Helion

Odnieś sukces na platformie Windows 8!

Windows® 8
Tworzenie aplikacji *z użyciem* **C#** i **XAML**

Jeremy **Likness**

Wintellect
Know how.

Tytuł oryginału: Building Windows 8 Apps with C# and XAML

Tłumaczenie: Paweł Gonera

ISBN: 978-83-246-7062-8

Authorized translation from the English language edition, entitled:
BUILDING WINDOWS 8 APPS WITH C# AND XAML; ISBN 0321822161;
by Jeremy Likness; published by Pearson Education, Inc, publishing as Addison Wesley.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/w8twap
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

5

Cykl życia aplikacji

W TRADYCYJNYM ŚRODOWISKU WINDOWS użytkownik zarządza czasem życia aplikacji — uruchamia ją, i ona działa do momentu, gdy użytkownik zdecyduje o jej zamknięciu. W modelu tym występuje poważny problem, ponieważ aplikacja wykorzystuje takie zasoby systemu, jak pamięć i CPU, nawet jeżeli nie znajduje się ona na pierwszym planie. Wpływa to na wydajność aplikacji uruchamianych przez użytkownika, a także powoduje szybsze wyczerpanie baterii, jeżeli urządzenie nie jest podłączone do sieci elektrycznej.

Bardzo łatwo jest pokazać to tradycyjne podejście w trybie pulpitu Windows 8. Jeżeli nie posiadasz pliku wideo, skorzystaj z poniższej witryny, udostępniającej wszystkie wideo z sesji konferencji BUILD, na której Microsoft zaprezentował Windows 8: *http://channel9.msdn.com/Events/BUILD/BUILD2011*.

(Być może czytasz to po konferencji z roku 2012; jeżeli tak, to powinieneś być w stanie skorzystać ze zaktualizowanej witryny i pobrać z niej aktualne nagrania).

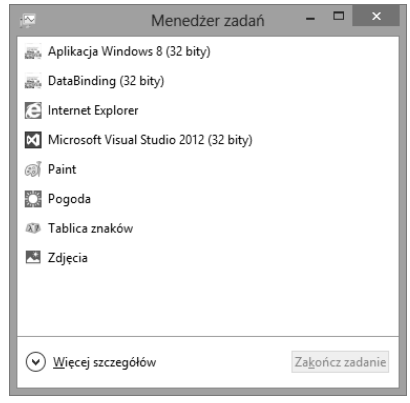
Przejdź do wybranego filmu i pobierz go w średniej lub wysokiej jakości na swój dysk. Może to zająć od kilku minut do godziny, w zależności od rozmiaru pliku. W czasie oczekiwania na pobranie pliku uruchom Menedżer zadań, który pozwala na przegląd aplikacji działających w systemie. Istnieje kilka sposobów na jego uruchomienie.

- W menu *Start* Windows 8 wpisz `menedżer zadań` i wybierz program z wyświetlonych wyników wyszukiwania.
- Kliknij na pulpicie prawym przyciskiem myszy pasek zadań i wybierz opcję *Menedżer zadań*.
- W dowolnym miejscu naciśnij jednocześnie *Ctrl+Shift+Esc*.

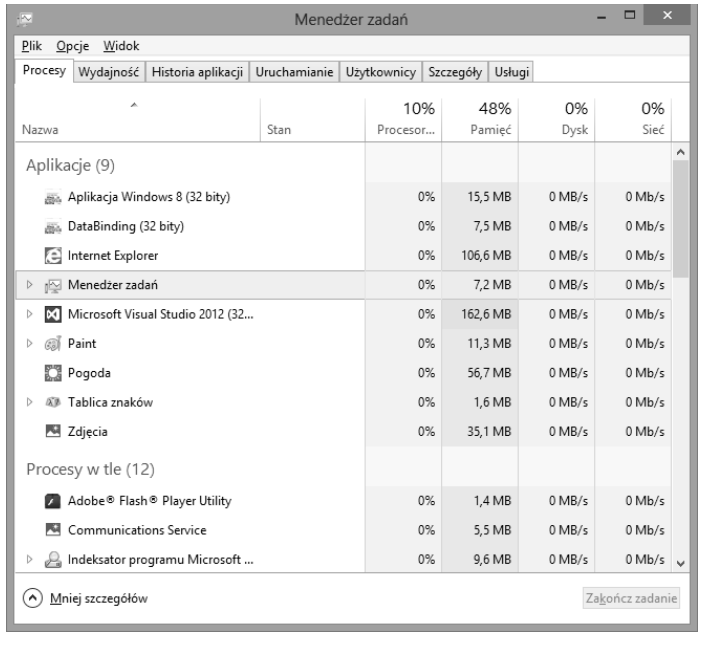
Po uruchomieniu Menedżera zadań wyświetli się prosty widok pokazany na rysunku 5.1.

Domyślny widok zawiera listę działających aplikacji. Aby zobaczyć bardziej zaawansowany widok, kliknij łącze *Więcej szczegółów* w lewej dolnej części ekranu. Spowoduje to otwarcie widoku zaawansowanego przedstawionego na rysunku 5.2. Zostaną pokazane nie tylko działające aplikacje z informacjami na temat zużycia zasobów, ale również działające procesy tła, takie jak sterowniki w systemie, które obsługują różne urządzenia.

Upewnij się, że Menedżer zadań działa w widoku zaawansowanym i że jest widoczny na ekranie. Teraz uruchom pobrany plik wideo przez kliknięcie prawym przyciskiem myszy w oknie Eksploratora Windows i wybranie *Odtwarzanie za pomocą programu Windows Media Player*. Możesz zauważyć, że zużycie procesora oraz pamięci zacznie rosnąć. Wielkość tych wzrostów zależy od konfiguracji systemu. Teraz otwórz Notatnik i rozciągnij go, aby całkowicie zakrył film. Po tej operacji zużycie procesora nie zmieni



Rysunek 5.1. Prosty widok Menedżera zadań



Rysunek 5.2. Zaawansowany widok Menedżera zadań

się, choć już nie oglądamy filmu. Aplikacja i tak realizuje zadanie przetwarzania wideo, choć przez otwarcie programu Notatnik na pierwszym planie pokazaliśmy, że nie jesteśmy obecnie zainteresowani filmem.

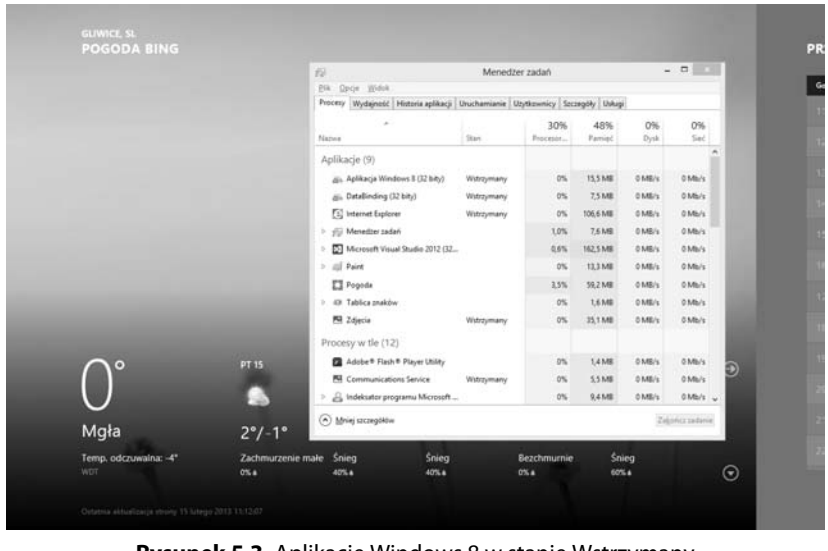
W aplikacjach Windows 8 scenariusz ten uległ zmianie. Użytkownik decyduje, która aplikacja będzie działać na pierwszym planie, i tylko jedna aplikacja może działać w taki sposób (z wyłączeniem widoku przyciągniętego, pozwalającego na działanie dwóch aplikacji, z których jedna zajmuje mały pasek przestrzeni). Następnie system określa, co się powinno stać z pozostałymi aplikacjami tła. Jeżeli nie zbudujesz swojej aplikacji prawidłowo, będzie ona powodowała negatywne efekty uboczne, wywołujące problemy w interakcji z użytkownikiem. Zarządzanie aplikacjami w Windows Runtime jest nazywane zarządzaniem czasem życia procesu (ang. *Process Lifetime Management* — PLM).

Zarządzanie czasem życia procesu

Aplikacje Windows 8 działają wyłącznie wtedy, gdy są na pierwszym planie. Pozwala to użytkownikowi na skupienie się na głównej aplikacji, z którą zamierza pracować. Aplikacje w tle przechodzą w stan wstrzymania, w którym wątki aplikacji są zamrożone. Aplikacja przestaje korzystać z zasobów systemowych, dzięki czemu zmniejsza się tempo rozładowywania baterii. W większości przypadków aplikacja pozostaje w pamięci. Pozwala to na szybkie przełączanie aplikacji, więc gdy użytkownik cofnie się do aplikacji poprzedniej, będzie ona natychmiast wznawiana, tak jakby cały czas działała.

Istnieją jednak przypadki, gdy wstrzymana aplikacja może być wyłączona przez system. Jest to zależne od zasobów systemu. Jeżeli na przykład aplikacja działająca na pierwszym planie wymaga przydzielenia dużego obszaru pamięci, który spowoduje zajęcie niemal całej dostępnej pamięci, system może wyłączyć przynajmniej jedną aplikację będącą w stanie wstrzymania. System wybiera do wyłączenia tę aplikację, która zajmuje najwięcej pamięci.

Aby zobaczyć ten proces w działaniu, uruchom menu *Start* przez naciśnięcie klawisza *Windows*. Otwórz Menedżer zadań i umieść go na ekranie Windows 8. Upewnij się, że wybrana została opcja *Zawsze na wierzchu*, znajdująca się w menu *Opcje*. Teraz uruchom aplikację, a następnie wróć do menu *Start* i uruchom kolejną. Powtarzaj ten proces kilka razy. Zauważysz, że po kilku sekundach aplikacje będą przechodziły w stan *Wstrzymany*. Jeżeli status nie jest wyświetlany, przejdź do *Widok*, *Wartości stanu* i zaznacz *Pokaż stan wstrzymany*. Na rysunku 5.3 pokazany jest wynik uruchomienia kilku aplikacji, gdy na pierwszym planie znajduje się aplikacja Pogoda. Zwróć uwagę, że inne aplikacje Windows 8 są w stanie *Wstrzymany*.



Rysunek 5.3. Aplikacje Windows 8 w stanie Wstrzymany

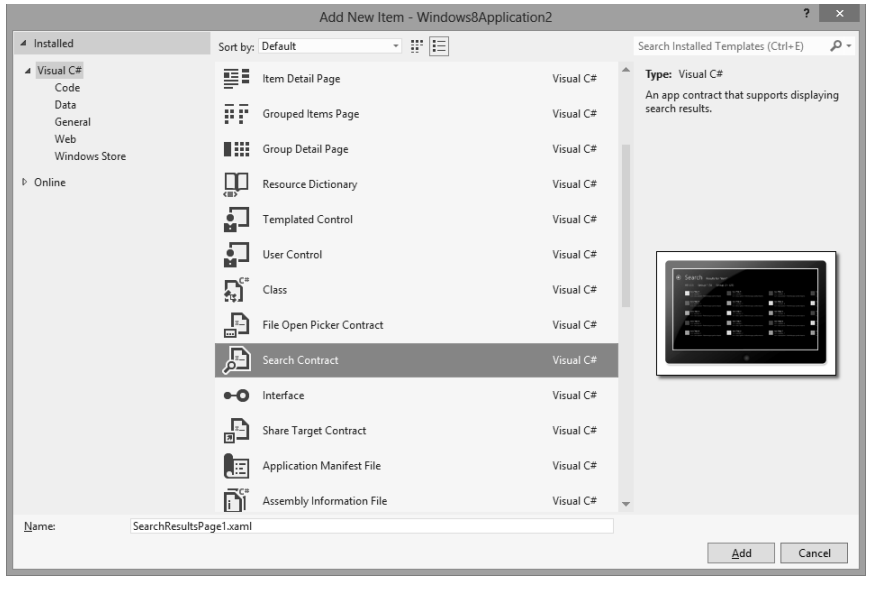
Jeżeli w systemie zacznie brakować zasobów, wiele z aplikacji w stanie *Wstrzymany* może być zakończonych i usuniętych z listy Menedżera zadań. Możesz również emulować wstrzymanie z Visual Studio 2012 w trybie debugowania. Pokażę to w dalszej części rozdziału, ale na początek warto poznać pełny cykl PLM. Życie aplikacji zaczyna się od *aktywacji*.

Aktywacja

Aktywacja jest pierwszym krokiem w cyklu życia aplikacji. W poprzednich wersjach Windows aktywacja zazwyczaj następowała w wyniku uruchomienia aplikacji przez użytkownika — za pośrednictwem menu *Start* albo przez wpisanie jej nazwy w wierszu polecenia. W Windows 8 aplikacje są zawsze aktywowane poprzez kontrakt. Informacje na temat sposobu aktywacji znajdziesz w rozdziale 2., „Zaczynamy".

Najpowszechniej stosowanym kontraktem jest *Launch*. Jest to kontrakt wywoływany w momencie stuknięcia kafelka w menu *Start*. Kontrakt *Launch* aktywuje aplikację i wywołuje metodę `OnLaunched`. Innym kontraktem, który może aktywować aplikację, jest *Share*, jeżeli aplikacja jest celem udostępniania, podobnie jak ImageHelper z rozdziału 2. Aktywacja aplikacji może być wykonana również przez kilka innych kontraktów. Aplikacja Windows8Application2 bazuje na aplikacji z rozdziału 4., „Aplikacje Windows 8", ale została rozbudowana o obsługę kontraktu *Search*. Aplikację tę możesz pobrać z serwera wydawnictwa Helion: *ftp://ftp.helion.pl/przyklady/w8twap.zip*.

Dodanie tego kontraktu do aplikacji Windows 8 jest bardzo proste. Zostało to wykonane w przykładowej aplikacji, ale jeżeli zaczynasz jej tworzenie od początku, możesz kliknąć prawym przyciskiem myszy węzeł projektu w oknie *Solution Explorer* i wybrać *Add/New Item* lub zastosować kombinację klawiszy *Ctrl+Shift+A*. Wybierz opcję *Search Contract* i nazwij stronę wyników wyszukiwania, jak jest to pokazane na rysunku 5.4.



Rysunek 5.4. Dodawanie kontraktu Search do projektu

Kiedy jest dodawany nowy element, w tle jest wykonywane kilka operacji. Oprócz utworzenia nowej klasy `Page` do pliku *App.xaml.cs* dodawana jest metoda `OnSearchActivated`, co pozwala na obsługę przychodzących żądań wyszukiwania. Generowany jest fragment kodu, który powinien zostać przez nas przeniesiony do procedury obsługi `OnLaunched`, dzięki czemu będziemy mogli obsłużyć zapytania bez uruchamiania aplikacji. Aktualizowana jest również sekcja z pakietami aplikacji, do której dodawana jest deklaracja *Search*.

Gdy cała infrastruktura jest na swoim miejscu, szablony mogą być wykorzystane do udostępnienia funkcji wyszukiwania. W moim przykładzie wyniki wyszukiwania są pokazywane na stronie z grupą filtrów. Pozwala to użytkownikowi na określenie wyszukiwania we wszystkich elementach lub ograniczyć wyszukiwanie do elementów z danej grupy. Wyszukiwanie pozwala na obsługiwanie przychodzących żądań niezależnie od tego, czy aplikacja właśnie działa, czy nie.

Gdy aplikacja jest aktywowana przez którykolwiek z kontraktów, wyświetlony zostaje ekran powitalny. Ekran powitalny składa się z tła o wybranym kolorze oraz statycznego obrazu (sposób definiowania tych elementów został opisany w rozdziale 4., a tutaj, w dalszej części rozdziału, opiszę sposób modyfikacji tego ekranu). Aplikacja jest powiadamiana o aktywacji przez wywołanie odpowiedniej metody z pliku *App.xaml.cs* (`OnLaunching` dla kontraktu *Launch*, `OnSearchActivated` dla kontraktu *Search* itd.).

Aplikacja ma około 15 sekund na pokazanie pierwszej strony. Jeżeli w tym czasie strona nie będzie wyświetlona, aplikacja zostanie zakończona, a dla użytkownika końcowego będzie to raportowane jako awaria. Jest to ważne, ponieważ aplikacje wymagające pobrania dużej ilości danych lub złożonej inicjalizacji muszą pokazać pierwszą stronę przed tymi operacjami, a następnie prawidłowo obsłużyć oczekiwanie na zakończenie tych zadań. Można to zrealizować przez rozszerzenie ekranu powitalnego. Więcej informacji na temat rozszerzonych ekranów powitalnych znajduje się w dalszej części rozdziału.

Gdy aplikacja zostanie aktywowana, staje się aplikacją pierwszoplanową i będzie działać w sposób normalny do momentu przełączenia się użytkownika do innej aplikacji, w tym pulpitu. Gdy użytkownik dokonuje przełączenia, aplikacja jest o tym fakcie powiadamiana za pomocą zdarzenia `Suspending`. Pozwala to poinformować aplikację o operacji wstrzymania i umożliwia programiście wykonanie kodu niezbędnego do zachowania stanu przed zakończeniem działania aplikacji.

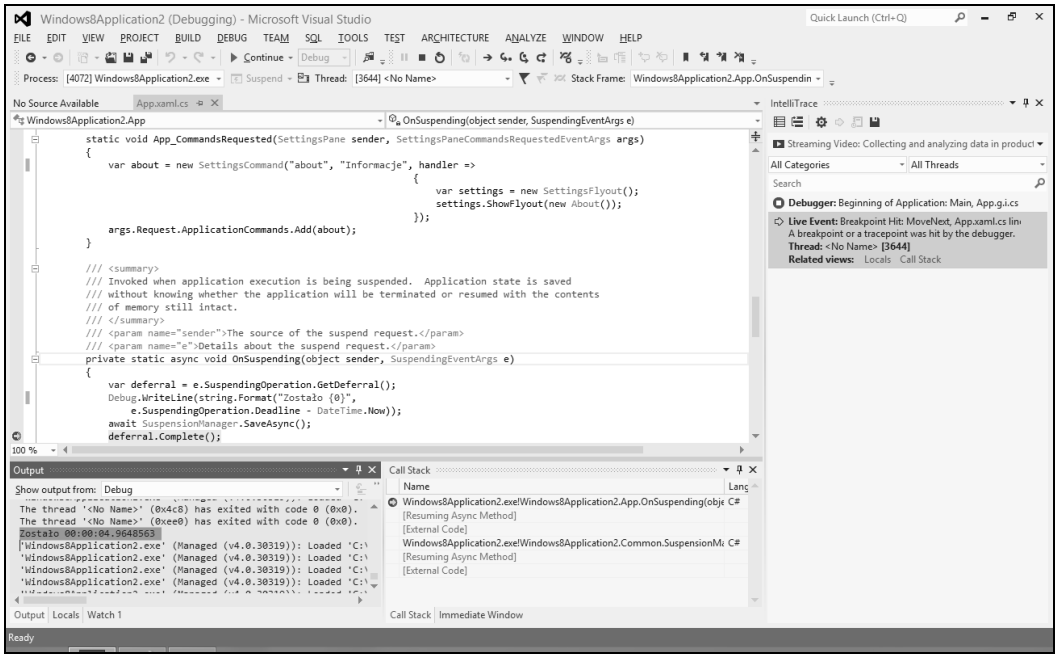
Wstrzymanie

W czasie normalnego działania aplikacja zostanie wstrzymana od razu po tym, jak przestanie działać na pierwszym planie. Nie jest łatwo wymusić wstrzymanie aplikacji w czasie testowania, więc Visual Studio 2012 pozwala na ręczne wstrzymanie programu w celu wykonania testów tego procesu. Aby zapoznać się ze wstrzymywaniem aplikacji, załaduj projekt Windows8Application2.

Otwórz plik *App.xaml.cs* i umieść punkt zatrzymania w wyróżnionym wierszu kodu metody `OnSuspending`:

```
private static async void OnSuspending(object sender,
    SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    Debug.WriteLine(string.Format("Zostało {0}",
        e.SuspendingOperation.Deadline - DateTime.Now));
    await SuspensionManager.SaveAsync();
    deferral.Complete();
}
```

Uruchom aplikację w trybie debugowania. Na pasku narzędzi możesz znaleźć przycisk z ikoną, która wygląda jak przycisk pauzy dla aplikacji na drugim planie (nie pomyl jej z przyciskiem pauzy, który wymusza operację *Break All*). Dymek podpowiedzi jest pokazany na rysunku 5.5. Użyj tego przycisku do emulowania wstrzymania aplikacji. Po kliknięciu przycisku aplikacja zostanie zatrzymana w wyróżnionym wierszu metody `OnSuspending`. Zwróć uwagę na wartość wypisaną w oknie *Output* (jeżeli nie widzisz go, zastosuj kombinację klawiszy *Ctrl+Alt+O*), a następnie kliknij *Continue*.



Rysunek 5.5. Wstrzymywanie aplikacji w trybie debugowania

Wartość `Deadline` zawiera czas, po którym nastąpi wstrzymanie. Jeżeli aplikacja nie wykona niezbędnych zadań w tym okresie, zostanie zakończona. Przy wykonywaniu zadań asynchronicznych musisz zażądać wprowadzenia opóźnienia. Gdy nastąpi przekazanie opóźnienia w argumentach żądania, można wykonać operacje asynchroniczne, na przykład zapisanie stanu aplikacji na dysku. W przykładowej aplikacji zapisanie stanu jest realizowane w zdarzeniu, więc nie potrzeba wykonywać dodatkowych czynności przy wstrzymaniu aplikacji. Gdy aplikacja jest gotowa do wstrzymania, wywołujemy metodę `Complete`, aby przejść dalej.

WSKAZÓWKA

Do tej pory nauczyłeś się, w jaki sposób używać debugera Visual Studio 2012 do debugowania aplikacji działających lokalnie lub w symulatorze. Istnieje również możliwość uruchomienia debugowania na zdalnym komputerze. Jest to szczególnie przydatne, jeżeli posiadasz urządzenie do testowania operacji dotykowych, ale wolisz tworzyć aplikacje na laptopie lub stacji roboczej. Aby zdalnie debugować aplikacje, musisz wcześniej uruchomić na zdalnym komputerze aplikację, która konfiguruje środowisko debugowania. Następnie możesz zdefiniować zdalny komputer w opcji *Remote Machine*. Może zajść konieczność uwierzytelnienia się przed rozpoczęciem sesji. Dokładna instrukcja konfiguracji tej opcji jest dostępna pod adresem *http://msdn.microsoft.com/en-us/library/bt727f1t(v=vs.110).aspx*.

Na tym etapie aplikacja stale znajduje się w pamięci, ale nie zużywa żadnych innych zasobów. Jeżeli uruchomisz Menedżer zadań, zauważysz, że użycie procesora wynosi 0%. Aplikacja nie wpływa już na

zużycie baterii, bo po prostu czeka. Jeżeli wznowisz działanie (przez przełączenie aplikacji na pierwszy plan lub przez kliknięcie operacji *Resume*, znajdującej się w Visual Studio 2012 w tym samym menu rozwijanym co *Suspend*), aplikacja zostanie przywrócona do działania bez widocznego opóźnienia.

Jądro posiada jeszcze jedną usługę do zatrzymywania aplikacji. Może się zdarzyć, że aplikacja może wykonywać sekcję krytyczną kodu, na przykład zwalnianie blokad plików. Jądro nie zatrzyma aplikacji w czasie wykonywania sekcji krytycznej kodu (zwykle, gdy sterowanie jest przekazywane do komponentu WinRT), a zamiast tego poczeka do momentu zakończenia wykonywania tej sekcji krytycznej, unikając dzięki temu powstania blokad w systemie.

Możesz zauważyć jeszcze jedną opcję obok *Suspend* oraz *Resume*, opisaną jako *Suspend and shutdown*. Przycisk ten jest używany do zakończenia działania aplikacji. Symuluje on sytuację, w której aplikacja jest w stanie zatrzymanym, a brak zasobów wymusza na systemie wyłączenie Twojej aplikacji w celu zwolnienia pamięci.

Zakończenie działania

Gdy system potrzebuje pamięci, kiedy następuje przełączenie aktywnego użytkownika, system jest zamykany lub nasza aplikacja ulegnie awarii, Windows Runtime kończy działanie aplikacji. Nie istnieje zdarzenie informujące o zakończeniu działania. Dlatego tak ważne jest zapisanie stanu, gdy aplikacja jeszcze działa lub po wywołaniu procedury zdarzenia `OnSuspending`.

Typowym zachowaniem aplikacji jest wyświetlenie pierwszego ekranu, gdy została ona uruchomiona po raz pierwszy lub po jej zamknięciu przez użytkownika. Aplikacja powinna zostać przywrócona do zapisanego stanu wyłącznie wtedy, gdy nastąpił powrót ze stanu zakończenia. Na szczęście poprzedni stan aplikacji zawsze jest przekazywany do zdarzenia `OnLaunched`. Pozwala to na sprawdzenie poprzedniego stanu i podjęcie odpowiedniej akcji w zależności od tego, czy użytkownik przywraca aplikację po jej zakończeniu, czy uruchamia ją normalnie po jej zamknięciu.

Wznowienie

Aplikacja może zarejestrować zdarzenie `Resuming`. Zdarzenie to jest wywoływane, gdy aplikacja jest nadal w pamięci, a użytkownik przełącza ją na pierwszy plan. Nie ma potrzeby odtwarzania stanu, ponieważ wszystkie wątki aplikacji, stos oraz magazyn pozostają dostępne. Do czego może być przydatne takie zdarzenie?

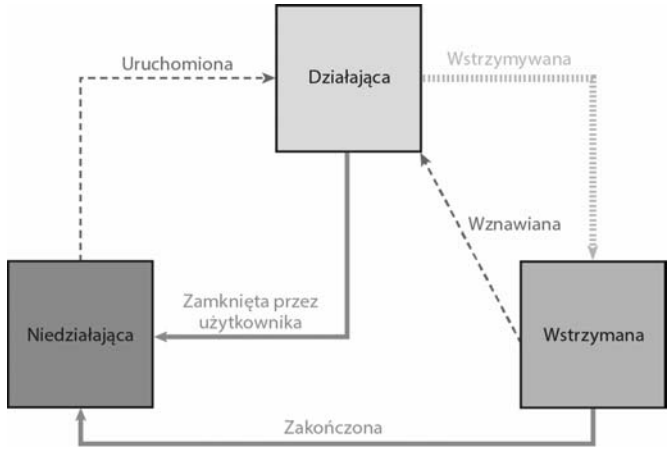
Głównym jego zastosowaniem jest zapewnienie okresowego odświeżania danych w aplikacji. Wyobraźmy sobie aplikację z prognozą pogody, która była wstrzymana przez kilka godzin, na czas pasjonującej rozgrywki w Twojej ulubionej grze. Aplikacja pozostawała zamrożona w pamięci wraz ze wszystkimi wątkami. Następnie wracasz do aplikacji. Co widzisz? Bez zdarzenia `Resuming` będziesz widział stare informacje pogodowe sprzed kilku godzin. Zdarzenie to pozwoli aplikacji pobrać najnowszą prognozę i pokazać bieżący stan pogody, w tym alarmy, jakie mogły się pojawić.

Jeżeli aplikacja jest zakończona, to jest ponownie uruchamiana przy próbie powrotu do niej przez użytkownika. Zdarzenie `Resuming` nie jest wtedy uruchamiane. Zamiast tego aplikacja może sprawdzać poprzedni stan w celu określenia, który wariant kodu startowego należy wykonać. Zajrzyj do pliku *ExtendedSplashScreen.xaml.cs*, do metody `ExtendedSplashScreen_Loaded`, w której znajduje się następujący warunek:

```
if (_activationArgs.PreviousExecutionState ==
   ApplicationExecutionState.Terminated)
```

Jeżeli aplikacja była wyłączona, zostanie odczytany poprzedni stan i użytkownik wróci do miejsca, w którym wcześniej skończył pracę. Sposób zapisu i przywracania stanu jest przedstawiony w dalszej części rozdziału. W przeciwnym razie aplikacja inicjuje się i jest przywoływany ekran *Start*. Innymi stanami, które można sprawdzić, są: `ClosedByUser`, `NotRunning`, `Running` oraz `Suspended`.

Na rysunku 5.6 przedstawiony jest schemat cyklu życia aplikacji.



Rysunek 5.6. Schemat cyklu życia aplikacji

Poniższa lista wskazówek pomoże Ci budować aplikację w taki sposób, aby zapewnić użytkownikowi możliwie płynne działanie przez zarządzanie PLM:

- **przyrostowo zapisuj dane użytkownika** — pozwala to uniknąć zapisywania wszystkiego naraz przy zatrzymywaniu aplikacji, co niesie ze sobą niebezpieczeństwo przekroczenia czasu,
- **zapisuj i przywracaj stan aplikacji** — jest to ważne, gdyż należy się upewnić, czy użytkownik będzie miał wrażenie ciągłej pracy po przywróceniu aplikacji,
- **zapisuj i przywracaj wyłącznie metadane sesji (dotyczące tego, w którym punkcie znajduje się użytkownik)** — pozostałe dane powinny być obsługiwane w innym miejscu aplikacji.

Teraz, gdy znasz już cykl życia aplikacji, możesz użyć kodu z szablonów do zarządzania stanem aplikacji. Pomoże Ci to zrozumieć, w jaki sposób działa nawigacja w aplikacjach Windows 8. Gdy wcześniej zatrzymana aplikacja jest uruchamiana z ekranu *Start*, będzie ona wznowiona, a nie uruchomiona od początku.

Nawigacja

W aplikacjach Windows 8 występują dwa podstawowe typy nawigacji — hierarchiczna i płaska. W aplikacjach hierarchicznych zaczynamy od widoku wysokiego poziomu i zagłębiamy się aż do interesującego nas obszaru. W przykładowej aplikacji operowaliśmy na grupach kolekcji, następnie wchodziliśmy do kolekcji i ostatecznie do poszczególnych elementów. Nawigacja płaska działa podobnie jak w kreatorze, kierując nas zgodnie z kolejnymi krokami procesu.

Po pierwszym uruchomieniu aplikacji pocztowej najprawdopodobniej zaczniemy od scenariusza nawigacji płaskiej. Kreator poprowadzi nas przez proces wprowadzania informacji na temat serwera

pocztowego, danych uwierzytelniających, testowania połączenia oraz dodawania konta do listy. Gdy zaczniemy korzystać z aplikacji pocztowej, widok stanie się hierarchiczny. Na początku otwiera się widok kont, a następnie wchodzimy głębiej, do poszczególnych wiadomości.

W przypadku aplikacji Windows 8 zbudowanych z wykorzystaniem XAML oraz C# domyślnym kontenerem dla aplikacji jest `Frame`. Zajrzyj do pliku *ExtendedSplashScreen.xaml.cs* w aplikacji Windows8Application2. Kod w metodzie `ExtendedSplashScreen_Loaded` tworzy obiekt `Frame`, podłącza zdarzenia, a następnie przechodzi do gotowej strony. Ekran powitalny wykonuje pewne operacje przed podłączeniem głównego mechanizmu nawigacyjnego aplikacji (więcej informacji na temat tworzenia własnych ekranów startowych znajduje się w dalszej części rozdziału):

```
var rootFrame = new Frame();
rootFrame.Navigating += rootFrame_Navigating;
// Tu znajduje się dodatkowy kod
rootFrame.Navigate(target, parameter);
```

Obiekt `Frame` jest kontenerem wizualnym obsługującym nawigację. Jednostką nawigacji wewnątrz `Frame` jest `Page`. Obiekt `Page` jest specjalnym rodzajem kontrolki, który ma możliwość współpracy z mechanizmem nawigacji. Można go uznać za kontener dla elementów wizualnych, które chcemy zaprezentować w określonym obszarze aplikacji. Obiekt `Frame` pozwala na nawigowanie przez przekazywanie typu obiektu `Page`, który chcemy wyświetlić, jak również opcjonalnego parametru. W przykładowej aplikacji celem jest domyślnie typ głównego okna zawierającego różne grupy, a jako parametr przekazywana jest kolekcja grup ze źródła danych:

```
if (!rootFrame.Navigate(typeof(GroupedItemsPage), "ItemGroups"))
```

Wywołanie metody `Navigate` z `Frame` powoduje utworzenie obiektu docelowej strony. Gdy docelowa strona zostanie załadowana, będzie wywołana metoda `OnNavigateTo`. Pozwala to skonfigurować interfejs użytkownika. Przykładowa aplikacja korzysta z tej metody do podłączenia danych z bazowego modelu widoku (więcej na temat modelu widoku znajdziesz w rozdziale 9., „MVVM i testowanie”), jak również do ustawienia źródła dla elementów kontrolki.

Obiekt `Page` przechowuje historię wszystkich obiektów `Page`, do których nawigował użytkownik. Pozwala to na realizację funkcji zbliżonej do przeglądarki internetowej, ponieważ możesz cofnąć się do poprzedniego obiektu lub przejść do przodu, jeżeli wcześniej nawigowałeś wstecz. Akcje te są obsługiwane za pomocą metod `GoBack` oraz `GoForward`, a aby sprawdzić, czy akcje te są możliwe, powinieneś odczytać wartości właściwości `CanGoBack` i `CanGoForward` z obiektu `Frame`.

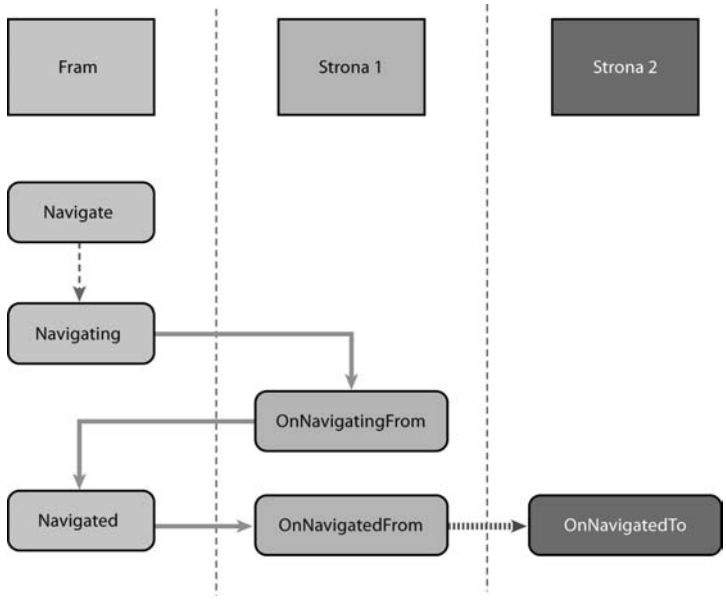
Otwórz plik *GroupDetailPage.xaml*. Kod XAML dla przycisku cofnięcia wygląda następująco:

```
<Button x:Name="backButton" Click="GoBack"
   IsEnabled="{Binding Frame.CanGoBack,
   ElementName=pageRoot}"
   Style="{StaticResource BackButtonStyle}"/>
```

Przycisk jest aktywny wyłącznie wtedy, gdy w historii znajduje się poprzednia strona. Jeżeli jest on aktywny i użytkownik kliknie przycisk, wywołana zostanie metoda `GoBack`. Metoda ta jest zdefiniowana w bazowej klasie `LayoutAwarePage`, która z kolei dziedziczy po `Page`:

```
protected virtual void GoBack(object sender, RoutedEventArgs e)
{
    if (this.Frame != null && this.Frame.CanGoBack)
       this.Frame.GoBack();
}
```

Na rysunku 5.7 zamieszczony jest typowy przebieg wystąpień zdarzeń nawigacji. Na obiekcie `Frame` została wywołana metoda `Navigate`, co powoduje przejście do nowej strony o nazwie *Strona 2*. W obiekcie `Frame` generowane jest zdarzenie `Navigating`. Obiekt *Strona 1* jest informowany o nawigacji za pośrednictwem zdarzenia `OnNavigatingFrom`. Jeżeli istnieje powód anulowania nawigacji (na przykład użytkownik wprowadził dane w formularzu i nie zapisał ich), *Strona 1* może to wykonać przez ustawienie znacznika `Cancel` w zdarzeniu.



Rysunek 5.7. Przegląd zdarzeń nawigacji

Gdy *Strona 1* nie decyduje się na anulowanie nawigacji, obiekt `Frame` usuwa z widoku stronę i generuje zdarzenie `Navigated`. Informuje obiekt *Strona 1* o jego usunięciu z widoku za pomocą metody `OnNavigatedFrom`. Następnie umieszcza obiekt *Strona 2* w widocznej ramce i powiadamia go o tym fakcie za pomocą zdarzenia `OnNavigatedTo`. Obiekt `Frame` może utworzyć nowy obiekt *Strona 2* lub ponownie użyć istniejącego. *Strona 1* wchodzi teraz do historii, więc właściwość `CanGoBack` jest włączona i wywołanie `GoBack` w obiekcie `Frame` spowoduje ponowne pokazanie *Strona 1*.

Zarządzanie obiektami stron przez `Frame` odbywa się za pośrednictwem właściwości `CacheSize`. Jej wartość reprezentuje liczbę stron znajdujących się w pamięci podręcznej. Jeżeli strona jest umieszczona w pamięci podręcznej, to nie będzie tworzony nowy obiekt, tylko zostanie ona powtórnie wykorzystana. Aby obiekt `Page` był przechowywany w pamięci podręcznej, należy ustawić w nim właściwość `NavigationCacheMode` na `Enabled` lub `Required`. Gdy wartością tej właściwości jest `Required`, będzie ona zapamiętywana niezależnie od ustawienia `CacheSize` w obiekcie `Frame` i nie będzie uwzględniana przy ustalaniu liczby w `CacheSize`.

Zrozumienie sposobu działania nawigacji ułatwia zarządzanie lokalizacją użytkownika. Wbudowane szablony w Visual Studio 2012 posiadają klasę pomocniczą o nazwie `SuspensionManager`, która wspiera programistę przy zapisywaniu i odtwarzaniu stanu. Po załadowaniu aplikacji główny obiekt `Frame` jest rejestrowany wraz z `SuspensionManager` w pliku *App.xaml.cs*:

```
SuspensionManager.RegisterFrame(rootFrame, "AppFrame");
```

Powoduje to skonfigurowanie obiektu `SessionManager`, który obserwuje nawigację użytkownika w ramce. Następnie zachowuje stos nawigacji razem z parametrami przekazanymi do stron, dzięki czemu po wznowieniu działania aplikacji może odtworzyć zarówno lokalizację użytkownika, jak i historię nawigacji. Stan jest zapisywany w lokalnej pamięci podręcznej za pomocą API Application Data.

API Application Data

API Application Data zapewnia mechanizmy manipulowania danymi specyficznymi dla określonej aplikacji. Można korzystać z niego do zapisywania stanu, preferencji, lokalnej pamięci podręcznej oraz innych ustawień. Windows Runtime zarządza szczegółami zapisu specyficznymi dla aplikacji i izoluje je w ramach konta użytkownika i aplikacji w systemie. Dane są zachowywane przy aktualizacji aplikacji i usuwane w przypadku jej odinstalowania.

Mamy do dyspozycji trzy typy danych lokalnych dostępnych dla aplikacji. W tabeli 5.1 wymienione są te typy wraz z opisem ich przeznaczenia.

Tabela 5.1. Typy przechowanych danych aplikacji

Magazyn	Opis
`Local`	Są dostępne dla aplikacji i użytkowników bieżącego urządzenia. Są zachowywane pomiędzy sesjami działania aplikacji oraz uruchomieniami komputera.
`Roaming`	Są dostępne dla aplikacji i użytkowników wszystkich urządzeń. Magazyn ten jest ograniczony do około 100 kB danych. Wymaga konta Windows.
`Temporary`	Magazyn ten może być w dowolnym momencie usunięty z systemu i zwykle jest używany do lokalnych „plików roboczych" lub innych danych tymczasowych, które nie muszą być zachowywane pomiędzy kolejnymi uruchomieniami aplikacji.

Każdy magazyn zapewnia dwa rodzaje dostępu. Pierwszy rodzaj, *ustawienia aplikacji*, pozwala na przechowywanie dowolnego typu danych WinRT. Można w ten sposób zapisywać takie dane, jak pojedyncze wartości, wartości złożone lub w kontenerach, uporządkowane zbiory danych obu typów. Magazyn ten jest przeznaczony dla małych zbiorów danych. Zgodnie z dokumentacją ustawienia te są w rzeczywistości przechowywane w rejestrze. Dokumentacja dotycząca ustawień aplikacji znajduje się pod adresem *http://msdn.microsoft.com/en-us/library/windows/apps/hh464917.aspx*.

Drugim rodzajem są *pliki aplikacji*. Są to pliki w tradycyjnym sensie, które można odczytywać i zapisywać. Każdy magazyn ma przestrzeń przydzieloną dla aplikacji i użytkownika. Możemy tworzyć katalogi do 32 poziomów głębokości oraz odczytywać i zapisywać tyle plików, ile potrzebujemy. Pliki zapisane w magazynie tymczasowym mogą być usunięte w każdym momencie. Gdy użytkownik jest zalogowany do konta Windows (na przykład Live), pliki zapisywane do magazynu wspólnego są synchronizowane pomiędzy urządzeniami, ale muszą się one mieścić w ograniczeniach konta, które można odczytać za pomocą `ApplicationData.RoamingStorageQuota`.

Aplikacja kontroluje dane lokalne, więc zazwyczaj nie są one zmieniane poza kodem aplikacji. Dane wspólne są inne, ponieważ użytkownik może zmienić je na jednym urządzeniu, a inna instancja aplikacji działa na drugim urządzeniu. Gdy dane są synchronizowane, zmiany te powodują wygenerowanie zdarzenia `DataChanged`. Twoja aplikacja może używać tego zdarzenia do reagowania na zmiany i odświeżenia danych z lokalnego urządzenia. Możesz również jawnie zasygnalizować modyfikacje danych przez wywołanie `SignalDataChanged`.

Przykładowa aplikacja *Windows8Application2* korzysta z ustawień lokalnych do zapisywania ustawień lokalizacji dla aplikacji, co jest domyślnym działaniem klasy `SuspensionManager`. Korzystamy tu z ustawień lokalnych zamiast wspólnych, ponieważ dodany nowy element jest przechowywany wyłącznie lokalnie. Lista elementów może rozrosnąć się do sporych rozmiarów, więc nie ma sensu przechowywać jej w ustawieniach wspólnych. Więcej informacji na temat danych oraz zarządzania dużymi kolekcjami pomiędzy urządzeniami znajdziesz w rozdziale 6., „Dane".

Ustawienia przechowywania danych aplikacji są niezmiernie proste w użyciu. Jeżeli nie chcesz zapamiętywać historii nawigacji ani parametrów przekazanych do kolejnych stron, możesz bez trudu zapisać bieżącą stronę przez zaktualizowanie właściwości `NavigatedPage` w pliku *App.xaml.cs*:

```
set
{
    _navigatedPage = value;
    ApplicationData.Current.LocalSettings.Values["NavigatedPage"]
       = value.ToString();
}
```

Jest to naprawdę proste — wystarczy wybrać klucz dla właściwości i ustawić wartość. Możliwe jest również zapisanie grup wartości. Jeżeli potrzebujesz operacji atomowej, dzięki której zostanie zapisana cała grupa wartości lub nie zostanie zapisana żadna z nich, użyj właściwości `ApplicationDataCompositeValue`. Powoduje to przygotowanie atomowej grupy wartości do jej zapisania w postaci jednego ustawienia. Bieżąca grupa oraz element mogą być zapisane w następujący sposób, za pomocą kodu z pliku *App.xaml.cs*:

```
private void PersistCurrentSettings()
{
   var value = new ApplicationDataCompositeValue();
   value.Add("Group", _currentGroup == null ? string.Empty :
      _currentGroup.UniqueId);
   value.Add("Item", _item == null ? string.Empty : _item.UniqueId);
   ApplicationData.Current.LocalSettings.Values["Current"] = value;
}
```

Odczyt wartości jest równie prosty. Aby przywrócić bieżącą stronę, aplikacja może po prostu odczytać wartość jako ciąg znaków, a następnie skonwertować ją na aktualny typ dla nawigacji:

```
target = Type.GetType(ApplicationData.Current.LocalSettings
   .Values["NavigatedPage"].ToString(), true);
```

Do mechanizmu nawigacji zostanie przekazana wartość bieżącego elementu lub grupy. Wartości złożone są odczytywane w poniższy sposób:

```
var value = ApplicationData.Current.LocalSettings.Values["Current"]
   as ApplicationDataCompositeValue;
```

Następnie wartość może być odtworzona i pobrana do wewnętrznych kolekcji, co pozwoli wygenerować pełny obiekt:

```
var currentItemId = value["Item"];
CurrentItem = (from g in DataSource.ItemGroups
                from i in g.Items
                where i.UniqueId.Equals(currentItemId)
                select i).FirstOrDefault();
```

Zapytanie jest zapisane z użyciem składni Language Integrated Query, czyli LINQ. Więcej informacji na temat LINQ można znaleźć pod adresem *http://msdn.microsoft.com/en-us/library/bb397926(v=vs.110).aspx*.

Możliwe jest również rozdzielenie ustawień na osobne kontenery. Do tego celu należy użyć klasy `ApplicationDataContainer`. Magazyn posiada metodę `CreateContainer`, jak również listę bieżących kontenerów w `Containers`. Po odwołaniu się do danego kontenera możesz odczytać słownik `Values` i określić lub odczytywać ustawienia specyficzne dla tego kontenera.

Magazyn nie jest ograniczony do prostych par klucz-wartość. Można również tworzyć foldery i pliki. Zamiast zapisywać pojedyncze strony i parametry, jak przed chwilą pokazałem, można użyć generycznej klasy pomocniczej `SuspensionManager` do obsłużenia większości scenariuszy nawigacji. Za pomocą `SuspensionManager` deklarujemy nazwę pliku, w którym będą zapisane wszystkie dane stanu dla aplikacji:

```
private const string sessionStateFilename = "_sessionState.xml";
```

Klasa ta korzysta z `DataContractSerializer` do utworzenia obrazu zapisanego stanu nawigacji. Klasa ta może przejrzeć wszystkie właściwości i wartości w obiekcie i zapisać je w postaci reprezentacji XML. Poniżej przedstawiony jest kod, który realizuje tę operację przez przekształcenie obiektu na postać pamięciowego dokumentu XML:

```
MemoryStream sessionData = new MemoryStream();
DataContractSerializer serializer =
    new DataContractSerializer(typeof(Dictionary<string, object>),
        _knownTypes);
serializer.WriteObject(sessionData, _sessionState);
```

Kod ten jest zapisywany na urządzeniu. API Application Data jest używane do utworzenia instancji pliku, a następnie serializowany strumień jest zapisywany na dysku:

```
StorageFile file = await ApplicationData.Current.LocalFolder
    .CreateFileAsync(sessionStateFilename,
        CreationCollisionOption.ReplaceExisting);
using (Stream fileStream = await file.OpenStreamForWriteAsync())
{
    sessionData.Seek(0, SeekOrigin.Begin);
    await sessionData.CopyToAsync(fileStream);
    await fileStream.FlushAsync();
}
```

Aby przy powrocie użytkownika do aplikacji odtworzyć stan, wykonywana jest odwrotna operacja:

```
StorageFile file = await ApplicationData.Current.LocalFolder
    .GetFileAsync(sessionStateFilename);
using (IInputStream inStream = await file.OpenSequentialReadAsync())
{
    DataContractSerializer serializer = new DataContractSerializer(
    typeof(Dictionary<string, object>), _knownTypes);
    _sessionState = (Dictionary<string, object>)serializer
        .ReadObject(inStream.AsStreamForRead());
}
```

Skompiluj i uruchom aplikację, a następnie przejdź do grupy i elementu. Zakończ aplikację przez przeciągnięcie palcem poza dolną krawędź ekranu lub naciśnięcie *Alt+F4*. Teraz otwórz *Eksplorator plików* i przejdź do następującego katalogu:

```
C:\users\{nazwa_użytkownika}\AppData\Local\Packages\{nazwa_pakietu}
```

W miejsce *{nazwa_użytkownika}* wstaw swoją nazwę użytkownika, a w miejsce *{nazwa_pakietu}* nazwę pakietu ustawioną w polu *Package name*, na zakładce *Packaging*, w manifeście aplikacji. Na moim komputerze ścieżka ta wygląda następująco:

```
C:\Users\Jeremy\AppData\Local\Packages\34F9460F-0BA2-4952-8627-439D8DC45427_req6rhny9ggkj
```

Otwórz katalog *LocalState*, w którym powinieneś znaleźć plik *_sessionState.xml*. W pliku tym zapisany jest stan aplikacji.

Przykładowy element XML przechowujący historię nawigacji wygląda następująco:

```
<KeyValueOfstringanyType>
<Key>Navigation</Key>
<Value i:type="a:string" xmlns:a="http://www.w3.org/2001/XMLSchema">1,3,2,33,
➥Windows8Application2.GroupedItemsPage,12,10,ItemGroups,32,Windows8Application2.
➥GroupDetailPage,12,7,Group-2,31,Windows8Application2.ItemDetailPage,12,14,Group-2-Item-2
</Value>
</KeyValueOfstringanyType>
```

Więcej informacji na temat zapisywania i odczytywania danych znajduje się w rozdziale 6., „Dane”.

Żywy i podłączony

Choć aplikacje są zatrzymywane i potencjalnie wyłączane po przełączeniu, to jednak nie oznacza to, że nie mogą pozostawać podłączone lub sprawiać wrażenia, że normalnie działają. Jednym z efektywnych sposobów na zapewnienie aktualności aplikacji jest użycie kafelków i powiadomień. Więcej informacji na te tematy znajduje się w rozdziale 7., „Kafelki i powiadomienia”.

Istnieją również inne sposoby na zapewnienie ciągłej pracy aplikacji. Poniższa lista zawiera kilka z możliwości:

- **dźwięk** — niektóre aplikacje, takie jak odtwarzacze podcastów i muzyki, mogą zarejestrować operację odtwarzania dźwięku w tle, korzystając z podstawowego zbioru kontrolek dźwięku,
- **pobieranie** — do pobrania dużych ilości danych przy oszczędnym wykorzystaniu baterii można użyć Background Transfer API,
- **ekran blokady** — aplikacje ekranu blokady mogą wyświetlać podstawowe informacje na zablokowanym ekranie (na przykład aplikacja pocztowa może pokazywać liczbę nieprzeczytanych wiadomości),
- **gniazda** — klasa `ControlChannelTrigger` może być użyta do zarejestrowania zdarzeń sieciowych, które powiadamiają aplikację nawet, gdy jest zatrzymana.

Nieprawdziwe jest twierdzenie, że aplikacja Windows 8 zatrzymuje się w momencie jej przełączenia w tło. Kombinacja usług zapewnianych przez platformę pozwala na zachowanie połączeń i aktualizowanie danych bez konieczności działania na pierwszym planie. Użycie odpowiednich API pozwala aplikacji na oszczędzanie zasobów systemowych oraz zwiększanie czasu pracy na bateriach dzięki obsłudze zdarzeń we właściwy sposób.

Własny ekran startowy

We wcześniejszej części tego rozdziału wspomniałem, że aplikacja musi wyświetlić pierwszą stronę w określonym czasie i że w przeciwnym razie zostanie ona zakończona. Gdy potrzebujesz więcej czasu na inicjalizację aplikacji, możesz wyświetlić własny ekran startowy i z jego poziomu realizować inicjalizację. W tym podrozdziale pokażę, w jaki sposób zbudować własny ekran startowy.

W aplikacji *Windows8Application2* długi czas inicjalizacji jest symulowany przez wykonanie operacji w pętli. W trybie debugowania pętla ta działa długo, ponieważ zapisuje wartości do okna debugera:

```
await Task.Run(() =>
                    {
                        for (var x = 0; x < 2000; x++)
                        {
                            Debug.WriteLine(x);
                        }
                    });
```

Pętla ta ma symulować operacje często wykonywane w czasie inicjalizacji aplikacji. Operacje takie mogą obejmować podłączenie się do usług sieciowych i pobranie treści lub odczyt elementów zapisanych w pamięci podręcznej. Jeżeli aplikacja wykonuje te operacje zbyt długo, po 15 sekundach zostanie zamknięta przez Windows 8.

Aby rozwiązać ten problem, dane mogą być ładowane przez własny ekran startowy przed wyświetleniem głównej strony aplikacji. W pliku *App.xaml.cs*, w metodzie `OnLaunched`, jako główny ekran ustawiany jest obiekt typu `ExtendedSplashScreen`, do którego przekazywany jest bieżący ekran startowy:

```
var splashScreen = args.SplashScreen;
var eSplash = new ExtendedSplashScreen(splashScreen, false, args);
splashScreen.Dismissed += eSplash.DismissedEventHandler;
Window.Current.Content = eSplash;
Window.Current.Activate();
```

Kontrolka `ExtendedSplashScreen` przechowuje referencję oryginalnego ekranu startowego. Zawiera ona kopię obrazu wyświetlanego na oryginalnym ekranie startowym; w celu ustawienia tego obrazu na ekranie korzysta ze współrzędnych określonych przez system. Obraz pokazuje się dokładnie w tym samym miejscu, dzięki czemu wykonywane jest niezauważalne przejście od obrazu z ekranu startowego do rozszerzonego ekranu startowego.

Po załadowaniu kontrolki wykonywana jest metoda `Initialize` na źródle danych i następuje asynchroniczne oczekiwanie na załadowanie elementów. Gdy wszystkie elementy zostaną załadowane, kontrolka jest zastępowana przez obiekt `Frame` i następuje przejście do pierwszej strony. Cała logika normalnie znajduje się w pliku kodu ukrytego *App.xaml.cs*, ale została przeniesiona do kontrolki `ExtendedSplashScreen`, dzięki czemu może być opóźniona do momentu załadowania kompletu danych.

Kontrolka `ProgressRing` została umieszczona na ekranie startowym poniżej grafiki, dzięki czemu użytkownik widzi, że jest realizowane przetwarzanie. Aby zobaczyć wskaźnik postępu, uruchom aplikację w trybie debugowania. Zobaczysz wtedy obraz ekranu startowego wraz z obracającym się wskaźnikiem postępu. Jest on animowany nawet pomimo zablokowania programu przez asynchroniczną obsługę operacji, więc nie blokuje on wątku UI. Przykład ten ilustruje bardzo prosty mechanizm sprzężenia zwrotnego, ale aplikacje z bardziej złożonymi procesami inicjującymi mogą korzystać z paska postępu i wyświetlać tekstowe informacje o ładowanych komponentach. Bardziej złożone przykłady są przedstawione w kolejnych rozdziałach; tutaj ekran startowy wykonuje jedynie pętlę spowalniającą, a następnie uruchamia główną stronę.

Podsumowanie

W tym rozdziale przedstawiłem cykl życia aplikacji oraz rolę, jaką spełnia mechanizm Process Lifetime Management (PLM). Aplikacje mogą być zatrzymywane i wyłączane, ale mają mechanizmy pozwalające na zapisanie i odtworzenie stanu, które umożliwiają synchronizację wielu urządzeń. Zapoznałeś się z nawigacją bazującą na ramkach, która jest wykorzystywana w aplikacjach Windows 8, oraz ze sposobami

wykorzystania lokalnego magazynu do zapisania lokalizacji użytkownika w aplikacji. Lokalny magazyn pozwala na zapisanie i odtwarzanie danych specyficznych dla użytkownika i aplikacji.

Nauczyłeś się również, w jaki sposób utworzyć własny ekran startowy, który pozwala aplikacjom na wykonanie inicjalizacji bez niebezpieczeństwa wyłączenia, gdy zajmuje ona ponad 15 sekund. Wspomniałem też o danych aplikacji, demonstrując lokalny system plików. Aplikacje Windows 8 mogą tworzyć dane i korzystać z nich na wiele sposobów, od usług sieciowych do plików lokalnych. W następnym rozdziale, „Dane", przedstawię dokładniej obsługę danych.

Skorowidz

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ź

Ż

Poznaj kompletny proces przygotowywania i publikowania aplikacji w Sklepie Windows!

Windows 8 na dobre zadomowił się na komputerach użytkowników. Jego nowy interfejs budzi wiele kontrowersji — jedni go wychwalają, drugim ciężko się do niego przyzwyczaić. Pewne jest jednak, że to podejście do interfejsu użytkownika stanie się standardem na kolejne lata. Dlatego już teraz warto poznać zasady tworzenia aplikacji dla nowej platformy.

W tym zadaniu pomoże Ci ten znakomity podręcznik, który jako pierwszy przedstawia kompletny proces tworzenia aplikacji dla Windows 8 — od projektu aż po publikację w Sklepie Windows. W trakcie lektury zobaczysz, jak wykorzystać potencjał najnowszej wersji Visual Studio 2012. Ponadto poznasz wzorzec MVVM, bibliotekę WinRT oraz sposoby wykorzystywania zasobów sieci w Twojej aplikacji. Niezwykle istotny jest rozdział poświęcony testowaniu i debugowaniu aplikacji — tylko bezbłędna aplikacja ma szansę odnieść sukces w Sklepie Windows. Jak sprzedać w nim aplikację? Tego też dowiesz się z tej książki. Warto ją mieć!

Bądź na czasie i:

- przygotuj aplikację dla Windows 8
- poznaj możliwości Visual Studio 2012
- wykorzystaj bibliotekę WinRT
- opublikuj aplikację w Sklepie Windows
- odnieś sukces!

Addison-Wesley
Pearson Education

helion.pl
księgarnia internetowa

Nr katalogowy: 14213

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

cena 49,00 zł

ISBN 978-83-246-7062-8

9 788324 670628

PEARSON