Krzysztof Ferenc

©2016

Projekt aplikacji MULTIPLATFORMOWej DO SYNCHRONIZACJI PLIKÓW MIĘDZY URZĄDZENIAMI

# Cel

Celem projektu jest stworzenie schematu aplikacji multiplatformowej do synchronizacji danych między wieloma urządzeniami, w tym omówienie i porównanie technologii, które można wykorzystać do wykonania tego zadania. Projekt będzie teoretycznym schematem na podstawie którego będzie mogła zostać wykonana część praktyczna.

Kolejną częścią będzie wdrożenie, czyli próba zastosowania zaprezentowanego projektu w postaci przykładowego, gotowego do wykorzystania kodu aplikacji wraz z opisem poszczególnych elementów i ich zadań. Wynikiem będzie implementacja chmury, która będzie mogła posłużyć jako odniesienie do stworzenia pełnej i funkcjonalnej wersji aplikacji.

# Projekt

## Założenia

Głównym założeniem projektu jest bezpieczeństwo danych użytkownika, który sam decyduje o tym w jaki sposób chce przechowywać je fizycznie. Dane nie są tym samym przechowywane przez firmy zewnętrzne bez żadnej kontroli nad ich bezpieczeństwem i poufnością. Równie ważna jest także możliwość synchronizacji danych na każdym z popularnych systemów operacyjnych na rynku takich jak Windows 10/Windows 10 Mobile, Windows 8.1, Windows Phone 8.1, Android i iOS. Wszystkie założenia opisane są w tabeli poniżej.

|  |  |
| --- | --- |
| Założenie | Opis |
| Jasny podział na projekty | Cała solucja (inaczej pakiet) będzie się składała z kilku pomniejszych projektów, które będą tworzyć dwie główne części: usługę serwera i aplikację klienta |
| Bezpieczeństwo danych | Dane są przechowywane przez samego użytkownika na własnym serwerze |
| Bezpieczeństwo komunikacji | Synchronizacja danych domyślnie odbywa się jedynie w obrębie sieci lokalnej |
| Multiplatformowość | Aplikacja klienta dostępna jest na najpopularniejszych platformach mobilnych i desktopowych |
| Łatwość wdrożenia | Pakiet ma być prosty do instalacji i łatwy w użyciu, tak aby średniozaawansowany użytkownik był w stanie go użyć |
| Synchronizacja | Oprócz możliwości przesyłania plików, aplikacja może automatycznie zbierać informacje dotyczące potrzeby wysłania lub pobrania plików poprzez komunikację z serwerem |

Tabela 1: Założenia projektu

### Schemat logiczny

#### Warstwy abstrakcji

Ponieważ głównym celem aplikacji jest synchronizacja plików, niezbędny jest wachlarz funkcji zaimplementowanych na różnych warstwach abstrakcji programistycznej. Warstwy abstrakcji nie są ściśle powiązane z miejscem implementacji konkretnej metody, są jedynie logicznym elementem pomagającym projektować oprogramowanie tak, aby było łatwe do zrozumienia i dalszego rozwijania.

Funkcjonalność pakietu można podzielić na 4 warstwy abstrakcji: podstawowa (warstwa framework'a), komunikacyjna (warstwa komunikacji), implementacyjna (warstwa aplikacji) i ostatnia – widok (warstwa interfejsu użytkownika).

Tabela 2: Warstwy abstrakcji aplikacji

Każdą z wymienionych typów funkcjonalności należy zaprojektować i wdrożyć oddzielnie. Aby to zrobić niezbędne jest porównanie dostępnych framework'ów i bibliotek, które nie tylko muszą spełniać swoją rolę osobno, ale także mieć fizyczną możliwość współgrania ze sobą.

#### Topologia sieci

Przy projektowaniu aplikacji działającej w sieci niezbędne jest określenie w jaki sposób poszczególne aplikacje będą się ze sobą komunikować. Topologie sieci określają model i sposób połączenia urządzeń bądź też aplikacji. W przypadku niniejszego projektu na jednym urządzeniu będzie działała jedynie jedna logiczna instancja aplikacji, a także urządzenia te będą połączone ze sobą przez sieć lokalną, co umożliwia założenie że wybrana topologia fizyczna sieci nie jest ważna.

Niezbędna jednak jest określnie jaką logiczną topologię będzie wykorzystywał projekt do synchronizacji danych między urządzeniami. Dla potrzeb projektu do wyboru będą dwa typy połączenia. Ich zalety i wady przedstawione są w tabeli.

|  |  |  |  |
| --- | --- | --- | --- |
| Topologia | Opis | Zalety | Wady |
| Gwiazda | Urządzenia podłączone są do jednego, centralnego koncentratora pełniącego rolę serwera danych | * Posiada koncentrator, który jest centralnym punktem referencyjnym dla pozostałych urządzeń w sieci * Łatwa w implementacji | * Wrażliwa na awarię serwera * Wymaga do działania centralnego serwera z dodatkowym oprogramowaniem |
| Siatka | Każde urządzenie jest połączone z innymi; tworzy to nadmiarową ilość połączeń między poszczególnymi elementami sieci, ale zapewnia decentralizację sieci | * Niewrażliwość na awarie dowolnego elementu sieci * Pozwala na przesyłanie danych bezpośrednio do urządzenia docelowego | * Czasochłonna w implementacji * Zużywa dużo transferu do komunikacji * Potencjalnie stwarza problemy z zachowaniem aktualnej wersji plików * Nie posiada punktu odniesienia; wszystkie urządzenia mają równe uprawnienia w hierarchii |

Tabela 3: Topologie sieci dla projektu

W ramach projektu wykorzystana zostanie topologia gwiazdy z jednym centralnym serwerem, który umożliwi połączenie urządzeń będących klientami zainstalowanymi na oddzielnych urządzeniach.

Podsumowując zatem – pakiet będzie dzielił się na dwa logiczne elementy, z czego jeden będzie instancją serwera działającą na fizycznej maszynie pełniącej rolę serwera danych. Drugi element będzie wieloma instancjami aplikacji na urządzeniach użytkownika, które będą się łączyć z koncentratorem w celu zainicjalizowania procesu synchronizacji.

### Technologie

#### Język

Przegląd dostępnych możliwości należy zacząć od wyboru języka programowania. Ponieważ w założeniach jednym z elementów była multiplatformowowość, nie może być on ekskluzywny jedynie dla niektórych platform bądź systemów. Ważnym aspektem jest również możliwość tworzenia wspólnego kodu, który będzie implementował funkcjonalność aplikacji dla wielu platform jednocześnie. Warto również zaznaczyć, że większość języków jest bezpośrednio powiązana z framework'iem, który go obsługuje (dotyczy języków dynamicznych). Tabela poniżej przedstawia ogólne porównanie technologii pasujących do wymagań.

|  |  |  |  |
| --- | --- | --- | --- |
| Język | Opis | Zalety | Wady |
| C/C++ | Jeden z popularniejszych języków programowania; najstarszy z branych tu pod uwagę; posiada silne wsparcie społeczności i duże grono wielbicieli. | * Umożliwia pisanie aplikacji na każdą platformę bez dodatkowych bibliotek * Język kompilowany do postaci binarnej; nie wymaga żadnego framework'a | * Brak możliwości tworzenia kodu współdzielonego dla różnych platform * Skromne biblioteki standardowe * Trudny w utrzymaniu; miejscami nieco archaiczny |
| Java | Zyskała ogromną popularność za sprawą szerokiego wachlarzu urządzeń z którymi współpracuje. | * Natywny język programowania na platformę Android * Pozwala na pisanie aplikacji na każdą platformę (wymaga zewnętrznych bibliotek) * Bogaty wybór bibliotek | * Do działania wymaga zasobożernej maszyny wirtualnej |
| C# | Jeden z języków wspieranych i silnie powiązanych z technologią .NET Framework. Stworzony i mocno promowany przez firmę Microsoft; od niedawna posiada otwarty kod źródłowy. | * Natywny język dla platformy Windows, Windows Mobile i Windows Phone * Bardzo rozbudowany framework * Umożliwia tworzenie aplikacji na każdą platformę (z użyciem zewnętrznej biblioteki) | * Trudny do opanowania na poziomie wyższym niż podstawowy |
| Python i Ruby | Jedne z popularniejszych języków dynamicznych, które do działania nie potrzebują zaawansowanego framework'a. Proste i dostępne na każdej platformie. | * Multiplatformowość z wykorzystaniem zewnętrznych bibliotek * Proste do nauki i łatwe do opanowania | * Brak zaawansowanych narzędzi do tworzenia usług sieciowych |
| JavaScript | Pierwotnie stworzony jako język interpretowany do zastosowań na stronach internetowych wykonywania po stronie przeglądarki. Z powodu swojej prostoty, a wyniku czego popularności; wykorzystywany do tworzenia aplikacji (głównie mobilnych). | * Natywne wsparcie dla praktycznie każdej platformy * Łatwy w użyciu * Biblioteka Node.js umożliwia tworzenie aplikacji serwerowych | * Mało wydajny * Nie jest nadaje się do tworzenia zaawansowanych aplikacji |

Tabela 4: Porównanie języków programowania

Przy wyborze odpowiedniej technologii niezbędna jest także wiedza, która pozwoli na wykorzystanie języka programowania i odpowiadającego mu framework’a (jeśli taki posiada). W świetle tych argumentów najrozsądniejszym wyborem jest język C# w najnowszej wersji 6 wspomagany przez szeroki wachlarz technologii udostępnianych w pakiecie .NET Framework. Wybór ten ma także inne podstawy związane z obsługiwanymi technologami, które zostaną przedstawione w kolejnych rozdziałach.

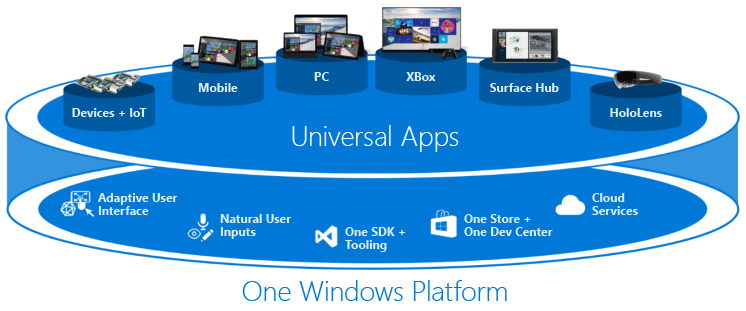
#### Framework

Pomimo iż język C# jest wspierany oficjalnie przede wszystkim przez Microsoft .NET Framework, który dostępny jest jedynie na platformę Windows, Windows Mobile i Windows Phone, istnieją alternatywy, dzięki którym możliwe jest używanie języka C# i większości jego funkcji na innych platformach. Porównanie różnych rozwiązań przedstawione jest w tabeli poniżej.

|  |  |  |
| --- | --- | --- |
| Framework | Opis | Wspierane platformy |
| .NET Framework | Pełna wersja pakietu .NET; pierwsze wydanie zostało udostępnione w 2002 roku przez firmę Microsoft z myślną o wykorzystaniu na platformie Windows | * Windows Desktop * Windows UWP * Windows Phone |
| .NET Core | Otwarta wersja pakietu .NET Framework wydana przez Microsoft w 2016 roku (wersja stabilna) na wolnej licencji MIT, charakteryzuje się wsparciem dla 3 różnych platform | * Windows * Linux (różne dystrybucje) * macOS |
| Mono | Otwarta implementacja pełnego pakietu .NET Framework dla 3 platform; pierwsza publiczna wersja powstała w 2004 roku; projekt jest sponsorowany przez Microsoft i jest częścią .NET Foundation | * Windows * Linux * OS X |
| Xamarin | Jest rozszerzeniem implementacji Mono dla najpopularniejszych platform mobilnych; w 2016 roku przejęte przez Microsoft razem z firmą o tej samej nazwie | * Windows UWP * Windows Phone * Android * iOS |

Tabela 5: Porównanie framework'ów

Należy zauważyć, że platforma UWP (Universal Windows Platform) nie jest ograniczona jedynie do urządzeń mobilnych. Aplikacje UWP działają na wachlarzu urządzeń z systemem Windows 10 (komputery i tablety), Windows 10 Mobile (smartphon’y), Windows 10 IoT (mikrokomputery, sprzętowe platformy developerskie) a także eksperymentalne urządzenia wirtualnej rzeczywistości jak Microsoft HoloLens.



Rysunek 1: Urządzenia wspierające platformę UWP

W przypadku projektu aplikacji do synchronizacji danych bardzo ważne jest wsparcie dla urządzeń mobilnych ponieważ to na nich chmura ma najwięcej zastosowań, poza tym urządzenia mobilne są trudniejsze do synchronizacji - posiadają mniejszy zestaw fizycznych interfejsów do przesyłania danych, a ich pamięć jest często mocno ograniczona i stanowi ułamek możliwości przechowywania danych w porównaniu do urządzeń stacjonarnych bądź też serwerów. Z tego powodu najrozsądniejsze jest wybranie technologii Xamarin, która wspiera zarówno aplikacje na komputery stacjonarne (Windows 10) jak i urządzenia mobilne z systemami Windows 10 Mobile, Android i iOS.

#### Komunikacja

Niezbędnym aspektem funkcjonowania pakietu będzie sposób łączenia się poszczególnych elementów w sieci. Jednym z aspektów branych pod uwagę są usługi sieciowe do obsługi warstwy komunikacji.

##### Usługi sieciowe

Zarówno wybrany wcześniej framework Mono jak i jego rozszerzenie – Xamarin oddają programiście kilka usług, dzięki którym może stworzyć własny sposób komunikacji wykorzystując do tego celu gotowe moduły. Porównanie tych elementów znajduje się w poniższej tabeli.

|  |  |  |
| --- | --- | --- |
| Nazwa usługi | Zalety | Wady |
| Windows Communication Foundation (WCF) | * Bardzo zaawansowany zestaw usług * Nadaje się do każdych zastosowań * Rozbudowana funkcjonalność związana z bezpieczeństwem usługi * Współpracuje z innymi platformami i usługami * Nie wymusza konkretnego protokołu komunikacji; bardzo elastyczny * Obsługuje zarówno komunikację jedno, jak i dwukierunkową * Mnogość sposobów hostowania usługi sieciowej | * Trudny do opanowania * Wymaga większej wiedzy przy projektowaniu aplikacji niż pozostałe usługi * Ze względu na swoją złożoność nie wszystkie funkcje są w pełni obsługiwane przez technologie Mono i Xamarin |
| ASP.NET Web Services (ASMX) | * Obsługuje komunikację jedno i dwukierunkową * Wbudowane funkcje zabezpieczeń * Wspierany na każdej platformie frameworka | * Obsługuje przesyłanie danych jedynie przez protokół HTTP |
| REST Services | * Prosty do projektowania i wdrażania * Niezależny od platformy i framework’a | * Brak natywnego wsparcia przez framework; wymaga zewnętrznych bibliotek * Obsługuje jedynie protokół HTTP * Obsługuje jedynie komunikację jednokierunkową * Brak wbudowanych funkcji związanych z zabezpieczeniami |

Tabela 6: Dostępne usługi komunikacji sieciowej

Z zestawienia wynika, że w przypadku znajomości technologii Windows Communication Foundation stanowi ona najlepszy środek do zaprojektowania komunikacji w projekcie.

##### Transport

Zaawansowana technologia WCF realizuje komunikację sieciową z wykorzystaniem protokołu SOAP (Simple Object Access Protocol), który z kolei umożliwia wybranie zarówno z pośród dostępnych sposobów serializacji danych jak i protokołu przenoszenia.

Niestety na chwilę aktualną z powodu braku pełnego wsparcia technologii WCF dla każdej platformy jedynym dostępnym protokołem przenoszenia możliwym do wykorzystania jest protokół HTTP realizowany z wykorzystaniem elementu „BasicHttpBinding”, który dokładniej zostanie omówiony w dalszej części pracy.

Oddzielnym problemem jest za to wybranie odpowiedniego formatu kodowania danych. Zalety i wady każdego rozwiązania przedstawione są w tabeli.

|  |  |  |  |
| --- | --- | --- | --- |
| Format | Przykład | Zalety | Wady |
| XML | <menu id="file" value="File">  <popup>  <menuitem value="New" onclick="CreateNewDoc()" />  <menuitem value="Open" onclick="OpenDoc()" />  <menuitem value="Close" onclick="CloseDoc()" />  </popup>  </menu> | * Dobry do transportu złożonych typów danych * Domyślny sposób kodowania; nie wymaga dodatkowej konfiguracji | * Zajmuje więcej miejsca; zwiększa ilość przesłanych danych |
| JSON | {"menu": {  "id": "file",  "value": "File",  "popup": {  "menuitem": [  {"value": "New", "onclick": "CreateNewDoc()"},  {"value": "Open", "onclick": "OpenDoc()"},  {"value": "Close", "onclick": "CloseDoc()"}  ]  }  )) | * Kompaktowy; zajmuje mniej miejsca * Dobry do przesyłania dużej ilości prostych typów danych * Łatwy do serializacji i deserializacji danych | * Nie nadaje się do przesyłania złożonych typów danych * Wymaga dodatkowej konfiguracji w celu użycia |

Tabela 7: Formaty transportu obsługiwane przez SOAP

Windows Communication Foundation domyślnie używa formatu XML do transportu danych. Ponieważ ich ilość będzie wyraźnie mniejsza od wielkości, a ponadto ich typy będą złożone – najrozsądniejszym wyborem będzie wykorzystanie tego formatu.

#### Implementacja

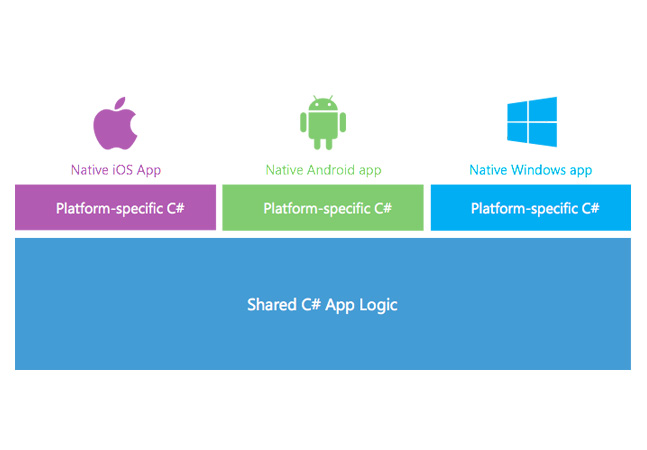
##### Wspóldzielenie kodu

Technologia Xamarin pozwala na pisanie aplikacji dla różnych platform. Niestety każda z nich posiada własne rozwiązania dotyczące tworzenia interfejsu użytkownika. Xamarin oddaje programiście różne narzędzia i możliwość wyboru, które z nich wykorzysta w swoim projekcie. Porównanie tych narzędzi wygląda następująco.

|  |  |  |  |
| --- | --- | --- | --- |
| Narzędzia | Opis | Zalety | Wady |
| * Xamarin Shared Project * Xamarin.iOS * Xamarin.Android | Solucja, w którym kod pisany jest we wspólnym projekcie; implementacja funkcji, których framework nie pokrywa odbywa się przez użycie znaczników warunkowych #IF, #ELSE, #ENDIF | * Pełna kontrola nad działaniem aplikacji * Dostęp do natywnych funkcji platformy * Pozwala na używanie natywnych bibliotek platformy | * Niezbędna jest pełna znajomość każdej platformy z osobna * Ilość czasu jaką jeden programista musiałby poświęcić na napisanie aplikacji tego projektu (zakładając że zna już każdą platformę docelową) przekracza pół roku * Każda implementacja funkcji musi być przeprowadzona oddzielnie dla każdej platformy, to sprawia że mogą pojawić się rozbieżności |
| * Portable Class Library (PCL) * Xamarin.iOS * Xamarin.Android | Podobnie jak wyżej, jednak kod wspólny jest zapisany w oddzielnym projekcie (PCL), a funkcje natywne dla platformy osobno | * Niemal pełna kontrola nad działaniem aplikacji * Dostęp do natywnych funkcji platformy * Możliwość korzystania z wielu bibliotek przygotowanych z myślą o wykorzystaniu z PCL | * Niezbędna jest znajomość części każdej platformy z osobna |
| * Portable Class Library (PCL) * Xamarin.Forms | Najłatwiejszy sposób na stworzenie aplikacji multiplatformowej; dobry przy używaniu głównie funkcji framework’a; jako jedyny pozwala na utworzenie wspólnego widoku dla każdej platformy | * Znajomość każdej platformy nie jest konieczna do implementacji większości funkcji aplikacji * Relatywnie szybki sposób na stworzenie aplikacji wieloplatformowej * Pozwala na zaprojektowanie wspólnego widoku przy użyciu zastępczych kontrolek, które w procesie kompilacji są tłumaczone na kontrolki natywne dla platformy | * Niewielka kontrola nad działaniem aplikacji na każdej z platform z osobna * Ograniczona możliwość korzystania z natywnych funkcji platformy |

Tabela 8: Rodzaje projektów Xamarin

Xamarin umożliwia wykorzystanie jednego projektu dla całej logiki aplikacji w projekcie współdzielonym (Shared Project) lub użycie dostępnej już w Mono czy .NET Framework biblioteki przenośnej (Portable Class Library). Obie opcje przynoszą jedynie rozwiązanie wspólnej implementacji modelu, jednak nadal wymagają od programisty zaprogramowania interfejsu na każdej platformie z osobna przy wykorzystaniu natywnych kontrolek i funkcji każdej z nich. Takie rozwiązanie jest dobrym wyjściem dla dużych zespołów programistycznych, które mogą sobie pozwolić na projektowanie aplikacji z częściowo współdzielonym kodem.

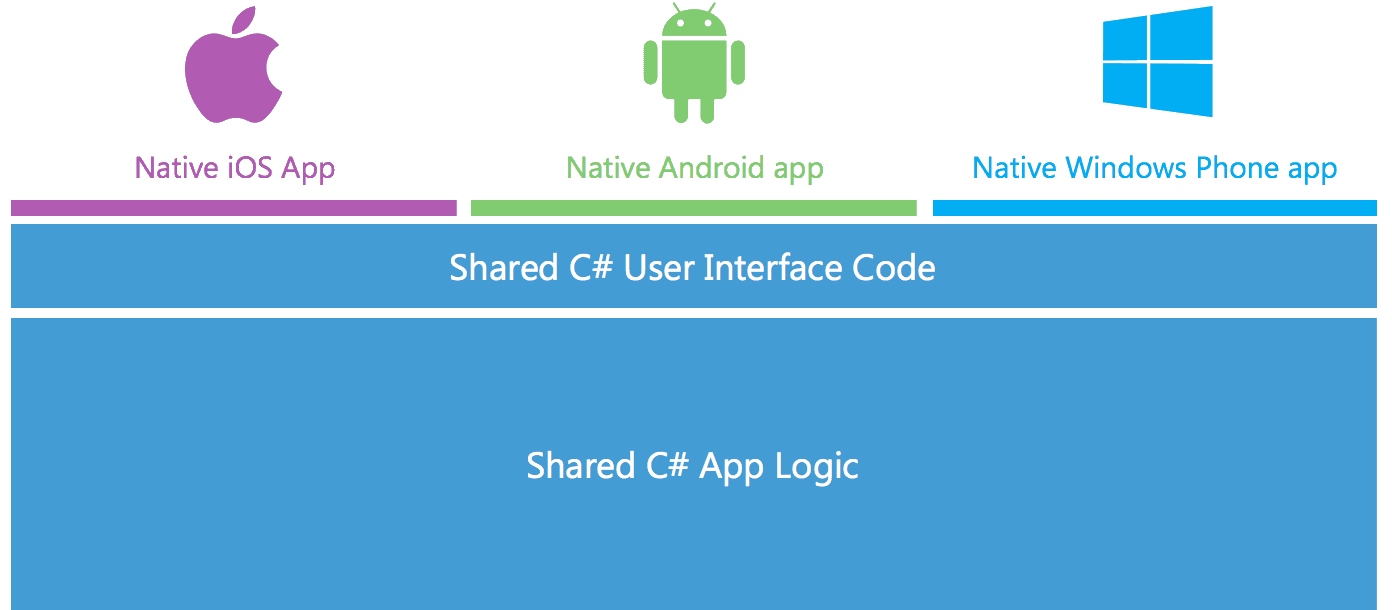


Rysunek 2: Schemat podziału kodu w projekcie aplikacji Xamarin

Pozostają jednak jeszcze programiści pracujący samodzielnie, którzy nie mogą sobie pozwolić na takie poświęcenie czasu lub nie chcą uczyć się każdej platformy oddzielnie. W takich przypadkach najczęściej programista decyduje się na wspieranie jednej platformy, jednak Xamarin opracował jeszcze jedną technologię, która pozwala na ograniczenie kodu natywnego do absolutnego minimum. To rozwiązanie zostanie opisane w kolejnej części.

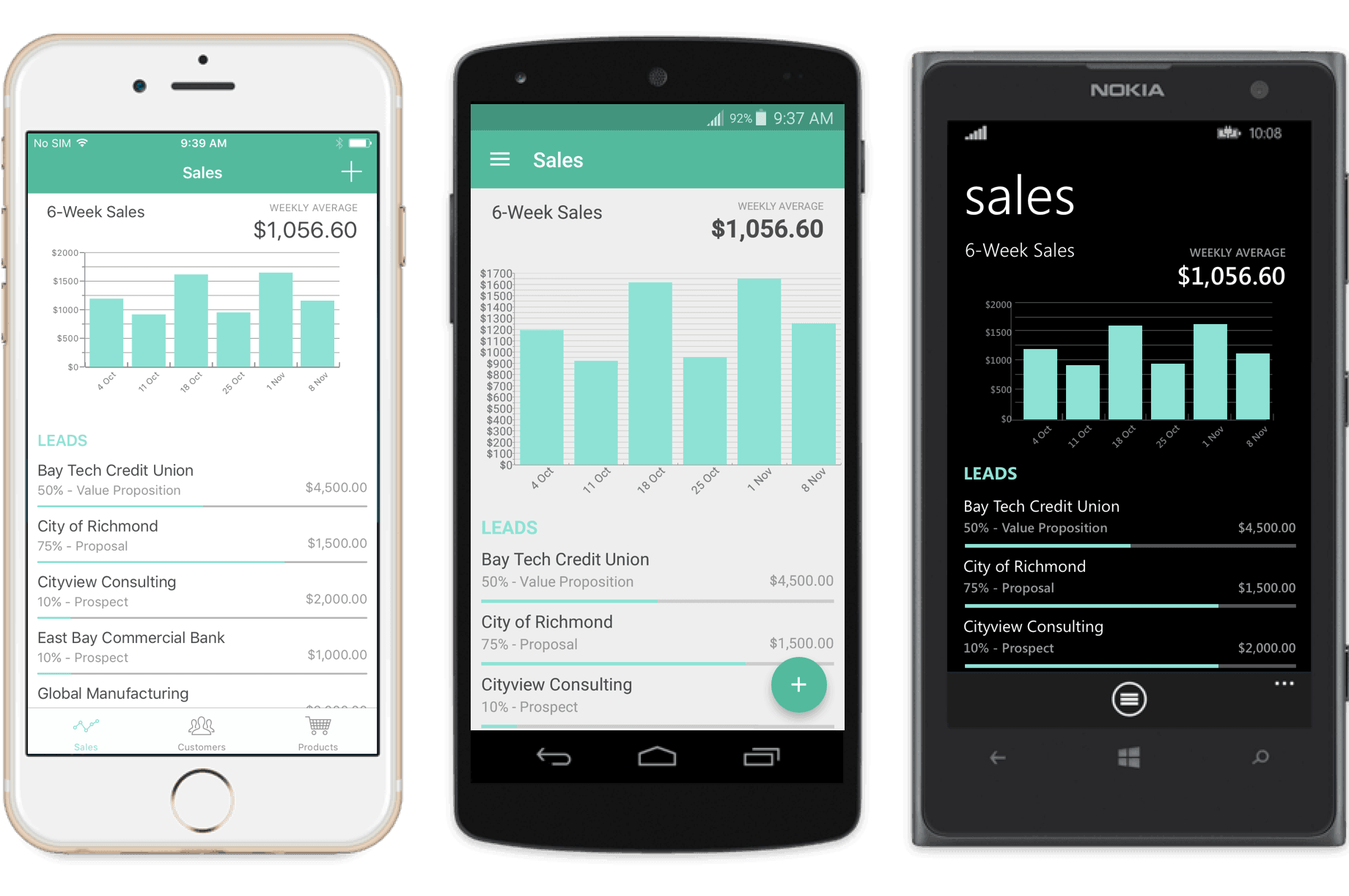
##### Współdzielenie widoku

Technologia Xamarin udostępnia użytkownikowi możliwość pisania współdzielonego kodu w języku C#, który zadziała na każdej wspieranej platformie sprzętowej. Problemem dla wielu programistów było jednak wdrożenie odpowiedniej warstwy widoku, która musiała być pisana oddzielnie dla każdej platformy i nie było sposobu aby można było to obejść.



Rysunek 3: Schemat kodu współdzielonego z wykorzystaniem Xamarin.Forms

Bez wątpienia w przypadku niniejszego projektu uwaga powinna zostać skupiona na ostatniej z wymienionych opcji współdzielenia kodu. Powodem takiej decyzji jest możliwość wykorzystania zarówno potężnego narzędzia jakim jest Portable Class Library, jak i dodatkowej, współdzielonej warstwy widoku realizowanej przez bibliotekę Xamarin.Forms. Pozwoli to zaoszczędzić dużo czasu i jednocześnie stworzyć wydajny kod, który zadziała na wielu platformach.



Rysunek 4: Współdzielony widok, który na każdej z platform używa innego zestawu natywnych kontrolek

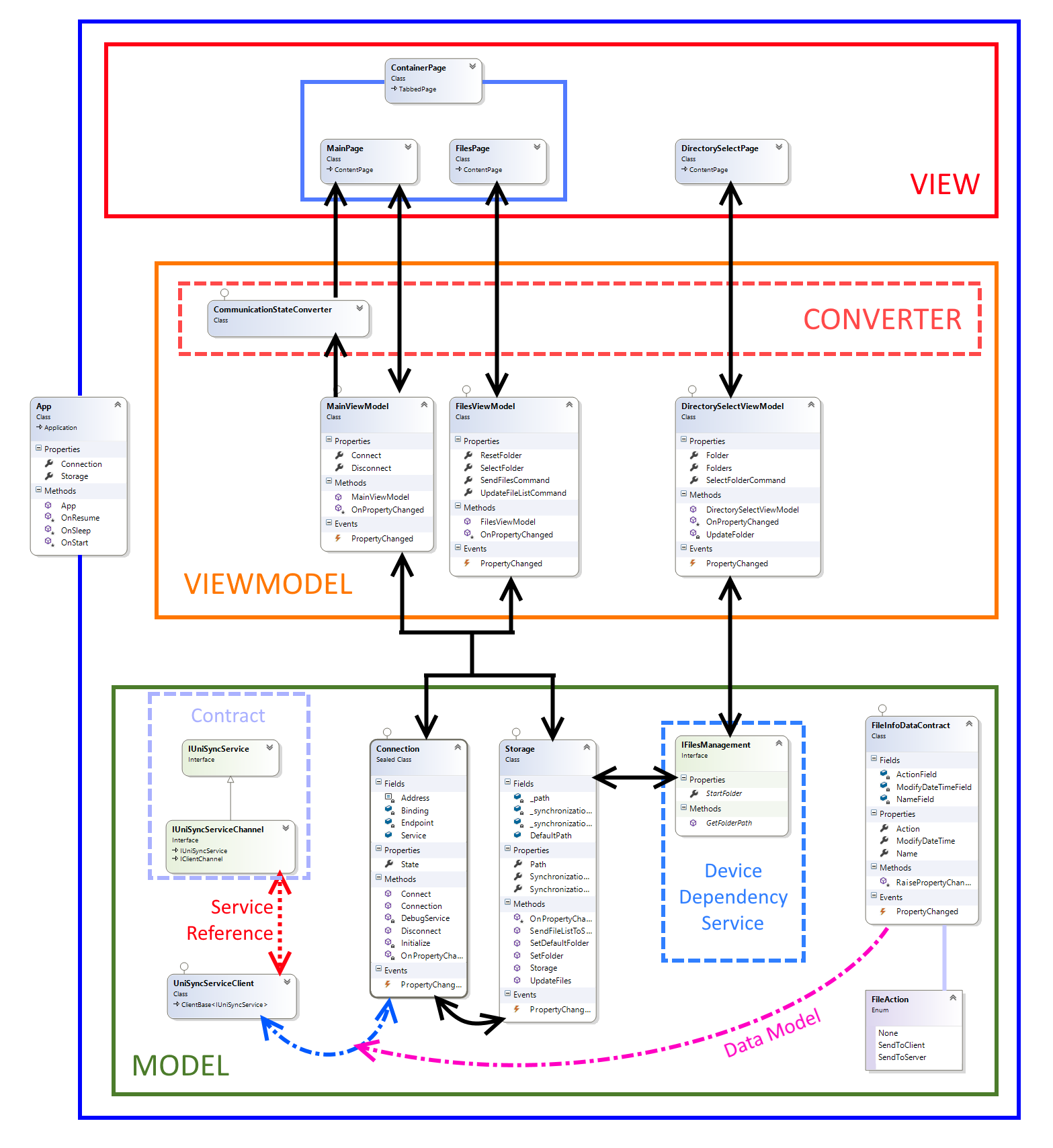
Działanie takiego interfejsu oparte jest o autorski zestaw kontrolek od firmy Xamarin, a także zestawu instrukcji, które są w stanie przetłumaczyć kod napisany zarówno w języku C#, jak i rozszerzonym formacie XML, rozwijanym przez firmę Microsoft pod nazwą XAML, który służy do lepszego oddzielenia warstwy widoku od modelu. Z wykorzystaniem języka XAML powiązana jest możliwość projektowania aplikacji zgodnie ze wzorcem projektowym MVVM (Model-View-ViewModel).

# Wdrożenie

## Klient

Aplikacja kliencka zgodnie ze wcześniejszymi założeniami ma być dostępna na wiele platform, dlatego konieczne było ograniczenie ilości kodu, który nie jest dostępny na każdej platformie. Biblioteki Xamarin.Forms razem z podstawowymi bibliotekami Mono pozwalają na skorzystanie z uniwersalnych elementów .NET Framework, który pierwotnie miał działać jedynie na platformie Windows.

Aplikacja posiada ścisły podział kodu pod względem spełnianej funkcji. Schemat podziału w aplikacji klienta na poszczególne elementy przedstawia się następująco:



Rysunek 5: Diagram klas i warstw w aplikacji klienta

### View

Widok składa się przede wszystkim z ContainerPage, który jest stroną-kontenerem dla MainPage i FilesPage. ContainerPage pozwala na łatwą nawigację między stronami podrzędnymi przy pomocy gestu przesunięcia w bok.

MainPage, jak sama nazwa wskazuje jest główną stroną widoczną po uruchomieniu aplikacji na urządzeniu i udostępnia akcję nawiązania połączenia z serwerem, a także wyświetla jego stan.

FilesPage jest drugą stroną, której celem jest zarządzanie plikami synchronizowanymi, jak i tymi dostępnymi do synchronizacji. Oprócz tego umożliwia ręczne wymuszenie przesyłania plików, a także odświeżenie ich listy.

DirectorySelectPage jest stroną dostępną jedynie na niektórych platformach. Technologia Xamarin.Forms nie udostępnia ujednoliconego sposobu na wybieranie katalogu synchronizacji, dlatego dla niektórych systemów (np. Android) niezbędne jest dodanie funkcji nawigacji po katalogach lokalnych wewnątrz aplikacji.

### ViewModel

ViewModel spaja widok z modelem, dzięki czemu akcje użytkownika mogą zostać odpowiednio zinterpretowane przez program i wykonane przy użyciu modelu aplikacji. Jego drugim zadaniem jest wizualna reprezentacja aktualnego stanu działania programu w sposób zrozumiały dla użytkownika.

MainViewModel jest bezpośrednim kontrolerem MainPage i odpowiada jedynie za elementy znajdujące się na tej stronie. Posiada komendy takie jak Connect i Disconnect, które obsługują odpowiednio możliwość łączenia się do serwera, jak i jego ręczne odłączanie. Klasa ta wykorzystuje konwerter CommunicationState Converter, który zapewnia tłumaczenie nazw stanów połączenia takich jak CommunicationState.Open na postać zrozumiałą dla człowieka, czyli np. „Połączono”.

FilesViewModel obsługuje działania i informacje wyświetlane na stronie FilesPage. Zadaniem klasy jest zbieranie informacji o danych z systemu plików, a także umożliwia wywoływanie akcji związanych z plikami i ich synchronizacją.

DirectorySelectViewModel jak w przypadku poprzednich klas dotyczy jedynie odpowiadającej mu stronie DirectorySelectPage i steruje akcją wybierania głównego katalogu synchronizacji na platformach, które nie posiadają wbudowanej do tego celu funkcji.

### Model

Model jest najbardziej złożoną częścią całego kodu, jego zadaniem jest udostępnienie gotowych klas, metod i właściwości, które tworzą wyższą warstwę abstrakcji i wykonują większość zadań w taki sposób, aby ViewModel był w stanie wykonywać jedynie bezpośrednie polecenia użytkownika bez zagłębiania się w ich złożony proces.

Pierwsza główna część modelu, czyli statyczna klasa Connection zajmuje się wszystkim co związane z połączeniem do zdalnego serwera, od odpowiedniej konfiguracji punktów końcowych, po zestawienie komunikacji i sprawdzenie poprawności połączenia. Ponadto przechowuje obiekty niezbędne do komunikacji przez cały czas działania programu i pozwala klasom z ViewModel’u na wysyłanie zarówno danych jak i poleceń do serwera.

Windows Communication Foundation zapewnia transparentny sposób obsługi metod po stronie serwera poprzez dodanie odpowiedniej warstwy abstrakcji. Sprawia to że aplikacja klienta wykonuje funkcje zdalne w sposób w jaki robiłaby to z funkcjami lokalnymi. Wygenerowana klasa UniSyncServiceClient sama interpretuje działania klienta i przez dodaną referencję usługi tłumaczy je na odpowiednie zapytania transportu HTTP.

Ważnym elementem podczas komunikacji jest klasa FileInfoDataContract, która nie jest niczym innym jak modelem danych znanym dla obu stron połączenia. Klasa ta jest kontraktem w jaki sposób dane są pakowane i przesyłanie przez sieć, tym samym bardzo ułatwia serializację strumienia XML.

Drugim z głównych elementów modelu jest klasa statyczna Storage. Odpowiada za wszelkie niskopoziomowe działania na plikach, w tym obsługuje wybieranie katalogu synchronizacji, a dzięki wbudowanej we framework Xamarin funkcji DependencyService, klasa Storage dobiera sposób obsługi tego działania w zależności od platformy poprzez użycie interfejsu IFilesManagement co skutkuje wyświetleniem dodatkowego widoku w sytuacji gdy jest taka potrzeba (np. system Android) lub nie (np. system Windows).

## Serwer

Aplikacja serwera jest prostym projektem WCF Service zawierającym jedynie prostą implementację obsługi plików i strumieni, które zapewniają możliwość przesyłania danych binarnych przez sieć.

Z powodu braku możliwości podziału kodu według wzorca MVVM (brak widoku) i prostotę projektu, nie został tutaj użyty żaden wzorzec architektoniczny poza wymaganym dla usług technologii WCF, czyli podział na kontrakt i jego implementację.

Dodatkowym projektem pomocniczym jest konsola serwera, która zapewnia możliwość użycia usługi WCF jako samo-hostującego programu na system Windows. W przypadku zrezygnowania z konsoli usługi można użyć np. na serwerze IIS dostępnym w każdym systemie Windows Server, a także jako usługa sieciowa, co wymaga napisania odpowiedniego projektu-kontenera.

# Wnioski

W pracy zostały zaprezentowane, porównane i omówione różne technologie, które można zastosować do stworzenia aplikacji do synchronizacji danych. Oprócz samych narzędzi możliwych do wykorzystania projekt określa również logiczny podział na części i ich rolę, a także takie aspekty jak wzorzec projektowy i warstwy abstrakcji.

Wdrożenie projektu jednoznacznie obrazuje sposób zastosowania rozwiązań wybranych w poprzedniej części pracy, a także prezentuje przykładowy sposób, w jaki projekt może zostać zaprogramowany przy pomocy dostępnych technologii.

Część wdrożeniowa nie obsługuje wszystkich funkcji projektu. Lista elementów użytych w części praktycznej, jak i tych które nie zostały zaimplementowane przedstawia się następująco:

|  |  |  |
| --- | --- | --- |
| Funkcja | Implementacja | Szczegóły |
| Obsługa wielu platform | Częściowo | * Z powodu ograniczonego czasu na wykonanie aplikacji i braku odpowiedniego zaplecza sprzętowego, nie wszystkie platformy obsługiwane przez technologię Xamarin są wspierane przez aplikację * Obecnie wspierane platformy to Windows UWP i Android * System Windows Phone nie jest wspierany, ponieważ jest on zastąpiony przez platformę Windows UWP i nie jest dalej rozwijany; poza tym brakuje odpowiedniego sprzętu do testów * System iOS nie jest wspierany, ponieważ zarówno do kompilacji jak i do emulowania systemu w procesie rozwijania niezbędny jest oddzielny komputer typu Mac z systemem macOS, którego brakuje w dostępnym zapleczu sprzętowym * Wsparcie dla dodatkowych platform może zostać łatwo dodane, gdyż jedynie nieliczne elementy muszą być wdrożone oddzielnie na różnych platformach |
| Wybór katalogu do synchronizacji | TAK | * Pomimo kilku ograniczeń związanych z koniecznością implementacji tej funkcji na każdej platformie osobno, nie było szczególnie trudne umożliwienie użytkownikowi wybrania katalogu * Mimo iż istnieje taka możliwość, użytkownik nie musi zmieniać domyślnej ścieżki synchronizacji, którą jest folder „Files” w katalogu aplikacji |
| Listowanie plików w katalogu synchronizacji | TAK | * Aplikacja potrafi odczytać informacje o plikach w katalogu |
| Listowanie katalogów w katalogu synchronizacji | NIE | * Synchronizacja katalogów, jak i plików w katalogach niższych poziomów nie jest zaimplementowana * Powodem takiego stanu rzeczy są odrębne ograniczenia związane z systemami plików, ich ograniczeniami i sposobem obsługi na każdej platformie * Aplikacja nie widzi katalogów, jedynie pliki |
| Aktualizacja listy plików z serwera | TAK | * Aplikacja potrafi pobrać aktualną listę plików znajdujących się na serwerze wraz z informacjami o nich |
| Sprawdzanie dostępności plików na serwerze | TAK | * Aplikacja dostaje informacje o tym które pliki wymagają synchronizacji, a które nie |
| Pobieranie plików z serwera | TAK | * Aplikacja potrafi pobrać plik z serwera |
| Wysyłanie plików na serwer | TAK | * Aplikacja potrafi wysłać plik na serwer |
| Ręczne wymuszenie procesu synchronizacji | TAK | * Klient decyduje kiedy rozpocząć proces aktualizacji |
| Synchronizacja automatyczna | NIE | * Automatyczny proces synchronizacji plików nie jest dostępny * Wynika to z różnego sposobu obsługi zadań w tle na każdej platformie * Ilość czasu potrzebna na zaimplementowanie tej funkcjonalności jest zbyt wysoka * Funkcja ta nie jest niezbędna do zaprezentowania procesu synchronizacji plików między urządzeniami * Znajomość każdej platformy jest niezbędna aby wprowadzić tę funkcję |

Tabela 9: Lista zaimplementowanych funkcji