



Prism for the Windows Runtime for Windows 8:

Developing a Windows Store
business app using
C#, XAML, and Prism

David Britch
Colin Campbell
Francis Cheung
Diego Antonio Poza
Rohit Sharma
Mariano Vazquez
Blaine Wastell



May 2013

patterns & practices

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet web site references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2013 Microsoft. All rights reserved.

Microsoft, Visual Basic, Visual Studio, Windows Azure, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Contents

Developing a business app for the Windows Store using C#: AdventureWorks Shopper.....	1
Download.....	1
Prerequisites	1
Table of contents at a glance	2
Learning resources.....	3
Getting started with AdventureWorks Shopper	4
Download.....	4
Building and running the sample.....	4
Projects and solution folders	5
The AdventureWorks.Shopper project.....	6
The AdventureWorks.UILogic project	7
The AdventureWorks.WebServices project	7
The Microsoft.Practices.Prism.PubSubEvents project	8
The Microsoft.Practices.Prism.StoreApps project	8
Guidance summary for AdventureWorks Shopper	9
Applies to.....	9
Making key decisions	9
Designing the AdventureWorks Shopper user experience.....	11
Using the Model-View-ViewModel (MVVM) pattern in AdventureWorks Shopper	11
Creating and navigating between pages in AdventureWorks Shopper	12
Using touch in AdventureWorks Shopper	13
Validating user input in AdventureWorks Shopper.....	13
Managing application data in AdventureWorks Shopper	14
Handling suspend, resume, and activation in AdventureWorks Shopper	14
Communicating between loosely coupled components in AdventureWorks Shopper.....	15
Working with tiles in AdventureWorks Shopper.....	16
Implementing search in AdventureWorks Shopper	17
Improving performance in AdventureWorks Shopper	18
Testing and deploying AdventureWorks Shopper.....	18
Using Prism for the Windows Runtime	19
You will learn.....	19
Applies to.....	19
Getting started	20

Creating a view.....	22
Creating a view model class.....	22
Creating a model class with validation support.....	23
Creating a Flyout and showing it programmatically.....	23
Adding items to the Settings pane.....	24
Changing the convention for naming and locating views	25
Changing the convention for naming, locating, and associating view models with views	25
Registering a view model factory with views instead of using a dependency injection container..	26
Designing the AdventureWorks Shopper user experience	27
You will learn.....	27
Applies to.....	27
Making key decisions	27
AdventureWorks Shopper user experiences	28
Deciding the user experience goals.....	28
Deciding the app flow.....	29
Deciding what Windows 8 features to use	31
Fundamentals.....	32
Page design.....	32
Snapping and scaling	32
Touch interaction	33
Capabilities.....	33
Tiles and notifications.....	33
Data	34
Deciding how to monetize the app	34
Making a good first impression	34
Validating the design.....	34
Using the Model-View-ViewModel (MVVM) pattern in AdventureWorks Shopper	35
You will learn.....	35
Applies to.....	35
Making key decisions	35
MVVM in AdventureWorks Shopper	39
What is MVVM?.....	40
Using a dependency injection container.....	40
Bootstrapping an MVVM app using the MvvmAppBase class.....	41

Using the ViewModelLocator class to connect view models to views	43
Using a convention-based approach	44
Other approaches to connect view models to views.....	44
Creating a view model declaratively	44
Creating a view model programmatically	45
Creating a view defined as a data template.....	45
Data binding with the BindableBase class.....	46
Additional considerations	47
UI interaction using the DelegateCommand class and attached behaviors.....	48
Implementing command objects	48
Invoking commands from a view	49
Implementing behaviors to supplement the functionality of XAML elements.....	50
Invoking behaviors from a view	51
Additional considerations.....	52
Centralize data conversions in the view model or a conversion layer.....	52
Expose operational modes in the view model	52
Keep views and view models independent.....	52
Use asynchronous programming techniques to keep the UI responsive	53
Creating and navigating between pages in AdventureWorks Shopper	54
You will learn.....	54
Applies to.....	54
Making key decisions	54
Creating pages and navigating between them in AdventureWorks Shopper	58
Creating pages.....	58
Adding design time data.....	60
Supporting portrait, snap, and fill layouts.....	61
Loading the hub page at runtime	61
Styling controls.....	63
Overriding built-in controls.....	63
Enabling page localization	65
Separate resources for each locale	65
Ensure that each piece of text that appears in the UI is defined by a string resource	66
Add contextual comments to the app resource file.....	66
Define the flow direction for all pages	66

Ensure error messages are read from the resource file	66
Enabling page accessibility	67
Navigating between pages.....	68
Handling navigation requests.....	70
Invoking navigation	71
Using touch in AdventureWorks Shopper	74
You will learn.....	74
Applies to.....	74
Making key decisions	74
Touch in AdventureWorks Shopper.....	76
Tap for primary action.....	76
Slide to pan	79
Swipe to select, command, and move	81
Pinch and stretch to zoom.....	84
Swipe from edge for app commands.....	86
Swipe from edge for system commands	89
Validating user input in AdventureWorks Shopper	90
You will learn.....	90
Applies to.....	90
Making key decisions	90
Validation in AdventureWorks Shopper.....	91
Specifying validation rules	92
Triggering validation when properties change	95
Triggering validation of all properties.....	97
Triggering server-side validation	98
Highlighting validation errors with attached behaviors.....	99
Persisting user input and validation errors when the app suspends and resumes.....	101
Managing application data in AdventureWorks Shopper	104
You will learn.....	104
Applies to.....	104
Making key decisions	104
Managing application data in AdventureWorks Shopper	107
Storing data in the app data stores	107
Local application data.....	108

Roaming application data	108
Storing and roaming user credentials.....	109
Temporary application data	111
Exposing settings through the Settings charm.....	111
Using model classes as data transfer objects	114
Accessing data through a web service	115
Consumption	116
Exposing data.....	116
Data formats.....	117
Consuming data	117
Caching data.....	121
Authentication.....	122
Handling suspend, resume, and activation in AdventureWorks Shopper	127
You will learn.....	127
Applies to.....	127
Making key decisions	127
Suspend and resume in AdventureWorks Shopper.....	128
Understanding possible execution states.....	129
Implementation approaches for suspend and resume.....	131
Suspending an app.....	132
Resuming an app	135
Activating an app.....	136
Other ways to close the app	138
Communicating between loosely coupled components in AdventureWorks Shopper	140
You will learn.....	140
Applies to.....	140
Making key decisions	140
Event aggregation in AdventureWorks Shopper.....	141
Event aggregation.....	142
Defining and publishing pub/sub events.....	143
Defining an event.....	143
Publishing an event	143
Subscribing to events.....	144
Default subscription	144

Subscribing on the UI thread.....	144
Subscription filtering	145
Subscribing using strong references.....	146
Unsubscribing from pub/sub events.....	147
Working with tiles in AdventureWorks Shopper	148
You will learn.....	148
Applies to.....	148
Making key decisions	148
Tiles in AdventureWorks Shopper	149
Creating app tiles.....	150
Using periodic notifications to update tile content	151
Creating secondary tiles	152
Launching the app from a secondary tile.....	156
Implementing search in AdventureWorks Shopper	157
You will learn.....	157
Applies to.....	157
Making key decisions	157
Search in AdventureWorks Shopper.....	158
Participating in the Search contract.....	159
Responding to search queries	160
Populating the search results page with data	162
Navigating to the result's detail page.....	163
Enabling users to type into the search box.....	164
Improving performance in AdventureWorks Shopper	166
You will learn.....	166
Applies to.....	166
Making key decisions	166
Performance considerations.....	168
Limit the startup time.....	168
Emphasize responsiveness.....	169
Trim resource dictionaries	169
Optimize the element count	169
Reuse identical brushes.....	169
Use independent animations	169

Minimize the communication between the app and the web service.....	170
Limit the amount of data downloaded from the web service	170
Use UI virtualization	170
Avoid unnecessary termination.....	171
Keep your app's memory usage low when it's suspended	171
Reduce battery consumption	172
Minimize the amount of resources that your app uses.....	172
Limit the time spent in transition between managed and native code	172
Reduce garbage collection time.....	172
Additional considerations.....	173
Testing and deploying AdventureWorks Shopper	174
You will learn.....	174
Applies to.....	174
Making key decisions	174
Testing AdventureWorks Shopper.....	175
Unit and integration testing.....	176
Testing synchronous functionality	177
Testing asynchronous functionality	178
Suspend and resume testing.....	179
Security testing.....	179
Localization testing.....	179
Accessibility testing.....	180
Performance testing.....	180
Device testing.....	180
Testing your app with the Windows App Certification Kit	181
Creating a Windows Store certification checklist.....	182
Deploying and managing Windows Store apps.....	182
Meet the AdventureWorks Shopper team	183
Meet the team	183
Quickstarts for AdventureWorks Shopper	185
Validation Quickstart for Windows Store apps using the MVVM pattern.....	186
You will learn.....	186
Applies to.....	186
Building and running the Quickstart	186

Solution structure	187
Key classes in the Quickstart.....	188
Specifying validation rules	189
Triggering validation explicitly	190
Triggering validation implicitly on property change	191
Highlighting validation errors.....	191
Event aggregation Quickstart for Windows Store apps.....	194
You will learn.....	194
Applies to.....	194
Building and running the Quickstart	195
Solution structure	196
Key classes in the Quickstart.....	196
Defining the ShoppingCartChangedEvent class	197
Notifying subscribers of the ShoppingCartChangedEvent	198
Registering to receive notifications of the ShoppingCartChangedEvent.....	199
Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime	201
You will learn.....	201
Applies to.....	201
Building and running the Quickstart	201
Solution structure	202
Key classes in the Quickstart.....	203
Bootstrapping an MVVM app using the MvvmAppBase class.....	203
Adding app specific startup behavior to the App class.....	204
Bootstrapping without a dependency injection container.....	207
Prism for the Windows Runtime reference	208
You will learn.....	208
Applies to.....	208
Microsoft.Practices.Prism.StoreApps library	209
Microsoft.Practices.Prism.PubSubEvents library	211

Developing a business app for the Windows Store using C#: AdventureWorks Shopper

This guide provides guidance to developers who want to create a Windows Store business app using C#, Extensible Application Markup Language (XAML), the Windows Runtime, and modern development practices. The guide comes with source code for [Prism for the Windows Runtime](#), source code for the AdventureWorks Shopper product catalog and shopping cart reference implementation, and documentation. The guide provides guidance on how to implement MVVM with navigation and app lifecycle management, validation, manage application data, implement controls, accessible and localizable pages, touch, search, tiles, and tile notifications. It also provides guidance on testing your app and tuning its performance.

Download

Download AdventureWorks Shopper sample

Download Prism StoreApps library

Download Prism PubSubEvents library

After you download the code, see [Getting started with AdventureWorks Shopper](#) for instructions on how to compile and run the reference implementation, as well as understand the Microsoft Visual Studio solution structure.

Here's what you'll learn:

- How to implement pages, controls, touch, navigation, settings, suspend/resume, search, tiles, and tile notifications.
- How to implement the Model-View-ViewModel (MVVM) pattern.
- How to validate user input for correctness.
- How to manage application data.
- How to test your app and tune its performance.

Note If you're just getting started with Windows Store apps, read [Create your first Windows Store app using C# or Visual Basic](#) to learn how to create a simple Windows Store app with C# and XAML. Then download the AdventureWorks Shopper reference implementation to see a complete business app that demonstrates recommended implementation patterns.

Prerequisites

- Windows 8
- Microsoft Visual Studio 2012
- An interest in C# and XAML programming

Go to [Windows Store app development](#) to download the latest tools for Windows Store app development.

The AdventureWorks Shopper Visual Studio solution has a number of nuget package dependencies, which Visual Studio will attempt to download when the solution is first loaded. The required nuget packages are:

- Unity v3.0
- Microsoft.AspNet.WebApi.Client v4.1.0-alpha-120809
- Newtonsoft.Json v4.5.11
- Microsoft.AspNet.Mvc v4.0.20710.0
- Microsoft.AspNet.Razor v2.0.20715.0
- Microsoft.AspNet.WebApi v4.0.20710.0
- Microsoft.AspNet.WebApi.Client v4.1.0-alpha-120809
- Microsoft.AspNet.WebApi.Core v4.0.20710.0
- Microsoft.AspNet.WebApi.WebHost v4.0.20710.0
- Microsoft.AspNet.WebPages v2.0.20710.0
- Microsoft.Net.Http v2.0.20710.0
- Microsoft.Web.Infrastructure v1.0.0.0

Table of contents at a glance

Here are the major topics in this guide. For the full table of contents, see [AdventureWorks Shopper table of contents](#).

- [Getting started with AdventureWorks Shopper](#)
- [Guidance summary for AdventureWorks Shopper](#)
- [Using Prism for the Windows Runtime](#)
- [Designing the AdventureWorks Shopper user experience](#)
- [Using the Model-View-ViewModel \(MVVM\) pattern in AdventureWorks Shopper](#)
- [Creating and navigating between pages in AdventureWorks Shopper](#)
- [Using touch in AdventureWorks Shopper](#)
- [Validating user input in AdventureWorks Shopper](#)
- [Managing application data in AdventureWorks Shopper](#)
- [Handling suspend, resume, and activation in AdventureWorks Shopper](#)
- [Communicating between loosely coupled components in AdventureWorks Shopper](#)
- [Working with tiles in AdventureWorks Shopper](#)
- [Implementing search in AdventureWorks Shopper](#)
- [Improving performance in AdventureWorks Shopper](#)
- [Testing and deploying AdventureWorks Shopper](#)
- [Meet the AdventureWorks Shopper team](#)
- [Quickstarts for AdventureWorks Shopper](#)
- [Prism for the Windows Runtime reference](#)

Note This content is available on the web as well. For more info, see [Developing a business app for the Windows Store using C#: AdventureWorks Shopper](#).

Learning resources

If you're new to C# programming for Windows Store apps, read [Roadmap for Windows Store app using C# or Visual Basic](#). To find out about debugging Windows Store apps see [Debugging Windows Store apps](#).

If you're familiar with using XAML you'll be able to continue using your skills when you create Windows Store apps. For more info about XAML as it relates to Windows Store apps, see [XAML overview](#).

The Windows Runtime is a programming interface that you can use to create Windows Store apps. The Windows Runtime supports the distinctive visual style and touch-based interaction model of Windows Store apps as well as access to network, disks, devices, and printing. For more info about the Windows Runtime API, see [Windows API reference for Windows Store apps](#).

The .NET framework provides a subset of managed types that you can use to create Windows Store apps using C#. This subset of managed types is called .NET for Windows Store apps and enables .NET framework developers to create Windows Store apps within a familiar programming framework. You use these managed types with types from the Windows Runtime API to create Windows Store apps. You won't notice any differences between using the managed types and the Windows Runtime types except that the managed types reside in namespaces that start with **System**, and the Windows Runtime types reside in namespaces that start with **Windows**. The entire set of assemblies for .NET for Windows Store apps is automatically referenced in your project when you create a Windows Store app using C#. For more info see [.NET for Windows Store apps overview](#).

To learn about the components and tools that determine what platform capabilities are available to your app, and how to access these capabilities see [App capability declarations \(Windows Store apps\)](#).

The AdventureWorks Shopper reference implementation makes much use of the task-based asynchronous pattern (TAP). To learn how to use TAP to implement and consume asynchronous operations see [Task-based Asynchronous Pattern](#).

You might also want to read [Index of UX guidelines for Windows Store apps](#) and [Blend for Visual Studio](#) to learn more about how to implement a great user experience.

Getting started with AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

In this article we explain how to build and run the AdventureWorks Shopper reference implementation, and how the source code is organized. The reference implementation demonstrates how to create a Windows Store business app by using [Prism for the Windows Runtime](#) to accelerate development.

Download

Download AdventureWorks Shopper sample

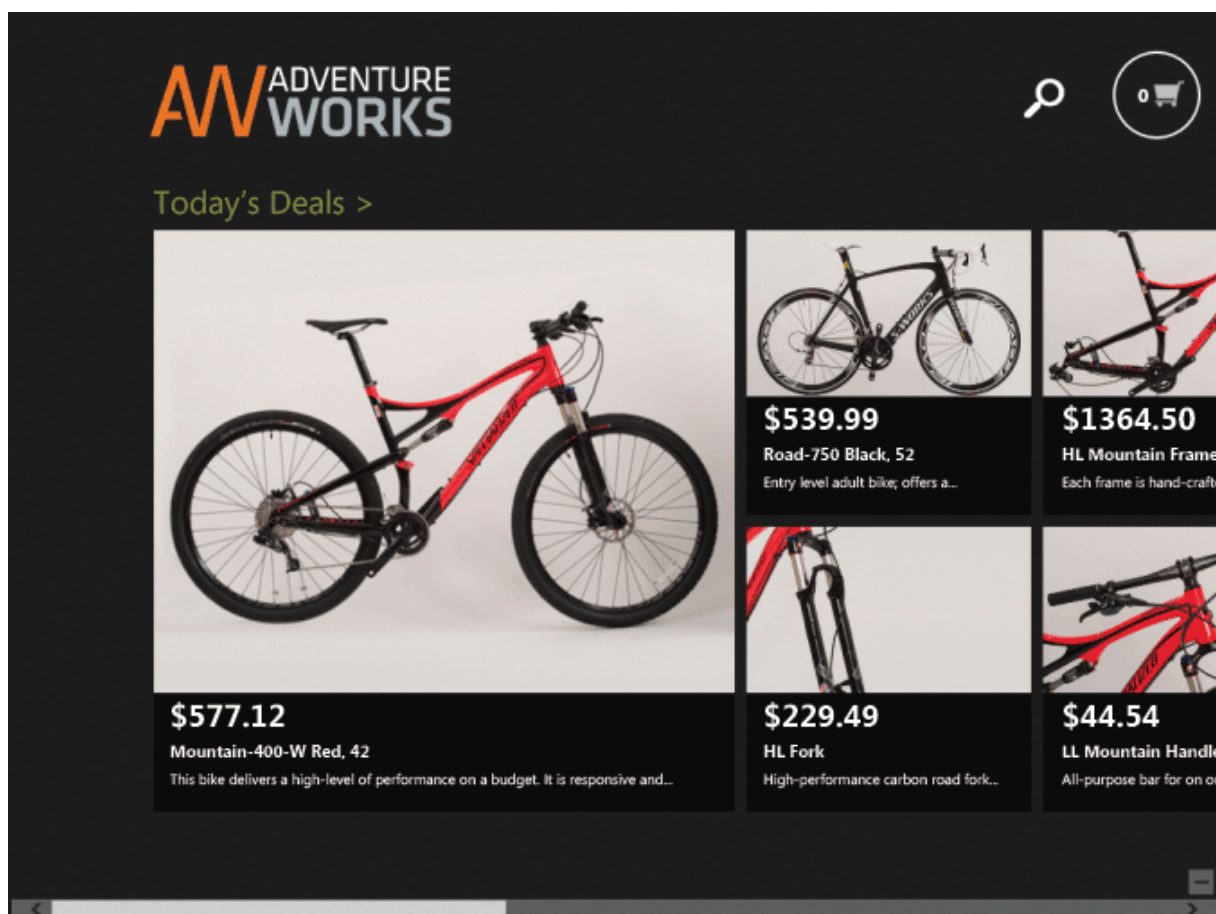
Download Prism StoreApps library

Download Prism PubSubEvents library

Building and running the sample

Build the AdventureWorks Shopper Microsoft Visual Studio solution as you would build a standard solution.

1. On the Visual Studio menu bar, choose **Build > Build Solution**.
2. After you build the solution, you must deploy it. On the menu bar, choose **Build > Deploy Solution**. Visual Studio also deploys the project when you run the app from the debugger.
3. After you deploy the project, you should run it. On the menu bar, choose **Debug > Start Debugging**. Make sure that AdventureWorks.Shopper is the startup project. When you run the app, the hub page appears.



Projects and solution folders

The AdventureWorks Shopper Visual Studio solution organizes the source code and other resources into projects. All of the projects use Visual Studio solution folders to organize the source code and other resources into categories. The following table outlines the projects that make up the AdventureWorks Shopper reference implementation.

Project	Description
AdventureWorks.Shopper	This project contains the views for the AdventureWorks Shopper reference implementation, the package manifest, and the App class that defines the startup behavior of the app, along with supporting classes and resources. For more info see The AdventureWorks.Shopper project .
AdventureWorks.UILogic	This project contains the business logic for the AdventureWorks Shopper reference implementation, and comprises view models, models, repositories, and service classes. For more info see The AdventureWorks.UILogic project .
AdventureWorks.WebServices	This project contains the web service for the AdventureWorks Shopper reference implementation. For more info see The AdventureWorks.WebServices project .

Microsoft.Practices.Prism.PubSubEvents	This project contains classes that implement the event aggregator. For more info see The Microsoft.Practices.Prism.PubSubEvents project .
Microsoft.Practices.Prism.StoreApps	This project contains interfaces and classes that provide MVVM support with lifecycle management, and core services to the AdventureWorks Shopper reference implementation. For more info see The Microsoft.Practices.Prism.StoreApps project .
AdventureWorks.Shopper.Tests	This project contains unit tests for the AdventureWorks.Shopper project.
AdventureWorks.UILogic.Tests	This project contains unit tests for the AdventureWorks.UILogic project.
AdventureWorks.WebServices.Tests	This project contains unit tests for the AdventureWorks.WebServices project.
Microsoft.Practices.Prism.PubSubEvents.Tests	This project contains unit tests for the Microsoft.Practices.Prism.PubSubEvents project.
Microsoft.Practices.Prism.StoreApps.Tests	This project contains unit tests for the Microsoft.Practices.Prism.StoreApps project.

You can reuse some of the components in the AdventureWorks Shopper reference implementation in any Windows Store app with little or no modification. For your own app, you can adapt the organization and ideas that these files provide.

The AdventureWorks.Shopper project

The AdventureWorks.Shopper project contains the following folders:

- The **Assets** folder contains images for the splash screen, tile, and other Windows Store app required images.
- The **Behaviors** folder contains attached behaviors that are exposed to view classes.
- The **Common** folder contains the **DependencyPropertyChangedHelper** class which monitors a dependency property for changes, and standard styles used by the app.
- The **Controls** folder contains the **FormFieldTextBox** and **MultipleSizedGridView** controls.
- The **Converters** folder contains data converters such as the **BooleanToVisibilityConverter** and the **NullToVisibleConverter**.
- The **DesignViewModels** folder contains design-time view model classes that are used to display sample data in the visual designer.
- The **Services** folder contains the **AlertMessageService** and **SecondaryTileService** classes.
- The **Strings** folder contains resource strings used by this project, with subfolders for each supported locale.
- The **Themes** folder contains the application styles used by the app.
- The **Views** folder contains the pages and Flyouts for the app. The app uses a default convention that attempts to locate pages in the "Views" namespace.

The AdventureWorks.UILogic project

The AdventureWorks.UILogic project contains the model, repository, service, and view model classes. Placing the model and view model classes into a separate assembly provides a simple mechanism for ensuring that view models are independent from their corresponding views.

The AdventureWorks.UILogic project contains the following folders:

- The **Models** folder contains the entities that are used by view model classes.
- The **Repositories** folder contains repository classes that access the web service.
- The **Services** folder contains interfaces and classes that implement services that are provided to the app, such as the **AccountService** and **TemporaryFolderCacheService** classes.
- The **Strings** folder contains resource strings used by this project, with subfolders for each supported locale.
- The **ViewModels** folder contains the application logic that is exposed to XAML controls. When a view class is associated with a view model class a default convention is used which will attempt to locate the view model class in the "ViewModels" namespace.

The AdventureWorks.WebServices project

The AdventureWorks.WebServices project is a sample web service that uses an in-memory database to provide data to the AdventureWorks Shopper reference implementation. When the reference implementation is deployed through Visual Studio this web service is deployed locally on the ASP.NET development server.

The AdventureWorks.WebServices project contains the following folders:

- The **App_Start** folder contains the configuration logic for the web service.
- The **Controllers** folder contains the controller classes used by the web service.
- The **Images** folder contains product images.
- The **Models** folder contains the entities that are used by the web service. These entities contain the same properties as the entities in the AdventureWorks.UILogic project, with some containing additional validation logic.
- The **Repositories** folder contains the repository classes that implement the in-memory database used by the web service.
- The **Strings** folder contains a resource file containing strings used by the web service.
- The **Views** folder contains the Web.config settings and configuration file for the web service. It does not contain views because it uses the ASP.NET Web API, which returns data rather than displays views.

Note The AdventureWorks.WebServices project does not provide guidance for building a web service.

The `Microsoft.Practices.Prism.PubSubEvents` project

The [Microsoft.Practices.Prism.PubSubEvents](#) project is a Portable Class Library that contains classes that implement event aggregation. You can use this library for communicating between loosely coupled components in your own app. The project has no dependencies on any other projects. For more info about this library, see [Prism for the Windows Runtime reference](#).

The `Microsoft.Practices.Prism.StoreApps` project

This project contains the reusable infrastructure of the AdventureWorks Shopper reference implementation, which you can use for building your own Windows Store app. It contains classes to build Windows Store apps that support MVVM, navigation, state management, validation, Flyouts, and commands.

The [Microsoft.Practices.Prism.StoreApps](#) project uses Visual Studio solution folders to organize the source code and other resources into these categories:

- The **Interfaces** folder contains the interfaces that are implemented by classes in this project.
- The **Strings** folder contains resource strings used by this project, with subfolders for each supported locale.

For more info about this library, see [Prism for the Windows Runtime reference](#).

Guidance summary for AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Business apps create a unique set of challenges for developers. In this article read about the key decisions you will have to make when developing a Windows Store business app. In addition, you can consult the checklists that provide a consolidated view of the guidance included with the documentation and illustrated in the AdventureWorks Shopper reference implementation.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

This guidance provides information to developers who want to create a Windows Store app using C#, XAML, the Windows Runtime, and modern development practices. When you develop a new Windows Store app, you need to determine some key factors that will define the architecture of your app. The following are many of the key decisions that you will need to make:

- **Decide on the design of the end user experience**. When planning Windows Store apps, you should think more about what experience you want to provide to your users and less about what Windows 8 features you want to include. For more info see [Designing the user experience](#).
- **Decide whether to use a dependency injection container**. Dependency injection containers reduce the dependency coupling between objects by providing a facility to construct instances of classes with their dependencies injected, and manage their lifetime based on the configuration of the container. You will need to decide whether to use a dependency injection container, which container to use, and how to register the lifetime of components. For more info see [Using the Model-View-ViewModel pattern](#).
- **Decide whether to provide a clean separation of concerns between the user interface controls and their logic**. One of the most important decisions when creating a Windows Store app is whether to place business logic in code-behind files, or whether to create a clean separation of concerns between the user interface controls and their logic, in order to make the app more maintainable and testable. If you decide to provide a clean separation of concerns, there are then many decisions to be made about how to do this. For more info see [Using the Model-View-ViewModel pattern](#).
- **Decide how to create pages and navigate between them**. There are many decisions to be made about page design including the page layout, what content should be displayed in different page views, whether to include design time data on your pages, and whether to make pages localizable and accessible. In addition, you must also make decisions about page navigation including how to invoke navigation, and where navigation logic should reside. For more info see [Creating and navigating between pages](#).

- **Choose the touch interactions that the app will support.** This includes selecting the gestures from the Windows 8 touch language that your app requires, and whether to design and implement your own custom touch interactions. For more info see [Using touch](#).
- **Decide how to validate user input for correctness.** The decision must include how to validate user input across physical tiers, and how to notify the user about validation errors. For more info see [Validating user input](#).
- **Decide how to manage application data.** This should include deciding upon which of the app data stores to use, what data to roam, deciding how to manage large data sets, how to perform authentication between your app and a web service, and how to reliably retrieve data from a web service. For more info see [Managing application data](#).
- **Decide how to manage the lifecycle of the app.** The purpose and usage patterns of your app must be carefully designed to ensure that users have the best possible experience when an app suspends and resumes. This includes deciding whether your app needs to update the UI when resuming from suspension, and whether the app should start fresh if a long period of time has elapsed since the user last accessed it. For more info see [Handling suspend, resume, and activation](#).
- **Choose between platform provided eventing and loosely coupled eventing.** Event aggregation allows communication between loosely coupled components in an app, removing the need for components to have a reference to each other. If you decide to use event aggregation, you must decide how to subscribe to events and unsubscribe from them. For more info see [Communicating between loosely coupled components](#).
- **Decide how to create tiles that are engaging for users.** A tile is an app's representation on the Start screen and allows you to present rich and engaging content to your users when the app is not running. In order to create engaging tiles you must decide on their shape and size, how to update tile content, and how often to update tile content. For more info see [Working with tiles](#).
- **Choose how to participate in search.** To add search to your app you must participate in the Search contract. When you add the Search contract, users can search your app from anywhere in their system by selecting the Search charm. However, there are still decisions to make that include whether to provide query and result suggestions, filtering, and what to display on the search results page. For more info see [Implementing search](#).
- **Consider how to improve app performance.** A well-performing app should respond to user actions quickly, with no noticeable delay. In order to deliver a well-performing app you will need to decide which tools to use to measure performance, and where to optimize code. For more info see [Improving performance](#).
- **Decide how to test and deploy the app.** Windows Store apps should undergo various modes of testing in order to ensure that reliable, high quality apps are deployed. Therefore, you will need to decide how to test your app, how to deploy it, and how to manage it after deployment. For more info see [Testing and deploying Windows Store apps](#).

Designing the AdventureWorks Shopper user experience

Good Windows Store apps share an important set of traits that provide a consistent, elegant, and compelling user experience. Planning ahead for different form factors, accessibility, monetization, and selling in the global market can reduce your development time and make it easier to create a high quality app and get it certified.

Check	Description
<input type="checkbox"/>	Created a "great at" statement to guide user experience planning.
<input type="checkbox"/>	Decided the user experiences to provide in the app.
<input type="checkbox"/>	Followed the Index of UX guidelines for Windows Store apps for the experiences the app provides.
<input type="checkbox"/>	Storyboarded the different app flows to decide how the app behaves.
<input type="checkbox"/>	Designed the app for different form factors.
<input type="checkbox"/>	Designed the app for all users regardless of their abilities, disabilities, or preferences.

For more info see [Designing the user experience](#).

Using the Model-View-ViewModel (MVVM) pattern in AdventureWorks Shopper

MVVM provides a way for developers to cleanly separate the user interface controls from their logic. This separation makes it easy to test the business logic of the app.

Check	Description
<input type="checkbox"/>	Used a dependency injection container to decouple concrete types from the code that depends on those types, if appropriate.
<input type="checkbox"/>	Used view-first composition because the app is conceptually composed of views that connect to the view models they depend upon.
<input type="checkbox"/>	Limited view model instantiation to a single class by using a view model locator object.
<input type="checkbox"/>	Used a convention-based approach for view model construction to remove the need for some boilerplate code.
<input type="checkbox"/>	Used an attached property to automatically connect views to view models.
<input type="checkbox"/>	Promoted the testability of the app by exposing commands from the view models for ButtonBase -derived controls on the views.
<input type="checkbox"/>	Promoted the testability of the app by exposing behaviors to views for non-ButtonBase-derived controls.
<input type="checkbox"/>	Supported a view model hierarchy in order to eliminate redundant code in the view model classes.

For more info see [Using the MVVM pattern](#).

Creating and navigating between pages in AdventureWorks Shopper

The app page is the focal point for designing your UI. It holds all of your content and controls. Whenever possible, you should integrate your UI elements inline into the app page. Presenting your UI inline lets users fully immerse themselves in your app and stay in context.

Check	Description
<input type="checkbox"/>	Used Visual Studio to work with the code-focused aspects of the app.
<input type="checkbox"/>	Used Blend for Microsoft Visual Studio 2012 for Windows 8 or the Visual Studio designer to work on the visual appearance of the app.
<input type="checkbox"/>	Provided flexible page layouts that support landscape, portrait, snap, and fill views.
<input type="checkbox"/>	Followed a consistent layout pattern for margins, page headers, gutter widths, and other page elements.
<input type="checkbox"/>	Maintained state in snap view and possess feature parity across states.
<input type="checkbox"/>	Used the Windows simulator to test the app on a variety of screen sizes, orientations, and pixel densities.
<input type="checkbox"/>	Added sample data to each page to easily view styling results and layout sizes, and to support the designer-developer workflow.
<input type="checkbox"/>	Incorporated accessible design principles into the pages, and planned for them to be localized.
<input type="checkbox"/>	Placed navigation logic in view model classes to promote testability.
<input type="checkbox"/>	Used commands to implement a navigation action in a view model class, for ButtonBase-derived controls.
<input type="checkbox"/>	Used attached behaviors to implement a navigation action in a view model class, for non-ButtonBase-derived controls.
<input type="checkbox"/>	Used the top app bar for navigational elements that move the user to a different page and used the bottom app bar for commands that act on the current page.
<input type="checkbox"/>	Implemented common page navigation functionality as a user control that is easily included on each page.
<input type="checkbox"/>	Used strings to specify navigation targets.

For more info see [Creating and navigating between pages](#).

Using touch in AdventureWorks Shopper

Touch interactions in Windows 8 use physical interactions to emulate the direct manipulation of UI elements and provide a more natural, real-world experience when interacting with those elements on the screen.

Check	Description
<input type="checkbox"/>	Used the Windows 8 touch language to provide a concise set of touch interactions that are used consistently throughout the system.
<input type="checkbox"/>	Used data binding to connect standard Windows controls to the view models that implement the touch interaction behavior.
<input type="checkbox"/>	Ensured that touch targets are large enough to support direct manipulation.
<input type="checkbox"/>	Provided immediate visual feedback to touch interactions.
<input type="checkbox"/>	Ensured that the app is safe to explore by making touch interactions reversible.
<input type="checkbox"/>	Avoided timed touch interactions.
<input type="checkbox"/>	Used static gestures to handle single-finger touch interactions.
<input type="checkbox"/>	Used manipulation gestures to handle dynamic multi-touch interactions.

For more info see [Using touch](#).

Validating user input in AdventureWorks Shopper

Any app that accepts input from users should ensure that the data is valid. Validation has many uses including enforcing business rules, providing responses to user input, and preventing an attacker from injecting malicious data.

Check	Description
<input type="checkbox"/>	Performed client-side validation to provide immediate feedback to users, and server-side validation to improve security and enforce business rules on the server.
<input type="checkbox"/>	Performed synchronous validation to check the range, length, and structure of user input.
<input type="checkbox"/>	Derived model classes from the ValidatableBindableBase class in order to participate in client-side validation.
<input type="checkbox"/>	Specified validation rules for model properties by adding data annotation attributes to the properties.
<input type="checkbox"/>	Used dependency properties and data binding to make validation errors visible to the user when the properties of the model objects change.
<input type="checkbox"/>	Notified users about validation errors by highlighting the control that contains the invalid data, and by displaying an error message that informs the user why the data is invalid.
<input type="checkbox"/>	Saved user input and any validation error messages when the app suspends, so that the app can resume as the user left it following reactivation.

For more info see [Validating user input](#).

Managing application data in AdventureWorks Shopper

Application data is data that the app itself creates and manages. It is specific to the internal functions or configuration of an app, and includes runtime state, user preferences, reference content, and other settings.

Check	Description
<input type="checkbox"/>	Used the application data APIs to work with application data, to make the system responsible for managing the physical storage of data.
<input type="checkbox"/>	Stored passwords in the Credential Locker only if the user has successfully signed into the app, and has opted to save passwords.
<input type="checkbox"/>	Used ASP.NET Web API to create a resource-oriented web service that can pass different content types.
<input type="checkbox"/>	Cached web service data locally when accessing data that rarely changes.

For more info see [Managing application data](#).

Handling suspend, resume, and activation in AdventureWorks Shopper

Windows Store apps should be designed to suspend when the user switches away from them and resume when the user switches back to them.

Check	Description
<input type="checkbox"/>	Saved application data when the app is being suspended.
<input type="checkbox"/>	Saved the page state to memory when navigating away from a page.
<input type="checkbox"/>	Allowed views and view models to save and restore state that's relevant to each.
<input type="checkbox"/>	Updated the UI when the app resumes if the content has changed.
<input type="checkbox"/>	Used the saved application data to restore the app state, when the app resumes after being terminated.

For more info see [Handling suspend, resume, and activation](#).

Communicating between loosely coupled components in AdventureWorks Shopper

Event aggregation allows communication between loosely coupled components in an app, removing the need for components to have a reference to each other.

Check	Description
<input type="checkbox"/>	Used Microsoft .NET events for communication between components that have object reference relationships.
<input type="checkbox"/>	Used event aggregation for communication between loosely coupled components.
<input type="checkbox"/>	Used the Microsoft.Practices.Prism.PubSubEvents library to communicate between loosely coupled components.
<input type="checkbox"/>	Defined a pub/sub event by creating an empty class that derives from the <code>PubSubEvent<TPayload></code> class.
<input type="checkbox"/>	Notified subscribers by retrieving the event from the event aggregator and called its <code>Publish</code> method.
<input type="checkbox"/>	Registered to receive notifications by using one of the <code>Subscribe</code> method overloads available in the <code>PubSubEvent<TPayload></code> class.
<input type="checkbox"/>	Request that notification of the pub/sub event will occur in the UI thread when needing to update the UI in response to the event.
<input type="checkbox"/>	Filtered required pub/sub events by specifying a delegate to be executed once when the event is published, to determine whether or not to invoke the subscriber callback.
<input type="checkbox"/>	Used strongly referenced delegates when subscribing to a pub/sub event, where performance problems have been observed.

For more info see [Communicating between loosely coupled components](#).

Working with tiles in AdventureWorks Shopper

Tiles represent your app on the Start screen and are used to launch your app. They have the ability to display a continuously changing set of content that can be used to keep users aware of events associated with your app when it's not running.

Check	Description
<input type="checkbox"/>	Used live tiles to present engaging new content to users, which invites them to launch the app.
<input type="checkbox"/>	Made live tiles compelling by providing fresh, frequently updated content that makes users feel that the app is active even when it's not running.
<input type="checkbox"/>	Used a wide tile to display new and interesting content to the user, and periodic notifications to update the tile content.
<input type="checkbox"/>	Used peek templates to break tile content into two frames.
<input type="checkbox"/>	Set an expiration on all periodic tile notifications to ensure that the tile's content does not persist longer than it's relevant.
<input type="checkbox"/>	Updated the live tile as information becomes available, for personalized content.
<input type="checkbox"/>	Updated the live tile no more than every 30 minutes, for non-personalized content.
<input type="checkbox"/>	Allowed the user to create secondary tiles for any content that they wish to monitor.

For more info see [Working with tiles](#).

Implementing search in AdventureWorks Shopper

To add search to your app you must participate in the Search contract. When you add the Search contract, users can search your app from anywhere in their system by selecting the Search charm.

Check	Description
<input type="checkbox"/>	Used the Search charm to let users search for content in an app.
<input type="checkbox"/>	Responded to <code>OnQuerySubmitted</code> and <code>OnSearchApplication</code> notifications.
<input type="checkbox"/>	Added a search icon to the app canvas for users to get started using the app.
<input type="checkbox"/>	Implemented <i>type to search</i> for the app's hub, browse, and search full screen pages.
<input type="checkbox"/>	Disabled <i>type to search</i> before showing Flyouts, and restored it when Flyouts close.
<input type="checkbox"/>	Showed placeholder text in the search box, to describe what users can search for.
<input type="checkbox"/>	Used a ListView or GridView control to display search results.
<input type="checkbox"/>	Showed the user's query text on the search results page.
<input type="checkbox"/>	Used hit highlighting to highlight the user's query on the search results page.
<input type="checkbox"/>	Enabled users to navigate back to the last-viewed page after they look at the details for a search result.
<input type="checkbox"/>	Provided app bar navigation on the search results page.
<input type="checkbox"/>	Provided a suitable message if the search query returns no results.
<input type="checkbox"/>	Abstracted search classes that have view dependencies, to keep the app testable.
<input type="checkbox"/>	Restored page state correctly upon reactivation.
<input type="checkbox"/>	Saved the search results page for the last query in case the app is activated to search for that query again.

For more info see [Using search](#).

Improving performance in AdventureWorks Shopper

To deliver a well-performing, responsive Windows Store app you must think of performance as a feature, to be planned for and measured throughout the lifecycle of your project.

Check	Description
<input type="checkbox"/>	Performed app profiling to determine where code optimizations will have the greatest effect in reducing performance problems.
<input type="checkbox"/>	Measured app performance once you have code that performs meaningful work.
<input type="checkbox"/>	Taken performance measurements on hardware that has the lowest anticipated specification.
<input type="checkbox"/>	Optimized actual app performance and perceived app performance.
<input type="checkbox"/>	Limited the startup time of the app.
<input type="checkbox"/>	Emphasized responsiveness in the UI.
<input type="checkbox"/>	Trimmed resource dictionaries to reduce the amount of XAML the framework parses when the app starts.
<input type="checkbox"/>	Reduced the number of XAML elements on a page to make the app render faster.
<input type="checkbox"/>	Reused brushes in order to reduce memory consumption.
<input type="checkbox"/>	Used independent animations to avoid blocking the UI thread.
<input type="checkbox"/>	Minimized the communication between the app and the web service.
<input type="checkbox"/>	Limited the amount of data downloaded from the web service.
<input type="checkbox"/>	Used UI virtualization to only load into memory those UI elements that are near the viewport.
<input type="checkbox"/>	Avoided unnecessary app termination.
<input type="checkbox"/>	Kept the app's memory usage low when it's suspended.
<input type="checkbox"/>	Reduced the battery consumption of the app.
<input type="checkbox"/>	Minimized the amount of resources that the app uses.
<input type="checkbox"/>	Limited the time spent in transition between managed and native code.
<input type="checkbox"/>	Reduced garbage collection time.

For more info see [Improving performance](#).

Testing and deploying AdventureWorks Shopper

Testing helps to ensure that an app is reliable, correct, and of high quality.

Check	Description
<input type="checkbox"/>	Performed unit testing, integration testing, user interface testing, suspend and resume testing, security testing, localization testing, accessibility testing, performance testing, device testing, and Windows certification testing.
<input type="checkbox"/>	Validated and test a release build of the app by using the Windows App Certification Kit.

For more info see [Testing and deploying AdventureWorks Shopper](#).

Using Prism for the Windows Runtime (Windows Store business apps using C#, XAML, and Prism)

Summary

- Use Prism to implement the Model-View-ViewModel (MVVM) pattern in your Windows Store app.
- Use Prism to add validation support to your model classes.
- Use Prism to implement Flyouts and add items to the Settings pane.

[Prism for the Windows Runtime](#) provides two libraries that help developers create managed Windows Store apps. The libraries accelerate development by providing support for bootstrapping MVVM apps, state management, validation of user input, navigation, event aggregation, data binding, commands, Flyouts, settings, and search.

You will learn

- How to accelerate the development of your Windows Store app by using Prism.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

This article describes the general steps a developer needs to perform to use [Prism](#) to accomplish different tasks. It is not meant to provide you with detailed steps required to complete a task. If you require more info, each section has links to the relevant documentation.

Many of the topics in this article assume that you are using the [Unity](#) dependency injection container, and that you are using conventions defined by Prism. This guidance is provided to make it easier for you to understand how to get started with Prism. However, you are not required to use Unity, or any other dependency injection container, and you do not have to use the default conventions to associate views and view models. To understand how to use Prism without a dependency injection container, or change the default conventions, see [Changing the convention for naming and locating views](#), [Changing the convention for naming, locating, and associating view models with views](#), [Registering a view model factory with views instead of using a dependency injection container](#).

For more info about the conventions defined by Prism, see [Using a convention-based approach](#). For more info about Prism, see [Prism for the Windows Runtime reference](#).

Getting started

The following procedure shows how to update a Windows Store app to use the services provided by Prism.

1. Add a reference to the [Microsoft.Practices.Prism.StoreApps](#) library to your project to use the services provided by the library.
2. Derive the **App** class from the **MvvmAppBase** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, in order to gain support for MVVM and the core services required by Windows Store apps.
3. Delete the **OnLaunched** and **OnSuspending** methods from the **App** class, as these methods are provided by the **MvvmAppBase** class.
4. Override the **OnLaunchApplication** abstract method of the **MvvmAppBase** class, in the **App** class, and add code to navigate to the first page of the app.

C#

```
protected override void OnLaunchApplication(LaunchActivatedEventArgs args)
{
    NavigationService.Navigate("PageName", null);
}
```

5. **Note** *PageName* should be without the "Page" suffix. For example, use *Home* for *HomePage*.
6. Add a reference to the [Unity](#) library to your project to use the Unity dependency injection container.

Note The [Microsoft.Practices.Prism.StoreApps](#) library is not dependent on the [Unity](#) library. To avoid using a dependency injection container see [Registering a view model factory with views instead of using a dependency injection container](#).

7. Create an instance of the **UnityContainer** class in the **App** class, so that you can use the Unity dependency injection container to register and resolve types and instances.

C#

```
private readonly IUnityContainer _container = new UnityContainer();
```

8. Override the **OnRegisterKnownTypesForSerialization** method in the **App** class to register any non-primitive types that need to be saved and restored to survive app termination.

C#

```
SessionStateService.RegisterKnownType(typeof(Address));
```

9. Override the **OnInitialize** method in the **App** class in order to register types for the Unity container and perform any other initialization. Examples of app specific initialization behavior include:
 - Registering infrastructure services.
 - Registering types and instances that you use in constructors.
 - Providing a delegate that returns a view model type for a given view type.

C#

```
protected override void OnInitialize(IActivatedEventArgs args)
{
    _container.RegisterInstance(NavigationService);
    _container.RegisterType<IAccountService, AccountService>
        (new ContainerControlledLifetimeManager());
    _container.RegisterType<IShippingAddressUserControlViewModel,
        ShippingAddressUserControlViewModel>();

    ViewModelLocator.SetDefaultViewTypeToViewModelTypeResolver((viewType)
        =>
        {
            ...
            return viewModelType;
        });
}
```

10. **Note** For a detailed example of an **OnInitialize** method see the **App** class in the AdventureWorks Shopper reference implementation.
11. Override the **Resolve** method in the **App** class to return a constructed view model instance.

C#

```
protected override object Resolve(Type type)
{
    return _container.Resolve(type);
}
```

For more info see [Using the MVVM pattern](#), [Registering a view model factory with views instead of using a dependency injection container](#), [Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime](#), [Creating and navigating between pages](#), and [Prism for the Windows Runtime reference](#).

Creating a view

The following procedure shows how to create a view class that has support for layout changes, navigation, and state management.

1. Complete the [Getting started](#) procedure.
2. Add a folder named **Views** to the root folder of your project.
3. Create a new page in the **Views** folder whose name ends with "Page," in order to use the **FrameNavigationService's** default convention to navigate to pages in the **Views** folder.
4. Modify the page class to derive from the **VisualStateAwarePage** class, which provides support for layout changes, navigation, and state management.
5. Add the **ViewModelLocator.AutoWireViewModel** attached property to your view XAML in order to use the **ViewModelLocator** class to instantiate the view model class and associate it with the view class.

XAML

```
prism:ViewModelLocator.AutoWireViewModel="true"
```

6. Override the **OnNavigatedTo** and **OnNavigatedFrom** methods if your page class needs to perform additional logic, such as subscribing to an event or unsubscribing from an event, when page navigation occurs. Ensure that the **OnNavigatedTo** and **OnNavigatedFrom** overrides call **base.OnNavigatedTo** and **base.OnNavigatedFrom**, respectively.
7. Override the **SaveState** and **LoadState** methods if you have view state, such as scroll position, that needs to survive termination and be restored when the app is reactivated.

For more info see [Creating and navigating between pages](#), [Using the MVVM pattern](#), and [Handling suspend, resume, and activation](#).

Creating a view model class

The following procedure shows how to create a view model class that has support for property change notification, navigation, and state management.

1. Complete the [Getting started](#) procedure.
2. Add a folder named **ViewModels** to the root folder of your project.
3. Create a new class in the **ViewModels** folder whose name corresponds with the name of a view and ends with "ViewModel," in order to use the **ViewModelLocator's** default convention to instantiate and associate view model classes with view classes.
4. Derive the view model class from the **ViewModel** base class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, so that you can use the base class's implementation of the [INotifyPropertyChanged](#) interface and gain support for navigation and state management.
5. Modify the view model constructor so that it accepts the services required by the view model, such as an **INavigationService** instance.

6. Annotate properties with the **[RestorableState]** custom attribute if you want their values to survive termination.

For more info see [Using the MVVM pattern](#).

Creating a model class with validation support

The following procedure shows how to create a model class that has support for validation. You should complete the [Getting started](#) procedure before starting this procedure.

1. Add a model class to your project and derive the model class from the **ValidatableBindableBase** class, which provides validation support.
2. Add a property to the model class and add the appropriate attributes that derive from the [ValidationAttribute](#) attribute, in order to specify the client side validation.

C#

```
[Required(ErrorMessage = "First name is required.")]
public string FirstName
{
    get { return _firstName; }
    set { SetProperty(ref _firstName, value); }
}
```

3. Update the view XAML that binds to the property created in the previous step to show validation error messages.

XAML

```
<TextBox Text="{Binding UserInfo.FirstName, Mode=TwoWay}"
          behaviors:HighlightOnErrors.PropertyErrors=
            "{Binding UserInfo.Errors[FirstName]}" />
```

Note The **HighlightOnErrors** attached behavior can be found in the AdventureWorks Shopper reference implementation.

For more info [Validating user input](#) and [Validation Quickstart](#).

Creating a Flyout and showing it programmatically

The following procedure shows how to create a Flyout view that appears from the right side of the screen.

1. Complete the [Getting started](#) procedure.
2. Create a new page in the **Views** folder whose name ends with "Flyout," in order to use the **FlyoutService's** default convention to show Flyouts in the **Views** folder.

3. Derive the page from the **FlyoutView** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, in order to display the view as a Flyout.
4. Modify the **Flyout** view constructor to specify the width of the **Flyout**. The **StandardFlyoutSize** class provides the two standard sizes for Flyouts.

C#

```
public CustomSettingFlyout() : base(StandardFlyoutSize.Narrow)
{
    this.InitializeComponent();
}
```

5. Pass the **IFlyoutService** instance as a constructor parameter to the view model class that needs to show the Flyout. Then, use the **FlyoutService.ShowFlyout** method to programmatically display the Flyout from the view model class.

C#

```
FlyoutService.ShowFlyout("CustomSetting");
```

For more info see [Creating and navigating between pages](#) and [Managing application data](#).

Adding items to the Settings pane

The following procedure shows how to add an item to the Settings pane that can invoke an action.

1. Complete the [Getting started](#) procedure.
2. Override the **GetSettingsCharmActionItems** method in the **App** class and add code to add items to the Settings pane.

C#

```
protected override IList<SettingsCharmActionItem>
GetSettingsCharmActionItems()
{
    var settingsCharmItems = new List<SettingsCharmActionItem>();
    settingsCharmItems.Add(new SettingsCharmActionItem("Text to show in
        Settings pane", ActionToBePerformed));
    settingsCharmItems.Add(new SettingsCharmActionItem("Custom setting",
        () => FlyoutService.ShowFlyout("CustomSetting")));
    return settingsCharmItems;
}
```

For more info see [Managing application data](#).

Changing the convention for naming and locating views

The following procedure shows how to configure the **FrameNavigationService** class to look for views in a location other than the **Views** folder.

1. Complete the [Getting started](#) procedure.
2. Override the **GetPageType** method in the **App** class and add code to define the page location and naming convention appropriate to your app.

C#

```
protected override Type GetPageType(string pageToken)
{
    var assemblyQualifiedAppType = this.GetType().GetTypeInfo()
        .AssemblyQualifiedName;
    var pageNameWithParameter = assemblyQualifiedAppType.Replace(
        (this.GetType().FullName, this.GetType().Namespace +
        ".Pages.{0}View");
    var viewFullName = string.Format(CultureInfo.InvariantCulture,
        pageNameWithParameter, pageToken);
    var viewType = Type.GetType(viewFullName);
    return viewType;
}
```

For more info see [Using the MVVM pattern](#).

Changing the convention for naming, locating, and associating view models with views

The following procedure shows how to configure the **ViewModelLocator** class to look for view models in a location other than the **ViewModels** folder in the same assembly.

1. Complete the [Getting started](#) procedure.
2. Override the **OnInitialize** method in the **App** class and invoke the static **ViewModelLocator.SetDefaultViewTypeToViewModelTypeResolver** method, passing in a delegate that specifies a view type and returns a corresponding view model type.

C#

```
protected override void OnInitialize(IActivatedEventArgs args)
{
    ...
    ViewModelLocator.SetDefaultViewTypeToViewModelTypeResolver((viewType)
        =>
        {
            var viewModelTypeName = string.Format(
                CultureInfo.InvariantCulture,
                "MyProject.VMs.{0}ViewModel, MyProject, Version=1.0.0.0,
                Culture=neutral, PublicKeyToken=public_Key_Token",
```

```

        viewType.Name);
        var viewModelType = Type.GetType(viewModelTypeName);
        return viewModelType;
    });
    ...
}

```

For more info see [Using the MVVM pattern](#).

Registering a view model factory with views instead of using a dependency injection container

The following procedure shows how to configure the **ViewModelLocator** class to explicitly specify how to construct a view model for a given view type, instead of using a container for dependency resolution and construction.

1. Complete the [Getting started](#) procedure.
2. Override the **OnInitialize** method in the **App** class and register a factory with the **ViewModelLocator** class that will create a view model instance that will be associated with a view.

```

C#

protected override void OnInitialize(IActivatedEventArgs args)
{
    ...
    ViewModelLocator.Register(typeof(MyPage).ToString(), () =>
        new MyPageViewModel(NavigationService));
    ...
}

```

For more info see [Using the MVVM pattern](#) and [Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime](#).

Designing the AdventureWorks Shopper user experience (Windows Store business apps using C#, XAML, and Prism)

Summary

- Focus on the user experience and not on the features the app will have.
- Use storyboards to iterate quickly on the user experience.
- Use standard Windows features to provide a user experience that is consistent with other apps. In addition, validate the user experience with the [Index of UX guidelines for Windows Store apps](#).

In this article we explain the design process for the AdventureWorks Shopper user experience and the Windows 8 features that were used as part of the reference implementation.

You will learn

- How to plan a Windows Store app.
- How you can tie your "great at" statement to the app flow.
- How storyboards and prototypes drive user experience design.
- Which Windows 8 features to consider as you plan your app.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Good Windows Store apps share an important set of traits that provide a consistent, elegant, and compelling user experience. Planning ahead for different form factors, accessibility, monetization, and selling in the global market can reduce your development time and make it easier to create a high quality app and get it certified. The following list summarizes the decisions to make when planning your app:

- How should I plan a Windows Store app?
- What guidelines should I follow to ensure a good overall user experience?
- What experience do you want to provide to your users?
- Should the app run on different form factors?
- How do I make the app accessible to users regardless of their abilities, disabilities, or preferences?
- Should the app be available in the global market?

When planning a Windows Store app you should think more about what experience you want to provide to your users and less about what Windows 8 features you want to include. We recommend that you follow these steps:

1. Decide the user experience goals.
2. Decide the app flow.
3. Decide what Windows 8 features to include.
4. Decide how to monetize your app.
5. Make a good first impression.
6. Validate the design.

For more info see [Planning Windows Store apps](#) and [AdventureWorks Shopper user experiences](#).

There are many user experience guidelines that can help you create a good Windows Store app. However, the exact guidelines that you will follow will be dependent on the experiences present in your app. For more info see [Index of UX guidelines for Windows Store apps](#).

In order to decide what experience you want to provide to your users we recommend that create a "great at" statement to guide your user experience planning. Following this, you should design your app flow. An app flow is a set of related interactions that your users have with the app to achieve their goals. To validate the design you should follow these steps:

1. Outline the flow of the app. What interaction comes first? What interaction follows the previous interaction?
2. Storyboard the flow of the app. How should users move through the UI to complete the flow?
3. Prototype the app. Try out the app flow with a quick prototype.

For more info see [Deciding the user experience goals](#) and [Deciding the app flow](#).

Apps should be designed for different form factors, letting users manipulate the content to fit their needs and preferences. Think of landscape view first so that your app will run on all form factors, but remember that some screens rotate, so plan the layout of your content for different resolutions and screen sizes. In addition, because Windows is used worldwide, you need to design your app so that resources, such as strings and images, are separated from their code to help make localization easier. Also, your app should be available to all users regardless of their abilities, disabilities, or preferences. If you use the built-in UI controls, you can get accessibility support with little extra effort. For more info see [Deciding what Windows 8 features to use](#).

AdventureWorks Shopper user experiences

The AdventureWorks Shopper reference implementation is a shopping app, and so we wanted to design experiences that would enable users to shop easily and efficiently.

Deciding the user experience goals

Our first step was to create a "great at" statement to guide our user experience planning. Here's the "great at" statement for the AdventureWorks Shopper reference implementation:

AdventureWorks Shopper is great at letting users easily and efficiently order products from AdventureWorks.

The goal of the AdventureWorks Shopper reference implementation is not to provide a complete shopping app, but to demonstrate how to architect a Windows Store business app. We used our "great at" statement to guide the design tradeoffs as we built the app, making the focus on what our users want to do, rather than what the app can do.

Deciding the app flow

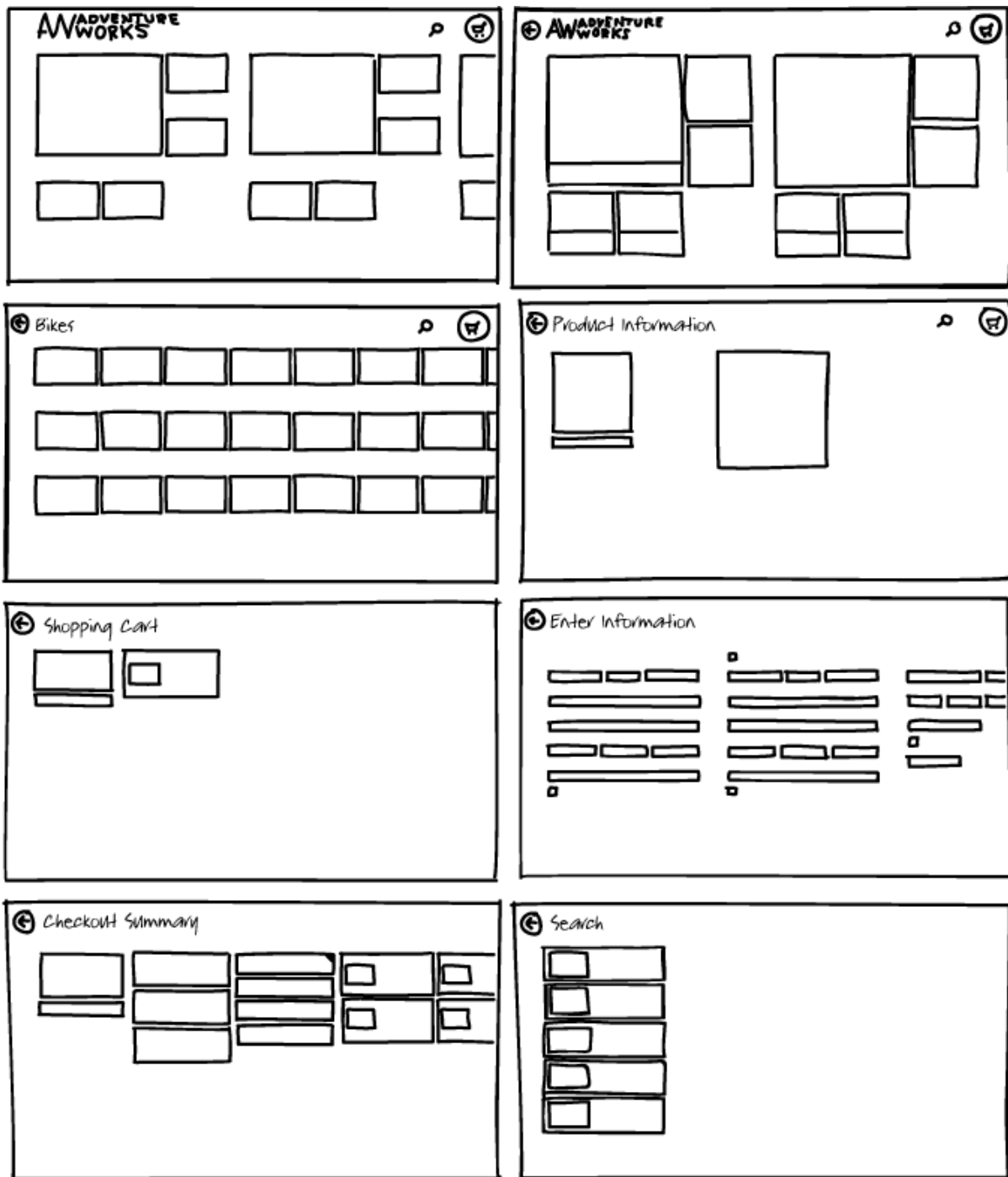
We then brainstormed which aspects of a shopping app are the most crucial for a good user experience, to let these features guide us through the design process. The features that we came up with are:

- Display and navigate products.
- Search for products.
- Authenticate user credentials.
- Validate user input.
- Order products.
- Pay for orders.
- Enable roaming data for user credentials.
- Pin products to the Start screen.

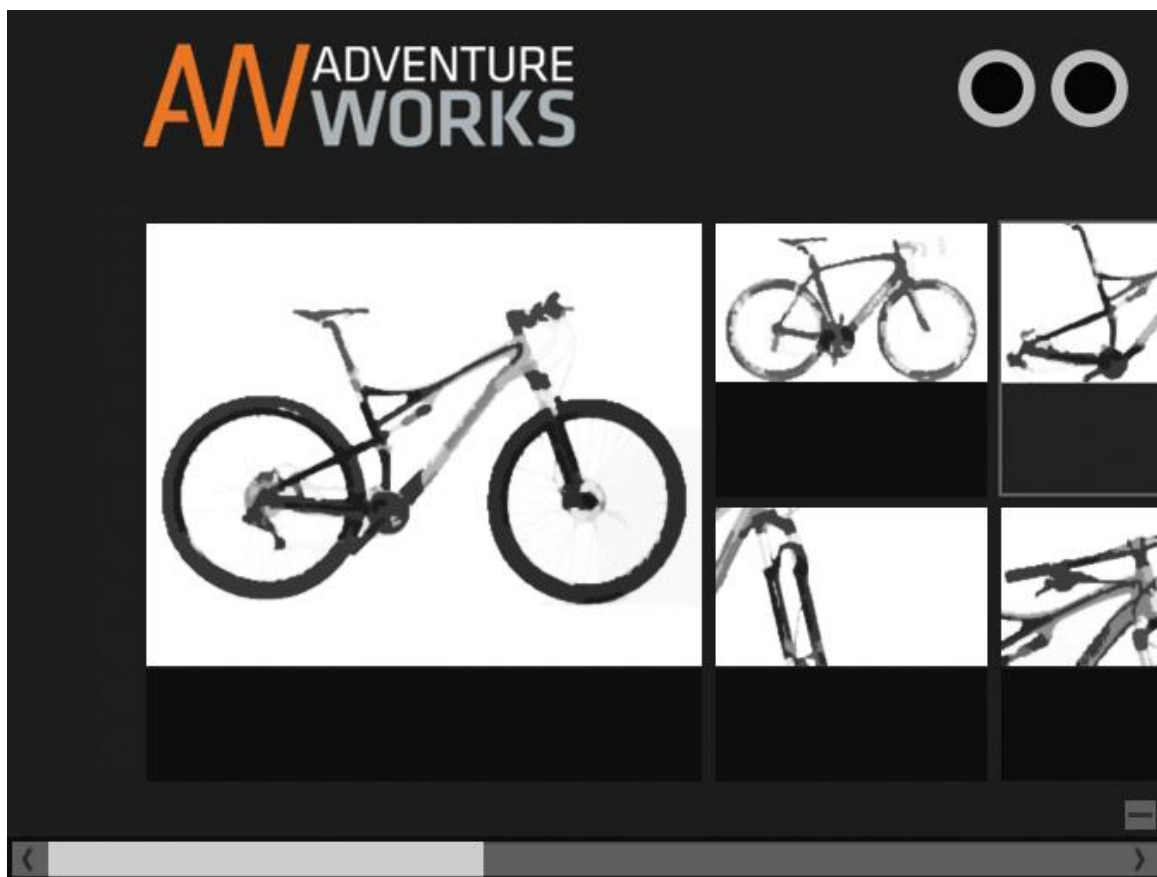
There is plenty of other functionality that we could provide in the AdventureWorks Shopper reference implementation. But we felt that the ability to browse, search, and order products best demonstrate the functionality for creating a shopping app.

The app flow is connected to our "great at" statement. A flow defines how the user interacts with the app to perform tasks. Windows Store apps should be intuitive and require as few interactions as possible. We used two techniques to help meet these goals: creating storyboards and mock-ups.

A *storyboard* defines the flow of an app. Storyboards focus on how we intend the app to behave, and not the specific details of what it will look like. Storyboards help bridge the gap between the idea of the app and its implementation, but are typically faster and cheaper to produce than prototyping the app. For the AdventureWorks Shopper reference implementation, storyboards were critical to helping us to define the app flow. This technique is commonly used in the film industry and is now becoming standard in user experience design. The following storyboard shows the main app flow for the AdventureWorks Shopper reference implementation.



A *mockup* demonstrates the flow of the user experience, but more closely resembles what the end product will look like. We created mock-ups based on our storyboards and iterated over their design as a team. These mockups also helped each team member get a feel for what the app should look like. The following mockup shows the hub page.



During the planning phase of the app, we also created small prototypes to validate feasibility. A *prototype* is a small app that demonstrates the flow of the UI or some minimal functionality. For example, a prototype could be created that only contains page navigation and commands, but doesn't implement any other functionality. By making the experience real through software, prototyping enables you to test and validate the flow of your design on devices such as tablets. You can also create prototypes that demonstrate core aspects of the app. For example, we created a prototype that performs validation of user input and notifies the user of any invalid input. Prototypes enable you to safely explore design approaches before deciding on the approach for the app. Although you can prototype during the planning phase of your app, try not to focus too much on writing code. Design the user experience that you want and then implement that design when it's ready.

For more info see [Laying out your UI](#), [Laying out an app page](#), and [Guidelines for snapped and fill views](#).

Deciding what Windows 8 features to use

When planning a new app it's important to provide an experience that's consistent with other Windows Store apps. Doing so will make your app intuitive to use. We researched the features that the Windows platform provides by looking at the [Windows Developer Center](#), and by prototyping and team discussion. We brainstormed on which platform features would best support our app flow and decided on the features outlined here.

Fundamentals

- **Splash screen.** The splash screen will be used to smooth the transition between when users launch the app and when it's ready for use. The splash screen should reinforce the AdventureWorks Shopper brand to users, rather than distract them or advertise to them. For more info see [Guidelines for splash screens](#).
- **Controls.** The app's UI will showcase its content. Distractions will be minimized by only having relevant elements on the screen so that users become immersed in the content. For more info see [Index of UX guidelines for Windows Store apps](#).
- **Suspend and resume app state.** Users will switch away from the app and back to it, and Windows will terminate it in the background when it's unused. The AdventureWorks Shopper reference implementation will save and resume state when required, in order to maintain context. This state includes the scroll position on the product catalog pages and partially entered data on the checkout pages. For more info see [Handling suspend, resume, and activation](#), and [Guidelines for app suspend and resume](#).
- **Globalization, localization, and app resources.** Because the app could be used worldwide, the app will be designed so that resources, such as strings and images, are separated from their code to help make localization easier. For more info see [Guidelines and checklist for globalizing your app](#) and [Guidelines for app resources](#).
- **Accessibility.** The app will be available to all users regardless of their abilities, disabilities, or preferences. For more info see [Plan for accessibility](#).

Page design

- **Layout and navigation.** The UI will have a layout that users can intuitively and easily navigate. For more info see [Navigation design for Windows Store apps](#).
- **Layout and commanding.** Commands will be placed consistently on the UI, to instill user confidence and to ease user interaction. For more info see [Laying out your UI](#) and [Commanding design for Windows Store apps](#).
- **Layout and page design.** Pages in the app will use a grid layout so that they adhere to the Windows 8 silhouette. For more info see [Laying out an app page](#).
- **Typography.** The app UI will be clean and uncluttered, and so will use appropriate font sizes, weights, and colors. For more info see [Guidelines for fonts](#).

Snapping and scaling

- **Flexible layouts.** The app will handle landscape and portrait orientations and let users manipulate the content to fit their needs and preferences. For more info see [Guidelines for layouts](#).
- **Snapped and fill views.** The app will be designed for users' multi-tasking needs. Users will be able to use the app while they perform tasks in another app, and so snapped views must be useful and maintain context when switching between snapped and unsnapped views. For more info see [Creating and navigating between pages](#) and [Guidelines for snapped and fill views](#).

- **Scaling to screens.** The app UI must look good on different sized devices, from small tablet screens to large desktop screens. For more info see [Guidelines for scaling to screens](#).
- **Scaling to pixel density.** Images in the app must look good when scaled. Windows scales apps to ensure consistent physical sizing regardless of the pixel density of the device. For more info see [Guidelines for scaling to pixel density](#).
- **Resizing.** The app must look good when Windows resizes it. Windows automatically resizes apps when the user changes the view state. For more info see [Guidelines for resizing](#).

Touch interaction

- **Touch interaction.** The app will provide a consistent and well-performing set of user interactions. For more info see [Using touch](#) and [Guidelines for common user interactions](#).
- **Touch targeting.** The app will provide appropriately sized and located touch targets. For more info see [Guidelines for targeting](#).
- **Visual feedback.** The app will provide clear visual feedback for user actions. For more info see [Guidelines for visual feedback](#).
- **Semantic Zoom.** The app will help users to navigate large amounts of related data. For more info see [Using touch](#) and [Guidelines for Semantic Zoom](#).
- **Swipe and cross-slide.** The app will use this standard interaction to select items from a list. For more info see [Using touch](#) and [Guidelines for cross-slide](#).
- **Panning.** The app will use this standard interaction to browse through content. For more info see [Using touch](#) and [Guidelines for panning](#).
- **Selecting text and images.** The app will use this standard interaction with content. For more info see [Using touch](#) and [Guidelines for selecting text and images](#).
- **Mouse interaction.** The app will provide a good mouse experience for users without touch screens. For more info see [Using touch](#) and [Responding to mouse interactions](#).
- **Keyboard interaction.** The app will provide a complete interaction experience for users who prefer using a keyboard. For more info see [Responding to keyboard interactions](#).

Capabilities

- **Search.** The app will let users search the app's content quickly from anywhere in the system. For more info see [Guidelines and checklist for search](#).

Tiles and notifications

- **App tiles and secondary tiles.** The app's tile will engage users, encouraging them to use the app, and keeping the app feeling fresh and relevant. In addition, you can use secondary tiles to promote interesting content from your app on the Start screen, and let users launch directly into a specific experience within your app. For more info see [Working with tiles](#), [Guidelines and checklist for tiles and badges](#), and [Guidelines and checklist for secondary tiles](#).
- **Notifications.** The app's tile will be updated with new content through periodic notifications. For more info see [Guidelines and checklist for periodic notifications](#).

Data

- **Roaming.** The app will roam the user credentials. For more info see [Managing application data](#) and [Guidelines for roaming app data](#).
- **Settings.** The app's settings will be accessible from one UI surface, so that users can configure the app through a common mechanism that they are familiar with. We decided that billing, shipping, and payment data should be accessed from the Settings charm. Initially we used Flyouts to display and enter this data, but after using the app we decided that it would be more appropriate to use a page. This removed the problem of a light dismiss on a Flyout losing any data that the user entered. For more info see [Managing application data](#) and [Guidelines for app settings](#).

Deciding how to monetize the app

Although AdventureWorks Shopper is a free app, its purpose is to drive sales for AdventureWorks through customers placing and paying for orders. In order to significantly increase the number of users who could use the app we decided to make it world-ready. Being world-ready not only means supporting localized strings and images, it also means being aware of how users from different cultures will use the app. For more info see [Guidelines and checklist for globalizing your app](#) and [Guidelines for app resources](#).

For more info about monetizing your app see [Plan for monetization](#) and [Advertising Guidelines](#).

Making a good first impression

Windows Store apps should convey their "great at" statement to users when they first launch the app. After referring back to our "great at" statement (*AdventureWorks Shopper is great at letting users easily and efficiently order products from AdventureWorks*) we realized that product promotion was key to allowing users to easily and efficiently order products from AdventureWorks. This could be enabled by:

- Having a live tile, that uses tile notifications to promote products. When a user leaves the app, we wanted to maintain a good impression by regularly updating the live tile with product offers.
- Using the splash screen to express the app's personality. We chose a splash screen image that fits the AdventureWorks branding and that reinforces the whole user experience.
- Having a home page that clearly shows the primary purpose of the app. Users will be more likely to explore the rest of the app if their initial impression is favorable.

Validating the design

Before beginning development, we presented our mockups and prototypes to stakeholders in order to gain feedback to validate and polish our design. We also cross-checked the design against the [Index of UX guidelines for Windows Store apps](#) to ensure that we complied with the Windows Store user experience guidelines. This prevented us from having to make core design changes later in the development cycle.

Using the Model-View-ViewModel (MVVM) pattern in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Use the [Microsoft.Practices.Prism.StoreApps](#) library to accelerate the development of managed Windows Store apps that use the MVVM pattern.
- Use commands to implement actions in view model classes for controls that derive from [ButtonBase](#).
- Use attached behaviors to implement actions in view model classes for controls that don't derive from [ButtonBase](#).

The Model-View-ViewModel (MVVM) pattern lends itself naturally to Windows Store apps that use XAML. The AdventureWorks Shopper reference implementation uses [Prism for the Windows Runtime](#) to provide support for MVVM. This article describes how to use Prism to implement MVVM in your Windows Store app.

You will learn

- How Windows Store apps can benefit from MVVM.
- How to use dependency injection to decouple concrete types from the code that depends on the types.
- How to bootstrap a Windows Store app that uses the MVVM pattern, by using a dependency injection container.
- How to connect view models to views.
- How a view is updated in response to changes in the underlying view model.
- How to invoke commands and behaviors from views.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

When you choose to use the MVVM pattern to construct your app, you will have to make certain design decisions that will be difficult to change later on. Generally, these decisions are app-wide and their consistent use throughout the app will improve developer and designer productivity. The following list summarizes the decisions to make when implementing the MVVM pattern:

- Should I use Prism to provide support for MVVM?
- Should I use a dependency injection container?
 - Which dependency injection container should I use?

- When is it appropriate to register and resolve components with a dependency injection container?
 - Should a component's lifetime be managed by the container?
- Should the app construct views or view models first?
- How should I connect view models to views?
 - Should I use XAML or code-behind to set the view's [DataContext](#) property?
 - Should I use a view model locator object?
 - Should I use an attached property to automatically connect view models to views?
 - Should I use a convention-based approach?
- Should I expose commands from my view models?
- Should I use behaviors in my views?
- Should I include design time data support in my views?
- Do I need to support a view model hierarchy?

Prism includes components to help accelerate the development of a managed Windows Store app that uses the MVVM pattern. It helps to accelerate development by providing core services commonly required by a Windows Store app, allowing you to focus on developing the user experiences for your app. Alternatively, you could choose to develop the core services yourself. For more info see [Prism for the Windows Runtime reference](#).

There are several advantages to using a dependency injection container. First, a container removes the need for a component to locate its dependencies and manage their lifetime. Second, a container allows mapping of implemented dependencies without affecting the component. Third, a container facilitates testability by allowing dependencies to be mocked. Forth, a container increases maintainability by allowing new components to be easily added to the system.

In the context of a Windows Store app that uses the MVVM pattern, there are specific advantages to a dependency injection container. A container can be used for registering and resolving view models and views. In addition, a container can be used for registering services, and injecting them into view models. Also, a container can create the view models and inject the views.

There are several dependency injection containers available, with two common choices being Unity and MEF. Both Unity and MEF provide the same basic functionality for dependency injection, even though they work very differently. When considering which container to use, keep in mind the capabilities shown in the following table and determine which fits your scenario better.

Both containers	Unity only	MEF only
Register types and instances with the container.	Resolves concrete types without registration.	Discovers assemblies in a directory.
Imperatively create instances of registered types.	Resolves open generics.	Recomposes properties and collections as new types are discovered.
Inject instances of registered types into constructors and properties.	Uses interception to capture calls to objects and add additional functionality to the target object.	Automatically exports derived types.
Have declarative attributes for marking types and dependencies that need to be managed.		Is deployed with the .NET Framework.
Resolve dependencies in an object graph.		

If you decide to use a dependency injection container, you should also consider whether it is appropriate to register and resolve components using the container. Registering and resolving instances from a container has a performance cost because of the container's use of reflection for creating each type, especially if components are being reconstructed for each page navigation in the app. If there are many or deep dependencies, the cost of creation can increase significantly. In addition, if the component does not have any dependencies or is not a dependency for other types, it may not make sense to put it in the container. Also, if the component has a single set of dependencies that are integral to the type and will never change, it may not make sense to put it in the container.

You should also consider whether a component's lifetime should be managed by the container. When you register a type the default behavior for the Unity container is to create a new instance of the registered type each time the type is resolved or when the dependency mechanism injects instances into other classes. When you register an instance the default behavior for the Unity container is to manage the lifetime of the object as a singleton. This means that the instance remains in scope as long as the container is in scope, and it is disposed when the container goes out of scope and is garbage-collected or when code explicitly disposes the container. If you want this singleton behavior for an object that Unity creates when you register types, you must explicitly specify the **ContainerControlledLifetimeManager** class when registering the type. For more info see [Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime](#).

If you decide not to use a dependency injection container you can use the **ViewModelLocator** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, to register view model factories for views, or infer the view model using a convention-based approach. For more info see [Using the ViewModelLocator class to connect view models to views](#) and [Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime](#).

Deciding whether your app will construct views or the view models first is an issue of preference and complexity. With view first composition the app is conceptually composed of views which connect to the view models they depend upon. The primary benefit of this approach is that it makes it easy to construct loosely coupled, unit testable apps because the view models have no dependence on the views themselves. It's also easy to understand the structure of an app by following its visual structure, rather than having to track code execution in order to understand how classes are created and connected together. Finally, view first construction aligns better with the Windows Runtime navigation system because it is responsible for constructing the pages when navigation occurs, which makes a view model first composition complex and misaligned with the platform. View model first composition feels more natural to some developers, since the view creation can be abstracted away allowing them to focus on the logical non-UI structure of the app. However, this approach is often complex, and it can become difficult to understand how the various parts of the app are created and connected together. It can be difficult to understand the structure of an app constructed this way, as it often involves time spent in the debugger examining what classes gets created, when, and by whom.

The decision on how to connect view models to views is based on complexity, performance, and resilience:

- If code-behind is used to connect view models to views it can cause problems for visual designers such as Blend for Microsoft Visual Studio 2012 for Windows 8 and Visual Studio.
- Using a view model locator object has the advantage that the app has a single class that is responsible for the instantiation of view models. The view model locator can also be used as a point of substitution for alternate implementations of dependencies, such as for unit testing or design time data.
- A convention-based connection approach removes the need for much boilerplate code.
- An attached property can be used to perform the connection automatically. This offers the advantage of simplicity, with the view having no explicit knowledge of the view model.

Note The view will *implicitly* depend on specific properties, commands, and methods on the view model because of the data bindings it defines.

In Windows Store apps, you typically invoke some action in response to a user action, such as a button click that can be implemented by creating an event handler in the code-behind file. However, MVVM discourages placing code in the code-behind file as it's not easily testable because it doesn't maintain a good separation of concerns. If you wish to promote the testability of your app, by reducing the number of event handlers in your code-behind files, you should expose commands from your view models for [ButtonBase](#)-derived controls, and use behaviors in your views for controls that don't derive from **ButtonBase**, in order to connect them to view model exposed commands and actions.

If you will be using a visual designer to design and maintain your UI you'll need to include design time data support in your app so that you can view layouts accurately and see realistic results for sizing and styling decisions.

You should support a view model hierarchy if it will help to eliminate redundant code in your view model classes. If you find identical functionality in multiple view model classes, such as code to handle navigation, it should be refactored into a base view model class from which all view models classes will derive.

MVVM in AdventureWorks Shopper

The AdventureWorks Shopper reference implementation uses the Unity dependency injection container. The Unity container reduces the dependency coupling between objects by providing a facility to instantiate instances of classes and manage their lifetime. During an object's creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves them first. For more info see [Using a dependency injection container](#), [Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime](#) and [Unity Container](#).

In the AdventureWorks Shopper reference implementation, views are constructed before view models. There is one view class per page of the UI (a page is an instance of the [Windows.UI.Xaml.Controls.Page](#) class), with design time data being supported on each view in order to promote the designer-developer workflow. For more info see [Creating and navigating between pages](#).

Each view model is declaratively connected to a corresponding view using an attached property on a view model locator object to automatically perform the connection. View model dependencies are registered with the Unity dependency injection container, and resolved when the view model is created. A base view model class implements common functionality such as navigation and suspend/resume support for view model state. View model classes then derive from this base class in order to inherit the common functionality. For more info see [Using the ViewModelLocator class to connect view models to views](#).

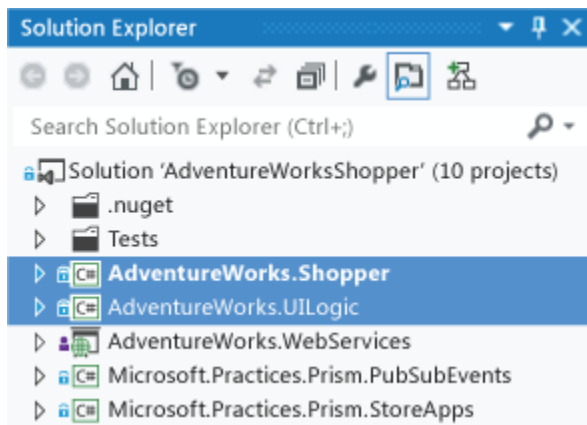
In order for a view model to participate in two-way data binding with the view, its properties must raise the [PropertyChanged](#) event. View models satisfy this requirement by implementing the [INotifyPropertyChanged](#) interface and raising the **PropertyChanged** event when a property is changed. Listeners can respond appropriately to the property changes when they occur. For more info see [Data binding with the BindableBase class](#).

The AdventureWorks Shopper reference implementation uses two options for executing code on a view model in response to interactions on a view, such as a button click or item selection. If the control is a command source, the control's [Command](#) property is data-bound to an [ICommand](#) property on the view model. When the control's command is invoked, the code in the view model will be executed. In addition to commands, behaviors can be attached to an object in the view and can listen for an event to be raised. In response, the behavior can then invoke an [Action](#) or an [ICommand](#) on the view model. For more info see [UI interaction using the DelegateCommand class and attached behaviors](#).

All of the view models in the AdventureWorks Shopper reference implementation share the app's domain model, which is often just called the model. The model consists of classes that the view models use to implement the app's functionality. View models are connected to the model classes through model properties on the view model. However, if you want a strong separation between the model and the view models, you can package model classes in a separate library.

In the AdventureWorks Shopper Visual Studio solution there are two projects that contain the view, view model, and model classes:

- The view classes are located in the AdventureWorks.Shopper project.
- The view model and model classes are located in the AdventureWorks.UILogic project.



What is MVVM?

MVVM is an architectural pattern that's a specialization of the presentation model pattern. It can be used on many different platforms and its intent is to provide a clean separation of concerns between the user interface controls and their logic. For more info about MVVM see [MVVM Quickstart](#), [Implementing the MVVM Pattern](#), [Advanced MVVM Scenarios](#), and [Developing a Windows Phone Application using the MVVM Pattern](#).

Using a dependency injection container

Dependency injection enables decoupling of concrete types from the code that depends on these types. It uses a container that holds a list of registrations and mappings between interfaces and abstract types and the concrete types that implement or extend these types. The AdventureWorks Shopper reference implementation uses the Unity dependency injection container to manage the instantiation of the view model and service classes in the app.

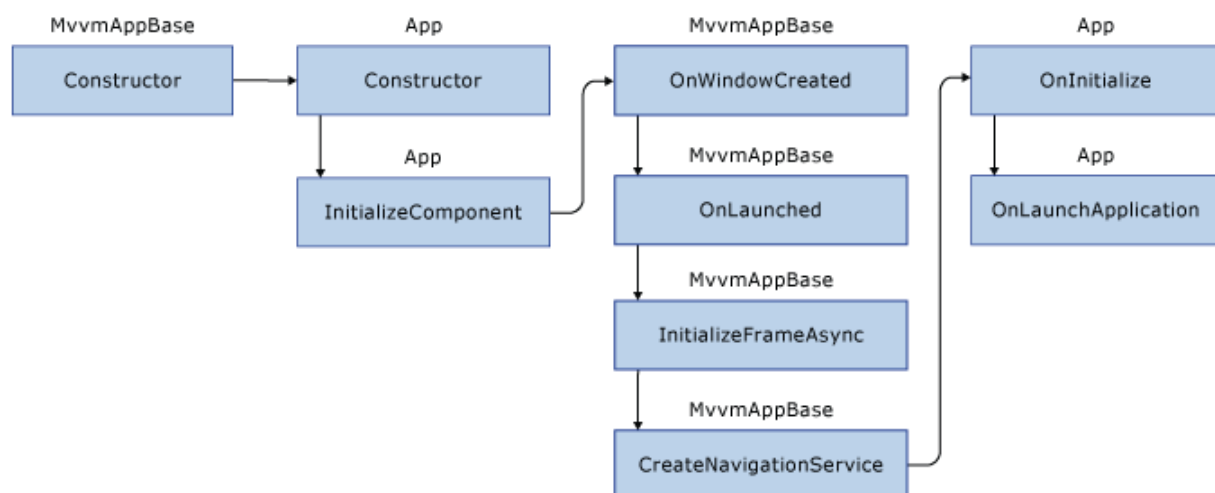
Before you can inject dependencies into an object, the types of the dependencies need to be registered with the container. After a type is registered, it can be resolved or injected as a dependency. For more info see [Unity](#).

In the AdventureWorks Shopper reference implementation, the **App** class instantiates the **UnityContainer** object and is the only class in the app that holds a reference to a **UnityContainer** object. Types are registered in the **OnInitialize** method in the **App** class.

Bootstrapping an MVVM app using the **MvvmAppBase** class

When you create a Windows Store app from a Visual Studio template, the **App** class derives from the **Application** class. In the AdventureWorks Shopper reference implementation, the **App** class derives from the **MvvmAppBase** class. The **MvvmAppBase** class provides support for suspension, navigation, Flyouts, settings, search, and resolving view types from view names. The **App** class derives from the **MvvmAppBase** class and provides app specific startup behavior.

The **MvvmAppBase** class, provided by the [Microsoft.Practices.Prism.StoreApps](https://github.com/Microsoft/Prism.StoreApps) library, is responsible for providing core startup behavior for an MVVM app, and derives from the **Application** class. The **MvvmAppBase** class constructor is the entry point for the app. The following diagram shows a conceptual view of how app startup occurs.



When deriving from the **MvvmAppBase** class, a required override is the **OnLaunchApplication** method from where you will typically perform your initial navigation to a launch page, or to the appropriate page based on a secondary tile launch of the app. The following code example shows the **OnLaunchApplication** method in the **App** class.

C#: AdventureWorks.Shopper\App.xaml.cs

```

protected override void OnLaunchApplication(LaunchActivatedEventArgs args)
{
    if (args != null && !string.IsNullOrEmpty(args.Arguments))
    {
        // The app was launched from a Secondary Tile
        // Navigate to the item's page
        NavigationService.Navigate("ItemDetail", args.Arguments);
    }
    else
    {

```

```

        // Navigate to the initial page
        NavigationService.Navigate("Hub", null);
    }
}

```

This method navigates to the **HubPage** in the app, when the app launches normally, or the **ItemDetailPage** if the app is launched from a secondary tile. "Hub" and "ItemDetail" are specified as the logical names of the views that will be navigated to. The default convention specified in the **MvvmAppBase** class is to append "Page" to the name and look for that page in a .Views child namespace in the project. Alternatively, another convention can be specified by overriding the **GetPageType** method in the **MvvmAppBase** class. For more info see [Handling navigation requests](#).

The app uses the Unity dependency injection container to reduce the dependency coupling between objects by providing a facility to instantiate instances of classes and manage their lifetime based on the configuration of the container. An instance of the container is created as a singleton in the **App** class, as shown in the following code example.

C#: AdventureWorks.Shopper\App.xaml.cs

```
private readonly IUnityContainer _container = new UnityContainer();
```

The **OnInitialize** method in the **MvvmAppBase** class is overridden in the **App** class with app specific initialization behavior. For instance, this method should be overridden if you need to initialize services, or set a default factory or default view model resolver for the **ViewModelLocator** object. The following code example shows some of the **OnInitialize** method in the **App** class.

C#: AdventureWorks.Shopper\App.xaml.cs

```

_container.RegisterInstance<INavigationService>(NavigationService);
_container.RegisterInstance<ISessionStateService>(SessionStateService);
_container.RegisterInstance<IFlyoutService>(FlyoutService);
_container.RegisterInstance<IEventAggregator>(_eventAggregator);
_container.RegisterInstance<IResourceLoader>(new ResourceLoaderAdapter(
    new ResourceLoader()));

```

This code registers service instances with the container as singletons, based on their respective interfaces, so that the view model classes can take dependencies on them. This means that the container will cache the instances on behalf of the app, with the lifetime of the instances then being tied to the lifetime of the container.

A view model locator object is responsible for managing the instantiation of view models and their association to views. For more info see [Using the ViewModelLocator class to connect view models to views](#). When the view model classes are instantiated the container will inject the dependencies that are required. If the dependencies have not yet been created, the container creates and resolves them first. This approach removes the need for an object to locate its dependencies or manage their

lifetimes, allows swapping of implemented dependencies without affecting the object, and facilitating testability by allowing dependencies to be mocked.

Using the **ViewModelLocator** class to connect view models to views

The AdventureWorks Shopper reference implementation uses a view model locator object to manage the instantiation of view models and their association to views. This has the advantage that the app has a single class that is responsible for the instantiation.

The **ViewModelLocator** class, in the [Microsoft.Practices.Prism.StoreApps](https://github.com/Microsoft/Prism.StoreApps) library, has an attached property, **AutoWireViewModel** that is used to associate view models with views. In the view's XAML this attached property is set to **true** to indicate that the view model should be automatically connected to the view, as shown in the following code example.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
Infrastructure:ViewModelLocator.AutoWireViewModel="true"
```

The **AutoWireViewModel** property is a dependency property that is initialized to **false**, and when its value changes the **AutoWireViewModelChanged** event handler is called. This method resolves the view model for the view. The following code example shows how this is achieved.

C#: Microsoft.Practices.Prism.StoreApps\ViewModelLocator.cs

```
private static void AutoWireViewModelChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs e)
{
    FrameworkElement view = d as FrameworkElement;
    if (view == null) return; // Incorrect hookup, do no harm

    // Try mappings first
    object viewModel = GetViewModelForView(view);
    // Fallback to convention based
    if (viewModel == null)
    {
        var viewModelType = defaultViewTypeToViewModelTypeResolver(
            view.GetType());
        if (viewModelType == null) return;

        // Really need Container or Factories here to deal with injecting
        // dependencies on construction
        viewModel = defaultViewModelFactory(viewModelType);
    }
    view.DataContext = viewModel;
}
```

The **AutoWireViewModelChanged** method first attempts to resolve the view model from any mappings that may have been registered by the **Register** method of the **ViewModelLocator** class. If the view model cannot be resolved using this approach, for instance if the mapping wasn't created,

the method falls back to using a convention-based approach to resolve the correct view model type. This convention assumes that view models are in a `.ViewModels` child namespace, and that view model names correspond with view names and end with `"ViewModel"`. For more info see [Using a convention-based approach](#). Finally, the method sets the `DataContext` property of the view type to the registered view model type.

Using a convention-based approach

A convention-based approach to connecting view models to views removes the need for much boilerplate code. The AdventureWorks Shopper reference implementation redefines the convention for resolving view model types from view types. The convention assumes that:

1. View model types are located in a separate assembly from the view types.
2. View model types are located in the AdventureWorks.UILogic assembly.
3. View model type names append `"ViewModel"` to the view type names.

Using this convention, a view named **HubPage** will have a view model named **HubPageViewModel**. The following code example shows how the **App** class overrides the **SetDefaultViewTypeToViewModelTypeResolver** delegate in the **ViewModelLocator** class, to define how to resolve view model type names from view type names.

C#: AdventureWorks.Shopper\App.xaml.cs

```
ViewModelLocator.SetDefaultViewTypeToViewModelTypeResolver((viewType) =>
{
    var viewModelTypeName = string.Format(CultureInfo.InvariantCulture,
        "AdventureWorks.UILogic.ViewModels.{0}ViewModel",
        AdventureWorks.UILogic, Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=634ac3171ee5190a", viewType.Name);
    var viewModelType = Type.GetType(viewModelTypeName);
    return viewModelType;
});
```

Other approaches to connect view models to views

There are many approaches that can be used to connect view and view model classes at run time. The following sections describe three of these approaches.

Creating a view model declaratively

The simplest approach is for the view to declaratively instantiate its corresponding view model in XAML. When the view is constructed, the corresponding view model object will also be constructed. This approach can be demonstrated in the following code.

XAML

```
<Page.DataContext>
    <HubPageViewModel />
</Page.DataContext>
```

When the [Page](#) is created, an instance of the **HubPageViewModel** is automatically constructed and set as the view's data context. This approach requires your view model to have a default (parameter-less) constructor.

This declarative construction and assignment of the view model by the view has the advantage that it is simple and works well in design-time tools such as Blend and Visual Studio.

Creating a view model programmatically

A view can have code in the code-behind file that results in the view model being assigned to its [DataContext](#) property. This is often accomplished in the view's constructor, as shown in the following code example.

```
C#  
  
public HubPage()  
{  
    InitializeComponent();  
    this.DataContext = new HubPageViewModel();  
}
```

Connecting a view model to a view in a code-behind file is discouraged as it can cause problems for designers in both Blend and Visual Studio.

Creating a view defined as a data template

A view can be defined as a data template and associated with a view model type. Data templates can be defined as resources, or they can be defined inline within the control that will display the view model. The content of the control is the view model instance, and the data template is used to visually represent it. This technique is an example of a situation in which the view model is instantiated first, followed by the creation of the view.

Data templates are flexible and lightweight. The UI designer can use them to easily define the visual representation of a view model without requiring any complex code. Data templates are restricted to views that do not require any UI logic (code-behind). Blend can be used to visually design and edit data templates.

The following example shows a [GridView](#) that is bound to a collection of **ShoppingCartItemViewModels**. Each object in the **ShoppingCartItemViewModels** collection is a view model instance. The view for each **ShoppingCartItemViewModel** is defined by the [ItemTemplate](#) property. The **ShoppingCartItemTemplate** specifies that the view for each **ShoppingCartItemViewModel** consists of a [Grid](#) containing multiple child elements, including an [Image](#) and several [TextBlocks](#).

XAML: AdventureWorks.Shopper\Views\ShoppingCartPage.xaml

```
<GridView x:Name="ShoppingCartItemsGridView"
    x:Uid="ShoppingCartItemsGridView"
    AutomationProperties.AutomationId="ShoppingCartItemsGridView"
    SelectionMode="Single"
    Width="Auto"
    Grid.Row="2"
    Grid.Column="1"
    Grid.RowSpan="3"
    VerticalAlignment="Top"
    ItemsSource="{Binding ShoppingCartItemViewModels}"
    SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
    ItemTemplate="{StaticResource ShoppingCartItemTemplate}"
    Margin="0,0,0,0" />
```

As well as defining a data template as a resource, they can also be defined inline, or you could place the detailed code from the template into a user control and declare an instance of the user control inside the template.

Data binding with the BindableBase class

The Windows Runtime provides powerful data binding capabilities. Your view model and model classes should be designed to support data binding so that they can take advantage of these capabilities. For more info about data binding in the Windows Runtime, see [Data binding overview](#).

All view model and model classes that are accessible to the view should implement the [INotifyPropertyChanged](#) interface. Implementing the **INotifyPropertyChanged** interface in your view model or model classes allows them to provide change notifications to any data-bound controls in the view when the underlying property value changes. However, this can be repetitive and error-prone. Therefore, the [Microsoft.Practices.Prism.StoreApps](#) library provides the **BindableBase** class that implements the **INotifyPropertyChanged** interface. The following code example shows this class.

C#: Microsoft.Practices.Prism.StoreApps\BindableBase.cs

```
public abstract class BindableBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual bool SetProperty<T>(ref T storage, T value,
        [CallerMemberName] string propertyName = null)
    {
        if (object.Equals(storage, value)) return false;

        storage = value;
        this.OnPropertyChanged(propertyName);

        return true;
    }
}
```



```
protected void OnPropertyChanged(string propertyName)
{
    var eventHandler = this.PropertyChanged;
    if (eventHandler != null)
    {
        eventHandler(this, new PropertyChangedEventArgs(propertyName));
    }
}
}
```

Note The **BindableBase** class, provided by the [Microsoft.Practices.Prism.StoreApps](https://docs.microsoft.com/en-us/windows/uwp/xaml-platform/microsoft-practices-prism-storeapps) library, is identical to the **BindableBase** class provided by the Visual Studio project templates.

Each view model class in the AdventureWorks Shopper reference implementation derives from the **ViewModel** base class that in turn derives from the **BindableBase** base class. Therefore, each view model class uses the **SetProperty** method in the **BindableBase** class to provide property change notification. The following code example shows how property change notification is implemented in a view model class in the AdventureWorks Shopper reference implementation.

C#: AdventureWorks.UILogic\ViewModels\HubPageViewModel.cs

```
public IReadOnlyCollection<CategoryViewModel> RootCategories
{
    get { return _rootCategories; }
    private set { SetProperty(ref _rootCategories, value); }
}
```

Additional considerations

You should design your app for the correct use of property change notification. Here are some points to remember:

- Never raise the [PropertyChanged](#) event during your object's constructor if you are initializing a property. Data-bound controls in the view cannot have subscribed to receive change notifications at this point.
- Always implement the [INotifyPropertyChanged](#) interface on any view model or model classes that are accessible to the view.
- Always raise a **PropertyChanged** event if a public property's value changes. Do not assume that you can ignore raising the **PropertyChanged** event because of knowledge of how XAML binding occurs. Such assumptions lead to brittle code.
- Never use a public property's get method to modify fields or raise the **PropertyChanged** event.
- Always raise the **PropertyChanged** event for any calculated properties whose values are used by other properties in the view model or model.
- Never raise a **PropertyChanged** event if the property does not change. This means that you must compare the old and new values before raising the **PropertyChanged** event.

- Never raise more than one **PropertyChanged** event with the same property name argument within a single synchronous invocation of a public method of your class. For example, suppose you have a **Count** property whose backing store is the **_count** field. If a method increments **_count** a hundred times during the execution of a loop, it should only raise property change notification on the **Count** property once after all the work is complete. For asynchronous methods you can raise the **PropertyChanged** event for a given property name in each synchronous segment of an asynchronous continuation chain.
- Always raise the **PropertyChanged** event at the end of the method that makes a property change, or when your object is known to be in a safe state. Raising the event interrupts your operation by invoking the event's handlers synchronously. If this happens in the middle of your operation, you may expose your object to callback functions when it is in an unsafe, partially updated state. It is also possible for cascading changes to be triggered by **PropertyChanged** events. Cascading changes generally require updates to be complete before the cascading change is safe to execute.

UI interaction using the **DelegateCommand** class and attached behaviors

In Windows Store apps, you typically invoke some action in response to a user action (such as a button click) that can be implemented by creating an event handler in the code-behind file. However, in the MVVM pattern, the responsibility for implementing the action lies with the view model, and you should try to avoid placing code in the code-behind file.

Implementing command objects

Commands provide a convenient way to represent actions that can be easily bound to controls in the UI. They encapsulate the actual code that implements the action or operation and help to keep it decoupled from its actual visual representation in the view. The [Microsoft.Practices.Prism.StoreApps](#) library provides the **DelegateCommand** class to implement commands.

View models typically expose command properties, for binding from the view, that are object instances that implement the **ICommand** interface. XAML inherently supports commands and **ButtonBase**-derived controls provide a **Command** property that can be data bound to an **ICommand** object provided by the view model. The **ICommand** interface defines an **Execute** method, which encapsulates the operation itself, and a **CanExecute** method, which indicates whether the command can be invoked at a particular time. Alternatively, a command behavior can be used to associate a control with a command method provided by the view model.

Note Behaviors are a powerful and flexible extensibility mechanism that can be used to encapsulate interaction logic and behavior that can be declaratively associated with controls in the view. Command behaviors can be used to associate methods with controls that were not specifically designed to interact with commands. For more info see [Implementing behaviors to supplement the functionality of XAML elements](#).

The AdventureWorks Shopper reference implementation uses the **DelegateCommand** class that encapsulates two delegates that each reference a method implemented within your view model

class. It inherits from the **DelegateCommandBase** class that implements the **ICommand** interface's **Execute** and **CanExecute** methods by invoking these delegates. You specify the delegates to your view model methods in the **DelegateCommand** class constructor, which is defined as follows.

C#: Microsoft.Practices.Prism.StoreApps\DelegateCommand.cs

```
public DelegateCommand(Action<T> executeMethod, Func<T, bool> canExecuteMethod)
    : base((o) => executeMethod((T)o), (o) => canExecuteMethod((T)o))
{
    if (executeMethod == null || canExecuteMethod == null)
        throw new ArgumentNullException("executeMethod");
}
```

For example, the following code shows how a **DelegateCommand** instance, which represents a sign in command, is constructed by specifying delegates to the **SignInAsync** and **CanSignIn** view model methods. The command is then exposed to the view through a read-only property that returns a reference to an [ICommand](#).

C#: AdventureWorks.UILogic\ViewModels\SignInFlyoutViewModel.cs

```
public DelegateCommand SignInCommand { get; private set; }
...
SignInCommand = DelegateCommand.FromAsyncHandler(SignInAsync, CanSignIn);
```

The **DelegateCommand** class is a generic type. The type argument specifies the type of the command parameter passed to the [Execute](#) and [CanExecute](#) methods. A non-generic version of the **DelegateCommand** class is also provided for use when a command parameter is not required.

When the **Execute** method is called on the **DelegateCommand** object, it simply forwards the call to the method in the view model class via the delegate that you specified in the constructor. Similarly, when the **CanExecute** method is called, the corresponding method in the view model class is called. The delegate to the **CanExecute** method in the constructor is optional. If a delegate is not specified, the **DelegateCommand** will always return true for **CanExecute**.

The view model can indicate a change in the command's **CanExecute** status by calling the **RaiseCanExecuteChanged** method on the **DelegateCommand** object. This causes the [CanExecuteChanged](#) event to be raised. Any controls in the UI that are bound to the command will update their enabled status to reflect the availability of the bound command.

Invoking commands from a view

Any controls that derive from [ButtonBase](#), such as [Button](#) or [HyperlinkButton](#), can be easily data bound to a command through the [Command](#) property. The following code example shows how the **SubmitButton** in the **SignInFlyout** binds to the **SignInCommand** in the **SignInFlyoutViewModel** class.

XAML: AdventureWorks.Shopper\Views\SignInFlyout.xaml

```

<Button x:Uid="SubmitButton"
        x:Name="SubmitButton"
        Background="{StaticResource AWShopperAccentBrush}"
        Content="Submit"
        Width="280"
        Foreground="{StaticResource AWShopperButtonForegroundBrush}"
        Margin="0,25,0,0"
        Command="{Binding SignInCommand}"
        AutomationProperties.AutomationId="SignInSubmitButton"
        Style="{StaticResource LightButtonStyle}" />

```

A command parameter can also be optionally defined using the [CommandParameter](#) property. The type of the expected argument is specified in the [Execute](#) and [CanExecute](#) target methods. The control will automatically invoke the target command when the user interacts with that control, and the command parameter, if provided, will be passed as the argument to the command's **Execute** method.

Implementing behaviors to supplement the functionality of XAML elements

A behavior allows you to add functionality to a XAML element by writing that functionality in a behavior class and attaching it to the element as if it was part of the element itself. A behavior can be attached to a XAML element through attached properties. The behavior can then use the exposed API of the element to which it is attached to add functionality to that element or other elements in the visual tree of the view. For more info see [Dependency properties overview](#), [Attached properties overview](#), and [Custom attached properties](#).

Behaviors enable you to implement code that you would normally have to write as code-behind because it directly interacts with the API of XAML elements, in such a way that it can be concisely attached to a XAML element and packaged for reuse across more than one view or app. In the context of MVVM, behaviors are a good approach to connecting items that are occurring in the view due to user interaction, with the execution in a view model.

An *attached behavior* is a behavior that is defined as a static class with one or more attached properties contained within it. An attached property can define a change callback handler when the dependency property is set on a target element. The callback handler gets passed a reference to the element on which it is being attached and an argument that defines what the old and new values for the property are. The change callback handler is then used to connect new functionality to the XAML element the property is attached to by manipulating the reference to it that gets passed in. The typical pattern is that the change callback handler will cast the element reference to a known element type that the behavior is designed to enhance. Then it will connect to events exposed by that element type, modify properties of the element, or call methods on the element to manifest the desired behavior. For example, the AdventureWorks Shopper reference implementation provides the **ListViewItemClickedToAction** behavior that casts the element reference to the [ListViewBase](#) type, which is the base class for the [GridView](#) and [ListView](#) controls, then subscribes to the [ItemClick](#) event and in the handler for the event invokes an [Action](#).

C#: AdventureWorks.Shopper\Behaviors\ListViewItemClickedToAction.cs

```
public static class ListViewItemClickedToAction
{
    public static DependencyProperty ActionProperty =
        DependencyProperty.RegisterAttached("Action", typeof(Action<object>),
            typeof(ListViewItemClickedToAction),
            new PropertyMetadata(null, OnActionChanged));

    ...

    private static void OnActionChanged(DependencyObject d,
        DependencyPropertyChangedEventArgs args)
    {
        ListViewBase listView = (ListViewBase)d;

        if (listView != null)
        {
            listView.ItemClick += listView_ItemClick;
            listView.Unloaded += listView_Unloaded;
        }
    }

    static void listView_Unloaded(object sender, RoutedEventArgs e)
    {
        ListViewBase listView = (ListViewBase)sender;
        listView.ItemClick -= listView_ItemClick;
        listView.Unloaded -= listView_Unloaded;
    }

    static void listView_ItemClick(object sender, ItemClickEventArgs e)
    {
        var listView = (ListViewBase)sender;
        Action<object> action = (Action<object>)listView.GetValue(ActionProperty);
        action(e.ClickedItem);
    }
}
```

Unlike controls that can be bound directly to a command, the **ListViewItemClickedToAction** behavior does not automatically enable or disable the control based on a value returned by a [CanExecute](#) delegate. To implement this behavior, you have to data bind the **IsEnabled** property of the control directly to a suitable property on the view model.

In addition, when writing attached behaviors it is important that you unsubscribe from subscribed events at the appropriate time, so that you do not cause memory leaks.

Invoking behaviors from a view

Behaviors are particularly useful if you want to attach a command method to a control that does not derive from [ButtonBase](#). For example, the AdventureWorks Shopper reference implementation uses the **ListViewItemClickedToAction** attached behavior to enable the [ItemClick](#) event of the

MultipleSizedGridView control to be handled in a view model, rather than in the page's code-behind.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<controls:MultipleSizedGridView x:Name="itemsGridView"
    AutomationProperties.AutomationId="HubPageItemGridView"
    AutomationProperties.Name="Grouped Items"
    Margin="0,-3,0,0"
    Padding="116,0,40,46"
    ItemsSource="{Binding Source={StaticResource groupedItemsViewSource}}"
    ItemTemplate="{StaticResource AWShopperItemTemplate}"
    SelectionMode="None"
    ScrollViewer.IsHorizontalScrollChainingEnabled="False"
    IsItemClickEnabled="True"
    behaviors:ListViewItemClickedToAction.Action=
        "{Binding ProductNavigationAction}">
```

This behavior binds the [ItemClick](#) event of the **MultipleSizedGridView** to the **ProductNavigationAction** property in the **HubPageViewModel** class. So when a [GridViewItem](#) is selected the **ProductNavigationAction** is executed which navigates from the **HubPage** to the **ItemDetailPage**.

Additional considerations

Here are some additional considerations when applying the MVVM pattern to Windows Store apps in C#.

Centralize data conversions in the view model or a conversion layer

The view model provides data from the model in a form that the view can easily use. To do this the view model sometimes has to perform data conversion. Placing this data conversion in the view model is a good idea because it provides properties in a form that the UI can bind to. It is also possible to have a separate data conversion layer that sits between the view model and the view. This might occur, for example, when data types need special formatting that the view model doesn't provide.

Expose operational modes in the view model

The view model may also be responsible for defining logical state changes that affect some aspect of the display in the view, such as an indication that some operation is pending or whether a particular command is available. You don't need code-behind to enable and disable UI elements—you can achieve this by binding to a view model property, or with visual states.

Keep views and view models independent

The binding of views to a particular property in its data source should be a view's principal dependency on its corresponding view model. In particular, do not reference view types or the

[**Windows.Current**](#) object from view models. If you follow the principles we outlined here, you will have the ability to test view models in isolation, and reduce the likelihood of software defects by limiting scope.

Use asynchronous programming techniques to keep the UI responsive

Windows Store apps are about a fast and fluid user experience. For that reason the AdventureWorks Shopper reference implementation keeps the UI thread unblocked. AdventureWorks Shopper uses asynchronous library methods for I/O operations and raises events to asynchronously notify the view of a property change.

Creating and navigating between pages in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Create pages using the MVVM pattern if appropriate to your requirements. When using MVVM, use XAML data binding to link each page to a view model object.
- Design your pages for landscape, portrait, snap, and fill layout. In addition, use the **VisualStateAwarePage** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, to provide view management.
- Implement the **INavigationAware** interface, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, to enable a class to participate in a navigation operation. Use the **FrameNavigationService** class, provided by the Microsoft.Practices.Prism.StoreApps library, to provide navigation support to a class.

In the AdventureWorks Shopper reference implementation there is one page for each screen that a user can navigate to. The app creates the first page on startup and then creates subsequent pages in response to navigation requests. View management and navigation support is provided to pages by [Prism for the Windows Runtime](#). AdventureWorks Shopper's pages support landscape and portrait orientations as well as snap and fill layouts. In addition, pages are localizable and accessible.

You will learn

- How pages were designed in AdventureWorks Shopper.
- How AdventureWorks Shopper creates pages and their data sources at run time.
- How to create design time data to support designers.
- How AdventureWorks Shopper pages support app view states such as the snapped view.
- How AdventureWorks Shopper pages support localization and accessibility.
- How AdventureWorks Shopper performs navigation between pages.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

The app page is the focal point for designing your UI. It holds all of your content and controls for a single point of interaction with the user within your app. Whenever possible, you should integrate your UI elements inline into the app page. Presenting your UI inline lets users fully immerse themselves in your app and stay in context, as opposed to using pop-ups, dialogs, or overlapping windows that were common in previous Windows desktop application platforms. You can create as many app pages as you need to support your user scenarios.

The following list summarizes the decisions to make when creating pages in your app:

- What tool should I use to create page content?
- What minimum resolution should I design my pages for?
- Should my page content fill the screen, regardless of resolution?
- Should my pages adapt to different orientations and layouts?
- How should I lay out UI elements on each page?
- What should I display in snap and fill view?
- How should I test my page layout on different screen sizes?
- Should I add design time data to my pages?
- Should I make my pages easily localizable?
- Should I make my pages accessible?
- Should I cache pages in my app?
- Where should navigation logic reside?
- How should I invoke navigation from a view?
- What commands belong on the top app bar and the bottom app bar?
- Should common page navigation functionality be implemented on each page, or can it be encapsulated into a single control for reuse on each page?
- Should the page being navigated to reside in the same assembly that the navigation request originates from?
- How should I specify a navigation target?

We recommend that you use Visual Studio to work with the code-focused aspects of your app. Visual Studio is best suited for writing code, running, and debugging your app. We recommend that you use Blend for Microsoft Visual Studio 2012 for Windows 8 to work on the visual appearance of your app. You can use Blend to create pages and custom controls, change templates and styles, and create animations. Blend comes with minimal code-behind support. For more info about XAML editing tools, see [Blend for Visual Studio 2012](#) and [Creating a UI by using the XAML Designer](#).

There are two primary screen resolutions that your app should support. The minimum resolution at which Windows Store apps will run is 1024x768. However, the minimum resolution required to support all of the features of Windows 8, including multitasking with snap, is 1366x768. When designing pages for a minimum resolution of 1024x768 you should ensure that all of your UI fits on the screen without clipping. When designing pages for an optimal resolution of 1366x768 you should ensure that all of your UI fits on the screen without blank regions. Page content should fill the screen to the best of its ability and should appear to be thoughtfully designed for varying screen sizes. Users who buy larger monitors expect that their apps will continue to look good on these large screens and fill the screen with more content, where possible. For more info see [Guidelines for scaling to screens](#).

Users can rotate and flip their tablets, slates, and monitors, so you should ensure that your app can handle both *landscape* and *portrait* orientations. In addition, because users can work with up to two apps at once, you should provide layouts that are fluid and flexible enough to support *fill*, and *snap* layouts. A snapped app occupies a narrow region of the screen, while an app in the fill view fills the screen area not occupied by the snapped app. Snapped and fill views are only available on displays

with a horizontal resolution of 1366 pixels or greater. This is because the snapped view is 320 pixels wide, and can be placed on either side of the screen. The remaining 1046 pixels are allocated to the splitter and the fill view, which must always have a horizontal resolution of 1024 pixels or greater. For more info see [Guidelines for layouts](#) and [Guidelines for snapped and fill views](#).

The user interface in Windows 8 strives to maintain a consistent silhouette across its apps. The signature characteristic of the silhouette is a wide margin on the top, bottom, and left edges. This wide margin helps users understand the horizontal panning direction of the content. You should follow a consistent layout pattern for margins, page headers, gutter widths, and other such elements on your pages. For more info see [Laying out an app page](#).

When you plan for full screen, snap, and fill views, your app's UI should reflow smoothly and gracefully to accommodate screen size, orientation, and user interactions. You should maintain state in snap view, even if it means showing less content or reducing functionality. In addition, you should have feature parity across states. The user still expects to be able to interact with your app when it is snapped. If you can't keep parity for a specific feature, we recommend that you include an entry point to the feature and programmatically unsnap the app when the user triggers that entry point. However, you should never add UI controls to programmatically unsnap your app. The splitter between the apps is always present and lets the user unsnap whenever they want. For more info see [Guidelines for snapped and fill views](#).

Most people don't have many devices at their disposal for testing page layout on different screen sizes. However, you can use the Windows Simulator to run your app on a variety of screen sizes, orientations, and pixel densities. In addition, Blend offers a platform menu that enables you to design your app on different screen sizes and pixel densities on the fly. The Blend canvas then updates dynamically based upon the chosen screen option.

Sample data should be added to each page if you want to easily view styling results and layout sizes at design time. This has the additional advantage of supporting the designer-developer workflow.

Preparing your pages for localization can help your app reach more users in international markets. It's important to consider localization early on in the development process, as there are some issues that will affect UI elements across various locales. As you design your pages, keep in mind that users have a wide range of abilities, disabilities, and preferences. If you incorporate accessible design principles into your pages you will help to ensure that your app is accessible to the widest possible audience, thus attracting more customers to your app. For more info see [Globalizing your app](#) and [Design for accessibility](#).

Deciding whether to cache pages will be dependent upon how well-performing and responsive the app is. Page caching results in memory consumption for views that are not currently displayed, which would increase the chance of termination when the app is suspended. However, without page caching it does mean that XAML parsing and construction of the page and its view model will occur every time you navigate to a new page, which could have a performance impact for a complicated page. For a well-designed page that does not use too many controls, the performance should be

sufficient. However, if you encounter slow page load times you should test to see if enabling page caching alleviates the problem. For more info see [Quickstart: Navigating between pages](#).

Navigation within a Windows Store app can result from the user's interaction with the UI or from the app itself as a result of internal logic-driven state changes. Page navigation requests are usually triggered from a view, with the navigation logic either being in the view's code-behind, or in the data bound view model. While placing navigation logic in the view may be the simplest approach, it is not easily testable through automated tests. Placing navigation logic in the view model classes means that the navigation logic can be exercised through automated tests. In addition, the view model can then implement logic to control navigation to ensure that certain business rules are enforced. For instance, an app may not allow the user to navigate away from a page without first ensuring that the entered data is correct.

Users will trigger navigation from a view by selecting a UI control, with the navigation logic residing in the appropriate view model class. For controls derived from [ButtonBase](#), such as [Button](#), you should use commands to implement a navigation action in the view model class. For controls that do not derive from **ButtonBase**, you should use an attached behavior to implement a navigation action in the view model class. For more info see [Using the Model-View-ViewModel \(MVVM\) pattern](#).

In general, you should use the top app bar for navigational elements that move the user to a different page and use the bottom app bar for commands that act on the current page. If every page of your app is going to include a top app bar that allows the user to move to different pages, it does not make sense to implement this functionality individually on each page. Rather, the functionality should be implemented as a user control that can be easily be included on each page. In addition, you should follow placement conventions for commands on the bottom app bar. You should place **New/Add/Create** buttons on the far right, with view switching buttons being placed on the far left. Also, you should place **Accept, Yes, and OK** buttons to the left of **Reject, No, and Cancel** buttons. For more info see [Guidelines for app bars](#).

The view classes that define your pages and the view model classes that implement the business logic for those pages can reside in the same assembly or different assemblies. That is a design decision to be made when architecting your app. A page type resolution strategy should be used to navigate to a page in any assembly, regardless of the assembly from which the navigation request originates.

One approach for specifying a navigation target is to use a navigation service, which would require the type of the view to navigate to. Because a navigation service is usually invoked from view models in order to promote testability, this approach would require view models to reference views (and particularly views that the view model isn't associated with), which is not recommended. The recommended approach is to use a string to specify the navigation target that can be easily passed to a navigation service, and which is easily testable.

Creating pages and navigating between them in AdventureWorks Shopper

We used Blend and the Visual Studio XAML Designer to work with XAML because these tools make it straightforward to quickly add and modify page layout. Blend was useful to initially define pages and controls; we used Visual Studio to optimize their appearances. These tools also enabled us to iterate quickly through design choices because they give immediate visual feedback. In many cases, our user experience designer was able to work in parallel with the developers because changing the visual appearance of a page does not affect its behavior. For more info see [Creating pages](#).

Pages were designed for a minimum resolution of 1024x768, and an optimal minimum resolution of 1366x768, in order to support all of the features of Windows 8, particularly multitasking with snap. In addition, pages were designed to fill the screen for varying screen sizes. Each page is able to adapt to *landscape* and *portrait* orientations, and *fill* and *snap* layouts. A consistent silhouette is maintained across all pages, with some pages including design time data. Page layout was tested on a variety of devices, and in the Windows simulator. Pages maintain state when switching to and from snap view, and possess feature parity across states. For more info see [Adding design time data](#), [Supporting portrait, snap, and fill layouts](#) and [Laying out an app page](#).

Page caching is not used in the app. This prevents views that are not currently displayed from consuming memory, which would increase the chance of termination when the app is suspended. All pages are accessible, and support easy localization. For more info see [Enabling page localization](#) and [Enabling page accessibility](#).

In the app, the view classes that define pages are in a different assembly to the view model classes that implement the business logic for those pages. Therefore, a page type resolution strategy implemented as a delegate is used to navigate to the pages in the AdventureWorks.Shopper assembly when the navigation request originates from view model classes in the AdventureWorks.UI.Logic assembly. In addition, common page navigation functionality is implemented as a user control that is embedded in the top app bar for each page. Both commands and attached behaviors are used to implement navigation actions in view model classes, depending on the control type. Navigation targets are specified by strings that represent the page to navigate to. For more info see [Navigating between pages](#), [Handling navigation requests](#), and [Invoking navigation](#).

Creating pages

Pages in Windows Store apps are user controls that support navigation and contain other controls. All page classes are subtypes of the [Windows.UI.Xaml.Page](#) class, and represent content that can be navigated to by the user.

When you add a new page to a project created from the Grid App (XAML) template, each page is derived from the Visual Studio's **LayoutAwarePage** class (except when you add a Blank Page, which derives from the [Page](#) class) that provides navigation, state management, and view management. However, MVVM apps such as the AdventureWorks Shopper reference implementation should derive each page from the **VisualStateAwarePage** class in the [Microsoft.Practices.Prism.StoreApps](#)

library. The **VisualStateAwarePage** class provides view management and navigation support. The following code example shows how the **HubPage** derives from the **VisualStateAwarePage** class.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<Infrastructure:VisualStateAwarePage x:Name="pageRoot"
    x:Class="AdventureWorks.Shopper.Views.HubPage"
    IsTabStop="false"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:behaviors="using:AdventureWorks.Shopper.Behaviors"
    xmlns:views="using:AdventureWorks.Shopper.Views"
    xmlns:controls="using:AdventureWorks.Shopper.Controls"
    xmlns:designViewModels="using:AdventureWorks.Shopper.DesignViewModels"
    xmlns:Infrastructure="using:Microsoft.Practices.Prism.StoreApps"
    x:Uid="Page"
    mc:Ignorable="d"
    Infrastructure:ViewModelLocator.AutoWireViewModel="true"
    d:DataContext="{d:DesignInstance designViewModels:HubPageDesignViewModel,
        IsDesignTimeCreatable=True}">
```

Note The [Microsoft.Practices.Prism.StoreApps](#) library also provides the **FlyoutView** class, from which all Flyout classes should derive in an MVVM app.

There are twelve pages in the AdventureWorks Shopper reference implementation, with the pages being the views of the MVVM pattern.

Page	View model
BillingAddressPage	BillingAddressPageViewModel
CategoryPage	CategoryPageViewModel
CheckoutHubPage	CheckoutHubPageViewModel
CheckoutSummaryPage	CheckoutSummaryPageViewModel
GroupDetailPage	GroupDetailPageViewModel
HubPage	HubPageViewModel
ItemDetailPage	ItemDetailPageViewModel
OrderConfirmationPage	OrderConfirmationPageViewModel
PaymentMethodPage	PaymentMethodPageViewModel
SearchResultsPage	SearchResultsPageViewModel
ShippingAddressPage	ShippingAddressPageViewModel
ShoppingCartPage	ShoppingCartPageViewModel

Data binding links each page to its view model class in the AdventureWorks Shopper reference implementation. The view model class gives the page access to the underlying app logic by using the conventions of the MVVM pattern. For more info see [Using the MVVM pattern](#).

Tip AdventureWorks Shopper uses the MVVM pattern that abstracts the user interface for the app. With MVVM you rarely need to customize the code-behind files. Instead, the controls of the user interface are bound to properties of a view model object. If page-related code is required, it should be limited to conveying data to and from the page's view model object.

If you are interested in AdventureWorks Shopper's interaction model and how we designed the user experience, see [Designing the AdventureWorks Shopper user experience](#).

Adding design time data

When you create a data bound user interface, you can display sample data in the visual designer to view styling results and layout sizes. To display data in the designer you must declare it in XAML. This is necessary because the designer parses the XAML for a page but does not run its code-behind. In the AdventureWorks Shopper reference implementation, we wanted to display design time data in order to support the designer-developer workflow.

Sample data can be displayed at design time by declaring it in XAML by using the various data attributes from the designer namespace. This namespace is typically declared with a **d:** prefix, as shown in the following code example.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

Attributes with **d:** prefixes are then interpreted only at design time and are ignored at run time. For example, in a [CollectionViewSource](#) the **d:DesignSource** attribute is used for design time sample data, and the [Source](#) attribute is used for run time data.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<CollectionViewSource x:Name="groupedItemsViewSource"
    Source="{Binding Path=RootCategories}"
    d:DesignSource="{Binding RootCategories,
        Source={d:DesignInstance
            designViewModels:HubPageDesignViewModel,
            IsDesignTimeCreatable=True}}"
    IsSourceGrouped="true"
    ItemsPath="Products" />
```

The **d:DesignInstance** attribute indicates that the design time source is a designer created instance based on the **HubPageDesignViewModel** type. The **IsDesignTimeCreatable** setting indicates that the designer will instantiate that type directly, which is necessary to display the sample data generated by the type constructor.

For more info see [Data binding overview](#).

Supporting portrait, snap, and fill layouts

The AdventureWorks Shopper reference implementation was designed to be viewed full-screen in landscape orientation. Windows Store apps must adapt to different application view states, including both landscape and portrait orientations. AdventureWorks Shopper supports *FullScreenLandscape*, *FullScreenPortrait*, *Filled*, and *Snapped* layouts. AdventureWorks Shopper uses the [VisualState](#) class to specify changes to the visual display to support each layout. The [VisualStateManager](#) class, used by the **VisualStateAwarePage** class, manages state and the logic for transitioning between states for controls. For example, here is the XAML specification of the layout changes for the *FullScreenPortrait* layout on the hub page.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<VisualState x:Name="FullScreenPortrait">
  <Storyboard>
    <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemsGridView"
                                   Storyboard.TargetProperty="Padding">
      <DiscreteObjectKeyFrame KeyTime="0"
                              Value="96,0,10,56" />
    </ObjectAnimationUsingKeyFrames>
  </Storyboard>
</VisualState>
```

We directly update individual properties for XAML elements, in order to specify changes to the visual display. For instance, here the [Storyboard](#) specifies that the [Padding](#) property of the [GridView](#) control named **itemGridView** will change to a value of "96,0,10,56" when the view state changes to portrait. However, you could update the [Style](#) property when you need to update multiple properties or when there is a defined style that does what you want. Although styles enable you to control multiple properties and also provide a consistent appearance throughout your app, providing too many can make your app difficult to maintain. Therefore, only use styles when it makes sense to do so. For more info about styling controls, see [Quickstart: styling controls](#).

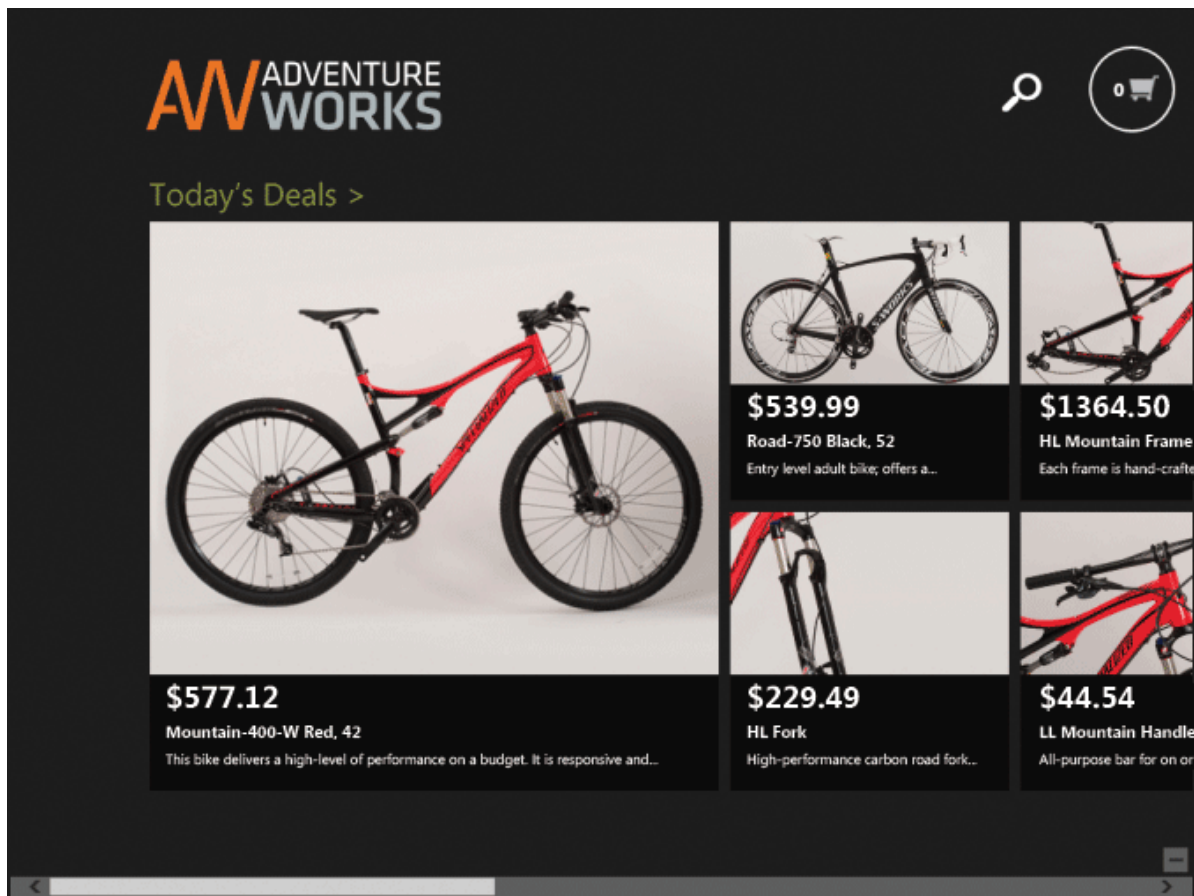
Tip When you develop an app in Visual Studio, you can use the Windows Simulator debugger to test layouts. To do this, press F5 and use the debugger tool bar to debug with the Windows Simulator. You can also use Blend to define and test layouts.

For more info see [Adapting to different layouts](#).

Loading the hub page at runtime

The XAML UI framework provides a built-in navigation model that uses [Frame](#) and [Page](#) elements and works much like the navigation in a web browser. The **Frame** control hosts **Pages**, and has a navigation history that you can use to go back and forward through pages you've visited. You can also pass primitive type data between pages as you navigate. In the Visual Studio project templates, a **Frame** named **rootFrame** is set as the content of the app window.

When the AdventureWorks Shopper reference implementation starts up, and after the bootstrapping process has completed, the **OnLaunchApplication** method of the **App** class navigates to the app's hub page, provided that the app hasn't been launched from a secondary tile.



The **App** class derives from the **MvvmAppBase** class in the [Microsoft.Practices.Prism.StoreApps](#) library that in turn derives from the [Windows.UI.Xaml.Application](#) class and overrides the [OnLaunched](#) method. The **OnLaunched** method override calls the **OnLaunchApplication** method in the **App** class, which is shown in the following code example.

C#: AdventureWorks.Shopper\App.xaml.cs

```
protected override void OnLaunchApplication(LaunchActivatedEventArgs args)
{
    if (args != null && !string.IsNullOrEmpty(args.Arguments))
    {
        // The app was launched from a Secondary Tile
        // Navigate to the item's page
        NavigationService.Navigate("ItemDetail", args.Arguments);
    }
    else
    {
        // Navigate to the initial page
        NavigationService.Navigate("Hub", null);
    }
}
```


This code example shows how AdventureWorks Shopper calls the **Navigate** method of the **NavigationService** object to load content that is specified by the page type. The [OnLaunched](#) method override in the **MvvmAppBase** class only calls the **OnLaunchApplication** if the [Frame](#) instance's [Content](#) property is null, as a way of determining whether the app is resuming from a previous state or starting in its default navigation state. For more info about resuming from previous states see [Handling suspend, resume and activation](#). For more info about navigation between pages see [Navigating between pages](#).

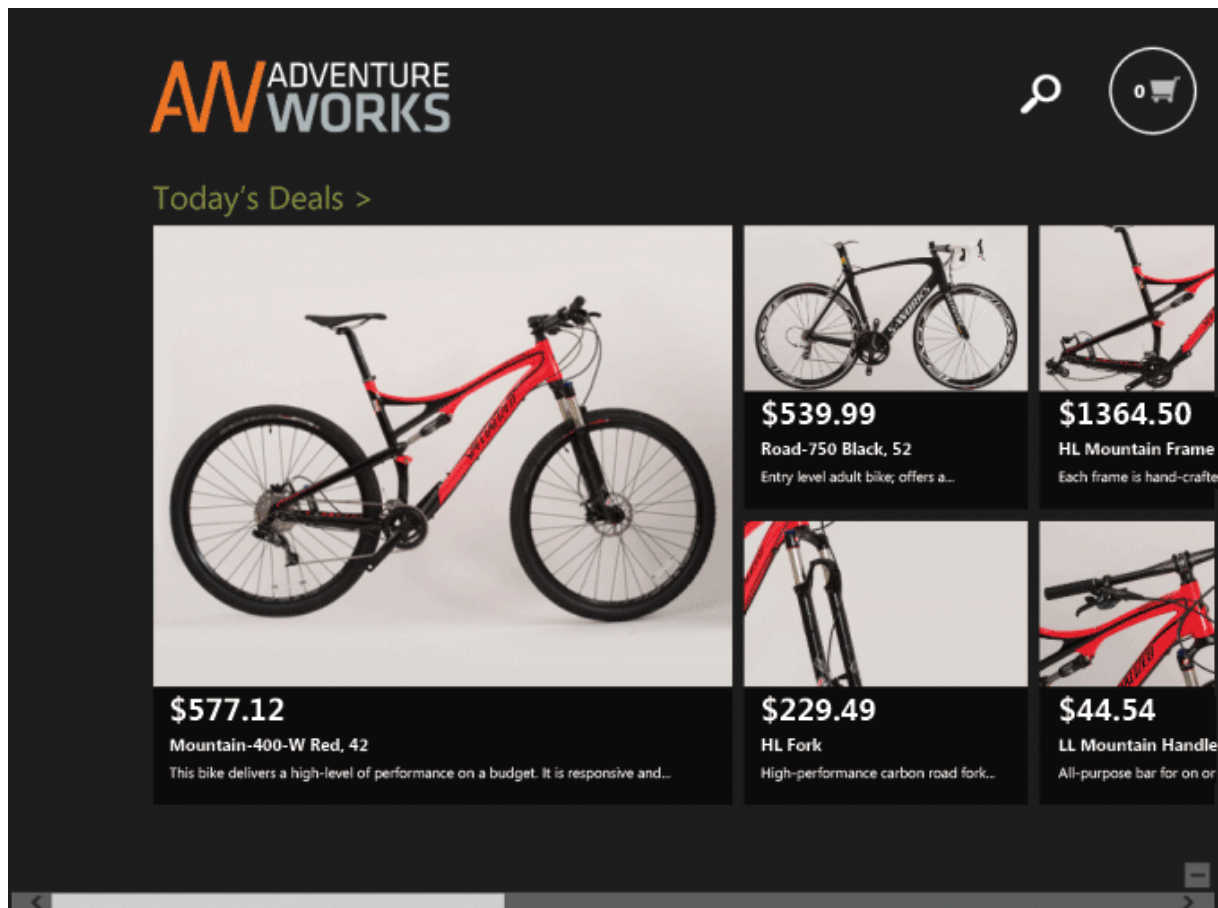
Styling controls

AdventureWorks Shopper's appearance was customized by styling and templating the controls used in the app. Styles enable you to set control properties and reuse those settings for a consistent appearance across multiple controls. Styles are defined in XAML either inline for a control, or as a reusable resource. Resources can be defined at the page level, app level, or in a separate resource dictionary. A resource dictionary can be shared across apps, and an app can use multiple resource dictionaries. For more info see [Quickstart: styling controls](#).

The structure and appearance of a control can be customized by defining a new [ControlTemplate](#) for the control. Templating a control can be used to avoid having to write a custom control. For more information, see [Quickstart: control templates](#). An example of this in AdventureWorks Shopper is the **FormFieldTextBox** control that derives from the [TextBox](#) control and adds a **Watermark** property to it.

Overriding built-in controls

On the hub page we wanted the first product to be displayed at twice the dimensions of the other products.



To do this we defined a new class named **MultipleSizeGridView** that derives from the [GridView](#) control. We then overrode the [PrepareContainerForItemOverride](#) method to enable the first product to span multiple rows and columns of the **MultipleSizeGridView**, as shown in the following code example.

C#: AdventureWorks.Shopper\Controls\MultipleSizeGridView.cs

```
protected override void PrepareContainerForItemOverride(DependencyObject element,
    object item)
{
    base.PrepareContainerForItemOverride(element, item);
    var dataItem = item as ProductViewModel;

    if (dataItem != null && dataItem.ItemPosition == 0)
    {
        _colVal = (int)LayoutSizes.PrimaryItem.Width;
        _rowVal = (int)LayoutSizes.PrimaryItem.Height;
    }
    else
    {
        _colVal = (int)LayoutSizes.SecondaryItem.Width;
        _rowVal = (int)LayoutSizes.SecondaryItem.Height;
    }
}
```

```

    }

    var uiElement = element as UIElement;
    VariableSizedWrapGrid.SetRowSpan(uiElement, _rowVal);
    VariableSizedWrapGrid.SetColumnSpan(uiElement, _colVal);
}

```

The [PrepareContainerForItemOverride](#) method gets the first item in the **MultipleSizedGridView** and sets it to span two rows and two columns, with subsequent items occupying one row and one column. The static **LayoutSizes** class simply defines two [Size](#) objects that specify the number of rows and columns to span for the first item, and subsequent items in the **MultipleSizedGridView**, respectively.

C#: AdventureWorks.Shopper\Controls\MultipleSizedGridView.cs

```

public static class LayoutSizes
{
    public static Size PrimaryItem
    {
        get { return new Size(2, 2); }
    }
    public static Size SecondaryItem
    {
        get { return new Size(1, 1); }
    }
}

```

Enabling page localization

Preparing for international markets can help you reach more users. [Globalizing your app](#) provides guidelines, checklists, and tasks to help you create a user experience that reaches more users by helping you to globalize and localize each page of your app. It's important to consider localization early on in the development process, as there are some issues that will effect user interface elements across various locales. Here's the tasks that we carried out to support page localization in the AdventureWorks Shopper reference implementation.

- Separate resources for each locale.
- Ensure that each piece of text that appears in the UI is defined by a string resource.
- Add contextual comments to the app resource file.
- Define the flow direction for all pages.
- Ensure error messages are read from the resource file.

Separate resources for each locale

We maintain separate solution folders for each locale. For example, **Strings -> en-US -> Resources.resw** defines the strings for the en-US locale. For more info see [Quickstart: Using string resources](#), and [How to name resources using qualifiers](#).

Ensure that each piece of text that appears in the UI is defined by a string resource

We used the [x:Uid](#) directive to provide a unique name for the localization process to associate localized strings with text that appears on screen. The following example shows the XAML that defines the app title that appears on the hub page.

XAML: AdventureWorks.Shopper\Views\ShoppingCartPage.xaml

```
<TextBlock x:Uid="ShoppingCartTitle"
           x:Name="pageTitle"
           Text="Shopping Cart"
           Grid.Column="1"
           TextTrimming="WordEllipsis"
           Style="{StaticResource PageHeaderTextStyle}" />
```

For the en-US locale, we define **ShoppingCartTitle.Text** in the resource file as "Shopping Cart." We specify the **.Text** part so that the XAML runtime will override the [Text](#) property of the [TextBlock](#) control with the value from the resource file. We also use this technique to set [Button](#) content ([ContentControl.Content](#)).

Add contextual comments to the app resource file

Comments in the resource file provide contextual information that helps localizers more accurately translate strings. For more info see [How to prepare for localization](#).

Define the flow direction for all pages

We define the [Page.FlowDirection](#) property in the string resources file to set the flow direction for all pages. For languages that use left-to-right reading order, such as English or German, we define "LeftToRight" as its value. For languages that read right-to-left, such as Arabic and Hebrew, you define this value as "RightToLeft". We also defined the flow direction for all app bars by defining **AppBar.FlowDirection** in the resource files.

Ensure error messages are read from the resource file

It's important to localize error messages strings, including exception message strings, because these strings will appear to the user. The AdventureWorks Shopper reference implementation uses an instance of the **ResourceLoaderAdapter** class to retrieve error messages from the resource file for your locale. This class uses an instance of the [ResourceLoader](#) class to load strings from the resource file. When we provide an error message when an exception is thrown, we use the **ResourceLoaderAdapter** instance to read the message text. The following code example shows how the **SubmitOrderTransactionAsync** method in the **CheckoutSummaryPageViewModel** class uses the **ResourceLoaderAdapter** instance to retrieve error message strings from the resource file.

C#: AdventureWorks.UILogic\ViewModels\CheckoutSummaryPageViewModel.cs

```
catch (ModelValidationException mvex)
{
    errorMessage = string.Format(CultureInfo.CurrentCulture,
        _resourceLoader.GetString("GeneralServiceErrorMessage"),
        Environment.NewLine, mvex.Message);
}

if (!string.IsNullOrEmpty(errorMessage))
{
    await _alertMessageService.ShowAsync(errorMessage,
        _resourceLoader.GetString("ErrorProcessingOrder"));
}
```

This code displays an exception error message to the user, if a **ModelValidationException** occurs when submitting an order. For the en-US locale, the "GeneralServiceErrorMessage" string is defined as "The following error messages were received from the service: {0}{1}," and the "ErrorProcessingOrder" string is defined as "There was an error processing your order." Other locales would have messages that convey the same error message.

Note When creating an instance of the **ResourceLoader** class that uses strings that are defined in a class library and not in the executable project, the **ResourceLoader** class has to be passed a path to the resources in the library. The path must be specified as /project name/Resources/ (for example, /Microsoft.Practices.Prism.StoreApps/Strings/).

You can test your app's localization by configuring the list of preferred languages in Control Panel. For more info about localizing your app and making it accessible, see [How to prepare for localization, Guidelines and checklist for application resources](#), and [Quickstart: Translating UI resources](#).

Enabling page accessibility

Accessibility is about making your app usable by people who have limitations that impede or prevent the use of conventional user interfaces. This typically means providing support for screen readers, implementing keyboard accessibility, and supporting high-contrast themes.

Accessibility support for Windows Store apps written in C# comes from the integrated support for the Microsoft UI Automation framework that is present in the base classes and the built-in behavior of the class implementation for XAML control types. Each control class uses automation peers and automation patterns that report the control's role and content to UI automation clients. If you use non-standard controls you will be responsible for making the controls accessible.

Here are the tasks that we carried out to support page accessibility in the AdventureWorks Shopper reference implementation:

- Set the accessible name for each UI element. An accessible name is a short, descriptive text string that a screen reader uses to announce a UI element. For example, in AdventureWorks Shopper XAML controls specify [AutomationProperties.AutomationId](#) and [AutomationProperties.Name](#) attached properties to make the control accessible to screen readers.

XAML: AdventureWorks.Shopper\Views\ItemDetailPage.xaml

```
<FlipView x:Name="flipView"
    AutomationProperties.AutomationId="ItemsFlipView"
    AutomationProperties.Name="Item Details"
    TabIndex="1"
    Grid.Row="1"
    ItemsSource="{Binding Items}"
    SelectedIndex="{Binding SelectedIndex, Mode=TwoWay}"
    SelectedItem="{Binding SelectedProduct, Mode=TwoWay}">
```

For more info see [Exposing basic information about UI elements](#).

- Overridden the **ToString** method of the **ShippingMethod**, **ProductViewModel**, **CheckoutDataViewModel**, and **ShoppingCartItemViewModel** classes in order to support [Windows Narrator](#). When instances of these classes are bound to the view they are styled using data templates, but Windows Narrator uses the result of the **ToString** overrides.
- Implemented keyboard accessibility. Ensure that the tab order of controls corresponds to the visual order of controls, and that UI elements that can be clicked can also be invoked by using the keyboard. For more info see [Implementing keyboard accessibility](#).
- Visually verified the UI to ensure that the text contrast is appropriate, and that elements render correctly in high-contrast themes. For more info see [Meeting requirements for accessible text](#) and [Supporting high contrast themes](#).
- Ran accessibility tools to verify the screen reading experience. For more info see [Testing your app for accessibility](#).
- Ensured that the app manifest follows accessibility guidelines. For more info see [Making tiles accessible](#).

For more info see [Making your app accessible](#).

Navigating between pages

Navigation within a Windows Store app can result from the user's interaction with the UI or from the app itself as a result of internal logic-driven state changes. Navigation usually involves moving from one page to another page in the app. In some cases, the app may implement complex logic to programmatically control navigation to ensure that certain business requirements are enforced. For

example, the app may not allow the user to navigate away from a page without first ensuring that the entered data is correct.

The AdventureWorks Shopper reference implementation typically triggers navigation requests from user interaction in the views. These requests could be to navigate to a particular view or navigate back to the previous view. In some scenarios, for example if the app needs to navigate to a new view when a command completes, the view model will need to send a message to the view. In other scenarios, you might want to trigger the navigation request directly from the view without involving the view model directly. When you're using the MVVM pattern, you want to be able to navigate without using any code-behind in the view, and without introducing any dependency on the view implementation in the view model classes.

The **INavigationAware** interface, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, allows an implementing class to participate in a navigation operation, and is implemented by view models classes. The interface defines two methods, as shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\INavigationAware.cs

```
public interface INavigationAware
{
    void OnNavigatedTo(object navigationParameter, NavigationMode navigationMode,
        Dictionary<string, object> viewModelState);
    void OnNavigatedFrom(Dictionary<string, object> viewModelState,
        bool suspending);
}
```

The **OnNavigatedFrom** and **OnNavigatedTo** methods are called during a navigation operation. If the view model class for the page being navigated from implements this interface, its **OnNavigatedFrom** method is called before navigation takes place. The **OnNavigatedFrom** method allows the page to save any state before it is disposed of. If the view model class for the page being navigated to implements this interface, its **OnNavigatedTo** method is called after navigation is complete. The **OnNavigatedTo** method allows the newly displayed page to initialize itself by loading any page state, and by using any navigation parameters passed to it. For example, the **OnNavigatedTo** method in the **ItemDetailPageViewModel** class accepts a product number as a parameter that is used to load the product information for display on the **ItemDetailPage**.

The **ViewModel** base class implements the **INavigationAware** interface, providing virtual **OnNavigatedFrom** and **OnNavigatedTo** methods that save and load view model state, respectively. This avoids each view model class having to implement this functionality to support the suspend and resume process. The view model classes for each page derive from the **ViewModel** class. The **OnNavigatedFrom** and **OnNavigatedTo** methods can then be overridden in the view model class for the page if any additional navigation logic is required, such as processing a navigation parameter that has been passed to the page.

Note The **OnNavigatedFrom** and **OnNavigatedTo** methods in the **ViewModel** base class control loading and saving page state during navigation operations. For more info see [Handling suspend, resume, and activation](#).

Handling navigation requests

Navigation is performed using the **FrameNavigationService** class. This class, which implements the **INavigationService** interface, uses the [Frame](#) instance created in the **InitializeFrameAsync** method in the **MvvmAppBase** class to perform the navigation request for the app. The **MvvmAppBase** class creates an instance of the **FrameNavigationService** class by calling the **CreateNavigationService** method, which is shown in the following code example.

Note Using the [Frame](#) instance ensures that the correct navigation stack is maintained for the app, so that navigating backwards works the way users expect.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```
private INavigationService CreateNavigationService(IFrameFacade rootFrame,
    ISessionStateService sessionStateService)
{
    var navigationService = new FrameNavigationService(rootFrame, GetPageType,
        sessionStateService);
    return navigationService;
}
```

The **CreateNavigationService** method creates an instance of the **FrameNavigationService** class, which takes the **GetPageType** delegate to implement a page type resolution strategy. This strategy assumes that the views that define pages are in the AdventureWorks.Shopper assembly and that the view names end with "Page".

After creating the instance of the **FrameNavigationService** class the **MvvmAppBase** class calls the **OnInitialize** override in the **App** class to register service instances with the Unity dependency injection container. When viewmodel classes are instantiated, the container will inject the dependencies that are required including the **FrameNavigationService** instance. View models can then invoke the **Navigate** method on the **FrameNavigationService** instance to cause the app to navigate to a particular view in the app or the **GoBack** method to return to the previous view. The following code example shows the **Navigate** method in the **FrameNavigationService** class.

C#: Microsoft.Practices.Prism.StoreApps\FrameNavigationService.cs

```
public bool Navigate(string pageToken, object parameter)
{
    Type pageType = _navigationResolver(pageToken);

    if (pageType == null)
    {
        var resourceLoader =
            new ResourceLoader(Constants.StoreAppsInfrastructureResourceMapId);
```



```

        var error = string.Format(CultureInfo.CurrentCulture,
            resourceLoader.GetString(
                "FrameNavigationServiceUnableResolveMessage"), pageToken);
        throw new ArgumentException(error, "pageToken");
    }

    // Get the page type and parameter of the last navigation to check if we
    // are trying to navigate to the exact same page that we are currently on
    var lastNavigationParameter =
        _sessionStateService.SessionState.ContainsKey(LastNavigationParameterKey)
        ? _sessionStateService.SessionState[LastNavigationParameterKey] : null;
    var lastPageTypeFullName =
        _sessionStateService.SessionState.ContainsKey(LastNavigationPageKey) ?
        _sessionStateService.SessionState[LastNavigationPageKey] as string :
        string.Empty;

    if (lastPageTypeFullName != pageType.FullName ||
        !AreEquals(lastNavigationParameter, parameter))
    {
        return _frame.Navigate(pageType, parameter);
    }

    return false;
}

```

The **Navigate** method accepts a string parameter that represents the page to be navigated to, and a navigation parameter that represents the data to pass to the page being navigated to. Any data being passed to the page being navigated to will be received by the **OnNavigatedTo** method of the view model class for the page type. A **null** value is used as the navigation parameter if no data needs to be passed to the page being navigated to.

Placing the navigation logic in view model classes means that the navigation logic can be exercised through automated tests. In addition, the view model can then implement logic to control navigation to ensure that certain business rules are enforced. For instance, an app may not allow the user to navigate away from a page without first ensuring that the entered data is correct.

Invoking navigation

Navigation is usually triggered from a view by a user action. For instance, each page in the app has a top app bar which contains [ButtonBase](#)-derived controls that allow the user to navigate to the hub page and the shopping cart page. Rather than implement this functionality separately on each page, it is implemented as a user control named **TopAppBarUserControl** that is added to each page. The following code example shows the [Button](#) controls from the **TopAppBarUserControl** that allow the user to navigate to the hub page and the shopping cart page.

XAML: AdventureWorks.Shopper\Views\TopAppBarUserControl.xaml

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Left" Height="125"
```

```

Margin="0,15,0,0">
<Button x:Name="HomeAppBarButton" x:Uid="HomeAppBarButton"
AutomationProperties.AutomationId="HomeAppBarButton"
Command="{Binding HomeNavigationCommand}"
Margin="5,0"
Style="{StaticResource HouseStyle}"
Content="Home"
Height="125"/>
<Button x:Uid="ShoppingCartAppBarButton" x:Name="ShoppingCartAppBarButton"
AutomationProperties.AutomationId="ShoppingCartAppBarButton"
Command="{Binding ShoppingCartNavigationCommand}"
Margin="0,0,5,0"
Height="125"
Style="{StaticResource CartStyle}"
Content="Shopping Cart" />
</StackPanel>

```

In this scenario, navigation is triggered from one of the [ButtonBase](#)-derived controls by invoking a command in the **TopAppBarUserControlViewModel** class. For instance, executing the **ShoppingCartNavigationCommand** causes the app to navigate to the **ShoppingCartPage**, and so the navigation is initiated from the view model. The following code example shows how the **TopAppBarUserControlViewModel** constructor defines the **ShoppingCartNavigationCommand** property to be an instance of the **DelegateCommand** class that will invoke the navigation.

C#: AdventureWorks.UILogic\ViewModels\TopAppBarUserControlViewModel.cs

```

public TopAppBarUserControlViewModel(INavigationService navigationService)
{
    HomeNavigationCommand = new DelegateCommand(() =>
        navigationService.Navigate("Hub", null));
    ShoppingCartNavigationCommand = new DelegateCommand(() =>
        navigationService.Navigate("ShoppingCart", null));
}

public DelegateCommand HomeNavigationCommand { get; private set; }
public DelegateCommand ShoppingCartNavigationCommand { get; private set; }

```

For controls that do not derive from [ButtonBase](#), you can use an attached behavior to implement a navigation action in the view model class. For instance, when the user selects a product on the **HubPage**, they are taken to the **ItemDetailPage**. This functionality is provided by the **ListViewItemClickedToAction** attached behavior, which enables the [ItemClick](#) event of the [GridView](#) control to be handled in a view model, rather than in the page's code-behind.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```

<controls:MultipleSizedGridView x:Name="itemsGridView"
    AutomationProperties.AutomationId="HubPageItemGridView"
    AutomationProperties.Name="Grouped Items"
    Margin="0,-3,0,0"
    Padding="116,0,40,46"
    ItemsSource="{Binding Source={StaticResource groupedItemsViewSource}}"
    ItemTemplate="{StaticResource AWShopperItemTemplate}"
    SelectionMode="None"
    ScrollViewer.IsHorizontalScrollChainingEnabled="False"
    IsItemClickEnabled="True"
    behaviors:ListViewItemClickedToAction.Action=
        "{Binding ProductNavigationAction}">

```

The **ListViewItemClickedToAction** behavior binds the [ItemClick](#) event of the [GridView](#) to the **ProductNavigationAction** property in the **HubPageViewModel** class. So when a [GridViewItem](#) is selected the **ProductNavigationAction** is executed that in turn calls the **NavigateToItem** method to navigate from the **HubPage** to the **ItemDetailPage**.

C#: AdventureWorks.UILogic\ViewModels\HubPageViewModel.cs

```

private void NavigateToItem(object parameter)
{
    var product = parameter as ProductViewModel;
    if (product != null)
    {
        _navigationService.Navigate("ItemDetail", product.ProductNumber);
    }
}

```

For more info see [UI interaction using the DelegateCommand class and attached behaviors](#).

Using touch in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- When possible, use the standard touch gestures and controls that Windows 8 provides.
- Provide visual feedback when a touch interaction occurs.
- Use data binding to connect standard Windows controls to the view models that implement the touch interaction behavior.

Part of providing a great user experience is ensuring that an app is accessible and intuitive to use on a traditional desktop computer and on a small tablet. For the AdventureWorks Shopper reference implementation we put touch at the forefront of our user experience planning because it adds an important experience by providing a more engaging interaction between the user and the app. AdventureWorks Shopper provides tap, slide, pinch, and swipe gestures. Data binding is used to connect standard Windows controls that use touch gestures to the view models that implement those gestures.

You will learn

- How the Windows 8 touch language was used in AdventureWorks Shopper.
- How the Windows Runtime supports non-touch devices.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Touch interactions in Windows 8 use physical interactions to emulate the direct manipulation of UI elements and provide a more natural, real-world experience when interacting with those elements on the screen. The following list summarizes the decisions to make when implementing touch interactions in your app:

- Does the Windows 8 touch language provide the experience your app requires?
- What size should your touch targets be?
- When displaying a list of items, do the touch targets for each item need to be identically sized?
- Should you provide feedback to touch interactions?
- Should touch interactions be reversible?
- How long should a touch interaction last?
- When should you use static gestures versus manipulation gestures?
- Do you need to design and implement a custom interaction?

- Does the custom interaction require specific hardware support such as a minimum number of touch points?
- How will the custom interaction be provided on a non-touch device?

Windows 8 provides a concise set of touch interactions that are used throughout the system. Applying this language consistently makes your app feel familiar to what users already know, increasing user confidence by making your app easier to learn and use. Most apps will not require touch interactions that are not part of the Windows 8 touch language. For more info see [Touch interaction design](#).

There are no definitive recommendations for how large a touch target should be or where it should be placed within your app. However, there are some guidelines that should be followed. The size and target area of an object depend on various factors, including the user experience scenarios and interaction context. They should be large enough to support direct manipulation and provide rich touch interaction data. It is acceptable in some user experience scenarios for touch targets in a collection of items to be different sizes. For instance, when displaying a collection of products you could choose to display some products at a larger size than the majority of the collection, in order to draw attention to specific products. Touch targets should react by changing color, changing size, or by moving. Non-moving elements should return to their default state when the user slides or lifts their finger off the element. In addition, touch interactions should be reversible. You can make your app safe to explore using touch by providing visual feedback to indicate what will happen when the user lifts their finger. For more info see [Guidelines for targeting](#) and [Guidelines for visual feedback](#).

Touch interactions that require compound or custom gestures need to be performed within a certain amount of time. Try to avoid timed interactions like these because they can often be triggered accidentally and can be difficult to time correctly. For more info see [Responding to user interaction](#).

Static gestures events are triggered after an interaction is complete and are used to handle single-finger interactions such as tapping. Manipulation gesture events indicate an ongoing interaction and are used for dynamic multi-touch interactions such as pinching and stretching, and interactions that use inertia and velocity data such as panning. This data is then used to determine the manipulation and perform the interaction. Manipulation gesture events start firing when the user touches the element and continue until the user lifts their finger or the manipulation is cancelled. For more info see [Gestures, manipulations, and interactions](#).

Only create a custom interaction and if there is a clear, well-defined requirement and no interaction from the Windows 8 touch language can support your scenario. If an existing interaction provides the experience your app requires, adapt your app to support that interaction. If you do need to design and implement a custom interaction you will need to consider your interaction experience. If the interaction depends on items such as the number of touch points, velocity, and inertia, ensure that these constraints and dependencies are consistent and discoverable. For example, how users interpret speed can directly affect the functionality of your app and the users satisfaction with the experience. In addition, you will also have to design and implement an equivalent version of the interaction for non-touch devices. For more info see [Responding to user interaction](#).

Important To avoid confusing users, do not create custom interactions that duplicate or redefine existing, standard interactions.

Touch in AdventureWorks Shopper

As previously described in [Designing the UX](#), touch is more than simply an alternative to using a mouse. We wanted to make touch an integrated part of the app because touch can add a personal connection between the user and the app. Touch is also a natural way to enable users to browse and select products. In addition, we use SemanticZoom to highlight how levels of related complexity can easily be navigated. With SemanticZoom users can easily visualize high level content such as categories, and then zoom into those categories to view category items.

The AdventureWorks Shopper reference implementation uses the Windows 8 touch language. We use the standard touch interactions that Windows provides for these reasons:

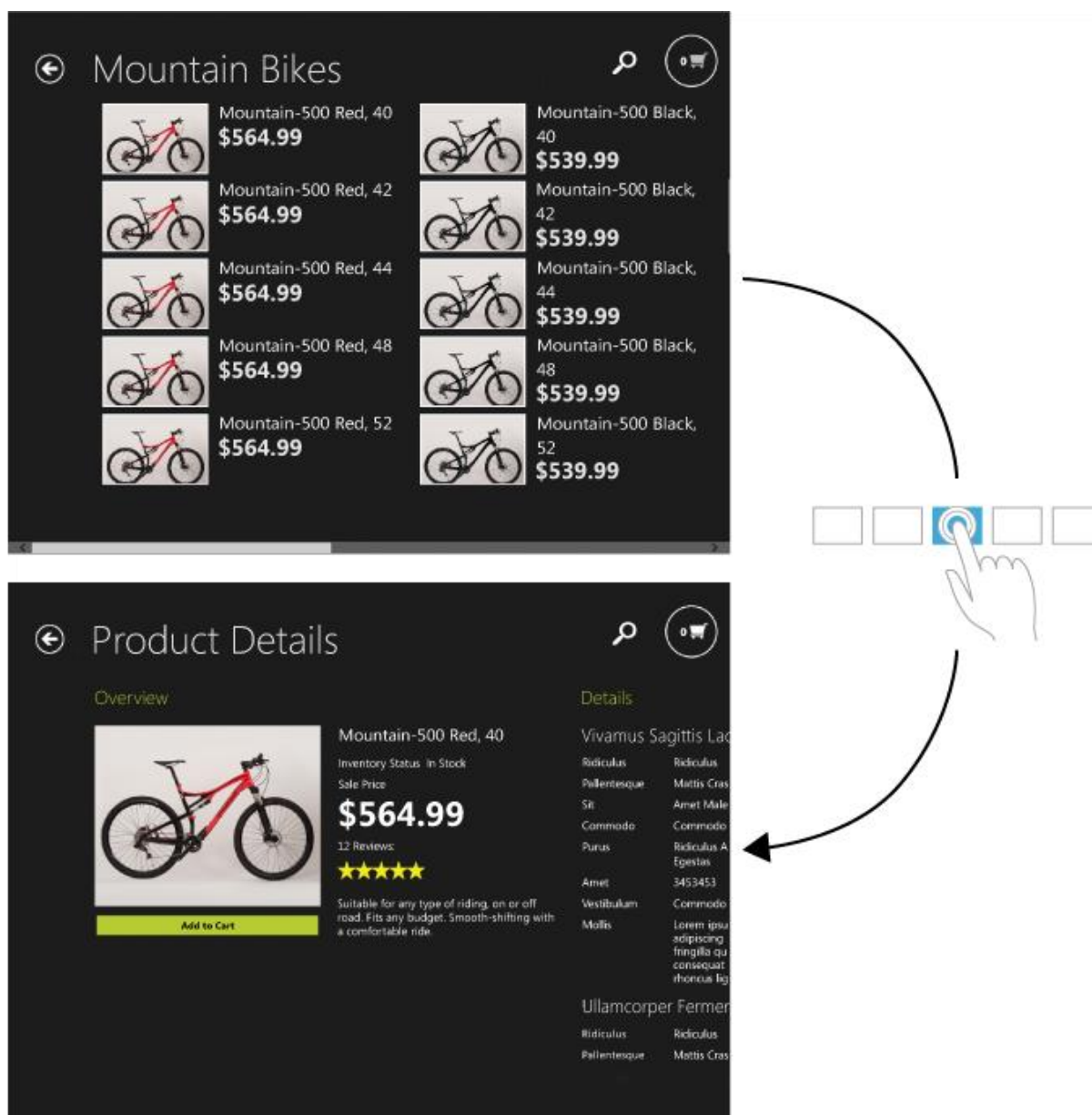
- The Windows Runtime provides an easy way to work with them.
- We don't want to confuse users by creating custom interactions.
- We want users to use the interactions that they already know to explore the app, and not need to learn new interactions.

We also wanted AdventureWorks Shopper to be intuitive for users who use a mouse or similar pointing device. The built-in controls work as well with a mouse or other pointing device as they do with touch. So when you design for touch, you also get mouse and pen functionality. For example, you can use the left mouse button to invoke commands. In addition, mouse and keyboard equivalents are provided for many commands. For example, you can use the right mouse button to activate the app bar, and holding the Ctrl key down while scrolling the mouse wheel controls SemanticZoom interaction. For more info see [Guidelines for common user interactions](#).

The document [Touch interaction design](#) explains the Windows 8 touch language. The following sections describe how we applied the Windows 8 touch language in AdventureWorks Shopper.

Tap for primary action

Tapping an element invokes its primary action. For example, on the **GroupDetailPage**, you tap on a product to navigate to the **ItemDetailPage**. The following diagram shows an example of the tap for primary action gesture in the AdventureWorks Shopper reference implementation.



Products are displayed on the **GroupDetailPage** in a [GridView](#) control. A **GridView** displays a collection of items in a horizontal grid. The **GridView** control is an [ItemsControl](#) class, so it can contain a collection of items of any type. A benefit of using the **GridView** control is that it has touch capabilities built in, removing the need for additional code.

To populate a **GridView** you can add objects directly to its [Items](#) collection or bind its [ItemsSource](#) property to a collection of data items. When you add items to a **GridView** they are automatically placed in a [GridViewItem](#) container that can be styled to change how an item is displayed.

XAML: AdventureWorks.Shopper\Views\GroupDetailPage.xaml

```

<GridView Grid.Row="1"
    x:Name="itemsGridView"
    AutomationProperties.AutomationId="ItemsGridView"
    AutomationProperties.Name="Items In Category"
    TabIndex="1"
    Margin="0,-14,0,0"
    Padding="120,0,120,50"
    ItemsSource="{Binding Items}"
    ItemTemplate="{StaticResource ProductTemplate}"
    SelectionMode="None"
    IsItemClickEnabled="True"
    behaviors:ListViewItemClickedToAction.Action=
        "{Binding ProductNavigationAction}">
    <GridView.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapGrid Loaded="wrapGrid_Loaded" />
        </ItemsPanelTemplate>
    </GridView.ItemsPanel>
</GridView>

```

The **ItemsSource** property specifies that the **GridView** will bind to the **Items** property of the **GroupDetailPageViewModel** class. The **Items** property is initialized to a collection of type **ProductViewModel** when the **GroupDetailPage** is navigated to.

The appearance of individual items in the **GridView** is defined by the [ItemTemplate](#) property. A [DataTemplate](#) is assigned to the **ItemTemplate** property that specifies that each item in the **GridView** will display the product subtitle, image, and description.

When a user clicks an item in the [GridView](#) the app navigates to the **ItemDetailPage**. This behavior is enabled by setting the [SelectionMode](#) property to **None**, setting the [IsItemClickEnabled](#) property to **true**, and handling the [ItemClick](#) event. The **GridView** uses an attached behavior named **ListViewItemClickedToAction** that enables the **ItemClick** event to be handled in a view model, rather than in the page's code-behind. The behavior binds the **ItemClick** event to the **ProductNavigationAction** property in the **GroupDetailPageViewModel** class.

In the **GroupDetailPageViewModel** constructor, the **ProductNavigationAction** property is initialized to the **NavigateToProduct** method. This method navigates to the **ItemDetailPage**, and passes in a specific product number to the page for loading.

C#: AdventureWorks.UILogic\ViewModels\GroupDetailPageViewModel.cs

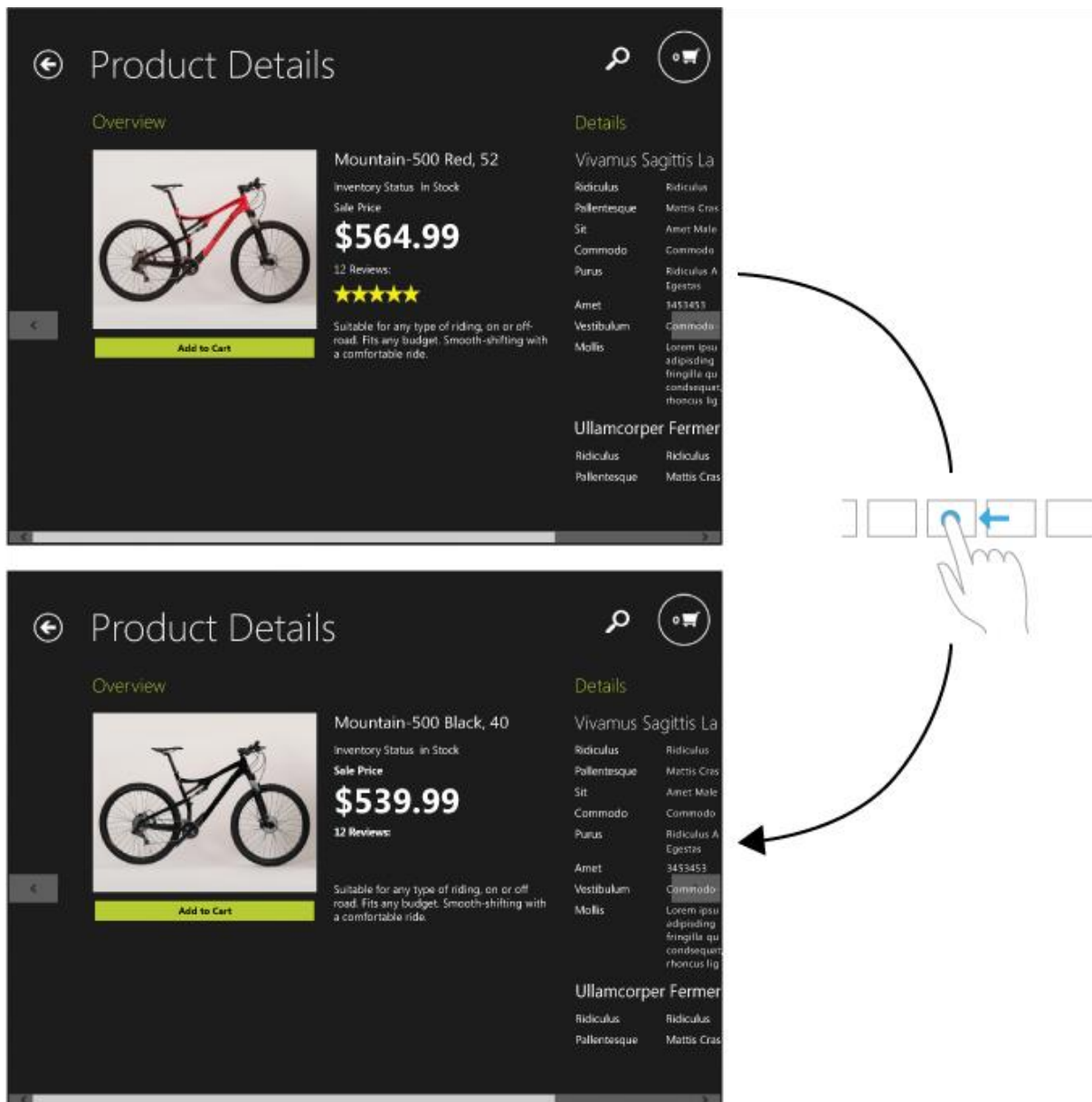
```
private void NavigateToProduct(object parameter)
{
    var product = parameter as ProductViewModel;
    if (product != null)
    {
        _navigationService.Navigate("ItemDetail", product.ProductNumber);
    }
}
```

The overall effect is that when a [GridViewItem](#) is clicked on the **ProductNavigationAction** is executed and navigates to the **ItemDetailPage** to display detailed product information. For more info about behaviors see [Implementing behaviors to supplement the functionality of XAML elements](#).

For more info see [Adding ListView and GridView controls](#).

Slide to pan

The slide gesture is primarily used for panning interactions. Panning is a technique for navigating short distances over small sets of content within a single view. Panning is only necessary when the amount of content in the view causes the content area to overflow the viewable area. For more info see [Guidelines for panning](#). One of the uses of the slide gesture in the AdventureWorks Shopper reference implementation is to pan among products in a category. For example, when you browse to a product, you can use the slide gesture to navigate to the previous or next product in the subcategory. The following diagram shows an example of the slide to pan gesture in AdventureWorks Shopper.



In AdventureWorks Shopper this gesture is implemented by the [FlipView](#) control. The **FlipView** control displays a collection of items, and lets you flip through them one at a time. The **FlipView** control is derived from the [ItemsControl](#) class, like the [GridView](#) control, and so it shares many of the same features. A benefit of using the **FlipView** control is that it has touch capabilities built in, removing the need for additional code.

To populate a **FlipView** you can add objects directly to its [Items](#) collection or bind its [ItemsSource](#) property to a collection of data items. When you add items to a **FlipView** they are automatically placed in a [FlipViewItem](#) container that can be styled to change how an item is displayed.

XAML: AdventureWorks.Shopper\Views\ItemDetailPage.xaml

```
<FlipView x:Name="flipView"
    AutomationProperties.AutomationId="ItemsFlipView"
    AutomationProperties.Name="Item Details "
    TabIndex="1"
    Grid.Row="1"
    ItemsSource="{Binding Items}"
    SelectedIndex="{Binding SelectedIndex, Mode=TwoWay}"
    SelectedItem="{Binding SelectedProduct, Mode=TwoWay}">
```

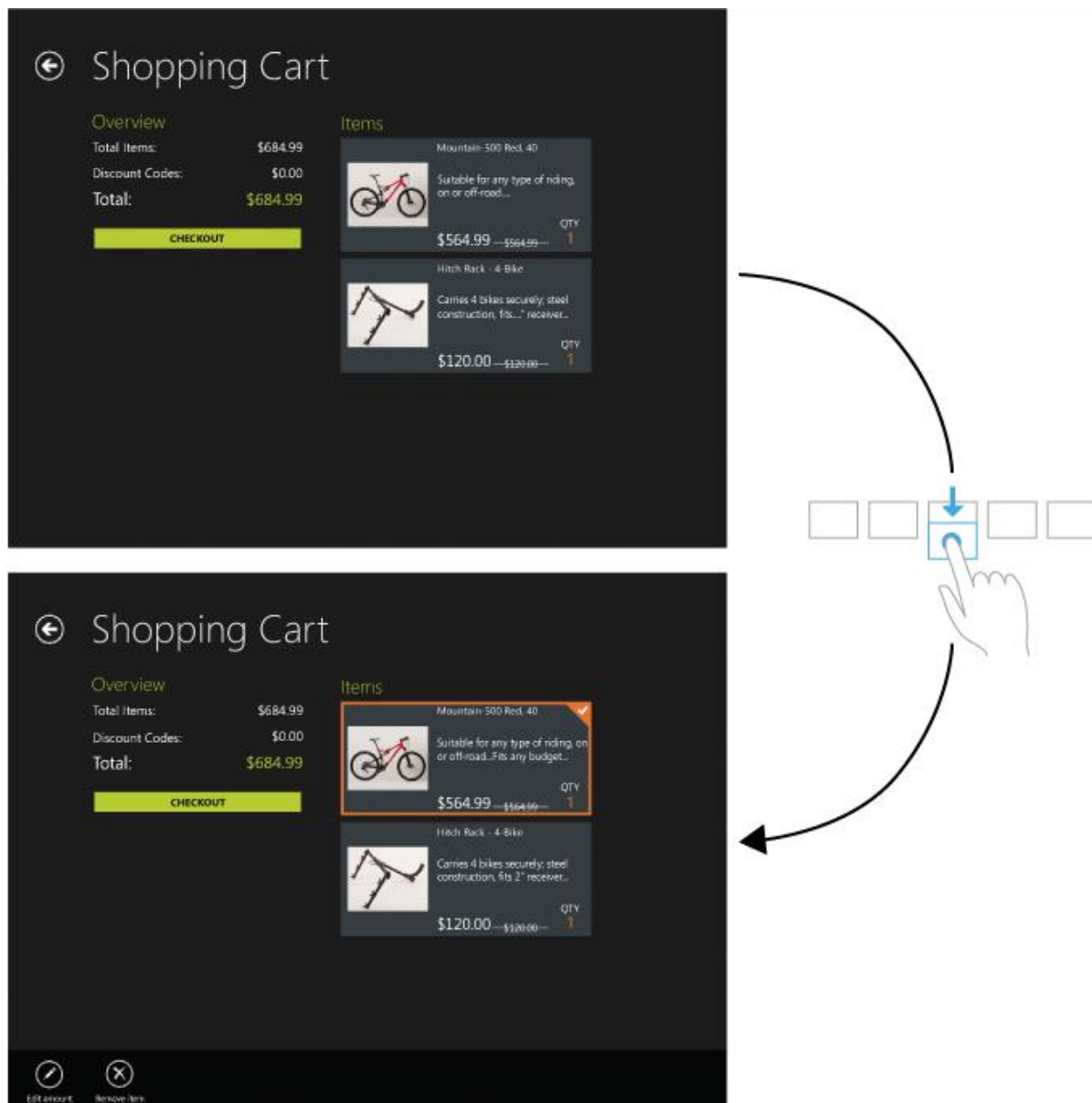
The **ItemsSource** property specifies that the **FlipView** binds to the **Items** property of the **ItemDetailPageViewModel** class, which is a collection of type **ProductViewModel**.

For more info see [Quickstart: Adding FlipView controls](#), [How to add a flip view](#), [Guidelines and checklist for FlipView controls](#).

Swipe to select, command, and move

With the swipe gesture, you slide your finger perpendicular to the panning direction to select objects. The ability to use the swipe gesture depends upon the value of the [SelectionMode](#) property on the [ListView](#) or [GridView](#) control. A value of [None](#) indicates that item selection is disabled, while a value of **Single** indicates that single items can be selected using this gesture.

In the AdventureWorks Shopper reference implementation, the swipe gesture can be used to select items on the **ChangeDefaultsFlyout**, the **CheckoutSummaryPage**, and the **ShoppingCartPage**. When an item is selected on the **ShoppingCartPage** the bottom app bar appears with the app bar commands applying to the selected item. The following diagram shows an example of the swipe to select, command, and move gesture in AdventureWorks Shopper.



The [IsSwipeEnabled](#) property of the **GridView** control indicates whether a swipe gesture is enabled for the control. Setting **IsSwipeEnabled** to **false** disables some default touch interactions, so it should be set to **true** when these interactions are required. For example, when **IsSwipeEnabled** is **false**:

- If item selection is enabled, a user can deselect items by right-clicking with the mouse, but cannot deselect an item with touch by using the swipe gesture.
- If [CanDragItems](#) is **true**, a user can drag items with the mouse, but not with touch.
- If [CanReorderItems](#) is **true**, a user can reorder items with the mouse, but not with touch.

The AdventureWorks Shopper reference implementation does not explicitly set the [IsSwipeEnabled](#) property, as its default value is **true**. The following code example shows how an item on the **ShoppingCartPage** can be selected with the swipe gesture.

XAML: AdventureWorks.Shopper\Views\ShoppingCartPage.xaml

```
<GridView x:Name="ShoppingCartItemsGridView"
    x:Uid="ShoppingCartItemsGridView"
    AutomationProperties.AutomationId="ShoppingCartItemsGridView"
    SelectionMode="Single"
    Width="Auto"
    Grid.Row="2"
    Grid.Column="1"
    Grid.RowSpan="3"
    VerticalAlignment="Top"
    ItemsSource="{Binding ShoppingCartItemViewModels}"
    SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
    ItemTemplate="{StaticResource ShoppingCartItemTemplate}"
    Margin="0,0,0,0" />
```

The [SelectedItem](#) property of the **GridView** control can be used to retrieve the item selected by the swipe gesture. Here the **SelectedItem** property performs a two-way binding to the **SelectedItem** property of the **ShoppingCartPageViewModel** class, which is shown in the following code example.

C#: AdventureWorks.UILogic\ViewModels\ShoppingCartPageViewModel.cs

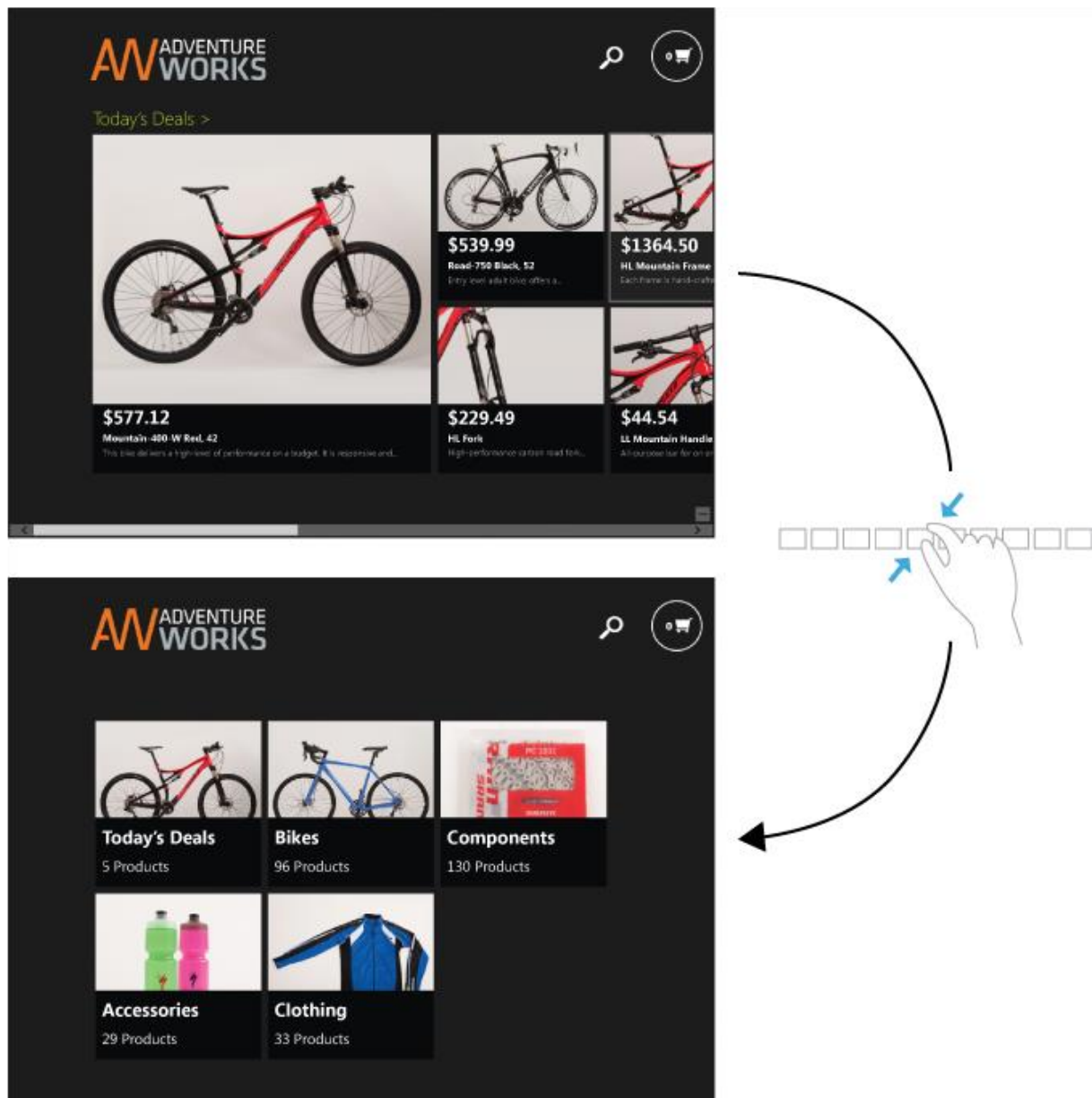
```
public ShoppingCartItemViewModel SelectedItem
{
    get { return _selectedItem; }
    set
    {
        if (SetProperty(ref _selectedItem, value))
        {
            if (_selectedItem != null)
            {
                // Display the AppBar
                IsBottomAppBarOpened = true;

                IncrementCountCommand.RaiseCanExecuteChanged();
                DecrementCountCommand.RaiseCanExecuteChanged();
            }
            else
            {
                IsBottomAppBarOpened = false;
            }
        }
    }
}
```

When the **SelectedItem** property is set the **IsBottomAppBarOpened** property will be set to control whether or not to display the bottom app bar.

Pinch and stretch to zoom

Pinch and stretch gestures are not just for magnification, or performing *optical zoom*. The AdventureWorks Shopper reference implementation uses SemanticZoom to help users navigate between large sets of data. SemanticZoom enables you to switch between two different views of the same content. You typically have a main view of your content and a second view that allows users to quickly navigate through it. Users can pan or scroll through categories of content, and then zoom into those categories to view detailed information. The following diagram shows an example of the pinch and stretch to zoom gesture in AdventureWorks Shopper.



To provide this zooming functionality, the [SemanticZoom](#) control uses two other controls—one to provide the zoomed-in view and one to provide the zoomed-out view. These controls can be any two controls that implement the [ISemanticZoomInformation](#) interface. XAML provides the **Listview** and **GridView** controls that meet this criteria.

Tip When you use a **GridView** in a [SemanticZoom](#) control, always set the [ScrollViewer.IsHorizontalScrollChainingEnabled](#) attached property to false on the [ScrollViewer](#) that's in the **GridView**'s control template.

For the zoomed-in view, we display a **GridView** that binds to products that are grouped by sub-category. The **GridView** also shows a title (the category) for each group.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<SemanticZoom.ZoomedInView>
  <!-- Horizontal scrolling grid used in most view states -->
  <controls:MultipleSizedGridView x:Name="itemsGridView"
    AutomationProperties.AutomationId="HubPageItemGridView"
    AutomationProperties.Name="Grouped Items"
    Margin="0,-3,0,0"
    Padding="116,0,40,46"
    ItemsSource="{Binding Source={StaticResource groupedItemsViewSource}}"
    ItemTemplate="{StaticResource AWShopperItemTemplate}"
    SelectionMode="None"
    ScrollViewer.IsHorizontalScrollChainingEnabled="False"
    IsItemClickEnabled="True"
    behaviors:ListViewItemClickedToAction.Action=
      "{Binding ProductNavigationAction}" />
```

The **ItemsSource** property specifies the items to be displayed by the **GridView**. The **groupedItemsViewSource** static resource is a [CollectionViewSource](#) that provides the source data for the control.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<CollectionViewSource x:Name="groupedItemsViewSource"
  Source="{Binding Path=RootCategories}"
  d:DesignSource="{Binding RootCategories,
    Source={d:DesignInstance
      designViewModels:HubPageDesignViewModel,
      IsDesignTimeCreatable=True}}"
  IsSourceGrouped="true"
  ItemsPath="Products" />
```

The **RootCategories** property on the **HubPageViewModel** specifies the data that is bound to the **GridView** for the zoomed-in view. **RootCategories** is a collection of **CategoryViewModel** objects. The [ItemsPath](#) property refers to the **Products** property of the **CategoryViewModel** class. Therefore, the **GridView** will show each product grouped by the category it belongs to.

For the zoomed-out view, we display a **GridView** that binds to filled rectangles for each category. Within each category the category title and number of products is displayed.

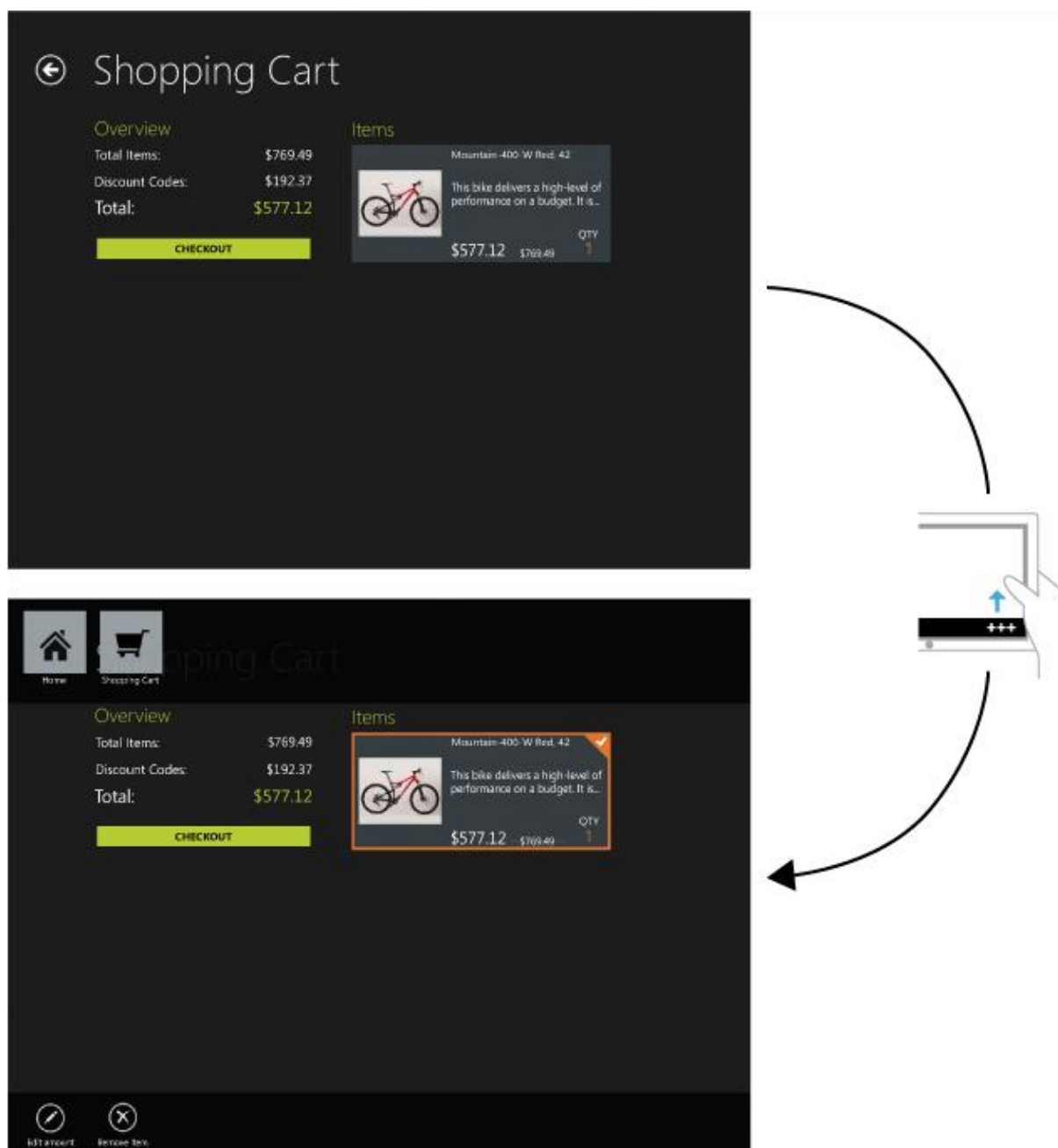
XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<SemanticZoom.ZoomedOutView>
  <GridView Padding="120,0,0,0"
    Foreground="White"
    AutomationProperties.AutomationId="HubPageGridView"
    ScrollViewer.IsHorizontalScrollChainingEnabled="False"
    ItemTemplate="{StaticResource AWShopperItemTemplateSemanticZoom}">
```

For more info about SemanticZoom, see [Adding SemanticZoom controls](#), and [Guidelines for SemanticZoom](#).

Swipe from edge for app commands

When there are relevant commands to display, the Adventure Works Shopper reference implementation displays the app bar when the user swipes from the bottom or top edge of the screen. Every page can define a top app bar, a bottom app bar, or both. For instance, AdventureWorks Shopper displays both when you activate the app bars on the **ShoppingCartPage**. The following diagram shows an example of the swipe from edge for app commands gesture in AdventureWorks Shopper.



The [AppBar](#) control is a toolbar for displaying app-specific commands. AdventureWorks Shopper displays app bars on each page. The [Page.TopAppBar](#) property can be used to define the top app bar, with the [Page.BottomAppBar](#) property being used to define the bottom app bar. Each of these properties will contain an **AppBar** control that holds the app bar's UI components. In general, you should use the bottom app bar for contextual commands that act on the currently selected item on the page. Use the top app bar for navigational elements that move the user to a different page.

AdventureWorks Shopper implements the top app bar for each page as a user control named **TopAppBarUserControl**. This user control simply defines the [Button](#) controls that will appear in the top app bar. Each **Button** binds to a command in the **TopAppBarUserControlViewModel** class.

XAML: AdventureWorks.Shopper\Views\TopAppBarUserControl.xaml

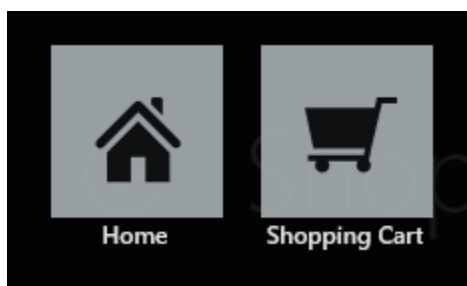
```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Left" Height="125"
    Margin="0,15,0,0">
    <Button x:Name="HomeAppBarButton" x:Uid="HomeAppBarButton"
        AutomationProperties.AutomationId="HomeAppBarButton"
        Command="{Binding HomeNavigationCommand}"
        Margin="5,0"
        Style="{StaticResource HouseStyle}"
        Content="Home"
        Height="125"/>
    <Button x:Uid="ShoppingCartAppBarButton" x:Name="ShoppingCartAppBarButton"
        AutomationProperties.AutomationId="ShoppingCartAppBarButton"
        Command="{Binding ShoppingCartNavigationCommand}"
        Margin="0,0,5,0"
        Height="125"
        Style="{StaticResource CartStyle}"
        Content="Shopping Cart" />
</StackPanel>
```

The **Page.TopAppBar** property on each page then uses the **TopAppBarUserControl** to define the top app bar.

XAML: AdventureWorks.Shopper\Views\HubPage.xaml

```
<Page.TopAppBar>
    <AppBar Style="{StaticResource AppBarStyle}"
        x:Uid="TopAppBar">
        <views:TopAppBarUserControl />
    </AppBar>
</Page.TopAppBar>
```

The following diagram shows the top app bar buttons for each page.



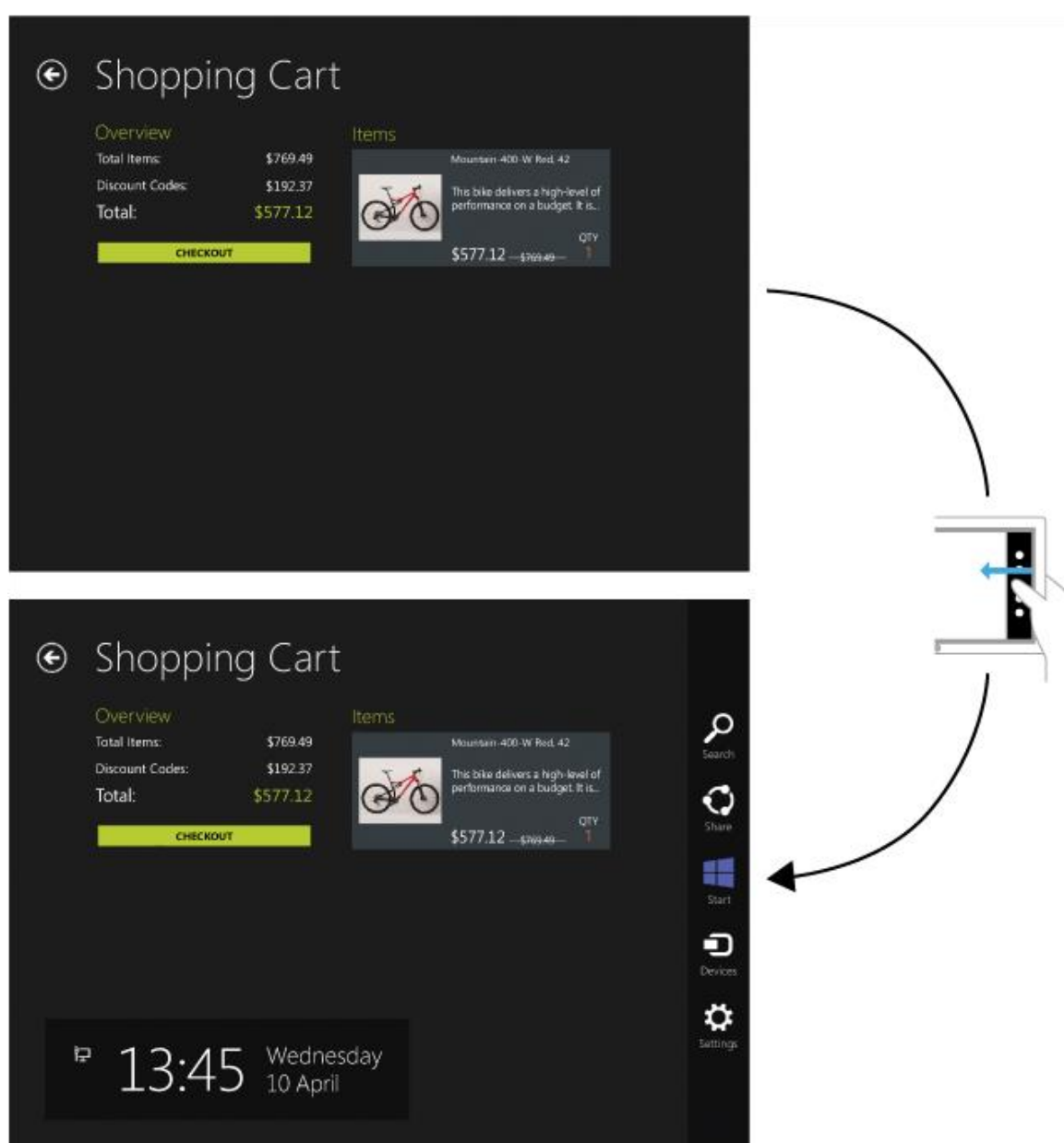
When an item on a page is selected, the app bar is shown in order to display contextual commands, by setting the [IsOpen](#) property on the **AppBar** control. If you have contextual commands on an app bar, the mode should be set to sticky while the context exists so that the bar doesn't automatically hide when the user interacts with the app. When the context is no longer present, sticky mode can be turned off. This can be achieved by setting the [IsSticky](#) property on the **AppBar** control.

Per UI guidelines for app bars, we display labels on the app bar buttons in landscape mode and hide the labels in snap and portrait mode.

For more information see [Adding app bars](#), [How to use an app bar in different views](#), and [Guidelines and checklist for app bars](#).

Swipe from edge for system commands

Users can swipe from the edge of the screen to reveal app bars and charms, or to display previously used apps. Therefore, it is important to maintain a sufficient distance between app controls and the screen edges. The following diagram shows an example of the swipe from edge for system commands gesture in AdventureWorks Shopper.



For more info see [Laying out an app page](#).

Validating user input in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Derive model classes from the **ValidatableBindableBase** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, in order to participate in client-side validation.
- Specify validation rules for model properties by adding data annotation attributes to the properties.
- Call the **ValidatableBindableBase.ValidateProperties** method to validate all the properties in a model object that possess an attribute that derives from the [ValidationAttribute](#) attribute.

Business apps such as shopping cart apps require users to enter data that must be validated for correctness. This article demonstrates how to validate form-based input in a Windows Store app by using [Prism for the Windows Runtime](#).

You will learn

- How to validate data stored in a bound model object.
- How to specify validation rules for model properties by using data annotations.
- How to trigger validation when property values change.
- How to highlight validation errors with attached behaviors.
- How to save validation errors when the app suspends, and restore them when the app is reactivated after termination.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Any app that accepts input from users should ensure that the data is valid. An app could, for example, check that the input contains only characters in a particular range, is of a certain length, or matches a particular format. Without validation, a user can supply data that causes the app to fail. Validation forces business rules, and prevents an attacker from injecting malicious data. The following list summarizes the decisions to make when implementing validation in your app:

- Should I validate user input on the client, on the server, or on both?
- Should I validate user input synchronously or asynchronously?
- Should I validate user input in view model objects or in model objects?
- How should I specify validation rules?
- How should I notify the user about validation errors?
- What approach should I use for saving validation errors when the app suspends?

Validation can be performed client-side, server-side, or both. Validation on the client provides a convenient way for the user to correct input mistakes without round trips to the server. Validation on the server should be used when server-side resources are required, such as a list of valid values stored in a database, against which the input can be compared. Although client-side validation is necessary, you should not rely solely on it because it can easily be bypassed. Therefore, you should provide client-side and server-side validation. This approach provides a security barrier that stops malicious users who bypass the client-side validation.

Synchronous validation can check the range, length, or structure of user input. User input should be validated synchronously when it is captured.

User input could be validated in view model objects or in model objects. However, validating data in view models often means duplicating model properties. Instead, view models can delegate validation to the model objects they contain, with validation then being performed on the model objects. Validation rules can be specified on the model properties by using data annotations that derive from the [ValidationAttribute](#) class.

Users should be notified about validation errors by highlighting the control that contains the invalid data, and by displaying an error message that informs the user why the data is invalid. There are guidelines and requirements for the placement of error messages in Windows Store apps. For more info see [Guidelines and checklist for text input](#).

When a suspended app is terminated and later reactivated by the operating system, the app should return to its previous operational and visual state. If your app is on a data entry page when it suspends, user input and any validation error messages should be saved to disk, and restored if the app is terminated and subsequently reactivated. For more info see [Guidelines for app suspend and resume](#).

Validation in AdventureWorks Shopper

The AdventureWorks Shopper reference implementation uses the [Microsoft.Practices.Prism.StoreApps](#) library to perform client-side and server-side validation. Synchronous validation of data stored in model objects is performed client-side in order to check the range, length, and structure of user input. Validation that involves server-side business rules, such as ensuring that entered zip codes are valid for the entered state, and checking if a credit card has sufficient funds to allow the purchase, occurs on the server. In addition, AdventureWorks Shopper shows how the results of server-side validation can be returned to the client.

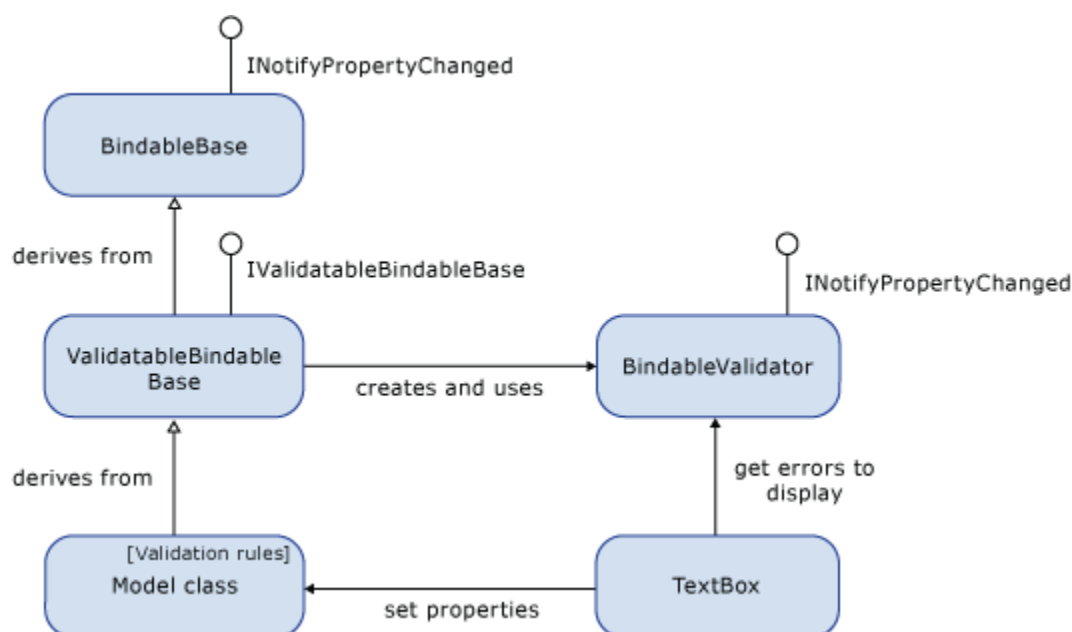
Model classes must derive from the **ValidatableBindableBase** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, in order to participate in validation. This class provides an error container (an instance of the **BindableValidator** class that is the type of the **Errors** property) whose contents are updated whenever a model class property value changes. The **BindableValidator** class and **ValidatableBindableBase** class derive from the **BindableBase** class, which raises property change notification events. For more info see [Triggering validation when properties change](#).

The **SetProperty** method in the **ValidatableBindableBase** class performs validation when a model property is set to a new value. The validation rules come from data annotation attributes that derive from the [ValidationAttribute](#) class. The attributes are taken from the declaration of the model property being validated. For more info see [Specifying validation rules](#) and [Triggering validation when properties change](#).

In the AdventureWorks Shopper reference implementation, users are notified about validation errors by highlighting the controls that contain the invalid data with red borders, and by displaying error messages that inform the user why the data is invalid below the controls containing invalid data.

If the app suspends while a data entry page is active, user input and any validation error messages are saved to disk, and restored when the app resumes following reactivation. Therefore, when the app suspends it will later resume as the user left it. For more info see [Highlighting validation errors with attached behaviors](#) and [Persisting user input and validation errors when the app suspends and resumes](#).

The following diagram shows the classes involved in performing validation in AdventureWorks Shopper.



Specifying validation rules

Validation rules are specified by adding data annotation attributes to properties in model classes that will require validation. To participate in validation a model class must derive from the **ValidatableBindableBase** class.

The data annotation attributes added to a model property whose data requires validation derive from the [ValidationAttribute](#) class. The following code example shows the **FirstName** property from the **Address** class.

C#: AdventureWorks.UILogic\Models\Address.cs

```
[Required(ErrorMessageResourceType = typeof(ErrorMessagesHelper),
    ErrorMessageResourceName = "RequiredErrorMessage")]
[RegularExpression(NAMES_REGEX_PATTERN, ErrorMessageResourceType =
    typeof(ErrorMessagesHelper), ErrorMessageResourceName = "RegexErrorMessage")]
public string FirstName
{
    get { return _firstName; }
    set { SetProperty(ref _firstName, value); }
}
```

The [Required](#) attribute of the **FirstName** property specifies that a validation failure occurs if the field is null, contains an empty string, or contains only white-space characters. The [RegularExpression](#) attribute specifies that the **FirstName** property must match the regular expression given by the **NAMES_REGEX_PATTERN** constant. This regular expression allows user input to consist of all unicode name characters as well as spaces and hyphens, as long as the spaces and hyphens don't occur in sequences and are not leading or trailing characters.

The static **ErrorMessagesHelper** class is used to retrieve validation error messages from the resource dictionary for the current locale, and is used by the [Required](#) and [RegularExpression](#) validation attributes. For example, the **Required** attribute on the **FirstName** property specifies that if the property doesn't contain a value, the validation error message will be that returned by the **RequiredErrorMessage** property of the **ErrorMessagesHelper** class.

In the AdventureWorks Shopper reference implementation, all of the validation rules that are specified on the client also appear on the server. Performing validation on the client helps users correct input mistakes without round trips to the server. Performing validation on the server prevents attackers from bypassing validation code in the client. Client validation occurs when each property changes. Server validation happens less frequently, usually when the user has finished entering all of the data on a page.

In AdventureWorks Shopper, additional validation rules exist on the server side, for example to validate zip codes and authorize credit card purchases. The following example shows how the AdventureWorks Shopper web service performs server-side validation of the zip code data entered by the user.

C#: AdventureWorks.WebServices\Models\Address.cs

```
[Required(ErrorMessageResourceType = typeof(Resources),
    ErrorMessageResourceName = "ErrorRequired")]
[RegularExpression(NUMBERS_REGEX_PATTERN, ErrorMessageResourceType =
    typeof(Resources), ErrorMessageResourceName = "ErrorRegex")]
[CustomValidation(typeof(Address), "ValidateZipCodeState")]
public string ZipCode { get; set; }
```

The [CustomValidation](#) attribute specifies an application-provided method that will be invoked to validate the property whenever a value is assigned to it. The validation method must be public and

static, and its first parameter must be the object to validate. The following code example shows the **ValidateZipCodeState** method that is used to validate the value of the **ZipCode** property on the server.

C#: AdventureWorks.WebServices\Models\Address.cs

```
public static ValidationResult ValidateZipCodeState(object value,
    ValidationContext validationContext)
{
    bool isValid = false;
    try
    {
        if (value == null)
        {
            throw new ArgumentNullException("value");
        }

        if (validationContext == null)
        {
            throw new ArgumentNullException("validationContext");
        }

        var address = (Address)validationContext.ObjectInstance;

        if (address.ZipCode.Length < 3)
        {
            return new ValidationResult(Resources.ErrorZipCodeInvalidLength);
        }

        string stateName = address.State;
        State state = new StateRepository().GetAll().FirstOrDefault(
            c => c.Name == stateName);
        int zipCode = Convert.ToInt32(address.ZipCode.Substring(0, 3),
            CultureInfo.InvariantCulture);

        foreach (var range in state.ValidZipCodeRanges)
        {
            // If the first 3 digits of the Zip Code falls within the given range,
            // it is valid.
            int minValue = Convert.ToInt32(range.Split('-')[0],
                CultureInfo.InvariantCulture);
            int maxValue = Convert.ToInt32(range.Split('-')[1],
                CultureInfo.InvariantCulture);

            isValid = zipCode >= minValue && zipCode <= maxValue;

            if (isValid) break;
        }
    }
    catch (ArgumentNullException)
    {
        isValid = false;
    }
}
```



```

    if (isValid)
    {
        return ValidationResult.Success;
    }
    else
    {
        return new ValidationResult(Resources.ErrorInvalidZipCodeInState);
    }
}

```

The method checks that the zip code value is within the allowable range for a given state. The [ValidationContext](#) method parameter provides additional contextual information that is used to determine the context in which the validation is performed. This parameter enables access to the **Address** object instance, from which the value of the **State** and **ZipCode** properties can be retrieved. The server's **StateRepository** class returns the zip code ranges for each state, and the value of the **ZipCode** property is then checked against the zip code range for the state. Finally, the validation result is returned as a [ValidationResult](#) object, in order to enable the method to return an error message if required. For more info about custom validation methods, see [CustomValidationAttribute.Method](#) property.

Note

Although it does not occur in the AdventureWorks Shopper reference implementation, property validation can sometimes involve dependent properties. An example of dependent properties occurs when the set of valid values for property A depends on the particular value that has been set in property B. If you want to check that the value of property A is one of the allowed values, you would first need to retrieve the value of property B. In addition, when the value of property B changes you would need to revalidate property A.

Validating dependent properties can be achieved by specifying a [CustomValidation](#) attribute and passing the value of property B in the [ValidationContext](#) method parameter. Custom validation logic in the model class could then validate the value of property A while taking the current value of property B into consideration.

Triggering validation when properties change

Validation is automatically triggered on the client whenever a bound property changes. For example, when a two way binding in a view sets a bound property in a model class, that class should invoke the **SetProperty** method. This method, provided by the **BindableBase** class, sets the property value and raises the [PropertyChanged](#) event. However, the **SetProperty** method is also overridden by the **ValidatableBindableBase** class. The **ValidatableBindableBase.SetProperty** method calls the **BindableBase.SetProperty** method, and performs validation if the property has changed. The following code example shows how validation happens after a property change.

C#: Microsoft.Practices.Prism.StoreApps\BindableValidator.cs

```

public bool ValidateProperty(string propertyName)
{
    if (string.IsNullOrEmpty(propertyName))
    {
        throw new ArgumentNullException("propertyName");
    }

    var propertyInfo = _entityToValidate.GetType()
        .GetRuntimeProperty(propertyName);

    if (propertyInfo == null)
    {
        var resourceLoader =
            new ResourceLoader(Constants.StoreAppsInfrastructureResourceMapId);
        var errorString =
            resourceLoader.GetString("InvalidPropertyNameException");

        throw new ArgumentException(errorString, propertyName);
    }

    var propertyErrors = new List<string>();
    bool isValid = TryValidateProperty(propertyInfo, propertyErrors);
    bool errorsChanged = SetPropertyErrors(propertyInfo.Name, propertyErrors);

    if (errorsChanged)
    {
        OnErrorsChanged(propertyName);
        OnPropertyChanged(string.Format(CultureInfo.CurrentCulture,
            "Item[{0}]", propertyName));
    }

    return isValid;
}

```

This method retrieves the property that is to be validated, and attempts to validate it by calling the **TryValidateProperty** method. If the validation results change, for example, when new validation errors are found or when previous errors have been corrected, then the **ErrorsChanged** and [PropertyChanged](#) events are raised for the property. The following code example shows the **TryValidateProperty** method.

C#: Microsoft.Practices.Prism.StoreApps\BindableValidator.cs

```

private bool TryValidateProperty(PropertyInfo propertyInfo,
    List<string> propertyErrors)
{
    var results = new List<ValidationResult>();
    var context = new ValidationContext(_entityToValidate)
        { MemberName = propertyInfo.Name };
    var propertyValue = propertyInfo.GetValue(_entityToValidate);

```

```
// Validate the property
bool isValid = Validator.TryValidateProperty(propertyValue, context, results);

if (results.Any())
{
    propertyErrors.AddRange(results.Select(c => c.ErrorMessage));
}

return isValid;
}
```

This method calls the **TryValidateProperty** method from the [Validator](#) class to validate the property value against the validation rules for the property. Any validation errors are added to a new list.

Triggering validation of all properties

Validation can also be triggered manually for all properties of a model object. For example, this occurs in AdventureWorks Shopper when the user selects the **Submit** button on the **CheckoutHubPage**. The button's command delegate calls the **ValidateForm** methods on the **ShippingAddressUserControlViewModel**, **BillingAddressUserControlViewModel**, and **PaymentMethodUserControlViewModel** classes. These methods call the **ValidateProperties** method of the **BindableValidator** class. The following code example shows the implementation of the **BindableValidator** class's **ValidateProperties** method.

C#: Microsoft.Practices.Prism.StoreApps\BindableValidator.cs

```
public bool ValidateProperties()
{
    var propertiesWithChangedErrors = new List<string>();

    // Get all the properties decorated with the ValidationAttribute attribute.
    var propertiesToValidate = _entityToValidate.GetType().GetRuntimeProperties()
        .Where(c => c.GetCustomAttributes(typeof(ValidationAttribute)).Any());

    foreach (PropertyInfo propertyInfo in propertiesToValidate)
    {
        var propertyErrors = new List<string>();
        TryValidateProperty(propertyInfo, propertyErrors);

        // If the errors have changed, save the property name to notify the update
        // at the end of this method.
        bool errorsChanged = SetPropertyErrors(propertyInfo.Name, propertyErrors);
        if (errorsChanged &&
            !propertiesWithChangedErrors.Contains(propertyInfo.Name))
        {
            propertiesWithChangedErrors.Add(propertyInfo.Name);
        }
    }

    // Notify each property whose set of errors has changed since the last
```

```
// validation.
foreach (string propertyName in propertiesWithChangedErrors)
{
    OnErrorsChanged(propertyName);
    OnPropertyChanged(string.Format(CultureInfo.CurrentCulture,
        "Item[{0}]", propertyName));
}

return _errors.Values.Count == 0;
}
```

This method retrieves any properties that have attributes that derive from the [ValidationAttribute](#) attribute, and attempts to validate them by calling the **TryValidateProperty** method for each property. If the validation state changes, the **ErrorsChanged** and [PropertyChanged](#) events are raised for each property whose errors have changed. Changes occur when new errors are seen or when previously detected errors are no longer present.

Triggering server-side validation

Server-side validation uses web service calls. For example, when the user selects the **Submit** button on the **CheckoutHubPage**, server-side validation is triggered by the **GoNext** method calling the **ProcessFormAsync** method, once client-side validation has succeeded. The following code example shows part of the **ProcessFormAsync** method.

C#: AdventureWorks.UILogic\ViewModels\CheckoutHubPageViewModel.cs

```
try
{
    // Create an order with the values entered in the form
    await _orderRepository.CreateBasicOrderAsync(user.UserName, shoppingCart,
        ShippingAddressViewModel.Address, BillingAddressViewModel.Address,
        PaymentMethodViewModel.PaymentMethod);

    _navigationService.Navigate("CheckoutSummary", null);
}
catch (ModelValidationException mvex)
{
    DisplayOrderErrorMessage(mvex.ValidationResult);
    if (_shippingAddressViewModel.Address.Errors.Count > 0)
        IsShippingAddressInvalid = true;
    if (_billingAddressViewModel.Address.Errors.Count > 0 &&
        !UseSameAddressAsShipping) IsBillingAddressInvalid = true;
    if (_paymentMethodViewModel.PaymentMethod.Errors.Count > 0)
        IsPaymentMethodInvalid = true;
}
```

This method calls the **CreateBasicOrderAsync** method on the **OrderRepository** instance to submit the created order to the web service. If the **CreateBasicOrderAsync** method successfully completes, then the data has been validated on the server.

The **CreateBasicOrderAsync** method uses the [HttpClientHandler](#) and [HttpClient](#) classes to send the order to the web service, and then calls the **EnsureSuccessWithValidationSupport** extension method to process the response from the web service. The following code example shows the **EnsureSuccessWithValidationSupport** method.

C#: AdventureWorks.UILogic\Services\HttpResponseMessageExtensions.cs

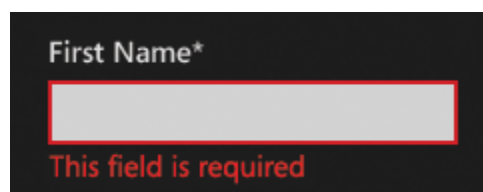
```
public static async Task EnsureSuccessWithValidationSupportAsync(this
HttpResponseMessage response)
{
    // If BadRequest, see if it contains a validation payload
    if (response.StatusCode == System.Net.HttpStatusCode.BadRequest)
    {
        ModelValidationResult result = null;
        try
        {
            result = await response.Content
                .ReadAsStringAsync<Models.ModelValidationResult>();
        }
        catch { } // Fall through logic will take care of it
        if (result != null) throw new ModelValidationException(result);
    }
    if (response.StatusCode == System.Net.HttpStatusCode.Unauthorized)
        throw new SecurityException();

    // Will throw for any other service call errors
    response.EnsureSuccessStatusCode();
}
```

If the response contains a [BadRequest](#) status code the **ModelValidationResult** is read from the response, and if the response isn't null a **ModelValidationException** is thrown, which indicates that server-side validation failed. This exception is caught by the **ProcessFormAsync** method, which will then call the **DisplayOrderErrorMessages** method to highlight the controls containing invalid data and display the validation error messages.

Highlighting validation errors with attached behaviors

In the AdventureWorks Shopper reference implementation, client-side validation errors are shown to the user by highlighting the control that contains invalid data, and by displaying an error message beneath the control, as shown in the following diagram.



The **HighlightFormFieldOnErrors** and **HighlightFormComboOnErrors** attached behaviors are used to highlight **FormFieldTextBox** and [ComboBox](#) controls when validation errors occur. The following

code example shows how the **HighlightFormFieldOnErrors** behavior is attached to a **FormFieldTextBox** control.

XAML: AdventureWorks.Shopper\Views\ShippingAddressUserControl.xaml

```
<controls:FormFieldTextBox x:Name="FirstName"
    x:Uid="FirstNameTextBox"
    AutomationProperties.AutomationId="FirstNameTextBox"
    AutomationProperties.LabeledBy=
        "{Binding ElementName=TitleFirstName}"
    Style="{StaticResource FormFieldStyle}"
    Grid.Row="1"
    Grid.Column="0"
    AutomationProperties.IsRequiredForForm="True"
    Text="{Binding Address.FirstName, Mode=TwoWay}"
    behaviors:HighlightFormFieldOnErrors.PropertyErrors=
        "{Binding Address.Errors[FirstName]}" />
```

The attached behavior gets and sets the **PropertyErrors** dependency property. The following code example shows how the **PropertyErrors** dependency property is defined in the **HighlightFormFieldOnErrors** class.

C#: AdventureWorks.Shopper\Behaviors\HighlightFormFieldOnErrors.cs

```
public static DependencyProperty PropertyErrorsProperty =
    DependencyProperty.RegisterAttached("PropertyErrors",
        typeof(ReadOnlyCollection<string>), typeof(HighlightFormFieldOnErrors),
        new PropertyMetadata(BindableValidator.EmptyErrorsCollection,
            OnPropertyErrorsChanged));
```

The **PropertyErrors** dependency property is registered as a [ReadOnlyCollection](#) of strings, by the [RegisterAttached](#) method. When the value of the **PropertyErrors** dependency property changes, the **OnPropertyErrorsChanged** method is invoked to change the highlighting style of the input control. The following code example shows the **OnPropertyErrorsChanged** method.

C#: AdventureWorks.Shopper\Behaviors\HighlightFormFieldOnErrors.cs

```
private static void OnPropertyErrorsChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs args)
{
    if (args == null || args.NewValue == null)
    {
        return;
    }

    var control = (FrameworkElement)d;
    var propertyErrors = (ReadOnlyCollection<string>)args.NewValue;

    Style style = (propertyErrors.Any()) ? (Style)Application.Current
        .Resources["HighlightFormFieldStyle"] : (Style)Application.Current
        .Resources["FormFieldStyle"];
    control.Style = style;
}
```

The **OnPropertyErrorsChanged** method's parameters give the instance of the **FormFieldTextBox** that the **PropertyErrors** dependency property is attached to, and any validation errors for the **FormFieldTextBox**. Then, if validation errors are present the **HighlightFormFieldStyle** is applied to the **FormFieldTextBox**, so that it is highlighted with a red [BorderBrush](#).

The UI also displays validation error messages in [TextBlock](#)s below each control whose data failed validation. The following code example shows the **TextBlock** that displays a validation error message if the user has entered an invalid first name for their shipping details.

XAML: AdventureWorks.Shopper\Views\ShippingAddressUserControl.xaml

```
<TextBlock x:Name="ErrorsFirstName"
    Style="{StaticResource ErrorMessageStyle}"
    Grid.Row="2"
    Grid.Column="0"
    Text="{Binding Address.Errors[FirstName],
        Converter={StaticResource FirstErrorConverter}}"
    TextWrapping="Wrap" />
```

Each [TextBlock](#) binds to the **Errors** property of the model object whose properties are being validated. The **Errors** property is provided by the **ValidateableBindableBase** class, and is an instance of the **BindableValidator** class. The indexer of the **BindableValidator** class returns a [ReadOnlyCollection](#) of error strings, with the **FirstErrorConverter** retrieving the first error from the collection, for display.

Persisting user input and validation errors when the app suspends and resumes

Windows Store apps should be designed to suspend when the user switches away from them and resume when the user switches back to them. Suspended apps that are terminated by the operating system and subsequently reactivated should resume in the state that the user left them rather than starting afresh. This has an impact on validation in that if an app suspends on a data entry page, any user input and validation error messages should be saved. Then, on reactivation the user input and validation error messages should be restored to the page. For more info see [Guidelines for app suspend and resume](#).

AdventureWorks Shopper accomplishes this task by using overridden **OnNavigatedFrom** and **OnNavigatedTo** methods in the view model class for the page. The **OnNavigatedFrom** method allows the view model to save any state before it is disposed of prior to suspension. The **OnNavigatedTo** method allows a newly displayed page to initialize itself by loading any view model state when the app resumes.

All of the view model classes derive from the **ViewModel** base class, which implements **OnNavigatedFrom** and **OnNavigatedTo** methods that save and restore view model state, respectively. This avoids each view model class having to implement this functionality to support the suspend and resume process. However, the **OnNavigatedFrom** and **OnNavigatedTo** methods can be

overridden in the view model class for the page if any additional navigation logic is required, such as adding the validation errors collection to the view state dictionary. The following code example shows how the **OnNavigatedFrom** method in the **BillingAddressUserControlViewModel** class adds any billing address validation errors to the session state dictionary that will be serialized to disk by the **SessionStateService** class when the app suspends.

C#: AdventureWorks.UILogic\ViewModels\BillingAddressUserControlViewModel.cs

```
public override void OnNavigatedFrom(Dictionary<string, object> viewState,
    bool suspending)
{
    base.OnNavigatedFrom(viewState, suspending);

    // Store the errors collection manually
    if (viewState != null)
    {
        AddEntityStateValue("errorsCollection", _address.GetAllErrors(),
            viewState);
    }
}
```

This method ensures that when the app suspends, the **BillingAddressUserControlViewModel** state and any billing address validation error messages will be serialized to disk. View model properties that have the **RestorableState** attribute will be added to the session state dictionary by the **ViewModel.OnNavigatedFrom** method before the **ViewModel.AddEntityStateValue** method adds the validation error message collection to the session state dictionary. The **GetAllErrors** method is implemented by the **ValidatableBindableBase** class, which in turn calls the **GetAllErrors** method of the **BindableValidator** class to return the validation error messages for the **Address** model instance.

When the app is reactivated after termination and page navigation is complete, the **OnNavigatedTo** method in the active view model class will be called. The following code example shows how the **OnNavigatedTo** method in the **BillingAddressUserControlViewModel** restores any billing address validation errors from the session state dictionary.

C#: AdventureWorks.UILogic\ViewModels\BillingAddressUserControlViewModel.cs

```
public override async void OnNavigatedTo(object navigationParameter,
    NavigationMode navigationMode, Dictionary<string, object> viewState)
{
    // The States collection needs to be populated before setting the State
    // property
    await PopulateStatesAsync();

    if (viewState != null)
    {
        base.OnNavigatedTo(navigationParameter, navigationMode, viewState);

        if (navigationMode == NavigationMode.Refresh)
        {
            // Restore the errors collection manually
        }
    }
}
```



```

        var errorsCollection = RetrieveEntityStateValue<IDictionary<string,
            ReadOnlyCollection<string>>>("errorsCollection", viewState);

        if (errorsCollection != null)
        {
            _address.SetAllErrors(errorsCollection);
        }
    }
}
if (navigationMode == NavigationMode.New)
{
    var addressId = navigationParameter as string;
    if (addressId != null)
    {
        Address =
            await _checkoutDataRepository.GetBillingAddressAsync(addressId);
        return;
    }

    if (_loadDefault)
    {
        var defaultAddress =
            await _checkoutDataRepository.GetDefaultBillingAddressAsync();
        if (defaultAddress != null)
        {
            // Update the information and validate the values
            Address.FirstName = defaultAddress.FirstName;
            Address.MiddleInitial = defaultAddress.MiddleInitial;
            Address.LastName = defaultAddress.LastName;
            Address.StreetAddress = defaultAddress.StreetAddress;
            Address.OptionalAddress = defaultAddress.OptionalAddress;
            Address.City = defaultAddress.City;
            Address.State = defaultAddress.State;
            Address.ZipCode = defaultAddress.ZipCode;
            Address.Phone = defaultAddress.Phone;
        }
    }
}
}

```

This method ensures that when the app is reactivated following termination, the **BillingAddressUserControlViewModel** state and any billing address validation error messages will be restored from disk. View model properties that have the **RestorableState** attribute will be restored from the session state dictionary by the **ViewModel.OnNavigatedTo** method, before the **ViewModel.RetrieveEntityStateValue** method retrieves any validation error messages. The **SetAllErrors** method is implemented by the **ValidatableBindableBase** class, which in turn calls the **SetAllErrors** method of the **BindableValidator** class to set the validation error messages for the **Address** model instance. Then, provided that the navigation is to a new instance of a page, the billing address is retrieved.

For more info see [Creating and navigating between pages](#) and [Handling suspend, resume, and activation](#).

Managing application data in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Use the application data APIs to work with application data, making the system responsible for managing the physical storage of data.
- Only store passwords in the credential locker if the user has successfully signed into the app and has opted to save passwords.
- Use ASP.NET Web API to create a resource-oriented web service that can pass different content types.

Application data is data that is specific to a Windows Store app and includes runtime state, user preferences, and other settings. Application data is created, read, updated, deleted, and cached when the app is running. This article examines how the AdventureWorks Shopper reference implementation manages its application data including storing passwords in the credential locker, authenticating users, and retrieving data from a web service while minimizing the network traffic and battery life of the device. AdventureWorks Shopper uses [Prism for the Windows Runtime](#) to customize the default Settings pane shown in the Settings charm.

You will learn

- How to store data in the app data stores.
- How to store passwords in the credential locker.
- How to use the Settings charm to allow users to change app settings.
- How to use model objects as data transfer objects.
- How to perform credentials-based authentication between a Windows Store app and a web service.
- How to reliably retrieve data from a web service.
- How to cache data from a web service on disk.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Application data is data that the app itself creates and manages. It is specific to the internal functions or configuration of an app, and includes runtime state, user preferences, reference content, and other settings. App data is tied to the existence of the app and is only meaningful to that app. The following list summarizes the decisions to make when managing application data in your app:

- Where and how should I store application data?

- What type of data should I store as application data?
- Do I need to provide a privacy policy for my app, and if so, where should it be displayed to users?
- How many entries should I include in the Settings charm?
- What data should be allowed to roam?
- How should I implement a web service that a Windows Store app will connect to?
- How should I authenticate users with a web service in a Windows Store app?
- Should I cache data from the web service locally?

Windows Store apps should use app data stores for settings and files that are specific to each app and user. The system manages the data stores for an app, ensuring that they are kept isolated from other apps and users. In addition, the system preserves the contents of these data stores when the user installs an update to your app and removes the contents of these data stores completely and cleanly when your app is uninstalled.

Application data should not be used to store user data or anything that users might perceive as valuable and irreplaceable. The user's libraries and Microsoft SkyDrive should be used to store this sort of information. Application data is ideal for storing app-specific user preferences, settings, reference data, and favorites. For more info see [App data](#).

If your app uses or enables access to any Internet-based services, or collects or transmits any user's personal information, you must maintain a privacy policy. You are responsible for informing users of your privacy policy. The policy must comply with applicable laws and regulations, inform users of the information collected by your app and how that information is used, stored, secured, and disclosed, describe the controls that users have over the use and sharing of their information, and how they may access their information. You must provide access to your privacy policy in the app's settings as displayed in the Settings charm. If you submit your app to the Windows Store you must also provide access to your privacy policy in the **Description** page of your app on the Windows Store. For more info see [Windows 8 app certification requirements](#).

The top part of the Settings pane lists entry points for your app settings, with each entry point performing an action such as opening a Flyout, or opening an external link. Similar or related options should be grouped together under one entry point in order to avoid adding more than four entry points. For more info see [Guidelines for app settings](#).

Utilizing roaming application data in app is easy and does not require significant code changes. It is best to utilize roaming application data for all size-bound data and settings that are used to preserve a user's settings preferences. For more info see [Guidelines for roaming application data](#).

There are a number of approaches for implementing a web service that a Windows Store app can connect to:

- Windows Azure Mobile Services allow you to add a cloud-based service to your Windows Store app. For more info see [Windows Azure Mobile Services Dev Center](#).

- Windows Communication Foundation (WCF) enables you to develop web services based on SOAP. These services focus on separating the service from the transport protocol. Therefore, you can expose the same service using different endpoints and different protocols such as TCP, User Datagram Protocol (UDP), HTTP, Secure Hypertext Transfer Protocol (HTTPS), and Message Queuing. However, this flexibility comes at the expense of the extensive use of configuration and attributes, and the resulting infrastructure is not always easily testable. In addition, new client proxies need to be generated whenever the input or output model for the service changes.
- The ASP.NET Web API allows you to develop web services that are exposed directly over HTTP, thus enabling you to fully harness HTTP as an application layer protocol. Web services can then communicate with a broad set of clients whether they are apps, browsers, or back-end services. The ASP.NET Web API is designed to support apps built with REST, but it does not force apps to use a RESTful architecture. Therefore, if the input or output model for the service changes, the client simply has to change the query string that is sent to the web service, or parse the data received from the web service differently.

The primary difference between WCF and the ASP.NET Web API is that while WCF is based on SOAP, the ASP.NET Web API is based on HTTP. HTTP offers the following advantages:

- It supports verbs that define actions. For example, you query information using GET, and create information using POST.
- It contains message headers that are meaningful and descriptive. For example, the headers suggest the content type of the message's body.
- It contains a body that can be used for any type of content, not just XML content as SOAP enforces. The body of HTTP messages can be anything you want including HTML, XML, JavaScript Object Notation (JSON), and binary files.
- It uses Uniform Resource Identifiers (URIs) to identify both resources and actions.

The decision of whether to use WCF or the ASP.NET Web API in your app can be made by answering the following the following questions:

- Do you want to create a service that supports special scenarios such as one-way messaging, message queues, and duplex communication? If so you should use WCF.
- Do you want to create a service that uses fast transport channels when available, such as TCP, named pipes, or UDP? If so you should use WCF.
- Do you want to create a service that uses fast transport channels when available, but uses HTTP when all other transport channels are unavailable? If so you should use WCF.
- Do you want to simply serialize objects and deserialize them as the same strongly-typed objects at the other side of the transmission? If so you should use WCF.
- Do you need to use a protocol other than HTTP? If so you should use WCF.
- Do you want to create a resource-oriented service that is activated through simple action-oriented verbs such as GET, and that responds by sending content as HTML, XML, a JSON string, or binary data? If so you should use the ASP.NET Web API.
- Do you have bandwidth constraints? If so you should use the ASP.NET Web API with JSON, as it sends a smaller payload than SOAP.

- Do you need to support clients that don't have a SOAP stack? If so you should use the ASP.NET Web API.

There are a number of approaches that could be taken to authenticate users of a Windows Store app with a web service. For instance, credentials-based authentication or single sign-on with a Microsoft account could be used. A user can link a local Windows 8 account with his or her Microsoft account. Then, when the user signs in to a device using that Microsoft account, any Windows Store app that supports Microsoft account sign-in can automatically detect that the user is already authenticated and the app doesn't require the user to sign in app. The advantage of this approach over credential roaming is that the Microsoft account works for websites and apps, meaning that app developers don't have to create their own authentication system. Alternatively, apps could use the web authentication broker instead. This allows apps to use internet authentication and authorization protocols like Open Identification (OpenID) or Open Authentication (OAuth) to connect to online identity providers. This isolates the user's credentials from the app, as the broker is the facilitator that communicates with the app. For more info see [Managing user info](#).

Local caching of web service data should be used if you repeatedly access static data or data that rarely changes, or when data access is expensive in terms of creation, access, or transportation. This brings many benefits including improving app performance by storing relevant data as close as possible to the data consumer, and saving network and battery resources.

Managing application data in AdventureWorks Shopper

The AdventureWorks Shopper reference implementation uses app data stores to store the user's credentials and cached data from the web service. The user's credentials are roamed. For more info see [Storing data in the app data stores](#) and [Roaming application data](#).

AdventureWorks Shopper provides access to its privacy policy in the app's settings as displayed in the Settings charm. The privacy policy is one of several entry points in the Settings charm, and informs users of the personal information that is transmitted, how that information is used, stored, secured, and disclosed. It describes the controls that users have over the use and sharing of their information and how they may access their information. For more info see [Local application data](#).

AdventureWorks Shopper uses the ASP.NET Web API to implement its web service, and performs credentials-based authentication with this web service. This approach creates a web service that can communicate with a broad set of clients including apps, browsers, or back-end services. Product data from the web service is cached locally in the temporary app data store. For more info see [Accessing data through a web service](#) and [Caching data](#).

Storing data in the app data stores

When an app is installed, the system gives it its own per-user data stores for application data such as settings and files. The lifetime of application data is tied to the lifetime of the app. If the app is removed, all of the application data will be lost.

There are three data stores for application data:

- The *local* data store is used for persistent data that exists only on the device.
- The *roaming* data store is used for data that exists on all trusted devices on which the user has installed the app.
- The *temporary* data store is used for data that could be removed by the system at any time.

You use the application data API to work with application data with the system being responsible for managing its physical storage.

Settings in the app data store are stored in the registry. When you use the application data API, registry access is transparent. Within its app data store each app has a root container for settings. Your app can add settings and new containers to the root container.

Files in the app data store are stored in the file system. Within its app data store, each app has system-defined root directories—one for local files, one for roaming files, and one for temporary files. Your app can add new files and new directories to the root directory.

App settings and files can be local or roaming. The settings and files that your app adds to the local data store are only present on the local device. The system automatically synchronizes settings and files that your app adds to the roaming data store on all trusted devices on which the user has installed the app.

For more info see [Accessing app data with the Windows Runtime](#).

Local application data

Local application data should be used to store data that needs to be preserved between application sessions, and it is not suitable type or size wise for roaming data. There is no size restriction on local data.

In the AdventureWorks Shopper reference implementation only the **SessionStateService** class stores data in the local application data store. For more info see [Handling suspend, resume, and activation](#).

For more info see [Quickstart: Local application data](#).

Roaming application data

If you use roaming data in your app, and a user installs your app on multiple devices, Windows keeps the application data in sync. Windows replicates roaming data to the cloud when it is updated and synchronizes the data to the other trusted devices on which the app is installed. This provides a desirable user experience, since the app on different devices is automatically configured according to the user preferences on the first device. Any future changes to the settings and preferences will also transition automatically. Windows 8 can also transition session or state information. This enables users to continue to use an app session that was abandoned on one device when they transfer to a second device.

Roaming data should be used for all size-bound data and settings that are used to preserve a user's settings preferences as well as app session state. Any data that is only meaningful on a specific device, such as the path to a local file, should not be roamed.

Each app has a quota for roaming application data that is defined by the [ApplicationData.RoamingStorageQuota](#) property. If your roaming data exceeds the quota it won't roam until its size is less than the quota again. In AdventureWorks Shopper, we wanted to use roaming data to transfer partially completed shopping cart data to other devices when the initial device is abandoned. However, this was not feasible due to the enforced quota. Instead, this functionality is provided by the web service that the AdventureWorks Shopper reference implementation connects to. The data that roams in AdventureWorks Shopper are the user's credentials.

Note Roaming data for an app is available in the cloud as long as it is accessed by the user from some device within 30 days. If the user does not run an app for longer than 30 days, its roaming data is removed from the cloud. If the user uninstalls an app, its roaming data isn't automatically removed from the cloud. If the user reinstalls the app within 30 days, the roaming data is synchronized from the cloud.

Windows 8 roams app data opportunistically and so an instant sync is not guaranteed. For time critical settings a special high priority settings unit is available that provides more frequent updates. It is limited to one specific setting that must be named "HighPriority." It can be a composite setting, but the total size is limited to 8KB. This limit is not enforced and the setting will be treated as a regular setting, meaning that it will be roamed under regular priority, in case the limit is exceeded. However, if you are using a high latency network, roaming could still be significantly delayed.

For more info see [Guidelines for roaming application data](#).

Storing and roaming user credentials

Apps can store the user's password in the credential locker by using the [Windows.Security.Credentials](#) namespace. The credential locker provides a common approach for storing and managing passwords in a protected store. However, passwords should only be saved in the credential locker if the user has successfully signed in and opted to save passwords.

Note The credential locker should only be used for storing passwords and not for other items of data.

A credential in the credential locker is associated with a specific app or service. Apps and services do not have access to credentials associated with other apps or services. The credential locker from one trusted device is automatically transferred to any other trusted device for that user. This means that credential roaming is enabled by default for credentials stored in the credential locker on non-domain joined devices. Credentials from local connected accounts on domain-joined computers can roam. However, domain-connected accounts are subject to roaming restrictions if the credentials have only been saved on the domain-joined device.

You can enable credential roaming by connecting your device to the cloud by using your Microsoft account. This allows your credentials to roam to all of your trusted devices whenever you sign in with a Microsoft account.

Note Data stored in the credential locker will only roam if a user has made a device trusted.

The **ICredentialStore** interface, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, defines method signatures for loading and saving credentials. The following code example shows this interface.

C#: Microsoft.Practices.Prism.StoreApps\ICredentialStore.cs

```
public interface ICredentialStore
{
    void SaveCredentials(string resource, string userName, string password);
    PasswordCredential GetSavedCredentials(string resource);
    void RemoveSavedCredentials(string resource);
}
```

This interface is implemented by the **RoamingCredentialStore** class in the AdventureWorks.UILogic project.

The user is invited to enter their credentials on the sign in Flyout, which can be invoked from the Settings charm, or on the sign in dialog. When the user selects the **Submit** button on the **SignInFlyOut** view, the **SignInCommand** in the **SignInFlyOutViewModel** class is executed, which in turn calls the **SignInAsync** method. This method then calls the **SignInUserAsync** method on the **AccountService** instance, which in turn calls the **LogOnAsync** method on the **IdentityServiceProxy** instance. The instance of the **AccountService** class is created by the [Unity](#) dependency injection container. Then, provided that the credentials are valid and the user has opted to save the credentials, they are stored in the credential locker by calling the **SaveCredentials** method in the **RoamingCredentialStore** instance. The following code example shows how the **RoamingCredentialStore** class implements the **SaveCredentials** method to save the credentials in the credential locker.

C#: AdventureWorks.UILogic\Services\RoamingCredentialStore.cs

```
public void SaveCredentials(string resource, string userName, string password)
{
    var vault = new PasswordVault();

    RemoveAllCredentialsByResource(resource, vault);

    // Add the new credential
    var passwordCredential = new PasswordCredential(resource, userName, password);
    vault.Add(passwordCredential);
}
```


The **SaveCredentials** method creates a new instance of the **PasswordVault** class that represents a credential locker of credentials. The old stored credentials for the app are retrieved and removed before the new credentials are added to the credential locker.

For more info see [Credential Locker Overview](#) and [Storing user credentials](#).

Temporary application data

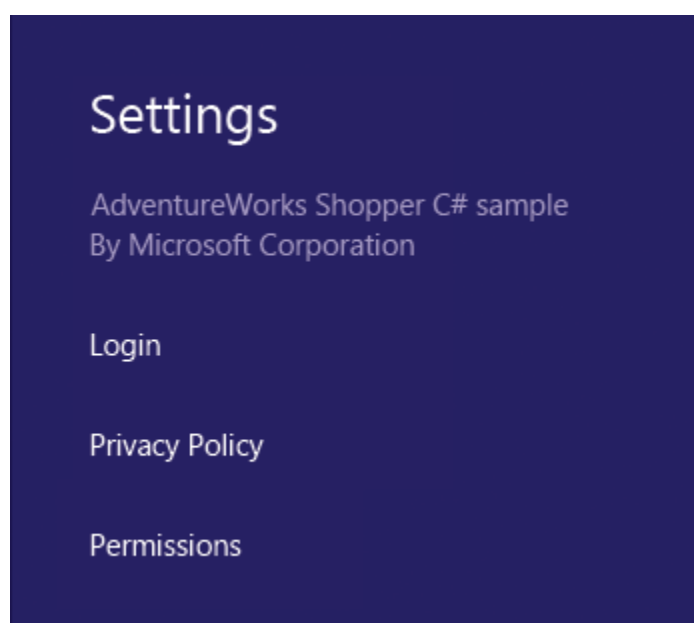
Temporary application data should be using for storing temporary information during an application session. The temporary data store works like a cache and its files do not roam. The System Maintenance task can automatically delete data at this location at any time, and the user could also clear files from the temporary data store using Disk Cleanup.

For more info about how AdventureWorks Shopper uses the temporary app data store see [Caching data](#).

Exposing settings through the Settings charm

The Settings charm is a fundamental part of any Windows Store app, and is used to expose app settings. It is invoked by making a horizontal edge gesture, swiping left with a finger or stylus from the right of the screen. This displays the charms and you can then select the Settings charm to display the Settings pane. The Settings pane includes both app and system settings.

The top part of the Settings pane lists entry points for your app settings. Each entry point opens a settings Flyout that displays the settings themselves. Entry points let you create categories of settings, grouping related controls together. Windows provides the **Permissions** and **Rate and review** entry points for apps that have been installed through the Windows Store. Side-loaded apps do not have the **Rate and review** entry point. The following diagram shows the top part of the default Settings pane for AdventureWorks Shopper.



Additional app settings are shown when a user is logged into the app. The bottom part of the Settings pane includes device settings provided by the system, such as volume, brightness, and power.

In order to customize the default Settings pane you can add a [SettingsCommand](#) that represents a settings entry. In the AdventureWorks Shopper reference implementation this is performed by the **MvvmAppBase** class in the [Microsoft.Practices.Prism.StoreApps](#) library. The **InitializeFrameAsync** method in the **MvvmAppBase** class subscribes to the [CommandsRequested](#) event of the [SettingsPane](#) class that is raised when the user opens the Settings pane. This is shown in the following code example.

C#: AdventureWorks.Shopper\App.xaml.cs

```
SettingsPane.GetForCurrentView().CommandsRequested += OnCommandsRequested;
```

When the event is raised the **OnCommandsRequested** event handler in the **MvvmAppBase** class creates a [SettingsCommand](#) collection, as shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```
private void OnCommandsRequested(SettingsPane sender,
SettingsPaneCommandsRequestedEventArgs args)
{
    if (args == null || args.Request == null ||
        args.Request.ApplicationCommands == null)
    {
        return;
    }

    var applicationCommands = args.Request.ApplicationCommands;
    var settingsCharmActionItems = GetSettingsCharmActionItems();

    foreach (var item in settingsCharmActionItems)
    {
        var settingsCommand = new SettingsCommand(item.Id, item.Title,
            (o) => item.Action.Invoke());
        applicationCommands.Add(settingsCommand);
    }
}
```

This method creates a [SettingsCommand](#) for each **SettingsCharmActionItem** and adds each **SettingsCommand** to the [ApplicationCommands](#). All the **SettingsCommands** will be shown on the Settings pane before the **Permissions** entry point. The **SettingsCharmActionItem** class is provided by the [Microsoft.Practices.Prism.StoreApps](#) library.

The **SettingsCharmActionItems** for the app are defined by the **GetSettingsCharmActionItems** override in the **App** class, as shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```
protected override IList<SettingsCharmActionItem> GetSettingsCharmActionItems()
{
    var settingsCharmItems = new List<SettingsCharmActionItem>();
    var accountService = _container.Resolve<IAccountService>();
    var resourceLoader = _container.Resolve<IResourceLoader>();

    if (accountService.SignedInUser == null)
    {
        settingsCharmItems.Add(
            new SettingsCharmActionItem(resourceLoader.GetString("LoginText"),
                () => FlyoutService.ShowFlyout("SignIn")));
    }
    else
    {
        settingsCharmItems.Add(new SettingsCharmActionItem(resourceLoader
            .GetString("LogoutText"), () => FlyoutService.ShowFlyout("SignOut")));
        settingsCharmItems.Add(new SettingsCharmActionItem(resourceLoader
            .GetString("AddShippingAddressTitle"),
                () => NavigationService.Navigate("ShippingAddress", null)));
        settingsCharmItems.Add(new SettingsCharmActionItem(resourceLoader
            .GetString("AddBillingAddressTitle"),
                () => NavigationService.Navigate("BillingAddress", null)));
        settingsCharmItems.Add(new SettingsCharmActionItem(resourceLoader
            .GetString("AddPaymentMethodTitle"),
                () => NavigationService.Navigate("PaymentMethod", null)));
        settingsCharmItems.Add(new SettingsCharmActionItem(resourceLoader
            .GetString("ChangeDefaults"),
                () => FlyoutService.ShowFlyout("ChangeDefaults")));
    }
    settingsCharmItems.Add(new SettingsCharmActionItem(resourceLoader
        .GetString("PrivacyPolicy"),
            async () => await Launcher.LaunchUriAsync(
                new Uri(resourceLoader.GetString("PrivacyPolicyUrl"))));

    return settingsCharmItems;
}
```

Each **SettingsCharmActionItem** allows one of three possible actions to occur—a Flyout to be shown, in-app navigation to take place, or an external hyperlink to be launched.

When a user selects the **Login** entry point, the **SignInFlyout** must be displayed. This Flyout class derives from the **FlyoutView** class in the [Microsoft.Practices.Prism.StoreApps](#) library. The **FlyoutView** class uses a **Popup** control to display the Flyout. The **Popup** control provides the light dismiss behavior that's seen throughout Windows 8. Therefore, when the user selects a UI element that is not part of the Flyout, the Flyout automatically dismisses itself.

The **FlyoutView** class subscribes to the **Activated** event of the **Window** class in the **Open** method to ensure that when the **Window** is deactivated, the **Popup** is dismissed correctly. The following code

example shows the **OnPopupClosed** and **OnWindowActivated** event handlers in the **FlyOutView** class.

C#: Microsoft.Practices.Prism.StoreApps\Flyouts\FlyOutView.cs

```
private void OnPopupClosed(object sender, object e)
{
    _popup.Child = null;
    Window.Current.Activated -= OnWindowActivated;
    if (_wasSearchOnKeyboardInputEnabled)
    {
        SearchPane.GetForCurrentView().ShowOnKeyboardInput = true;
    }
}

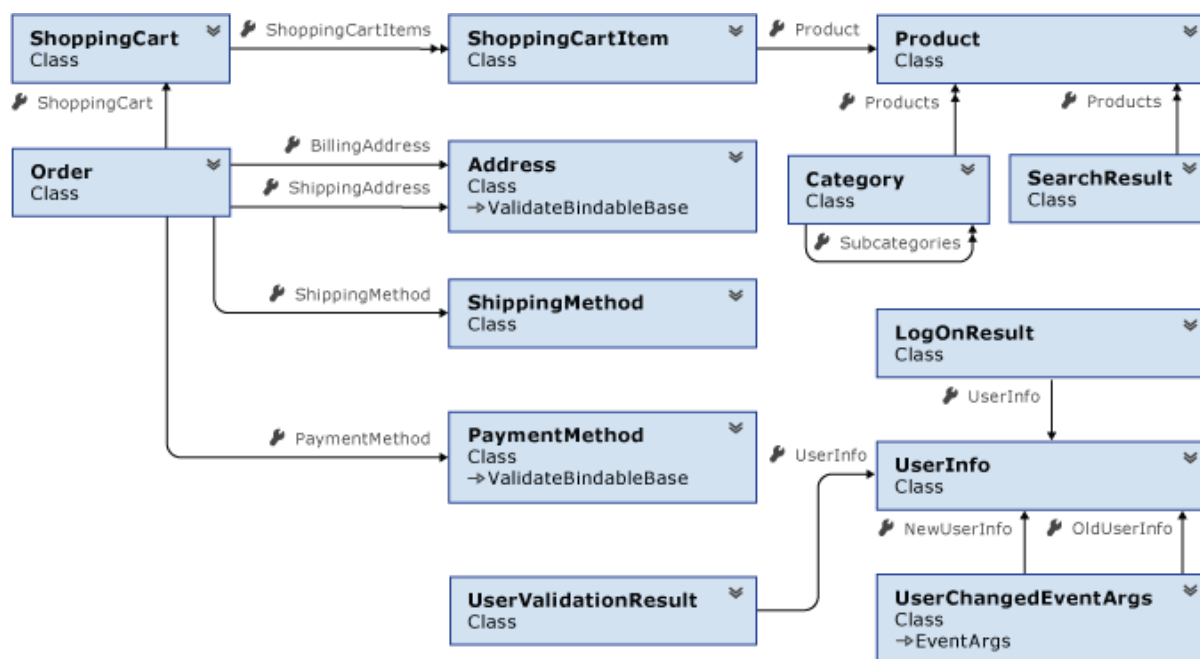
private void OnWindowActivated(object sender,
Windows.UI.Core.WindowActivatedEventArgs e)
{
    if (e.WindowActivationState == Windows.UI.Core.CoreWindowActivationState
        .Deactivated)
    {
        Close();
    }
}
```

The **OnWindowActivated** event handler calls the **Close** method, to close the [Popup](#) control if the [Window](#) is deactivated. In turn, this calls the **OnPopupClosed** event handler which removes the subscription to the [Activated](#) event in order to prevent a memory leak, and enables type to search functionality. For more info about type to search functionality, see [Implementing search](#).

For more info see [Guidelines for app settings](#).

Using model classes as data transfer objects

[Using the Model-View-ViewModel pattern](#) describes the Model-View-ViewModel (MVVM) pattern used in AdventureWorks Shopper. The model elements of the pattern are contained in the AdventureWorks.UILogic and AdventureWorks.WebServices projects, which represent the domain entities used in the app. The following diagram shows the key model classes in the AdventureWorks.UILogic project, and the relationships between them.



The repository and controller classes in the AdventureWorks.WebServices project accept and return the majority of these model objects. Therefore, they are used as data transfer objects (DTOs) that hold all the data that is passed between the app and the web service. A DTO is a container for a set of aggregated data that needs to be transferred across a network boundary. DTOs should contain no business logic and limit their behavior to activities such as validation.

The benefits of using DTOs to pass data to and receive data from a web service are that:

- By transmitting more data in a single remote call, the app can reduce the number of remote calls. In most scenarios, a remote call carrying a larger amount of data takes virtually the same time as a call that carries only a small amount of data.
- Passing more data in a single remote call more effectively hides the internals of the web service behind a coarse-grained interface.
- Defining a DTO can help in the discovery of meaningful business objects. When creating DTOs, you often notice groupings of elements that are presented to a user as a cohesive set of information. Often these groups serve as useful prototypes for objects that describe the business domain that the app deals with.
- Encapsulating data into a serializable object can improve testability.

Accessing data through a web service

Web services extend the World Wide Web infrastructure to provide the means for software to connect to other software apps. Apps access web services via ubiquitous web protocols and data formats such as HTTP, XML, SOAP, with no need to worry about how the web service is implemented.

Connecting to a web service from a Windows Store app introduces a set of development challenges:

- The app must minimize the use of network bandwidth.
- The app must minimize its impact on the device's battery life.
- The web service must offer an appropriate level of security.
- The web service must be easy to develop against.
- The web service should potentially support a range of client platforms.

These challenges will be addressed in the following sections.

Consumption

The AdventureWorks Shopper reference implementation stores data in an in-memory database that's accessed through a web service. The app must be able to send data to and receive data from the web service. For example, it must be able to retrieve product data in order to display it to the user, and it must be able to retrieve and send billing data and shopping cart data.

Users may be using AdventureWorks Shopper in a limited bandwidth environment, and so the developers wanted to limit the amount of bandwidth used to transfer data between the app and the web service. In addition to this, the developers wanted to ensure that the data transfer is reliable. Ensuring that data reliably downloads from the web service is important in ensuring a good user experience and hence maximizing the number of potential orders that will be made. Ensuring that shopping cart data reliably uploads to the web service is important in order to maximize actual orders, and their correctness.

The developers also wanted a solution that was simple to implement, and that could be easily customized in the future if, for example, authentication requirements were to change. In addition, the developers wanted a solution that could potentially work with platforms other than Windows 8.

With these requirements in mind, the AdventureWorks Shopper team had to consider three separate aspects of the solution: how to expose data from the web service, the format of the data that moves between the web service and the app, and how to consume web service data in the app.

Exposing data

The AdventureWorks Shopper reference implementation uses the ASP.NET Web API to implement its web service, and performs credentials-based authentication with this web service. This approach creates a resource-oriented web service that is activated through simple action-oriented verbs such as GET, and that can respond by sending content in a variety of formats such as HTML, XML, a JSON string, or binary data. The web service can communicate with a broad set of clients including apps, browsers, or back-end services. In addition, it offers the advantage that if the input or output model for the service changes in future, the app simply has to change the query string that is sent to the web service, or parse the data received from the web service differently.

Data formats

The AdventureWorks Shopper reference implementation uses the JSON format to transfer order data to the web service, and to cache web service data locally on disk, because it produces a compact payload that reduces bandwidth requirements and is relatively easy to use.

The AdventureWorks developers considered compressing data before transferring it to the web service in order to reduce bandwidth utilization, but decided that the additional CPU and battery usage on devices would outweigh the benefits. You should evaluate this tradeoff between the cost of bandwidth and battery consumption in your app before you decide whether to compress data you need to move over the network.

Note Additional CPU usage affects both the responsiveness of the device and its battery life.

For more info about caching see [Caching data](#).

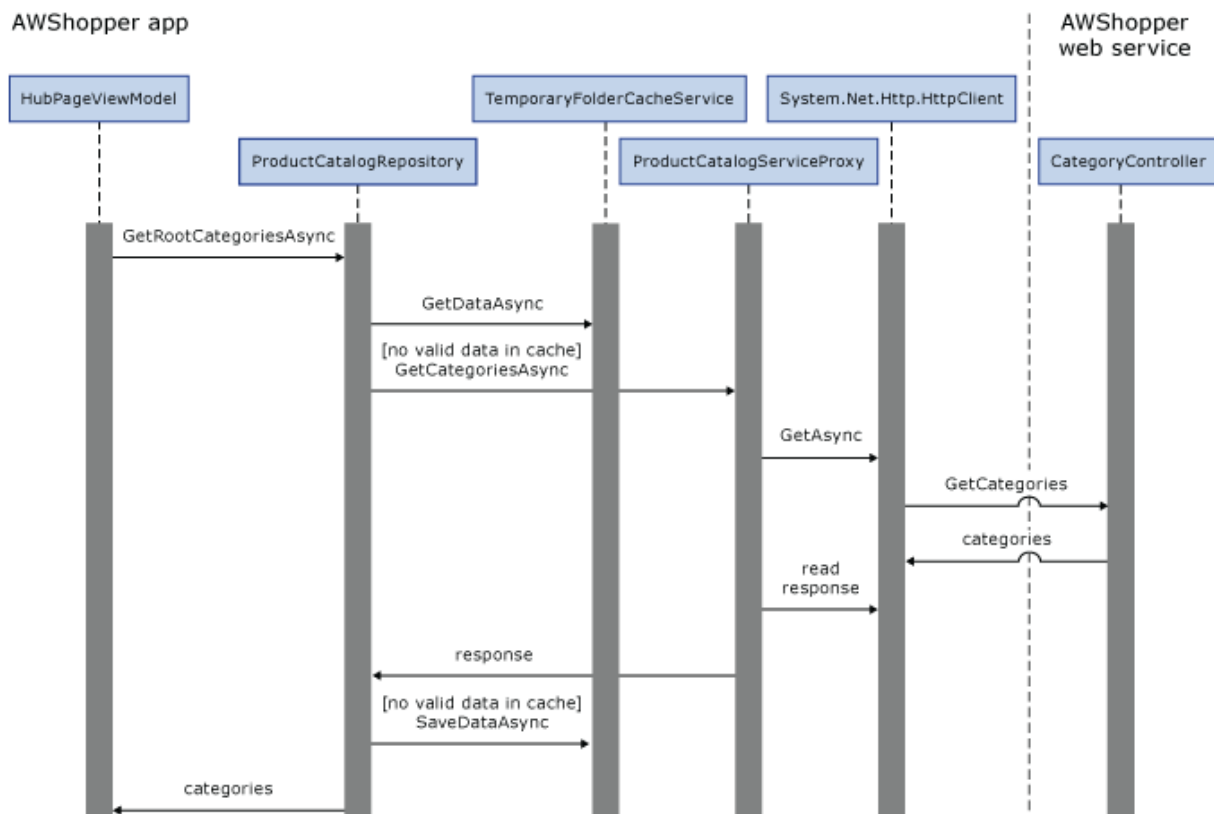
Consuming data

Analysis of the data transfer requirements revealed only limited interactions with the web service, so AdventureWorks Shopper implements a set of custom DTO classes to handle the data transfer with the web service. For more info see [Using model classes as data transfer objects](#). In order to further reduce the interaction with the web service, as much data as possible is retrieved in a single call to it. For example, instead of retrieving product categories in one web service call, and then retrieving products for a category in a second web service call, AdventureWorks Shopper retrieves a category and its products in a single web service call.

In the future, AdventureWorks may decide to use the OData protocol in order to use features such as batching and conflict resolution.

Note AdventureWorks Shopper does not secure the web service with Secure Sockets Layer (SSL), so a malicious client could impersonate the app and send malicious data. In your own app, you should protect any sensitive data that you need to transfer between the app and a web service by using SSL.

The following diagram shows the interaction of the classes that implement reading product category data for the hub page in AdventureWorks Shopper.



The **ProductCatalogRepository** is used to manage the data retrieval process, either from the web service or from a temporary cache stored on disk. The **ProductCatalogServiceProxy** class is used to retrieve product category data from the web service, with the **TemporaryFolderCacheService** class being used to retrieve product category data from the temporary cache.

In the **OnInitialize** method in the **App** class, the **ProductCatalogRepository** class is registered as a type mapping against the **IProductCatalogRepository** type with the Unity dependency injection container. Similarly, the **ProductCatalogServiceProxy** class is registered as a type mapping against the **IProductCatalogService** type. Then, when a view model class such as the **HubPageViewModel** class accepts an **IProductCatalogRepository** type, the Unity container will resolve the type and return an instance of the **ProductCatalogRepository** class.

When the **HubPage** is navigated to, the **OnNavigatedTo** method in the **HubPageViewModel** class is called. The following example shows code from the **OnNavigatedTo** method, which uses the **ProductCatalogRepository** instance to retrieve category data for display on the **HubPage**.

C#: AdventureWorks.UILogic\ViewModels\HubPageViewModel.cs

```
rootCategories = await _productCatalogRepository.GetRootCategoriesAsync(5);
```

The call to the **GetRootCategoriesAsync** method specifies the maximum amount of products to be returned for each category. This parameter can be used to optimize the amount of data returned by the web service, by avoiding returning an indeterminate number of products for each category.

The **ProductCatalogRepository** class, which implements the **IProductCatalogRepository** interface, uses instances of the **ProductCatalogServiceProxy** and **TemporaryFolderCacheService** classes to retrieve data for display on the UI. The following code example shows the **GetSubCategoriesAsync** method, which is called by the **GetRootCategoriesAsync** method, to asynchronously retrieve data from either the temporary cache on disk, or from the web service.

C#: AdventureWorks.UILogic\Repositories\ProductCatalogRepository.cs

```
public async Task<ReadOnlyCollection<Category>> GetSubcategoriesAsync(
    int parentId, int maxAmountOfProducts)
{
    string cacheFileName = String.Format("Categories-{0}-{1}", parentId,
        maxAmountOfProducts);

    try
    {
        // Case 1: Retrieve the items from the cache
        return await _cacheService
            .GetDataAsync<ReadOnlyCollection<Category>>(cacheFileName);
    }
    catch (FileNotFoundException)
    { }

    // Retrieve the items from the service
    var categories = await _productCatalogService
        .GetCategoriesAsync(parentId, maxAmountOfProducts);

    // Save the items in the cache
    await _cacheService.SaveDataAsync(cacheFileName, categories);

    return categories;
}
```

The method first calls the **GetDataAsync** method in the **TemporaryFolderCacheService** class to check if the requested data exists in the cache, and if it does, whether it has expired or not. Expiration is judged to have occurred if the data is present in the cache, but it is more than 5 minutes old. If the data exists in the cache and hasn't expired it is returned, otherwise a [FileNotFoundException](#) is thrown. If the data does not exist in the cache, or if it exists and has expired, a call to the **GetCategoriesAsync** method in the **ProductCatalogServiceProxy** class retrieves the data from the web service before it is cached.

The **ProductCatalogServiceProxy** class, which implements the **IProductCatalogService** interface, is used to retrieve product data from the web service if the data is not cached, or the cached data has expired. The following code example show the **GetCategoriesAsync** method that is invoked by the **GetSubCategoriesAsync** method in the **ProductCatalogRepository** class.

C#: AdventureWorks.UILogic\Services\ProductCatalogServiceProxy.cs

```

public async Task<ReadOnlyCollection<Category>> GetCategoriesAsync(
    int parentId, int maxAmountOfProducts)
{
    using (var httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(
            string.Format("{0}?parentId={1}&maxAmountOfProducts={2}",
                _categoriesBaseUrl, parentId, maxAmountOfProducts));
        response.EnsureSuccessStatusCode();
        var result = await response.Content
            .ReadAsStringAsync();

        return result;
    }
}

```

This method asynchronously retrieves the product categories from the web service by using the [HttpClient](#) class to send HTTP requests and receive HTTP responses from a URI. The call to [HttpClient.GetAsync](#) sends a GET request to the specified URI as an asynchronous operation, and returns a [Task](#) of type [HttpResponseMessage](#) that represents the asynchronous operation. The returned [Task](#) will complete after the content from the response is read. For more info about the [HttpClient](#) class see [Quickstart: Connecting using HttpClient](#).

When the [GetCategoriesAsync](#) method calls [HttpClient.GetAsync](#) this calls the [GetCategories](#) method in the [CategoryController](#) class in the AdventureWorks.WebServices project, which is shown in the following code example.

C#: AdventureWorks.WebServices\Controllers\CategoryController.cs

```

public IEnumerable<Category> GetCategories(int parentId, int maxAmountOfProducts)
{
    var categories = _categoryRepository.GetAll()
        .Where(c => c.ParentId == parentId);

    var trimmedCategories = categories.Select(NewCategory).ToList();
    FillProducts(trimmedCategories);

    foreach (var trimmedCategory in trimmedCategories)
    {
        var products = trimmedCategory.Products.ToList();
        if (maxAmountOfProducts > 0)
        {
            products = products.Take(maxAmountOfProducts).ToList();
        }
        trimmedCategory.Products = products;
    }

    return trimmedCategories;
}

```

This method uses an instance of the **CategoryRepository** class to return a static collection of **Category** objects that contain the category data returned by the web service.

Caching data

The **TemporaryFolderCacheService** class is used to cache data from the web service to the temporary app data store. This service is used by the **ProductCatalogRepository** class to decide whether to retrieve products from the web service, or from the cache in the temporary app data store.

As previously mentioned, the **GetSubCategoriesAsync** method in the **ProductCatalogRepository** class is used to asynchronously retrieve data from the product catalog. When it does this it first attempts to retrieve cached data from the temporary app data store by calling the **GetDataAsync** method, which is shown in the following code example.

C#: AdventureWorks.UILogic\Services\TemporaryFolderCacheService.cs

```
public async Task<T> GetDataAsync<T>(string cacheKey)
{
    await CacheKeyPreviousTask(cacheKey);
    var result = GetDataAsyncInternal<T>(cacheKey);
    SetCacheKeyPreviousTask(cacheKey, result);
    return await result;
}

private async Task<T> GetDataAsyncInternal<T>(string cacheKey)
{
    StorageFile file = await _cacheFolder.GetFileAsync(cacheKey);
    if (file == null) throw new FileNotFoundException("File does not exist");

    // Check if the file has expired
    var fileBasicProperties = await file.GetBasicPropertiesAsync();
    var expirationDate = fileBasicProperties.DateModified
        .Add(_expirationPolicy).DateTime;
    bool fileIsValid = DateTime.Now.CompareTo(expirationDate) < 0;
    if (!fileIsValid) throw new FileNotFoundException("Cache entry has expired.");

    string text = await FileIO.ReadTextAsync(file);
    var toReturn = Deserialize<T>(text);

    return toReturn;
}
```

The **CacheKeyPreviousTask** method ensures that since only one I/O operation at a time may access a cache key, cache read operations always wait for the prior task of the current cache key to complete before they start. The **GetDataAsyncInternal** method is called to see if the requested data exists in the cache, and if it does, whether it has expired or not.

The **SaveDataAsync** method in the **TemporaryFolderCacheService** class saves data retrieved from the web service to the cache, and is shown in the following code example.

C#: AdventureWorks.UILogic\Services\TemporaryFolderCacheService.cs

```
public async Task SaveDataAsync<T>(string cacheKey, T content)
{
    await CacheKeyPreviousTask(cacheKey);
    var result = SaveDataAsyncInternal<T>(cacheKey, content);
    SetCacheKeyPreviousTask(cacheKey, result);
    await result;
}

private async Task SaveDataAsyncInternal<T>(string cacheKey, T content)
{
    StorageFile file = await _cacheFolder.CreateFileAsync(cacheKey,
        CreationCollisionOption.ReplaceExisting);

    var textContent = Serialize<T>(content);
    await FileIO.WriteTextAsync(file, textContent);
}
```

As with the read operations, since only one I/O operation at a time may access a cache key, cache write operations always wait for the prior task of the current cache key to complete before they start. The **SaveDataAsyncInternal** method is called to serialize the data from the web service to the cache.

Note AdventureWorks Shopper does not directly cache images from the web service. Instead, we rely on the [Image](#) control's ability to cache images and display them if the server responds with an image.

Authentication

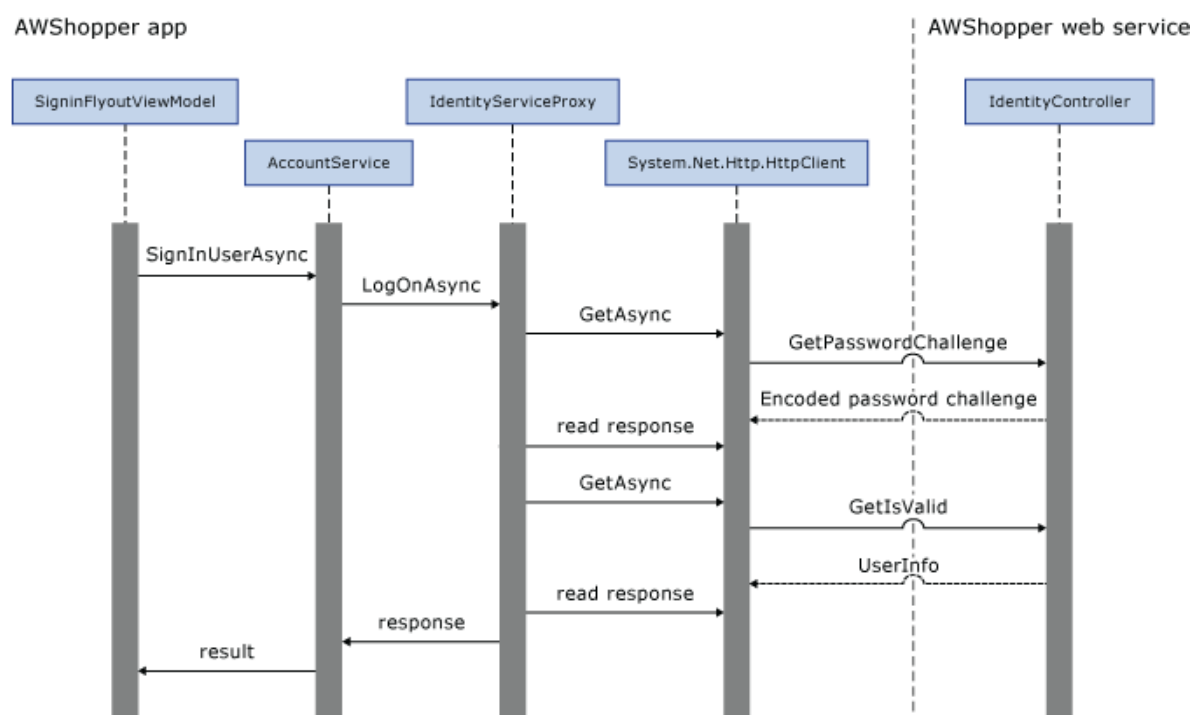
The AdventureWorks Shopper web service needs to know the identity of the user who places an order. The app externalizes as much of the authentication functionality as possible. This provides the flexibility to make changes to the approach used to handle authentication in the future without affecting the app. For example, the approach could be changed to enable users to identify themselves by using a Microsoft account. It's also important to ensure that the mechanism that the app uses to authenticate users is easy to implement on other platforms.

Ideally the web service should use a flexible, standards-based approach to authentication. However, such an approach is beyond the scope of this project. The approach adopted here is that the app requests a password challenge string from the web service that it then hashes using the user's password as the key. This hashed data is then sent to the web service where it's compared against a newly computed hashed version of the password challenge string, using the user's password stored in the web service as the key. Authentication only succeeds if the app and the web service have computed the same hash for the password challenge string. This approach avoids sending the user's password to the web service.

Note In the future, the app could replace the simple credentials authentication system with a claims-based approach. One option is to use the Simple Web Token and OAuth 2.0 protocol. This approach offers the following benefits:

- The authentication process is managed externally from the app.
- The authentication process uses established standards.
- The app can use a claims-based approach to handle any future authorization requirements.

The following illustration shows the interaction of the classes that implement credentials-based authentication in the AdventureWorks Shopper reference implementation.



Credentials-based user authentication is performed by the **AccountService** and **IdentityServiceProxy** classes in the app, and by the **IdentityController** class in the web service. In the **OnInitialize** method in the **App** class the **AccountService** class is registered as a type mapping against the **IAccountService** type with the Unity dependency injection container. Then, when a view model class such as the **SignInFlyoutViewModel** class accepts an **IAccountService** type, the Unity container will resolve the type and return an instance of the **AccountService** class.

When the user selects the **Submit** button on the **SignInFlyout**, the **SignInCommand** in the **SignInFlyOutViewModel** class is executed, which in turn calls the **SignInAsync** method. This method then calls the **SignInUserAsync** method on the **AccountService** instance. If the sign in is successful, the **SignInFlyOut** view is closed. The following code example shows part of the **SignInUserAsync** method in the **AccountService** class.

C#: AdventureWorks.UILogic\Services\AccountService.cs

```
var result = await _identityService.LogOnAsync(userName, password);
```

The **SignInUserAsync** method calls the **LogOnAsync** method in the instance of the **IdentityServiceProxy** class that's injected into the **AccountService** constructor from the Unity dependency injection container. The **IdentityServiceProxy** class, which implements the **IIdentityService** interface, uses the **LogOnAsync** method to authenticate user credentials with the web service. The following code example shows this method.

C#: AdventureWorks.UILogic\Services\IdentityServiceProxy.cs

```
public async Task<LogOnResult> LogOnAsync(string userId, string password)
{
    using (var handler = new HttpClientHandler
        { CookieContainer = new CookieContainer() })
    {
        using (var client = new HttpClient(handler))
        {
            // Ask the server for a password challenge string
            var requestId = CryptographicBuffer
                .EncodeToHexString(CryptographicBuffer.GenerateRandom(4));
            var challengeResponse = await client.GetAsync(_clientBaseUrl +
                "GetPasswordChallenge?requestId=" + requestId);
            challengeResponse.EnsureSuccessStatusCode();
            var challengeEncoded = await challengeResponse.Content
                .ReadAsStringAsync();
            var challengeBuffer = CryptographicBuffer
                .DecodeFromHexString(challengeEncoded);

            // Use HMAC_SHA512 hash to encode the challenge string using the
            // password being authenticated as the key.
            var provider = MacAlgorithmProvider
                .OpenAlgorithm(MacAlgorithmNames.HmacSha512);
            var passwordBuffer = CryptographicBuffer
                .ConvertStringToBinary(password, BinaryStringEncoding.Utf8);
            var hmacKey = provider.CreateKey(passwordBuffer);
            var buffHmac = CryptographicEngine.Sign(hmacKey, challengeBuffer);
            var hmacString = CryptographicBuffer.EncodeToHexString(buffHmac);

            // Send the encoded challenge to the server for authentication (to
            // avoid sending the password itself)
            var response = await client.GetAsync(_clientBaseUrl + userId +
                "?requestID=" + requestId + "&passwordHash=" + hmacString);

            // Raise exception if sign in failed
            response.EnsureSuccessStatusCode();

            // On success, return sign in results from the server response packet
            var result = await response.Content.ReadAsAsync<UserInfo>();
            var serverUri = new Uri(Constants.ServerAddress);
            return new LogOnResult { ServerCookieHeader = handler.CookieContainer
                .GetCookieHeader(serverUri), UserInfo = result };
        }
    }
}
```

This method generates a random request identifier that is encoded as a hex string and sent to the web service. The **GetPasswordChallenge** method in the **IdentityController** class in the AdventureWorks.WebServices project receives the request identifier and responds with a hexadecimal encoded password challenge string that the app reads and decodes. The app then hashes the password challenge with the [HMACSHA512](#) hash function, using the user's password as the key. The hashed password challenge is then sent to the web service for authentication by the **GetIsValid** method in the **IdentityController** class in the AdventureWorks.WebServices project. If authentication succeeds, a new instance of the **LogOnResult** class is returned by the method.

The **LogOnAsync** method communicates with the web service through calls to [HttpClient.GetAsync](#), which sends a GET request to the specified URI as an asynchronous operation, and returns a [Task](#) of type [HttpResponseMessage](#) that represents the asynchronous operation. The returned **Task** will complete after the content from the response is read. For more info about the [HttpClient](#) class see [Quickstart: Connecting using HttpClient](#).

The **IdentityController** class, in the AdventureWorks.WebServices project, is responsible for sending hexadecimal encoded password challenge strings to the app, and for performing authentication of the hashed password challenges it receives from the app. The class contains a static [Dictionary](#) named **Identities** that contains the valid credentials for the web service. The following code example shows the **GetIsValid** method in the **IdentityController** class.

C#: AdventureWorks.WebServices\Controllers\IdentityController.cs

```
public UserInfo GetIsValid(string id, string requestId, string passwordHash)
{
    byte[] challenge = null;
    if (requestId != null && ChallengeCache.Contains(requestId))
    {
        // Retrieve the saved challenge bytes
        challenge = (byte[])ChallengeCache[requestId];
        // Delete saved challenge (each challenge is used just one time).
        ChallengeCache.Remove(requestId);
    }

    lock (Identities)
    {
        // Check that credentials are valid.
        if (challenge != null && id != null && passwordHash != null &&
            Identities.ContainsKey(id))
        {
            // Compute hash for the previously issued challenge string using the
            // password from the server's credentials store as the key.
            var serverPassword = Encoding.UTF8.GetBytes(Identities[id]);
            using (var provider = new HMACSHA512(serverPassword))
            {
                var serverHashBytes = provider.ComputeHash(challenge);
                // Authentication succeeds only if client and server have computed
                // the same hash for the challenge string.
                var clientHashBytes = DecodeFromHexString(passwordHash);
                if (!serverHashBytes.SequenceEqual(clientHashBytes))
            }
        }
    }
}
```

```

        throw new HttpResponseException(HttpStatusCode.Unauthorized);
    }

    if (HttpContext.Current != null)
        FormsAuthentication.SetAuthCookie(id, false);
    return new UserInfo { UserName = id };
}
else
{
    throw new HttpResponseException(HttpStatusCode.Unauthorized);
}
}
}

```

This method is called in response to the **LogOnAsync** method sending a hashed password challenge string to the web service. The method retrieves the previously issued password challenge string that was sent to the app, and then removes it from the cache as each password challenge string is used only once. The retrieved password challenge is then hashed with the [HMACSHA512](#) hash function, using the user's password stored in the web service as the key. The newly computed hashed password challenge string is then compared against the hashed challenge string received from the app. Authentication only succeeds if the app and the web service have computed the same hash for the password challenge string, in which case a new **UserInfo** instance containing the user name is returned to the **LogOnAsync** method.

Note The Windows Runtime includes APIs that provide authentication, authorization and data security. For example, the AdventureWorks Shopper reference implementation uses the [MacAlgorithmProvider](#) class to securely authenticate user credentials over an unsecured channel. However, this is only one choice among many. For more info see [Introduction to Windows Store app security](#).

Handling suspend, resume, and activation in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Save application data when the app is being suspended.
- Use the saved application data to restore the app when needed.
- Allow views and view models to save and restore state that's relevant to each by using the **MvvmAppBase** class, the **VisualStateAwarePage** class, and the **RestorableState** custom attribute, provided by the [Microsoft.Practices.Prism.StoreApps](https://github.com/PrismLibrary/Microsoft.Practices.Prism.StoreApps) library.

The AdventureWorks Shopper reference implementation fully manages its execution lifecycle by using [Prism for the Windows Runtime](https://github.com/PrismLibrary/Prism), which helps to manage the execution lifecycle of Windows Store apps that use the MVVM pattern. Suspension can happen at any time, and when it does you need to save your app's data so that the app can resume correctly.

You will learn

- How Windows determines an app's execution state.
- How the app's activation history affects its behavior.
- How to implement support for suspend, resume, and activation by using the [Microsoft.Practices.Prism.StoreApps](https://github.com/PrismLibrary/Microsoft.Practices.Prism.StoreApps) library.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Windows Store apps should be designed to save their state and suspend when the user switches away from them. They could restore their state and resume when the user switches back to them. The following list summarizes the decisions to make when implementing suspend and resume in your app:

- Should your app be activated through any contracts or extensions or will it only be activated by the user launching it?
- Does your app need to behave differently when it's closed by the user rather than when it's closed by Windows?
- Does your app need to resume as the user left it, rather than starting it fresh, following suspension?
- Does your app need to start fresh if a long period of time has elapsed since the user last accessed it?
- Should your app update the UI when resuming from suspension?

- Does your app need to request data from a network or retrieve large amounts of data from disk when launched?

Your app must register to receive the [Activated](#) event in order to participate in activation. If your app needs to be activated through any contracts or extensions other than just normal launch by the user, you can use your app's **Activated** event handler to test to see how the app was activated. Examples of activation other than normal user launch include another app launching a file whose file type your app is registered to handle, and your app being chosen as the target for a share operation. For more info see [Activating an app](#).

If your app needs to behave differently when it is closed by the user, rather than when it is closed by Windows, the [Activated](#) event handler can be used to determine whether the app was terminated by the user or by Windows. For more info see [Activating an app](#).

Following suspension, most Windows Store apps should resume as the user left them rather than starting fresh. Explicitly saving your application data helps ensure that the user can resume your app even if Windows terminates it. It's a best practice to have your app save its state when it's suspended and restore its state when it's launched after termination. However, if your app was unexpectedly closed, assume that stored application data is possibly corrupt. The app should not try to resume but rather start fresh. Otherwise, restoring corrupt application data could lead to an endless cycle of activation, crash, and being closed. For more info see [Guidelines for app suspend and resume \(Windows Store apps\)](#), [Suspending an app](#), [Resuming an app](#), and [Activating an app](#).

If there's a good chance that users won't remember or care about what was happening when they last saw your app, launch it from its default launch state. You must determine an appropriate period after which your app should start fresh. For example, a news reader app should start afresh if the downloaded news articles are stale. However, if there is any doubt about whether to resume or start fresh, you should resume the app right where the user left off. For more info see [Resuming an app](#) and [Activating an app](#).

When resuming your app after it was suspended, update the UI if the content has changed since it was last visible to the user. This ensures that to the user the app appears as though it was running in the background. For more info see [Resuming an app](#).

If your app needs to request data from a network or retrieve large amounts of data from disk, when the app is launched, these activities should be completed outside of activation. Use a custom loading UI or an extended splash screen while the app waits for these long running operations to finish. For more info see [How to activate an app](#).

Suspend and resume in AdventureWorks Shopper

The AdventureWorks Shopper reference implementation was designed to suspend correctly when the user moves away from it, or when Windows enters a low power state. It was also designed to resume correctly when the user moves back to it, or when Windows leaves the low power state.

AdventureWorks Shopper uses the [Microsoft.Practices.Prism.StoreApps](#) library to provide both view and view model support for suspend and resume. This was achieved by:

- Saving application data when the app is being suspended.
- Resuming the app in the state that the user left it in.
- Saving the page state to minimize the time required to suspend the app when navigating away from a page.
- Allowing views and view models to save and restore state that's relevant to each. For example, AdventureWorks Shopper saves the scroll position of certain [GridView](#) controls as view state. This is achieved by overriding the **SaveState** and **LoadState** methods of the **VisualStateAwarePage** class in a view's class.
- Using the saved application data to restore the app state, when the app resumes after being terminated.

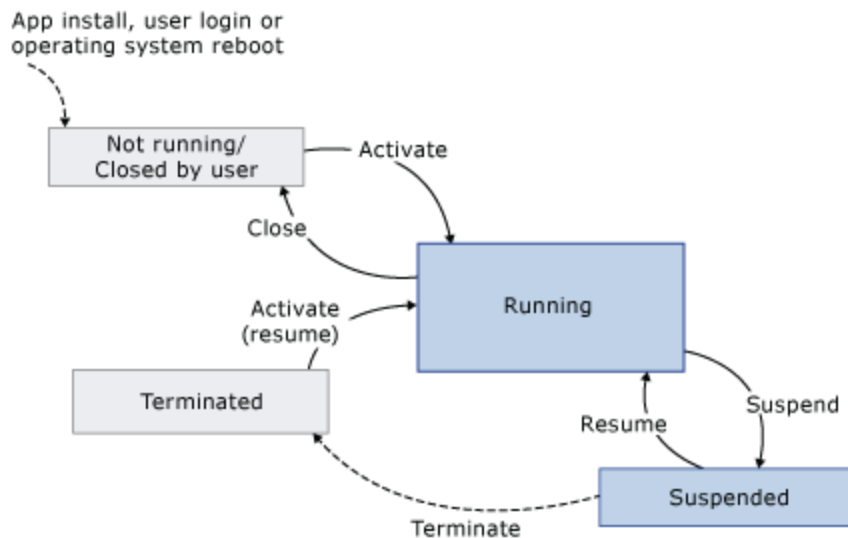
For more info, see [Guidelines for app suspend and resume \(Windows Store apps\)](#).

Understanding possible execution states

Which events occur when you activate an app depends on the app's execution history. There are five cases to consider. The cases correspond to the values of the [Windows.ActivationModel.Activation.ApplicationExecutionState](#) enumeration.

- **NotRunning**
- **Terminated**
- **ClosedByUser**
- **Suspended**
- **Running**

The following diagram shows how Windows determines an app's execution state. In the diagram, the white rectangles indicate that the app isn't loaded into system memory. The blue rectangles indicate that the app is in memory. The dashed arcs are changes that occur without any notification to the running app. The solid arcs are actions that include app notification.



The execution state depends on the app's history. For example, when the user starts the app for the first time after installing it or after restarting Windows, the previous execution state is **NotRunning**, and the state after activation is **Running**. When activation occurs, the activation event arguments include a [PreviousExecutionState](#) property that indicates the state the app was in before it was activated.

If the user switches to a different app or if the system enters a low power mode of operation, Windows notifies the app that it's being suspended. At this time, you must save the navigation state and all user data that represents the user's session. You should also free exclusive system resources, like open files and network connections.

Windows allows 5 seconds for an app to handle the [Suspending](#) event. If the **Suspending** event handler doesn't complete within that amount of time, Windows behaves as though the app has stopped responding and terminates it. After the app responds to the **Suspending** event, its state is **Suspended**. If the user switches back to the app, Windows resumes it and allows it to run again.

Windows might terminate an app, without notification, after it has been suspended. For example, if the device is low on resources it might reclaim resources that are held by suspended apps. If the user launches your app after Windows has terminated it, the app's previous execution state at the time of activation is **Terminated**.

You can use the previous execution state to determine whether your app needs to restore the data that it saved when it was last suspended, or whether you must load your app's default data. In general, if the app stops responding or the user closes it, restarting the app should take the user to the app's default initial navigation state. When an app is activated after being terminated, it should load the application data that it saved during suspension so that the app appears as it did when it was suspended.

When an app is suspended but hasn't yet been terminated, you can resume the app without additional work as it will still be in memory.

For a description of the suspend and resume process, see [Application lifecycle \(Windows Store apps\)](#). For more info about each of the possible previous execution states, see the [ApplicationExecutionState](#) enumeration. You might also want to consult [Guidelines for app suspend and resume \(Windows Store apps\)](#) for info about the recommended user experience for suspend and resume.

Implementation approaches for suspend and resume

For Windows Store apps such as the AdventureWorks Shopper reference implementation that use the [Microsoft.Practices.Prism.StoreApps](#) library, implementing suspend and resume involves four components:

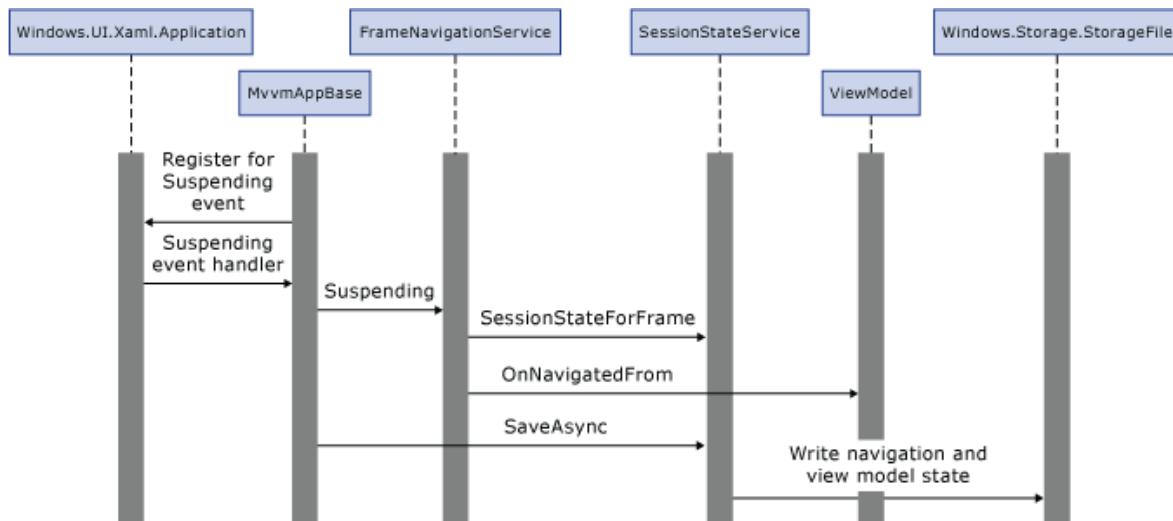
- **Windows Core.** The [CoreApplicationView](#) class's [Activated](#) event allows an app to receive activation-related notifications.
- **XAML.** The [Application](#) class provides the [OnLaunched](#) method that your app's class should override to perform application initialization and to display the initial content. The **Application** class invokes the **OnLaunched** method when the user starts the app. When you create a new project for a Windows Store app using one of the Visual Studio project templates for C# apps, Visual Studio creates an **App** class that derives from **Application** and overrides the **OnLaunched** method. In MVVM apps such as AdventureWorks Shopper, much of the Visual Studio created code in the **App** class has been moved to the **MvvmAppBase** class that the **App** class then derives from.
- **Microsoft.Practices.Prism.StoreApps** classes. If you base your MVVM app on the reusable classes of the Microsoft.Practices.Prism.StoreApps library, many aspects of suspend/resume will be provided for you. For example, the **SessionStateService** class will provide a way to save and restore state. If you annotate properties of your view models with the **RestorableState** custom attribute, they will automatically be saved and restored at the correct time. The **SessionStateService** also interacts with the [Frame](#) class to save and restore the app's navigation stack for you.
- **Your app's classes.** View classes can save view state with each invocation of the **OnNavigatedFrom** method. For example, some view classes in AdventureWorks Shopper save user interface state such as scroll bar position. Model state is saved by view model classes, through the base **ViewModel** class.

Note A user can activate an app through a variety of contracts and extensions. The **Application** class only calls the [OnLaunched](#) method in the case of a normal launch. For more info about how to detect other activation events see the [Application](#) class. In the AdventureWorks Shopper reference implementation we handle both normal launch and launch through the Search contract.

AdventureWorks Shopper does not directly interact with the [CoreApplicationView](#) class's activation-related events. We mention them here in case your app needs access to these lower-level notifications.

Suspending an app

Suspension support is provided by the [Microsoft.Practices.Prism.StoreApps](#) library. In order to add suspension support to an app that derives from the **MvvmAppBase** class in this library, you only need to annotate properties of view models that you wish to save during suspension with the **RestorableState** custom attribute. In addition, if additional suspension logic is required you should override the **OnNavigatedFrom** method of the base **ViewModel** class. The following diagram shows the interaction of the classes that implement the suspend operation in AdventureWorks Shopper.



Here, the **MvvmAppBase** class registers a handler for the [Suspending](#) event that is provided by the [Application](#) base class.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```
this.Suspending += OnSuspending;
```

Windows invokes the **OnSuspending** event handler before it suspends the app. The **MvvmAppBase** class uses the event handler to save relevant app and user data to persistent storage.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
    IsSuspending = true;
    try
    {
        var deferral = e.SuspendingOperation.GetDeferral();

        // Bootstrap inform navigation service that app is suspending.
        NavigationService.Suspending();

        // Save application state
        await SessionStateService.SaveAsync();
    }
}
```

```

        deferral.Complete();
    }
    finally
    {
        IsSuspending = false;
    }
}

```

The **OnSuspending** event handler is asynchronous. If a [Suspending](#) event's handler is asynchronous, it must notify its caller when its work has finished. To do this, the handler invokes the [GetDeferral](#) method that returns a [SuspendingDeferral](#) object. The **Suspending** method of the **FrameNavigationService** class is then called. The **SessionStateService** class's **SaveAsync** method then persists the app's navigation and user data to disk. After the save operation has finished, the [Complete](#) method of the **SuspendingDeferral** object is called to notify the operating system that the app is ready to be suspended. The following code example shows the **Suspending** method of the **FrameNavigationService** class.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```

public void Suspending()
{
    NavigateFromCurrentViewModel(true);
}

```

The **Suspending** method of the **FrameNavigationService** class calls the **NavigateFromCurrentViewModel** method that handles the suspension. The following code example shows the **NavigateFromCurrentViewModel** method of the **FrameNavigationService** class.

C#: Microsoft.Practices.Prism.StoreApps\FrameNavigationService.cs

```

private void NavigateFromCurrentViewModel(bool suspending)
{
    var departingView = _frame.Content as FrameworkElement;
    if (departingView == null) return;
    var frameState = _sessionStateService.GetSessionStateForFrame(_frame);
    var departingViewModel = departingView.DataContext as INavigationAware;

    var viewModelKey = "ViewModel-" + _frame.BackStackDepth;
    if (departingViewModel != null)
    {
        var viewModelState = frameState.ContainsKey(viewModelKey) ?
            frameState[viewModelKey] as Dictionary<string, object> : null;

        departingViewModel.OnNavigatedFrom(viewModelState, suspending);
    }
}

```

The **NavigateFromCurrentViewModel** method gets the session state for the current view and calls the **OnNavigatedFrom** method on the current view model. All **OnNavigatedFrom** methods feature a

suspending parameter that tells the view model whether it is being suspended. If the parameter is **true** it means that no change should be made to state that would invalidate the page and that a subsequent **OnNavigatedTo** method might not be called, for instance if the app resumes without being terminated. This allows you to implement additional functionality in view model classes that may be required when the **OnNavigatedFrom** method is called when the app isn't being suspended.

In the **NavigateFromCurrentViewModel** method the **frameState** dictionary is the dictionary for the frame. Each item in the dictionary is a view model that is at a specific depth in the frame back stack. Each view model also has its own state dictionary, **viewModelState**, that is passed to the view model's **OnNavigatedFrom** method. This approach is preferable to each view model creating entries in the **frameState** dictionary using the view models type as the key.

All of the view model classes in the AdventureWorks Shopper reference implementation derive from the **ViewModel** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, that implements the **OnNavigatedFrom** method. This method calls the **FillStateDictionary** method to add any view model state to the frame state, as shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\ViewModel.cs

```
public virtual void OnNavigatedFrom(Dictionary<string, object> viewModelState,
    bool suspending)
{
    if (viewModelState != null)
    {
        FillStateDictionary(viewModelState, this);
    }
}
```

The **FillStateDictionary** method iterates through any properties in the view model and stores the value of any properties that possess the **[RestorableState]** custom attribute.

In the AdventureWorks Shopper reference implementation we use the **suspending** parameter that's passed to the **OnNavigatedFrom** methods in the view model classes, and the **MvvmAppBase.IsSuspending** property that's used by view classes, to establish context when the app receives callbacks from the **SaveAsync** method. We need to know if the **OnNavigatedFrom** method is called in the case of normal operation, or if it is being called while saving state in response to a **Suspending** event. For example, when the user navigates away from the **HubPage** we want to remove the redirection of input to the Search charm. However, if the app is suspending while on the **HubPage**, the redirection of input to the Search charm should not be removed. For more info see [Enabling users to type into the search box](#). The following code example shows the **OnNavigatedFrom** method of the **HubPageViewModel** class.

C#: AdventureWorks.UILogic\ViewModels\HubPageViewModel.cs

```

public override void OnNavigatedFrom(Dictionary<string, object> viewModelState,
    bool suspending)
{
    base.OnNavigatedFrom(viewModelState, suspending);
    if (!suspending)
    {
        _searchPaneService.ShowOnKeyboardInput(false);
    }
}

```

The **SaveAsync** method of the **SessionStateService** class writes the current session state to disk. The **SaveAsync** method calls the [GetNavigationState](#) method of each registered [Frame](#) object in order to persist the serialized navigation history (the frame stack). In AdventureWorks Shopper there is only one registered frame, and it corresponds to the **rootFrame** in the **InitializeFrameAsync** method in the **MvvmAppBase** class.

Note As a side effect, the [GetNavigationState](#) method invokes the **OnNavigatedFrom** method of each of the frame's associated page objects. This allows each page to save view state such as the current scroll position of its controls.

Some service and repository classes also persist state to survive termination. In order to do this they use an instance of the **SessionStateService** class that implements the **ISessionStateService** interface. The following code example shows how the **AccountService** class persists the user's credentials.

C#: AdventureWorks.UILogic\Services\AccountService.cs

```

_sessionStateService.SessionState[UserNameKey] = userName;
_sessionStateService.SessionState[PasswordKey] = password;

```

The state is persisted to the same file that the view models persist state to through the **[RestorableState]** attribute.

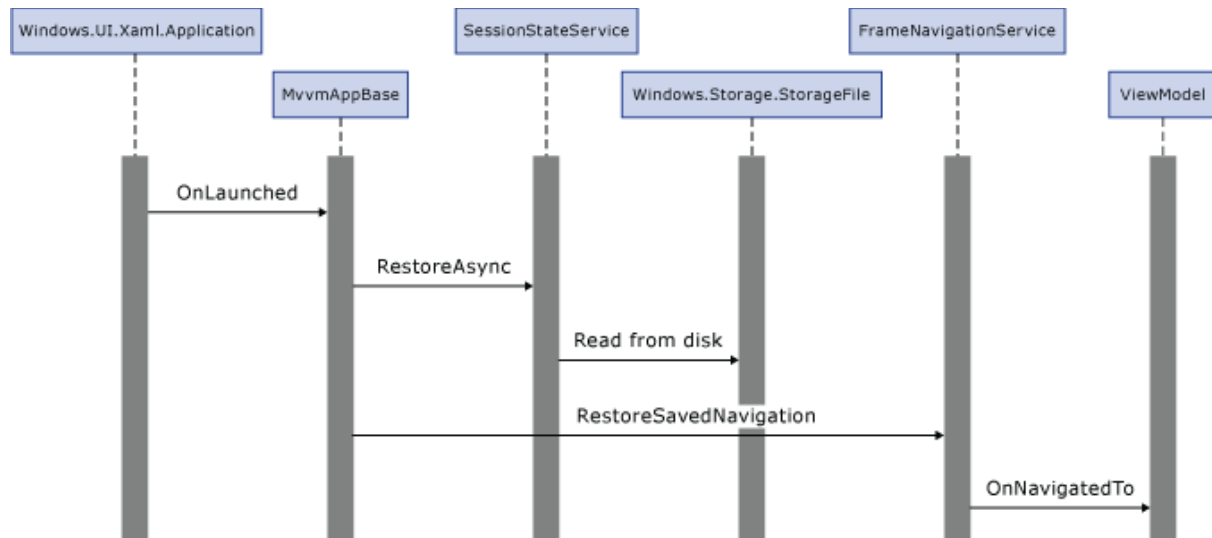
Resuming an app

When an app resumes from the **Suspended** state, it enters the **Running** state and continues from where it was when it was suspended. No application data is lost, because it has not been removed from memory. Most apps don't need to do anything if they are resumed before they are terminated by the operating system.

The AdventureWorks Shopper reference implementation does not register an event handler for the [Resuming](#) event. In the rare case when an app does register an event handler for the **Resuming** event, the handler is called when the app resumes from the **Suspended** state.

Activating an app

Activation support is provided by the [Microsoft.Practices.Prism.StoreApps](#) library. If Windows has terminated a suspended app, the [Application](#) base class calls the [OnLaunched](#) method when the app becomes active again. The following diagram shows the interaction of classes in AdventureWorks Shopper that restore the app after it has been terminated.



The **MvvmAppBase** class overrides the [OnLaunched](#) method of the [Windows.UI.Xaml.Application](#) base class. When the **OnLaunched** method runs, its argument is a [LaunchActivatedEventArgs](#) object. This object contains an [ApplicationExecutionState](#) enumeration that tells you the app's previous execution state. The **OnLaunched** method calls the **InitializeFrameAsync** method to initialize the app's [Frame](#) object. The following code example shows the relevant code from the **InitializeFrameAsync** method.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```

if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
    await SessionStateService.RestoreSessionStateAsync();
}

OnInitialize(args);
if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
    // Restore the saved session state and navigate to the last page visited
    try
    {
        SessionStateService.RestoreFrameState();
        NavigationService.RestoreSavedNavigation();
    }
    catch (SessionStateServiceException)
    {
        // Something went wrong restoring state.
        // Assume there is no state and continue
    }
}

```

The code checks its argument to see whether the previous state was **Terminated**. If so, the method calls the **SessionStateService** class's **RestoreSessionStateAsync** method to recover saved settings. The **RestoreSessionStateAsync** method reads the saved state info, and then the **OnInitialize** method is called which is overridden in the **App** class. This method registers instances and types with the Unity dependency injection container. Then, if the previous execution state of the app was **Terminated**, the saved session state is restored and the app navigates to the last page was that visited prior to termination. This is achieved by calling the **RestoreSavedNavigation** method of the **FrameNavigationService** class that in turn simply calls the **NavigateToCurrentViewModel** method, which gets the session state for the current view, and calls the **OnNavigatedTo** method on the current view model.

C#: Microsoft.Practices.Prism.StoreApps\FrameNavigationService.cs

```
private void NavigateToCurrentViewModel(NavigationMode navigationMode,
    object parameter)
{
    var frameState = _sessionStateService.GetSessionStateForFrame(_frame);
    var viewModelKey = "ViewModel-" + _frame.BackStackDepth;

    if (navigationMode == NavigationMode.New)
    {
        // Clear existing state for forward navigation when adding a new
        // page/view model to the navigation stack
        var nextViewModelKey = viewModelKey;
        int nextViewModelIndex = _frame.BackStackDepth;
        while (frameState.Remove(nextViewModelKey))
        {
            nextViewModelIndex++;
            nextViewModelKey = "ViewModel-" + nextViewModelIndex;
        }
    }

    var newView = _frame.Content as FrameworkElement;
    if (newView == null) return;
    var newViewModel = newView.DataContext as INavigationAware;
    if (newViewModel != null)
    {
        Dictionary<string, object> viewModelState;
        if (frameState.ContainsKey(viewModelKey))
        {
            viewModelState = frameState[viewModelKey] as
                Dictionary<string, object>;
        }
        else
        {
            viewModelState = new Dictionary<string, object>();
        }
        newViewModel.OnNavigatedTo(parameter, navigationMode, viewModelState);
        frameState[viewModelKey] = viewModelState;
    }
}
```

All of the view model classes in the AdventureWorks Shopper reference implementation derive from the **ViewModel** base class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, which implements the **OnNavigatedTo** method. This method simply calls the **RestoreViewModel** method to restore any view model state from the frame state, as shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\ViewModel.cs

```
public virtual void OnNavigatedTo(object navigationParameter,
    NavigationMode navigationMode, Dictionary<string, object> viewModelState)
{
    if (viewModelState != null)
    {
        RestoreViewModel(viewModelState, this);
    }
}
```

The **RestoreViewModel** method iterates through any properties in the view model and restores the values of any properties that possess the **[RestorableState]** attribute, from the frame state.

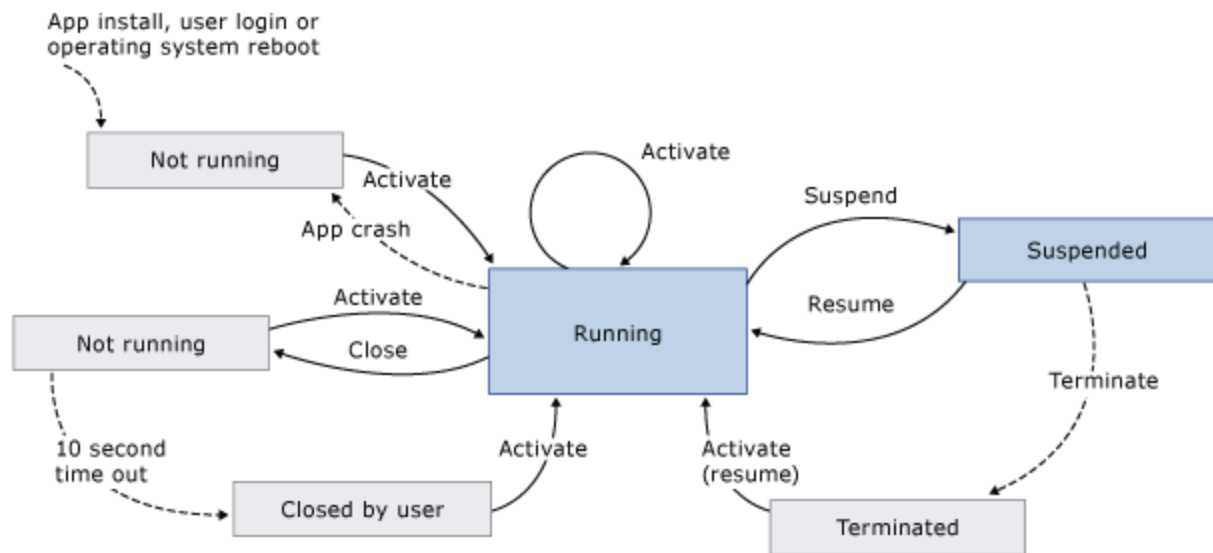
AdventureWorks Shopper can also be activated through the Search contract. For more info see [Responding to search queries](#).

Other ways to close the app

Apps don't contain UI for closing the app, but users can choose to close an app by pressing Alt+F4, dragging the app to the bottom of the screen, or selecting the **Close** context menu for the app when it's in the sidebar. When an app is closed by any of these methods, it enters the **NotRunning** state for approximately 10 seconds and then transitions to the **ClosedByUser** state.

Apps shouldn't close themselves programmatically as part of normal execution. When you close an app programmatically, Windows treats this as an app crash. The app enters the **NotRunning** state and remains there until the user activates it again.

The following diagram shows how Windows determines an app's execution state. Windows takes app crashes and user close actions into account, as well as the suspend or resume state. In the diagram, the white rectangles indicate that the app isn't loaded into system memory. The blue rectangles indicate that the app is in memory. The dashed lines are changes that occur without any modification to the running app. The solid lines are actions that include app notification.



Communicating between loosely coupled components in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Use the [Microsoft.Practices.Prism.PubSubEvents](#) library to communicate between loosely coupled components in your app.
- Notify subscribers by retrieving the pub/sub event from the event aggregator and calling its **Publish** method of the **PubSubEvent<TPayload>** class.
- Register to receive notifications by using one of the **Subscribe** method overloads available in the **PubSubEvent<TPayload>** class.

When developing business apps for the Windows Store, a common approach is to separate functionality into loosely coupled components so that the app is easily maintainable. Two objects are loosely coupled if they exchange data but have no type or object references to each other. The types of the objects may be defined in separate, unrelated assemblies, with unrelated lifetimes. This allows the components to be independently developed, tested, deployed, and updated. In this article we examine a technique for communication between loosely coupled components called event aggregation. Event aggregation can reduce dependencies between assemblies in a solution. The AdventureWorks Shopper reference implementation uses the event aggregator provided by [Prism for the Windows Runtime](#).

You will learn

- How event aggregation enables communication between loosely coupled components in an app.
- How to define a pub/sub event, publish it, and subscribe to it.
- How to manually unsubscribe from a pub/sub event when using a strong delegate reference.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Event aggregation allows communication between loosely coupled components in an app, removing the need for components to have a reference to each other. The following list summarizes the decisions to make when using event aggregation in your app:

- When should I use event aggregation over Microsoft .NET events?
- How should I subscribe to pub/sub events?
- How can a subscriber update the UI if the event is published from a background thread?
- Does the subscriber need to handle every instance of a published event?

- Do I need to unsubscribe from subscribed events?

Events in .NET implement the publish-subscribe pattern. The publisher and subscriber lifetimes are coupled by object references to each other, and the subscriber type must have a reference to the publisher type.

Event aggregation is a design pattern that enables communication between classes that are inconvenient to link by object and type references. This mechanism allows publishers and subscribers to communicate without having a reference to each other. Therefore, .NET events should be used for communication between components that already have object reference relationships (such as a control and the page that contains it), with event aggregation being used for communication between loosely coupled components (such as two separate page view models in an app). For more info see [Event aggregation](#).

There are several ways to subscribe to events when using event aggregation. The simplest is to register a delegate reference of the event handler method that will be called on the publisher's thread. For more info see [Subscribing to events](#).

If you need to be able to update UI elements when an event is received, you can subscribe to receive the event on the UI thread.

When subscribing to a pub/sub event, you can request that notification of the event will occur in the UI thread. This is useful, for example, when you need to update the UI in response to the event. For more info see [Subscribing on the UI thread](#).

Subscribers do not need to handle every instance of a published event, as they can specify a delegate that is executed when the event is published to determine if the payload of the published event matches a set of criteria required to have the subscriber callback invoked. For more info see [Subscription filtering](#).

By default, event aggregation maintains a weak delegate reference to a subscriber's handler. This means that the reference will not prevent garbage collection of the subscriber, and it relieves the subscriber from the need to unsubscribe. If you have observed performance problems with events, you can use strongly referenced delegates when subscribing to an event, and then unsubscribe from the event when it's no longer required. For more info see [Subscribing using strong references](#).

Event aggregation in AdventureWorks Shopper

The AdventureWorks Shopper reference implementation uses the [Microsoft.Practices.Prism.PubSubEvents](#) library to communicate between loosely coupled components. This is a Portable Class Library that contains classes that implement event aggregation. For more info see [Prism for the Windows Runtime reference](#).

The AdventureWorks Shopper reference implementation defines the **ShoppingCartUpdatedEvent** class and **ShoppingCartItemUpdatedEvent** class for use with event aggregation. You invoke the **ShoppingCartUpdatedEvent** singleton instance's **Publish** method when the signed in user has

changed, to notify the **ShoppingCartTabUserControl** of the change. The **ShoppingCartTabUserControl** is included on the **HubPage**, **GroupDetailPage**, and **ItemDetailPage** views, with there being no type or object references between the **ShoppingCartTabUserControl** and its parent pages.

The **ShoppingCartItemUpdated** event is published whenever a product is added to the shopping cart, so that the **ShoppingCartTabUserControlViewModel** class can be updated. For more info see [Event aggregation](#).

Pub/sub events in the AdventureWorks Shopper reference implementation are published on the UI thread, with the subscribers receiving the event on the same thread. Weak reference delegates are used for both events, and so the events do not need to be unsubscribed from. For more info see [Subscribing to events](#).

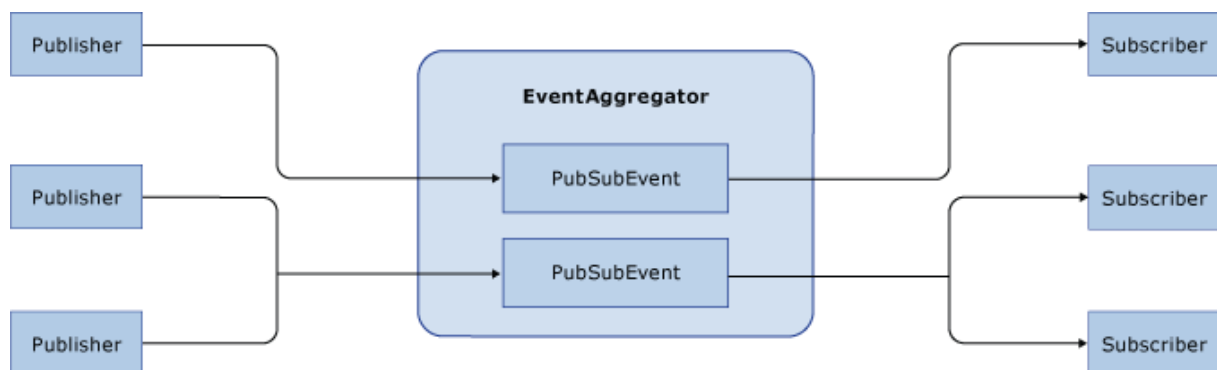
Note Lambda expressions that capture the **this** reference cannot be used as weak references. You should use instance methods as the **Subscribe** method's **action** and **filter** parameters if you want to take advantage of the **PubSubEvent** class's weak reference feature.

Event aggregation

.NET events are the most simple and straightforward approach for a communication layer between components if loose coupling is not required. Event aggregation should be used for communication when it's inconvenient to link objects with type and object references.

Note If you use .NET events, you have to consider memory management, especially if you have a short lived object that subscribes to an event of a static or long lived object. If you do not remove the event handler, the subscriber will be kept alive by the reference to it in the publisher, and this will prevent or delay the garbage collection of the subscriber.

The event aggregator provides multicast publish/subscribe functionality. This means that there can be multiple publishers that invoke the **Publish** method of a given **PubSubEvent<TPayload>** instance and there can be multiple subscribers listening to the same **PubSubEvent<TPayload>** instance. A subscriber can have more than one subscription to a single **PubSubEvent<TPayload>** instance. The following diagram shows this relationship.



The **EventAggregator** class is responsible for locating or building singleton instances of pub/sub event classes. The class implements the **IEventAggregator** interface, shown in the following code example.

C#: Microsoft.Practices.Prism.PubSubEvents\IEventAggregator.cs

```
public interface IEventAggregator
{
    TEventType GetEvent<TEventType>() where TEventType : EventBase, new();
}
```

In the AdventureWorks Shopper reference implementation, an instance of the **EventAggregator** class is created in the **OnLaunched** method in the **App** class. This instance is then passed as an argument to the constructors of view model classes that need it.

Defining and publishing pub/sub events

In apps such as the AdventureWorks Shopper reference implementation that use event aggregation, event publishers and subscribers are connected by the **PubSubEvent<TPayload>** class, which is the base class for an app's specific events. **TPayload** is the type of the event's payload. The **PubSubEvent<TPayload>** class maintains the list of subscribers and handles event dispatching to the subscribers. The class contains **Subscribe** method overloads, and **Publish**, **Unsubscribe**, and **Contains** methods.

Defining an event

A pub/sub event can be defined by creating an empty class that derives from the **PubSubEvent<TPayload>** class. The events in the AdventureWorks Shopper reference implementation do not pass a payload because the event handling only needs to know that the event occurred and then retrieve the updated state related to the event through a service. As a result, they declare the **TPayload** type as an **Object** and pass a **null** reference when publishing. The following code example shows how the **ShoppingCartUpdatedEvent** from AdventureWorks Shopper is defined.

C#: AdventureWorks.UILogic\Events\ShoppingCartUpdatedEvent.cs

```
public class ShoppingCartUpdatedEvent : PubSubEvent<object>
{
}
```

Publishing an event

Publishers notify subscribers of a pub/sub event by retrieving a singleton instance that represents the event from the **EventAggregator** class and calling the **Publish** method of that instance. The **EventAggregator** class constructs the instance on first access. The following code demonstrates publishing the **ShoppingCartUpdatedEvent**.

C#: AdventureWorks.UILogic\Repositories\ShoppingCartRepository.cs

```
private void RaiseShoppingCartUpdated()
{
    _eventAggregator.GetEvent<ShoppingCartUpdatedEvent>().Publish(null);
}
```

Subscribing to events

Subscribers can enlist with an event using one of the **Subscribe** method overloads available in the **PubSubEvent<TPayload>** class. There are several approaches to event subscription.

Default subscription

In the simplest case, the subscriber must provide a handler to be invoked whenever the pub/sub event is published. This is shown in the following code example.

C#: AdventureWorks.UILogic\ViewModels\ShoppingCartPageViewModel.cs

```
public ShoppingCartPageViewModel(...)
{
    ...
    eventAggregator.GetEvent<ShoppingCartUpdatedEvent>()
        .Subscribe(UpdateShoppingCartAsync);
    ...
}

public async void UpdateShoppingCartAsync(object notUsed)
{
    ...
}
```

In the code, the **ShoppingCartPageViewModel** class subscribes to the **ShoppingCartUpdatedEvent** using the **UpdateShoppingCartAsync** method as the handler.

Subscribing on the UI thread

A subscriber will sometimes need to update UI elements in response to events. In Windows Store apps, only the app's main thread can update UI elements.

By default, each subscribed handler action is invoked synchronously from the **Publish** method, in no defined order. If your handler action needs to be called from the UI thread, for example, in order to update UI elements, you can specify a **ThreadOption** when you subscribe. This is shown in the following code example.

C#: EventAggregatorQuickstart\ViewModels\SubscriberViewModel.cs

```
public SubscriberViewModel(IEventAggregator eventAggregator)
{
    ...
    _eventAggregator.GetEvent<ShoppingCartChangedEvent>()
        .Subscribe(HandleShoppingCartUpdate, ThreadOption.UIThread);
    ...
}
```

The **ThreadOption** enumeration allows three possible values:

- **PublisherThread**. This value should be used to receive the event on the publishers' thread, and is the default setting. The invocation of the handler action is synchronous.
- **BackgroundThread**. This value should be used to asynchronously receive the event on a thread-pool thread. The handler action is queued using a new task.
- **UIThread**. This value should be used to receive the event on the UI thread. The handler action is posted to the synchronization context that was used to instantiate the event aggregator.

Note For UI thread dispatching to work, the **EventAggregator** class must be created on the UI thread. This allows it to capture and store the [SynchronizationContext](#) that is used to dispatch to the UI thread for subscribers that use the **ThreadOption.UIThread** value.

In addition, it is not recommended that you modify the payload object from within a callback delegate because several threads could be accessing the payload object simultaneously. In this scenario you should have the payload be immutable to avoid concurrency errors.

Subscription filtering

A subscriber may not need to handle every instance of a published event. In this case, the subscriber can use a **Subscribe** method overload that accepts a **filter** parameter. The **filter** parameter is of type **System.Predicate<TPayload>** and is executed when the event is published. If the payload does satisfy the predicate, the subscriber callback is not executed. The **filter** parameter is shown in the following code example.

C#: EventAggregatorQuickstart\ViewModels\SubscriberViewModel.cs

```
public SubscriberViewModel(IEventAggregator eventAggregator)
{
    ...
    _eventAggregator.GetEvent<ShoppingCartChangedEvent>()
        .Subscribe(HandleShoppingCartUpdateFiltered, ThreadOption.UIThread, false,
            IsCartCountPossiblyTooHigh);
    ...
}
```

The **Subscribe** method returns a subscription token of type **Microsoft.Practices.Prism.PubSubEvents.SubscriptionToken** that can later be used to remove a subscription to the event. This token is useful if you are using anonymous delegates as the callback delegate or when you are subscribing to the same event handler with different filters.

Note The filter action is executed synchronously from the context of the **Publish** method regardless of the **ThreadOption** value of the current subscription.

Subscribing using strong references

The **PubSubEvent<TPayload>** class, by default, maintains a weak delegate reference to the subscriber's handler and any filter, on subscription. This means that the reference that the **PubSubEvent<TPayload>** class holds onto will not prevent garbage collection of the subscriber. Therefore, using a weak delegate reference relieves the subscriber from the need to unsubscribe from the event, and allows for garbage collection.

Maintaining a weak delegate reference has a slightly higher performance impact than using a corresponding strong delegate reference. If your app publishes many events in a very short period of time, you may notice a performance cost when using weak delegate references. However, for most apps the performance will not be noticeable. In the event of noticing a performance cost, you may need to subscribe to events by using strong delegate references instead. If you do use strong delegate references, your subscriber will need to unsubscribe from events when the subscription is no longer needed.

To subscribe with a strong delegate reference, use an overload of the **Subscribe** method that has the **keepSubscriberReferenceAlive** parameter, as shown in the following code example.

C#

```
public SubscriberViewModel(IEventAggregator eventAggregator)
{
    ...
    bool keepSubscriberReferenceAlive = true;
    _eventAggregator.GetEvent<ShoppingCartChangedEvent>()
        .Subscribe(HandleShoppingCartUpdateFiltered, ThreadOption.UIThread,
            keepSubscriberReferenceAlive);
    ...
}
```

The **keepSubscriberReferenceAlive** parameter is of type **bool**. When set to **true**, the event instance keeps a strong reference to the subscriber instance, thereby not allowing it to be garbage collected. For info about how to unsubscribe see "Unsubscribing from pub/sub events" in the following section. When set to **false**, which is the default value when the parameter is omitted, the event maintains a weak reference to the subscriber instance, thereby allowing the garbage collector to dispose the subscriber instance when there are no other references to it. When the subscriber instance is garbage collected, the event is automatically unsubscribed.

Unsubscribing from pub/sub events

If your subscriber no longer wants to receive events, you can unsubscribe by using your subscriber's handler or by using a subscription token. The following code example shows how to unsubscribe by using your subscriber's handler.

C#

```
ShoppingCartChangedEvent shoppingCartChangedEvent =  
    _eventAggregator.GetEvent<ShoppingCartChangedEvent>();  
shoppingCartChangedEvent.Subscribe(HandleShoppingCartUpdate,  
    ThreadOption.PublisherThread);  
...  
shoppingCartChangedEvent.Unsubscribe(HandleShoppingCartUpdate);
```

The following code example shows how to unsubscribe by using a subscription token. The token is supplied as a return value from the **Subscribe** method.

C#

```
ShoppingCartChangedEvent shoppingCartChangedEvent =  
    _eventAggregator.GetEvent<ShoppingCartChangedEvent>();  
subscriptionToken = shoppingCartChangedEvent.Subscribe(HandleShoppingCartUpdate,  
    ThreadOption.UIThread, false, IsCartCountPossiblyTooHigh);  
...  
shoppingCartChangedEvent.Unsubscribe(subscriptionToken);
```

Working with tiles in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Use live tiles to present engaging new content to users that invites them to launch the app.
- Use secondary tiles and deep links to promote specific content in your app.
- Use periodic notifications to update tiles on a fixed schedule.

Tiles represent your app on the Start screen and are used to launch your app. They have the ability to display a continuously changing set of content that can be used to keep users aware of events associated with your app when it's not running. When you use tiles effectively you can give your users a great first-impression of your Windows Store app. This article discusses how to create an app tile that is updated by periodic notifications, and how to create secondary tiles and deep links to promote specific content from the app onto the Start screen.

You will learn

- How to create and update an app tile with periodic notifications.
- How to pin and unpin secondary tiles to the Start screen from within an app.
- How to launch the app to a specific page from a secondary tile.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

A tile is an app's representation on the Start screen and allows you to present rich and engaging content to your users when the app is not running. Tiles should be appealing to users in order to give them great first-impression of your Windows Store app. The following list summarizes the decisions to make when creating tiles for your app:

- Why should I invest in a live tile?
- How do I make a live tile compelling to users?
- What shape should my tile be?
- What size should my tile image be?
- Which tile templates should I use?
- What mechanism should I use to deliver tile notifications?
- How often should my live tile content change?
- Should my app include the ability to pin secondary tiles to Start?

Tiles can be live, meaning they are updated through notifications, or static. For info about tiles, including why you should invest in a live tile, how to make a live tile compelling to users, what shape

and size a tile should be, which tile templates you should use, how often your live tile content should change, and secondary tiles, see [Guidelines and checklist for tiles and badges](#), [Tile and toast image sizes](#), [The tile template catalog](#), [Sending notifications](#), and [Secondary tiles overview](#).

The choice of which mechanism to use to deliver a tile notification depends on the content you want to show and how frequently that content should be updated. Local notifications are a good way to keep the app tile current, even if you also use other notification mechanisms. Many apps will use local notifications to update the tile when the app is launched or when state changes within the app. This ensures that the tile is up-to-date when the app launches and exits. Scheduled notifications are ideal for situations where the content to be updated is known in advance, such as a meeting invitation. Periodic notifications provide tile updates with minimal web or cloud service and client investment, and are an excellent method of distributing the same content to a wide audience. Push notifications are ideal for situations where your app has real-time data or data that is personalized for your user. Push notifications are also useful in situations where the data is time-sensitive, and where the content is generated at unpredictable times. Periodic notifications offer the most suitable notification solution for side-loaded apps, but don't provide notifications on demand. In addition, with periodic notifications, after the initial poll to the web or cloud service Windows will continue to poll for tile updates even if your app is never launched again. For more info see [Choosing a notification delivery method](#).

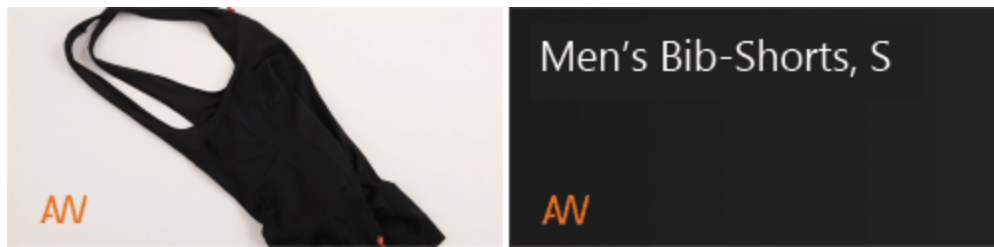
Note Push notifications use the Windows Push Notification Services (WNS) to deliver updates to users. Before you can send notifications using WNS, your app must be registered with the Windows Store Dashboard. For more info see [Push notification overview](#).

Tiles in AdventureWorks Shopper

The AdventureWorks Shopper reference implementation includes square and wide default tiles, which were created according to the pixel requirements for each. Choosing a small logo that represents your app is important so that users can identify it when the tile displays custom content. For more info see [Creating app tiles](#).

The default tiles are made live by updating them with periodic notifications, at 30 minute intervals, to advertise specific products to users on their Start screen. The periodic notifications use peek templates so that the live tile will animate between two frames. The first frame shows an image of the advertised product, with the second frame showing product details. Both wide and square peek tile templates are used. While AdventureWorks Shopper will default to the wide tile, it can be changed to the square tile by the user. For more info see [Using periodic notifications to update tile content](#).

AdventureWorks Shopper includes the ability to create secondary tiles by pinning specific products to the Start screen from the **ItemDetailPage**. The following diagram shows the two frames of a secondary tile created from one of the products sold in AdventureWorks Shopper.



Selecting a secondary tile launches the app and displays the previously pinned product on the **ItemDetailPage**. For more info see [Creating secondary tiles](#).

Creating app tiles

Tiles begin as a default tile defined in the app's manifest. A static tile will always display the default content, which is generally a full-tile logo image. A live tile can update the default tile to show new content, but can return to the default if the notification expires or is removed. The following diagrams shows the default small, square, and wide logo images that can be found in the **Assets** folder in the AdventureWorks Shopper VisualStudio solution. Each logo has a transparent background. This is particularly important for the small logo so that it will blend in with tile notification content.



30 x 30 pixels



150 x 150 pixels



310 x 150 pixels

Note Image assets, including the logos, are placeholders and meant for training purposes only. They cannot be used as a trademark or for other commercial purposes.

The Visual Studio manifest editor makes the process of adding the default tiles easy. For more info see [Quickstart: Creating a default tile using the Visual Studio manifest editor](#). For more info about working with image resources, see [Quickstart: Using file or image resources](#) and [How to name resources using qualifiers](#).

If only a square logo is provided in the app's manifest file, the app's tile will always be square. If both a square and a wide logo are provided in the manifest, the app's tile will default to a wide tile when it is installed. You must decide whether you want to allow a wide tile as well. This choice is made by providing a wide logo image when you define your default tile in your app manifest.

Using periodic notifications to update tile content

Periodic notifications, which are sometimes called polled notifications, update tiles at a fixed interval by downloading content directly from a web or cloud service. To use periodic notifications your app must specify the Uniform Resource Identifier (URI) of a web location that Windows polls for tile updates, and how often that URI should be polled.

Periodic notifications require that your app hosts a web or cloud service. Any valid HTTP or Secure Hypertext Transfer Protocol (HTTPS) web address can be used as the URI to be polled by Windows. The following code example shows the **GetTileNotification** method in the **TileNotificationController** class in the AdventureWorks.WebServices project, which is used to send tile content to the AdventureWorks Shopper reference implementation.

C#: AdventureWorks.WebServices\Controllers\TileNotificationController.cs

```
public HttpResponseMessage GetTileNotification()
{
    var tileXml =
        GetDefaultTileXml("http://localhost:2112/Images/hotrodbike_red_large.jpg",
            "Mountain-400-W Red, 42");
    tileXml = string.Format(CultureInfo.InvariantCulture, tileXml,
        DateTime.Now.ToShortDateString(), DateTime.Now.ToShortTimeString());

    // create HTTP response
    var response = new HttpResponseMessage();

    // format response
    response.StatusCode = System.Net.HttpStatusCode.OK;
    response.Content = new StringContent(tileXml);

    // Need to return xml format to TileUpdater.StartPeriodicUpdate
    response.Content.Headers.ContentType =
        new System.Net.Http.Headers.MediaTypeHeaderValue("text/xml");
    return response;
}
```

This method generates the XML tile content, formats it, and returns it as a HTTP response. The tile content must conform to the [Tile schema](#) and be 8-bit Unicode Transformation Format (UTF-8)

encoded. The tile content is specified using the **TileWidePeekImage01** and **TileSquarePeekImageAndText02** templates. This is necessary because while the app will use the wide tile by default, it can be changed to the square tile by the user. For more info see [The tile template catalog](#).

At a polling interval of 30 minutes, Windows sends an HTTP GET request to the URI, downloads the requested tile content as XML, and displays the content on the app's tile. This is accomplished by the **OnInitialize** method in the **App** class, as shown in the following code example.

C#: AdventureWorks.Shopper\App.xaml.cs

```
_tileUpdater = TileUpdateManager.CreateTileUpdaterForApplication();
_tileUpdater.StartPeriodicUpdate(new Uri(Constants.ServerAddress +
    "/api/TileNotification"), PeriodicUpdateRecurrence.HalfHour);
```

A new [TileUpdater](#) instance is created by the [CreateTileUpdaterForApplication](#) method in the [TileUpdateManager](#) class, in order to update the app tile. By default, a tile on the Start screen shows the content of a single notification until it is replaced by a new notification. However, you can enable notification cycling so that up to five notifications are maintained in a queue and the tile cycles through them. This is accomplished by calling the [EnableNotificationQueue](#) method with a parameter of **true**, on the **TileUpdater** instance. Finally, a call to [StartPeriodicUpdate](#) is made to poll the specified URI in order to update the tile with the received content. After this initial poll, Windows will continue to provide updates every 30 minutes, as specified. Polling then continues until you explicitly stop it, or your app is uninstalled. Otherwise Windows will continue to poll for updates to your tile even if your app is never launched again.

Note While Windows makes a best effort to poll as requested, the interval is not precise. The requested poll interval can be delayed by up to 15 minutes.

By default, periodic tile notifications expire three days from the time they are downloaded. Therefore, it is recommended that you set an expiration on all periodic tile notifications, using a time that makes sense for your app, to ensure that your tile's content does not persist longer than it's relevant. This also ensures the removal of stale content if your web or cloud service becomes unreachable, or if the user disconnects from the network for an extended period of time. This is accomplished by returning the X-WNS-Expires HTTP header to specify the expiration date and time.

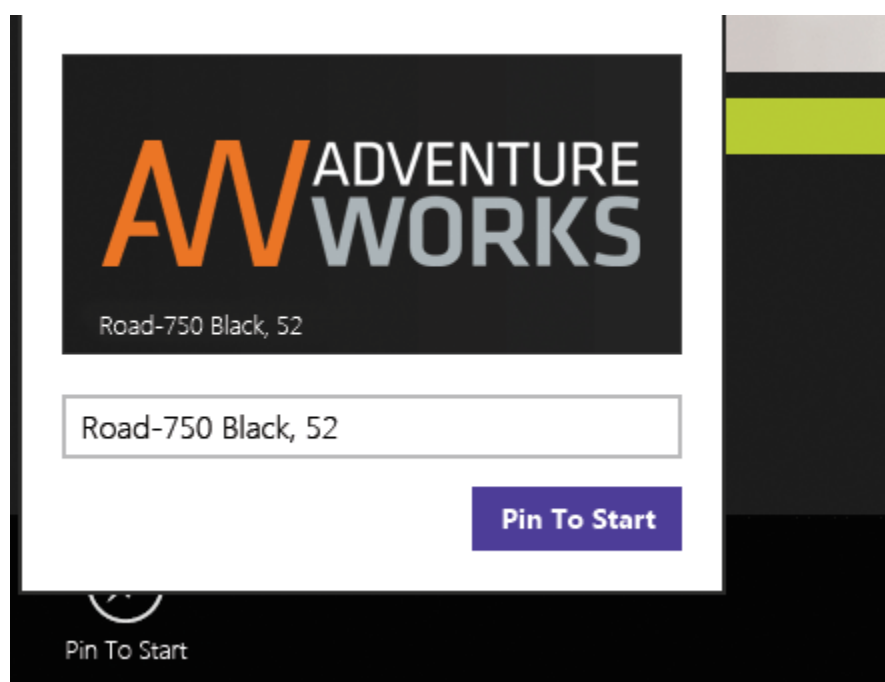
For more info see [Periodic notification overview](#), [Using the notification queue](#), and [Guidelines and checklist for periodic notifications](#).

Creating secondary tiles

A secondary tile allows a user to launch to a specific location in an app directly from the Start screen. Apps cannot pin secondary tiles programmatically without user approval. Users also have explicit control over secondary tile removal. This allows users to personalize their Start screen with the experiences that they use the most.

Secondary tiles are independent of the main app tile and can receive tile notifications independently. When a secondary tile is activated, an activation context is presented to the parent app so that it can launch in the context of the secondary tile.

The option to create a secondary tile is seen on the bottom app bar of the **ItemDetailPage** as the **Pin to Start** app bar button. This enables you to create a secondary tile for the product being displayed. Selecting the secondary tile launches the app and displays the previously pinned product on the **ItemDetailPage**. The following diagram shows an example of the Flyout that is displayed when you select the **Pin to Start** button. The Flyout shows a preview of the secondary tile, and asks you to confirm its creation.



Pinning and unpinning secondary tile functionality is provided by the **SecondaryTileService** class, which implements the **ISecundaryTileService** interface. In the **OnInitialize** method in the **App** class, the **SecondaryTileService** class is registered as a type mapping against the **ISecundaryTileService** type with the Unity dependency injection container. Then, when the **ItemDetailPageViewModel** class is instantiated, which accepts an **ISecundaryTileService** type, the Unity container will resolve the type and return an instance of the **SecondaryTileService** class.

The workflow AdventureWorks Shopper uses to pin a secondary tile to Start is as follows:

1. You invoke the **PinProductCommand** through the **Pin to Start** app bar button on the **ItemDetailPage**.

C#: AdventureWorks.UILogic\ViewModels\ItemDetailPageViewModel.cs

```
PinProductCommand = DelegateCommand.FromAsyncHandler(PinProduct,
    () => SelectedProduct != null);
```

2. AdventureWorks Shopper checks to ensure that the tile hasn't already been pinned by calling the **SecondaryTileExists** predicate in the **SecondaryTileService** instance.

C#: AdventureWorks.UILogic\ViewModels\ItemDetailPageViewModel.cs

```
bool isPinned = _secondaryTileService.SecondaryTileExists(tileId);
```

3. AdventureWorks Shopper calls the **PinWideSecondaryTile** method in the **SecondaryTileService** instance to create a secondary tile. The **SelectedProduct.ProductNumber** property is used as a unique ID.

C#: AdventureWorks.UILogic\ViewModels\ItemDetailPageViewModel.cs

```
isPinned = await _secondaryTileService.PinWideSecondaryTile(tileId,
    SelectedProduct.Title, SelectedProduct.Description,
    SelectedProduct.ProductNumber);
```

The **PinWideSecondaryTile** method creates a new instance of the [SecondaryTile](#) class, providing information such as the short name, the display name, the logo, and more.

C#: AdventureWorks.UILogic\Services\SecondaryTileService.cs

```
var secondaryTile = new SecondaryTile(tileId, shortName, displayName,
    arguments, TileOptions.ShowNameOnWideLogo, _squareLogoUri,
    _wideLogoUri);
```

4. The **RequestCreateAsync** method is called on the [SecondaryTile](#) instance to display a Flyout that shows a preview of the tile, asking you to confirm its creation.

C#: AdventureWorks.UILogic\Services\SecondaryTileService.cs

```
bool isPinned = await secondaryTile.RequestCreateAsync();
```

5. You confirm and the secondary tile is added to the Start screen.

The workflow AdventureWorks Shopper uses to unpin a secondary tile from Start is as follows:

1. AdventureWorks Shopper invokes the **UnpinProductCommand** through the **Unpin from Start** app bar button on the **ItemDetailPage**.

C#: AdventureWorks.UILogic\ViewModels\ItemDetailPageViewModel.cs

```
UnpinProductCommand = DelegateCommand.FromAsyncHandler(UnpinProduct,
    () => SelectedProduct != null);
```

2. AdventureWorks Shopper checks to ensure that the tile hasn't already been unpinned by calling the **SecondaryTileExists** predicate in the **SecondaryTileService** instance.

C#: AdventureWorks.UILogic\ViewModels\ItemDetailPageViewModel.cs

```
bool isPinned = _secondaryTileService.SecondaryTileExists(tileId);
```

3. AdventureWorks Shopper calls the **UnpinTile** method on the **SecondaryTileService** instance to remove the secondary tile. The tile can be identified by the **SelectedProduct.ProductNumber** property as the unique ID.

C#: AdventureWorks.UILogic\ViewModels\ItemDetailPageViewModel.cs

```
isPinned = (await _secondaryTileService.UnpinTile(tileId)) == false;
```

The **UnpinTile** method creates a new instance of the **SecondaryTile** class, using the **SelectedProduct.ProductNumber** property as the unique ID. By providing an ID for an existing secondary tile, the existing secondary tile will be overwritten.

C#: AdventureWorks.UILogic\Services\SecondaryTileService.cs

```
var secondaryTile = new SecondaryTile(tileId);
```

4. The **RequestDeleteAsync** method is called on the **SecondaryTile** instance to display a Flyout that shows a preview of the tile to be removed asking you to confirm its removal.

C#: AdventureWorks.UILogic\Services\SecondaryTileService.cs

```
bool isUnpinned = await secondaryTile.RequestDeleteAsync();
```

5. You confirm and the secondary tile is removed from the Start screen.

Note Secondary tiles can also be removed through the Start screen app bar. When this occurs the app is not contacted for removal information, the user is not asked for a confirmation, and the app is not notified that the tile is no longer present. Any additional cleanup action that the app would have taken in unpinning the tile must be performed by the app at its next launch.

For more info see [Secondary tiles overview](#) and [Guidelines and checklist for secondary tiles](#).

Launching the app from a secondary tile

Whenever the app is launched the **OnLaunched** method in the **MvvmAppBase** class is called (the **MvvmAppBase** class is provided by the [Microsoft.Practices.Prism.StoreApps](#) library). The [LaunchActivatedEventArgs](#) parameter in the **OnLaunched** method will contain the previous state of the app and the activation arguments. If the app is launched by its primary tile, the [TileId](#) property of the **LaunchActivatedEventArgs** parameter will have the same value as the application Id in the package manifest. If the app is launched by a secondary tile, the **TileId** property will have an ID that was specified when the secondary tile was created. The **OnLaunched** method in the **MvvmAppBase** class will call the **OnLaunchApplication** method in the **App** class only if the app is not resuming following suspension, or if the app was launched through a secondary tile. The **OnLaunchApplication** method, which is shown in the following code example, provides app specific launch behavior.

C#: AdventureWorks.Shopper\App.xaml.cs

```
protected override void OnLaunchApplication(LaunchActivatedEventArgs args)
{
    if (args != null && !string.IsNullOrEmpty(args.Arguments))
    {
        // The app was launched from a Secondary Tile
        // Navigate to the item's page
        NavigationService.Navigate("ItemDetail", args.Arguments);
    }
    else
    {
        // Navigate to the initial page
        NavigationService.Navigate("Hub", null);
    }
}
```

In this method the [LaunchActivatedEventArgs](#) parameter contains the previous state of the app and the activation arguments. If the app is being launched from the app tile then the activation [Arguments](#) property will not contain any data and so the **HubPage** will be navigated to. If the app is being launched from a secondary tile then the activation **Arguments** property will contain the product number of the product to be displayed. The **ItemDetailPage** will then be navigated to, with the product number being passed to the **OnNavigatedTo** override in the **ItemDetailPageViewModel** instance, so that the specified product is displayed.

Implementing search in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Respond to **OnQuerySubmitted** and **OnSearchApplication** notifications.
- Implement type to search for your app's hub, browse, and search pages.
- Save the search results page for the last query in case the app is activated to search for that query again.

To add search to your app you must participate in the Search contract. When you add the Search contract, users can search your app from anywhere in their system by selecting the Search charm. The AdventureWorks Shopper reference implementation uses the **SearchPaneService** class, provided by [Prism for the Windows Runtime](#), to represent and manage the search pane that opens when a user activates the Search charm in an app that uses the Model-View-ViewModel (MVVM) pattern.

You will learn

- How to use the Search contract to implement search functionality through the Search charm.
- How to respond to a search query while the app is the main app on screen.
- How to respond to a search query when the app is not the main app on screen.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

When you add search with the Search contract, users can search your app's content from anywhere in their system, through the Search charm. The following list summarizes the decisions to make when implementing search in your app:

- How should I include search functionality in my app?
- Should I add a search icon to the app canvas?
- Should I add a search box to the app canvas?
- What should I display on my search results page?

You should rely on the Search charm to let users search for content in your app, customizing it where necessary, in order to ensure that users have a consistent and predictable experience when they search and when they change search settings.

Regardless of where your app's content is located, you can use the Search charm to respond to user's queries and display search results in an app page of your own design. When a user selects the Search charm, a search pane opens with a search box where they can enter a query and a list of searchable apps is displayed. If your app is the main app on screen, it is automatically highlighted in the list of apps in the search pane. Otherwise, users can select the Search charm and then select your app from the list of apps in the search pane.

If users need search to get started using your app, add a search icon to your app canvas. A prominent search icon gives users a strong visual cue that tells them where to begin. When users select the icon, your app should open the Search charm programmatically so that users can enter their query using the search pane. Using the Search charm in this way helps make your app more intuitive and helps keep your app's search experience consistent with search in Windows 8 and other apps.

Note Snapped views do not need to have a search icon on the app canvas because searching automatically unsnaps the app.

If search is the primary purpose of your app and you want to show extensive suggestions, add a search box to your app canvas. A prominent search box helps indicate to users that your app specializes in search. When users enter a query, you can use your own custom layouts to display a greater number of more detailed suggestions on your app canvas. However, if you provide an in-app search box, the user might have two different search histories for your app — one history tied to the in-app search box, and another tied to the Search charm.

When users submit a search query to your app, they see a page that shows search results for the query. You design the search results page for your app, and so must ensure that the presented results are useful and have an appropriate layout. You should use a [ListView](#) or [GridView](#) control to display search results, and let users see their query text on the page. Also, you should indicate why a search result matches the query by highlighting the user's query in each result, which is known as hit highlighting. In addition, you should let users navigate back to the last-viewed page after they look at the details for a search result. This can be accomplished by including a back button in the app's UI. This back button should be used to go to the page that the user was interacting with before they submitted their search. However, if your app was activated through the Search contract, it will not have a page to navigate back to. In this scenario you must ensure that your app provides a mechanism to exit the search results page, such as a top app bar button that performs navigation.

For more info see [Guidelines and checklist for search](#).

Search in AdventureWorks Shopper

The AdventureWorks Shopper reference implementation uses the Search charm to respond to user's queries and display search results in an app page. When a user selects the Search charm, a search pane opens with a search box where they can enter a query. Search results are displayed using a [GridView](#) control, unless the app is snapped when a [ListView](#) control is used instead. The search results page includes the user's query text, hit highlighting to indicate why a search result matches

the query, and lets users navigate back to the last-viewed page. For more info see [Participating in the Search contract](#).

AdventureWorks Shopper includes a search icon on the app canvas for the **HubPage**, **GroupDetailPage**, and **ItemDetailPage**, as long as the app is not snapped. The search icon is prominently located next to the shopping cart icon, as shown in the following diagram. For more info see [Participating in the Search contract](#).



Note

AdventureWorks Shopper does not provide the following search functionality:

- Query and result suggestions on the Search charm.
- Filters and scope to refine the search results.

You should provide this functionality in your own apps to ensure that they are fully integrated with the Search contract. For more info see [Guidelines and checklist for search](#).

Participating in the Search contract

Windows Store apps use contracts and extensions to declare the interactions that they support with other apps. Apps must include required declarations in the package manifest and call required Windows Runtime APIs to communicate with Windows and other contract participants. A contract defines the requirements that apps must meet to participate in Windows interactions. When you participate in the Search contract, you agree to make your app's contract searchable by other participants and to present search results from those participants in your app. The advantage this offers is that it can help you to gain traffic and usage for your app.

To participate in the Search contract you should add the **Search Contract** template item to your project, from the center pane of the **Add a New Item** dialog. Visual Studio then customizes your project to fulfill the minimum requirements of the Search contract. This customization is required to ensure that your app is fully integrated with the Search charm, so that your users will have a positive experience when they search your app. The search contract allows users to invoke an application for search from anywhere in the system by using the Search charm. The AdventureWorks Shopper reference implementation will always respond to a search activation event by saving its state, thus allowing users to return to that previous state at a later time.

Note To check which contracts and extensions your app supports open the package manifest and select the **Declarations** tab.

The [SearchPane](#) class represents and manages the search pane that opens when a user activates the Search charm. AdventureWorks Shopper uses the **SearchPaneService** class, which is provided by the [Microsoft.Practices.Prism.StoreApps](#) library, as an abstraction of the **SearchPane** class. This is necessary in order to make the app testable, as the **SearchPane** class is a view dependency and should not be referenced directly from view model classes. In the **OnInitialize** method in the **App** class, the **SearchPaneService** class is registered as a type mapping against the **ISearchPaneService** type with the Unity dependency injection container. When any classes with a dependency on the **ISearchPaneService** type are instantiated, the Unity container will resolve the type and return an instance of the **SearchPaneService** class.

Placeholder text is shown in the search box in the search pane, to describe what users can search for in AdventureWorks Shopper. The text is only shown when the search box is empty, and is cleared if the user starts typing into the box. This is accomplished by setting the [PlaceholderText](#) property of the [SearchPane](#) class.

The **SearchQueryArguments** class, which is provided by the [Microsoft.Practices.Prism.StoreApps](#) library, is an abstraction of the [SearchPaneQuerySubmittedEventArgs](#) and the [SearchActivatedEventArgs](#) classes. This is required in order to have only one event handler that handles both of the search activation events. It has the added benefit of enabling testability, as the **SearchPaneQuerySubmittedEventArgs** and **SearchActivatedEventArgs** classes are view dependencies and should not be referenced directly from view model classes. For more info see [Responding to search queries](#).

The **SearchUserControl** class defines a search icon that's added to the app canvas on the **HubPage**, **GroupDetailPage**, and **ItemDetailPage**. When users select the search icon the Search charm is opened programmatically by the **ShowSearchPane** method in the **SearchUserControlViewModel** class, so that users can enter their query using the search pane.

The **SearchResultsPage** includes a back button and a top app bar that allows users to navigate to the **HubPage** and the **ShoppingCartPage**. If AdventureWorks Shopper is suspended while the **SearchResultsPage** is active, the app will correctly restore page state upon reactivation by using the [Microsoft.Practices.Prism.StoreApps](#) library. This includes the [GridView](#) scroll position, the user's query text, and the search results. This avoids the need to requery the data using the query text.

For more info see [Quickstart: Adding search to an app](#).

Responding to search queries

When the user searches AdventureWorks Shopper when it is the main app on screen, the system fires the [QuerySubmitted](#) event and stores the arguments for this event with an instance of the [SearchPaneQuerySubmittedEventArgs](#) class. The **OnQuerySubmitted** method in the **MvvmAppBase** class handles this event, and is shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```
private void OnQuerySubmitted(SearchPane sender, SearchPaneQuerySubmittedEventArgs args)
{
    var searchQueryArguments = new SearchQueryArguments(args);
    OnSearchApplication(searchQueryArguments);
}
```

This method responds to the [QuerySubmitted](#) event by displaying the search results page for the user's query. It does this by calling the **OnSearchApplication** method override in the **App** class, which is shown in the following code example.

C#: AdventureWorks.Shopper\App.xaml.cs

```
protected override void OnSearchApplication(SearchQueryArguments args)
{
    if (args != null && !string.IsNullOrEmpty(args.QueryText))
    {
        NavigationService.Navigate("SearchResults", args.QueryText);
    }
    else
    {
        NavigationService.Navigate("Hub", null);
    }
}
```

Users may select your app from the Search charm without entering query text. They may do this to scope their search to your app, or simply as a quick way to get back to what they were last looking at. Your app should anticipate the user's needs and respond differently in each case. This can be accomplished by checking the **QueryText** property of the **SearchQueryArguments** class in the **OnSearchApplication** method. Here, the **SearchResultsPage** will be navigated to provided that the **QueryText** property contains data. Otherwise the **HubPage** will be navigated to.

When the user searches AdventureWorks Shopper when it is not the main app on screen, the system fires the [Activated](#) event and stores the arguments for this event with an instance of the [SearchActivatedEventArgs](#) class. This **OnSearchActivated** method in the **MvvmAppBase** class handles this event, and is shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\MvvmAppBase.cs

```
protected async override void OnSearchActivated(SearchActivatedEventArgs args)
{
    // If the Window isn't already using Frame navigation, insert our own Frame
    var rootFrame = await InitializeFrameAsync(args);

    if (rootFrame != null)
    {
        var searchQueryArguments = new SearchQueryArguments(args);
```

```

        OnSearchApplication(searchQueryArguments);
    }

    // Ensure the current window is active
    Window.Current.Activate();
}

```

This method responds to the [Activated](#) event by displaying the search results page for the user's query. It does this by calling the **OnSearchApplication** method override in the **App** class.

Note The **OnSearchApplication** method in the **MvvmAppBase** class is **virtual**. This means that if your app does not implement the Search contract, this method does not need to be overridden.

For more info about activating an app see [App contracts and extensions](#) and [How to activate an app](#).

Populating the search results page with data

When users search AdventureWorks Shopper the **SearchResultsPage** is used to display search results. The **OnNavigatedTo** method in the **SearchResultsPageViewModel** class is used to populate the page with the search results, as shown in the following code example.

C#: AdventureWorks.UILogic\ViewModels\SearchResultsPageViewModel.cs

```

public async override void OnNavigatedTo(object navigationParameter,
    NavigationMode navigationMode, Dictionary<string, object> viewModelState)
{
    base.OnNavigatedTo(navigationParameter, navigationMode, viewModelState);
    var queryText = navigationParameter as String;
    string errorMessage = string.Empty;
    this.SearchTerm = queryText;
    this.QueryText = '\u201c' + queryText + '\u201d';

    try
    {
        ReadOnlyCollection<Product> products;
        if (queryText == PreviousSearchTerm)
        {
            products = PreviousResults;
        }
        else
        {
            var searchResults = await _productCatalogRepository
                .GetFilteredProductsAsync(queryText);
            products = searchResults.Products;
            TotalCount = searchResults.TotalCount;
            PreviousResults = products;
        }

        var productViewModels = new List<ProductViewModel>();
        foreach (var product in products)
    }

```

```

    {
        productViewModels.Add(new ProductViewModel(product));
    }

    // Communicate results through the view model
    this.Results =
        new ReadOnlyCollection<ProductViewModel>(productViewModels);
    this.NoResults = !this.Results.Any();

    // Update VM status
    PreviousSearchTerm = SearchTerm;
    _searchPaneService.ShowOnKeyboardInput(true);
}
catch (HttpRequestException ex)
{
    errorMessage = string.Format(CultureInfo.CurrentCulture,
        _resourceLoader.GetString("GeneralServiceErrorMessage"),
        Environment.NewLine, ex.Message);
}

if (!string.IsNullOrEmpty(errorMessage))
{
    await _alertMessageService.ShowAsync(errorMessage,
        _resourceLoader.GetString("ErrorServiceUnreachable"));
}
}

```

This method uses the **ProductCatalogRepository** instance to retrieve products from the web service if they match the **queryText** parameter, and store them in the **Results** property for display by the **SearchResultsPage**. If no results are returned by the **ProductCatalogRepository**, the **NoResults** property is set to **true** and the **SearchResultsPage** displays a message indicating that no products match your search. The method also saves the search results for the last query in case AdventureWorks Shopper is activated to search for that query again. This handles the scenario whereby the user might submit a search query to AdventureWorks Shopper and then switch to another app, searching it using the same query, and then come back to AdventureWorks Shopper to view the search results again. When this happens, AdventureWorks Shopper is activated for search again. If the current query is the same as the last query, it avoids retrieving a new set of search results, instead loading the previous search results.

For more info see [Guidelines and checklist for search](#).

Navigating to the result's detail page

To display detailed information about a user selected result the **ListViewItemClickedToAction** attached behavior binds the [ItemClick](#) event of the [GridView](#) in the **SearchResultsPage** to the **ProductNavigationAction** property in the **SearchResultsPageViewModel** class. So when a [GridViewItem](#) is selected the **ProductNavigationAction** is executed and calls the **NavigateToItem** method in the **SearchResultsPageViewModel** class, which is shown in the following code example.

C#: AdventureWorks.UILogic\ViewModels\SearchResultsPageViewModel.cs

```
private void NavigateToItem(object parameter)
{
    var product = parameter as ProductViewModel;
    if (product != null)
    {
        _navigationService.Navigate("ItemDetail", product.ProductNumber);
    }
}
```

This method navigates to the **ItemDetailPage**, with the product ID being passed to the **OnNavigatedTo** override in the **ItemDetailPageViewModel** instance, so that the specified product is displayed.

For more info about using an attached behavior to enable an event to be handled in a view model, rather than in a page's code-behind, see [Using the MVVM pattern](#).

Enabling users to type into the search box

When the user selects the search icon on the app canvas, the Search charm displays a search pane with a search box where they can enter a query. The AdventureWorks Shopper reference implementation also provides the ability to search for content in the app by typing directly into the search box of the Search charm, without selecting the Search charm first. This feature is known as *type to search*. Enabling type to search makes efficient use of keyboard interaction and makes the app's search experience consistent with the Start screen.

Type to search is enabled in AdventureWorks Shopper for the **HubPage**, **GroupDetailPage**, **ItemDetailPage**, and **SearchResultsPage**. When a view model's **OnNavigatedTo** method is executed the **ShowOnKeyboardInput** property in the **SearchPaneService** instance is set to **true** so that the search box receives input when users type. In turn, this sets the [ShowOnKeyboardInput](#) property in the [SearchPane](#) class to **true**.

C#: AdventureWorks.UILogic\ViewModels\HubPageViewModel.cs

```
_searchPaneService.ShowOnKeyboardInput(true);
```

Then, when a view model's **OnNavigatedFrom** method is executed, provided that the app isn't suspending, the **ShowOnKeyboardInput** property in the **SearchPaneService** instance is set to **false** so that the search box will not receive input when users type. In turn, this sets the [ShowOnKeyboardInput](#) property in the [SearchPane](#) class to **false**. The following code example shows the **OnNavigatedFrom** method.

C#: AdventureWorks.UILogic\ViewModels\HubPageViewModel.cs

```
public override void OnNavigatedFrom(Dictionary<string, object> viewModelState,
    bool suspending)
{
    base.OnNavigatedFrom(viewModelState, suspending);
    if (!suspending)
    {
        _searchPaneService.ShowOnKeyboardInput(false);
    }
}
```

If the app is suspending, the **OnNavigatedFrom** method should not set the **ShownOnKeyboardInput** property to **false**. This is because when the app suspends, the **OnNavigatedFrom** method is called. If the app resumes without being terminated, the corresponding **OnNavigatedTo** method is not called. Therefore, all **OnNavigatedFrom** methods feature a **suspending** parameter that tells the view model whether it is being suspended. If the parameter is **true** it means that no change should be made to state that would invalidate the page and that a subsequent **OnNavigatedTo** method might not be called.

In addition, the AdventureWorks Shopper disables type to search before showing a Flyout, and restores it when the Flyout closes. This is accomplished in the **Open** and **OnPopupClosed** methods in the **FlyoutView** class, respectively.

For more info see [Guidelines and checklist for search](#).

Improving performance in AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Plan for performance and measure it early and throughout the lifecycle of your project.
- Use asynchronous APIs that execute in the background and inform the app when they've completed.
- Use performance tools to measure, evaluate, and target performance-related issues in your app.

Users of Windows Store apps expect their apps to remain responsive and feel natural when they use them. In this article we discuss general performance best practices for the AdventureWorks Shopper reference implementation.

You will learn

- The differences between performance and perceived performance.
- Guidelines that help to create a well-performing, responsive app.
- Recommended strategies for profiling an app.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Users have a number of expectations for apps. They want immediate responses to touch, clicks, and key presses. They expect animations to be smooth. They expect that they'll never have to wait for the app to catch up with them. Performance problems show up in various ways. They can reduce battery life, cause panning and scrolling to lag behind the user's finger, or make the app appear unresponsive for a period of time. The following list summarizes the decisions to make when planning a well-performing, responsive app:

- Should I optimize actual app performance or perceived app performance?
- What performance tools should I use to discover performance-related issues?
- When should I take performance measurements?
- What devices should I take performance measurements on?
- Do I need to completely understand the platform to determine where to improve app performance?

Optimizing performance is more than just implementing efficient algorithms. Another way to think about performance is to consider the user's perception of app performance. The user's app experience can be separated into three categories—perception, tolerance, and responsiveness.

- **Perception.** User perception of performance can be defined as how favorably they recall the time it took to perform their tasks within the app. This perception doesn't always match reality. Perceived performance can be improved by reducing the amount of time between activities that the user needs to perform to accomplish a task, and by allowing computationally intensive operations to execute without blocking the user from performing other activities.
- **Tolerance.** A user's tolerance for delay depends on how long the user expects an operation to take. For example, a user might find sending data to a web service intolerable if the app becomes unresponsive during this process, even for a few seconds. You can increase a user's tolerance for delay by identifying tasks in your app that require substantial processing time and limiting or eliminating user uncertainty during those tasks by providing a visual indication of progress. And you can use async APIs to avoid making the app appear frozen.
- **Responsiveness.** Responsiveness of an app is relative to the activity being performed. To measure and rate the performance of an activity, you must have a time interval to compare it against. We used the guideline that if an activity takes longer than 500 milliseconds, the app might need to provide feedback to the user in the form of a visual indication of progress.

Therefore, both actual app performance and perceived app performance should be optimized in order to deliver a well-performing, responsive app.

One technique for determining where code optimizations have the greatest effect in reducing performance problems is to perform app profiling. The profiling tools for Windows Store apps enable you to measure, evaluate, and find performance-related issues in your code. The profiler collects timing information for apps by using a sampling method that collects CPU call stack information at regular intervals. Profiling reports display information about the performance of your app and help you navigate through the execution paths of your code and the execution cost of your functions so that you can find the best opportunities for optimization. For more info see [How to profile Visual C++, Visual C#, and Visual Basic code in Windows Store apps on a local machine](#). To learn how to analyze the data returned from the profiler see [Analyzing performance data for Visual C++, Visual C#, and Visual Basic code in Windows Store apps](#). In addition to using profiling tools to measure app performance, we also used PerfView and Windows Performance Analyzer (WPA). PerfView is a performance analysis tool that helps isolate CPU and memory-related performance issues. WPA is a set of performance monitoring tools used to produce performance profiles of apps. We used both of these tools for a general diagnosis of the app's performance. For more info about PerfView see [PerfView Tutorial](#). For more info about WPA see [Windows Performance Analyzer](#).

Measuring your app's performance during the early stages of development can add enormous value to your project. We recommend that you measure performance as soon as you have code that performs meaningful work. Early measurements give you a good idea of where the high costs in your app are, and can inform design decisions. It can be very costly to change design decisions later on in the project. Measuring performance late in the product cycle can result in last minute changes and poor performance. For more info see [General best practices for performance](#).

At a minimum, take performance measurements on hardware that has the lowest anticipated specifications. Windows 8 runs on a wide variety of devices, and taking performance measurements on one type of device won't always show the performance characteristics of other form factors.

You don't need to completely understand the platform to determine where you might need to improve performance. By knowing what parts of your code execute most frequently, you can determine the best places to optimize your app.

Performance considerations

A well-performing app responds to user actions quickly, and with no noticeable delay. We spent much time learning what works and what doesn't work when creating a responsive Windows Store app. Here are some things to remember:

- Limit the startup time.
- Emphasize responsiveness.
- Trim resource dictionaries
- Optimize the element count.
- Reuse identical brushes.
- Use independent animations.
- Minimize the communication between the app and the web service.
- Limit the amount of data downloaded from the web service.
- Use UI virtualization.
- Avoid unnecessary termination.
- Keep your app's memory usage low when it's suspended.
- Reduce battery consumption.
- Minimize the amount of resources that your app uses.
- Limit the time spent in transition between managed and native code.
- Reduce garbage collection time.

Limit the startup time

It's important to limit how much time the user spends waiting while your app starts. There are a number of techniques you can use to do this:

- You can dramatically improve the loading time of an app by packing its contents locally, including XAML, images, and any other important resources. If an app needs a particular file at initialization, you can reduce the overall startup time by loading it from disk instead of retrieving it remotely.
- You should only reference assemblies that are necessary to the launch of your app in startup code so that the common language runtime (CLR) doesn't load unnecessary modules.
- Defer loading large in-memory objects while the app is activating. If you have large tasks to complete, provide a custom splash screen so that your app can accomplish these tasks in the background.

In addition, apps have different startup performance at first install and at steady state. When your app is first installed on a user's machine, it is executed using the CLR's just-in-time (JIT) compiler. This means that the first time a method is executed it has to wait to be compiled. Later, a pre-compilation service pre-compiles all of the modules that have been loaded on a user's machine, typically within 24 hours. After this service has run most methods no longer need to be JIT compiled, and your app benefits from an improved startup performance. For more info see [Minimize startup time](#).

Emphasize responsiveness

Don't block your app with synchronous APIs, because if you do the app can't respond to new events while the API is executing. Instead, use asynchronous APIs that execute in the background and inform the app when they've completed by raising an event. For more info see [Keep the UI thread responsive](#).

Trim resource dictionaries

App-wide resources should be stored in the **Application** object to avoid duplication, but if you use a resource in a single page that is not the initial page, put the resource in the resource dictionary of that page. This reduces the amount of XAML the framework parses when the app starts. For more info see [Optimize loading XAML](#).

Optimize the element count

The XAML framework is designed to display thousands of objects, but reducing the number of elements on a page will make your app render faster. You can reduce a page's element count by avoiding unnecessary elements, and collapsing elements that aren't visible. For more info see [Optimize loading XAML](#).

Reuse identical brushes

Create commonly used brushes as root elements in a resource dictionary, and then refer to those objects in templates as needed. XAML will be able to use the same objects across the different templates and memory consumption will be less than if the brushes were duplicated in templates. For more info see [Optimize loading XAML](#).

Use independent animations

An independent animation runs independently from the UI thread. Many of the animation types used in XAML are composed by a composition engine that runs on a separate thread, with the engine's work being offloaded from the CPU to the graphics processing unit (GPU). Moving animation composition to a non-UI thread means that the animation won't jitter or be blocked by the app working on the UI thread. Composing the animation on the GPU greatly improves performance, allowing animations to run at a smooth and consistent frame rate.

You don't need additional markup to make your animations independent. The system determines when it's possible to compose the animation independently, but there are some limitations for independent animations. For more info see [Make animations smooth](#).

Minimize the communication between the app and the web service

In order to reduce the interaction between the AdventureWorks Shopper reference implementation and its web service as much data as possible is retrieved in a single call. For example, instead of retrieving product categories in one web service call, and then retrieving products for a category in a second web service call, AdventureWorks Shopper retrieves a category and its products in a single web service call.

In addition, the AdventureWorks Shopper reference implementation uses the **TemporaryFolderCacheService** class to cache data from the web service to the temporary app data store. This helps to minimize the communication between the app and the web service, provided that the cached data isn't stale. For more info see [Caching data](#).

Limit the amount of data downloaded from the web service

The **GetRootCategoriesAsync** method in **ProductCatalogRepository** class retrieves data for display on the **HubPage**, as shown in the following code example.

C#: AdventureWorks.UILogic\ViewModels\HubPageViewModel.cs

```
rootCategories = await _productCatalogRepository.GetRootCategoriesAsync(5);
```

The call to the **GetRootCategoriesAsync** method specifies the maximum amount of products to be returned by each category. This parameter can be used to limit the amount of data downloaded from the web service, by avoiding returning an indeterminate number of products for each category. For more info see [Consuming the data](#).

Use UI virtualization

UI virtualization enables controls that derive from [ItemsControl](#) (that is, controls that can be used to present a collection of items) to only load into memory those UI elements that are near the viewport, or visible region of the control. As the user pans through the collection, elements that were previously near the viewport are unloaded from memory and new elements are loaded.

Controls that derive from [ItemsControl](#), such as [ListView](#) and [GridView](#), perform UI virtualization by default. XAML generates the UI for the item and holds it in memory when the item is close to being visible on screen. When the item is no longer being displayed, the control reuses that memory for another item that is close to being displayed.

If you restyle an [ItemsControl](#) to use a panel other than its default panel, the control continues to support UI virtualization as long as it uses a virtualizing panel. Standard virtualizing panels include

[WrapGrid](#) and [VirtualizingStackPanel](#). Using standard non-virtualizing panels, which include [VariableSizedWrapGrid](#) and [StackPanel](#), disables UI virtualization for that control.

UI virtualization is not supported for grouped data. If performance is an issue, limit the size of your groups or if you have lots of items in a group, use another display strategy for group detail views like [SemanticZoom](#).

In addition, make sure that the UI objects that are created are not overly complex. As items come into view, the framework must update the elements in cached item templates with the data of the items coming onto the screen. Reducing the complexity of those XAML trees can pay off both in the amount of memory needed to store the elements and the time it takes to data bind and propagate the individual properties within the template. This reduces the amount of work that the UI thread must perform, which helps to ensure that items appear immediately in a collection that a user pans through. For more info see [Load, store, and display large sets of data efficiently](#).

Avoid unnecessary termination

An app can be suspended when the user moves it to the background or when the system enters a low power state. When the app is being suspended, it raises the [Suspending](#) event and has up to 5 seconds to save its data. If the app's **Suspending** event handler doesn't complete within 5 seconds, the system assumes that the app has stopped responding and terminates it. A terminated app has to go through the startup process again instead of being immediately loaded into memory when a user switches to it.

The AdventureWorks Shopper reference implementation saves page state while navigating away from a page, rather than saving all page state on suspension. This reduces the amount of time that it takes to suspend the app, and hence reduces the chance of the system terminating the app during suspension. In addition, AdventureWorks Shopper does not use page caching. This prevents views that are not currently active from consuming memory, which would increase the chance of termination when suspended. For more info see [Minimize suspend/resume time](#) and [Handling suspend, resume and activation](#).

Keep your app's memory usage low when it's suspended

When your app resumes from suspension, it reappears nearly instantly. But when your app restarts after being closed, it might take longer to appear. So preventing your app from being closed when it's suspended can help to manage the user's perception and tolerance of app responsiveness.

When your app begins the suspension process, it should free any large objects that can be easily rebuilt when it resumes. Doing so helps to keep your app's memory footprint low, and reduces the likelihood that Windows will terminate your app after suspension. For more info see [Minimize suspend/resume time](#) and [Handling suspend, resume and activation](#).

Reduce battery consumption

The CPU is a major consumer of battery power on devices, even at low utilization. Windows 8 tries to keep the CPU in a low power state when it is idle, but activates it as required. While most of the performance tuning that you undertake will naturally reduce the amount of power that your app consumes, you can further reduce your app's consumption of battery power by ensuring that it doesn't unnecessarily poll for data from web services and sensors. For more info see [General best practices for performance](#).

Minimize the amount of resources that your app uses

Windows has to accommodate the resource needs of all Windows Store apps by using the Process Lifetime Management (PLM) system to determine which apps to close in order to allow other apps to run. A side effect of this is that if your app requests a large amount of memory, other apps might be closed, even if your app then frees that memory soon after requesting it. Minimize the amount of resources that your app uses so that the user doesn't begin to attribute any perceived slowness in the system to your app. For more info see [Improve garbage collection performance](#) and [Garbage Collection and Performance](#).

Limit the time spent in transition between managed and native code

Most of the Windows Runtime APIs are implemented in native code. This has an implication for Windows Store apps written in managed code, because any Windows Runtime invocation requires that the CLR transitions from a managed stack frame to a native stack frame and marshals function parameters to representations accessible by native code. While this overhead is negligible for most apps, if you make many calls to Windows Runtime APIs in the critical path of an app, this cost can become noticeable. Therefore, you should try to ensure that the time spent in transition between languages is small relative to the execution of the rest of your code.

The [.NET for Windows Store apps](#) types don't incur this interop cost. You can assume that types in namespace which begin with "Windows." are part of the Windows Runtime, and types in namespace which begin with "System." are .NET types.

If your app is slow because of interop overhead, you can improve its performance by reducing calls to Windows Runtime APIs on critical code paths. For example, if a collection is frequently accessed, then it is more efficient to use a collection from the [System.Collections](#) namespace, rather than a collection from the [Windows.Foundation.Collections](#) namespace. For more info see [Keep your app fast when you use interop](#).

Reduce garbage collection time

Windows Store apps written in managed code get automatic memory management from the .NET garbage collector. The garbage collector determines when to run by balancing the memory consumption of the managed heap with the amount of work a garbage collection needs to do. Frequent garbage collections can contribute to increased CPU consumption, and therefore increased power consumption, longer loading times, and decreased frame rates in your app.

If you have an app with a managed heap size that's substantially larger than 100MB, you should attempt to reduce the amount of memory you allocate directly in order to reduce the frequency of garbage collections. For more info see [Improve garbage collection performance](#).

Additional considerations

When profiling your app, follow these guidelines to ensure that reliable and repeatable performance measurements are taken:

- Make sure that you profile the app on the device that's capturing performance measurements when it is plugged in, and when it is running on a battery. Many systems conserve power when running on a battery, and so operate differently.
- Make sure that the total memory use on the system is less than 50 percent. If it's higher, close apps until you reach 50 percent to make sure that you're measuring the impact of your app, rather than that of other processes.
- When you remotely profile an app, we recommend that you interact with the app directly on the remote device. Although you can interact with an app via Remote Desktop Connection, doing so can significantly alter the performance of the app and the performance data that you collect. For more info, see [How to profile Visual C++, Visual C#, and Visual Basic code in Windows Store apps on a remote device](#).
- To collect the most accurate performance results, profile a release build of your app. See [How to: Set Debug and Release Configurations](#).
- Avoid profiling your app in the simulator because the simulator can distort the performance of your app.

Testing and deploying AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

Summary

- Use multiple modes of testing for best results.
- Use unit tests and integration tests to identify bugs at their source.
- Test asynchronous functionality by creating a mock version of the instance that the class to be tested depends on, and specify an asynchronous delegate in the unit test that will be executed by the asynchronous method in the mock object.

Windows Store apps should undergo various modes of testing in order to ensure that reliable, high quality apps result. For the AdventureWorks Shopper reference implementation we performed unit testing, integration testing, user interface testing, suspend and resume testing, security testing, localization testing, accessibility testing, performance testing, device testing, and validation of the app user experience against the user experience guidelines on the Windows Dev Center.

You will learn

- How the various modes of testing contribute to the reliability and correctness of an app.
- How to use the testing tools that are available in Microsoft Visual Studio 2012.
- How to test asynchronous functionality in automated tests.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Making key decisions

Testing helps to ensure that an app is reliable, correct, and of high quality. The following list summarizes the decisions to make when testing a Windows Store app:

- How should I test the app?
- How should I deploy the app?
- How can I test the app for compliance with the Windows Store certification requirements?
- How should I manage the app after deployment?

You can test your app in many ways including unit testing, integration testing, user interface testing, suspend and resume testing, security testing, localization testing, accessibility testing, performance testing, device testing, and validation of the app user experience against the user experience guidelines on the Windows Dev Center. For more info see [Testing AdventureWorks Shopper](#).

While you can use the Windows Store to market and distribute apps, business apps will often be distributed directly to the end-user by the IT organization within a company. For more info see [Deploying and managing Windows Store apps](#).

Regardless of how your app will be deployed, you should validate and test it by using the Windows App Certification Kit. The kit performs a number of tests to verify that your app meets certification requirements for the Windows Store. In addition, as you plan your app, you should create a publishing-requirements checklist to use when you test your app. For more info see [Testing your app with the Windows App Certification Kit](#) and [Creating a Windows Store certification checklist](#).

Tools such as Windows Intune and System Center Configuration Manager can be used to manage access to business apps. In addition, IT staff can control the availability and functionality of the Windows Store to client computers based on the business policies of their environment. For more info see [Deploying and managing Windows Store apps](#).

Testing AdventureWorks Shopper

The AdventureWorks Shopper reference implementation was designed for testability, with the following modes of testing being performed:

- **Unit testing** tests individual methods in isolation. The goal of unit testing is to check that each unit of functionality performs as expected so that errors don't propagate throughout the app. Detecting a bug where it occurs is more efficient than observing the effect of a bug indirectly at a secondary point of failure. For more info see [Unit and integration testing](#).
- **Integration testing** verifies that the components of an app work together correctly. Integration tests examine app functionality in a manner that simulates the way the app is intended to be used. Normally, an integration test will drive the layer just below the user interface. In the AdventureWorks Shopper reference implementation, you can recognize this kind of test because it invokes methods of the view model. The separation of views from the view model makes integration testing possible. For more info see [Unit and integration testing](#).
- **User interface (UI) testing** involves direct interaction with the user interface. This type of testing often needs to be performed manually. Automated integration tests can be substituted for some UI testing but can't eliminate it completely.
- **Suspend and resume testing** ensures that your app behaves as expected when Windows suspends or resumes it, or activates it after a suspend and shutdown sequence. For more info see [Suspend and resume testing](#).
- **Security testing** focuses on potential security issues. It's based on a threat model that identifies possible classes of attack. For more info see [Security testing](#).
- **Localization testing** makes sure that an app works in all language environments. For more info see [Localization testing](#).
- **Accessibility testing** makes sure that an app supports touch, pointer, and keyboard navigation. It also makes sure that different screen configurations and contrasts are supported, and that the contents of the screen can be read with Windows Narrator. For more info see [Accessibility testing](#).

- **Performance testing** identifies how an app spends its time when it's running. In many cases, performance testing can locate bottlenecks or methods that take a large percentage of an app's CPU time. For more info see [Performance testing](#).
- **Device testing** ensures than app works properly on the range of hardware that it supports. For example, it's important to test that an app works with various screen resolutions and touch-input capabilities. For more info see [Device testing](#).

For more info on test automation, see [Testing for Continuous Delivery with Visual Studio 2012](#).

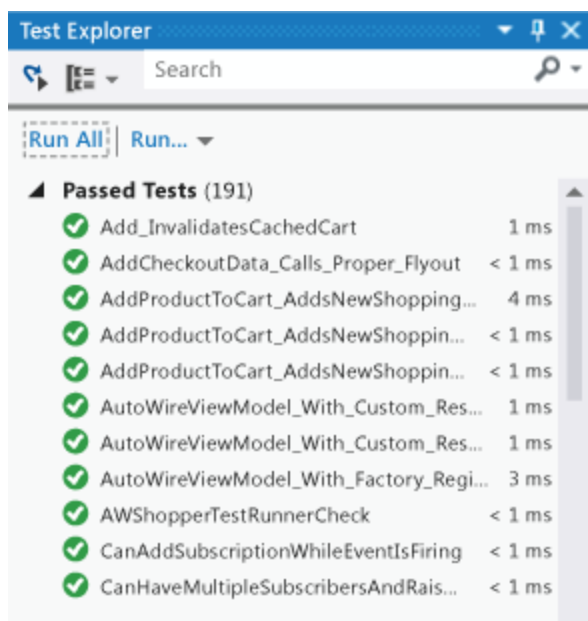
Unit and integration testing

You should expect to spend about the same amount of time writing unit and integration tests as you do writing the app's code. The effort is worth the work because it results in much more stable code that has fewer bugs and requires less revision.

In the AdventureWorks Shopper reference implementation we used the Model-View-ViewModel (MVVM) pattern to separate the concerns of presentation, presentation logic, and model. The MVVM pattern makes it easier to maintain and test your Windows Store app, especially as it grows. For more info see [Using the MVVM pattern](#).

The AdventureWorks.Shopper.Tests, AdventureWorks.UILogic.Tests, AdventureWorks.WebServices.Tests, Microsoft.Practices.Prism.PubSubEvents.Tests, and Microsoft.Practices.Prism.StoreApps.Tests projects of the AdventureWorks Shopper Visual Studio solution contain all the code that supports testing the [Microsoft.Practices.Prism.PubSubEvents](#) and [Microsoft.Practices.Prism.StoreApps](#) libraries, and the AdventureWorks Shopper reference implementation. The AdventureWorks.WebServices.Tests project uses the Microsoft.VisualStudio.QualityTools.UnitTestFramework, with the remaining test projects using the MsTestFramework for Windows Store apps. Test methods can be identified by the [TestMethod](#) attribute above the method name.

You can examine the unit tests by opening the AdventureWorks Shopper Visual Studio solution. On the menu bar, choose **Test > Windows > Test Explorer**. The **Test Explorer** window lists all of the AdventureWorks Shopper unit tests, as shown in the following diagram.



Unit tests should focus on how the code under test functions in response to values returned by dependent objects. A good approach to increase software testability is to isolate dependent objects and have them passed into your business logic using an abstraction such as an interface. This approach allows the dependent object to be passed into the business logic at run time. In addition, in the interests of testability, it allows a mock version of the dependent object to be passed in at test time. By using mocks, the return values or exceptions to be thrown by mock instances of dependent objects can easily be controlled.

Testing synchronous functionality

Synchronous functionality can easily be tested by unit tests. The following code example shows the **Validation_Of_Field_When_Valid_Should_Succeed** test method that demonstrates testing synchronous functionality. The unit test verifies that the **BindableValidator** class can successfully validate the value of the **Title** property in the **MockModelWithValidation** class.

C#: Microsoft.Practices.Prism.StoreApps.Tests\BindableValidatorFixture.cs

```
[TestMethod]
public void Validation_Of_Field_When_Valid_Should_Succeed()
{
    var model = new MockModelWithValidation() { Title = "A valid Title" };
    var target = new BindableValidator(model);

    bool isValid = target.ValidateProperty("Title");

    Assert.IsTrue(isValid);
    Assert.IsTrue(target.GetAllErrors().Values.Count == 0);
}
```

This method creates instances of the **MockModelWithValidation** and the **BindableValidator** classes. The **BindableValidator** instance is used to validate the contents of the **Title** property in the

MockModelWithValidation instance by calling the **ValidateProperty** method on the **BindableValidator** instance. The unit test passes if the **ValidateProperty** method returns true, and the **BindableValidator** instance has no errors.

For more info about validation, see [Validating user input](#).

Testing asynchronous functionality

Asynchronous functionality can be tested by creating a mock version of the dependent service that has an asynchronous method, and specifying an asynchronous delegate in the unit test that will be executed by the asynchronous method in the mock object. The following code example shows the **OnNavigatedTo_Fill_Root_Categories** test method, which demonstrates testing asynchronous functionality. The unit test verifies that when the hub page is navigated to the **RootCategories** property of the **HubPageViewModel** class will contain three categories.

C#: AdventureWorks.UILogic.Tests\ViewModels\HubPageViewModelFixture.cs

```
[TestMethod]
public void OnNavigatedTo_Fill_RootCategories()
{
    var repository = new MockProductCatalogRepository();
    var navigationService = new MockNavigationService();
    var searchPaneService = new MockSearchPaneService();

    repository.GetRootCategoriesAsyncDelegate = (maxAmountOfProducts) =>
    {
        var categories = new ReadOnlyCollection<Category>(new List<Category>{
            new Category(),
            new Category(),
            new Category()
        });

        return Task.FromResult(categories);
    };

    var viewModel = new HubPageViewModel(repository, navigationService, null,
        null, searchPaneService);
    viewModel.OnNavigatedTo(null, NavigationMode.New, null);

    Assert.IsNotNull(viewModel.RootCategories);
    Assert.AreEqual(((ICollection<CategoryViewModel>)viewModel.RootCategories)
        .Count, 3);
}
```

The method creates instances of the mock classes that are required to create an instance of the **HubPageViewModel** class. The **GetRootCategoriesAsyncDelegate**, when executed, returns a [Task](#) of type [ReadOnlyCollection](#) with three **Category** objects. An instance of the **HubPageViewModel** class is then created, with its **OnNavigatedTo** method being called. The **OnNavigatedTo** method calls the **GetRootCategoriesAsync** method, in this case on the **MockProductCatalogRepository** instance,

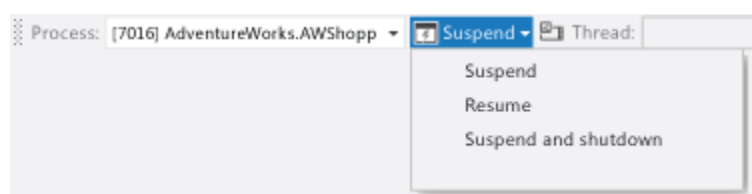
which in turn executes the **GetRootCategoriesAsyncDelegate**. The result of this is that the **RootCategories** property of the **HubPageViewModel** instance is populated with the data returned by the **GetRootCategoriesAsyncDelegate**. The unit test passes if the **RootCategories** property contains three items of data.

Note If you use the [await](#) operator in a test method, the test method must return a [Task](#) and use the [async](#) modifier in its method signature.

For more info about the unit testing tools in Visual Studio, see [Verifying Code by Using Unit Tests](#).

Suspend and resume testing

When you debug a Windows Store app, the **Debug Location** toolbar contains a drop-down menu that enables you to suspend, resume, or suspend and shut down (terminate) the running app. You can use this feature to ensure that your app behaves as expected when Windows suspends or resumes it, or activates it after a suspend and shutdown sequence. The following diagram shows the drop-down menu that enables you to suspend the running app.



If you want to demonstrate suspending from the debugger, run AdventureWorks Shopper in the Visual Studio debugger and set breakpoints in the **MvvmAppBase.OnSuspending** and **MvvmAppBase.InitializeFrameAsync** methods. Then select **Suspend and shutdown** from the **Debug Location** toolbar. The app will exit. Restart the app in the debugger, and the app will follow the code path for resuming from the **Terminated** state. In AdventureWorks Shopper, this logic is in the **MvvmAppBase.InitializeFrameAsync** method. For more info see [Guidelines for app suspend and resume](#) and [Handling suspend, resume, and activation](#).

Security testing

We used the STRIDE methodology for threat modeling as a basis for security testing in AdventureWorks Shopper. For more info see [Uncover Security Design Flaws Using The STRIDE Approach](#) and [Windows security features test](#).

Localization testing

We used the Multilingual App Toolkit to provide pseudo-localized versions of AdventureWorks Shopper for localization testing. For more info see [How to use the Multilingual App Toolkit](#), [Guidelines and checklist for app resources](#), and [Guidelines and checklist for globalizing your app](#).

Accessibility testing

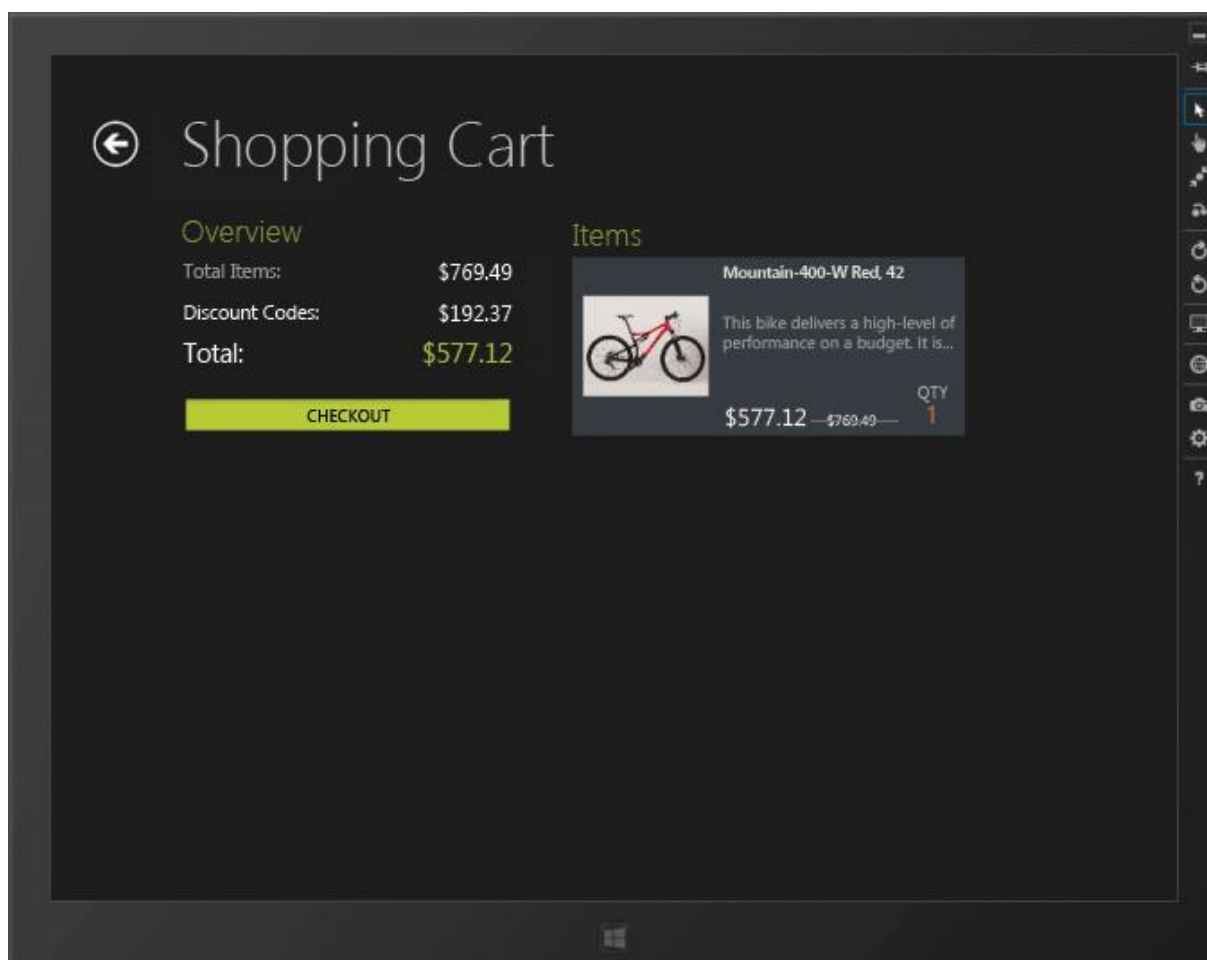
We used a number of testing tools to verify the accessibility of AdventureWorks Shopper, including [Windows Narrator](#), [Inspect](#), [UI Accessibility Checker](#), [UI Automation Verify](#), and [Accessible Event Watcher](#). For more info see [Testing your app for accessibility](#) and [Design for accessibility](#).

Performance testing

In addition to using profiling tools to measure app performance, we also used the Windows Performance Toolkit (WPT). WPT can be used to examine app performance, both in real time and by collecting log data for later analysis. We used this tool for a general diagnosis of the app's performance. For more info see [Windows Performance Toolkit Technical Reference](#), [General best practices for performance](#), and [Performance best practices for Windows Store apps using C++, C#, and Visual Basic](#).

Device testing

Visual Studio includes a simulator that you can use to run your Windows Store app in various device environments. For example, you can use the simulator to check whether your app works correctly with a variety of screen resolutions and with a variety of input hardware. You can simulate touch gestures even if you're developing the app on a computer that doesn't support touch. The following diagram shows AdventureWorks Shopper running in the simulator.



To start the simulator, click **Simulator** in the drop-down menu on the **Debug** toolbar in Visual Studio. The other choices in this drop-down menu are **Local Machine** and **Remote Machine**.

In addition to using the simulator, we also tested AdventureWorks Shopper on a variety of hardware. You can use remote debugging to test your app on a device that doesn't have Visual Studio installed on it. For more info see [Running Windows Store apps on a remote machine](#), [Testing Windows Store apps Running on a Device Using the Exploratory Test Window](#), and [Testing Windows Store apps Running on a Device Using Microsoft Test Runner](#).

Testing your app with the Windows App Certification Kit

Regardless of how your app will be deployed, you should validate and test it by using the Windows App Certification Kit. The kit performs a number of tests to verify that your app meets certain certification requirements for the Windows Store. These tests include:

- Examining the app manifest to verify that its contents are correct.
- Inspecting the resources defined in the app manifest to ensure that they are present and valid.
- Testing the app's resilience and stability.
- Determining how quickly the app starts and how fast it suspends.
- Inspecting the app to verify that it calls only APIs for Windows Store apps.

- Verifying that the app uses Windows security features.

You must run the Windows App Certification Kit on a release build of your app; otherwise, validation fails. For more info, see [How to: Set Debug and Release Configurations](#).

In addition, it's possible to validate your app whenever you build it. If you're running Team Foundation Build, you can modify settings on your build machine so that the Windows App Certification Kit runs automatically every time your app is built. For more info, see [Validating a package in automated builds](#).

For more info, see [Testing your app with the Windows App Certification Kit](#).

Creating a Windows Store certification checklist

You may choose to use the Windows Store as the primary method to make your app available. For info about how to prepare and submit your app, see [Selling apps](#).

As you plan your app, we recommend that you create a publishing-requirements checklist to use later when you test your app. This checklist can vary depending on how you've configured your business operations and the kind of app you're building. For more info and standard checklists, see [Publishing your app to the Store](#).

Before creating your app package for upload to the Windows Store, be sure to do the following:

- Review the app-submission checklist. This checklist indicates the information that you must provide when you upload your app. For more info, see [App submission checklist](#).
- Ensure that you have validated a release build of your app with the Windows App Certification Kit. For more info, see [Testing your app with the Windows App Certification Kit](#).
- Take some screen shots that show off the key features of your app.
- Have other developers test your app. For more info, see [Sharing an app package locally](#).

In addition, if your app collects personal data or uses software that is provided by others, you must also include a privacy statement or additional license terms.

Deploying and managing Windows Store apps

While you can use the Windows Store to market and distribute apps, business apps will often be distributed directly to the end-user by the IT organization within a company. The process of installing apps on Windows 8 devices without going through the Windows Store is called *side-loading*. For info about some best practices to help ensure that users have a good experience installing and running side-loaded apps for the first time, see [Deployment](#).

IT managers have several options for managing side-loaded apps and apps distributed from the Windows Store. For more info see [Management of Windows Store apps](#).

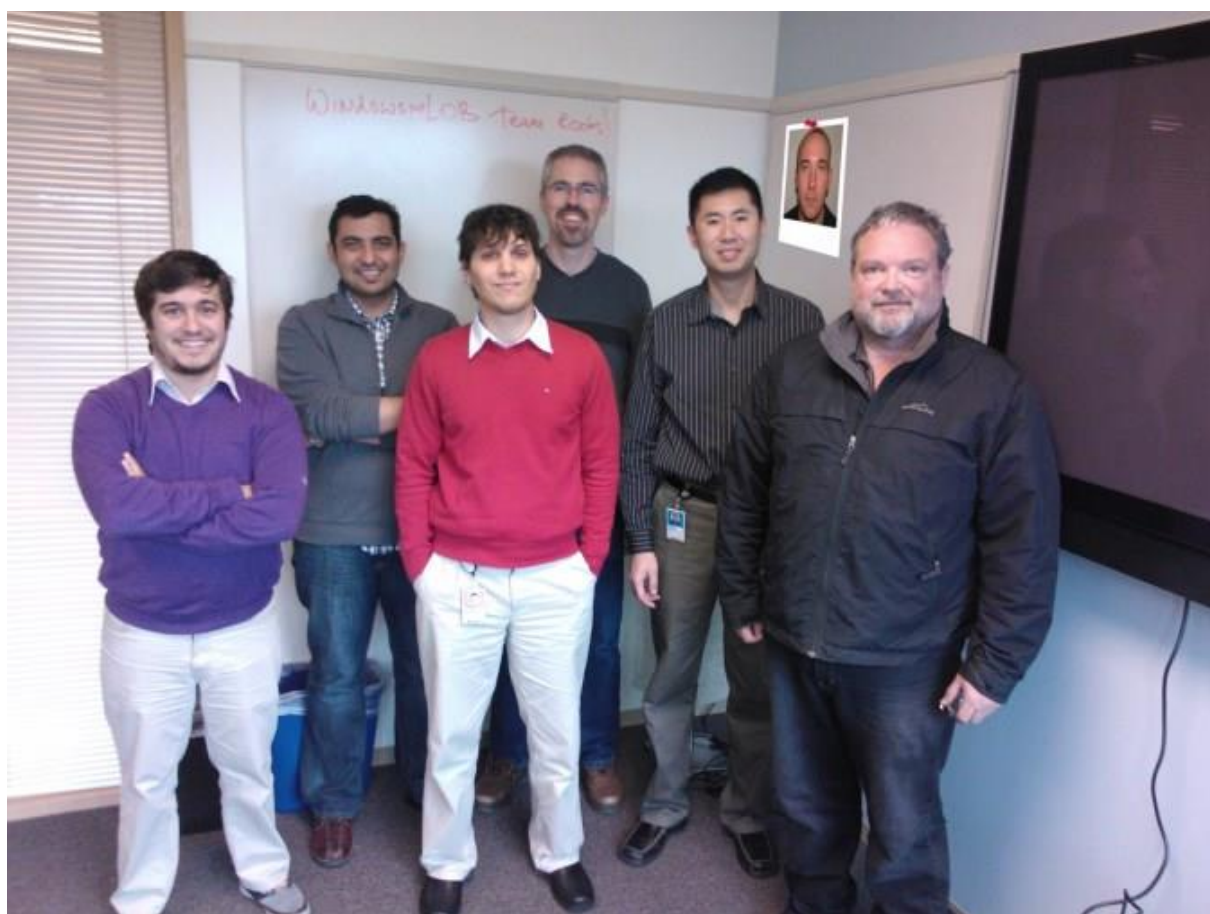
Meet the AdventureWorks Shopper team (Windows Store business apps using C#, XAML, and Prism)

This guide provides guidance to developers who want to create a Windows Store business app using C#, Extensible Application Markup Language (XAML), the Windows Runtime, and modern development practices. The guide comes with source code and documentation.

The goal of patterns & practices is to enhance developer success through guidance on designing and implementing software solutions. We develop content, reference implementations, samples, and frameworks that explain how to build scalable, secure, robust, maintainable software solutions. We work with community and industry experts on every project to ensure that some of the best minds in the industry have contributed to and reviewed the guidance as it develops. Visit the [patterns & practices Developer Center](#) to learn more about patterns & practices and what we do.

Meet the team

This guide was produced by:



- **Program Management:** Blaine Wastell
- **Development:** Francis Cheung, Brian Noyes (Solliance), Diego Poza (Southworks SRL), Mariano Vazquez (Southworks SRL)
- **Written guidance:** David Britch (Content Master Ltd)
- **Test:** Colin Campbell (Modeled Computation LLC), Carlos Farre, Mitesh Neema (Infosys Ltd), Hardik Patel (Infosys Ltd), Rohit Sharma, Veerapat Sriarunrungrueang (Adecco)
- **Graphic design:** Chris Burns (Linda Werner & Associates Inc.)
- **Bicycle Photography:** [Lincoln Potter](#) (Samaya LLC) and Mike Rabas ([Woodinville Bicycle](#))
- **Editorial support:** RoAnn Corbisier

We want to thank the customers, partners, and community members who have patiently reviewed our early content and drafts. We especially want to recognize Damir Arh, Christopher Bennage, Iñigo Bosque (Independent Consultant), Alon Fliess (Chief Architect, CodeValue), Ariel Ben Horesh (CodeValue), Ohad Israeli (Director of business development, NServiceBus), Brian Lagunas (Infragistics), Thomas Lebrun, Jeremy Likness (Principal Consultant, Wintellect), Chan Ming Man (Section Manager, AMD), Paulo Morgado, Oleg Nesterov (Senior Developer, Sberbank CIB), Jason De Oliveira (CTO at Cellenza, MVP C#), Caio Proiete (Senior Trainer, CICLO.pt), Jenner Maciejewsky Rocha (Consultor e Desenvolvedor, MVP Visual Basic), Mitchel Sellers (CEO/Director of Development, IowaComputerGurus Inc.), Tomer Shamam (Software Architect, CodeValue), Bruno Sonnino (Revolution Software), Perez Jones Tsisah (Freelance Software Developer), Daniel Vaughan, and Davide Zordan (Microsoft MVP) for their technical insights and support throughout this project.

We hope that you enjoy working with [Prism for the Windows Runtime](#), the AdventureWorks Shopper reference implementation source files, and this guide as much as we enjoyed creating it. Happy coding!

Quickstarts for AdventureWorks Shopper (Windows Store business apps using C#, XAML, and Prism)

This guidance includes a number of Quickstarts that illustrate specific concepts. These Quickstarts use [Prism for the Windows Runtime](#).

Quickstarts are small, focused apps that illustrate specific concepts. The following Quickstarts are included in this guidance:

- [Validation Quickstart for Windows Store apps using the MVVM pattern](#)
- [Event aggregation Quickstart for Windows Store apps](#)
- [Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime](#)

Validation Quickstart for Windows Store apps using the MVVM pattern

Summary

- Specify validation rules for model properties by adding data annotation attributes to the properties.
- Call the **ValidatableBindableBase.ValidateProperties** method to validate all the properties in a model object that possesses an attribute that derives from the [ValidationAttribute](#) attribute.
- Implement the **ValidatableBindableBase.ErrorsChanged** event in your view model class, in order to be notified when the validation errors change.

This Quickstart demonstrates how to validate user input for correctness in a Windows Store app by using [Prism for the Windows Runtime](#). The Quickstart uses the Model-View-ViewModel (MVVM) pattern, and demonstrates how to synchronously validate data using data annotations, and how to highlight validation errors on the UI by using an attached behavior.

You will learn

- How to synchronously validate data stored in a bound model object.
- How to specify validation rules for model properties by using data annotations.
- How to trigger validation through [PropertyChanged](#) events.
- How to highlight validation errors on the UI with an attached behavior.

Applies to

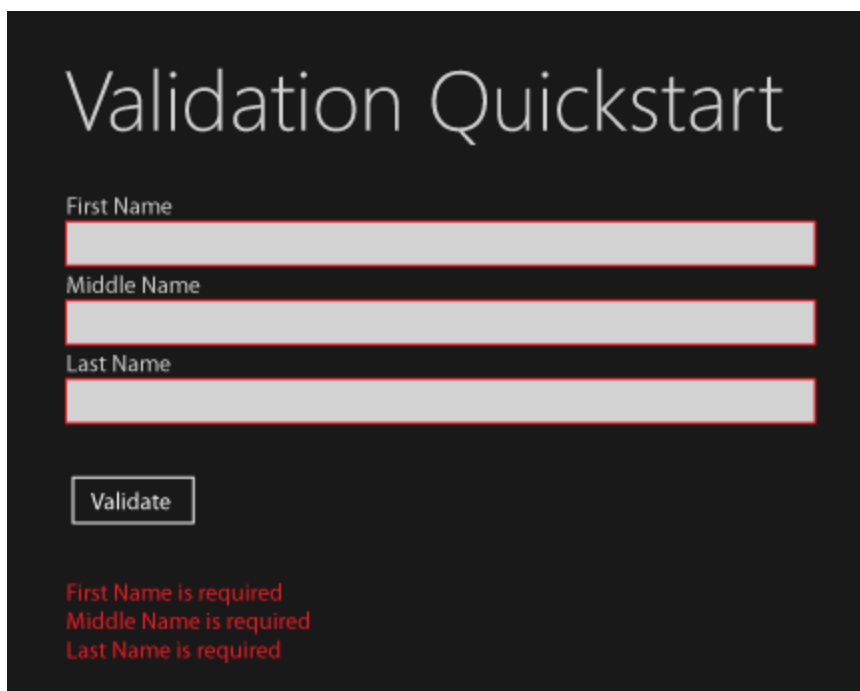
- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Building and running the Quickstart

Build the Quickstart as you would a standard project:

1. On the Microsoft Visual Studio menu bar, choose **Build > Build Solution**.
2. After you build the project, you must deploy it. On the menu bar, choose **Build > Deploy Solution**. Visual Studio also deploys the project when you run the app from the debugger.
3. After you deploy the project, pick the Quickstart tile to run the app. Alternatively, from Visual Studio, on the menu bar, choose **Debug > Start Debugging**.

When the app runs you will see a page similar to the one shown in the following diagram.



Validation Quickstart

First Name

Middle Name

Last Name

Validate

First Name is required
Middle Name is required
Last Name is required

This Quickstart performs synchronous validation of data stored in a model object. The page contains three text boxes that enable you to enter your name. When you enter data into a text box and the text box loses focus, the entered data is validated. In addition, when you select the **Submit** button, the content of each text box is validated. To pass validation each text box must contain data consisting of letters, spaces, and hyphens. If a validation error occurs, the text box containing the invalid data is highlighted with a red border and the validation error details are displayed in red text below the **Submit** button.

For more info about validation, see [Validating user input](#).

Solution structure

The ValidationQuickstart Visual Studio solution contains two projects: ValidationQuickstart, and [Microsoft.Practices.Prism.StoreApps](#). The ValidationQuickstart project uses Visual Studio solution folders to organize the source code into these logical categories:

- The **Assets** folder contains the splash screen and logo images.
- The **Behaviors** folder contains the attached behavior that is used to highlight controls that have validation errors.
- The **Common** folder contains the style resource dictionaries used in the app.
- The **Models** folder contains the model class used in the app, and a helper class that returns strings from the app's resource file.
- The **Strings** folder contains resource strings for the en-US locale.
- The **ViewModels** folder contains the view model class that is exposed to the view.
- The **Views** folder contains the view that makes up the UI for the app's page.

The [Microsoft.Practices.Prism.StoreApps](#) library contains reusable classes used by this Quickstart. For more info about this library, see [Prism for the Windows Runtime reference](#). With little or no modification, you can reuse many of the classes from this Quickstart in another app. You can also adapt the organization and ideas that this Quickstart provides.

Note This Quickstart does not include any suspend and resume functionality. For a validation implementation that includes suspend and resume functionality see [Validating user input](#).

Key classes in the Quickstart

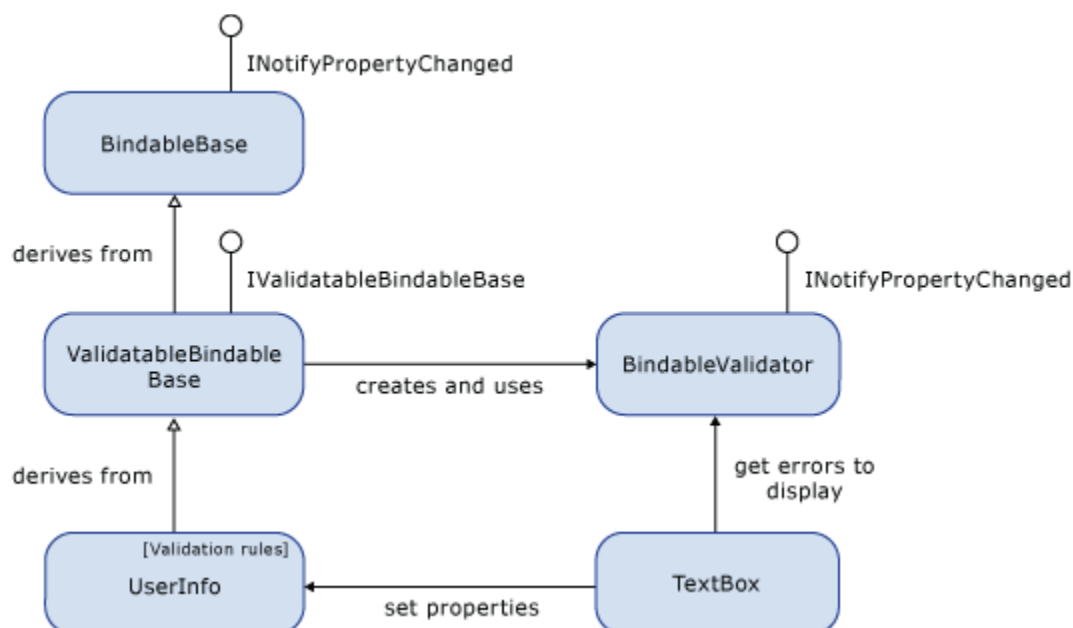
There are several classes involved in validation. The text boxes in the **UserInfoView** page bind to properties of a **UserInfo** model object.

The **UserInfo** class derives from the **ValidatableBindableBase** class that is provided by the [Microsoft.Practices.Prism.StoreApps](#) library. The base class contains an instance of the **BindableValidator** class, and uses it to invoke validation whenever a bound property changes, or when the user selects the **Validate** button.

The **BindableValidator** instance acts as the data source for validation error messages that are shown in the user interface. It is the type of the **ValidatableBindableBase** class's **Errors** property.

To perform the validation, the **BindableValidator** class retrieves validation rules that are encoded as custom attributes of the **UserInfo** object. It raises [PropertyChanged](#) and **ErrorsChanged** events when validation state changes.

The following diagram shows a conceptual view of the key classes involved in performing validation in this Quickstart.



Specifying validation rules

Validation rules for data are specified in the **UserInfo** model class. To participate in validation the **UserInfo** class must derive from the **ValidatableBindableBase** class.

The text boxes on the **UserInfoView** page use compound binding path expressions such as "{Binding UserInfo.FirstName, Mode=TwoWay}". This expression associates the text box's contents with the **FirstName** property of the object that is returned by the **UserInfo** property of the page's data context. This page's data context is a **UserInfoViewModel** object.

The **UserInfo** class contains properties for storing the first, middle, and last names. Validation rules for the value of each property are specified by adding attributes to each property that derive from the [ValidationAttribute](#) attribute. The following code example shows the **FirstName** property from the **UserInfo** class.

C#: ValidationQuickstart\Model\UserInfo.cs

```
private const string RegexPattern = @"^A\p{L}+([\p{Zs}\-][\p{L}]+)*\z";

[Required(ErrorMessageResourceType = typeof(ErrorMessagesHelper),
    ErrorMessageResourceName = "FirstNameRequired")]
[RegularExpression(RegexPattern, ErrorMessageResourceType =
    typeof(ErrorMessagesHelper), ErrorMessageResourceName = "FirstNameRegex")]
public string FirstName
{
    get { return _firstName; }
    set { SetProperty(ref _firstName, value); }
}
```

The [Required](#) attribute of the **FirstName** property specifies that a validation failure occurs if the field is null, contains an empty string, or contains only white-space characters. The [RegularExpression](#) attribute specifies that when the **FirstName** property is validated it must match the specified regular expression.

The static **ErrorMessagesHelper** class is used to retrieve validation error messages from the resource dictionary for the locale, and is used by the [Required](#) and [RegularExpression](#) validation attributes. For example, the **Required** attribute on the **FirstName** property specifies that if the property doesn't contain a value, the validation error message will be the resource string returned by the **FirstNameRequired** property of the **ErrorMessagesHelper** class. In addition, the **RegularExpression** attribute on the **FirstName** property specifies that if the data in the property contains characters other than letters, spaces, and hyphens, the validation error message will be the resource string returned by the **FirstNameRegex** property of the **ErrorMessagesHelper** class.

Note Using resource strings supports localization. However, this Quickstart only provides strings for the en-US locale.

Similarly, [Required](#) and [RegularExpression](#) attributes are specified on the **MiddleName** and **LastName** properties in the **UserInfo** class.

Triggering validation explicitly

Validation can be triggered manually when the user selects the **Validate** button. This calls the **ValidatableBindableBase.ValidateProperties** method, which in turn calls the **BindableValidator.ValidateProperties** method.

C#: Microsoft.Practices.Prism.StoreApps\BindableValidator.cs

```
public bool ValidateProperties()
{
    var propertiesWithChangedErrors = new List<string>();

    // Get all the properties decorated with the ValidationAttribute attribute.
    var propertiesToValidate = _entityToValidate.GetType().GetRuntimeProperties()
        .Where(c => c.GetCustomAttributes(typeof(ValidationAttribute)).Any());

    foreach (PropertyInfo propertyInfo in propertiesToValidate)
    {
        var propertyErrors = new List<string>();
        TryValidateProperty(propertyInfo, propertyErrors);

        // If the errors have changed, save the property name to notify the update
        // at the end of this method.
        bool errorsChanged = SetPropertyErrors(propertyInfo.Name, propertyErrors);
        if (errorsChanged &&
            !propertiesWithChangedErrors.Contains(propertyInfo.Name))
        {
            propertiesWithChangedErrors.Add(propertyInfo.Name);
        }
    }

    // Notify each property whose set of errors has changed since the last
    // validation.
    foreach (string propertyName in propertiesWithChangedErrors)
    {
        OnErrorsChanged(propertyName);
        OnPropertyChanged(string.Format(CultureInfo.CurrentCulture,
            "Item[{0}]", propertyName));
    }

    return _errors.Values.Count == 0;
}
```

This method retrieves all properties that have attributes that derive from the [ValidationAttribute](#) attribute, and attempts to validate them by calling the **TryValidateProperty** method for each property. If new validation errors occur the **ErrorsChanged** and [PropertyChanged](#) events are raised for each property than contains a new error.

The **TryValidateProperty** method uses the [Validator](#) class to apply the validation rules. This is shown in the following code example.

C#: Microsoft.Practices.Prism.StoreApps\BindableValidator.cs

```
private bool TryValidateProperty(PropertyInfo propertyInfo,
    List<string> propertyErrors)
{
    var results = new List<ValidationResult>();
    var context = new ValidationContext(_entityToValidate)
        { MemberName = propertyInfo.Name };
    var propertyValue = propertyInfo.GetValue(_entityToValidate);

    // Validate the property
    bool isValid = Validator.TryValidateProperty(propertyValue, context, results);

    if (results.Any())
    {
        propertyErrors.AddRange(results.Select(c => c.ErrorMessage));
    }

    return isValid;
}
```

Triggering validation implicitly on property change

Validation is automatically triggered whenever a bound property's value changes. When a two way binding in the **UserInfoView** class sets a bound property in the **UserInfo** class, the **SetProperty** method is called. This method, provided by the **BindableBase** class, sets the property value and raises the [PropertyChanged](#) event. However, the **SetProperty** method is also overridden by the **ValidatableBindableBase** class. The **ValidatableBindableBase.SetProperty** method calls the **BindableBase.SetProperty** method, and then provided that the property value has changed, calls the **ValidateProperty** method of the **BindableValidator** class instance.

The **ValidateProperty** method validates the property whose name is passed to the method by calling the **TryValidateProperty** method shown above. If a new validation error occurs the **ErrorsChanged** and **PropertyChanged** events are raised for the property.

Highlighting validation errors

Each text box on the UI uses the **HighlightOnErrors** attached behavior to highlight validation errors. The following code example shows how this behavior is attached to a text box.

XAML: ValidationQuickstart\Views\UserInfoView.xaml

```
<TextBox x:Name="FirstNameValue"
    Grid.Row="2"
    Text="{Binding UserInfo.FirstName, Mode=TwoWay}"
    behaviors:HighlightOnErrors.PropertyErrors=
        "{Binding UserInfo.Errors[FirstName]}" />
```

The attached behavior gets and sets the **PropertyErrors** dependency property. The following code example shows the **PropertyErrors** dependency property defined in the **HighlightOnErrors** class.

C#: ValidationQuickstart\Behaviors\HighlightOnErrors.cs

```
public static DependencyProperty PropertyErrorsProperty =
    DependencyProperty.RegisterAttached("PropertyErrors",
        typeof(ReadOnlyCollection<string>), typeof(HighlightOnErrors),
        new PropertyMetadata(BindableValidator.EmptyErrorsCollection,
            OnPropertyErrorsChanged));
```

The **PropertyErrors** dependency property is registered as a [ReadOnlyCollection](#) of strings, by the [RegisterAttached](#) method. The dependency property also has property metadata assigned to it. This metadata specifies a default value that the property system assigns to all cases of the property, and a static method that is automatically invoked by the property system whenever a new property value is detected. Therefore, when the value of the **PropertyErrors** dependency property changes, the **OnPropertyErrorsChanged** method is invoked. The following code example shows the **OnPropertyErrorsChanged** method.

C#: ValidationQuickstart\Behaviors\HighlightOnErrors.cs

```
private static void OnPropertyErrorsChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs args)
{
    if (args == null || args.NewValue == null)
    {
        return;
    }

    TextBox textBox = (TextBox)d;
    var propertyErrors = (ReadOnlyCollection<string>)args.NewValue;

    Style textBoxStyle = (propertyErrors.Count() > 0) ? (Style)Application.Current
        .Resources["HighlightTextStyle"] : null;

    textBox.Style = textBoxStyle;
}
```

The **OnPropertyErrorsChanged** method gets the instance of the [TextBox](#) that the **PropertyErrors** dependency property is attached to, and gets any validation errors for the **TextBox**. Then, if validation errors are present the **HighlightTextStyle** is applied to the **TextBox**, so that it is highlighted with a red [BorderBrush](#).

The UI also displays validation error messages below the **Submit** button in an [ItemsControl](#). This **ItemsControl** binds to the **AllErrors** property of the **UserInfoViewModel** class. The **UserInfoViewModel** constructor subscribes to the **ErrorsChanged** event of the **UserInfo** class, which is provided by the **ValidatableBindableBase** class. When this event is raised, the **OnErrorsChanged** handler updates the **AllErrors** property with the list of validation error strings from the dictionary

returned by the call to the **GetAllErrors** method on the **UserInfo** instance, as shown in the following code example.

C#: ValidationQuickstart\ViewModels\UserInfoViewModel.cs

```
private void OnErrorsChanged(object sender, DataErrorsChangedEventArgs e)
{
    AllErrors = new ReadOnlyCollection<string>(_userInfo.GetAllErrors()
        .Values.SelectMany(c => c).ToList());
}
```

Event aggregation Quickstart for Windows Store apps

Summary

- Define a pub/sub event by creating an empty class that derives from the **PubSubEvent<TPayload>** class.
- Notify subscribers by retrieving the pub/sub event from the event aggregator and calling its **Publish** method.
- Register to receive notifications by using one of the **Subscribe** method overloads available in the **PubSubEvent<TPayload>** class.

This Quickstart demonstrates event aggregation using [Prism for the Windows Runtime](#). Event aggregation allows communication between loosely coupled components in an app, removing the need for components to have a reference to each other.

You will learn

- How to define a pub/sub event.
- How to notify subscribers by retrieving a pub/sub event from the event aggregator.
- How to register to receive notifications for a pub/sub event.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

The Quickstart contains a publisher and several subscribers that communicate using an instance of the [Microsoft.Practices.Prism.PubSubEvents](#) library's **PubSubEvent<TPayload>** class. This instance is managed by an **EventAggregator** object.

In this Quickstart, the lifetimes of publishers and subscribers are independent because the objects are not connected by object references. There are also no type dependencies between publishers and subscribers—publisher and subscriber classes can be packaged in unrelated assemblies. Nonetheless, when the publisher invokes the **PubSubEvent<TPayload>** class's **Publish** method, the system will run all actions that have been registered by the **PubSubEvent<TPayload>** class's **Subscribe** method. Subscribers can control how the actions run. The Quickstart shows the following options:

- The action is invoked synchronously in the same thread as the **Publish** thread.
- The action is scheduled to run in the background on a thread-pool thread.
- The action is dispatched to the app's UI thread.

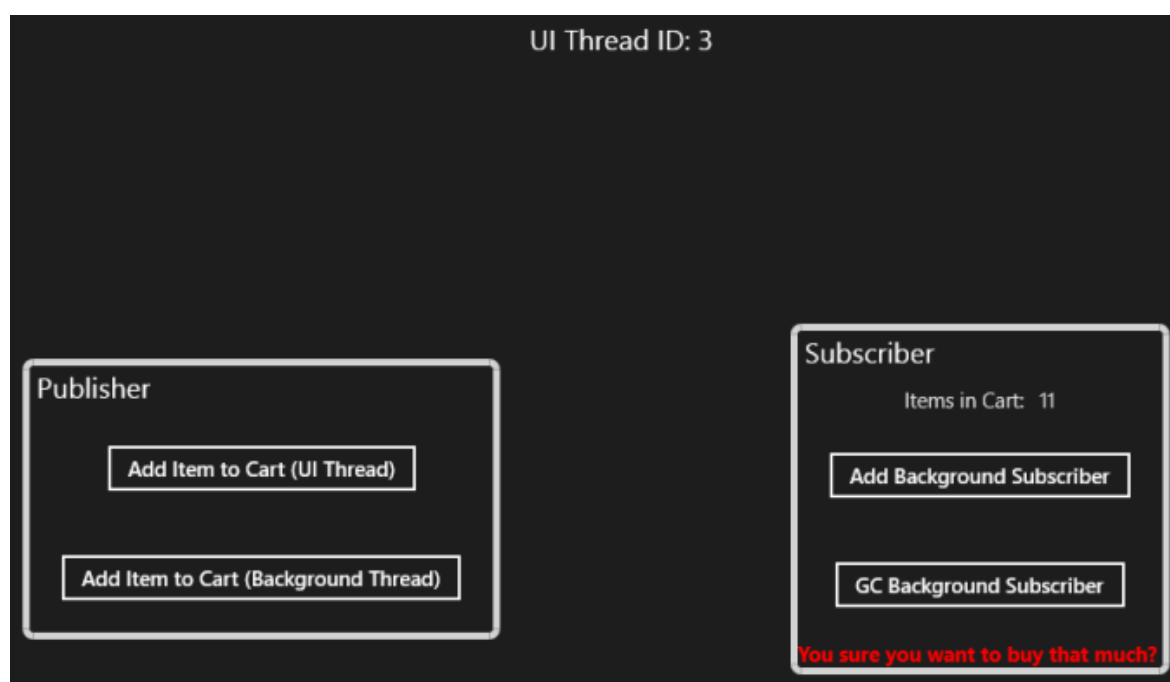
Subscriptions in this Quickstart use weak references. Registering a subscription action does not add a reference to the subscriber.

Building and running the Quickstart

Build the Quickstart as you would a standard project:

1. On the Microsoft Visual Studio menu bar, choose **Build > Build Solution**.
2. After you build the project, you must deploy it. On the menu bar, choose **Build > Deploy Solution**. Visual Studio also deploys the project when you run the app from the debugger.
3. After you deploy the project, pick the Quickstart tile to run the app. Alternatively, from Visual Studio, on the menu bar, choose **Debug > Start Debugging**.

When the app runs you will see a page similar to the one shown in the following diagram.



Panels represent the **PublisherViewModel** and **SubscriberViewModel** classes. In the left panel are two buttons that allow you to add items to a shopping cart, from the UI thread and from a background thread. Selecting either button causes the **PublisherViewModel** class to add an item to the shopping cart and invoke the **Publish** method of the **ShoppingCartChangedEvent** class that derives from the **PubSubEvent<TPayload>** class. The **SubscriberViewModel** class has two subscriptions to this event, in order to update the count of the number of items in the shopping cart, and to display a warning message once there are more than 10 items in the shopping cart.

On the right of the page there's a button for adding a background subscriber to the **ShoppingCartChangedEvent**. If this button is selected, a message dialog is shown from the background subscriber whenever the **ShoppingCartChangedEvent** is published. There's also a button that forces the background subscriber to be garbage collected. No special cleaned is required —the background subscriber did not need to call the **ShoppingCartChangedEvent** class's **Unsubscribe** method.

For more info about event aggregation, see [Communicating between loosely coupled components](#).

Solution structure

The EventAggregatorQuickstart Visual Studio solution contains three projects: EventAggregatorQuickstart, [Microsoft.Practices.Prism.PubSubEvents](#), and [Microsoft.Practices.Prism.StoreApps](#). The EventAggregatorQuickstart project uses Visual Studio solution folders to organize the source code into these logical categories:

- The **Assets** folder contains the splash screen and logo images.
- The **Common** folder contains the styles resource dictionary used in the app.
- The **Events** folder contains the **ShoppingCartChangedEvent** class.
- The **Models** folder contains the two model classes used in the app.
- The **ViewModels** folder contains the view model classes that are exposed to the views.
- The **Views** folder contains the views that make up the UI for the app's page.

The [Microsoft.Practices.Prism.StoreApps](#) library contains reusable classes used by this Quickstart. The [Microsoft.Practices.Prism.PubSubEvents](#) project is a Portable Class Library (PCL) that implements event aggregation. For more info about portable class libraries, see [Cross-Platform Development with the .NET Framework](#). This project has no dependencies on any other projects, and can be added to your own Visual Studio solution without the Microsoft.Practices.Prism.StoreApps library. For more info about these libraries, see [Prism for the Windows Runtime reference](#). With little or no modification, you can reuse many of the classes from this Quickstart in another app. You can also adapt the organization and ideas that this Quickstart provides.

Key classes in the Quickstart

The **EventAggregator** class is responsible for locating or building events and for managing the collection of events in the system. In this Quickstart, an instance of the **EventAggregator** class is created in the **OnLaunched** method in the **App** class. The **EventAggregator** instance must be created on the UI thread in order for UI thread dispatching to work. This instance is then passed into the view model classes through constructor injection. This is shown in the following code examples.

C#: EventAggregatorQuickstart\Bootstrapper.cs

```
public void Bootstrap(INavigationService navService)
{
    // Create the singleton EventAggregator so it can be dependency injected down
    // to the view models who need it
    _eventAggregator = new EventAggregator();
    ViewModelLocator.Register(typeof(MainPage).ToString(),
        () => new MainPageViewModel(_eventAggregator));
}
```

The app has a singleton instance of the **EventAggregator** class that is created on the UI thread.

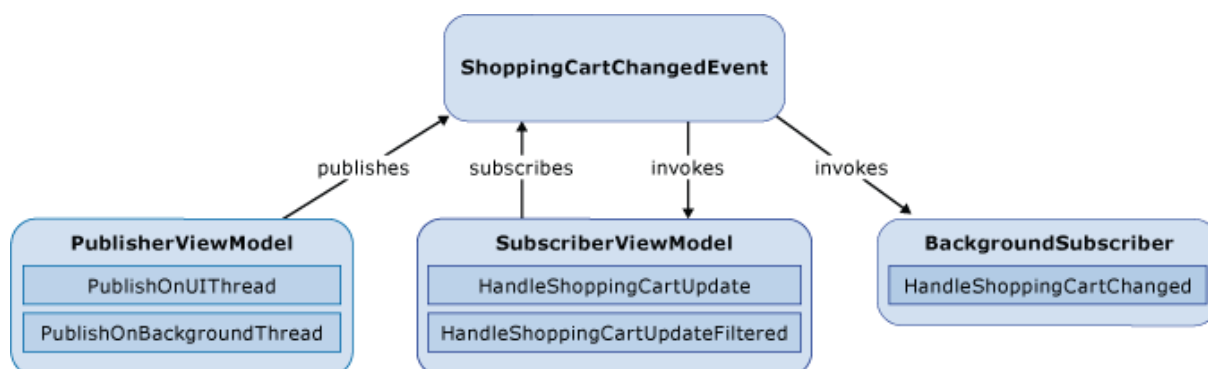
C#: EventAggregatorQuickstart\ViewModels\MainPageViewModel.cs

```
public MainPageViewModel(IEventAggregator eventAggregator)
{
    // Pass the injected event aggregator singleton down to children since there
    // is no container to do the dependency injection
    SubscriberViewModel = new SubscriberViewModel(eventAggregator);
    PublisherViewModel = new PublisherViewModel(eventAggregator);
}
```

View models, such as the **MainPageViewModel**, take the event aggregator object as a constructor parameter and pass this object to any of their child objects that need to use event aggregation. In the code example, the **MainPageViewModel** passes the event aggregator to the **SubscriberViewModel** and **PublisherViewModel** instances that it contains.

The **PubSubEvent<TPayload>** class connects event publishers and subscribers, and is the base class for an app's specific events. **TPayload** is the type of the event's payload, and is the argument that will be passed to subscribers when an event is published. Compile-time checking helps publishers and subscribers provide successful event connection.

The following diagram shows a conceptual view of how event aggregation is used in this Quickstart.



Defining the ShoppingCartChangedEvent class

The **ShoppingCartChangedEvent** class's **Publish** method is invoked when the user adds an item to the shopping cart. This class, which derives from the **PubSubEvent<TPayload>** class, is used to communicate between the loosely coupled **PublisherViewModel** and **SubscriberViewModel** classes. The following code example shows how the **ShoppingCartChangedEvent** is defined, specifying **ShoppingCart** as the payload type.

C#: EventAggregatorQuickstart\Events\ShoppingCartChangedEvent.cs

```
public class ShoppingCartChangedEvent : PubSubEvent<ShoppingCart> { }
```

Notifying subscribers of the ShoppingCartChangedEvent

Users can add an item to the shopping cart from both the UI thread and from a background thread. When an item is added to the shopping cart the **PublisherViewModel** class calls the **ShoppingCartChangedEvent**'s **Publish** method in order to alert subscribers of the change to the shopping cart. The following code example shows how the subscribers are notified.

C#: EventAggregatorQuickstart\ViewModels\PublisherViewModel.cs

```
private void PublishOnUIThread()
{
    AddItemToCart();
    // Fire the event on the UI thread
    _eventAggregator.GetEvent<ShoppingCartChangedEvent>().Publish(_cart);
}

private void PublishOnBackgroundThread()
{
    AddItemToCart();
    Task.Factory.StartNew(() =>
    {
        // Fire the event on a background thread
        _eventAggregator.GetEvent<ShoppingCartChangedEvent>().Publish(_cart);
        Debug.WriteLine(String.Format("Publishing from thread: {0}",
            Environment.CurrentManagedThreadId));
    });
}

private void AddItemToCart()
{
    var item = new ShoppingCartItem("Widget", 19.99m);
    _cart.AddItem(item);
}
```

Publishing can occur from any thread. The **EventAggregator** and **PubSubEvent<TPayload>** classes are thread safe. The Quickstart shows this by notifying subscribers from both the UI thread and a background thread.

Note If you access objects from more than one thread you must ensure that you appropriately serialize reads and writes. For example, the **ShoppingCart** class in this Quickstart is a thread safe class.

The **PublishOnUIThread** and **PublishOnBackgroundThread** methods add an item to the shopping cart by creating and initializing an instance of the **ShoppingCartItem** class. Then, the **ShoppingCartChangedEvent** is retrieved from the **EventAggregator** class and the **Publish** method is invoked on it. This supplies the **ShoppingCart** instance as the **ShoppingCartChangedEvent** event's parameter. The **EventAggregator** class's **GetEvent** method constructs the event if it has not already been constructed.

Registering to receive notifications of the ShoppingCartChangedEvent

Subscribers can register actions with a **PubSubEvent<TPayload>** instance using one of its **Subscribe** method overloads. The **SubscriberViewModel** class subscribes to the **ShoppingCartChangedEvent** on the UI thread, regardless of which thread published the event. The subscriber indicates this during subscription by specifying a **ThreadOption.UIThread** value, as shown in the following code example.

C#: EventAggregatorQuickstart\ViewModels\SubscriberViewModel.cs

```
// Subscribe indicating this handler should always be called on the UI Thread
_eventAggregator.GetEvent<ShoppingCartChangedEvent>()
    .Subscribe(HandleShoppingCartUpdate, ThreadOption.UIThread);
// Subscribe indicating that this handler should always be called on UI thread,
// but only if more than 10 items in cart
_eventAggregator.GetEvent<ShoppingCartChangedEvent>()
    .Subscribe(HandleShoppingCartUpdateFiltered, ThreadOption.UIThread, false,
        IsCartCountPossiblyTooHigh);
```

Subscribers provide an action with a signature that matches the payload of the pub/sub event. For example, the **HandleShoppingCartUpdate** method takes a **ShoppingCart** parameter. The method updates the number of items that are in the shopping cart.

A second subscription is made to the **ShoppingCartChangedEvent** using a filter expression. The filter expression defines a condition that the payload must meet for before the action will be invoked. In this case, the condition is satisfied if there are more than 10 items in the shopping cart. The **HandleShoppingCartUpdateFiltered** method shows a warning message to the user, indicating that they have more than 10 items in their shopping cart.

Note For UI thread dispatching to work, the **EventAggregator** class must be created on the UI thread. This allows it to capture and store the [SynchronizationContext](#) that is used to dispatch to the UI thread for subscribers that use the **ThreadOption.UIThread** value. If you want to use dispatching on the UI thread, you must make sure that you instantiate the **EventAggregator** class in your app's UI thread.

The **PubSubEvent<TPayload>** class, by default, maintains a weak delegate reference to the subscriber's registered action and any filter. This means that the reference that the **PubSubEvent<TPayload>** class holds onto will not prevent garbage collection of the subscriber. Using a weak delegate reference relieves the subscriber from the need to unsubscribe from the event. The garbage collector will dispose the subscriber instance when there are no references to it.

Note Lambda expressions that capture the **this** reference cannot be used as weak references. You should use instance methods as the **Subscribe** method's **action** and **filter** parameters if you want to take advantage of the **PubSubEvent<TPayload>** class's weak reference feature.

When the **Add Background Subscriber** button is selected the **AddBackgroundSubscriber** method is invoked. This method creates a background subscriber and holds onto the reference to the

subscribing object in order to prevent it from being garbage collected. The method also subscribes using the **HandleShoppingCartChanged** method as the subscribed action. After the subscription is established, any call to the **ShoppingCartChangedEvent's Publish** method will synchronously invoke the **HandleShoppingCartChanged** method that displays a message dialog that informs the user that the shopping cart has been updated. The messages gives the numerical thread ID of the calling thread. You can use this to see that the expected thread was used for the action, depending on which button you used to add the shopping cart item.

C#: EventAggregatorQuickstart\ViewModels\SubscriberViewModel.cs

```
private void AddBackgroundSubscriber()
{
    if (_subscriber != null) return;

    // Create subscriber and hold on to it so it does not get garbage collected
    _subscriber = new BackgroundSubscriber(Window.Current.Dispatcher);
    // Subscribe with defaults, pointing to subscriber method that
    // pops a message box when the event fires
    _eventAggregator.GetEvent<ShoppingCartChangedEvent>()
        .Subscribe(_subscriber.HandleShoppingCartChanged);
}
```

When the **GC Background Subscriber** button is selected the **GCBackgroundSubscriber** method is invoked. This method releases the reference to the background subscriber and forces the garbage collector to run. This garbage collects the background subscriber. The registered action will then no longer be invoked by the **Publish** method.

C#: EventAggregatorQuickstart\ViewModels\SubscriberViewModel.cs

```
private void GCBackgroundSubscriber()
{
    // Release and GC, showing that we don't have to unsubscribe to keep the
    // subscriber from being garbage collected
    _subscriber = null;
    GC.Collect();
}
```

Bootstrapping an MVVM Windows Store app Quickstart using Prism for the Windows Runtime

Summary

- Bootstrap your Windows Store app by deriving your **App** class from the **MvvmAppBase** class, and provide app specific startup behavior in your **App** class to supplement the core startup behavior of the **MvvmAppBase** class.
- Use a dependency injection container to abstract dependencies between objects, and automatically generate dependent object instances.
- Limit view model instantiation to a single class by using a view model locator object.

This Quickstart demonstrates how to bootstrap a Windows Store app that uses the Model -View-ViewModel (MVVM) pattern. The Quickstart uses [Prism for the Windows Runtime](#), which provides MVVM support with lifecycle management and core services to a Windows Store app.

You will learn

- How to bootstrap a Windows Store app that uses the MVVM pattern.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

This Quickstart uses the [Unity](#) container for dependency resolution and construction during the bootstrapping process. However, you are not required to use Unity, or any other dependency injection container, when bootstrapping an MVVM Windows Store app. To understand how to perform bootstrapping without using a dependency injection container, see [Bootstrapping without a dependency injection container](#).

Building and running the Quickstart

Build the HelloWorldWithContainer Quickstart as you would a standard project:

1. On the Microsoft Visual Studio menu bar, choose **Build > Build Solution**.
2. After you build the project, you must deploy it. On the menu bar, choose **Build > Deploy Solution**. Visual Studio also deploys the project when you run the app from the debugger.
3. After you deploy the project, pick the Quickstart tile to run the app. Alternatively, from Visual Studio, on the menu bar, choose **Debug > Start Debugging**.

When the app runs you will see the page shown in the following diagram.



The page lists some of the architectural features of Prism, and has a [Button](#) that allows you to navigate to a second page. Selecting the **Navigate To User Input Page** button will take you to the second page of the app, as shown in the following diagram.



This page allows you to enter data into two [TextBox](#) controls. If you suspend the app on this page any data will be serialized to disk, and when the app resumes the data will be deserialized and displayed in the **TextBox** controls. This is accomplished by using the **RestorableState** attribute for the data retained in the view model, and the **SessionStateService** class for the data retained in the repository. For more info about the **SessionStateService** class and the **RestorableState** attribute see [Handling suspend, resume, and activation](#).

Solution structure

The HelloWorldWithContainer Visual Studio solution contains two projects: HelloWorldWithContainer, and [Microsoft.Practices.Prism.StoreApps](#). The HelloWorldWithContainer project uses Visual Studio solution folders to organize the source code into these logical categories:

- The **Assets** folder contains the splash screen and logo images.
- The **Common** folder contains the styles resource dictionary used in the app.

- The **Services** folder contains the **IDataRepository** interface and its implementing class.
- The **ViewModels** folder contains the view model classes that are exposed to the views.
- The **Views** folder contains the views that make up the UI for the app's page.

The [Microsoft.Practices.Prism.StoreApps](#) library contains reusable classes used by this Quickstart.

For more info about this library, see [Prism for the Windows Runtime reference](#). With little or no modification, you can reuse many of the classes from this Quickstart in another app. You can also adapt the organization and ideas that this Quickstart provides.

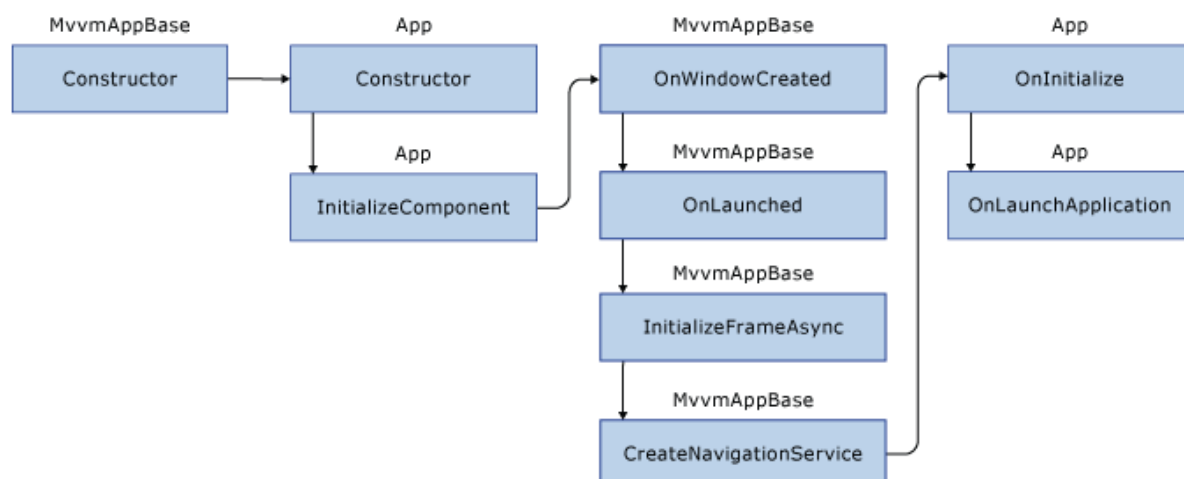
Key classes in the Quickstart

The **MvvmAppBase** class provides core startup behavior for an MVVM app, with its constructor being the entry point for the app. The **App** class adds app specific startup behavior to the app.

There are two view classes in the app, **MainPage** and **UserInputPage** that bind to the **MainPageViewModel** and **UserInputPageViewModel** classes respectively. Each view class derives from the **VisualStateAwarePage** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, that provides view management and navigation support. Each view model class derives from the **ViewModel** base class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, that provides support for navigation and suspend/resume functionality. A static **ViewModelLocator** object, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, is used to manage the instantiation of view models and their association to views. This approach has the advantage that the app has a single class that is responsible for the location and instantiation of view model classes. For more info about how the **ViewModelLocator** object manages the instantiation of view models and their association to views, see [Using the MVVM pattern](#).

Bootstrapping an MVVM app using the MvvmAppBase class

The **MvvmAppBase** class, provided by the [Microsoft.Practices.Prism.StoreApps](#) library, is responsible for providing core startup behavior for an MVVM app, and derives from the [Application](#) class. The **MvvmAppBase** class constructor is the entry point for the app. The following diagram shows a conceptual view of how app startup occurs.



The **MvvmAppBase** class helps bootstrap Windows Store apps with suspension, navigation, and other services.

In order to bootstrap an app using the **MvvmAppBase** class, the **App** class must derive from the **MvvmAppBase** class, as shown in the following code examples.

XAML: HelloWorldWithContainer\App.xaml

```
<Infrastructure:MvvmAppBase
    ...
    xmlns:Infrastructure="using:Microsoft.Practices.Prism.StoreApps">
    <Application.Resources>
        ...
    </Application.Resources>
</Infrastructure:MvvmAppBase>
```

C#: HelloWorldWithContainer\App.xaml.cs

```
sealed partial class App : MvvmAppBase
```

Adding app specific startup behavior to the App class

When deriving from the **MvvmAppBase** class, app specific startup behavior can be added to the **App** class. A required override in the **App** class is the **OnLaunchApplication** method from where you will typically perform your initial navigation to a launch page, or to the appropriate page based on a search, sharing, or secondary tile launch of the app. The following code example shows the **OnLaunchApplication** method in the **App** class.

C#: HelloWorldWithContainer\App.xaml.cs

```
public override OnLaunchApplication(LaunchActivatedEventArgs args)
{
    NavigationService.Navigate("Main", null);
}
```

This method navigates to the **MainPage** in the app, when the app launches. "Main" is specified as the logical name of the view that will be navigated to. The default convention specified in the **MvvmAppBase** class is to append "Page" to the name and look for that page in a .Views child namespace in the project. Alternatively, another convention can be specified by overriding the **GetPageType** method in the **MvvmAppBase** class.

The app uses the [Unity](#) dependency injection container to reduce the dependency coupling between objects by providing a facility to instantiate instances of classes and manage their lifetime based on the configuration of the container. An instance of the container is created as a singleton in the **App** class, as shown in the following code example.

C#: HelloWorldWithContainer\App.xaml.cs

```
IUnityContainer _container = new UnityContainer();
```

If you require app specific initialization behavior you should override the **OnInitialize** method in the **App** class. For instance, this method should be overridden if you need to initialize services, or set a default factory or default view model resolver for the **ViewModelLocator** object. The following code example shows the **OnInitialize** method.

C#: HelloWorldWithContainer\App.xaml.cs

```
protected override void OnInitialize(IActivatedEventArgs args)
{
    // Register MvvmAppBase services with the container so that view models can
    // take dependencies on them
    _container.RegisterInstance<ISessionStateService>(SessionStateService);
    _container.RegisterInstance<INavigationService>(NavigationService);
    // Register any app specific types with the container
    _container.RegisterType<IDataRepository>, DataRepository>();

    // Set a factory for the ViewModelLocator to use the container to construct
    // view models so their
    // dependencies get injected by the container
    ViewModelLocator.SetDefaultViewModelFactory((viewModelType)
        => _container.Resolve(viewModelType));
}
```

This method registers the **SessionStateService** and **NavigationService** instances from the **MvvmAppBase** class with the container as singletons, based on their respective interfaces, so that the view model classes can take dependencies on them. The **DataRepository** class is then registered with the container, based on its interface. The **DataRepository** class provides data for display on the **MainPage**, and methods for reading and writing data input from one of the [TextBox](#) controls on the **UserInputPage**. The **OnInitialize** method then sets the default view model factory for the **ViewModelLocator** object so that it uses the container to construct view model instances whose dependencies are injected by the container.

In this Quickstart the **ViewModelLocator** object uses a convention-based approach to locate and instantiate view models from views. This convention assumes that view models are in the same assembly as the view types, that view models are in a **.ViewModels** child namespace, that views are in a **.Views** child namespace, and that view model names correspond with view names and end with "ViewModel". The **ViewModelLocator** class has an attached property, **AutoWireViewModel**, that is used to manage the instantiation of view models and their association to views. In the view's XAML this attached property is set to **true** to indicate that the view model class should be automatically instantiated from the view class.

XAML: HelloWorldWithContainer\Views\MainPage.xaml

```
Infrastructure:ViewModelLocator.AutoWireViewModel="true"
```

The **AutoWireViewModel** property is a dependency property that is initialized to **false**, and when its value changes the **AutoWireViewModelChanged** event handler in the **ViewModelLocator** class is called to resolve the view model for the view. The following code example shows how this is achieved.

C#: Microsoft.Practices.Prism.StoreApps\ViewModelLocator.cs

```
private static void AutoWireViewModelChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs e)
{
    FrameworkElement view = d as FrameworkElement;
    if (view == null) return; // Incorrect hookup, do no harm

    // Try mappings first
    object viewModel = GetViewModelForView(view);
    // Fallback to convention based
    if (viewModel == null)
    {
        var viewModelType =
            defaultViewTypeToViewModelTypeResolver(view.GetType());
        if (viewModelType == null) return;

        // Really need Container or Factories here to deal with injecting
        // dependencies on construction
        viewModel = defaultViewModelFactory(viewModelType);
    }
    view.DataContext = viewModel;
}
```

The **AutoWireViewModelChanged** method first attempts to resolve the view model based on mappings that are not present in this Quickstart. If the view model cannot be resolved using this approach, for instance if the mapping wasn't registered, the method falls back to using the convention-based approach outlined earlier to resolve the correct view model type. The view model factory, set by the **OnInitialize** method in the **App** class, uses the dependency injection container to construct view model instances whose dependencies are injected by the container. When the view model instances are constructed, dependencies specified by the constructor parameters are resolved by the container and then passed into the view model. This is referred to as constructor injection. This approach removes the need for an object to locate its dependencies or manage their lifetimes, allows swapping of implemented dependencies without affecting the object, and facilitates testability by allowing dependencies to be mocked. Finally, the method sets the [DataContext](#) property of the view type to the registered view model instance.

Bootstrapping without a dependency injection container

You are not required to use [Unity](#), or any other dependency injection container, when bootstrapping Windows Store apps. The HelloWorld Quickstart demonstrates how to bootstrap a Windows Store app that uses the MVVM pattern by registering factory methods against view types, with a view model locator object.

As previously mentioned, if you require app specific initialization behavior you should override the **OnInitialize** method in the **App** class. For instance, this method should be overridden if you need to initialize services, or set a default factory or default view model resolver for the **ViewModelLocator** object. The following code example shows the **OnInitialize** method.

C#: HelloWorld\App.xaml.cs

```
protected override void OnInitialize(IActivatedEventArgs args)
{
    // New up the singleton data repository, and pass it the state service it
    // depends on from the base class
    _dataRepository = new DataRepository(SessionStateService);

    // Register factory methods for the ViewModelLocator for each view model that
    // takes dependencies so that you can pass in the dependent services from the
    // factory method here.
    ViewModelLocator.Register(typeof(MainPage).ToString(),
        () => new MainPageViewModel(_dataRepository, NavigationService));
    ViewModelLocator.Register(typeof(UserInputPage).ToString(),
        () => new UserInputPageViewModel(_dataRepository, NavigationService));
}
```

This method creates a singleton from the **DataRepository** class, passing in the **SessionStateService** from the **MvvmAppBase** class. The **DataRepository** class provides data for display on the **MainPage**, and methods for reading and writing data input from one of the [TextBox](#) controls on the **UserInputPage**. The **OnInitialize** method also registers a factory method for each view type with the static **ViewModelLocator** object. This ensures that the **ViewModelLocator** object instantiates the correct view model object for a view type, passing in dependent services to the view model constructor from the factory method.

Prism for the Windows Runtime reference (Windows Store business apps using C#, XAML, and Prism)

Summary

- Use the [Microsoft.Practices.Prism.StoreApps](#) library to add MVVM support with lifecycle management, and core services to your Windows Store app.
- Use the [Microsoft.Practices.Prism.PubSubEvents](#) library to communicate between loosely coupled components in your app.

Prism for the Windows Runtime provides two libraries that help developers create managed Windows Store apps. The libraries accelerate the development of Windows Store apps by using proven design patterns such as the Model-View-ViewModel (MVVM) pattern and the event aggregation pattern. The [Microsoft.Practices.Prism.StoreApps](#) library provides support for bootstrapping MVVM apps, state management, validation of user input, navigation, data binding, commands, Flyouts, settings, and search. The [Microsoft.Practices.Prism.PubSubEvents](#) library allows communication between loosely coupled components in an app, thus helping to reduce dependencies between assemblies in a Microsoft Visual Studio solution.

You will learn

- About the classes and interfaces contained in the [Microsoft.Practices.Prism.StoreApps](#) library.
- About the classes and interfaces contained in the [Microsoft.Practices.Prism.PubSubEvents](#) library.

Applies to

- Windows Runtime for Windows 8
- C#
- Extensible Application Markup Language (XAML)

Prism helps developers create managed Windows Store apps. It accelerates development by providing support for MVVM, loosely coupled communication, and the core services required in Windows Store apps. It is designed to help developers create apps that need to accomplish the following:

- Address the common Windows Store app development scenarios.
- Build apps composed of independent, yet cooperating, pieces.
- Separate the concerns of presentation, presentation logic, and model through support for Model-View-ViewModel (MVVM).
- Use an architectural infrastructure to produce a consistent and high quality app.

Both libraries in Prism ship as source, with the [Microsoft.Practices.Prism.PubSubEvents](#) library also shipping as a signed binary.

Microsoft.Practices.Prism.StoreApps library

The [Microsoft.Practices.Prism.StoreApps](#) library is a class library that provides MVVM support with lifecycle management, and core services to a Windows Store app.

The following table lists the classes contained in the [Microsoft.Practices.Prism.StoreApps](#) library:

Class	Description
AppManifestHelper	Loads the package manifest and allows you to retrieve the application id, and check if the app uses the Search contract. This class can be extended to retrieve other app manifest values that are not exposed by the API.
BindableBase	Implementation of the INotifyPropertyChanged interface, to simplify view model and model class property change notification.
BindableValidator	Validates entity property values against entity-defined validation rules and exposes, through an indexer, a collection of errors for properties that did not pass validation.
Constants	An internal class that contains constants used by the library.
DelegateCommand	An ICommand implementation whose delegates do not take any parameters for <code>Execute()</code> and <code>CanExecute()</code> .
DelegateCommand<T>	An ICommand implementation whose delegates can be attached for <code>Execute(T)</code> and <code>CanExecute(T)</code> .
DelegateCommandBase	The base ICommand implementation whose delegates can be attached for <code>Execute(Object)</code> and <code>CanExecute(Object)</code> .
FlyoutService	A service class that implements the <code>IFlyoutService</code> interface to display Flyouts that derive from the <code>FlyoutView</code> class.
FlyoutView	A base class for views that will be displayed as Flyouts.
FrameFacadeAdapter	A facade and adapter class that implements the <code>IFrameFacade</code> interface to abstract the Frame object.
FrameNavigationService	A service class that implements the <code>INavigationService</code> interface to navigate through the pages of an app.
MvvmAppBase	Helps to bootstrap Windows Store apps that use the MVVM pattern, with services provided by the Microsoft.Practices.Prism.StoreApps library.
MvvmNavigatedEventArgs	Provides data for navigation methods and event handlers that cannot cancel a navigation request.
ResourceLoaderAdapter	An adapter class that implements the <code>IResourceLoader</code> interface to adapt the ResourceLoader object.
RestorableStateAttribute	Defines an attribute that indicates that any marked property will save its state on suspension, provided that the marked property is in an instance of a class that derives from the <code>ViewModel</code> class.
SearchPaneService	A service class that implements the <code>ISearchPaneService</code> interface to abstract the SearchPane object.
SearchQueryArguments	Abstracts the SearchPaneQuerySubmittedEventArgs and the SearchActivatedEventArgs objects in order to provide one event handler that handles the search activation events for both when the app is running and when

	it is not.
SessionStateService	A service class that implements the <code>ISessionStateService</code> interface to capture global session state in order to simplify process lifetime management for an app.
SettingsCharmActionItem	Defines an item that is used to populate the Settings pane. Each item has an associated Action that will be executed when the item is selected in the Settings pane.
StandardFlyoutSize	A static class that defines the widths of narrow and wide Flyouts.
ValidatableBindableBase	Implements the <code>IValidatableBindableBase</code> interface to validate model property values against their validation rules and return any validation errors.
ViewModel	The base view model class that implements the <code>INavigationAware</code> interface to provide navigation support and state management to derived view model classes.
ViewModelLocator	Locates the view model class for views that have the <code>AutoWireViewModel</code> attached property set to true.
VisualStateAwarePage	The base view class for pages that need to be aware of layout changes and update their visual state accordingly.

The following table lists the interfaces contained in the [Microsoft.Practices.Prism.StoreApps](#) library:

Interface	Description
ICredentialStore	Defines an interface for the <code>RoamingCredentialStore</code> class that abstracts the PasswordVault object for managing user credentials.
IFlyoutService	Defines an interface that can be used to implement a service for displaying Flyouts.
IFlyoutViewModel	Defines an interface that should be implemented by Flyout view model classes to provide actions for opening and closing a Flyout, and navigation away from the Flyout.
IFrameFacade	Defines an interface for the <code>FrameFacadeAdapter</code> class that abstracts the Frame object for use by apps that derive from the <code>MwmAppBase</code> class.
INavigationAware	Defines an interface that allows an implementing class to participate in a navigation operation.
INavigationService	Defines an interface that allows an implementing class to create a navigation service.
IResourceLoader	Defines an interface for the <code>ResourceLoaderAdapter</code> class that abstracts the ResourceLoader object for use by apps that derive from the <code>MwmAppBase</code> class.
ISearchPaneService	Defines an interface for the <code>SearchPaneService</code> class that abstracts the SearchPane object for use by apps that derive from the <code>MwmAppBase</code> class.
ISessionStateService	Defines an interface that allows an implementing class to capture global session state.
IValidatableBindableBase	Defines an interface that allows an implementing class to add validation support to model classes that contain validation rules.

For info about how this library was used in the AdventureWorks Shopper reference implementation, see [Using the Model-View-ViewModel \(MVVM\) pattern](#), [Creating and navigating between pages](#),

[Validating user input](#), [Managing application data](#), [Handling suspend, resume, and activation](#), [Communicating between loosely coupled components](#), and [Implementing search](#).

Microsoft.Practices.Prism.PubSubEvents library

The [Microsoft.Practices.Prism.PubSubEvents](#) library is a Portable Class Library (PCL) that contains classes that implement event aggregation. You can use this library for communicating between loosely coupled components in your own app. The library has no dependencies on other libraries, and can be added to your Visual Studio solution without the [Microsoft.Practices.Prism.StoreApps](#) library. The PCL targets:

- Microsoft .NET for Windows Store apps
- .NET Framework 4 and higher
- Microsoft Silverlight 4 and higher
- Windows Phone 7 and higher
- Xbox 360

For more info about portable class libraries, see [Cross-Platform Development with the .NET Framework](#)

The following table lists the classes contained in the [Microsoft.Practices.Prism.PubSubEvents](#) library:

Class	Description
BackgroundEventSubscription<TPayload>	Extends EventSubscription<TPayload> to invoke the Action delegate in a background thread.
BackgroundEventSubscription<TPayload>	Extends EventSubscription<TPayload> to invoke the Action delegate in a background thread.
DelegateReference	Represents a reference to a Delegate that may contain a WeakReference to the target. This class is used internally.
DispatcherEventSubscription<TPayload>	Extends EventSubscription<TPayload> to invoke the Action delegate in a specific Dispatcher .
EventAggregator	Implements IEventAggregator.
EventBase	Defines a base class to publish and subscribe to events.
EventSubscription<TPayload>	Provides a way to retrieve a Delegate to execute an action depending on the value of a second filter predicate that returns true if the action should execute.
PubSubEvent<TPayload>	Defines a class that manages publication and subscription to events.
SubscriptionToken	Subscription token returned from EventBase on subscribe.

The following table lists the interfaces contained in the [Microsoft.Practices.Prism.PubSubEvents](#) library:

Interface	Description
IDelegateReference	Represents a reference to a Delegate .
IEventAggregator	Defines an interface to get instances of an event type.
IEventSubscription	Defines a contract for an event subscription to be used by EventBase.

The following table lists the enumerations contained in the [Microsoft.Practices.Prism.PubSubEvents](#) library:

Enumeration	Description
ThreadOption	Specifies on which thread a PubSubEvent<TPayload> subscriber will be called.

For info about publishing and subscribing to events, see [Communicating between loosely coupled components](#) and [Event aggregation Quickstart](#).