzenie projektu widoku dla urządzeń z systemem iOS zaczynamy od dodania do istniejącego rozwiązania *Kolory* nowego projektu typu *Blank App* (*iOS*), co zostało pokazane na rysunku 14.



Rys.14 Tworzenie projektu na platformę iOS

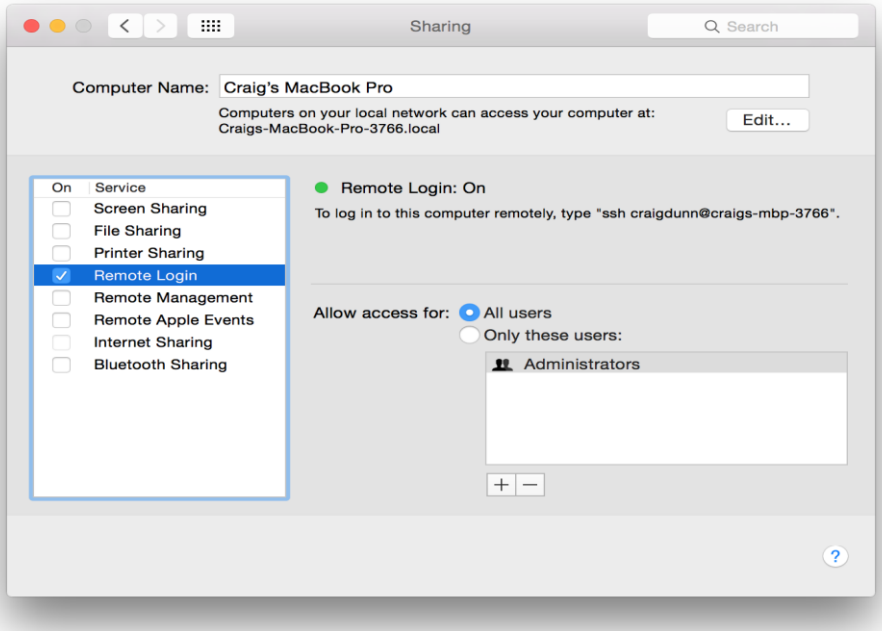
Następnie musimy dodać referencję do istniejącej części wspólnej oraz zainstalować pakiet startowy *MvvmCross.StarterPack*. W przypadku projektu dla iOS, podobnie jak w przypadku projektu dla systemu Android, pojawia się folder *ToDo*, w którym znajduje się plik tekstowy z opisem czynności, jakie należy wykonać, żeby framework mógł w sposób prawidłowy współpracować z platformą iOS.

[4] DataContext klasy `MainPage`, która dziedziczy po `MvxWindowsPage`, zostanie związany z `MainViewModel` za pomocą frameworka MvvmCross bez potrzeby dodatkowej deklaracji.



Rys.15 Zezwalanie na zdalny dostęp na maszynie z macOS
(źródło: *https://docs.microsoft.com/pl-pl/xamarin/ios/get-started/installation/windows/connecting-to-mac/*)

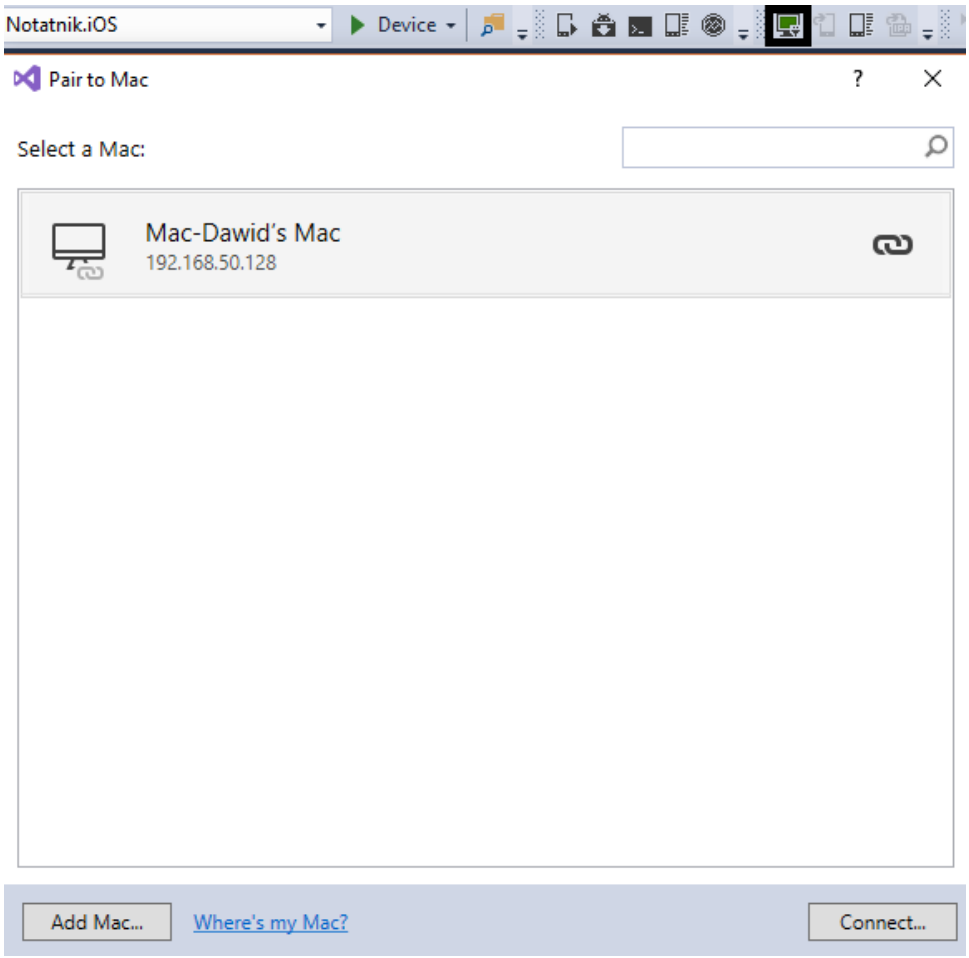
Zanim zajmiemy się samym projektem, należy najpierw zestawić połączenie z innym komputerem, na którym jest zainstalowany system macOS. Na komputerze z maxOS należy przejść do zakładki pozwalającej na zdalne logowanie. W tym celu otwieramy *Spotlight* (*⌘-Space*) i wyszukujemy pozycję *Remote Login*, a następnie wybieramy wynik *Sharing* (rysunek 15). Spowoduje to otwarcie preferencji systemowych w panelu udostępniania, które są widoczne na rysunku 16.



Rys.16 Zezwalanie na zdalny dostęp na maszynie z macOS, ustawianie wymaganych opcji
(źródło: *https://docs.microsoft.com/pl-pl/xamarin/ios/get-started/installation/windows/connecting-to-mac/*)

W nowo otwartym oknie należy zaznaczyć opcję zdalnego logowania (*Remote Login*) na liście usług po lewej stronie, aby umożliwić programowi *Xamarin for Visual Studio* połączenie z komputerem *Mac*. Co ważne, należy się upewnić, że opcja logowania zdalnego jest ustawiona tak, że zezwala na dostęp dla wszystkich użytkowników lub że nazwa bieżącego użytkownika znajduje się w grupie użytkowników, którym zezwalamy na zdalny dostęp do komputera. Lista ta znajduję się po prawej stronie w oknie *Sharing*.

Po wykonaniu tych czynności, jeżeli komputer z systemem macOS jest aktywny, powinien zostać wykryty przez środowisko Visual Studio (zakładamy oczywiście, że należą do tej samej sieci). Visual Studio sam uruchomi i później zatrzyma agenta na komputerze z systemem macOS.



Rys.17 Miejsce z którego możemy zestawić połączenie z komputerem z systemem macOS.

W tym momencie w środowisku Visual Studio, dostępny stanie się komputer z systemem macOs, do którego można połączyć się za pomocą protokołu SSH, podając

nazwę użytkownika i hasło. Robimy to wybierając odpowiedni komputer w karcie *Xamarin Agent*, do której można przejść klikając ikonę zaznaczoną czarnym prostokątem w prawym górnym rogu rysunku 17.

Następnie przejdźmy do przygotowania projektu. Zacznijmy od zmian w klasie `AppDelegate`, dodając do niej kod, który znajduje się w pliku *AppDelegate.cs.txt* dodanym do projektu w po zainstalowaniu pakietu *MvvmCross.StarterPack*. Klasa ta udostępnia zdarzenia, dzięki którym jesteśmy powiadamiani o zdarzeniach występujących w trakcie całego cyklu życia aplikacji.[25] Dostarcza również metody wirtualne, które zostaną w projekcie nadpisane, dzięki czemu framework MvvmCross jest w stanie zmienić cykl życia aplikacji na pożądany we wzorcu MVVM[5]. Po przekopiowaniu kodu z pliku znajdującego się w folderze *ToDo*, ciało klasy `AppDelegate` powinno wyglądać tak, jak na listingu 16.

Listing 16. Zawartość klasy `AppDelegate` po wprowadzonych zmianach

```
[Register("AppDelegate")]
public partial class AppDelegate : MvxApplicationDelegate
{
      public override UIWindow Window { get; set; }

      public override bool FinishedLaunching(UIApplication application,
      NSDictionary launchOptions)
      {
            Window = new UIWindow(UIScreen.MainScreen.Bounds);

            var setup = new Setup(this, Window);
            setup.Initialize();

            var startup = Mvx.Resolve<IMvxAppStart>();
            startup.Start();

            Window.MakeKeyAndVisible();

            return true;
      }
}
```

Klasa `AppDelegate` zarządza cyklem życia aplikacji. Jest tak dzięki poleceniu `UIApplication.Main(args, null, "AppDelegate")` wywołanemu w statycznej metodzie `Main` klasy `Application`. Po wywołaniu tej metody, przekazywane jest sterowanie cyklem życia aplikacji klasie `AppDelegate`. [25]

[5] Zamiast opierać się na zdarzeniach, reakcje na zmiany w widoku obsługiwane są w modelach widoku.

Kolejny raz pojawia się również klasa `Setup`. Jej domyślna implementacja, podobnie jak w przypadku Androida, nie musi być modyfikowana. Dzięki temu aplikacja na iOS będzie zachowywać się tak samo, jak w obu pozostałych projektach widoków.

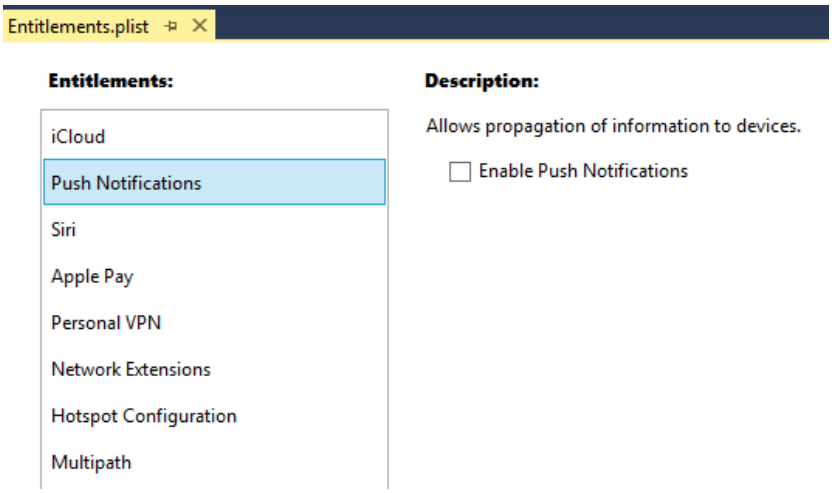
Dodatkowo w projekcie został utworzony folder `Views` z trzema plikami: *MainView.cs*, *MainView.designer.cs* oraz *MainView.xib*.

Zaczniemy od omówienia ostatniego z nich. To plik określający układ kontrolek, a w tym samym warstwę widoku aplikacji dla systemu iOS (odpowiednik pliku *.xaml* w projekcie dla UWP). Do jego edycji służy *designer*, w którym możemy budować wygląd interfejsu przeciągając kontrolki (np. przyciski) na podgląd okna, a który będzie działał tylko jeżeli zestawimy połączenie z komputerem z systemem macOS.

Kod w pliku *MainView.designer.cs* generowany jest automatycznie podczas umieszczaniu i konfigurowaniu kontrolek w pliku *MainView.xib*. Kod z tego pliku nie powinien być edytowany „ręcznie”.

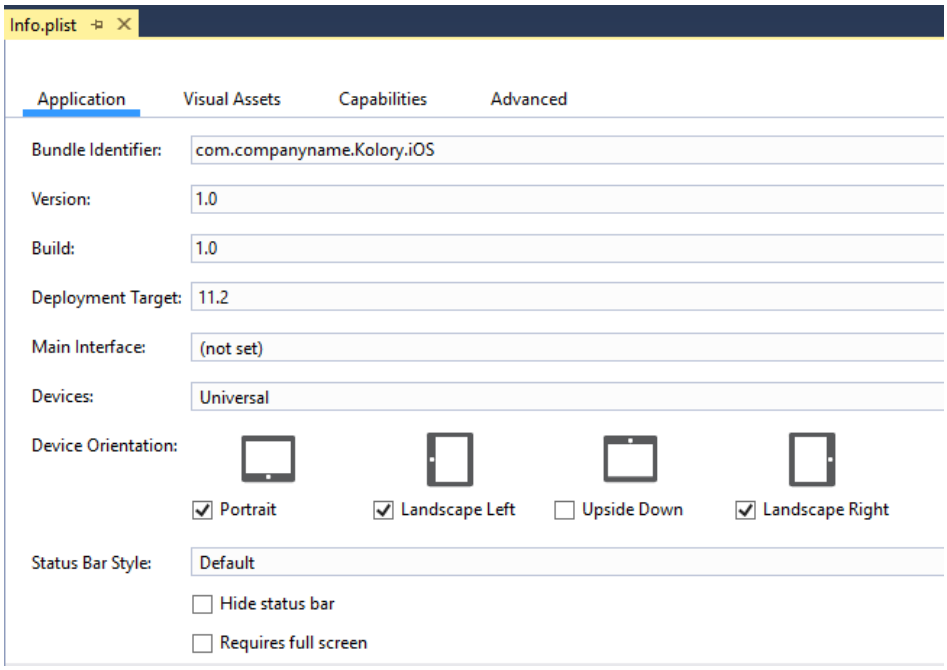
Na końcu musimy związać plik z rozszerzeniem *.xib* określający ułożenie kontrolek z modelami widoku z części wspólnej. Jak wspomniałem wyżej, w projektach dla systemu iOS możliwe jest to tylko z poziomu kodu. Odpowiednie polecenia umieścimy w klasie `MainView`. W konstruktorze tej klasy wywoływany jest konstruktor klasy bazowej `MvxViewController`, której pierwszym argumentem jest łańcuch zawierający nazwę widoku, z którym będzie związana. Następnie w metodzie `ViewDidLoad`, wskazujemy z jakim konkretnie modelem widoku ma zostać związany widok. W tej metodzie definiujemy również wiązania między klasą modelu widoku a konkretnymi kontrolkami użytymi w widoku zdefiniowanym przez plik *.xib*. W tym celu korzystamy z *Fluent API* (por. listing 10 z podrodziału dotyczącego projektu dla systemu Android). W przypadku platformy iOS mechanizm ten jest szczególnie przydatny, ponieważ format układu *xib* jest mało przyjazny do edycji przez programistę dlatego wszystkie wiązania w tej platformie definiowane są właśnie za pomocą kodu *C#*, a nie w języku znaczników.

W projekcie znajdują się dwa pliki z rozszerzeniami *.plist*. Pierwszy z nich to *Entitlements.plist*, w którym możemy określić uprawnienia, jakie przysługują aplikacji. Przykładowo, możemy zezwolić na wysyłanie powiadomień (ang. *push notifications*), które będą wyświetlane na belce powiadomień urządzenia (przykład został pokazany na rysunku 18).[26]



Rys.18 Konfiguracja ustawień w pliku `Entitlements.plist` w Visual Studio 2017

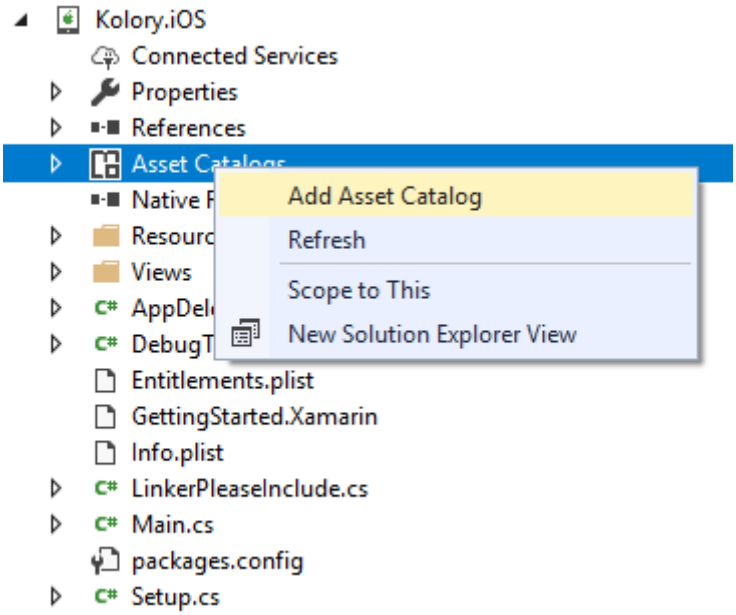
Drugi plik o nazwie *Info.plist* pozwala na zmianę ustawień i zachowania samej aplikacji. Tu możemy ustawić m.in. tryby w jakich może być wyświetlana aplikacja (poziomo/pionowo), wskazać domyślne źródło ikon, z których będzie korzystać aplikacja, czy wybrać minimalną wersję systemu, na jakim można uruchomić aplikację (rysunek 19).[26]



Rys.19 Konfiguracja ustawień w pliku `Info.plist` w Visual Studio 2017

W podoknie Solution Explorer projektu dla systemu iOS znajduje się również pozycja *Asset Catalogs*, w którym można umieszczać kontenery z plikami zasobów. Aby

utworzyć pierwszy kontenerkliknijmy prawym przyciskiem myszy na *Assets Catalogs* i wybierzmy polecenie *Add Asset Catalog* (rysunek 20). Nazwijmy go np. *Media*.



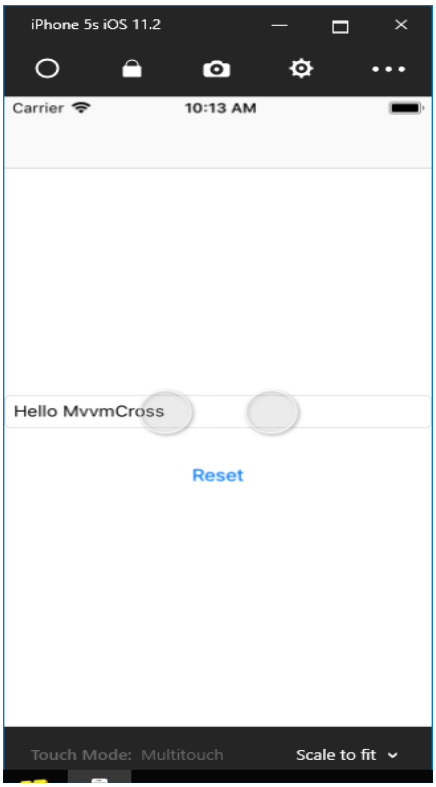
Rys.20 Dodawanie katalogu trzymającego zasoby graficzne dla aplikacji na platformę iOS.

Po dodaniu katalogu, możemy umieszczać w nim nowe zasoby obrazów. Czynności te zostały opisane w oficjalnej dokumentacji dostępnej pod adresem: *https://docs.microsoft.com/en-us/xamarin/ios/app-fundamentals/images-icons/displaying-an-image?tabs=vsmac*.

W tym momencie aplikacja jest gotowa do uruchomienia. Możemy to zrobić ustawiając tryb uruchomienia na *iPhoneSimulator* (rysunek 21), a następnie wybierając interesujący nas typ urządzenia.

Rys.21 Prawidłowe ustawienie trybu uruchomieniowego projektu Xamarin.iOS.

Po uruchomieniu aplikacji, uruchomiony zostanie w środowisku Windows emulator wybranego urządzenia. Korzysta z zasobów dostarczanych przez komputer z systemem macOS połączoną zdalnie z Visual Studio. Zrzut ekranu z przykładowego emulatora jest widoczny na rysunku 22.



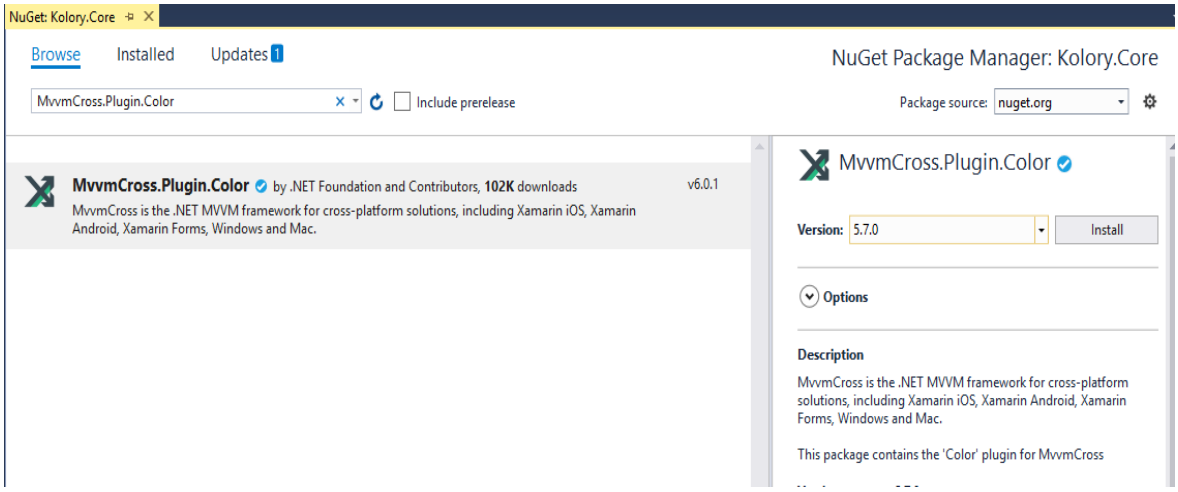
Rys.22 Emulator iPhone uruchomiony pod środowiskiem Windows

4. Przykładowe Aplikacje

4.1 Aplikacja Kolory

W kolejnej aplikacji wykorzystamy rozwiązanie opisane w rozdziale 3. Dodatkowo skorzystamy z dwóch nowych mechanizmów: wtyczek (ang. *plugins*) oraz konwerterów (ang. *converters*).

Wtyczki to gotowe biblioteki dostarczające interfejsy do części wspólnej projektu oraz gotowe implementacje klas do projektów przeznaczonych na konkretne platformy. W tym projekcie wykorzystamy wtyczkę o nazwie `MvvmCross.Plugin.Color`, która posłuży do poprawnej obsługi i konwersji kolorów z palety barw RGB na specyficzną reprezentacje barw na platformie *Android, UWP* oraz *iOS*.



Rys.23 Instalowanie wtyczki `MvvmCross.Plugin.Color` dla projektu części wspólnej.

Dodatkowo wykorzystamy mechanizm konwerterów, który pozwala przekształcać wartości przesyłane między modelami widoku a widokiem. Dzięki konwerterom, w sposób poprawny obsłużymy wartości składowych R, G oraz B koloru ustawiane przez użytkownika w widoku aplikacji i ich przekształcenie do reprezentacji całego koloru w niższych warstwach.[27]

Kontynuujemy rozwój aplikacji opisanej w poprzednim rozdziale. Dalszy jej rozwój rozpoczynamy od zainstalowania w projekcie Kolory.Core paczki *NuGet* o nazwie *MvvmCross.Plugin.Color*. Przykładowa instalacja została przedstawiona na rysunku nr. 23.

4.1.1 Aplikacja Kolory – projekt części wspólnej

Po zainstalowaniu tej wtyczki, należy dokonać modyfikacji kodu w istniejącym modelu widoku `MainViewModel` wg wzoru z listingu 17.

Listing 17. Zawartość klasy `MainViewModel`, po zmianach

```
public class MainViewModel : MvxViewModel
{
      private MvxColor _mvxColor;

      public MainViewModel() => _mvxColor = new MvxColor(0, 0, 0);

      public IMvxCommand ResetColorCommand => new MvxCommand(ResetColor);

      private void ResetColor()
      {
            R = 0;
            G = 0;
            B = 0;
      }

      public int R
      {
            get { return _mvxColor.R; }
            set
            {
                _mvxColor.R = value;
                RaisePropertyChanged("R");
                RaisePropertyChanged("CurrentColor");
            }
      }

      public int G
      {
            get { return _mvxColor.G; }
            set
            {
                _mvxColor.G = value;
                RaisePropertyChanged("G");
                RaisePropertyChanged("CurrentColor");
            }
      }

      public int B
      {
            get { return _mvxColor.B; }
            set
            {
                _mvxColor.B = value;
                RaisePropertyChanged("B");
                RaisePropertyChanged("CurrentColor");
            }
      }


      public MvxColor CurrentColor
      {
```

```
            get { return _mvxColor; }
            set { CurrentColor = value; }
        }

}
```

Dodane zostało prywatne pole typu `MvxColor`. Jest to klasa dostarczana wraz z zainstalowaną przez nas wtyczką. Posiada ona własności przechowujące składowe koloru R, G, B oraz jest używana przez konwertery do tworzenia typów opisujących kolory w poszczególnych platformach (zostaną one opisane w kolejnych podrozdziałach).

Zdefiniowane zostało również polecenie `ResetColorCommand`, które posłuży do wywołania metody, odpowiedzialnej za resetowanie składowych R, G, B do stanu początkowego, czyli do wartości 0.

Do klasy `MainViewModel` dodajemy również trzy własności odpowiedzialne za dostarczanie i aktualizowanie kontrolowanych przez użytkowników aplikacji składowych R, G, B koloru. Zwracają one wartości pobrane z instancji utworzonej na samym początku klasy `MvxColor`.[27] Podczas wykonywania kodu odpowiedzialnego za ustawienie wartości własności (sekcja `set`), wywoływana jest metoda `RaisePropertyChange`, która służy do powiadamiania widoku (poprzez interfejs `IMvxNotifyPropertyChanged` implementowany przez klasę abstrakcyjną `MvxNotifyPropertyChanged` po której dziedziczy `MvxViewModel`) o aktualizacji danej własności. Bez jej wywołania, w momencie przesuwania suwaków nie aktualizowały by się kontrolki tekstowe prezentujące składowe RGB.

Metoda `RaisePropertyChange` jest wywoływana również w przypadku własności o nazwie `CurrentColor`. Zwraca ona cały obiekt koloru. Obiekt ten, będzie konwertowany na poszczególnych platformach do obiektów tworzących jeden kolor ze poszczególnych składowych RGB.

W tym momencie część wspólna projektu zawiera wszystkie rzeczy niezbędne, do rozpoczęcia implementacji aplikacji na platformach Android, Uniwersal Windows Platform oraz iOS.

4.1.2 Aplikacja Kolory – projekt Android

Pracę nad stworzeniem aplikacji na Androida należy rozpocząć od instalacji w projekcie *Kolory.Android* wtyczki `MvvmCross.Plugin.Color`. Po jej zainstalowaniu, możemy zauważyć że został utworzony nowy folder o nazwie *Bootstrap* oraz została dodana tam klasa o nazwie `ColorPluginBootstrap`.

Listing 18. Zawartość klasy `ColorPluginBootstrap`

```
public class ColorPluginBootstrap
      : MvxPluginBootstrapAction<MvvmCross.Plugins.Color.PluginLoader>
{

}
```

Jak widać na listingu 18, jest to niewielka klasa, która dziedziczy po klasie `MvxPluginBootstrapAction` parametryzowanej typem dziedziczącym po `IMvxPluginLoader`. Typ ten pochodzi z zainstalowanej przez nas wtyczki *MvvmCross.Plugin.Color*.

MvvmCross pozwala na automatyczne rejestrowanie zainstalowanych w aplikacji wtyczek podczas jej uruchamiania. Wówczas framework szuka wszystkich typów implementujących interfejs `IMvxBootstrapAction`. Następnie w każdej wtyczce wywoływana jest metoda `EnsureLoaded` (w klasie `PluginLoader`), która rejestruje dany typ w projekcie przeznaczonym na specyficzną platformę.

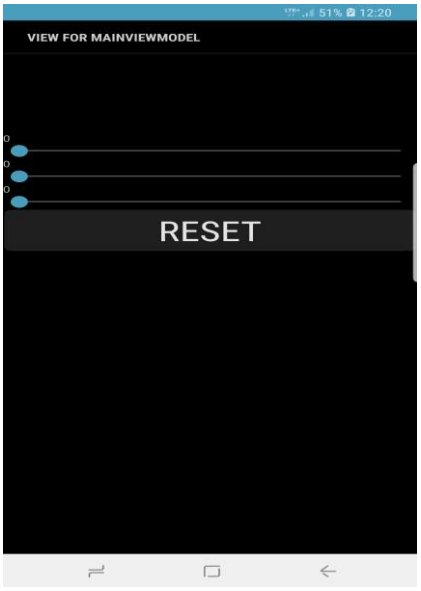
Istnieje również alternatywny sposób rejestracji wtyczek. Polega on na przeciążeniu metody znajdującej się w klasie `Setup`. Sposób takiego rejestrowania wtyczek, został pokazany w podrozdziale dotyczącym tworzenia aplikacji kolory na platformę UWP.

W tym momencie jedyne, co nam pozostało to uzupełnienie pliku *MainLayout.axml* o kod z listingu 19, który wykorzystuje utworzony w części wspólnej model widoku.

Listing 19. Zawartość pliku `MainView.axml`, po wprowadzonych zmianach

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:local="http://schemas.android.com/apk/res-auto"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <View
            local:MvxBind="BackgroundColor NativeColor(CurrentColor)"
            android:layout_width="match_parent"
            android:layout_height="300px"
            android:id="@+id/view1" />
      <TextView
            local:MvxBind="Text R"
            android:textAppearance="?android:attr/textAppearanceSmall"
```

```
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/textR" />
        <SeekBar
            local:MvxBind="Progress R,Mode=TwoWay"
            android:max="255"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/seekBarR"
            android:layout_marginRight="0.0dp" />
        <TextView
            local:MvxBind="Text G"
            android:textAppearance="?android:attr/textAppearanceSmall"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/textG" />
        <SeekBar
            android:max="255"
            local:MvxBind="Progress G,Mode=TwoWay "
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/seekBarG" />
        <TextView
            local:MvxBind="Text B"
            android:textAppearance="?android:attr/textAppearanceSmall"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/textB" />
        <SeekBar
            local:MvxBind="Progress B,Mode=TwoWay"
            android:max="255"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/seekBarB" />
        <Button
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Reset"
            android:textSize="40dp"
            local:MvxBind="Click ResetColorCommand" />
</LinearLayout>
```



Rys.24 Utworzony widok za pomocą kodu axml z listingu 19.

Na samym początku została utworzona kontrolka typu `View`, która wiąże wartość `BackgroundColor` do własności `CurrentColor` znajdującej się w modelu widoku. Dodatkowo jest tutaj wykorzystywany konwerter o nazwie `NativeColor`. W przypadku implementacji na platformę Android zamienia on obiekt typu `MvxColor` na obiekt typu `Android.Graphics.Color`.

W efekcie do `BackgroundColor` przypisywany będzie koloru typu właściwego dla danej platformy, utworzony na podstawie wartości zdefiniowanych we własnościach R, G i B. Ich wartości ustalane są przez użytkownika przy pomocy kontrolek typu `SeekBar`, w których elementach obecne są wiązania atrybutu `Progress` do własności `R`, `G` i `B` modelu widoku przy wykorzystaniu dwukierunkowego mechanizmu wiązania danych.

Oznacza to, że w momencie wywołania metody `RaisePropertyChanged` w modelu widoku `MainViewModel`, zaktualizowana zostanie własność `Progress` kontrolki `SeekBar` i tym samym zmieni się jej wygląd. Element XML opisujący poszczególne kontrolki `SeekBar` mają również atrybuty `max` ustalający maksymalną wartość przyjmowaną przez własności `Progress` równą 255..

Do widoku dodane zostały również kontrolki typu `TextView`, w których obecne jest atrybutu `Text` z własnościami `R`, `G` i `B` modelu widoku. Dzięki temu wyświetlają one wartości poszczególnych składowych koloru. Ich wartości są odświeżane za każdym razem, gdy zmienia się wartości wskazane w kontrolkach `SeekBar`. Ostatnim elementem

widoku jest przycisk, który związany jest z poleceniem `ResetColorCommand`. Będzie ono wywoływane za każdym razem, gdy przycisk zostanie naciśnięty.

Projekt na platformę Android jest gotowy. Należy go skompilować i uruchomić na urządzeniu testowym lub w emulatorze.

4.1.3 Aplikacja Kolory – projekt Uniwersal Windows Platform

Również i w przypadku projektu dla platformy UWP, pracę należy rozpocząć od zainstalowania wtyczki `Color`. Po jej zainstalowaniu należy utworzyć kilka dodatkowych klas (odpowiedzialnych m.in. za wczytanie zainstalowanej wtyczki), które z autmatu zostają utworzone w projektach dla Androida i iOS po zainstalowaniu pluginu.

Najpierw należy aktywować wtyczkę zainstalowaną przed chwilą w projekcie. W tym celu należy przeciążyć metodę `LoadPlugins` z klasy `Setup`. Argumentem tej metody jest obiekt typu `IMvxPluginManager`, na rzecz którego należy wywołać metodę `EnsurePluginLoaded`, przekazując jako parametr ogólny typ wtyczki, którą ładujemy do projektu. Należy również wywołać metodę `EnsurePluginLoaded` z klasy bazowej, bez której wtyczka nie zostanie załadowana do projektu. Zaktualizowany kod klasy `Setup` jest widoczny na listingu 20.

Listing 20 Zawartość klasy Setup po dodaniu metody ładującej wtyczkę do projektu

```
public class Setup : MvxWindowsSetup
{
      public Setup(Frame rootFrame) : base(rootFrame)
      {
      }

      protected override IMvxApplication CreateApp() => new Core.App();

      public override void LoadPlugins(IMvxPluginManager pluginManager)
      {
             pluginManager
            .EnsurePluginLoaded<MvvmCross.Plugins.Color.PluginLoader>();
            base.LoadPlugins(pluginManager);
      }

      protected override MvxLogProviderType GetDefaultLogProviderType()
      => MvxLogProviderType.None;
}
```

W następnym kroku należy utworzyć dodatkowe opakowanie (ang. *wrapper*), które będzie służyć do załadowania konwerterów zamieniających obiekt koloru typu `MvxColor` (własność `CurrentColor`) udostępniany przez model widoku na obiekt pędzla typu

SolidBrushColor, który jest używany w platformie UWP. W tym celu w projekcie należy utworzyć dodatkowy folder o nazwie *NativeConverters*, w którym należy utworzyć nową klasę o nazwie NativeColorConverter. Klasa ta, musi dziedziczyć po klasie MvxNativeValueConverter z parametrem generycznym przyjmującym nazwę klasy konwertera (MvxNativeColorValueConverter), odpowiedzialnego za odpowiednią konwersję obiektu MvxColor z na SolidBrushColor. Kod klasy został zaprezentowany na listingu 21. Ze względu na to, że plik klasy znajduje się w folderze *NativeConverters,* klasa zdefiniowana jest w przestrzeni Kolory.Uwp.NativeConverters.

Listing 21 Zawartość klasy opakowującej konwerter do zmiany MvxColor na SolidBrush

```
public class NativeColorConverter
         : MvxNativeValueConverter<MvxNativeColorValueConverter>
{

}
```

Następnie należy zarejestrować natywny konwerter w aplikacji. W tym celu w pliku *App.xaml* (listing 22) musimy dodać przestrzeń nazw converters, która będzie odnosić się do przestrzeni nazw, w jakiej znajduje się klasa natywnego konwertera. Następnie tworzymy dodatkowy znacznik XAML Application.Resources i umieszczamy w nim klasę NativeColorConverter, nadając jej unikalny klucz (atrybut x:Key) o nazwie NativeColor. Do tego klucza będziemy odwoływać się w kodzie tworzącym widok tj. pliku MainPage.xaml.

Listing 22 Zawartość pliku App.xaml po zarejestrowaniu w naszej aplikacji konwertera

```
<Application
    x:Class="Kolory.Uwp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Kolory.Uwp"
    xmlns:converters="using:Kolory.Uwp.NativeConverters"
    RequestedTheme="Light">
    <Application.Resources>
        <converters:NativeColorConverter x:Key="NativeColor" />
    </Application.Resources>
</Application>
```

W tym momencie należy zaktualizować także kod opisujący widok aplikacji z pliku *MainPage.xaml*, który będzie korzystał z modelu widoku utworzonego w części wspólnej. W tym celu dodajemy do siatki (pojemnik Grid) kontrolkę typu Rectangle. W niej

wykorzystujemy właściwość `Fill`, która przyjmuje obiekt typu `SolidBrushColor`, który określa kolor, jaki wypełnia kontrolkę. Własność `Fill` wiążemy w trybie jednostronnym tzn. wartości przesyłane są tylko z modelu widoku do widoku. W wiązaniu tym wykorzystujemy konwersje za pomocą omówionego wyżej konwertera, którego instancję umieściliśmy w zasobach statycznych w pliku *App.xaml*. Dzięki temu, zostanie poprawnie obsłużona konwersja z typu `MvxColor` znajdującej się w części wspólnej, na obiekt typu `SolidBrushColor` zdefiniowany w UWP.

Listing 23 Zawartość pliku `MainPage.xaml` po dostosowaniu pod model widokowy z części wspólnej

```
<local:BasePage
      x:Class="Kolory.Uwp.Pages.MainPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:local="using:Kolory.Uwp.Pages"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-
      compatibility/2006"
      mc:Ignorable="d">

      <Grid Background="White">
       <Rectangle x:Name="rectangle" Margin="10,10,10,130"
            Stroke="Black"
            Fill="{Binding CurrentColor,Mode=OneWay,
            Converter={StaticResource NativeColor}}" >
       </Rectangle>
       <Slider x:Name="sliderR"
            Margin="10,0,40,100" Height="22" VerticalAlignment="Bottom"
            Maximum="255" Value="{Binding R, Mode=TwoWay}"/>
       <TextBlock Height="22" VerticalAlignment="Bottom"
            HorizontalAlignment="Right" Margin="10,0,10,100"
            Text="{Binding R, Mode=OneWay}" />
       <Slider x:Name="sliderG"
            Margin="10,0,40,80" Height="22" VerticalAlignment="Bottom"
            Maximum="255" Value="{Binding G, Mode=TwoWay}" />
       <TextBlock Height="22" VerticalAlignment="Bottom"
            HorizontalAlignment="Right" Margin="10,0,10,80"
            Text="{Binding G, Mode=OneWay}" />
       <Slider x:Name="sliderB"
            Margin="10,0,40,60" Height="22" VerticalAlignment="Bottom"
            Maximum="255" Value="{Binding B, Mode=TwoWay}" />
       <TextBlock Height="22" VerticalAlignment="Bottom"
            HorizontalAlignment="Right" Margin="10,0,10,60"
            Text="{Binding B, Mode=OneWay}" />
       <Button Content="Resetuj" Height="30" Width="75"
            VerticalAlignment="Bottom" HorizontalAlignment="Left"
            Margin="10,10,0,10"
            Command="{Binding ResetColorCommand}" />
      </Grid>
</local:BasePage>
```

Rys.25 Utworzony widok za pomocą kodu *xaml* z listingu 23.

Widok budujemy w sposób podobny do tego z projektu dla systemu Android tj. wykorzystujemy trzy zestawy kontrolek typu `TextBlock` oraz `Slider` (suwaki). Pierwsza z nich jest odpowiedzialna za wyświetlanie aktualnie wprowadzonych przez użytkownika aplikacji wartości składowych koloru R, G, B. Własności `Text` każdej z tych kontrolek związane są z własnościami `R`, `G` i `B` zdefiniowanymi w modelu widoku.

Kontrolki typu `Slider` natomiast, będą wykorzystywane do zmiany składowych koloru. W opisujących je elementach wprowadziliśmy ograniczenie na dopuszczane wartości równe 255. Odpowiada za to atrybut `Maximum`.

Ostatnią kontrolką jest `Button`, która po naciśnięciu wywołuje zdefiniowane w modelu widoku polecenie `ResetColorCommand`. Po jego wywołaniu aplikacja przywróci zerowe wartości składowych koloru.

W tym momencie należy skompilować projekt i sprawdzić jego działanie na urządzeniu z systemem Windows 10.

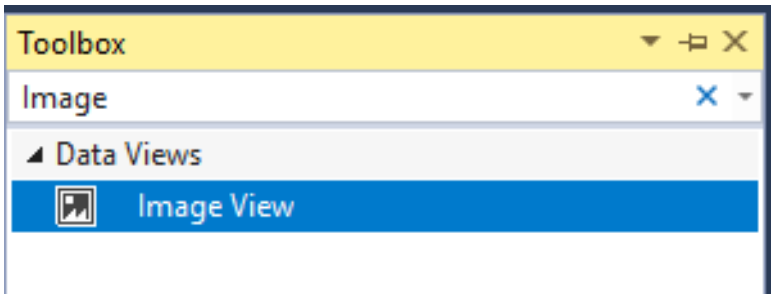
4.1.4 Aplikacja Kolory – projekt iOS

W przypadku projektu przeznaczonego na urządzenia z systemem iOS, po zainstalowaniu wtyczki `MvvmCross.Plugin.Color`, konfiguracja projektu sprowadza się do utworzenia widoku oraz zadeklarowania odpowiednich wiązań między modelem widoku a widokiem. Podobnie, jak w projekcie *Xamarin.Android*, po zainstalowaniu wtyczki został utworzony folder *Bootstrap*, a w nim klasa `ColorPluginBootstrap`, która jest odpowiedzialna za wczytanie komponentów dostarczanych przez zainstalowany plugin. Jedyna różnica polega na tym, że w przypadku tego projektu klasa wtyczki musi dziedziczyć po klasie `MvxLoaderPluginBootstrapAction`. Zamiast jednego typu generycznego, przyjmuje ona dwa: jest to typ, który odpowiada za ładowanie wtyczek oraz sama wtyczka. Kod klasy został przedstawiony na listingu nr. 24.

Listing 24 Zawartość klasy `ColorPluginBootstrap`

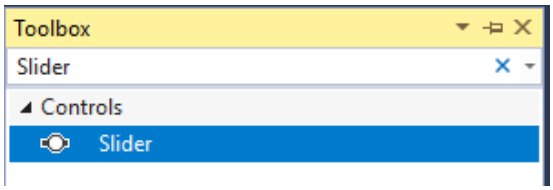
```
public class ColorPluginBootstrap
: MvxLoaderPluginBootstrapAction<MvvmCross.Plugins.Color.PluginLoader,
      MvvmCross.Plugins.Color.iOS.Plugin>
{

}
```

Tak, jak w dwóch poprzednich projektach, także teraz naszym głównym zadaniem będzie zbudowanie widoku. W tym celu w pliku `MainView.xib` należy dodać kontrolkę typu `Image View`, którą można dodać z zakładki *Toolbox*, klikając na nią lewym przyciskiem myszy i przeciągając ją na ekran aplikacji.



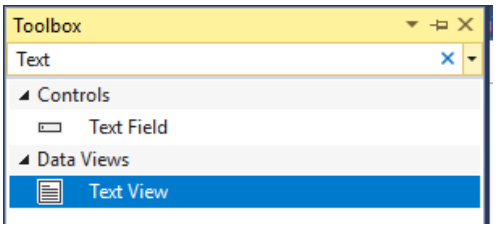
Rys.26 Dodawanie kontrolki odpowiedzialnej za wyświetlanie koloru powstałego z barw RGB

`Image View` posłuży do prezentacji koloru, który użytkownik będzie ustalał za pomocą trzech kontrolek typu `Slider`, kontrolujących składowe koloru.



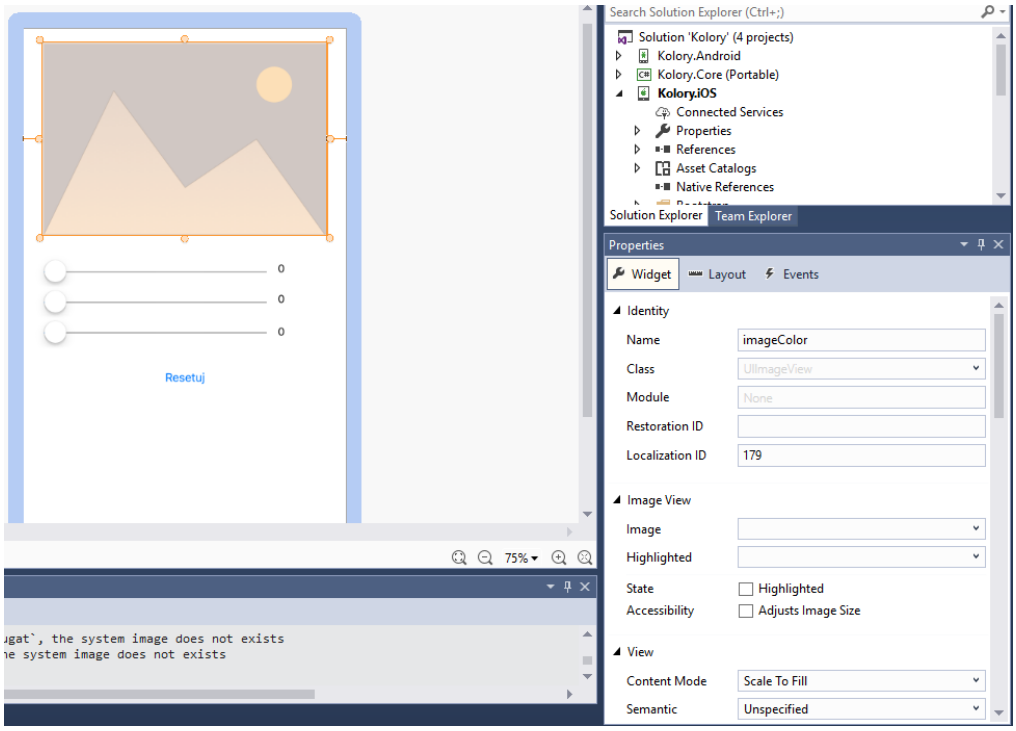
Rys.27 Dodawanie kontrolki odpowiedzialnej za wyświetlanie koloru powstałego z barw RGB

Przy każdej kontrolce `Slider`, po jej prawej stronie, umieszczamy kontrolkę typu `Text View`, która będzie odpowiedzialna za wyświetlanie wartości z suwaków.



Rys.28 Dodawanie kontrolki odpowiedzialnej za wyświetlanie koloru powstałego z barw RGB

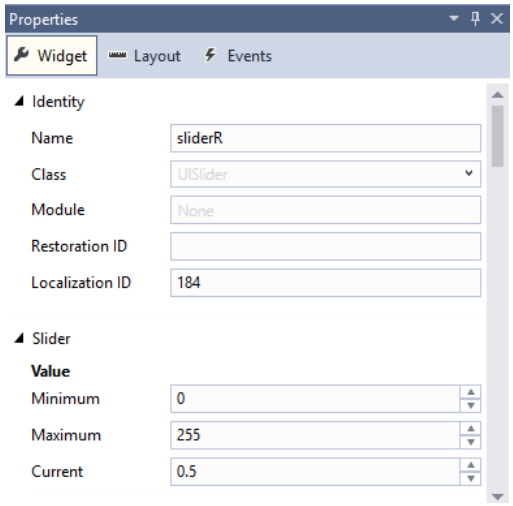
Każdej kontrolce musimy nadać unikalną nazwę, do której będziemy się odwoływać podczas konfiguracji wiązania z własnościami modelu widoku. Aby nadać kontrolkom nazwy, należy posłużyć się oknem własności (rysunek 29), które można przywołać klawiszem *F4*.



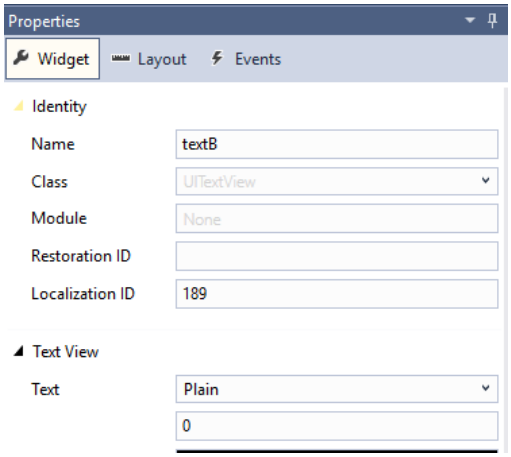
Rys.29 Dodawanie kontrolki odpowiedzialnej za wyświetlanie koloru powstałego z barw RGB.

Postępując w ten sposób nadajemy nazwy wszystkim kontrolkom według następującego schematu: suwaki kolejno od góry powinny mieć nazwy: `sliderR`, `sliderG` i `sliderB`. Pola tekstowe kolejno: `textR`, `textG` i `textB`. Przycisk – `buttonReset`.

Dodatkowo w przypadku kontrolek `Slider`, ustawiamy ich maksymalną wartość na 255 (rysunek 30), a dla wszystkich pól tekstowych ustawiamy domyślną wartość tekstową na 0 (rysunek 31).



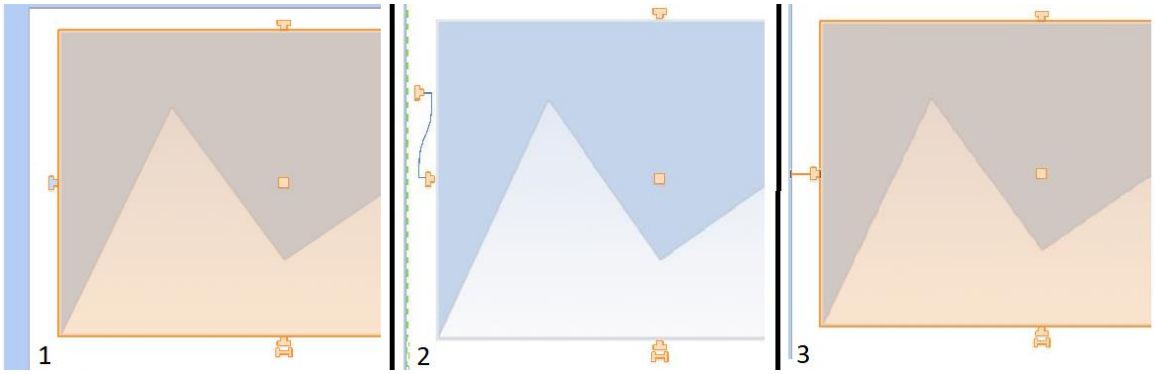
Rys.30 Dodawanie kontrolki odpowiedzialnej za wyświetlanie koloru powstałego z barw RGB



Rys.31 Dodawanie kontrolki odpowiedzialnej za wyświetlanie koloru powstałego z barw RGB

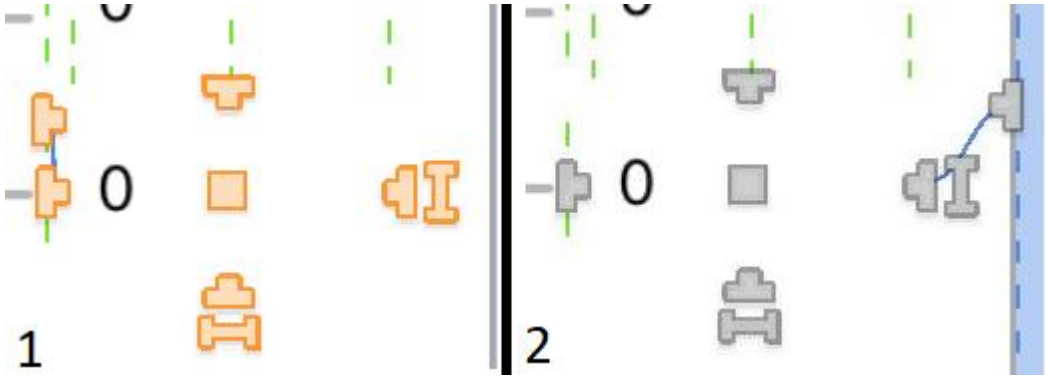
Po skonfigurowaniu widoku, należy dodać do kontrolek tzw. zakotwiczenie (ang. *constraints*), dzięki którym nasze kontrolki będą automatycznie dopasowywały się do rozmiaru ekranu niezależnie od jego wielkości. W tym celu klikamy dwa razy na interesującą nas kontrolkę (niech to będzie np. `Image View`). Następnie wybieramy róg

kontrolki który chcemy „zakotwiczyć” (w przypadku `Image View` będzie to prawy i lewy róg), klikamy go i przeciągamy na pole na ekranie, które zostało oznaczone zielonym kolorem. Po najechaniu na zielone pole, jego kolor zmieni się na niebieski. W tym momencie należy zwolnić przycisk myszy, a wówczas kontrolka zakotwiczy się w obszarze (będzie widoczna pomarańczowa linia pomiędzy kontrolką a odpowiednim obszarem ekranu). Przedstawia to rysunek 32.



Rys.32 Nadawanie `constraints` kontrolce typu `Image View`

Analogicznie, jak w przypadku kontrolki `Image View`, zakotwiczmy kontrolki `Slider` oraz `Button`. W ich przypadku dodajmy zakotwiczenie do lewej oraz prawej krawędzi okna aplikacji. Inaczej natomiast ustawiamy pola tekstowe, w których przypadku zakotwiczenie ustawiamy do prawej strony do ekranu, a z lewej do kontrolek prawych rogów naszych suwaków. Miejsca, w których należy ustawić zakotwiczenie dla pól tekstowych zostały przedstawione na rysunku 33.



Rys. 33 Dodawanie „constraints” podczas tworzenia skalowalnych pól tekstowych

Przejdźmy teraz do zdefiniowania wiązań między własnościami w modelu widoku z części wspólnej a kontrolkami w widoku w części dla iOS. W klasie MainView należy

przeciążyć metodę bazową `ViewDidLoad`, wiążąc widok `MainView` z modelem widoku `MainViewModel`. Następnie kontrolki suwaków wiążemy z własnościami `R`, `G` i `B` z modelu widoku. Analogicznie wiążemy kontrolki typu `Text View`.

W przypadku kontrolki `Image View` wykorzystujemy wiązanie do specyficznego miejsca (ang. *for*). W tym przypadku wskazujemy, że własność zdefiniowana w modelu widoku będzie udostępniać wartość dla atrybutu opisującego kolor tła korzystając z natywnego konwertera `NativeColor`, zamieniającego obiekt typu `MvxColor` na `UIKit.UIColor`. Ustawiamy także wiązanie przycisku z poleceniem `ResetColorCommand`. Kod zmodyfikowanej klasy `MainView`, został pokazany na listingu 25.

Listing 25 Zawartość klasy `MainView`

```
[MvxRootPresentation(WrapInNavigationController = true)]
public partial class MainView : MvxViewController
{
      public MainView() : base("MainView", null)
      {
      }
      public override void ViewDidLoad()
      {
            base.ViewDidLoad();
            var set = this.CreateBindingSet<MainView,
            Core.ViewModels.MainViewModel>();
            set.Bind(sliderR).To(vm => vm.R);
            set.Bind(sliderG).To(vm => vm.G);
            set.Bind(sliderB).To(vm => vm.B);
            set.Bind(textR).To(vm => vm.R);
            set.Bind(textG).To(vm => vm.G);
            set.Bind(textB).To(vm => vm.B);
            set.Bind(imageColor)
                  .For(field => field.BackgroundColor)
                  .To(vm => vm.CurrentColor)
                  .WithConversion("NativeColor");
            set.Bind(buttonReset).To(vm => vm.ResetColorCommand);
            set.Apply();
      }
}
```

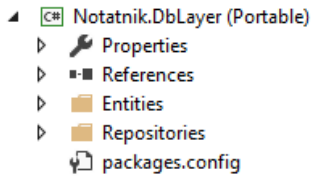
Warto skompilować i uruchomić aplikację na urządzeniu testowym lub emulatorze iOS.

4.2 Aplikacja Notatnik

W tym podrozdziale zaprezentuję sposób, w jaki można uzyskać dostęp do bazy danych typu *SQLite*, wykorzystując do tego wzorzec *repozytorium* w aplikacjach Xamarin. Zostanie przedstawiony również sposób, w jaki należy obsługiwać wyświetlanie list na poszczególnych platformach oraz w jaki należy zorganizować nawigację i przesyłanie informacji pomiędzy poszczególnymi modelami widoku. Pretekstem do tego będzie projekt *Notatnika*, czyli aplikacji pozwalającej na edycję niesformatowanych notatek w urządzeniach mobilnych. Notatki będą przechowywane w bazie danych.

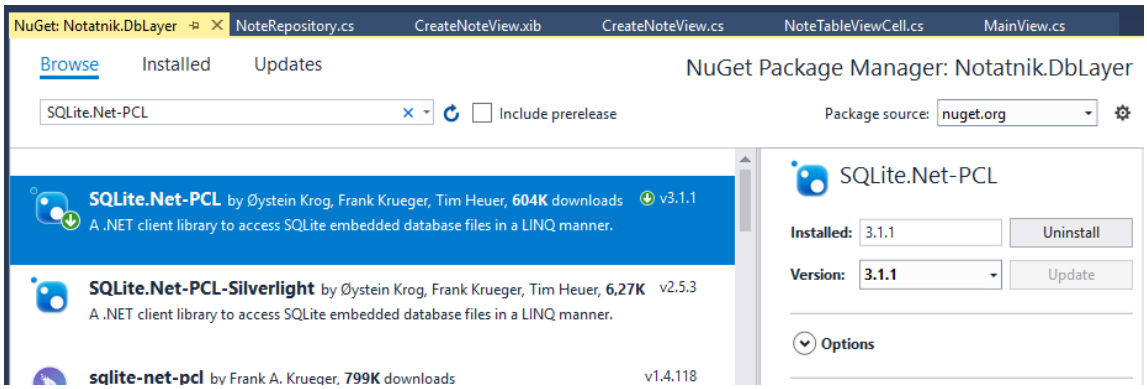
4.2.1 Aplikacja Notatnik – projekt bazy danych

Tworzymy nowe rozwiązanie typu *Blank Solution* o nazwie *Notatnik*. Następnie dodajemy do niego projekt biblioteki typu *PCL* o nazwie *Notatnik.DBLayer*, czyli takiego samego, jak w przypadku części wspólnej projektu. Po utworzeniu nowego projektu, dodajemy do niego dwa nowe foldery: *Entities* oraz *Repositories*. W pierwszym będziemy umieszczać klasy modeli, które opisują obiekty, jakie będą trzymane w naszej bazie danych (klasy encji), natomiast w drugim umieszczone będą tzw. repozytoria, które dostarczają metod służących do obsługi zapytań na bazie danych i przetrzymywanych tam encjach.



Rys. 34 Struktura projektu bazodanowego

Następnie przechodzimy do menadżera pakietów *NuGet* i wyszukujemy bibliotekę o nazwie *SQLite-Net-PCL* (rys. 35), która jest otwartą i „lekką" biblioteką, którą można używać w środowiskach *Mono* i *.NET*. Służy do zapisu i uzyskiwania dostępu do danych w bazach *SQLite*. [28]



Rys. 35 Biblioteka odpowiedzialna za dostęp do bazy SQLite

Co ważne, w każdym z projektów, w którym będziemy chcieli korzystać z dostępu do tej bazy danych, będziemy musieli zainstalować pakiet *SQLite.Net-PCL*. Po zainstalowaniu pakietu, dodajmy do folderu *Entities* pierwszą klasę o nazwie `NoteEntity`. Jej kod został przestawiony na listingu 26.

Listing 26 Zawartość klasy `NoteEntity`

```
public class NoteEntity
{
    public NoteEntity()
    {

    }
    public NoteEntity(string title, string content, DateTime date)
    {
        Title = title;
        Content = content;
        Date = date;
    }

    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }

    [MaxLength(15)]
    public string Title { get; set; }

    public string Content { get; set; }

    public DateTime Date { get; set; }
}
```

Klasa ta zawiera dwa konstruktory, jeden pusty, który jest wymagany przez bibliotekę *SQLite* do poprawnej obsługi modeli bazodanowych, drugi natomiast przyjmuje trzy wartości, która są przetrzymywane przez tabele w bazie danych, czyli tytuł, tekst notatki oraz datę ostatniej modyfikacji [28]. Dodatkowo zdefiniowane jest także pole typu `int(Id)`, które jest unikalnym identyfikatorem każdego obiektu przechowywanego w

bazie danych. Definiując je wykorzystaliśmy atrybuty `PrimmaryKey` oraz `AutoIncrement`, które informują bibliotekę pośredniczącą w połączeniu z bazą danych SQLite, że jest to właściwość identyfikująca dany rekord w tabeli.

Dodatkowo tworząc kod programu nie musimy martwić się o nadawanie unikalnych wartości tej własności, ponieważ baza danych sama je ustala podczas tworzenia nowych obiektów[6]. W przypadku własności `Title` wykorzystaliśmy natomiast atrybut `MaxLength(15)`, który nie pozwala na dodanie do bazy danych rekordu z napisu dłuższym niż 15 liter. [28]

Do folderu *Repositories* dodajmy klasę o nazwie `NoteRepository` (listing 27). Należy w niej zdefiniować metody odpowiedzialne za komunikację z bazą danych.

Listing 27 Zawartość klasy `NoteRepository`

```
public class NoteRepository : INoteRepository
{
    private static object _databaseLock = new object();

        private SQLiteConnection _dbConnection { get; set; }

        public NoteRepository(SQLiteConnection dbConnection)
        {
            _dbConnection = dbConnection;
        }

        public Task Add(NoteEntity note)
        {
            return Task.Run(() =>
            {
                lock (_databaseLock)
                {
                    _dbConnection.Insert(note);
                }
            });
        }

        public Task Update(NoteEntity note)
        {
            return Task.Run(() =>
            {
                lock (_databaseLock)
                {
                    _dbConnection.Update(note);
                }
            });
        }

        public Task<List<NoteEntity>> GetAll()
```

[6] Automatyczne ustalanie identyfikatora (tzw. autoincrement) dostępne jest w większości baz danych, oprócz SQLite także w Oracle, MySQL czy MSQL. Dzięki temu w naszym przypadku nie trzeba „na sztywno" wprowadzać wartości klucza głównego – baza danych sama ją ustali podczas dodawania nowej encji.

```
        {
            return Task.Run(() =>
            {
                lock (_databaseLock)
                {
                    return _dbConnection.Table<NoteEntity>().ToList();
                }
            });
        }
    }

    public interface INoteRepository
    {
        Task Add(NoteEntity note);

        Task Update(NoteEntity note);

        Task<List<NoteEntity>> GetAll();
    }
```

W klasie `NoteRepository` zdefiniowane są dwa prywatne pola. Pierwsze typu *object*, będziemy wykorzystywać do zablokowania równoczesnego dostępu do bazy, co uzyskamy korzystając z instrukcji `lock`. Dzięki temu, jeśli postanowimy, aby aplikacja działała wielowątkowo, unikniemy problemów z ewentualnych zakliszczeniem wątków lub z niekontrolowanymi zmianami wynikającymi z konkurencji wielu wątków w dostępie do warstwy bazodanowej.

Drugim prywatnym polem jest `dbConnection` typu `SQLiteConnection`, który jest specyficzny dla każdej platformy, na której będziemy używać biblioteki. W późniejszym etapie będziemy wykorzystywać mechanizm wstrzykiwania zależności, aby dostarczać odpowiedniej wersji tego obiektu. Bez nie niego nie możemy wykonywać operacji na bazie danych (charakterystyczną cechą dla każdej platformy jest m.in. ścieżka systemowa, w której możemy umieszczać pliki związane z bazą danych).

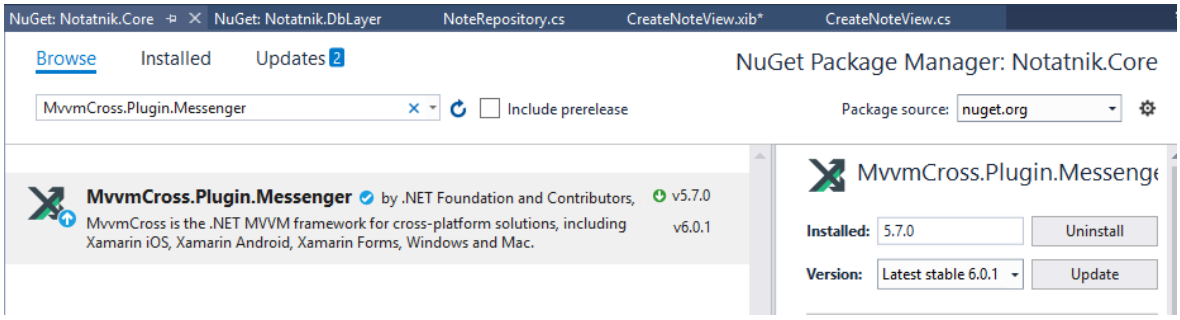
W klasie zdefiniowane są również trzy metody o nazwach `Add`, `Update` oraz `GetAll`. Pierwsza jest odpowiedzialna za tworzenie nowego obiektu i dodanie go do bazy danych, druga odpowiada za aktualizacje obiektów, a ostatnia zwraca wszystkie obiekty, jakie aktualnie znajdują się w bazie. Każda z metod zwraca zadanie (obiekt typu `Task`), które uruchamia operacje na bazie danych.

Oznacza to że w aplikacji możliwe będzie asynchroniczne wykonywanie operacji na bazie danych. Dzięki temu, aplikacja nie powinna sprawiać wrażenia „zastygania” podczas zapisu czy odczytu danych z bazy danych, ponieważ wykonywane wówczas operacje będą wykonywane w osobnych wątkach.

Klasa `NoteRepository` implementuje interfejs `INoteRepository`. Zastosowanie interfejsu ułatwia dalszy rozwój aplikacji umożliwiając np. nową implementacje klasy odpowiedzialnej za dostęp do bazy danych. Jedyne co będziemy musieli wówczas zrobić to dodanie nowej klasy implementującej interfejs, a następnie zarejestrowanie jej w kontenerze odpowiedzialnym za wstrzykiwanie zależności w naszym projekcie.

4.2.2 Aplikacja Notatnik – projekt części wspólnej

Dodajmy kolejny projekt PCL do części wspólnej. Tworzenie projektu rozpoczynamy od dodania paczki *NuGet* o nazwie *MvvmCross.StarterPack* oraz dodatkowo paczki *SQLite-Net-PCL*. Kolejną paczką, jaką wykorzystamy w projekcie będzie *MvvmCross.Plugin.Messegner*, który dostarcza mechanizmów znanych z wzorca Obserwator. Używając tej wtyczki, można subskrybować (nasłuchiwać) wiadomości oraz przekazywać wiadomości pomiędzy aktywnymi modelami widoku aplikacji.



Rys. 36 Biblioteka odpowiedzialna za rozgłaszanie wiadomości między modelami widoku

Następnie przechodzimy do menadżera referencji i dodajemy odwołanie do projektu `Notatnik.DbLayer`, tak żeby móc korzystać z zaimplementowanego już repozytorium.

Tworzymy folder *Models*, w którym tworzymy klasę o nazwie `Note`. Będzie ona odpowiedzialna za odpowiednie wyświetlanie danych w widoku. Klasa ta będzie dziedziczyć z klasy `NoteEntity` rozszerzając je o nowe pole o nazwie `CreateDate` (listing 28).

Listing 28 Zawartość klasy `Note`

```
public class Note :NoteEntity
{
	public Note(NoteEntity noteEntity)
		: base(noteEntity.Title, noteEntity.Content, noteEntity.Date)
	{
		Id = noteEntity.Id;
	}

	public string CreateDate { get => "Ostatnia modyfikacja: " +
	Date.ToString("dd.MM.yyyy HH:mm:ss"); }
}
```

Należy również dodać folder o nazwie *Messages*, w którym tworzymy klasę o nazwie `CreateNoteMessage`. Zawiera ona dwa publiczne pola, jedno typu `bool`, a drugie typu `NoteEntity` (listing 29).Pierwsze pole wykorzystamy do określenia stanu obiektu, jaki będzie przekazywany pomiędzy modelami widoku. Określa ono, czy obiekt należy dodać jako nowy do bazy danych, czy tylko zaktualizować wartości już istniejącego. Drugie pole to oczywiście sam obiekt, który jest aktualnie wyświetlany w widoku aplikacji i może być modyfikowany przez użytkownika.

Listing 29 Zawartość klasy `CreateNoteMessage`

```
public class CreateNoteMessage : MvxMessage
{
	public bool IsEditMode { get; set; }

	public NoteEntity Note { get; set; }

	public CreateNoteMessage(object sender, NoteEntity note) :
	base(sender)
	{
		Note = note;
	}
}
```

Następnie przechodzimy do dodania modeli widoku. Pierwszy model widoku to klasa `MainViewModel`, której kod jest widoczny na listingu 30.

Listing 30 Zawartość klasy `MainViewModel`

```
public class MainViewModel : MvxViewModel
{
	private readonly IMvxMessenger _messanger;
	private readonly INoteRepository _noteRepository;
	private readonly IMvxNavigationService _mvxNavigationService;
```

```
private ObservableCollection<Note> notes;

public ObservableCollection<Note> Notes
{
      get { return notes; }
      set
      {
          SetProperty(ref notes, value);
          RaisePropertyChanged(() => this.Notes);
      }
}

public MainViewModel(IMvxMessenger messanger,INoteRepository
noteRepository, IMvxNavigationService mvxNavigationService)
{
      _messanger = messanger;
      _noteRepository = noteRepository;
      _mvxNavigationService = mvxNavigationService;

      _messanger.Subscribe<CreateNoteMessage>(async x =>
      {
            if(x.IsEditMode)
                  await _noteRepository.Update(x.Note);
            else
                  await _noteRepository.Add(x.Note);

      }, MvxReference.Strong);
}

public IMvxCommand CreateNoteCommand => new MvxCommand(() => {
_mvxNavigationService.Navigate<CreateNoteViewModel>(); });

private MvxCommand<Note> _itemClickedCommand;
public IMvxCommand ItemClickedCommand
{
      get
      {
            return _itemClickedCommand = _itemClickedCommand ??
            new MvxCommand<Note>(item =>
            {
                  if(item!=null)
                  _mvxNavigationService.Navigate<CreateNoteViewMode
                  l, Note>(item);
            });
      }
}

public IMvxCommand HoldingNoteCommand => new MvxCommand(() => {
_mvxNavigationService.Navigate<CreateNoteViewModel>(); });

public async void OnResume()
{

      try
      {
            Notes = new ObservableCollection<Note>();

            var dbNotes = await _noteRepository.GetAll();

            foreach (var noteEntity in dbNotes)
            {
```

```
                        Notes.Add(new Note(noteEntity));
                    }
                }
                catch (Exception ex)
                {
                    System.Diagnostics.Debug.WriteLine(ex.Message);
                }
        }
}
```

W klasie `MainViewModel` zadeklarowane są cztery prywatne pole. Są to kolejno: 1) `IMvxMessenger`, za pomocą którego możemy nasłuchiwać i rozgłaszać wiadomości w obrębie naszych modeli widoku, 2) `INoteRepository`, do którego zostanie przypisana implementacja klasy `NoteRepository`, za pomocą której uzyskamy dostęp do danych z bazy danych, 3) `IMvxNavigationService`, czyli klasa, która zapewni możliwość nawigacji i przekazywania parametrów do aktualnie nawigowanego modelu widokowego [29]. Oraz 4) lista typu `ObservableCollection`, która będzie służyć do wyświetlania elementów pobranych z bazy danych, opakowanych w obiekt `Note`, który zawiera dodatkowe informacje przekazywane tylko do warstwy widoku.

Następnym elementem klasy jest konstruktor, który przyjmuje trzy argumenty typów `IMvxMessenger`, `INoteRepository` oraz `IMvxNavigationService`. Obiekty te będą dostarczane za pomocą kontenera *IoC*. Dodatkowo konstruktor modelu widoku subskrybuje wiadomości typu `CreateNoteMessage` i w zależności od przekazanej w nich wartości zmiennej logicznej `IsEditMode`, aplikacja będzie dodawać lub aktualizować istniejące obiekty z bazy danych.

Publiczne pola związane są z nawigacją i przekazywaniem parametrów do innego modelu widoku, który za chwilę utworzymy, czyli `CreateNoteViewModel`. W przypadku `CreateNoteCommand` wywołujemy nasz serwis tylko z jednym parametrem generycznym.

W tym momencie model widoku nie będzie próbował odbierać żadnych parametrów. Inaczej natomiast zachowa się w razie przekazania dwóch typów oraz dodania obiektu do ciała metody, tak jak ma to miejsce w przypadku komendy `ItemClickedCommand`. Pierwszym typem generycznym jest typ określający model widoku, do którego nawigujemy w aplikacji, a kolejny to typ przekazywanego obiektu. [29]

W klasie `MainViewModel` zdefiniowana została także asynchroniczna metoda o nazwie `OnResume`. Jest ona odpowiedzialna za pobieranie elementów znajdujących się w

bazie danych w sposób asynchroniczny. Całość została zawarta w blokach `try..catch`. Dzięki temu problemy z połączeniem z bazą danych zostaną zarejestrowane w aplikacji, a dodatkowo nie będzie to prowadzić do przerwania jej działania.

Musimy jeszcze dodać wspomniany model widoku o nazwie `CreateNoteViewModel`. Będzie on odpowiedzialny za tworzenie notatek w aplikacji oraz ich edycję. Kod klasy został umieszczony na listingu 31.

Listing 31 Zawartość klasy `CreateNoteViewModel`

```
public class CreateNoteViewModel : MvxViewModel<Note>
{
      private readonly IMvxMessenger _messanger;
      private Note _note;
      private bool _isEditMode = false;

      private string _title;
      public string Title
      {
            get { return _title; }
            set
            {
                _title = value;
                RaisePropertyChanged("Title");
            }
      }

      private string _content;
      public string Content
      {
            get { return _content; }
            set
            {
                _content = value;
                RaisePropertyChanged("Content");
            }
      }

      public CreateNoteViewModel(IMvxMessenger messanger)
      {
            _messanger = messanger;
      }

      public IMvxCommand CreateNoteCommand => new MvxCommand(() =>
      CreateNote());

      private void CreateNote()
      {
            NoteEntity noteEntity = new NoteEntity(Title, Content,
            DateTime.UtcNow);
            if (_isEditMode) noteEntity.Id = _note.Id;
            var noteMessage = new CreateNoteMessage(this, noteEntity);
            noteMessage.IsEditMode = _isEditMode;
            _messanger.Publish(noteMessage);
            Close(this);
      }
```

```
        public override void Prepare(Note parameter)
        {
                _note = parameter;
                _isEditMode = true;
                Title = parameter.Title;
                Content = parameter.Content;
        }
}
```

Zdefiniowane zostały w niej trzy prywatne pola. Po raz kolejny użyty został obiekt typu `IMvxMessenger`, który posłuży do wysyłania wiadomości zawierającej aktualnie tworzoną notatkę, obiekt `Note`, który może być przekazany do tego modelu widoku podczas nawigacji z `MainViewModel` oraz zmienna typu `bool` informująca o tym, czy utworzona została nowa notatka, czy edytowana jest notatka już istniejącą.

W klasie zdefiniowane są również własności odpowiedzialne za edycję i wyświetlanie w widoku tytułu oraz zawartości notatki użytkownikowi,. Konstruktor natomiast przyjmuje tylko obiekt implementujący interfejs `IMvxMessenger` odpowiedzialny za rozgłaszanie wiadomości.

Następnie zdefiniowane jest polecenie `CreateNoteCommand`, które odpowiada za zapis zmian w aktualnie tworzonej lub edytowanej notatce. Podczas jego wywołania, wykonywana jest metoda `CreateNote`. Tworzy ona nowy obiekt, który zostanie umieszczony lub zaktualizowany w bazie danych w zależności od tego, czy przypisane zostało mu ID. W metodzie `CreateNode` tworzymy obiekt wiadomości (ang. *message*), który zostanie odebrany przez model widoku `MainViewModel`. Jeżeli to się powiedzie, zostanie wywołana metoda `Close`, która spowoduje że aplikacja powróci do poprzedniego widoku.

W klasie `CreateNoteViewModel` nadpisana została również metoda `Prepare` pochodząca z klasy bazowej `MvxViewModel`, która przyjmuje jako parametr generyczny typ `Note`. Dzięki temu model widoku wie, że może spodziewać się przekazania właśnie tego typu argumentu podczas nawigacji.[29]

Jednocześnie potrafi również obsłużyć sytuację, w której ten obiekt nie zostanie przekazany – w tym momencie nie zostanie wywołana metoda `Prepare`. W tej metodzie zostają przypisane wartości z obiektu, który już istnieje, jednocześnie informując model widoku, że aktualnie tworzona notatka, tak naprawdę już istnieje w bazie danych i nie ma potrzeby tworzenia jej na nowo.

Po zdefiniowaniu klas modeli widoku, należy utworzyć folder o nazwie *Services* i umieścić w nim plik z definicją interfejsu o nazwie `IDatabaseService`. Wymusza on implementacje jednej publicznej własności tylko do odczytu typu `SQliteConnection`, która będzie dostarczać mechanizm służący do prawidłowego połączenia z bazą daną. Na każdej z trzech platform posiada on inną implementacje, które zostaną opisane w dalszej części tego rozdziału.

Listing 32 Zawartość interfejsu `IDatabaseService`

```
public interface IDatabaseService
{
      SQLiteConnection Connection { get; }
}
```

Należy również zmodyfikować kod klasy `App`, w której należy dodać instrukcje rejestrujące w kontenerze *IoC* obiekty implementujące interfejs `INoteRepository`. Dodatkowo należy dokonać tzw. odwzorowania (ang. *resolve*), czyli pozyskać implementację interfejsu `IDatabaseService`, zarejestrowanego na jednej z trzech platform i przekazać wymagany obiekt `SQLiteConnection` do naszego repozytorium bazodanowego. Dzięki temu, aplikacja będzie w stanie poprawnie nawiązać połączenie z bazą danych. Całość została pokazana na listingu 33.

Listing 33 Implementacja klasy `App`

```
public class App : MvvmCross.Core.ViewModels.MvxApplication
{
      public override void Initialize()
      {
            CreatableTypes()
                .EndingWith("Service")
                .AsInterfaces()
                .RegisterAsLazySingleton();

            RegisterNavigationServiceAppStart<MainViewModel>();
            DbConfiguration();
      }

      private void DbConfiguration()
      {
            var databaseService = Mvx.Resolve<IDatabaseService>();
            databaseService.Connection.CreateTable<NoteEntity>();
            Mvx.RegisterType<INoteRepository>(() => new
            NoteRepository(databaseService.Connection));
      }
}
```

4.2.3 Aplikacja Notatnik – projekt Android

Przygotowanie projektu przeznaczonego dla systemu Android rozpoczynamy od zainstalowania takich samych pakietów *NuGet*, jak w przypadku projektu części wspólnej. Następnie do nowego projektu dodajemy referencję do projektu części wspólnej. Po przygotowaniu projektu (zob. opis w rozdziale trzecim) przechodzimy do zaprojektowania widoków.

Zaczynamy od edycji istniejącego widok z pliku *MainView.axml*, zmieniając jego nazwę na *main_view.axml* oraz dodając kod pokazany na listingu 34.

Listing 34 Zawartość pliku `main_view.axml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Dodaj Notatkę"
        android:layout_margin="10dp"
        local:MvxBind="Click CreateNoteCommand" />
    <MvxListView
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        local:MvxBind="ItemsSource Notes; ItemClick ItemClickedCommand;"
        local:MvxItemTemplate="@layout/note_layout" />
</LinearLayout>
```

Rys.37 Utworzony widok za pomocą kodu *axml* z listingu 34.

Na górze ekranu został umieszczony przycisk, który będzie odpowiedzialny za przełączenie do widoku, w którym będzie można utworzyć nową notatkę. Przycisk ten został związany z poleceniem `CreateNoteCommand`. Druga kontrolka to `MvxListView`, którą użyjemy do wyświetlania elementów pobranych z bazy danych. Dodatkowo obsłużone zostało zdarzenie kliknięcia (tapnięcia) poszczególnych elementów listy, poprzez ustalenie wiązania typu `ItemClick` z poleceniem `ItemClickedCommand`. Ostatnim atrybutem w liście jest `MvxItemTemplate`, który przyjmuje nazwę pliku definiującego wygląd pojedynczego elementu na liście. Należy tam wpisać nazwę pliku bez rozszerzenia, który powinien mieć rozszerzenie *xml* np. *note_layout*. Następnie należy utworzyć taki plik i uzupełnić go kodem widocznym na listingu 35.

Listing 35 Zawartość pliku `note_layout.xml`

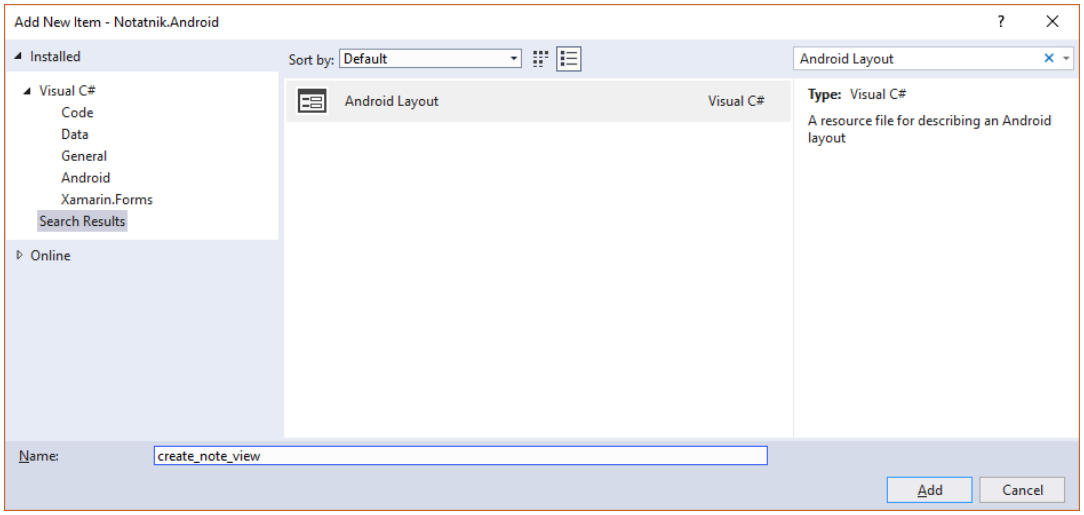
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="20dp"
    android:gravity="center">
```

```
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="left"
    android:textSize="10dp"
    local:MvxBind="Text Title" />
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="right"
    android:textSize="10dp"
    local:MvxBind="Text CreateDate" />
</LinearLayout>
```

Rys.38 Utworzony widok za pomocą kodu *axml* z listingu 35.

W pliku *note_layout.xml* definiującym wygląd pojedynczego elementu listy znajdują się dfwa elementy opisujące dwie kontrolki tekstowe, które będą wyświetlać treść notatki oraz datę ostatniej jej modyfikacji. Wartości będą pobierane z pojedynczego elementu listy `Notes` znajdującej się w modelu widoku. Kontrolki tekstowe z pojedynczego elementu listy zostaną związane z własnościami `Title` oraz `CreateDate` klasy `Note`.

Ostatnim widokiem, jaki należy stworzyć jest *create_note_view.axml*, który będzie obsługiwał tworzenie i edycję notatek. W tym celu, należy dodać w folderze layout nowy plik typu *Android Layout*(rysunek 39).



Rys. 39 Tworzenie nowego widoku w projekcie na Androida.

W tym widoku zostały umieszczone dwie kontrolki typu `EditText`. Użytkownicy mogą ich użyć, aby wprowadzić tytuł i treść notatki tj. własności `Title` oraz `Content` klasy `CreateNoteViewModel`. Pod nimi został umieszczony przycisk, po którego wciśnięciu notatka zostanie zapisana w bazie danych, a sama aplikacja wróci do wcześniej wyświetlanego widoku. Przycisk jest związany z poleceniem `CreateNoteCommand`. Kod tego widoku został przedstawiony na listingu 36.

Listing 36 Zawartość pliku `create_note_view.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <EditText
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:textSize="40dp"
      local:MvxBind="Text Title" />
  <EditText
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:textSize="40dp"
      android:scrollHorizontally="false"
      android:inputType="textCapSentences|textMultiLine"
      local:MvxBind="Text Content" />
```

```
  <Button
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Zapisz Notatkę"
      android:textSize="20dp"
      local:MvxBind="Click CreateNoteCommand" />
</LinearLayout>
```

Po utworzeniu widoków dla poszczególnych modeli widoku, należy dodać klasę aktywności, której jedynym zadaniem będzie wskazanie widoku. W tym celu w folderze *Views* należy utworzyć klasę o nazwie `CreateNoteActivity` (listing 37)[7]. Jej kod jest bardzo podobny do tego, który znajduję się domyślnie w klasie `MainView` w tym samym folderze. Jedyna różnica to wskazany widok oraz podpis w rogu nawigacyjnym aplikacji. (atrybut `Activity` z własnością `Label`).

Listing 37 Zawartość klasy `CreateNoteView`

```
[Activity(Label = "Tworzenie Notatki")]
public class CreateNoteView : MvxActivity
{
      protected override void OnCreate(Bundle bundle)
      {
            base.OnCreate(bundle);
            SetContentView(Resource.Layout.create_note_view);
      }
}
```

Dodatkowo należy zmienić kod klasy `MainView`, przeciążając w nim metodę `OnResume`, w której wywołamy metodę `OnResume`, którą zdefiniowaliśmy w modelu widoku (listing 38). Dzięki temu za każdym razem, gdy włączany będzie widok `MainView`, zostaną także załadowane wszystkie elementy z bazy danych. Dodatkowo należy zwrócić uwagę na wywołanie metody `SetContentView`, której argumentem powinna być klasa widoku `main_view` zamiast oryginalnej `MainView`.

[7] Nazwy klas widoku muszą kończyć się na `..View`, a nie `..Activity` jak mogłoby się narzucać programistom dla systemu Android. Dzięki temu framework MvvmCross automatycznie dopasuje model widoku z danym widokiem, odnajdując go właśnie po nazwie klasy z uwzględniem zakończenia `View` i `ViewModel`.

Listing 38 Zawartość klasy `MainView`

```
[Activity(Label = "Notatki")]
public class MainView : MvxActivity
{
      protected override void OnCreate(Bundle bundle)
      {
            base.OnCreate(bundle);
            SetContentView(Resource.Layout.main_view);
      }

      protected override void OnResume()
      {
            base.OnResume();
            (ViewModel as MainViewModel).OnResume();
      }
}
```

Na koniec należy utworzyć klasę implementującą interfejs `IDataBaseService` (zgodnie z zapowiedzią interfejs ten będzie osobno implementowany dla każdej platformy systemowej)oraz odpowiednio zarejestrować jej instancję w klasie `Setup`. Nową klasę umieśćmy w nowym folderzeZacznijmy od utworzenia folderu o nazwie *Services* i umieszczenia w nim klasy o nazwie `AndroidDatabaseService` (listing 39). Będzie ona odpowiedzialna za dostarczanie odpowiedniego połączenia z bazą danych *SQLite* na platformie Android. Kluczowe jest wskazanie właściwej ścieżki do pliku z bazą danych. Ustalając ją wykorzystujemy klasę `Environment`, żeby znaleźć ścieżkę do folderu z plikami użytkownika. Finalnie własność `Connection` udostępnia obiekt typu `SQLiteConnection`, za pomocą którego wykonywać operacje na bazie danych.

Listing 39 Zawartość klasy `AndroidDatabaseService`

```
public class AndroidDatabaseService : IDatabaseService
{
      private SQLiteConnection connection;
      public SQLiteConnection Connection
      {
            get
            {
                  if (connection == null)
                  {
                    var path =
                    Path.Combine(Environment.GetFolderPath(Environment.Sp
                    ecialFolder.MyDocuments), "notesDb.sqlite");
                    connection = new SQLiteConnection(new
                    SQLitePlatformAndroid(), path);
                  }
                  return connection;
            }
        }
    }
```

Ostatnią zmianą w projekcie, jaką należy wykonać jest rejestracja utworzonej przed chwilą klasy `AndroidDatabaseService`. W tym celu w klasie `Setup`, należy przeciążyć metodę `InitializePlatformServices` i korzystając z statycznej klasy `Mvx`, zarejestrować klasę `AndroidDatabaseService` (listing 40). Należy wywołać metodę `RegisterType`, która przyjmuje dwa parametry generyczne: typ interfejsu oraz typ klasy, która ten interfejs implementuje. Dzięki temu w części wspólnej aplikacja będzie w stanie używać skonfigurowanego przez nas połączenia z bazą danych.

Listing 40 Zawartość klasy `Setup.cs`

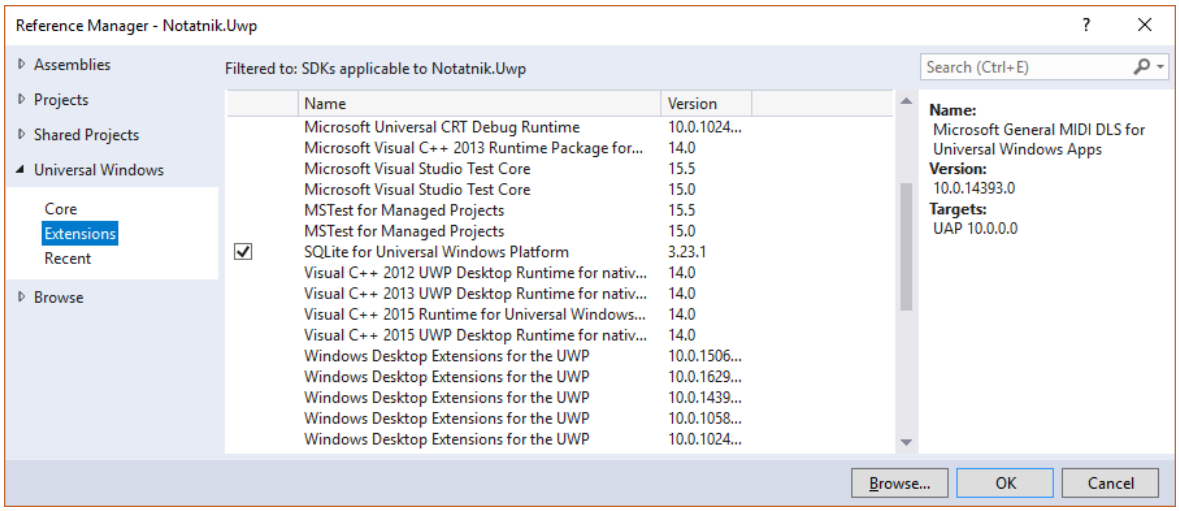
```
public class Setup : MvxAndroidSetup
{
      public Setup(Context applicationContext) : base(applicationContext)
      {
      }
      protected override IMvxApplication CreateApp() => new Core.App();
      protected override IMvxTrace CreateDebugTrace() => new
      DebugTrace();

      protected override void InitializePlatformServices()
      {
            base.InitializePlatformServices();
            Mvx.RegisterType<IDatabaseService, AndroidDatabaseService>();
      }
}
```

W ten sposób zakończyliśmy przygotowania projektu dla platformy Android, która bazuje oczywiście na utworzonej wcześniej części wspólnej oraz korzysta z bazy danych *SQLite*, w której zapisuje tworzone przez użytkownika notatki.

4.2.4 Aplikacja Notatnik – projekt UWP

Podobnie jak w przypadku projektu na Androida, w projekcie na platformę UWP również instalujemy te same pakiety NuGet oraz konfigurujemy projekt (por. opis w rozdziale 3). W nowym projekcie dodajemy referencję do projektu części wspólnej oraz dodatkowo do projektu-warstwy dostępu do bazy danych.

W przypadku dostępu do bazy danych na platformie UWP należy także dodać referencję do biblioteki o nazwie *SQLite for Universal Windows Platform*. Znajduje się ona w menadżerze referencji pod zakładką *Universal Windows* w podzakładce *Extensions* (rysunek 40).



Rys. 40 Miejsce z którego należy dodać referencję do biblioteki pliku *dll* z wsparciem dla *SQLite*

Zazwyczaj rozszerzenie to nie jest od razu dostępne w menadżerze referencji. Żeby je dodać, należy pobrać dodatek ze strony znajdującej się pod adresem *http://www.sqlite.org/download.html*. Nazwa tego dodatku to *sqlite-uwp-3230100.vsix*. Mimo opisu sugerującego że jest on przeznaczony tylko dla *Visual Studio 2015*, to rozszerzenie współpracuje również z *IDE* w wersji 2017. Po pobraniu pliku, wystarczy go zainstalować, a następnie zresetować *Visual Studio*. Po ponownym uruchomieniu dodatek powinien być dostępny w menadżerze referencji.

Po skonfigurowaniu projektu, należy stworzyć widoki, które będą związane z modelami widoku pochodzącymi z części wspólnej. Najpierw należy jednak zmodyfikować istniejący już pliku `MainPage.xaml`. Analogicznie jak w przypadku projektu dla systemu Android, należy na górze ekranu dodać przycisk, który będzie związany z poleceniem `CreateNoteCommand`. Drugim elementem tego widoku jest kontrolka typu `ListView`, związana z elementami listy znajdującej się w modelu widoku (`Notes`). Wiązanie z listą zostało zdefiniowane w atrybucie o nazwie `ItemsSource`.

W tym pliku jest także zdefiniowany widok dla każdego elementu listy. Jego definicja została umieszczona w znaczniku `ListView.ItemTemplate`. Wewnątrz tego znacznika został stworzony znacznik `StackPanel`, który zawiera dwa pola tekstowe związane z własnościami `Title` oraz `CreateDate`, znajdującymi się w poszczególnych elementach listy tj. w klasie `Note` z modelu widoku.

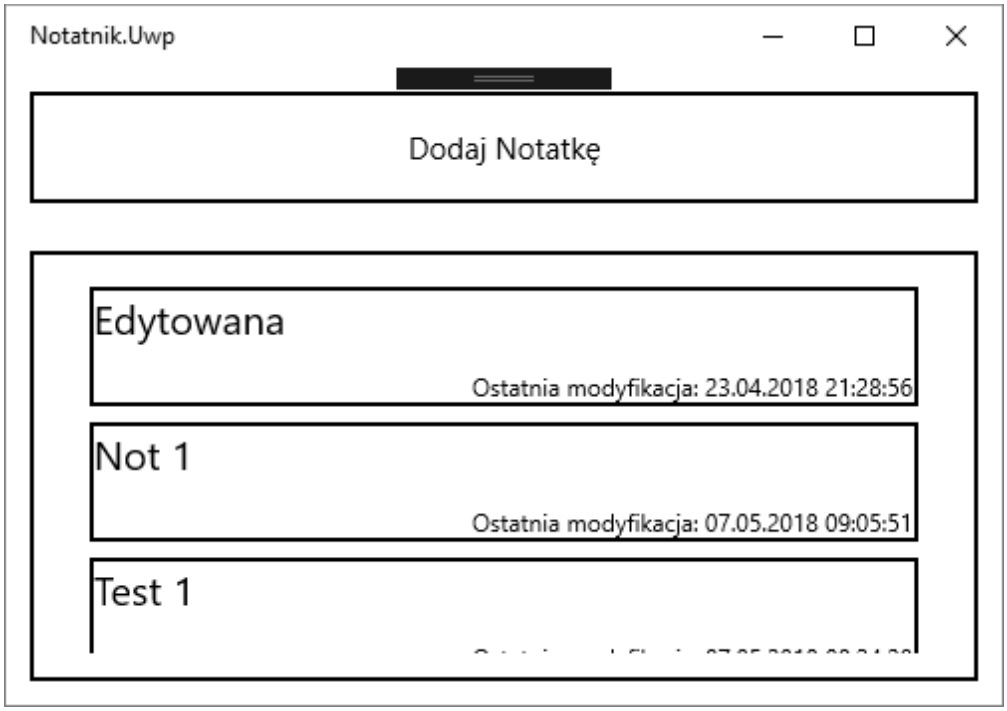
W przypadku platformy UWP nie ma możliwości bezpośredniego wiązania polecenia odpowiedzialnego za wykrycie naciśnięcia na danym elemencie listy. Dlatego, obsługa tego zdarzenia, została wykonana za pomocą zdarzeń, co powoduje konieczność

dodania kodu do tzw. *code-behind*, czyli klasy `MainPage` zdefiniowanej w pliku *MainPage.xaml.cs*. Nazwa metody (`ListView_ItemClick`), która będzie zajmowała się obsługą tego zdarzenia została przypisana do atrybutu `ItemClick` w znaczniku `ListView`. Dodatkowo należy ustawić wartość atrybutu `IsItemClickEnabled` na `true`. Bez tego aplikacja nie będzie w stanie obsługiwać zdarzenia kliknięcia (tapnięcia) elementów listy. Kod zdefiniowanego widoku został ukazany na listingu 41.

Listing 41 Zawartość pliku `MainPage.xaml`

```
<local:BasePage
    x:Class="Notatnik.Uwp.Pages.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Notatnik.Uwp.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource
    ApplicationPageBackgroundThemeBrush}">

        <Grid.RowDefinitions>
            <RowDefinition Height="80"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <Grid BorderBrush="Black"
              BorderThickness="2"
              Margin="12,12,12,12">
            <Button
                HorizontalAlignment="Stretch"
                Background="White"
                Command="{Binding CreateNoteCommand}"
                VerticalAlignment="Stretch" >
                Dodaj Notatkę
            </Button>
        </Grid>

        <Grid BorderBrush="Black"
              BorderThickness="2"
              Margin="12,12,12,12"
              VerticalAlignment="Top"
              Grid.Row="1" >
            <ListView
                IsItemClickEnabled="True"
                ItemClick="ListView_ItemClick"
                HorizontalContentAlignment="Stretch"
                ItemsSource="{Binding Notes}"
                HorizontalAlignment="Center"
                VerticalAlignment="Top"
                Margin="12,12,12,12">
```

```
                <ListView.ItemTemplate>
                    <DataTemplate>
                           <StackPanel BorderBrush="Black"
                               BorderThickness="2" Margin="4,4,4,4">
                                <TextBlock FontSize="20"
                                           Height="40"
                                           Width="1000"
                                           Foreground="Black"
                                           Text="{Binding Title}"
                                           HorizontalAlignment="Left"/>
                                <TextBlock FontSize="12"
                                           Foreground="Black"
                                           Text="{Binding CreateDate}"
                                           HorizontalAlignment="Right"/>
                            </StackPanel>
                        </DataTemplate>
                    </ListView.ItemTemplate>
                </ListView>
            </Grid>

    </Grid>
</local:BasePage>
```



Rys.41 Utworzony widok za pomocą kodu *xaml* z listingu 41.

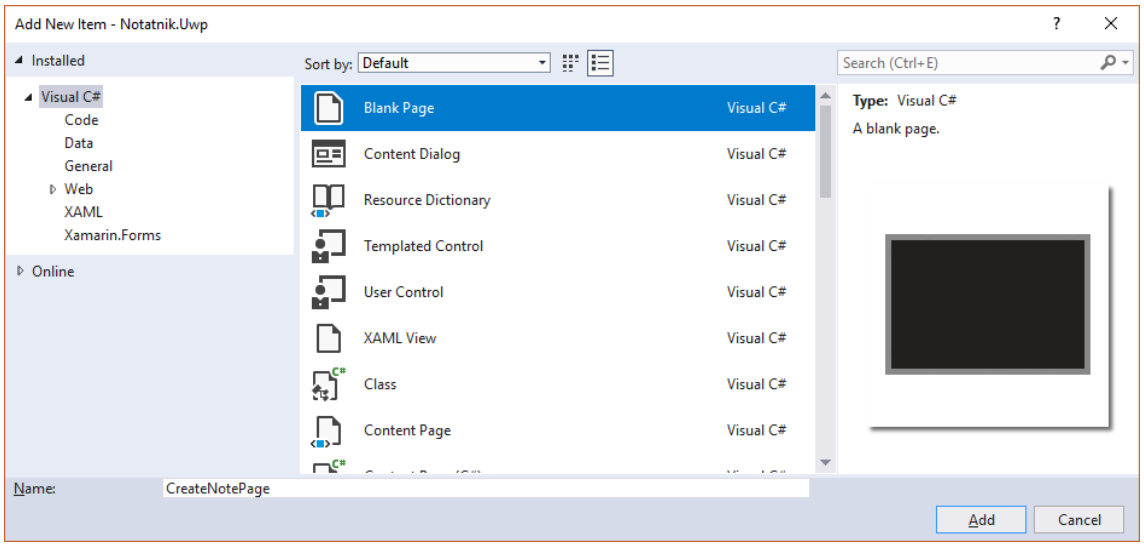
Następnie przechodzimy do edycji pliku *MainPage.xaml.cs*, w któryn należy nadpisać metodę o nazwie `OnNavigatedTo`. Jest ona wywoływana za każdym razem, gdy użytkownik otwiera daną stronę. W ciele tej metody należy wydobyć bieżący model widoku i wywołać na jego rzecz metodę `OnResume`. Dzięki temu aplikacja będzie pobierała elementy dostępne w bazie danych i aktualizowała wyświetlaną w widoku listę.

Dodatkowo należy utworzyć metodę `ListView_ItemClick`, której argumentami są obiekt nadawcy oraz obiekt `ItemClickEventArgs`, który we własności `ClickedItem` zawiera element kliknięty na wyświetlanej w widoku liście. W ciele tej metody, odwołujemy się ponownie do związanego z widokiem modelu widoku, wykonując polecenie `ItemClickedCommand` i przekazując do niego aktualnie kliknięty element listy (`e.ClickedItem`). Kod klasy `MainPage` został przedstawiony na listingu 42.

Listing 42 Zawartość pliku `MainPage.xaml.cs`

```
public sealed partial class MainPage : BasePage
{
      public MainPage()
      {
            InitializeComponent();
      }

      protected override void OnNavigatedTo(NavigationEventArgs e)
      {
            base.OnNavigatedTo(e);
            (ViewModel as MainViewModel).OnResume();
      }

      private void ListView_ItemClick(object sender,
      Windows.UI.Xaml.Controls.ItemClickEventArgs e)
      {
            (ViewModel as MainViewModel).
            ItemClickedCommand.Execute(e.ClickedItem);
      }
}
```

Następnie do projektu należy dodać kolejny widok o nazwie `CreateNotePage`, który będzie obsługiwany przez model widoku `CreateNoteViewModel`. Plik z nowym widokiem należy utworzyć w folderze *Pages*, korzystając z szablonu pustej strony (ang. *blank page*), co zostało przedstawione na rysunku 42.



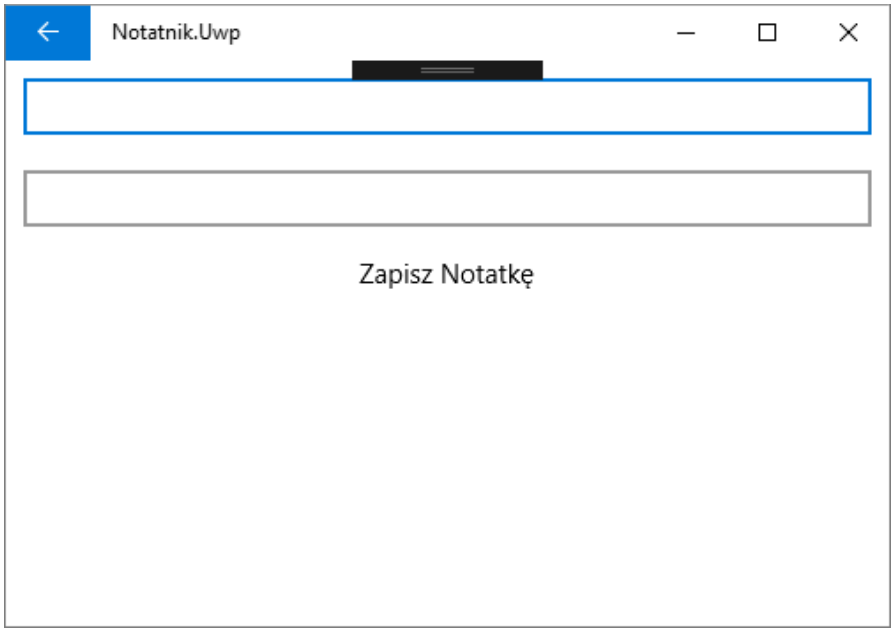
Rys. 42 Dodawanie nowej strony do projektu UWP

W pliku widoku, zostają utworzone dwie kontrolki `TextBox`, które będą odpowiedzialne za pobieranie informacji o tytule i zawartości tworzonej notatki. Został utworzony również przycisk, który został związany z komendą `CreateNoteCommand`. Cały kod został pokazany na listingu 43. W tym przypadku, należy również zmienić plik *CreateNotePage.xaml.cs*, a dokładnie zmienić klasę po której dziedziczy klasa `CreateNotePage` z `Page` na `BasePage`.

Listing 43 Zawartość pliku `CreateNotePage.xaml`

```
<local:BasePage
    x:Class="Notatnik.Uwp.Pages.CreateNotePage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Notatnik.Uwp.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource
    ApplicationPageBackgroundThemeBrush}">
        <StackPanel >
            <TextBox
                Foreground="Black"
                Text="{Binding Title,Mode=TwoWay}"
                HorizontalAlignment="Stretch"
                Margin="10"/>
            <TextBox
                Foreground="Black"
                TextWrapping="Wrap"
                Text="{Binding Content,Mode=TwoWay}"
                HorizontalAlignment="Stretch"
                Margin="10"/>
            <Button
```

```
                Command="{Binding CreateNoteCommand}"
                HorizontalAlignment="Stretch"
                Background="White"
                VerticalAlignment="Stretch" >
                Zapisz Notatkę
            </Button>
        </StackPanel>
    </Grid>
</local:BasePage>
```



Rys.43 Utworzony widok za pomocą kodu *xaml* z listingu 43.

W projekcie należy jeszcze odpowiednio załadować wtyczki oraz stworzyć i zarejestrować klasę odpowiedzialną za połączenie z bazą danych *SQLite*.

Najpierw tworzymy folder *Services* i dodajemt do niego klasę o nazwie `UwpDatabaseService`. Będzie ona implementowała interfejs `IDatabaseService`, w którym zostanie udostępnione połączenie z bazą danych *SQLite*, tak samo jak w przypadku projektu na platformę Android. W tej klasie należy określić poprawną ścieżkę bazy danych (folder zostanie ustalony dzięki obiektowi `ApplicationData`) z jej użyciem obiekt odpowiedzialny za połączenie specyficzny dla platformy UWP (klasa `SQLitePlatformWinRT`). Cały kod został przedstawiony na listingu 44.

Listing 44 Zawartość klasy `UwpDatabaseService`

```
public class UwpDatabaseService : IDatabaseService
{
      private SQLiteConnection connection;
      public SQLiteConnection Connection
      {
```

```
            get
            {
                  if (connection == null)
                  {
                        var path =
                        Path.Combine(ApplicationData.Current.LocalFolder.
                        Path, "notesDb.sqlite");
                        connection = new SQLiteConnection(new
                        SQLitePlatformWinRT(), path);
                  }

                return connection;
            }
      }
}
```

Należy także utworzyć klasę `Setup` (analogicznie jak to miało miejsce w projekcie dla systemu Android). W niej należy nadpisać kod metody `LoadPlugins`, wczytując wtyczkę `Messenger`. Następnie nadpisujemy metodę `InitializePlatformServices` i rejestrujemy w niej klase `UwpDatabaseService` implementującą interfejsu `IDatabaseService`. Kod klasy `Setup` został przedstawiony na listingu 45.

Listing 45 Zawartość klasy `Setup`

```
public class Setup : MvxWindowsSetup
{
      public Setup(Frame rootFrame) : base(rootFrame)
      {
      }

      protected override IMvxApplication CreateApp() => new Core.App();

      protected override MvxLogProviderType GetDefaultLogProviderType()
      => MvxLogProviderType.None;

      protected override void InitializePlatformServices()
      {
            base.InitializePlatformServices();
            Mvx.RegisterType<IDatabaseService, UwpDatabaseService>();
      }

      public override void LoadPlugins(IMvxPluginManager pluginManager)
      {
            pluginManager.EnsurePluginLoaded<MvvmCross.Plugins.Messenger.
            PluginLoader>();
            base.LoadPlugins(pluginManager);
      }
}
```

4.2.5 Aplikacja Notatnik – projekt iOS

Ostatni będzie projekt aplikacji dla systemu iOS. Postępujemy przy tym zgodnie z opisem z rozdziału 3: ponownie należy zainstalować pakiety *NuGet* (te same, jak w części wspólnej). Należy również dodać odwołanie do projektu części wspólnej oraz do projektu odpowiedzialnego za połączenie z bazą danych.

Pracę nad projektem rozpoczniemy od konfiguracji klasy, odpowiedzialnej za dostarczenia połączenia do bazy danych. I tym razem należy stworzyć folder *Services* i umieścić w nim klasę o nazwie `IphoneDatabaseService`, która będzie implementowała interfejs `IDatabaseService`. Podobnie jak w poprzednich projektach, należy zdefiniować poprawną ścieżkę do zapisu danych (tym razem ścieżkę folderu odczytujemy z klasy `Environment`) oraz specyficzny obiekt połączeniowy dla platformy iOS, czyli `SQLitePlatformIOS`. Kod klasy został przedstawiony na listingu 46.

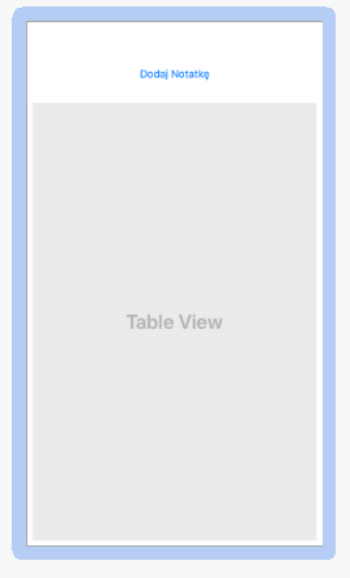
Listing 46 Zawartość klasy `IphoneDatabaseService`

```
public class IphoneDatabaseService : IDatabaseService
{
      private SQLiteConnection connection;
      public SQLiteConnection Connection
      {
            get
            {
                  if (connection == null)
                  {
                        var path = Path.Combine(Environment
                        .GetFolderPath(Environment.SpecialFolder
                        .MyDocuments), "notesDb.sqlite");
                        connection = new SQLiteConnection(new
                        SQLitePlatformIOS(), path);
                  }

                return connection;
            }
      }
}
```

Następnie w klasie `Setup`, podobnie jak w przypadku platformy Android, należy przeciążyć metodę `InitializePlatformServices`, rejestrując wewnątrz jej klasę `IphoneDatabaseService` (listing 47).

Listing 47 Zawartość klasy `Setup`

```
public class Setup : MvxIosSetup
{
      public Setup(IMvxApplicationDelegate applicationDelegate, UIWindow
      window): base(applicationDelegate, window)
      {
      }

      public Setup(IMvxApplicationDelegate applicationDelegate,
      IMvxIosViewPresenter presenter) : base(applicationDelegate,
      presenter)
      {
      }

      protected override IMvxApplication CreateApp() => new Core.App();

      protected override IMvxTrace CreateDebugTrace()
                        => new DebugTrace();

      protected override void InitializePlatformServices()
      {
            base.InitializePlatformServices();
            Mvx.RegisterType<IDatabaseService, IphoneDatabaseService>();
      }
}
```



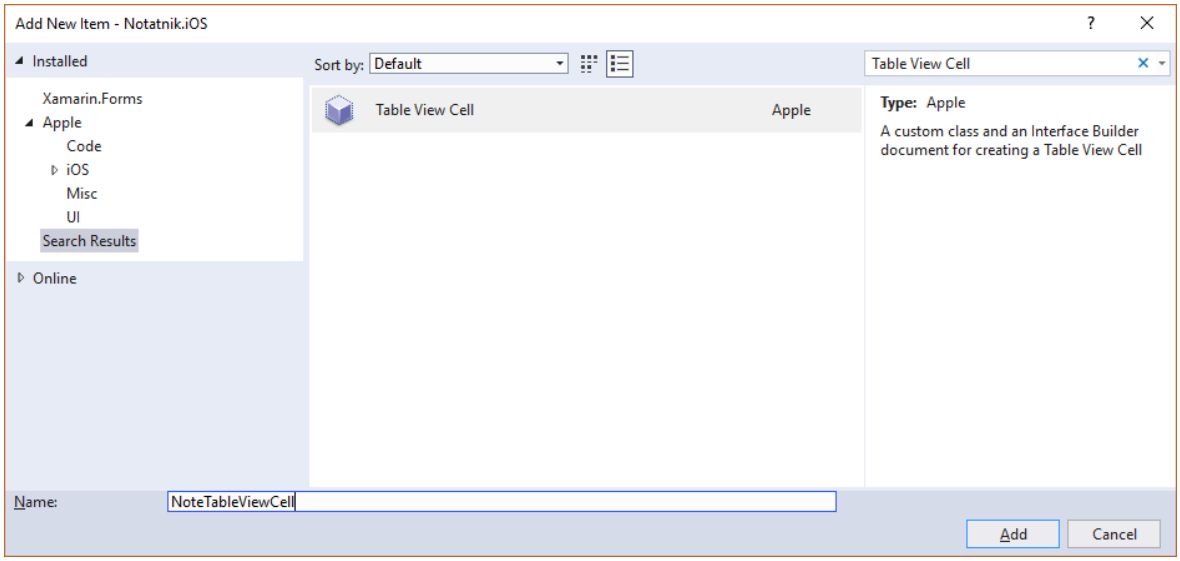
Rys. 44 Widok `MainView` po wprowadzonych zmianach

Po skonfigurowaniu połączenia z bazą danych *SQLite*, można zająć się tworzeniem widoków. Najpierw należy zmodyfikować istniejący już widok `MainView`. W tym celu należy otworzyć plik *MainView.xib* i zmienić nazwę (własność *Name* w sekcji *Identity*) istniejącego już tam przycisku na `addButton` oraz przesunąć go do górnej krawędzi ekranu. Dodatkowo zmieniamy jego tytuł (property *Title* w sekcji *Button*) na *Dodaj*

Notatkę. Następnie należy dodać kontrolkę typu `Table View`. W platformie iOS służy ona do wyświetlania wiele elementów, które mogą być pobieranie z listy. Samej kontrolce nadajemy nazwę `noteList`.

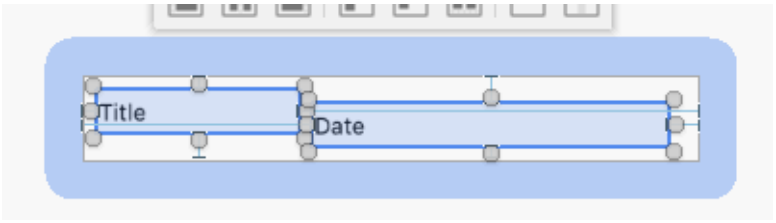
Na sam koniec ustalamy pozycję kontrolek względem siebie i do krawędzi ekranu (*constaints*). W przypadku przycisku zakotwiczamy go z górą oraz bocznymi krawędziami ekranu. W przypadku kontrolki `Table View`, zakotwiczamy go do dolnej krawędzi ekranu, bocznych oraz dodatkowo do środkowej części w kontrolce `button`. Dzięki temu nasz widok powinien dopasować się do każdego rozmiaru ekranu w telefonach z systemem iOS. Widok jaki powinniśmy uzyskać został pokazany na rysunku 44.

Następnie do folderu *Views* należy dodać nowy zasób typu `Table View Cell`. W nim zostanie zdefiniowany widok dla pojedynczego elementu pobranego z modelu widoku i wyświetlanego w liście.



Rys. 45 Tworzenie zasobu odpowiedzialnego za definicje widoku dla elementów w liście iOS

Należy zmodyfikować utworzomy przed chwilą plik *NoteTableViewCell.xib*, w którym należy zbudować widok z dwóch kontrolek `Label`, umożliwiający wyświetlanie informacji o tytule i dacie ostatniej modyfikacji notatki. Każda z kontrolek `Label` powinna mieć wielkość czcionki (property *Font* w sekcji *Label*) równą 12. Kontrolkę ustawioną z lewej strony nazwijmy `titleLabel`, a z prawej – `dateLabel`. Kontrolki zakotwiczamy z wszystkimi czterema krawędziami obszaru widoku. Widok jaki powinniśmy w ten sposób uzyskać powinien być zbliżony do tego, pokazanego na rysunku 46.



Rys. 46 Widok dla pojedynczej notatki na liście

Następnie należy zmodyfikować plik *NoteTableViewCell.cs*, w którym zostanie utworzone wiązanie danych dla pojedynczego elementu listy. W tym celu, podobnie jak w przypadku normalnych widoków, utworzony zostaje tzw. „binding set", który wiąże klasę `NoteTableViewCell` z `Note`. Ta ostatnia jest typem elementów listy przekazywanych z modelu widoku. Następnie należy związać obie te kontrolki z odpowiednimi własnościami pochodzącymi z klasy `Note`, a mianowicie z `Title` i `CreateDate` (listing 48).

Listing 48 Zawartość pliku `NoteTableViewCell.cs`

```
public partial class NoteTableViewCell : MvxTableViewCell
{
      public static readonly NSString Key
      = new NSString("NoteTableViewCell");
      public static readonly UINib Nib;

      static NoteTableViewCell()
      {
            Nib = UINib.FromName("NoteTableViewCell",
            NSBundle.MainBundle);
      }

      protected NoteTableViewCell(IntPtr handle) : base(handle)
      {
            this.DelayBind(() =>
            {
               var set = this.CreateBindingSet<NoteTableViewCell,Note>();
                set.Bind(titleLabel).To(vm => vm.Title);
                set.Bind(dateLabel).To(vm => vm.CreateDate);
                set.Apply();
            });
        }
}
```

W kolejnym kroku należy utworzyć klasę `NoteTableViewSource`, która będzie odpowiedzialna za związanie pojedynczego elementu listy przesłanej z modelu widoku z komórkami kontrolki `Table View`. W konstruktorze tej klasy oprócz referencji do kontrolki, przekazujemy również referencję model widoku. Zostanie ona wykorzystana

w przeciążonej metodzie `RowSelected`, zwracającej indeks elementu, który został aktualnie naciśnięty na wyświetlanej liście elementów. Dzięki referencji do modelu widoku oraz dzięki indeksowi można odwołać się do konkretnego elementu z listy, a następnie wywołać polecenie odpowiedzialne za edycję zaznaczonej notatki (tak jak miało to miejsce w przypadku projektu UWP). Kod całej klasy został przedstawiony na listingu 49.

Listing 49 Zawartość klasy `NoteTableViewSource.cs`

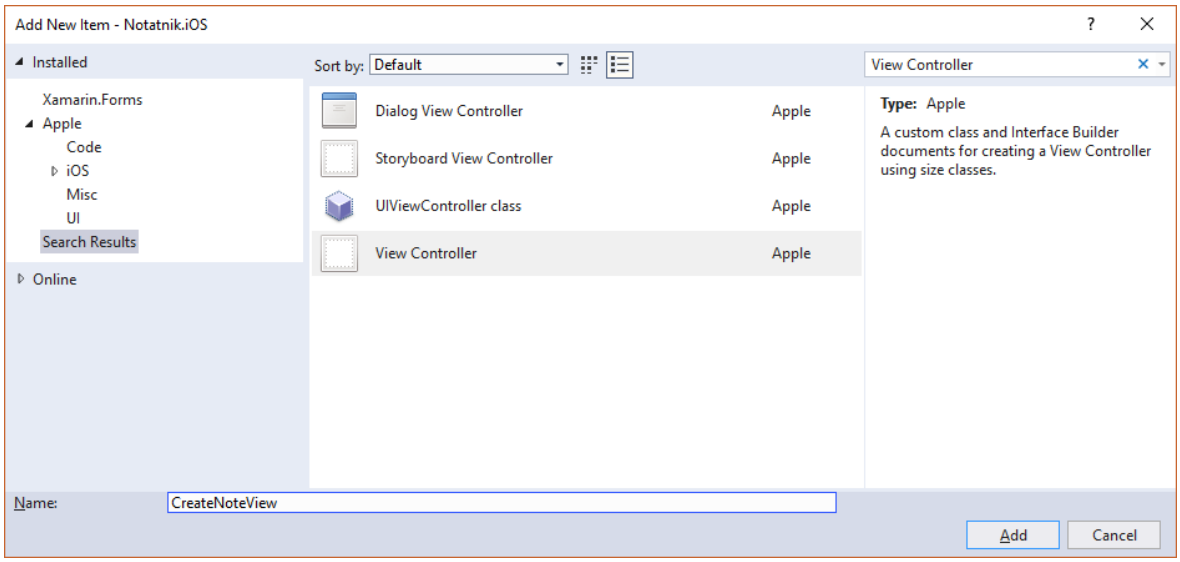
```
public class NoteTableViewSource : MvxTableViewSource
{
      private readonly MainViewModel _parentViewModel;

      public NoteTableViewSource(UITableView table,MainViewModel
      parentViewModel) : base (table)
      {
            table.RegisterNibForCellReuse(NoteTableViewCell.Nib,
            NoteTableViewCell.Key);
            _parentViewModel = parentViewModel;
      }

      protected override UITableViewCell GetOrCreateCellFor(UITableView
      tableView, NSIndexPath indexPath, object item)
      {
            return tableView.DequeueReusableCell(NoteTableViewCell.Key,
            indexPath);
      }

      public override void RowSelected(UITableView tableView, NSIndexPath
      indexPath)
      {
            var item = (ItemsSource as
            ObservableCollection<Note>)[indexPath.Row];
            _parentViewModel.ItemClickedCommand.Execute(item);
      }
}
```

Teraz można zająć się klasy `MainView`, w której zdefiniujemy wiązanie pomiędzy kontrolkami z warstwy widoku i modelem widoku. W kodzie tej klasy zostanie również utworzona nowa instancja obiektu `NoteTableViewSource`, odpowiedzialnej za odpowiednie wyświetlanie danych w kontrolce `TableView`. Dodatkowo należy nadpisać metodę `ViewWillAppear`, która wywoływana jest za każdym razem, gdy użytkownik przejdzie do widoku, na którym wyświetlana jest lista elementów. W tym momencie należy wywołać zdefiniowaną w części wspólnej metodę `OnResume`, która pobierze utworzone notatki z bazy danych i zaktualizuje listę, która jest wyświetlana w widoku aplikacji. Zmieniony kod klasy `MainView` został umieszczony na listingu 50.

Listing 50 Zawartość klasy `MainView`

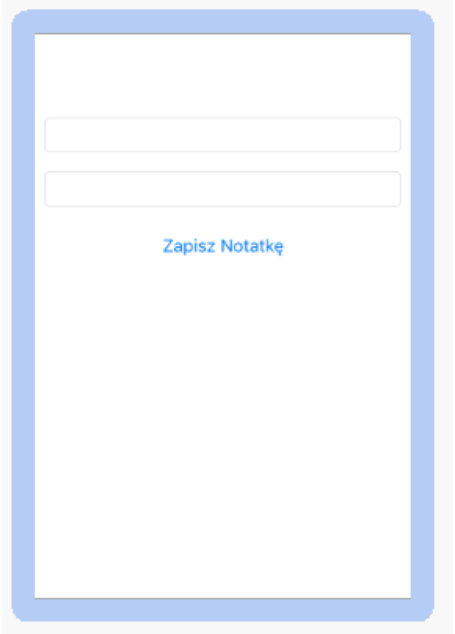
```
[MvxRootPresentation(WrapInNavigationController = true)]
public partial class MainView : MvxViewController
{
	public MainView() : base("MainView", null)
	{
	}

	public override void ViewDidLoad()
	{
		base.ViewDidLoad();

		var set = this.CreateBindingSet<MainView,
		Core.ViewModels.MainViewModel>();
		set.Bind(addButton).To(vm => vm.CreateNoteCommand);

		var source = new NoteTableViewSource(noteList,ViewModel as
		MainViewModel);
		noteList.Source = source;
		set.Bind(source).For(x => x.ItemsSource).To(vm => vm.Notes);

		set.Apply();
	}

	public override void ViewWillAppear(bool animated)
	{
		base.ViewWillAppear(animated);
		ViewModel as MainViewModel).OnResume();
	}
}
```

Pozostało jeszcze przygotowanie widoku, który pozwoli na dodawanie nowych i edytowanie już istniejących notatek. W tym celu do folderu *Views* należy dodać zasób typu *View Controller* (rysunek 40), a następnie edytujemy utworzony w ten sposób plik *xib*. Zacznijmy od dodania trzech kontrolek: dwóch pól tekstowych (*Text Field*) oraz przycisku (*Button*). Pola tekstowe ustawiamy na górze, zostawiając odstęp od górnej krawędzi. W platformie iOS, zostanie ona domyślnie wypełniona przez belkę nawigacyjną, która jest dodawana podczas przechodzenia z jednego widoku do innego.



Rys. 47 Tworzenie nowego widoku dla aplikacji iOS

Pola tekstowe nazywamy kolejno `titleText` oraz `contentText`. Pod nimi umieszczamy przycisk z nazwą `addNote` i tytułem *Zapisz Notatkę*. Na koniec zakotwiczamy elementy widoku. Pierwsze pole tekstowe zakotwiczamy do górnej krawędzi ekranu oraz krawędzi bocznych, drugie pole tekstowe – z pierwszym polem tekstowym oraz bocznymi, Natomiast przycisk – z bocznymi krawędziami oraz z kontrolką `contentText` (rysunek 44).



Rys. 44 Widok do tworzenia i edytowania notatek

Jako ostatnią zmienianą klasą jest `CreateNoteView` (listing 51). Powinna ona dziedziczyć po klasie `MvxViewController`, podobnie jak `MainView`. Należy w niej również zdefiniować wiązania pomiędzy własnościami i poleceniami udostępnianymi przez modelu widoku `CreateNoteViewModel`. W tym celu w nadpisanej metodzie `ViewDidLoad` pole tekstowe `titleText` ma być związane z własnością `Title`, kontrolka `contentText` – z własnością `Content`, a przycisk `addNote` z poleceniem `CreateNoteCommand`. Kod metody kończy się wywołaniem metody `Apply`, co powoduje faktyczne ustanowienie powyższych wiązań.

Listing 51 Zawartość klasy `CreateNoteView`

```
public partial class CreateNoteView : MvxViewController
{
      public CreateNoteView() : base("CreateNoteView", null)
      {
      }

      public override void ViewDidLoad()
      {
            base.ViewDidLoad();
            var set =
            this.CreateBindingSet<CreateNoteView,CreateNoteViewModel>();
            set.Bind(titleText).To(vm => vm.Title);
            set.Bind(contentText).To(vm => vm.Content);
            set.Bind(addNote).To(vm => vm.CreateNoteCommand);
            set.Apply();
        }
}
```

Po wprowadzeniu przedstawionych zmian projekt jest zakończony i może zostać uruchomiony na dowolnym emulatorze lub telefonie z system iOS.

4.3 Aplikacja Zadania

Ostatni omawiany w tej pracy projekt aplikacji będzie umożliwiał tworzenie systemowych powiadomień (ang. *notifications*). W aplikacjach dla poszczególnych platform zostaną zaimplementowane okna modalne tzw. okna *pop-up* – inaczej przygotowywane w każdej z platform. Dodatkowo każde powiadomienie będzie mogło zostać udostępnione innym aplikacjom, co umożliwi np. wysłanie maila z wpisanym w aplikacji tekstem. Wykorzystamy do tego gotowe moduły z platform Android, UWP oraz iOS.

4.3.1 Aplikacja Zadania – projekt części wspólnej

Tworzymy nowy projekt typu *Portable Class Library* o nazwie *Zadania.Core*. Projekt części wspólnej rozpoczynamy od zainstalowania w nim pakietu startowego *MvvmCross.StarterPack*. Następnie należy utworzyć w tym projekcie trzy foldery: *Converters* , *Services* oraz *ViewModels*. W pierwszym folderze należy utworzyć dwie klasy, które zostaną wykorzystane do konwersji daty i czasu. W pierwszym kroku należy dodać klasę o nazwie *DateConverter* (listing 52). Podobnie jak w poprzednio tworzonych konwerterach, klasa ta musi implementować interfejs *IMvxValueConverter*, który wymusza zdefiniowanie metod służących do konwersji obiektów z warstwy widoku do obiektów z warstwy modelu widoku oraz odwrotnie.

Listing 52 Zawartość klasy `DateConverter`

```
public class DateConverter : IMvxValueConverter
{
      public object Convert(object value, Type targetType, object
      parameter, CultureInfo culture)
      {
            try
            {
                DateTime date = (DateTime)value;
                return new DateTimeOffset(date);
            }
            catch (Exception ex)
            {
                return DateTimeOffset.MinValue;
            }
      }

      public object ConvertBack(object value, Type targetType, object
      parameter, CultureInfo culture)
      {
            try
            {
```

```
                DateTimeOffset dto = (DateTimeOffset)value;
                return dto.DateTime;
            }
            catch (Exception ex)
            {
                return DateTime.MinValue;
            }
        }
}
```

Druga klasa konwertera to `TimeConverter`. Będzie on odpowiedzialny za konwersję czasu z oraz do modelu widoku. Jej kod został przedstawiony na listingu 53.

Listing 53 Zawartość klasy `TimeConverter`

```
public class TimeConverter : IMvxValueConverter
{
      public object Convert(object value, Type targetType, object
      parameter, CultureInfo culture)
      {
            try
            {
                  var time = (TimeSpan)value;
                  return new DateTime(2018, 1, 1, time.Hours,
                  time.Minutes, time.Seconds);
            }
            catch (Exception ex)
            {
                return DateTime.MinValue;
            }
      }

      public object ConvertBack(object value, Type targetType, object
      parameter, CultureInfo culture)
      {
            try
            {
                DateTime date = (DateTime)value;
                return new TimeSpan(date.Hour, date.Minute, date.Second);
            }
            catch (Exception ex)
            {
                return DateTimeOffset.MinValue;
            }
      }
}
```

Następnie do folderu *Services* należy dodać trzy nowe pliki zawierające interfejsy o nazwach: `IModalScreenService`, `IScheduledNotificationService` oraz `IShareService`. Pierwszy z nich będzie wymuszał implementację metody, która na poszczególnych platformach pokaże okno modalne z pytaniem i możliwością odpowiedzi *Tak* lub *Nie* oraz zapewni wykonanie odpowiednich akcji zdefiniowanych w modelu widoku w zależności od wybranej opcji. Ostatnie dwa argumenty tej metody będą

przekazywały akcje (delegacja typu `Action`), które zostaną wywołane w zależności od wybranej przez użytkownika odpowiedzi. Kod interfejsu został przedstawiony na listingu 54.

Listing 54 Zawartość interfejsu `IModalScreenService`

```
public interface IModalScreenService
{
      void ConfirmAdditionalAction(string title, string content,
      Action confirmAction, Action cancelAction);
}
```

Z kolei interfejs `IScheduledNotificationService` (listing 55) będzie wymuszał metodę służącą do dodania powiadomienia, które zostanie wyświetlona o zadanym czasie (odpowiada za to argument `DateTime`) oraz z określonym tekstem (argument `string`).

Listing 55 Zawartość interfejsu `IScheduledNotificationService`

```
public interface IScheduledNotificationService
{
      void AddNotification(DateTime startDate, string content);
}
```

Ostatnim interfejsem jest `IShareService` (listing 56). Wymusza implementację metody `Share`, odpowiedzialnej za wyświetlanie możliwych sposobów udostępniania tekstu wpisanego podczas tworzenia zadania w aplikacji. Kod tego interfejsu pokazano na listingu 56.

Listing 56 Zawartość interfejsu `IShareService`

```
public interface IShareService
{
      void Share(string content);
}
```

Ostatnią zmianą w części wspólnej będzie zmodyfikowanie kodu klasy `MainViewModel` (należy umieścić go w folderze *ViewModels*). W argumentach konstruktora tej klasy, należy przekazać obiekty implementujące trzy interfejsy pokazane na listingach 54, 55 oraz 56. Przekazane obiekty, a właściwie ich referencje zapisujemy w prywatnych polach. W klasie `MainViewModel` należy umieścić również trzy własności. Pierwsza o nazwie `Work` typu `string` jest odpowiedzialna za przechowywanie treści,

jaka zostanie wyświetlone w powiadomieniu oraz może być udostępniona z pomocą klasy implementującej interfejs `IShareService`. Następna własność o nazwie `Date` to zmienna typu `DateTime`, która będzie odpowiedzialna za przechowywanie daty dnia, w którym ma zostać pokazana notyfikacja systemowa. Ostatnia własność o nazwie `Time` jest zmienną typu `TimeSpan`.[8] Jest odpowiedzialna za przetrzymywanie godziny o której zostanie wywołana notyfikacja z informacją o dodanym zadaniu.

Następnie w modelu widoku (klasa `MainViewModel`) należy utworzyć dwie metody. Pierwsza o nazwie `ConfirmShare` będzie służyła do udostępniania zadania za pomocą klasy implementującej interfejs `IShareService`. Druga o nazwie `SetDefaultValues` po pomyślnym zapisaniu powiadomienia czyści wcześniej wprowadzone w kontrolkach parametry.

Do klasy modelu widoku `MainViewModel` należy również dodać polecenie o nazwie `AddWorkCommand`, z którym zostanie związany przycisk w widoku, odpowiedzialny za zatwierdzenie nowego zadania. Wówczas wywołana zostanie metoda odpowiedzialna za dodanie notyfikacji (w klasie implementującej interfejs `IScheduledNotificationService` pochodzący z folderu *Services*) oraz zostanie wyświetlone okno modalne. Do metody, która je otworzy przekazane zostaną dwie akcje, które mogą zostać wykonane w zależności od wybranej przez użytkownika opcji. Kod klasy modelu widoku widoczny jest na listingu 57.

Listing 57 Zawartość klasy `CreateNoteView`

```
public class MainViewModel : MvxViewModel
{
      private readonly IShareService _shareService;
      private readonly IModalScreenService _popupService;
      private readonly IScheduledNotificationService
      _scheduledNotificationService;

      public MainViewModel(IScheduledNotificationService
      scheduledNotificationService,
      IShareService shareService, IModalScreenService popupService)
      {
            _scheduledNotificationService = scheduledNotificationService;
            _shareService = shareService;
            _popupService = popupService;
      }
```

[8] Platforma Android nie udostępnia żadnej standardowej kontrolki pozwalającej na jednoczesne ustawienie daty i czasu. W związku z tym, aby zachować spójność pomiędzy platformami, w każdym z projektów zostały użyte osobne kontrolki do kontroli daty i czasu. W konsekwencji przechowuję również osobno datę i czas we własnościach modelu widoku, pomimo że typ `DateTime` może przechowywać obie te wielkości.

```
        public IMvxCommand AddWorkCommand => new MvxCommand(() =>
        {
              _scheduledNotificationService.AddNotification(
              new DateTime(_date.Year, _date.Month, _date.Day, _time.Hours,
              _time.Minutes, 0),_work);
              _popupService.ConfirmAdditionalAction("Twoje zadanie zostało
              zapisane",
              "Czy chcesz udostępnić swoje zadanie?", () => {
              ConfirmShare(); }, () => { SetDefaultValues(); });
        });

        private void ConfirmShare()
        {
              _shareService.Share("Zadanie do wykonania:\n"+Work+
                  " dnia: "+Date.ToString("dd.mm.yyyy"));
              SetDefaultValues();
        }

        private void SetDefaultValues()
        {
              Date = DateTime.Now;
              Time = new TimeSpan(0, 0, 0);
              Work = "";
        }

        private DateTime _date = DateTime.Now;
        public DateTime Date
        {
              get { return _date; }
              set { SetProperty(ref _date, value); }
        }

        private TimeSpan _time = new TimeSpan(0, 0, 0);
        public TimeSpan Time
        {
              get { return _time; }
              set { SetProperty(ref _time, value); }
        }

        private string _work = "";
        public string Work
        {
              get { return _work; }
              set { SetProperty(ref _work, value); }
        }
}
```

Do zrobienia pozostaje zamiana wywołania nieaktualnej (ang. *obsolete*) metody `RegisterAppStart` na wywołanie metody `RegisterNavigationServiceAppStart`, która ustawi model widoku `MainViewModel` jak startowy w tworzonej aplikacji.

4.3.2 Aplikacja Zadania – projekt Android

W projekcie na platformę Android do udostępniania stworzonych zadań zostanie wykorzystany intencje. Intencje są obok aktywności jednym z podstawowych mechanizmów wykorzystywanych w projektach aplikacji dla systemu Android. Mechanizm ten odpowiedzialny jest przede wszystkim za obsługę poleceń wydawanych przez użytkowników aplikacji. Z pomocą intencji można również nawiązać komunikację pomiędzy odrębnymi aplikacjami (lub mniejszymi komponentami np. usługami czy aktywnościami). [30] W opisywanym w tym podrozdziale projekcie, intencje zostaną wykorzystane do udostępnienia zadania (jego treści) innym aplikacjom, które są zainstalowane w systemie. Dodatkowo intencje zostaną wykorzystane do pokazywania powiadomień systemowych oraz do tego, żeby powiadomienie zostało pokazane w zadanym dniu i o zadanej godzinie. Wykorzystany zostanie również menadżer dialogów do wyświetlania okien pop-up oraz menadżer powiadomień.

Po zainstalowaniu pakietu startowego frameworka MvvmCross oraz przeprowadzeniu jego konfiguracji opisanej w rozdziale 3, należy w projekcie dla systemu Android stworzyć folder o nazwie *Services*, a następnie dodać do niego cztery klasy: `AndroidShareService`, `AndroidModalScreenService`, `AndroidScheduledNotificationService` oraz `AlarmReceiverService`.

Pierwsza z nich implementuje interfejs z projektu *PCL* części wspólnej o nazwie `IShareService`. Metoda `Share` musi uzyskać dostęp do aktualnie wyświetlanej aktywności. Może go uzyskać wykorzystując globalny obiekt `Mvx`, a dokładnie jego metody `Resolve` podając jako typ generyczny `IMvxAndroidCurrentTopActivity`. Po uzyskaniu dostępu do obecnie używanej aktywności, można wykorzystać mechanizm intencji do wyświetlenia dostępnych aplikacji, którym można udostępnić wpisany przez użytkownika tekst. W tym celu po stworzeniu obiektu typu `Intent`, należy ustawić jego typ (za pomocą metody `SetType`) na `text/plain`. Następnie można przekazać parametry do aplikacji za pomocą metody `PutExtra`, a całość wyświetlić, za pomocą nowej aktywności. W tym celu należy uruchomić metodę `StartActivity` i przekazać do niej intencję odpowiedzialną za udostępnienie tekstu innej aplikacji. Zawartość klasy `AndroidShareService` została przedstawiona na listingu 58.

Listing 58 Zawartość klasy `AndroidShareService`

```
public class AndroidShareService : IShareService
{
	public void Share(string content)
	{
		var currentActivity =
		Mvx.Resolve<IMvxAndroidCurrentTopActivity>();

		 Intent sharingIntent = new Intent(Intent.ActionSend);
		 sharingIntent.SetType("text/plain");
		 sharingIntent.PutExtra(Intent.ExtraSubject, "subject");
		 sharingIntent.PutExtra(Intent.ExtraText, content);
		sharingIntent.PutExtra(Intent.ExtraTitle, "Zadanie do
		wykonania");

		currentActivity.Activity.StartActivity(Intent.CreateChooser(s
		haringIntent,"Udostępnij za pomocą"));
	}
}
```

Drugą klasą, jaką należy umieścić w folderze *Services* jest `AndroidModalScreenService`, która implementuje interfejs `IModalScreenService` z metodą `ConfirmAdditionalAction`. Podobnie jak w poprzedniej klasie, wykorzystujemy klasę `Mvx`, aby uzyskać referencję do bieżącej aktywności. Następnie tworzymy nowy obiekt typu `AlertDialog`, który po przyjęciu aktywności jako argumentu wyświetli na niej modalne okno z dwoma opcjami do wyboru. Dostępne opcje zostały ustawione za pomocą metody `SetButton` oraz `SetButton2`.

Rys. 45 Widok panelu udostępniania, zaimplementowanego w klasie AndroidShareService

Każda z tych metod przyjmuje nazwę opcji oraz wyrażenie Lambda z akcją uruchamianą po wybraniu danej opcji.. Dodatkowo za pomocą metod `SetTitle` oraz `SetMessage` ustawiamy tytuł i zawartość okna modalnego. Na końcu zostaje wywołana metoda `Show`, która spowoduje wyświetlenie okna modalnego. Cały kod klasy został przedstawiony na listingu 59.

Listing 59 Zawartość klasy `AndroidModalScreenService`

```
public class AndroidModalScreenService : IModalScreenService
{
      public void ConfirmAdditionalAction(string title, string content,
      Action confirmAction, Action cancelAction)
      {
            var currentActivity =
            Mvx.Resolve<IMvxAndroidCurrentTopActivity>();

            var popup = new
            AlertDialog.Builder(currentActivity.Activity).Create();

            popup.SetTitle(title);
            popup.SetMessage(content);
            popup.SetButton("Tak", (s, a) => { confirmAction(); });
            popup.SetButton2("Nie", (s, a) => { cancelAction(); });

            popup.Show();
      }
}
```

Następnie do folderu *Services* dodajemy klasę `AlarmReceiverService`, która dziedziczy po klasie abstrakcyjnej `BroadcastReceiver`. Jest to klasa, dzięki której można odbierać powiadomienia wewnątrz tworzonej aplikacji.[31] Dodatkowo posiada atrybut `BroadcastReceiver`, bez którego klasa nie mogła by odbierać powiadomień.W tworzonym projekcie będą to powiadomienia rozgłaszane przez menadżer alertów (klasa `AlarmManager`), wywoływane podczas dodawania nowych zadań w aplikacji. W tej klasie należy przeciążyć bazową metodę o nazwie `OnReceive`, która w argumentach oprócz kontekstu aplikacji pobiera także obiekt intencji, z którego można wyciągnąć tekst (za pomocą metody `GetStringExtra`), który ma zostać wyświetlony na belce powiadomień. Następnie należy utworzyć nową intencję, która posłuży do wyświetlenia powiadomienia w systemie. Dodatkowo zostaje utworzony obiekt `PendingIntent`, który zezwala innym aktywnościom systemowym na korzystanie z uprawnień tworzonej aplikacji do wykonywania zdefiniowanego fragmentu kodu. W tym przypadku po zbudowaniu obiektu powiadomienia (za pomocą klasy `Notification` i jej podklasy `Builder`), zezwalamy na dostęp do aplikacji, wywołując w obiekcie

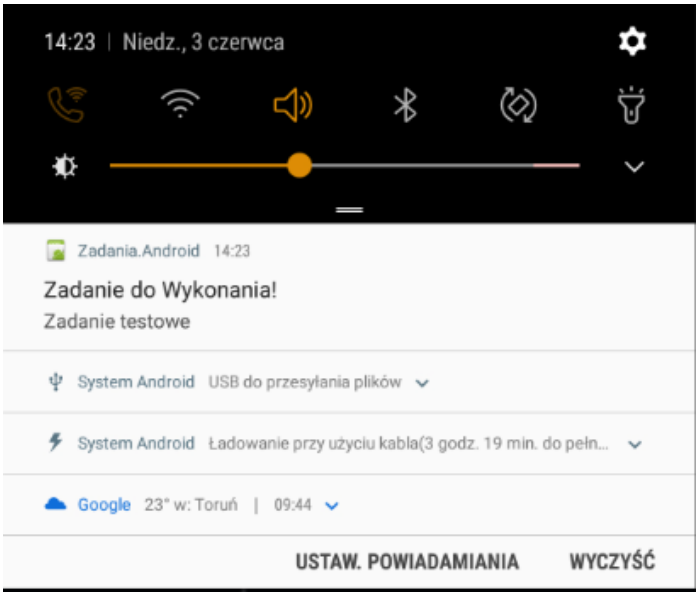
powiadomienia metodę `SetContentIntent`, która jako argument przyjmuje obiektu typu `PendingIntent`. Dzięki temu po tapnięciu na wyświetlane powiadomienie, uruchomiona lub przywołana zostanie projektowana aplikacja, dokładnie pokazany zostanie ekran, którym zarządza klasa modelu widoku `MainViewModel`. Na koniec z przekazanego kontekstu aplikacji, należy wyciągnąć obiektu zarządzający powiadomieniami (`NotificationManager`) i wywołać na nim metodę `Notify`, która spowoduje wyświetlenie powiadomienia w systemie Android. Cała implementacja została przedstawiona na listingu 60.

Listing 60 Zawartość klasy `AlarmReceiverService`

```
[BroadcastReceiver]
public class AlarmReceiverService : BroadcastReceiver
{
      public override void OnReceive(Context context, Intent intent)
      {
            var title = "Zadanie do Wykonania!";
            var message = intent.GetStringExtra("Content");
            Intent backIntent = new Intent(context, typeof(MainView));
            backIntent.SetFlags(ActivityFlags.NewTask);
            var resultIntent = new Intent(context, typeof(MainView));
            PendingIntent pending = PendingIntent.GetActivities(context,
            0,new Intent[] { backIntent, resultIntent },
                PendingIntentFlags.OneShot);

            var builder =
            new Notification.Builder(context)
                    .SetContentTitle(title)
                    .SetContentText(message)
                    .SetAutoCancel(true)
                    .SetSmallIcon(Resource.Mipmap.Icon)
                    .SetDefaults(NotificationDefaults.All);

            builder.SetContentIntent(pending);
            var notification = builder.Build();
            var manager = NotificationManager.FromContext(context);
            manager.Notify(1331, notification);
      }
}
```

Ostatnią klasą dodaną do folderu *Services* jest `AndroidScheduledNotificationService`. Klasa ta implementuje interfejs `IScheduledNotificationService`, który wymusza definicję metody `AddNotification`. W tej metodzie na samym początku wyciągamy bieżącą aktywność, następnie tworzymy nową intencję, która jako drugi argument przyjmuje typ poprzednio stworzonej klasy `AlarmReceiverService`. Za pomocą metody

PutExtra należy przekazać jej treść powiadomienia, która zostanie odebrana w klasie AlarmReceiverService.



Rys. 46 Widok notyfikacji aplikacji w systemie Android

Ostatnimi instrukcjami w metodzie AddNotification są polecenia tworzące obiekty PendingIntent oraz AlarmManager. Pierwszy z nich, zezwoli obiektowi AlarmManager na dostęp do zasobów naszej aplikacji i wysłania powiadomienia o zadanym czasie, po którego otrzymaniu nasza aplikacji stworzy i wyświetli powiadomienie w systemie Android. [32] Do wyliczenia czasu, zaimplementowana została dodatkowa funkcja GenerateNotificatiomTimeStamp, która na podstawie przekazanej daty, wylicza całkowity czas, po którym zostanie wyświetlona notyfikacji w systemie. Kod klasy został przedstawiony na listingu 61.

Listing 61 Zawartość klasy AndroidScheduledNotificationService

```
public class AndroidScheduledNotificationService :
IScheduledNotificationService
{
      public static long GenerateNotifcationTimeStamp(DateTime startDate)
      {
            var schedule = startDate;
            Java.Util.Calendar calendar = Java.Util.Calendar.Instance;
            calendar.Set(Java.Util.CalendarField.HourOfDay,
            schedule.Hour);
            calendar.Set(Java.Util.CalendarField.Minute,
            schedule.Minute);
            calendar.Set(Java.Util.CalendarField.Second, 00);
            return calendar.TimeInMillis;
```

```
        }

        public void AddNotification(DateTime startDate, string content)
        {
                var currentActivity =
                Mvx.Resolve<IMvxAndroidCurrentTopActivity>();
                var context = currentActivity.Activity.BaseContext;

                Intent alarmIntent = new Intent(context,
                typeof(AlarmReceiverService));
                alarmIntent.PutExtra("Content", content);
                PendingIntent pending = PendingIntent.GetBroadcast(context,
                0, alarmIntent, PendingIntentFlags.UpdateCurrent);
                AlarmManager alarmManager =
                context.GetSystemService("alarm").JavaCast<AlarmManager>();
                alarmManager.Set(AlarmType.RtcWakeup,
                GenerateNotifcationTimeStamp(startDate), pending);
        }
}
```

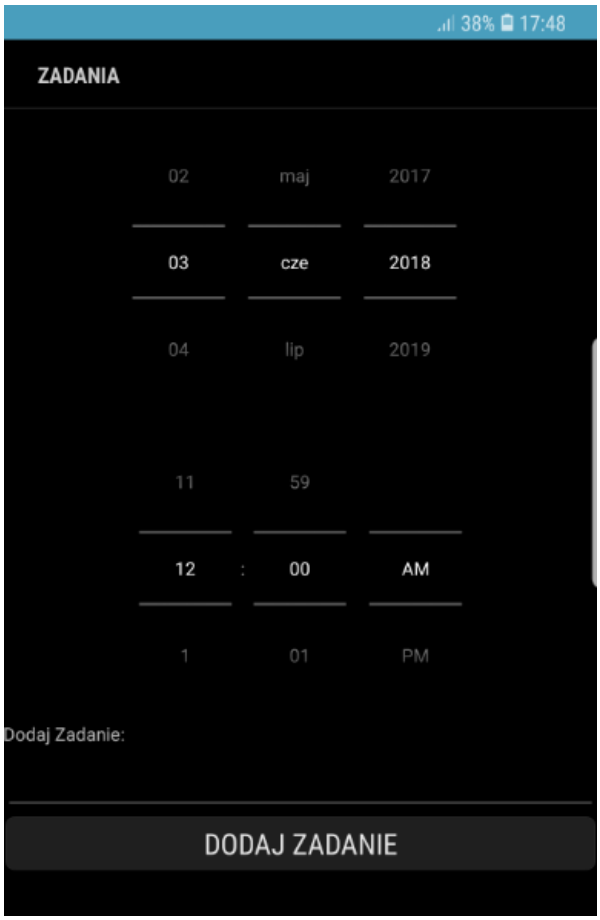
Po zaimplementowaniu klas odpowiedzialnych za obsługę serwisów zdefiniowanych w części wspólnej, należy je zarejestrować w klasie `Setup` (listing 62).

Listing 62 Zawartość klasy `Setup`

```
public class Setup : MvxAndroidSetup
{
        public Setup(Context applicationContext) : base(applicationContext)
        {
        }

        protected override IMvxApplication CreateApp()
        {
                return new Core.App();
        }

        protected override IMvxTrace CreateDebugTrace()
        {
                return new DebugTrace();
        }

        protected override void InitializePlatformServices()
        {
                base.InitializePlatformServices();
                Mvx.RegisterType<IModalScreenService,
                AndroidModalScreenService>();
                Mvx.RegisterType<IShareService, AndroidShareService>();
                Mvx.RegisterType<IScheduledNotificationService,
                AndroidScheduledNotificationService>();
        }
}
```

W projekcie należy zmodyfikować także kod istniejącego widoku `MainView`, którego kontrolki zostaną związane z własnościami zdefiniowanymi w modelu widoku znajdującym się w części wspólnej. Należy dodać kontrolkę `MvxDatePicker`, która

będzie służyła do wyboru daty. Zostanie ona związana z własnością `Date` zdefiniowaną w modelu widoku Z kolei kontrolka `MvxTimePicker` będzie zwracać godzinę. Powinna zostać związana z własnością `Time`. Kolejną kontrolką jest `TextView`, której atrybut `text` ustawiamy na„Dodaj Zadanie:”.



Rys.47 Utworzony widok za pomocą kodu *axml* z listingu 63.

Następna kontrolka, którą należy dodać to `EditText`. To pole tekstowe, które umożliwi wpisanie treści zadania. Jej atrybut `Text` zostanie związany z własnością `Work` modelu widoku. Ostatnią kontrolką jest przycisk (`Button`). Jego zdarzenie `Click` wiążemy z poleceniem `AddWorkCommand`. Kompletny kod AXML widoku został przedstawiony na listingu 63.

Listing 63 Zawartość widoku `MainView`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <MvvmCross.Binding.Droid.Views.MvxDatePicker
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:calendarViewShown="false"
        android:datePickerMode="spinner"
        local:MvxBind="Value Date"
        android:minWidth="25px"
        android:minHeight="25px" />
    <MvvmCross.Binding.Droid.Views.MvxTimePicker
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:minWidth="25px"
        android:minHeight="25px"
        local:MvxBind="Value Time"
        android:timePickerMode="spinner" />
    <TextView
        android:text="Dodaj Zadanie:"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:scrollHorizontally="false"
        android:inputType="textCapSentences|textMultiLine"
        local:MvxBind="Text Work" />
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Dodaj Zadanie"
        local:MvxBind="Click AddWorkCommand"
        android:textSize="20dp" />
</LinearLayout>
```

4.3.3 Aplikacja Zadania – projekt Uniwersal Windows Platform

Rozwijanie projektu dla UWP zaczynamy jak zwykle od instalacji pakietu startowego i jego konfiguracji projektu według wytycznych przedstawionych w rozdziale 3. W projekcie należy utworzyć dwa nowe foldery *NativeConverters* oraz *Services*. Do pierwszego folderu należy dodać klasę o nazwię `DateNativeConverter`.[9] Zostanie ona wykorzystana do konwersji dat używanych w części wspólnej, stworzonego w projkecie części wspólnej. Kod klasy został przedstawiony na listingu 64.

[9] W przypadku platformy UWP oficjalna dokumentacja MvvmCross zaleca otoczenie konwerterów z części wspólnej dodatkowymi klasami w warstwie widoku.

Listing 64 Zawartość widoku `DateNativeConverter`

```
public class DateNativeConverter : MvxNativeValueConverter<DateConverter>
{
}
```

Analogicznie jak w projekcie kolory, instancję tego konwetera należy umieścić w zasobach aplikacji definiowanych pliku *App.xaml* (listing 65).

Listing 65 Zawartość pliku `App.xaml`

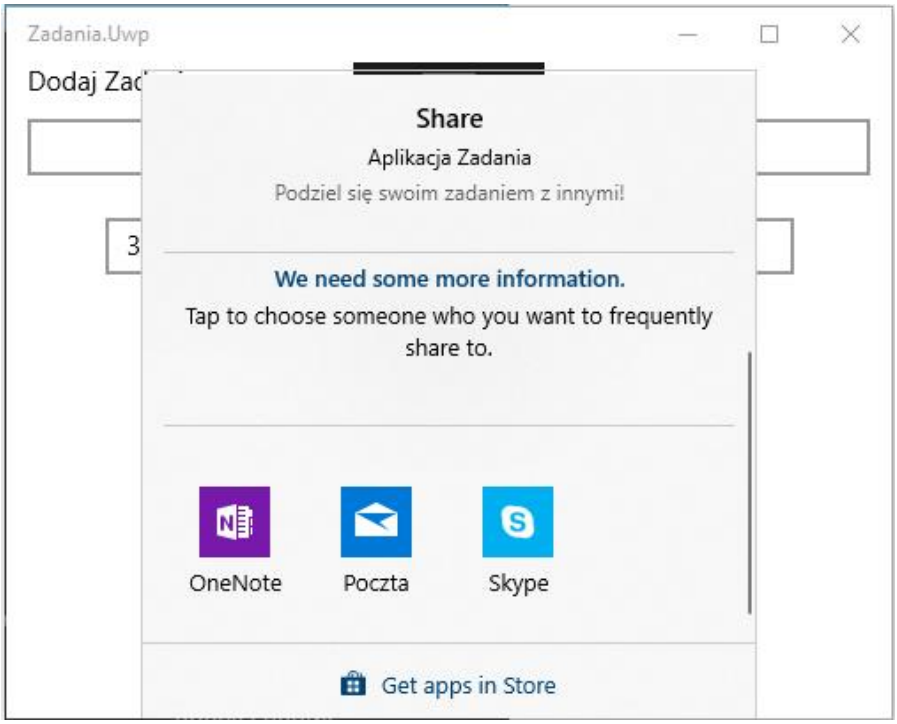
```
<Application
    x:Class="Zadania.Uwp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Zadania.Uwp"
    xmlns:converters="using:Zadania.Uwp.NativeConverters"
    RequestedTheme="Light">
    <Application.Resources>
        <converters:DateNativeConverter x:Key="NativeDateConverter" />
    </Application.Resources>
</Application>
```

Następnie należy dodać trzy nowe klasy do folderu *Services*: `UwpShareService`, `UwpScheduledNotificationService` oraz `UwpModalScreenService`. Pierwsza z nich powinna implementować metodę `Share` pochodzącą z interfejsu `IShareService`. Do udostępnienia treści zadań w systemie Windows 10, zostaną użyte klasy `DataTransferManager` oraz `DataRequest`.

Pierwsza z nich jest klasą statyczną, której można użyć do inicjowania operacji udostępniania. Aby z niej skorzystać należy najpierw wywołać metodę `GetForCurrentView`. Ta metoda zwraca obiekt typu `DataTransferManager`, który jest specyficzny dla aktywnego okna. Następnie trzeba dodać metodę do zdefiniowanego zdarzenia, które jest wywoływane za każdym razem, gdy zostaje odpalone okno udostępniania (po wywołaniu metody `ShowShareUI`). [33]

Druga klasa to `DataRequest`. Służy do przekazania informacji, które powinny zostać udostępnione innym aplikacjom. Obiekt ten zostaje przekazany podczas wywołania zdarzenia w klasie `DataTransferManager`. Po jego zarejestrowaniu należy dodać odpowiednie wartości; w przypadku tworzonej w tym podrozdziale aplikacji, zostaną

dodane wartości z własności `Title` oraz `Description`. Dodatkowo za pomocą metody `SetText` przekazany zostanie dodatkowy opis zawierający bieżącą datę na urządzeniu oraz treść zadania. [34]



Rys.48 Utworzony widok za pomocą kodu *axml* z listingu 63.

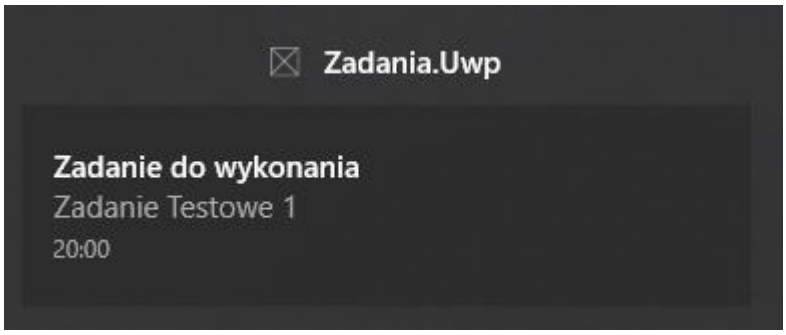
Na koniec z otrzymanego obiektu `DataRequest` metodą `GetDeferral` „wyciągamy" obiekt typu `DataRequestDeferral`. Po otrzymaniu tego obiektu, należy wywołać metodę `Complete`. Dzięki temu dane zostaną przekazane do mechanizmu odpowiedzialnego za ich udostępnienie innym aplikacjom i użytkownik zobaczy okno z wyborem aplikacji. [35] Kod klasy `UwpShareService` został przedstawiony na listingu 66.

Listing 66 Zawartość klasy `UwpShareService`

```
public class UwpShareService : IShareService
{
	private string _content;

	public void Share(string content)
	{
		_content = content;
		RegisterForShare();
		DataTransferManager.ShowShareUI();
	}
```

```
        private void RegisterForShare()
        {
              DataTransferManager dataTransferManager =
              DataTransferManager.GetForCurrentView();
              dataTransferManager.DataRequested += new
              TypedEventHandler<DataTransferManager,
              DataRequestedEventArgs>(this.ShareImageHandler);
        }

        private async void ShareImageHandler(DataTransferManager sender,
        DataRequestedEventArgs e)
        {
              DataRequest request = e.Request;
              request.Data.Properties.Title = "Aplikacja Zadania";
              request.Data.Properties.Description = "Podziel się swoim
              zadaniem z innymi!";
              request.Data.SetText($"Zadanie dodane dnia
              {DateTime.Now}\n{_content}");

              DataRequestDeferral deferral = request.GetDeferral();
              deferral.Complete();

        }
}
```

Następną klasą, jaką należy przygotować jest `UwpScheduledNotificationService`. Klasa ta musi implementować interfejs `IScheduledNotificationService`. Będzie wykorzystana do wywołania zaplanowych powiadomień systemowych w określonym przez użytkownika momencie. Format wyświetlania wiadomości w powiadomieniu określimy za pomocą XML (por. listing 67). W tym celu utworzymy nowy obiekt typu `XmlDocument`. Jego zawartość, a zarazem format wiadomości, zostaje określony za pomocą klasy `ToastNotificationManager` i jej metody `GetTemplateContent`. Do dostarczonego wzorca można dołożyć zestawy łańcuchów, które zostaną wyświetlone w powiadomieniu. Określamy także czy dodać dźwięk, który zostanie odtworzony po wyświetleniu notyfikacji w systemie (listing 67).



Rys.49 Powiadomienie stworzone za pomocą klasy `UwpScheduledNotificationService`

Klasa `ToastNotificationManager` jest odpowiedzialna za dostarczanie gotowych wzorców służących do wyświetlania powiadomień. Dostarcza również metodę `CreateToastNotifier`, dzięki której możemy utworzyć powiadomienie które zostanie wyświetlone w ustalonym czasie (metoda `AddToSchedule`). [36]

Aby zaplanować wyświetlenie powiadomienia w określonym czasie, należy dodatkowo skorzystać z obiektu `ScheduledToastNotification`. Obiekt ten jest odpowiedzialny za przechowywanie parametrów powiadomienia w systemie oraz dodatkowo przechowuje czas, po jakim powiadomienie zostanie wyświetlone (liczone w sekundach od bieżącej daty).[37]

Dlatego po zdefiniowaniu powiadomienia, należy wyliczyć całkowitą różnicę w sekundach między planowaną datą wyświetlenia powiadomienia a datą obecną, a następnie utworzyć obiekt `ScheduledToastNotification`, który zostanie przekazany do metody `AddToSchedule` klasy `ToastNotificationManager`. Po jej uruchomieniu system rejestruje nowe powiadomienie, które zostanie wyświetlone na standardowej belce powiadomień. Kod klasy został przedstawiony na listingu 67.

Listing 67 Zawartość klasy `UwpScheduledNotificationService`

```
public class UwpScheduledNotificationService :
IScheduledNotificationService
{
      public void AddNotification(DateTime startDate, string content)
      {
            Windows.Data.Xml.Dom.XmlDocument toastXml =
            ToastNotificationManager.GetTemplateContent(ToastTemplateType
            .ToastText02);
            Windows.Data.Xml.Dom.XmlNodeList toastNodeList =
            toastXml.GetElementsByTagName("text");

            toastNodeList.Item(0).AppendChild(toastXml.CreateTextNode("Za
            danie do wykonania\n"));

            toastNodeList.Item(1).AppendChild(toastXml.CreateTextNode(con
            tent));
            Windows.Data.Xml.Dom.IXmlNode toastNode =
            toastXml.SelectSingleNode("/toast");
            Windows.Data.Xml.Dom.XmlElement audio =
            toastXml.CreateElement("audio");
            audio.SetAttribute("src", "ms-
            winsoundevent:Notification.SMS");

            var diffrence = (startDate - DateTime.Now).TotalSeconds;

            DateTime EventDate = DateTime.Now.AddSeconds(diffrence);
            TimeSpan NotTime = EventDate.Subtract(DateTime.Now);
            DateTime dueTime = DateTime.Now.Add(NotTime);
```

```
            ScheduledToastNotification scheduledToast = new
            ScheduledToastNotification(toastXml, dueTime);

            ToastNotificationManager.CreateToastNotifier().AddToSchedule(
            scheduledToast);
        }
}
```

Ostatnią klasą w folderze *Services*, jaką należy zaimplementować jest `UwpModalScreenService`. Klasa ta implementuje interfejsu `IModalScreenService`, który wymusza obecność metody `ConfirmAdditionalAction`. Do wyświetlania okien modalnych zostanie wykorzystana klasa `MessageDialog`, za pomocą której w systemie Windows 10, także w edycji Mobile, można wyświetlić można okno z dwoma (systemy mobilne) lub trzema (systemy desktopowe) przyciskami (rysunek 50). [38]

Rys.50 Okno modalne stworzone za pomocą klasy `UwpModalScreenService`

Następnie w klasie `UwpModalScreenService` należy utworzyć dwa obiekty typu `UICommand`, które będą odpowiedzialne za wywoływanie odpowiednich akcji przekazywanych w metodzie `ConfirmAdditionalAction`. Każdy z nich przyjmuje tytuł, który zostanie wyświetlony jako opis opcji w oknie modalnym oraz wyrażenie lanmbda wywołujące metody, jakie zostaną wywołane po wybraniu jednej z opcji. Następnie należy utworzyć obiekt typu `MessageDialog`, który w konstruktorze przyjmuje dwa łańcuchy określające zawartość powiadomienia oraz jego tytuł. Do tego obiektu należy „przypiąć” stworzone polecenia, dodając je do listy o nazwie `Commands`, oraz ustawiając odpowiednie indeksy według których zostaną one wyświetlone. Kod metody należy zakończyć wywołaniem metody `ShowAsync` na obiekcie `MessageDialog`, co spowoduje faktyczne wyświetelenie okna modalnego (listing 68).

Listing 68 Zawartość klasy `UwpModalScreenService`

```
public class UwpModalScreenService : IModalScreenService
{
      public void ConfirmAdditionalAction(string title, string content,
      Action confirmAction, Action cancelAction)
      {
            var yesCommand = new UICommand("Tak",
            cmd => { confirmAction(); });
            var noCommand = new UICommand("Nie",
            cmd => { cancelAction(); });
            var dialog = new MessageDialog(content, title);
            dialog.Options = MessageDialogOptions.None;

            dialog.Commands.Add(yesCommand);
            dialog.DefaultCommandIndex = 0;
            dialog.CancelCommandIndex = 0;
            dialog.Commands.Add(noCommand);
            dialog.CancelCommandIndex = (uint)dialog.Commands.Count - 1;

            dialog.ShowAsync();
      }
}
```

Ostatnią rzeczą jaką należy zrobić jest dostosowanie zawartości pliku *MainPage.xaml*, który jest odpowiedzialny za wyświetlanie widoku całej aplikacji i wiązania z modelem widoku. Finalną wersję tego pliku prezentuje listing 69.

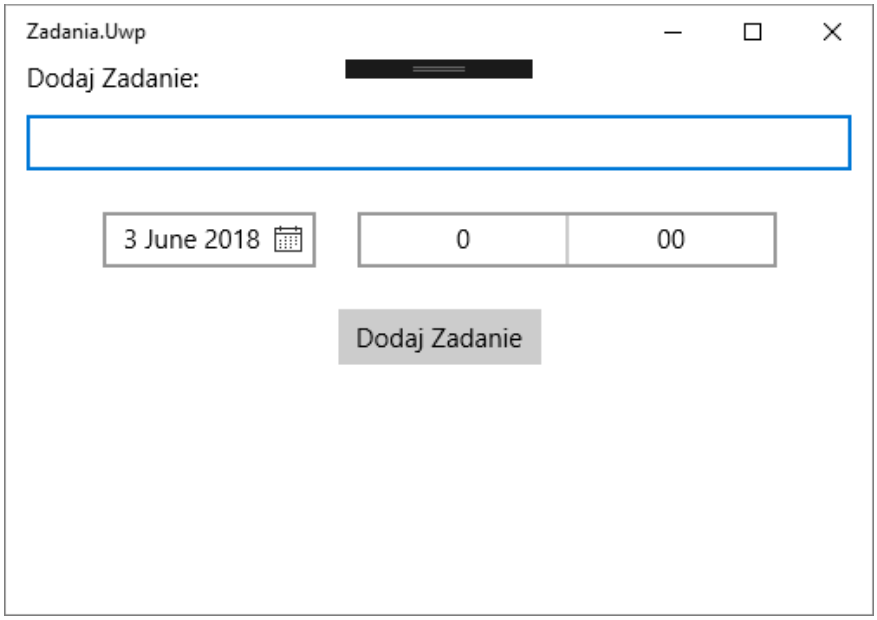
Listing 69 Zawartość klasy `UwpModalScreenService`

```
<local:BasePage
      x:Class="Zadania.Uwp.MainPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:local="using:Zadania.Uwp"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-
      compatibility/2006"
      mc:Ignorable="d">

      <StackPanel Background="{ThemeResource
      ApplicationPageBackgroundThemeBrush}">
        <TextBlock HorizontalAlignment="Left" Margin="12,0,12,0">
        Dodaj Zadanie:</TextBlock>
        <TextBox
            Text="{Binding Work,Mode=TwoWay}"
            VerticalAlignment="Center"
            Margin="12,12,12,12" />
        <StackPanel Orientation="Horizontal"
        VerticalAlignment="Center" HorizontalAlignment="Center">
            <CalendarDatePicker
                    DateFormat="{}{day.integer} {month.full} {year.full}"
                    Date="{Binding Date,Mode=TwoWay,
                    Converter={StaticResource NativeDateConverter}}"
                    Margin="12,12,12,12"/>
            <TimePicker
                ClockIdentifier="24HourClock"
```

```
                    Time="{Binding Time,Mode=TwoWay}"
                    Margin="12,12,12,12"
                    />
            </StackPanel>
            <StackPanel Orientation="Horizontal"
                VerticalAlignment="Center" HorizontalAlignment="Center">
                <Button
                    Margin="12,12,12,12"
                    Command="{Binding AddWorkCommand}"
                    Content="Dodaj Zadanie"/>
            </StackPanel>
        </StackPanel>
</local:BasePage>
```

Kontrolki zorganizowane są pojemnikiem `StackPanel`. Pierwszą kontrolką jest kontrolka `TextBlock`, która wyświetla tytuł „Dodaj Zadanie:”. Została ona umieszczona z lewej strony ekranu. Pod nią znajduje się pole edycyjne `TextBox`, którego atrybut `Text` został związany z własnością `Work` modelu widoku.

Pod nim znajduje się kolejny znacznik `StackPanel`, w którym została zmieniona domyślna wartość atrybutu `Orientation` na `Horizontal`, dzięki czemu kontrolki będą układały się jedna obok drugą poziomo, a nie pionowo. Wewnątrz tego panelu została umieszczona kontrolka `CalendarDatePicker`, której atrybut `Date`, został związany z własnością `Date` z modelu widoku. W tym wiązaniu wykorzystany został konwerter daty, zadeklarowany w pliku *App.xaml*.



Rys.51 Główny wygląd aplikacji określony w pliku *MainPage.xaml*

Drugą kontrolką jest `TimePicker`, który jest odpowiedzialny za wybór daty w formacie 24 godzinnym (atrybut `ClockIdentifier` ustawiony na `24HourClock`). Atrybut `Time`, został związany z własnością `Time` znajdującą się w modelu widoku. Całość została wyśrodkowana za pomocą atrybutów `VerticalAlignment` oraz `HorizontalAlignment` ustawionymi na wartość `Center`. Na sam końcu znajduje się przycisk (`Button`), którego atrybut `Command` został związany z poleceniem `AddWorkCommand`, które zostanie wywołane po jego naciśnięciu. Dodatkowo atrybut `Content` został ustawiony na „Dodaj Zadanie".

Ostatnią rzeczą jaką należy wykonać jest rejestracja utworzonych klas w klasię `Setup` z folderu *Services*, wywołując jej metodę `InitializePlatformServices`. Zaktualizowany kod klasy został przedstawiony na listingu 70. Po tym projekt jest gotowy do kompilacji i uruchomienia na dowolnym urządzeniu z systemem Windows 10.

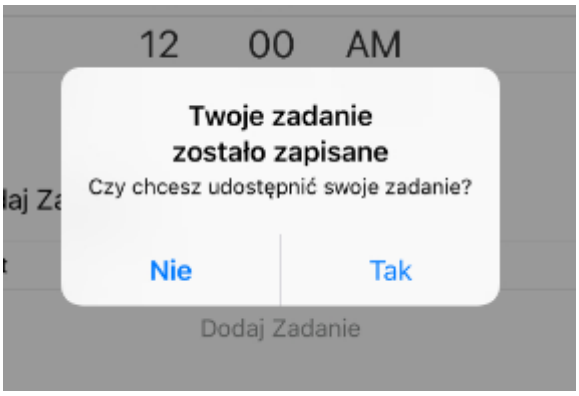
Listing 69 Zawartość klasy `UwpModalScreenService`

```
public class Setup : MvxWindowsSetup
{
      public Setup(Frame rootFrame) : base(rootFrame)
      {
      }

      protected override IMvxApplication CreateApp() => new Core.App();

      protected override MvxLogProviderType GetDefaultLogProviderType()
      => MvxLogProviderType.None;

      protected override void InitializePlatformServices()
      {
            base.InitializePlatformServices();
            Mvx.RegisterType<IModalScreenService,
            UwpModalScreenService>();
            Mvx.RegisterType<IShareService, UwpShareService>();
            Mvx.RegisterType<IScheduledNotificationService,
            UwpScheduledNotificationService>();
      }
}
```

4.3.4 Aplikacja Zadania – projekt iOS

Przejdźmy teraz do projektu dla platformy iOS. Po zainstalowaniu i skonfigurowaniu pakietu *MvvmCross.StarterPack*, należy utworzyć nowy folder o nazwie *Services*, a w nim trzy klasy: `IphoneModalScreenService`, `IphoneScheduledNotificationService` oraz `IphoneShareService`.

Pierwsza powinna implementować interfejs `IModalScreenService`, aby umożliwić wyświetlanie okna modalnego w aplikacji na platformę iOS. W tym celu należy wykorzystać klasę `UIAlertController`, która jest przeznaczona do wyświetlania powiadomień na tej platformie. Korzystając z tej klasy statycznej należy za pomocą metody `Create` utworzyć kontroler. Metoda ta przyjmuje trzy parametry: tytuł okna, wiadomość jaka ma zostać wyświetlona oraz rodzaj wyświetlanego okna: powiadomienie (ang. *alert*) lub arkusz działań (ang. *action sheet*).



Rys.52 Wygląd okna modalnego na platformie iOS

Następnie do tak utworzonego kontrolera, można za pomocą metody `AddAction` dodawać akcje, które zostaną reprezentowane za pomocą dodatkowych przycisków w oknie modalnym. Każdą z akcji reprezentuje obiekt klasy `UIAlertAction`. Jego instancję tworzymy za pomocą statycznej metody `Create` tej klasy. Metoda ta przyjmuje trzy parametry: tytuł przycisku, styl wyświetlonego przycisku (domyślny (ang. *default*), anulujący (ang. *cancel*) lub destrukcyjny (ang. *destructive*)) oraz delegację (typ `Action`), która zostanie wywołana po naciśnięciu danego przycisku. W naszym przypadku będą to akcje `confirm` i `cancel` przekazywane w metodzie `ConfirmAdditionalAction`.

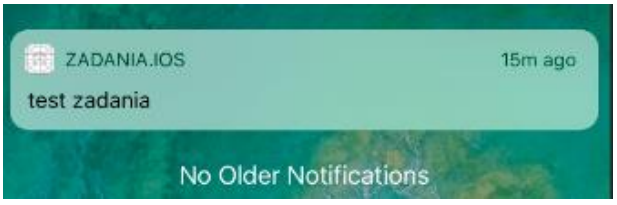
Po dodaniu akcji można utworzony kontroler użyć. W tym celu należy wykorzystać metodę `PresetViewController`, która jako parametry wejściowe przyjmuje kontroler, który chcemy wyświetlić, wartość logiczną określającą, czy wyświetlenie kontrolera ma być animowane oraz akcję, która może zostać wywołana po zamknięciu kontrolera. Dostęp do tej metody można uzyskać wykorzystując klasę `UIApplication`, która zawiera centralne właściwości administracyjne aplikacji działającej w systemie iOS. W tym przypadku należy za pośrednictwem właściwości `RootController` wywołać metodę `PresetViewController`, przekazując utworzony kontroler jako parametr.

Kod zaimplementowanej klasy `IphoneModalScreenService` został przedstawiony na listingu 70.

Listing 70 Zawartość klasy `IphoneModalScreenService`

```
public class IphoneModalScreenService : IModalScreenService
{
      public void ConfirmAdditionalAction(string title, string content,
      Action confirmAction, Action cancelAction)
      {
            var alertController = UIAlertController.Create(title,
            content, UIAlertControllerStyle.Alert);

            alertController.AddAction(UIAlertAction.Create("Tak",
            UIAlertActionStyle.Default, delegate
            {
                confirmAction();
            }));

            alertController.AddAction(UIAlertAction.Create("Nie",
            UIAlertActionStyle.Cancel, delegate
            {
                cancelAction();
            }));

            UIApplication.SharedApplication.KeyWindow.RootViewController.
            PresentViewController(alertController, true, null);}
}
```

Kolejna klasa to `IphoneScheduledNotificationService`. Implementuje interfejs `IScheduledNotificationService`. Do tworzenia zaplanowanych powiadomień w systemie iOS wykorzystany zostanie obiekt `UILocalNotification`. Po jego utworzeniu, możemy zdefiniować czas, po którym ma zostać odpalone powiadomienie (właściwość `FireDate`), tytuł powiadomienia (właściwość `AlertAction`), treść powiadomienia (właściwość `AlertBody`) i rodzaj dźwięku towarzyszącego powiadomieniu (właściwość `SoundName`).



Rys.53 Wygląd powiadomień w systemie iOS

Czas po którym ma zostać wykonane zadanie jest określony za pomocą liczby sekund pomiędzy obecną godziną a godziną wybraną przez użytkownika. Liczbę tą

obliczamy za pomocą klasy NSDate i jego statycznej metody FromTimeIntervalSinceNow Uzyskaną wartość przypisujemy właściwości FireDate obiektu powiadomienia. Po skonfigurowaniu obiektu powiadomienia, należy dodać go do kolejki przechowywanej przez system iOS za pomocą metody ScheduleLocalNotifcation udostępnianej przez statyczną własość UIApplication.SharedApplication. Pełen kod zmienianej klasy został przedstawiony na listingu 71.

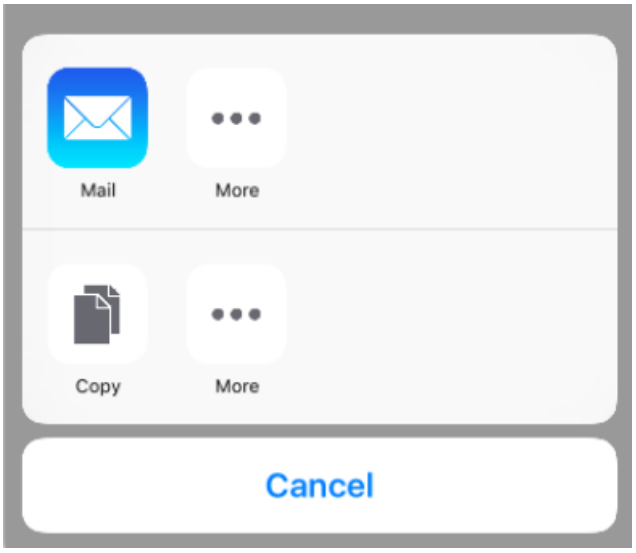
Listing 71 Zawartość klasy IphoneScheduledNotificationService

```
public class IphoneScheduledNotificationService :
IScheduledNotificationService
{
      public void AddNotification(DateTime startDate, string content)
      {
            var secondsDiffrence = (startDate -
            DateTime.Now).TotalSeconds;
            var notification = new UILocalNotification();

            notification.FireDate =
            NSDate.FromTimeIntervalSinceNow(secondsDiffrence);
            notification.AlertAction = "Twoje Zadanie!";
            notification.AlertBody = content;
            notification.ApplicationIconBadgeNumber = 1;
            notification.SoundName =
            UILocalNotification.DefaultSoundName;

            UIApplication.SharedApplication.ScheduleLocalNotification(not
            ification);
      }
}
```

Ostatnim serwisem jest IphoneShareService, który implementuje interfejs IShareService. Do jego zdefiniowana zostanie wykorzystana klasa UIActivityViewController, do której można przekazać listę obiektów, jakie można udostępnić innym aplikacjom. Każdy obiekt, który chcemy udostępnić musi zostać przekonwertowany do typu NSObject. Po przekonwertowaniu obiektów za pomocą metody FromObject oraz utworzeniu tablicy, która będzie wszystkie je przechowywać, należy przekazać je do konstruktora klasy UIActivityViewController. Tak utworzony kontroler, należy wyświetlić w taki sam sposób, jak zostały wyświetlane okna modalne, czyli wykorzystując metodę PresenViewController znajdującą się w klasie UIApplcation. Kod zaimplementowanej klasy został przedstawiony na listingu 72.

Listing 72 Zawartość klasy `IphoneShareService`

```
public class IphoneShareService : IShareService
{
	public void Share(string content)
	{
		var text = NSObject.FromObject(content);
		var items = new[] { text };
		var controller = new UIActivityViewController(items, null);

		UIApplication.SharedApplication.KeyWindow.RootViewController.
		PresentViewController(controller, true, null);
	}
}
```



Rys.54 Wygląd okna pozwalającego na udostępnianie zadania

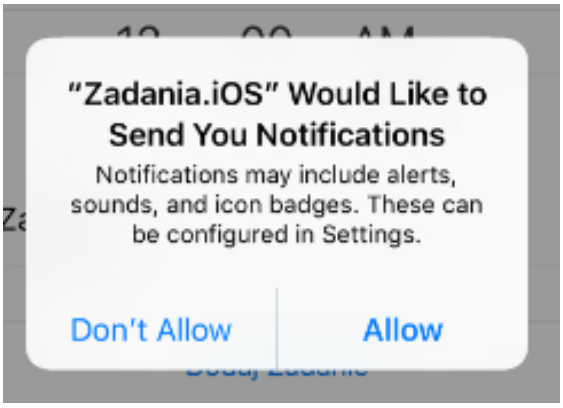
Po przygotowaniu serwisów, należy je zarejestrować w klasie `Setup`, przeciążając metodę `InitializePlatformServices` (listing 73).

Listing 73 Zawartość klasy `Setup`

```
public class Setup : MvxIosSetup
{
	public Setup(IMvxApplicationDelegate applicationDelegate, UIWindow
	window)
	: base(applicationDelegate, window)
	{

	}
	public Setup(IMvxApplicationDelegate applicationDelegate,
	IMvxIosViewPresenter presenter)
	: base(applicationDelegate, presenter)
	{
	}

	protected override void InitializePlatformServices()
	{
		base.InitializePlatformServices();
```

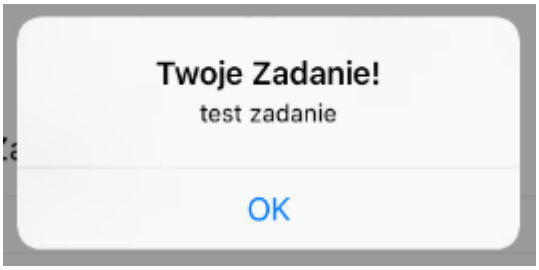
```
            Mvx.RegisterType<IModalScreenService,
            IphoneModalScreenService>();
            Mvx.RegisterType<IShareService, IphoneShareService>();
            Mvx.RegisterType<IScheduledNotificationService,
            IphoneScheduledNotificationService>();
        }

        protected override IMvxApplication CreateApp() => new Core.App();

        protected override IMvxTrace CreateDebugTrace() => new
        DebugTrace();
}
```

Aby móc wyświetlać powiadomienia w systemie iOS, należy uzyskać zgodę użytkownika aplikacji. W tym celu, należy przy pierwszym uruchomieniu aplikacji, zapytać o nią użytkownika. Można tego dokonać edytując metodę `FinishedLaunching` znajdującą się w klasie `AppDelegate`. Należy pobrać ustawienia powiadomień za pomocą metody `GetSettingsForTypes` znajdującej się w obiekcie `UIUserNotificationSettings`. Następnie należy uzyskać zezwolenie na wyświetlanie powiadomień i dźwięków, przekazując do metody typ wyliczeniowy `UIUserNotificationType` z typami `Alert`, `Badge` oraz `Sound`. Na koniec na obiekcie `UIApplication` należy wywołać metodę `RegisterUserNotificationSettings`, która wymusi potwierdzenie przez użytkownika praw dostępu.



Rys.55 Okno pojawiające się przy pierwszym uruchomieniu aplikacji

Klasa `AppDelegate` umozliwia także dodatkową obsługę powiadomień np. wyświetlając okno modalne, po otrzymaniu powiadomienia podczas używania aplikacji. Do obsługi powiadomień, należy przeciążyć metodę `ReceivedLocalNotification`.



Rys.56 Notyfikacja wyświetlona za pomocą klasy `AppDelegate`

W tej metodzie po raz kolejny zostanie utworzony kontroler typu `UIAlertController`, do którego zostaną przekazane wartości właściwości `AlertAction` i `AlertBody`, pochodzące z klasy `UILocalNotification`. Dodatkowo zostanie utworzona dodatkowa akcja z przyciskiem *OK*, która pozwoli potwierdzić otrzymanie notyfikacji. Kod zaktualizowanej klasy `AppDelegate` został przedstawiony na listingu 74.

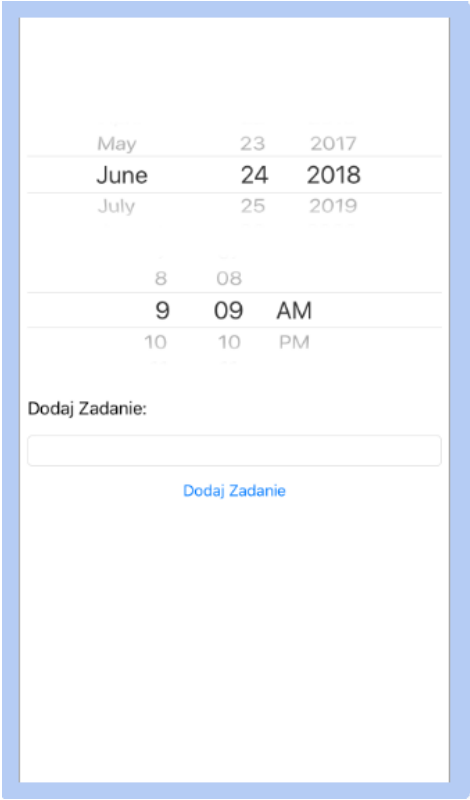
Listing 74 Zawartość klasy `AppDelegate`

```
[Register("AppDelegate")]
public partial class AppDelegate : MvxApplicationDelegate
{
      public override UIWindow Window { get; set; }

      public override void ReceivedLocalNotification(UIApplication
      application, UILocalNotification notification)
      {
            UIAlertController okayAlertController =
            UIAlertController.Create(notification.AlertAction,
            notification.AlertBody, UIAlertControllerStyle.Alert);

            okayAlertController.AddAction(UIAlertAction.Create("OK",
            UIAlertActionStyle.Default, null));

            Window.RootViewController.PresentViewController(okayAlertCont
            roller, true, null);
            UIApplication.SharedApplication
            .ApplicationIconBadgeNumber =  0;
      }


      public override bool FinishedLaunching(UIApplication application,
      NSDictionary launchOptions)
      {
            Window = new UIWindow(UIScreen.MainScreen.Bounds);

            var setup = new Setup(this, Window);
            setup.Initialize();
            var startup = Mvx.Resolve<IMvxAppStart>();
            startup.Start();
            Window.MakeKeyAndVisible();
```

```
            var notificationSettings =
            UIUserNotificationSettings.GetSettingsForTypes(
            UIUserNotificationType.Alert | UIUserNotificationType.Badge |
            UIUserNotificationType.Sound, null);

            application.RegisterUserNotificationSettings(notificationSett
            ings);

            return true;
        }
}
```

Ostatnią czynnością będzie utworzenie widoku oraz związanie wykorzystanych w nim kontrolek z własnościami modelu widoku. W widoku używamy przede wszystkim kontrolki `UIDatePicker`, która pozwoli użytkownikowi na wskazanie daty, w której ma być pokazane powiadomienie. Nadamy jej nazwę `datePicker`. Jej właściwość `Mode` ustawiamy na `Date`, dzięki czemu kontrolka będzie wyświetlać samą datę bez godziny. Godzinę będziemy ustawiali w osobnej kontrolce `UIDatePicker` o nazwie `timePicker`. Jej właściwość `Mode` ustawiamy na `Time`, dzięki czemu kontrolka będzie odpowiedzialna tylko za wyświetlanie i ustawianie godziny.

Kolejna wykorzystana kontrolka to `UILabel`, która będzie wyświetlać tekst: „Dodaj Zadanie”. Pod nią należy umieścić kontrolkę `UITextField` o nazwie `workText`, a na samym końcu kontrolkę `UIButton` o nazwie `addWorkButton`. Przykładowe rozmieszczenie kontrolek zostało przedstawione na rysunku 57.



Rys.57 Przykładowe rozmieszczenie kontrolek w widoku MainView

Dodane do widoku kontrolki należy następnie związać z modelem widoku. Można to zrobić w klasie `MainView`, a dokładnie w jej metodzie `ViewDidLoad` uruchamianej po załadowaniu widoku. Właściwość `Date` kontrolki `datePicker` powinna zostać związana z właściwością `Date` modelu widoku. Właściwość `Date` kontrolki `timePicker` zostanie związana z właściowścią `Time` modelu widoku. W tych wiązaniach użyty został konwerter `TimeConverter` utworzony w projekcie części wspólnej. Z kolei kontrolka `workText` została związana z właściwością `Work`, a przycisk `addWorkButton` został związany z komendą `AddWorkCommand` (listing 75).

Listing 75 Zawartość klasy `MainView`

```
[MvxRootPresentation(WrapInNavigationController = true)]
public partial class MainView : MvxViewController
{
	public MainView() : base("MainView", null)
	{

	}

	public override void ViewDidLoad()
	{
		base.ViewDidLoad();
```

```
            var set = this.CreateBindingSet<MainView,
            Core.ViewModels.MainViewModel>();
            set.Bind(datePicker).For(x => x.Date).To(vm => vm.Date);
            set.Bind(timePicker).For(x => x.Date)
                .To(vm => vm.Time)
                .WithConversion("Time");
            set.Bind(workText).To(x => x.Work);
            set.Bind(addWorkButton).To(x => x.AddWorkCommand);
            set.Apply();
        }
}
```

To kończy przygotowania - aplikacja jest gotowa do uruchomienia na urządzeniu z systemem iOS. Należy zwrócić uwagę uruchamiając aplikację iOS na emulatorze. Domyślnie zainstalowane emulatory nie mają wbudowanych aplikacji, które mogą odebrać udostępniany tekst (zadanie). Nie oznacza to jednak że funkcja udostępniania nie działa – na fizycznym urządzeniu, na którym jest zainstlowany i skonfigurowany np. klient poczty elektronicznej, metoda udostępniająca wyświetli tę aplikacje (rysunek 54).

5. Podsumowanie

Celem niniejszej pracy magisterskiej było przygotowanie tutorialu dotyczącego tworzenia multiplatformowych aplikacji w języku programowania C# z wykorzystaniem technologii Xamarin oraz frameworka MvvmCross. W tym celu stworzyłem i opisałem trzy projekty. Każdy z nich był pretekstem do omówienia zagadnień, które często spotykane są w codziennej praktyce programisty aplikacji mobilnych.

Zdaję sobie oczywiście sprawę, że tutorial o takim rozmiarze nie może być kompletny, ale starałem się, żeby ułatwiał rozpoczęcie pracy z frameworkiem MvvmCross oraz technologią Xamarin. Wydaje mi się, że ten cel został osiągnięty, czytelnik tej pracy po przeczytaniu całości, powinien móc bez problemu stworzyć jedną aplikację na wiele platform mniejszym nakładem pracy, niż podczas używania „natywnych" języków programowania dla każdej platformy osobno.

6. Literatura

[1] Xamarin https://www.xamarin.com/ (stan z dnia 01.06.2018)

[2] Xamarin iBeacons https://blog.xamarin.com/use-ibeacons-in-android-with-c/ (stan z dnia 01.06.2018)

[3] Proces budowania aplikacji https://xamarinhelp.com/xamarin-android-aot-works/ (stan z dnia 01.06.2018)

[4] Szczegóły Xamarin https://www.visualstudio.com/pl/xamarin/ (stan z dnia 01.06.2018)

[5] Xamarin Forms https://docs.microsoft.com/pl-pl/xamarin/xamarin-forms/get-started/introduction-to-xamarin-forms (stan z dnia 01.06.2018)

[6] Architektura Mvvm https://msdn.microsoft.com/pl-pl/library/wprowadzenie-do-wzorca-projektowego-model-view-viewmodel-na-przykladzie-aplikacji-wpf.aspx (stan z dnia 01.06.2018)

[7] Korzyści z Mvvm https://www.tutorialspoint.com/mvvm/mvvm_advantages.htm (stan z dnia 01.06.2018)

[8] Informacje ogólne o frameworku mvvmcross https://www.mvvmcross.com/ (stan z dnia 01.06.2018)

[9] Dostarczane funkcje przez framework mvvmcross https://www.mvvmcross.com/documentation/getting-started/getting-started (stan z dnia 01.06.2018)

[10] Dokumentacja SQLite https://www.sqlite.org/about.html (stan z dnia 01.06.2018)

[11] Informacje SQLite http://www.sqlitetutorial.net/what-is-sqlite/ (stan z dnia 01.06.2018)

[12] Projekt Shared Project https://docs.microsoft.com/pl-pl/xamarin/cross-platform/app-fundamentals/shared-projects?tabs=vsmac (stan z dnia 01.06.2018)

[13] Projekt Portable Class Library https://docs.microsoft.com/pl-pl/xamarin/cross-platform/app-fundamentals/pcl?tabs=vsmac (stan z dnia 01.06.2018)

[14] Porównanie projektów shared project oraz portable class library https://medium.com/@daRochaPires/xamarin-pcl-vs-shared-project-a838806d5cc6 (stan z dnia 01.06.2018)

[15] Kompatybilność API w projekcie PCL https://docs.microsoft.com/pl-pl/dotnet/standard/cross-platform/cross-platform-development-with-the-portable-class-library/ (stan z dnia 01.06.2018)

[16] Wzorzec Service Locator https://devstyle.pl/2016/02/11/antywzrorzec-service-locator/ (stan z dnia 01.06.2018)

[17] IOC, wzorzec wstrzykiwania zależności wady i zalety oraz rodzaje https://4programmers.net/In%C5%BCynieria_oprogramowania/Wstrzykiwanie_zale%C5%BCno%C5%9Bci/ (stan z dnia 01.06.2018)

[18] Opis klasy konfiguracyjnej projektu części wspólnej https://www.mvvmcross.com/documentation/advanced/customizing-using-App-and-Setup (stan z dnia 01.06.2018)

[19] Cykl życia modeli widoku oraz kolejność wykonywania zdefiniowanych metod https://www.mvvmcross.com/documentation/fundamentals/viewmodel-lifecycle/

[20] Aktywności platformy Android w frameworku MvvmCross https://www.mvvmcross.com/documentation/tutorials/tipcalc/a-xamarinandroid-ui-project?scroll=1864.4444580078125 (stan z dnia 01.06.2018)

[21] Satya Komatineni, Dave Maclean, Sayed Hashimi, *Android 3 tworzenie aplikacji*, (2012) , 91-101

[22] Satya Komatineni, Dave Maclean, Sayed Hashimi, *Android 3 tworzenie aplikacji*, (2012) , 69

[23] Wiązanie danych w frameworku MvvmCross (rodzaje oraz typy definiowania) https://www.mvvmcross.com/documentation/fundamentals/data-binding (stan z dnia 01.06.2018)

[24] Jesse Liberty, Jon Galloway, Philip Japikse, Jonathan Hartwell, *Windows 10 Development with XAML and C# 7 (Second Edition)*, (2018) , 33-34

[25] Sterowanie cyklem życia aplikacji Xamarin IOS https://developer.xamarin.com/api/type/MonoTouch.UIKit.UIApplicationDelegate/ (stan z dnia 01.06.2018)

[26] Nadawanie uprawnień aplikacji Xamarin iOS https://docs.microsoft.com/pl-pl/xamarin/ios/deploy-test/provisioning/entitlements?tabs=vsmac (stan z dnia 01.06.2018)

[27] Dokumentacja wtyczki Color frameworka MvvmCross https://www.mvvmcross.com/documentation/plugins/color (stan z dnia 01.06.2018)

[28] Dokumentacja biblioteki odpowiedzialnej za obsługę bazy SQLite w środowisku .NET https://github.com/oysteinkrog/SQLite.Net-PCL (stan z dnia 01.06.2018)

[29] Dokumentacja obsługi nawigacji w projektach z frameworkiem MvvmCross https://www.mvvmcross.com/documentation/fundamentals/navigation (stan z dnia 01.06.2018)

[30] Intencje Android https://developer.android.com/reference/android/content/Intent

[31] BroadcastReceiver Android https://www.google.pl/search?q=BroadcastReceiver&oq=BroadcastReceiver&aqs=chrome..69i57j0l5.182j0j7&sourceid=chrome&ie=UTF-8 (stan z dnia 01.06.2018)

[32] PendingIntent Android https://developer.android.com/reference/android/app/PendingIntent (stan z dnia 01.06.2018)

[33] DataTransferManager Uniwersal Windows Platform https://docs.microsoft.com/en-us/uwp/api/windows.applicationmodel.datatransfer.datatransfermanager (stan z dnia 01.06.2018)

[34] DataRequest Uniwersal Windows Platform https://docs.microsoft.com/en-us/uwp/api/windows.applicationmodel.datatransfer.datarequest (stan z dnia 01.06.2018)

[35] DataRequestDeferral Uniwersal Windows Platform https://docs.microsoft.com/en-us/uwp/api/windows.applicationmodel.datatransfer.datarequest.getdeferral (stan z dnia 01.06.2018)

[36] ToastNotificationManager Uniwersal Windows Platform https://docs.microsoft.com/en-us/uwp/api/windows.ui.notifications.toastnotificationmanager (stan z dnia 01.06.2018)

[37] ScheduledToastNotification Uniwersal Windows Platform https://docs.microsoft.com/en-us/uwp/api/windows.ui.notifications.scheduledtoastnotification (stan z dnia 01.06.2018)

[38] MessageDialog Uniwersal Windows Platform https://docs.microsoft.com/en-us/uwp/api/windows.ui.popups.messagedialog (stan z dnia 01.06.2018)