Windows 8

User experience guidelines

Windows 8

TABLE OF CONTENTS

INTRODUCTION

Microsoft design principles

Microsoft design has a set of five guiding principles to help you make the best choices when designing your app. These principles are the foundation for building great Windows Store apps. Consider these principles as you plan your app, and always ensure that your design and development choices live up to these principles.

Show pride in craftsmanship

Devote time and energy to small things that your users see often. Engineer the experience to be complete and polished at every stage.

- Sweat the details.
- Make using apps safe and reliable.
- Use balance, symmetry, and hierarchy to foster trust and a sense of integrity.
- Align your app layout to the grid, the Windows 8 UI layout for apps.
- Make your app accessible to the widest possible audience, including people who have impairments or disabilities.

Be fast and fluid

Let people interact directly with content, and respond to actions quickly with matching energy. Bring life to the experience by creating a sense of continuity and telling a story through meaningful use of motion.

- Be responsive to user interaction and ready for the next interaction.
- Design for touch and intuitive interaction.
- Delight your users with motion.
- Create a UI that is immersive and compelling.

Be authentically digital

Take full advantage of the digital medium. Remove physical boundaries to create experiences that are more efficient and effortless than reality. Being authentically digital means embracing the fact that apps are pixels on a screen. Then you can design with colors and images that go beyond the limits of the real world.

- Connect to the cloud so that your users can stay connected to each other.
- Be dynamic and alive with communication.
- Use typography beautifully.
- Use bold, vibrant colors.
- Use motion meaningfully.

Do more with less

You can do more with less by reducing your design to its essence, and solving for distractions, not discoverability. Create a clean and purposeful experience by leaving only the most relevant elements on screen so people can be immersed in the content.

- Be great at something instead of mediocre at many things.
- Put content before chrome.
- Be visually focused and direct, letting people get immersed in what they love, and they will explore the rest.
- Inspire confidence in users.

Desktop browsers have quite a lot of chrome (menus, options, status bars, and so on) that is only sometimes useful. Typically, however, users open a browser to see a webpage, not to interact with the browser. Moving commands off the browser chrome and into the app bar or into charms helps users focus on what they care about.

Win as one

Work with other apps, devices, and the system to complete scenarios for people, like picking content from one app and sharing it with another. To provide a sense of familiarity, control, and confidence, take advantage of what people already know, like standard touch gestures and charms,.

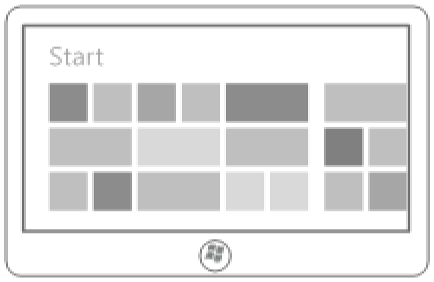
- Fit into the UI model.
- Reduce redundancy in your UI.
- Work with other apps to complete scenarios by participating in app contracts.
- To promote consistency, use our tools and templates.

Following these five Microsoft design principles helps you make the best choices when you design your app.

Anatomy of a Windows Store app

Windows Store apps have a new structure and differ from traditional desktop apps that have an always visible menu bar and modal dialogs. With Windows Store apps, you have a number of UI surfaces you can use, like the app canvas, charms, app bars, flyouts, and dialogs. Choosing the right surface at the right time can mean the difference between an app that is a breeze to use or a burden.

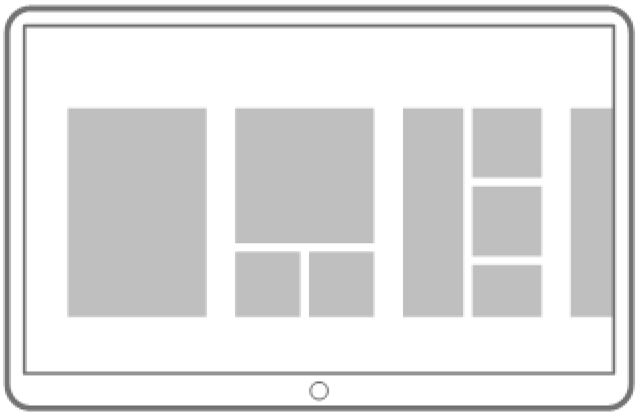
Tiles



Tiles appear on the Start screen and replace the application shortcuts that would have appeared on the desktop screen and on the old Start menu. People tap your app's tile to launch your app.

The app page, or canvas

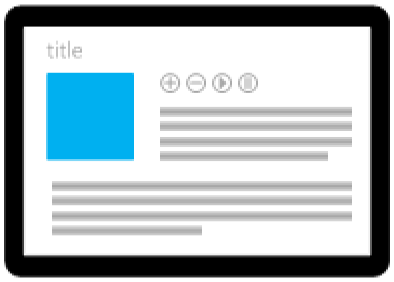
The app page, sometimes called the canvas, is the base of your UI. The canvas holds all of your content and controls. Whenever possible, integrate your UI elements into this base surface. For example, don't use a pop-up to display an error. Instead, you can smoothly show, hide, or shift the error message in the window with the built-in animations. Presenting your UI inline lets users fully immerse themselves in your app and stay in context.



You can create as many app pages as you need to support your user scenarios.

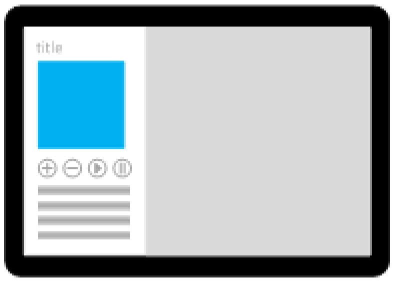
View states

People can snap your app so that they can use another app at the same time. Or, they can snap another app so they can use your app. You can design your app so that the content flows dynamically to optimize the user experience in each view state: full screen, snapped, or fill.



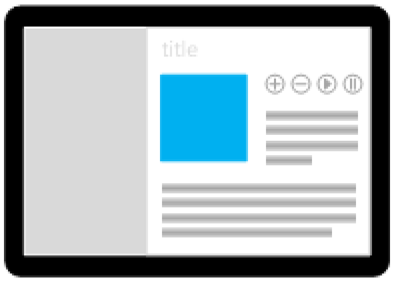
Full screen view

The app fills entire screen.



Snapped view

The app is snapped to a narrow region of the entire screen.

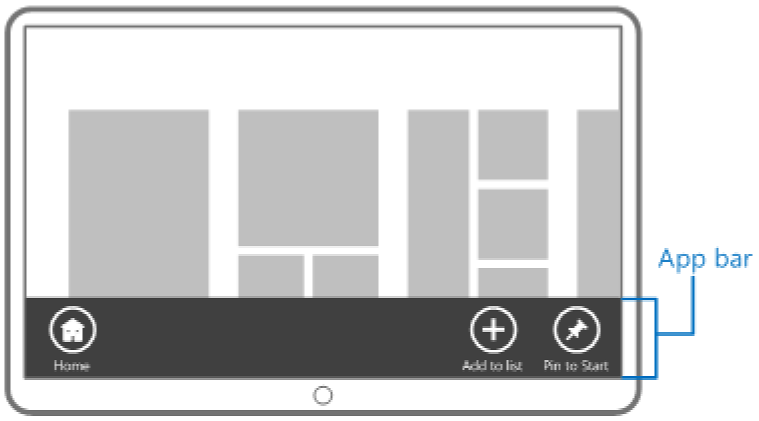


Fill view

The app fills remaining screen area not occupied by the app in the snapped view.

App bars

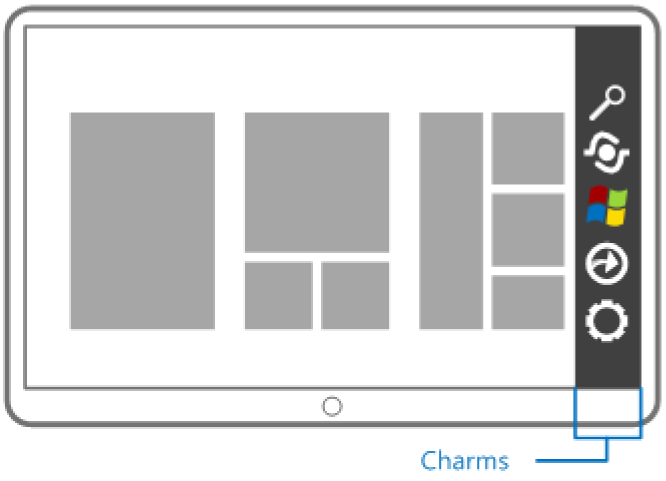
Outside of the app page, the app bar is the primary command surface for your app. Use the app bar to present navigation, commands, and tools to users. The app bar is hidden by default. It appears, and covers the content of the app, when people swipe from the top or bottom edge of the screen. People dismiss the app bar by swiping an edge or by interacting with the app.



Charms

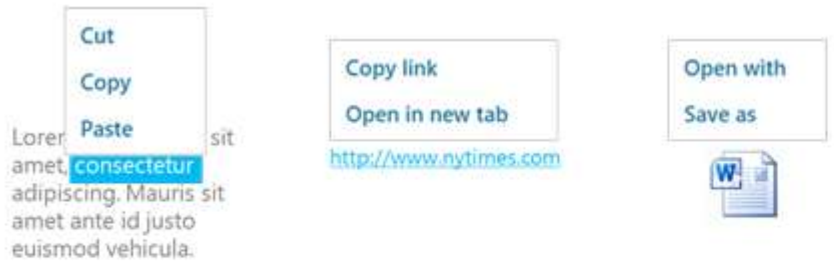
The charms are a set of buttons available in every app: search, share, connect, settings, and start. We believe these represent important actions that people want to complete in almost every app they use.

- **Search** People can search for content located your app or in another app, and they can search your app's content from another app.
- **Share** People can share content from your app with people or services.
- **Devices** People can connect to devices and send content, stream media, and print.
- **Settings** People can configure your app to their preferences and access user help.
- **Start** People can go directly to the Start screen.



Context menus

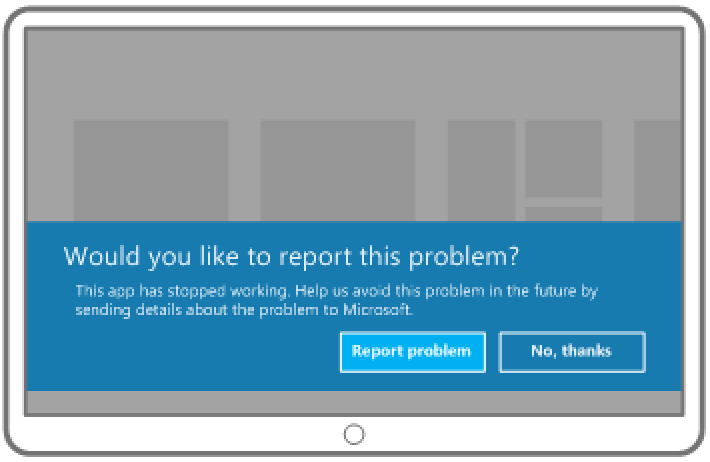
The context menu, sometimes called a popup menu, shows actions that people can perform on text or UI elements in an app. You can use up to five commands on each content menu, like cut, copy, or open with. This limit keeps the context menu uncluttered, easy-to-read, and directly relevant to the text or object that the commands act on.


Cut
Copy
Paste
Lorem consectetur sit amet, adipiscing. Mauris sit amet ante id justo euismod vehicula.
Copy link
Open in new tab
http://www.nytimes.com
Open with
Save as

Don't use context menus as the primary command interface for an app. That's what the app bar is for.

Message dialogs

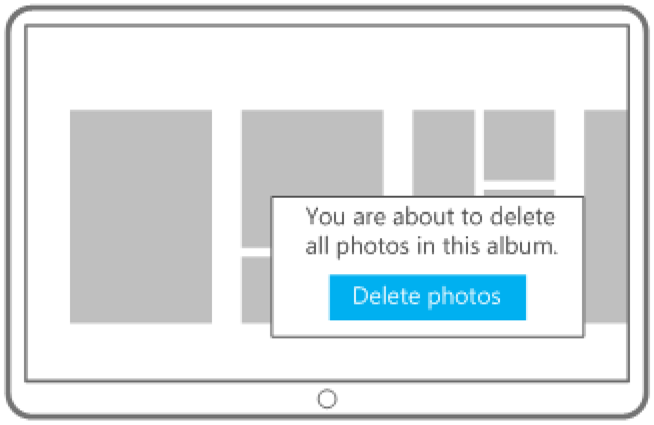
Message dialogs are dialogs that require explicit user interaction. They dim the app window and demand a response before continuing. Use message dialogs only when you intend to block people from using your app until they respond.


Would you like to report this problem?
This app has stopped working. Help us avoid this problem in the future by sending details about the problem to Microsoft.
Report problem
No, thanks

In the example above, the app window is dimmed, and the user must tap one of the two buttons to dismiss the dialog. That is, the message in the dialog cannot be ignored.

Flyouts

Flyouts show temporary, dismissible UI related to what people are currently doing in your app. For example, you can use flyouts to ask the user to confirm an action, to show a drop-down menu from a button on the app bar, or to show more details about an item. Flyouts are different from message dialogs. Unlike a dialog, you show a flyout only in response to a user tap or click, and you always dismiss the flyout when the user taps outside of it.



In the example above, the app stays active, and the user can tap the button or tap outside the flyout to dismiss it. That is, the message in the flyout can be ignored.

Toasts

Toasts are notifications that you show to people when your app is in the background. Toasts are good for updating people with information they want to know in real time, but it's ok if they miss. People tap on the toast to switch to your app and learn more.

People can find toast notifications disruptive and annoying, so be thoughtful about when you want to show a toast to users.

Design for form factors

Windows Store apps are at home on desktops, laptops, and slates. Design your apps to handle all these different form factors gracefully. Remember that people move between different devices, change the screen orientation, or shut everything off and on. Your Windows Store app needs to move, change, and react with them.

Touch, mouse, and keyboard input

When you design your app for touch input and manipulation, you get support for mouse and keyboard input for free. People can switch from one input method to another and not miss a beat of the app experience. Plug a keyboard into a slate? No problem. Your app adapts to your users' choices.

Device capabilities

Great apps take full advantage of the devices they run on. Windows 8 has built-in support for these device capabilities:

- **Accelerometers and other sensors** Devices come with a number of sensors nowadays. Your app can dim or brighten the display based on ambient light. It can also reflow the UI if the user turns the display, or react to any physical movement. Learn more about sensors.
- **Geolocation** Use geolocation information from standard web data or from geolocation sensors. This data can help your users get around, locate their position on a map, or get notices about nearby people, activities, and destinations. Learn more about geolocation.
- **Cameras** Connect your users to their built-in or plugged-in cameras. They can use the cameras for chatting and conferencing, recording vlogs, taking profile pics, documenting the world around them, or whatever activity your app is great at.
- **Proximity gestures** Let your users connect devices by physically tapping the devices together. This can light up experiences where you expect multiple users to be physically nearby (multiplayer games). Learn more about proximity gestures.

When planning your app's features, consider the devices your app might run on. Are some device capabilities required for your app to work well? Or can you get by without some? You declare which capabilities your app supports in your app manifest, but in the app itself, you can create fallbacks for optional capabilities. For example, let's say a travel mapping app lets users track their travels on a map, tag sites, include journal comments, send to social media, and add photos or videos from their trip. Geolocation would be a mandatory capability, but camera support could be optional. If the device doesn't have a camera, users could upload videos or photos taken on another device. Great apps cover all the options.

Design for form factors

Fluid, multiple views of your app

Windows 8 puts users in charge. You want your app UI to shine on any device, in any orientation, in whatever circumstance people decide to use it. When people change the orientation of their monitor or mobile device, your app gracefully reflows in response. When you design your app UI with fluid views, you get this behavior for free. Windows takes care of the rest.

- **Landscape** Design for landscape view first so your app looks good on all form factors.
- **Portrait** But remember some devices rotate! So optimize the layout of your content when in portrait view and retain functionality whenever possible.

The key to looking good in these views (as well as in snapped, fill, and fullscreen views) is defining layouts for the app in each view. When you plan ahead for each view, your app UI reflows pleasantly when a different view is triggered on the device.

Built-in graphical scaling

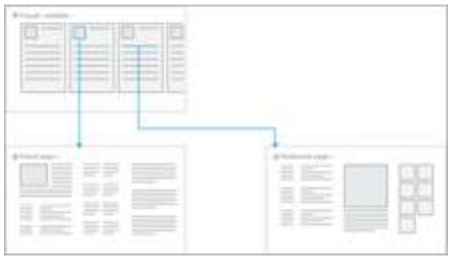
If people can access your app on multiple form factors, does that mean you have to design a different UI for every potential screen size Windows works on? That's a lot of different screen sizes! The answer is, not necessarily. Built-in scaling means your app and content always look great, whether on a small 7" device or on a big 30" monitor. You just need to use a fluid layout and make sure the graphics in your app look good when scaled.

Roaming data

What if people go from using your app on their work desktop to their slate at home? Their files, app state, and app preferences go home with them. They can pick up right where they left off, across different machines and user sessions.

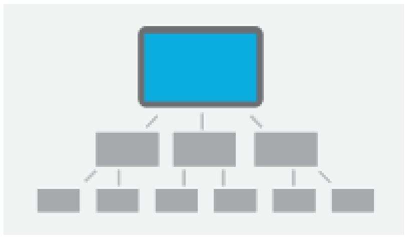
NAVIGATION, LAYOUTS, AND VIEWS

Navigation design



Learn how to organize the content in your Windows Store app so your users can navigate easily and intuitively. Using the right navigation patterns helps you limit the controls that are persistently on-screen, such as tabs. This navigation lets people focus on the current content.

Hierarchical system

	Most Windows Store apps in Windows 8 use a hierarchical system of navigation. This pattern is common and is familiar to people, but it is made even better by the Hub navigation pattern. This pattern makes Windows Store apps fast and fluid while still being easy to use. This pattern is best for apps with large content collections or many distinct sections of content for a user to explore.

Layers in the hierarchy

The essence of Hub design is the separation of content into different sections and different levels of detail.

Hub pages

Hub pages are the user's entry point to the app. Here content is displayed in a rich, horizontally panning view that allows users to get a glimpse of what's new and available.

The Hub consists of different categories of content, each of which maps to the app's Section pages. Each Section should bubble up content or functionality. The Hub should offer plenty of visual variety, engage users, and draw them in to different parts of the app.

Section pages

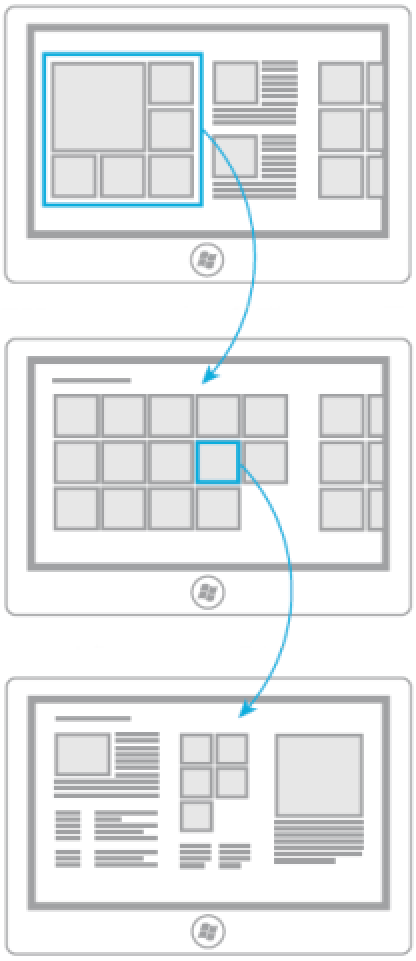
Section pages are the second level of an app. Here content can be displayed in any form that best represents the scenario and content the Section contains.

The Section page consists of individual items, each of which has its own Detail page. Section pages can also take advantage of grouping and a panorama style layout.

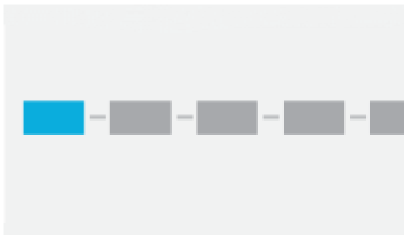
Detail pages

Detail pages are the third level of an app. Here the details of individual items are displayed, the format of which vary tremendously depending upon the particular type of content.

The Detail page consists of item details or functionality. Detail pages may contain a lot of information or may contain a single object, such as a picture or video.



Flat system



Many Windows Store apps in Windows 8 use a flat system of navigation. This pattern is often seen in games, browsers, or document creation apps, where the user moves between pages, tabs, or modes that all reside at the same hierarchical level.

This pattern is best when the core scenario involves fast switching between a few pages or tabs.

Content pages

The essence of the Flat system is the separation of content into different pages.

Top app bar

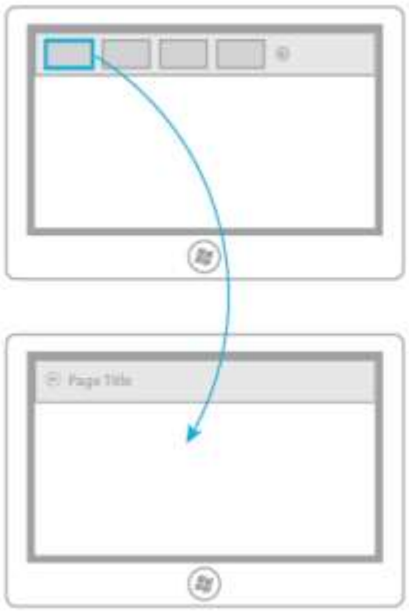
The top app bar is great for switching between multiple contexts. Examples include tabs, documents, and messaging or game sessions.

This app bar is a transient element that resides at the top of the screen, and is made visible when users swipe from the top or bottom edge. While formatting of items in the bar can vary, a typical treatment is the use of a simple thumbnail.

Switching

Unlike the hierarchical system, there is typically no persistent back button or navigation stack in the flat system, so moving between pages is usually done through direct links within the content or the top app bar.

You can choose to include other functionality within the top app bar, such as adding a '+' button to create a new tab, page, or session.



Navigation anatomy

The following show the anatomy navigating between sections in an app, between different levels in the hierarchy, and within a single app page.

1. Header and Back button

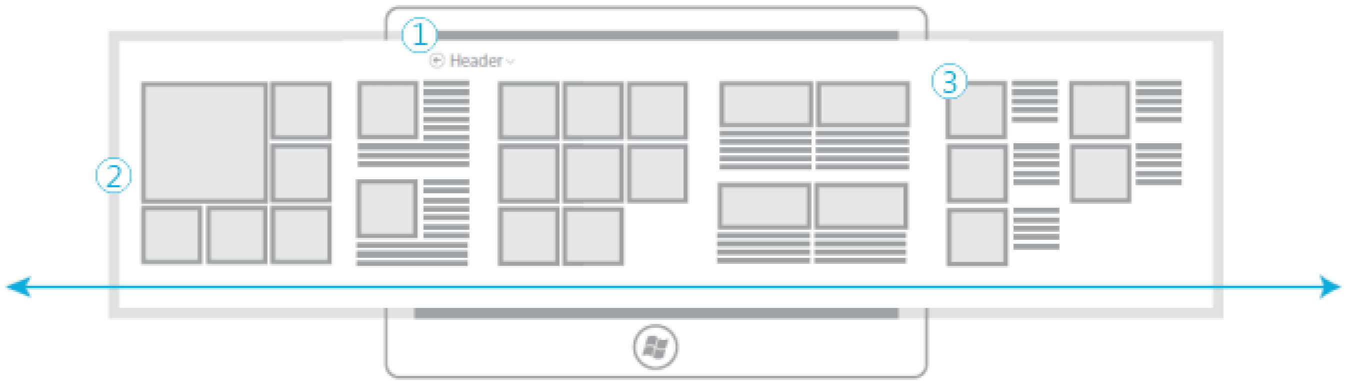
   The header labels the current page and is useful for way finding. The Back button makes it fast to get back to where you were.

2. Hub page

   The Hub page pulls information from different areas of the application onto one screen. It gives the user a bird's-eye view of everything available in the app.

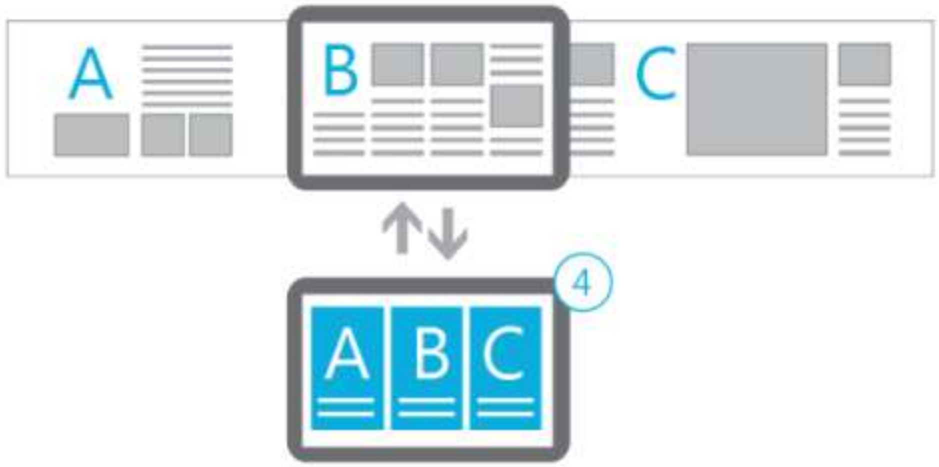
3. Content sections, or categories

   Content sections can be formatted to best display the functionality or items they promote.



4. Semantic zoom: navigating between levels in a hierarchy

   Semantic zoom makes scanning and moving around a view fast and fluid, especially when the view is a long panning list.



5. Top app bar

   The top app bar contains transient access to navigation controls or to other areas of the app.

6. Header menu

   The header menu is available from anywhere in the app, and allows users to jump quickly from one section of the app to another.

7. Home link

   The home link, at the bottom of the header menu, is a quick way to get back to the root of the app.

8. Bottom app bar

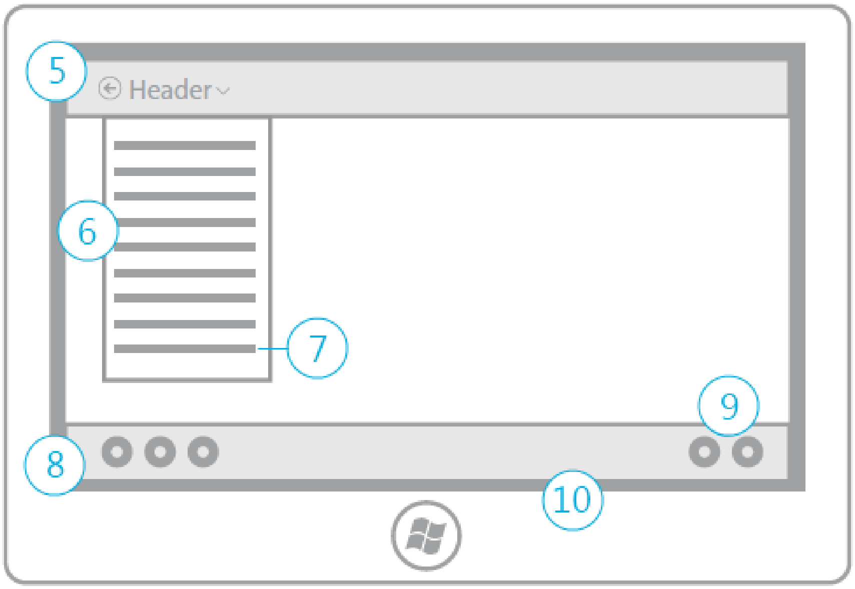
   The bottom app bar contains transient access to commands relevant to a particular view.

9. View/Sort/Filter

   These commands change how content is displayed within a specific view. The best place for them to reside is in the app bar.

10. Edge

    Swiping from the edge of the screen is what makes the app bars and charms appear.

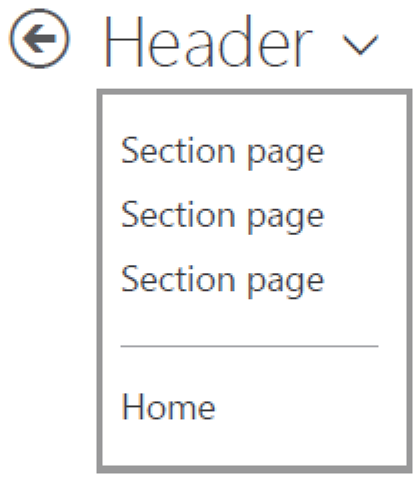


Navigating with the edge swipe

	Users can navigate within apps and throughout the system by swiping a finger or thumb from an edge. In order to use Windows Store apps efficiently, users learn what each of the following edge swipes does: • Swiping from the bottom or top edge of the screen reveals the navigation and command bars. • Swiping from the right edge of the screen reveals the charms that expose system commands. • Swiping from the left edge cycles through currently running apps. • Sliding from the top edge toward the bottom edge of the screen closes the current app. • Sliding from the top edge down and to the left or right edge snaps the current app to that side of the screen.

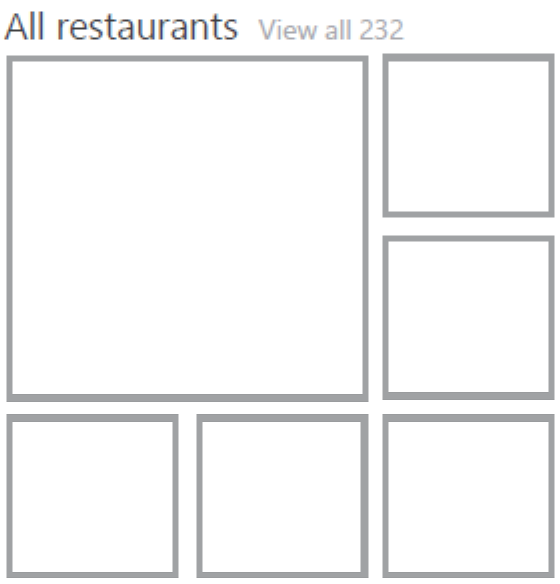
Navigating with header menus and section labels

We will use a sample app called Food with Friends to illustrate a pattern for using the back button, header menu, and content sections to navigate a Windows Store app.

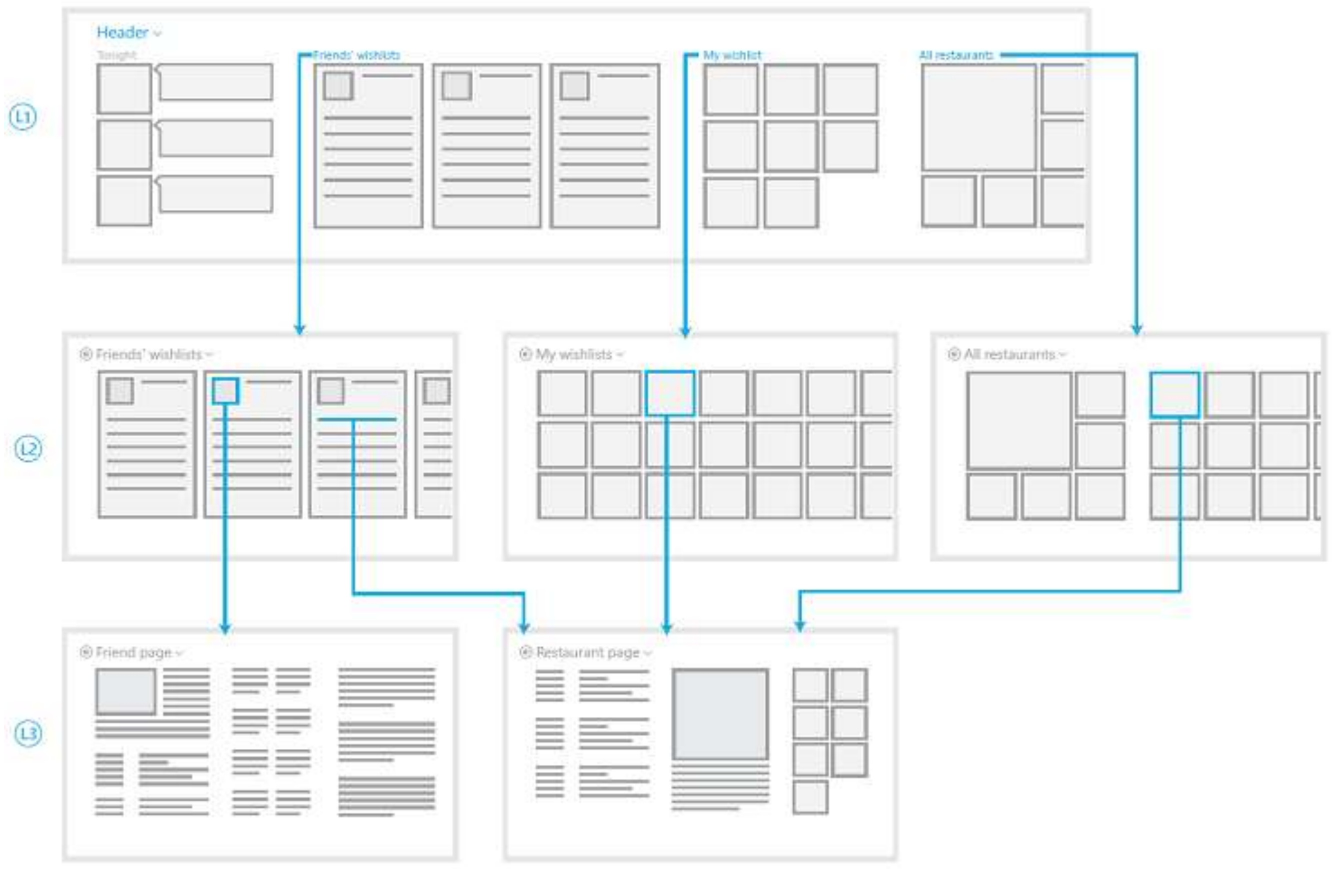


The header menu contains a link to each section page (level 2) as well as a link back to the hub (level 1). This enables users to move around the app quickly. The menu appears at each level and on every page of the app, making it an efficient and reliable way for users to get where they want to go.

Users can tap on the section label to drill in to the corresponding page for that section. Provide a visual cue, like **View all (x)**, to indicate to users that there are more items in this section that what is shown in the hub. Using this pattern avoids the need to use a tile space or place a link within the content.



Using this pattern, this is what the navigation diagram would look like for the Food with Friends example. This is a simplified diagram that shows only canonical examples of navigation elements, as representatives of everything that's interactive.



Navigating with filters, pivots, sorts, and views

Another part of app navigation is determining when, where, and how to give users more control over the way they experience content. Filters, pivots, sorts, and view switchers are all things to consider in your app design.

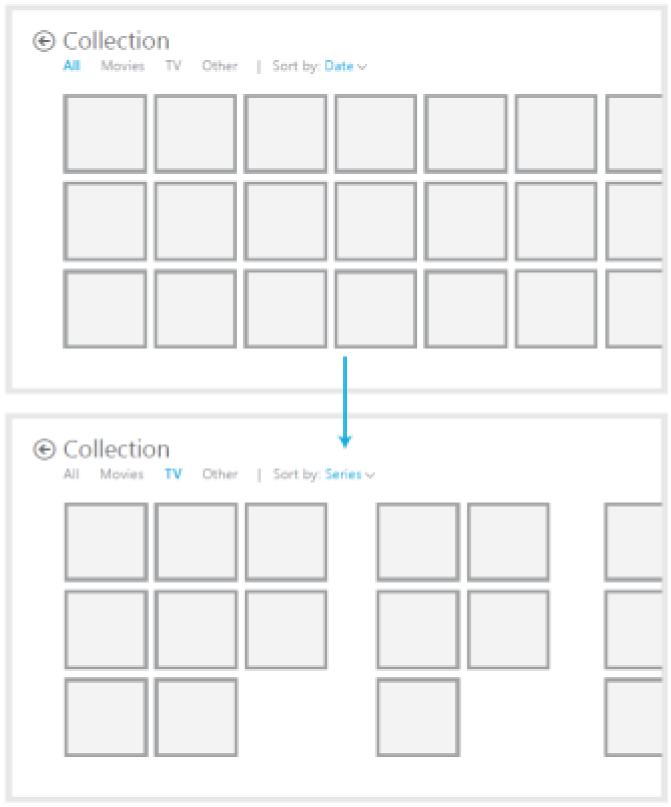
Term	Definition	Example
Filter	Removing or hiding content within a data set, based on some criteria.	When looking for a game to play, you might choose to view only those games categorized as "adventure."
Pivot	Reorganizing content within a data set, based on some criteria.	When looking at a music collection, you might choose to organize songs by artist, album, or genre.

Sort	Changing the order in which content is displayed within a data set.	When browsing for an article to read in a news app, you might choose to see the most recent articles listed first.
View	Changing the style or method in which content is displayed.	When browsing for a place to eat in a restaurant-finding app, you might choose to view restaurants on a map instead of in a list.

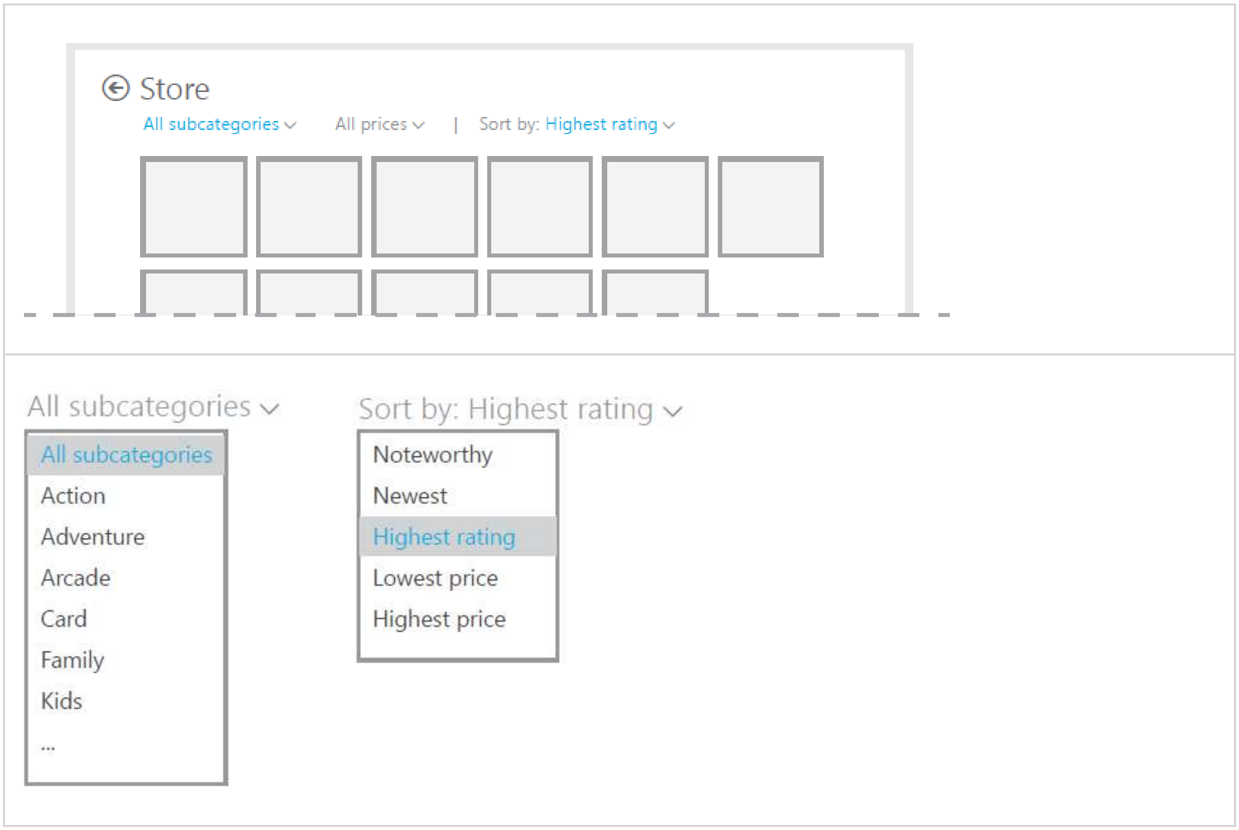
On-canvas Use on-canvas controls for filtering, pivoting, or sorting when finding an item is a primary task, like in a collection or search result page.

Controls should go into the app bar if the focus of the app is on browsing for content, as with a magazine or shopping app.

Page filters and sorts For filtering and sorting content within a collection view, filter, and sort commands can be placed in a row between the header and content. In the following example, the view is filtered to show only TV episodes, sorted and grouped by series.



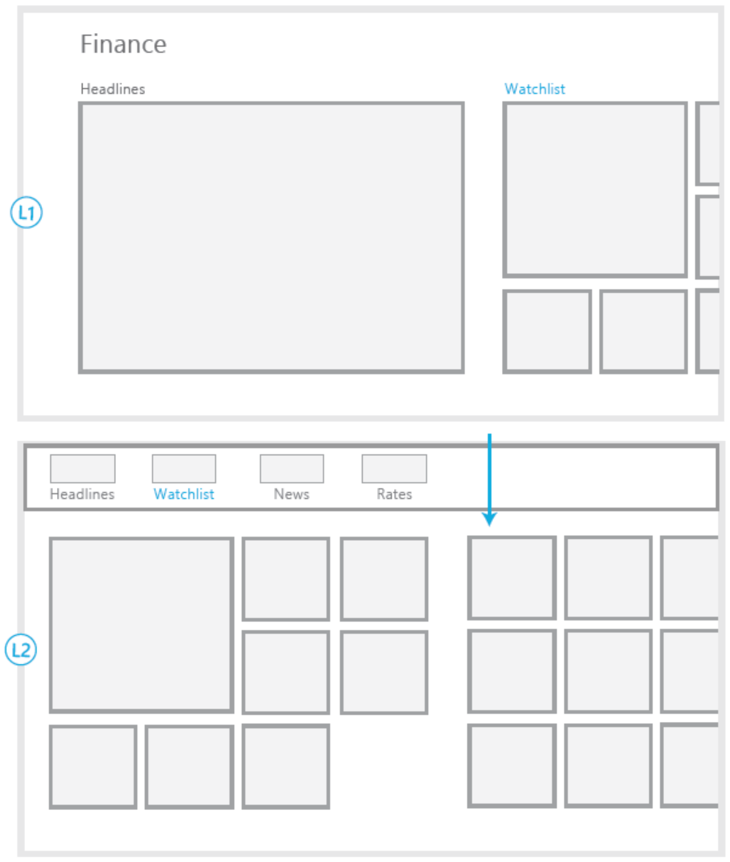
In this example for a marketplace app, drop-down selection controls filter the content for the current view. As the menus show, the currently active filter appears selected in the drop-down list.



The top app bar

The top app bar is used primarily for navigating sections or pages of an app that use the Flat navigation pattern. It can also be used along with the Hierarchical pattern, in lieu of the header menu, as a means for providing global navigation controls. The top app bar should show up on every page and at all levels of the app to provide users with a convenient, deterministic way of navigating around.

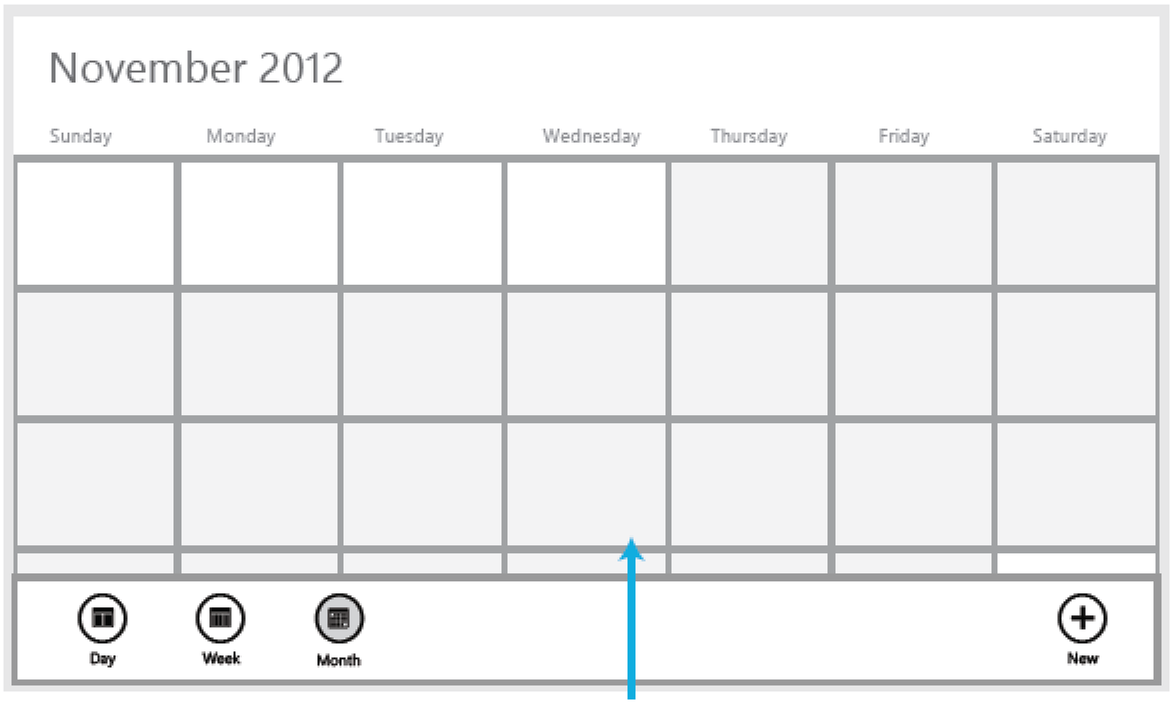
In this finance app example, the hub (L1) promotes sections of the app (Headlines, Watchlist) to the hub, and the section headers link in to them. At the section level (L2), when the top app bar is invoked by swiping the top or bottom edge, the user has access to the root and all other sections of the app.



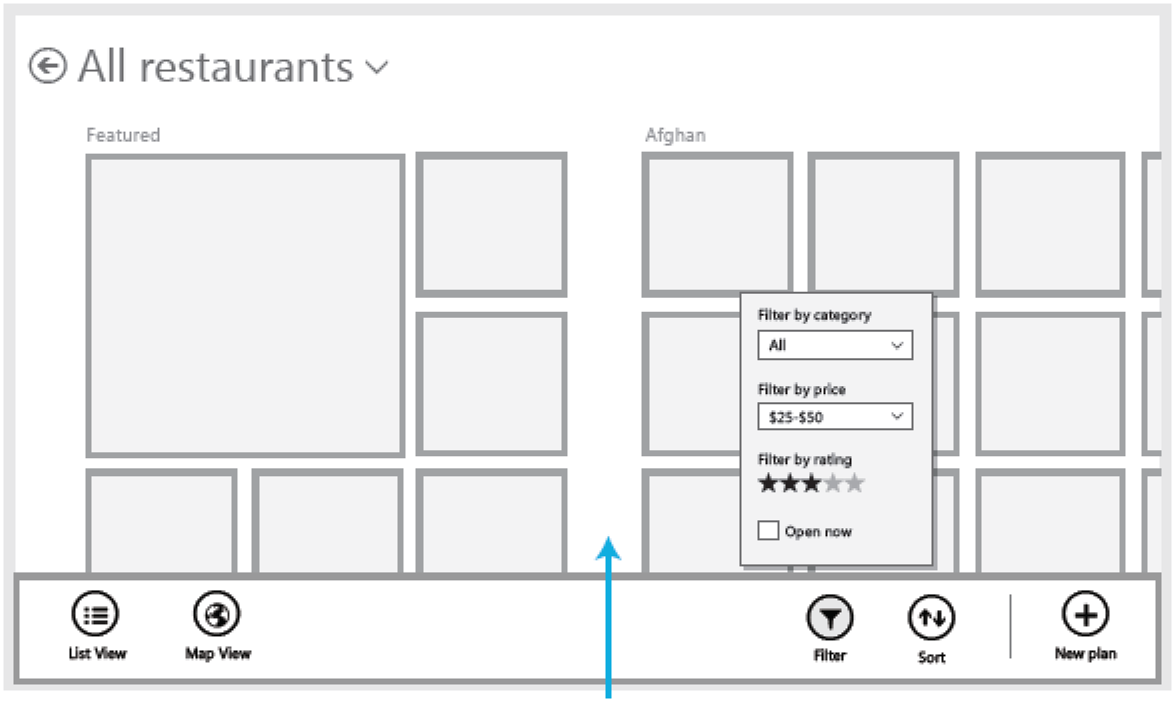
App bar view switching

The app bar is used primarily as a commanding surface, but it can also be used to alter how content is being viewed. Switching views, pivoting, filtering and sorting can all be done by using the app bar. Don't use the app bar for navigating from one place in the app to another. All app bar items should act on the content currently in view.

In this calendar app example, the view defaults to a month view, which this app has optimized for. Commands to choose other calendar views are in the app bar, accessed by swiping from the top or bottom edge. Other commands, such as making a new appointment, may appear in the bar as well.



In the **All Restaurants** page of the Food with Friends example, options for viewing items as a list or map are available, as are filtering and sorting the view based on certain criteria such as cost, location, and rating. Here, filtering options are exposed as controls in a menu Flyout.



Guidelines for navigation links

In a Windows Store app, you can use links to navigate to pages in your app or in your app's web context. You can also link to external pages outside of your app's web context; these links launch the browser.

When to use a link

Only use links for navigation, such as opening an external URL or switching to another page in the same app. Don't use links to perform actions other than navigating.

Dos and don'ts

Do	**Put a tooltip on every link.** That way, if the user's finger covers the link, the user can still see what the link does.
	When navigating to an external site, put the domain name inside the tooltip and style it with a secondary font color. Adding the domain name to the tooltip lets people know that they're about to navigate to an external site so they aren't surprised when they click the link. Because the tooltip from "title" attribute doesn't support styling, use the **Tooltip** control instead. For Windows Store apps using JavaScript, use the win-text-domain CSS class (provided by the Windows Library for JavaScript style sheets) to style the domain portion of the URL. It's enough to just show the top-level domain.
	When the user doesn't care whether he or she has visited a link, style the visited state for that link so that the link always looks the same, whether or not the user has clicked it. The default style for a visited link makes it look different than a link that hasn't been visited. Sometimes the user doesn't care whether a link has been visited. This is usually the case for links that are a part of your app's main navigation.
Don't	**Don't make the link text too long.** Keep the link text concise. If you want to provide more information, put it inside the link's tooltip.

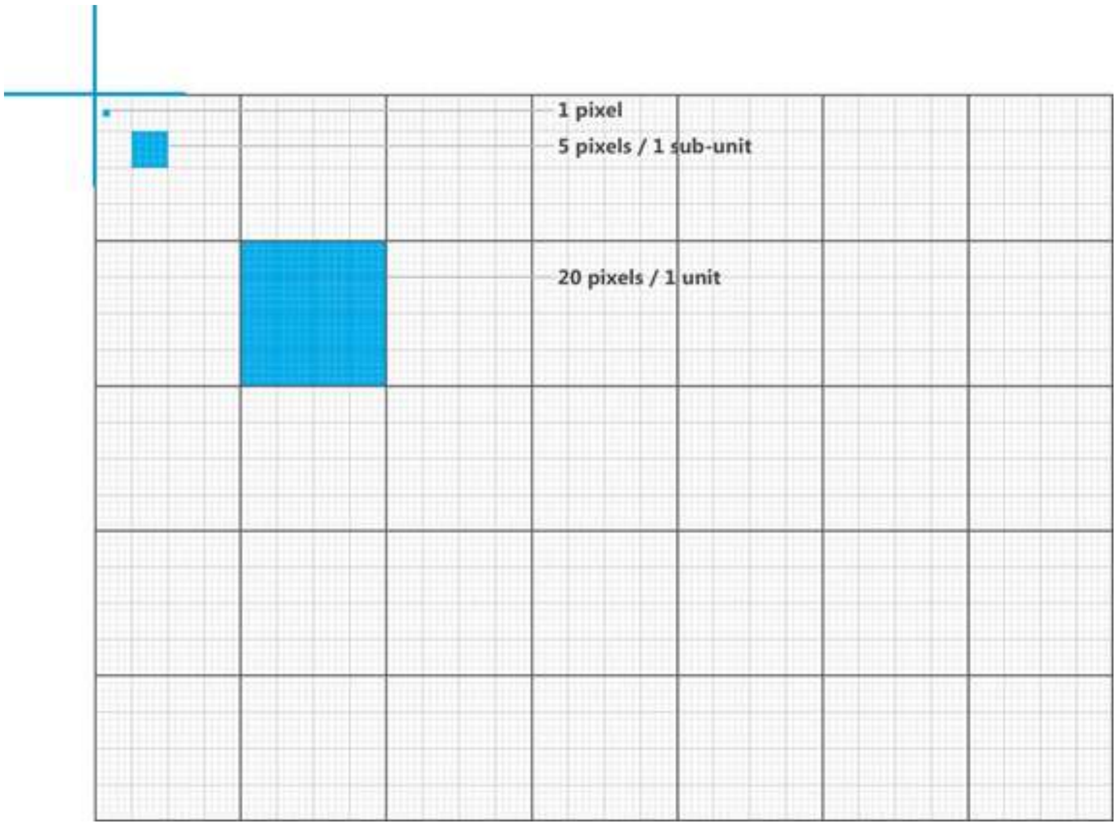
Page layout design

You can use the grid-based layout pattern described here to lay out UI elements on your app pages. Follow this consistent pattern for margins, page headers, gutter widths, and other such elements. This pattern speeds your app design, maintains unity across apps, and helps people easily understand interactions across the system.

The signature characteristic of this pattern is silhouette with a wide margin on the top, bottom, and left edges. The wide margin helps users understand the horizontal panning direction of the content.

What is the grid system?

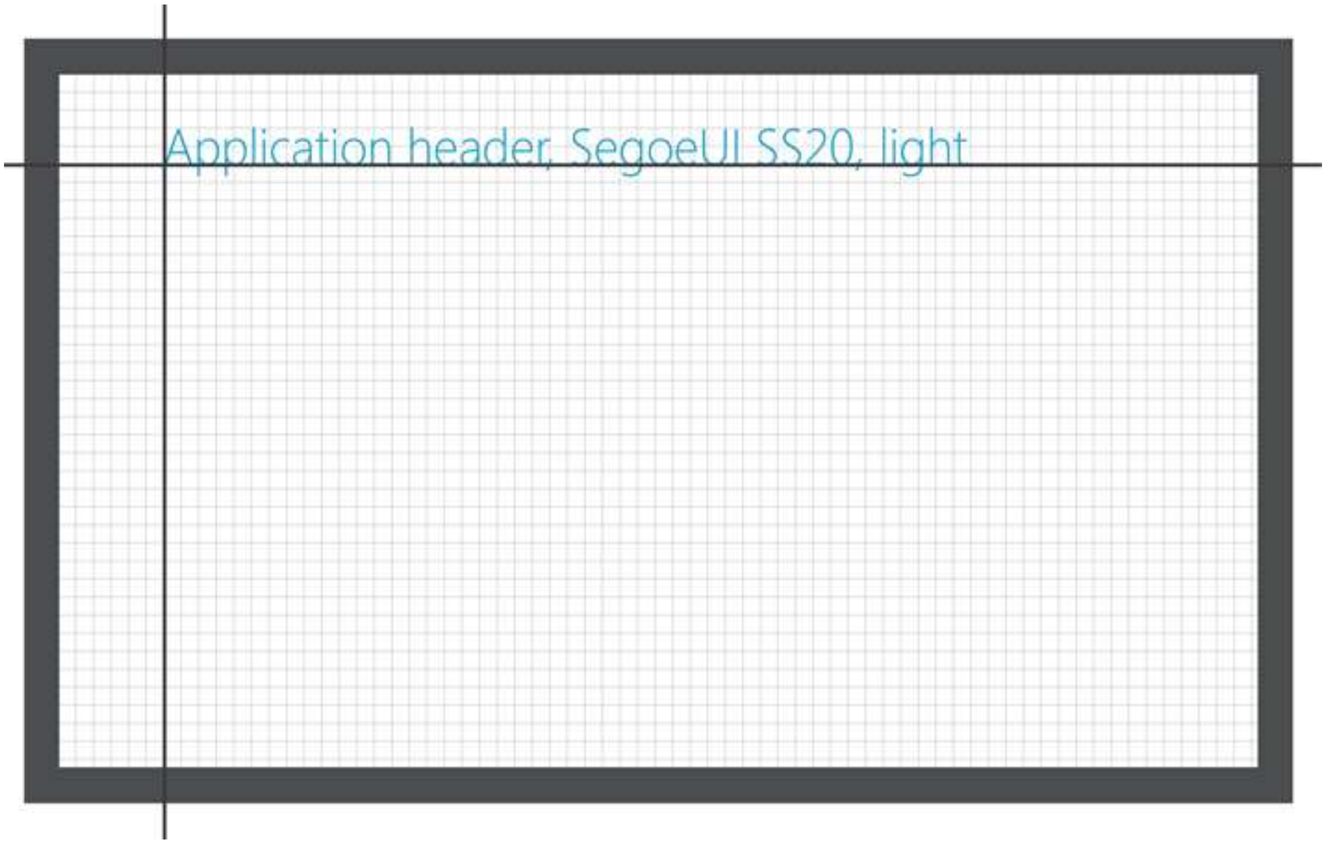
The grid system is built into the developer templates, and we designed our controls and collections with the grid system in mind. The grid is made up of units and subunits. The basic unit of measurement is the unit. One unit equals 20 × 20 pixels. Each unit is further divided into subunits of 5 × 5 pixels. There are 16 subunits per square unit. This image depicts the grid in the upper-left corner of a screen. The image is enlarged to show pixels, subunits, and units.



The entire app page can be laid out using this grid as a foundation.

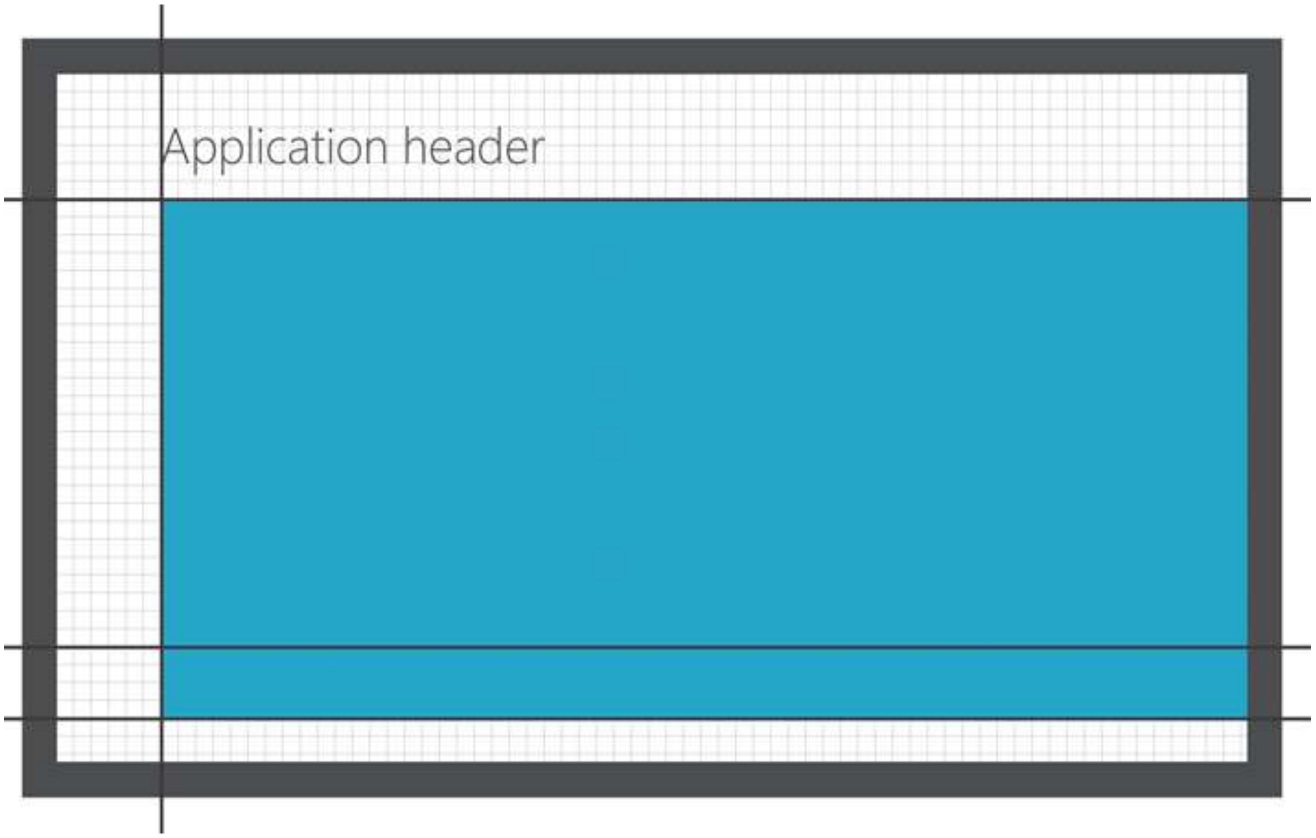
App page header

In the grid system, the baseline of the app page header is 5 units, or 100 pixels from the top. The left margin for the page header is 6 units, or 120 pixels. The typography is SegoeUI Stylistic Set 20, lightweight.



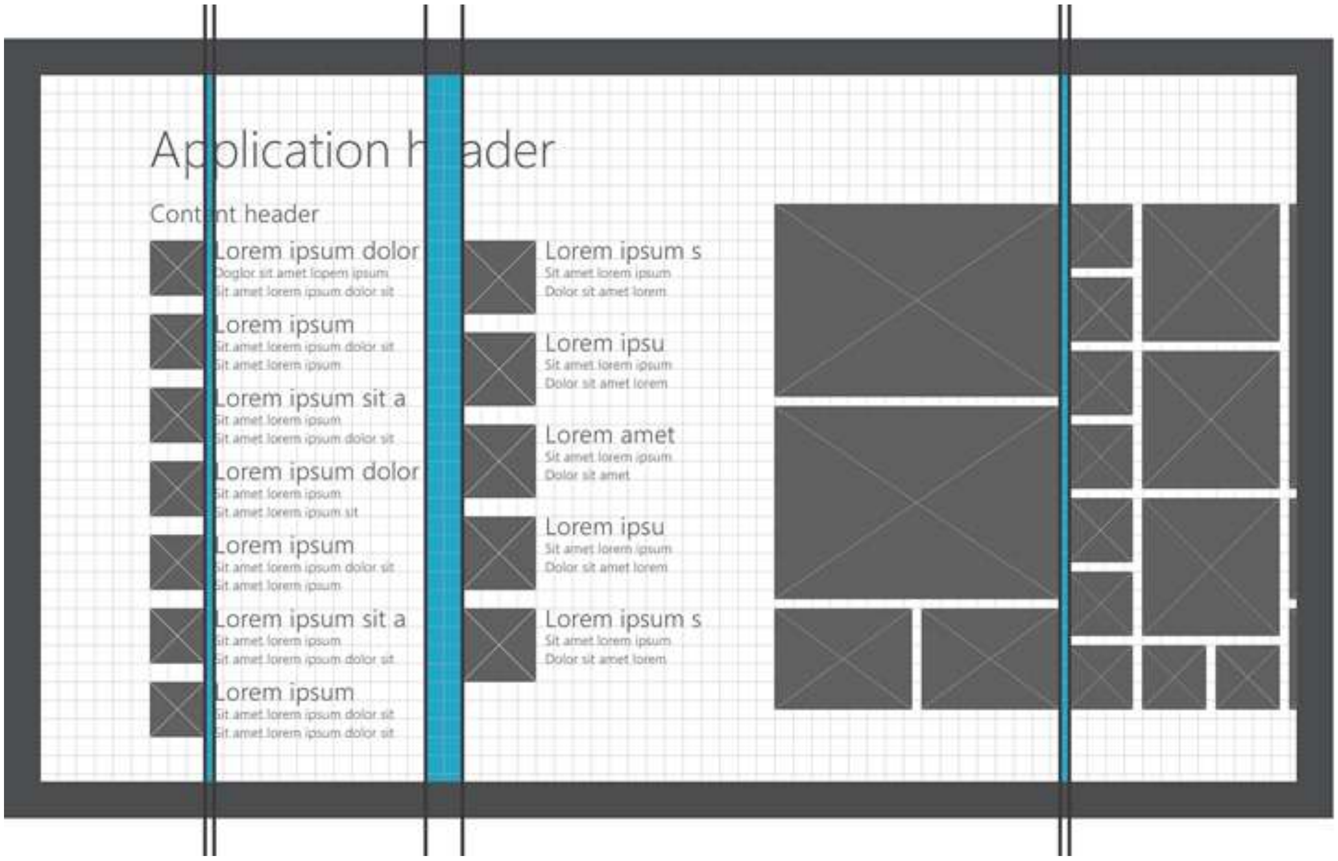
Content region

In the grid system, the content region has a top margin of 7 units, or 140 pixels. The left margin is 6 units, or 120 pixels. The bottom margin is flexible. For horizontally panning content, it's between 2.5 units (50 pixels) and 6.5 units (130 pixels). For vertically panning content, the top and left margins remain the same. There is no specified bottom margin because the content scrolls off the screen.



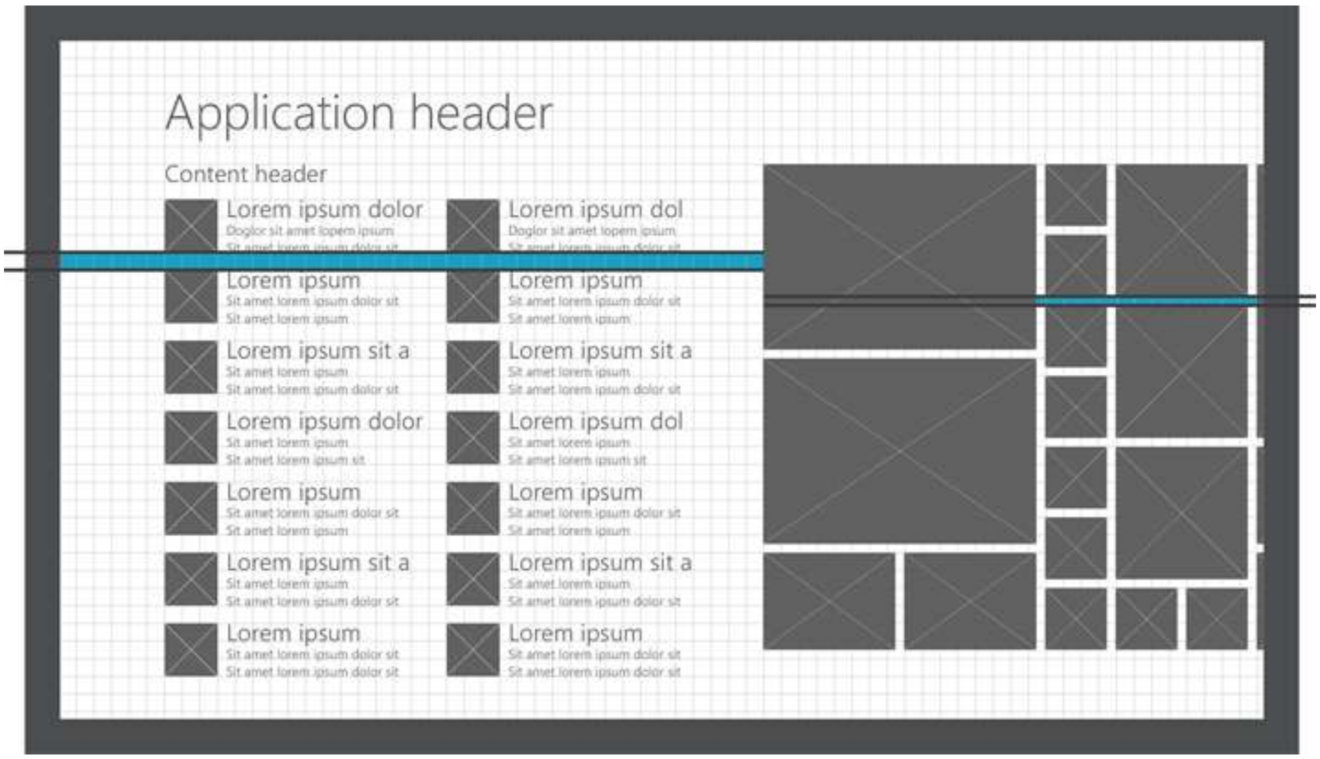
Horizontal padding

Horizontal padding between content items varies depending on the items. Hard-edged items (like images and user tiles) are separated from accompanying text by 2 subunits, or 10 pixels, of padding. Hard-edged items on their own have 2 subunits, or 10 pixels, of padding between columns. Lists in columns are separated by 2 units or 40 pixels of padding.



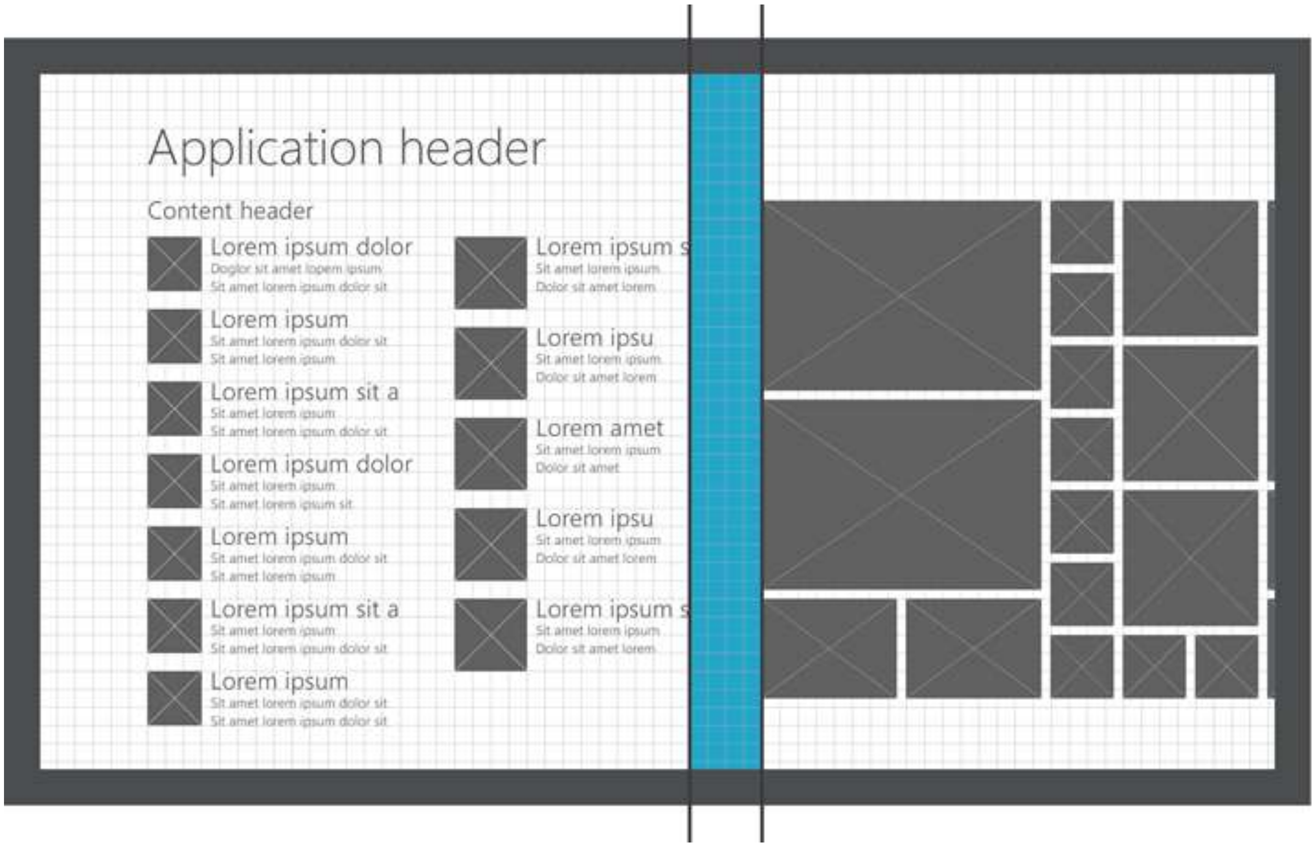
Vertical padding

Vertical padding between content items also varies depending on the types of items. Tile and text lists have 1 unit, or 20 pixels of vertical padding between items in a row. Hard-edged objects have 2 subunits, or 10 pixels, of padding between items in a row.



Horizontal padding between groups

The padding between groups is 4 units, or 80 pixels. This extra padding helps people easily distinguish one group from another, especially when panning across many groups.

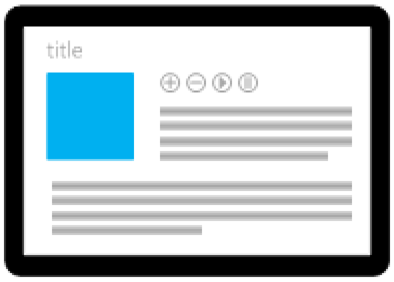


View state design

In Windows 8, a view is how the content of your UI adapts to how a user accesses and manipulates a Windows Store app. An app that is designed to support multiple views works well on devices of various sizes and orientations. It also lets users manipulate the content to fit their needs.

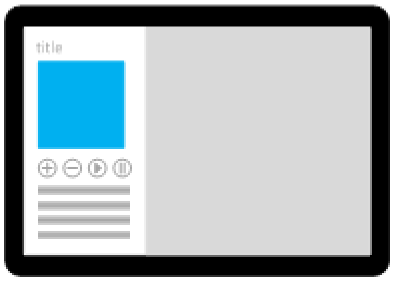
View state

View state refers to the three ways a user can choose to display your Windows Store apps: snap, fill, and full-screen as shown in the following images.



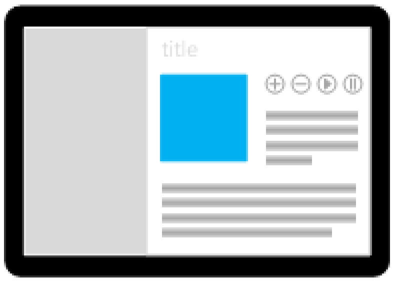
Full screen

App fills entire screen.



Snapped

App is snapped to a narrow region of the entire screen.



Fill

App fills remaining screen area not occupied by the app in the snapped state.

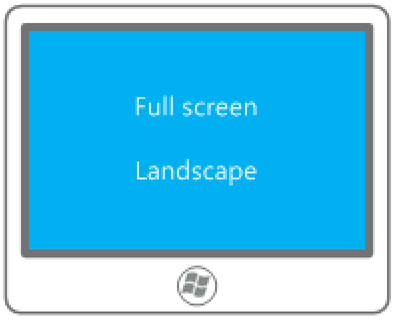
Because users can work with up to two apps at a time, you should provide layouts that are fluid and flexible enough to support all three states.

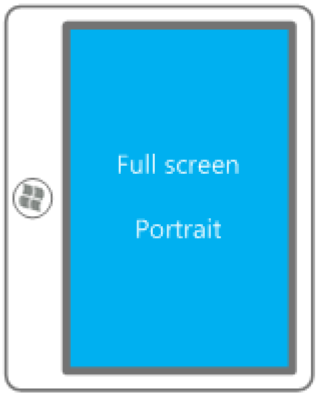
View state design

When you plan for full screen, snap, and fill views, your app's UI will reflow smoothly and gracefully to accommodate screen size, orientation, and user interactions.

Screen orientation

Users can rotate and flip their tablets, slates, and monitors, so ensure that your app can handle both landscape and portrait orientations.



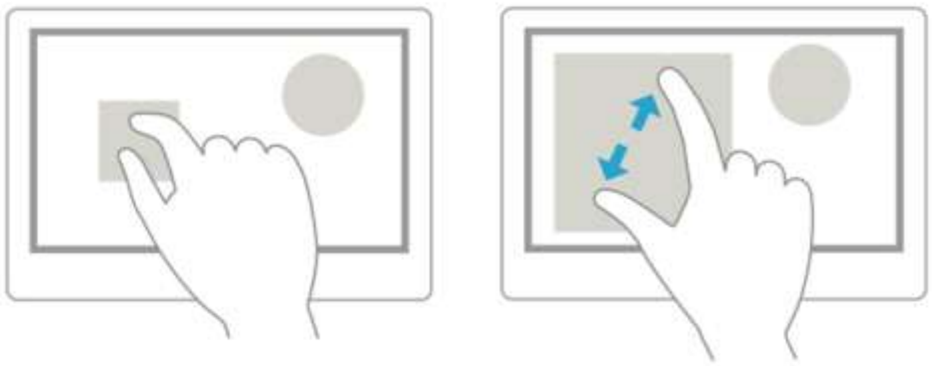


User interactions

Great apps help users do what they want with the content in the UI.

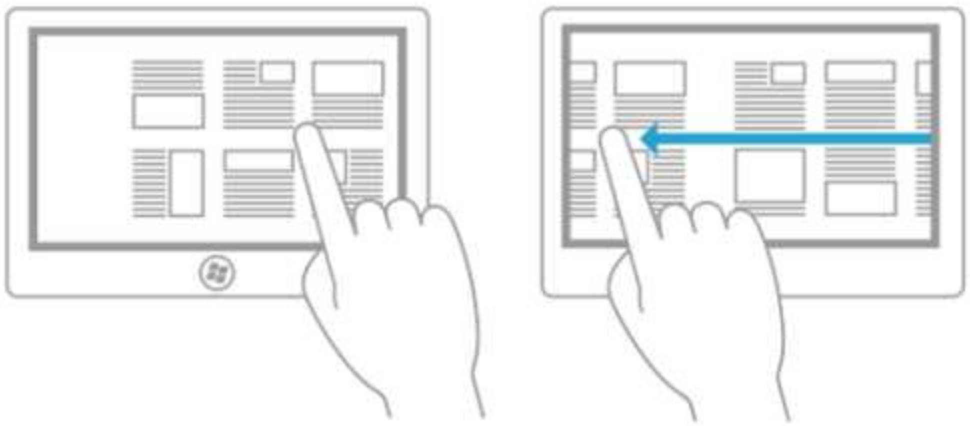
When you design your app, you should consider which regions of the UI should support panning and scrolling, optical and semantic zooming, and object resizing.

Create views that let users resize and zoom in on the content of your app.



Let the UI surface overflow the screen area onto more "pages" if necessary. In these cases, enable panning and scrolling to let users explore these large UI surfaces and discover the content on the additional pages.

View state design



Guidelines for views

These guidelines can help you build apps that provide an elegant experience across form factors, display sizes, and view states through consistent and predictable management of the user interface.

Follow these guidelines is designing your layout:

Do

- Use fluid layouts.
- Use media queries to identify the view state and resolution being targeted.
- Use layout features to adapt to display size.
- Use controls that are designed for fluid layout, like ListView.
- Consider vector-based UI (SVG, XAML) for application resources.

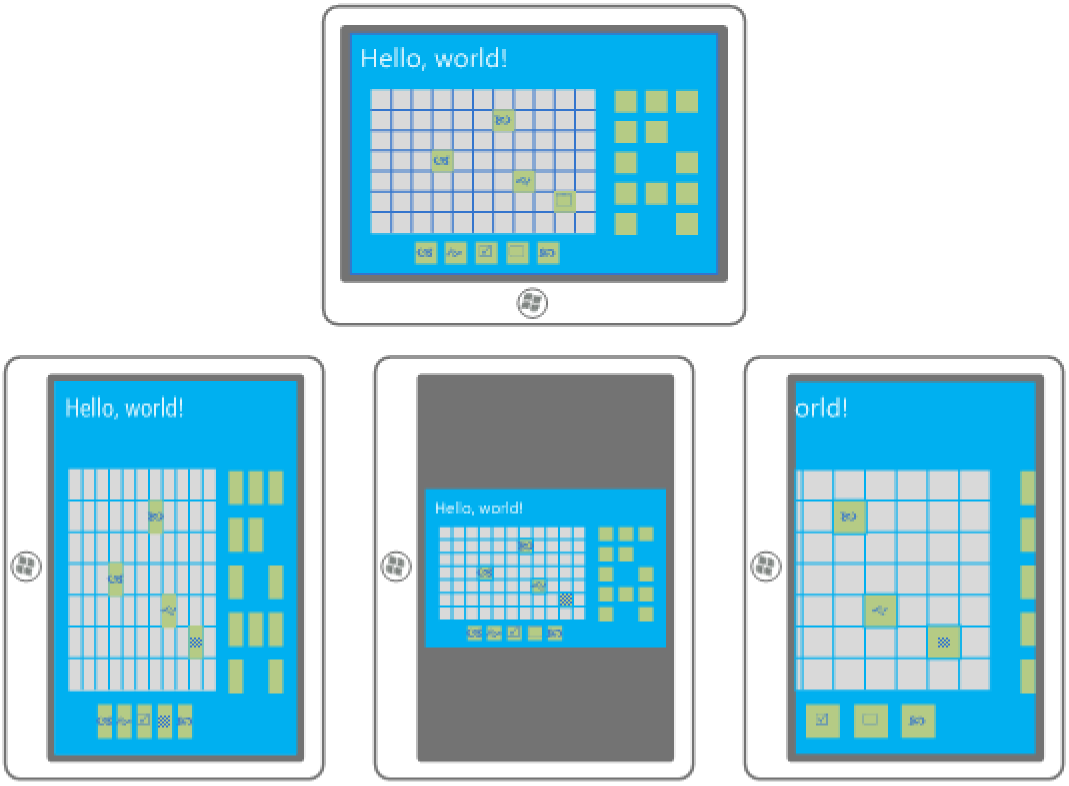
Don't

- Use static layouts.
- Use absolute positioning because it constrains your UI from responding to changes in view state and orientation.

View state design

You have two options for managing layout: static and fluid. Fluid layouts shrink, grow, or reflow to adapt to the visual space available on a device while static views do not. We recommend your use fluid layouts.

Avoid static layouts Static layouts can become stretched, shrunk, or clipped across different form factors and display sizes, as shown in these images.



Use fluid layouts Fluid Views encompass three primary layout concepts: layout management, grids, and flexible controls.

Layout management involves modifying the layout and visibility of UI elements for different views. Your best choice is to use CSS3 media queries to discover properties of the display device, such as screen dimensions and orientation. That way you can specify a layout that best suits the constraints of the display.

Layout management is most useful for applying coarse metrics to the UI. Grid layouts and spacing are used to provide another level of UI flexibility.

Guidelines for scaling to pixel density

Windows scales applications to ensure consistent physical sizes for UI elements regardless of the pixel density of the screen.

Standard Slate
10.6" 1366x768
148 PPI
HD Slate
10.6" 1920x1080
208 PPI

Without scaling, as the pixel density of a display device increases, the physical sizes of objects on screen get smaller. When UI would otherwise be too small to touch and when text gets too small to read, Windows scales the system and app UI to a percentage:

- 100% when no scaling is applied
- 140% for 1920 x 1080 devices that have a minimum DPI of 174
- 180% for 2560 x 1440 devices that have a minimum DPI of 240

Note For the Logo and WideLogo images specified in the manifest, you can specify an additional 80% scale percentage. For best results, provide all four scales of these images.

You don't have to do anything to get scaling for your app, but you do need to follow these guidelines to ensure that your app looks great when scaled.

Do **Use scalable vector graphics** Use SVG for JavaScript apps and XAML for C#/C++/VB apps. Windows scales these formats for you automatically, without noticeable artifacts.

Use resource loading for bitmap images in the app package For bitmap images stored in the app package, provide a separate image for each scale (100%, 140%, and 180%), and name your image files by using the "scale" naming convention. Windows loads the right image for the current scale automatically.

Save multiple versions of the image by using a file name or folder naming convention.

Option #1 - File naming convention:

```
...\test.scale-100.jpg
   \test.scale-140.jpg
   \test.scale-180.jpg
```

Option #2 - Folder naming convention:

```
...\scale-100\test.jpg
   \scale-140\test.jpg
   \scale-180\test.jpg
```

In markup, specify images without using the naming convention.

HTML

```
<img src="test.jpg" width=80 height=80/>
```

XAML

```
<Image Grid.Row="0" Grid.Column="1" x:Name="testImage" Source="test.jpg"
Margin="2,2,2,2"/>
```

Use the resolution media query for remote web images If your application is a JavaScript app and you have a remote web image, use the CSS @media min-resolution media query with a background-image to replace images at runtime.

For a Windows Store apps using C++, C#, or Visual Basic that uses remote web images, you need to query the **DisplayProperties.ResolutionScale** property to determine which remote web image to load.

Use the file access thumbnail APIs for user images on the file system If your app loads user images from the file system, these APIs automatically retrieves thumbnails corresponding to the current scale.

Manually load images based upon scale percentage at runtime If your app programmatically loads images at runtime, use the Windows Runtime APIs to determine the scale and manually load images based upon scale percentage.

Specify width and height for your images If you know the scale, specify a width and height on images, instead of using auto sizing, to prevent layouts from changing when larger images are loaded.

Use grid-units and sub-units Where possible, use the grid-defined sizes of 20px for major grid-units and 5px for minor grid-units. These sizes ensure that layouts aren't affected by pixel shifting due to pixel rounding. Any sized unit that is divisible by 5px does not experience pixel rounding.

You should avoid the following.

Don't **Don't use smaller images that are scaled up** Because images are scaled by default, images look blurry at 140% scale on HD slates if only 100% scale images are available. You should ensure that your images look great on the higher 140% scale by using the scaling guidance.

Don't use larger images that are scaled down Larger images that are scaled down show scaling artifacts and jagged edges on standard slates. Photographs are the only exception as they can look good when scaled down. You should ensure that your images look great on the 100% scale using the above guidance.

Avoid specifying sizes that aren't multiples of 5px Units that aren't multiples of 5px can experience pixel shifting when scaled to 140% and 180%.

Guidelines for scaling to screens

Windows 8 runs on a variety of screen sizes, from a small screen on a tablet, to a medium laptop screen all the way up to a large desktop or all-in-one screen. Windows Store apps can run on any of the screen sizes that Windows 8 supports. In general, larger screen sizes also have higher screen resolutions. The result is that on these larger screens there is more viewable area for your app to take advantage of.

The following terms are used in this document.

Screen size	The physical size of the screen, in inches. The size is usually measured on the diagonal.
Screen resolution	The number of pixels the screen supports, in horizontal and vertical dimensions. For example, 1366x768.
Aspect ratio	The shape of the screen as a proportion of width to height. For example, 16:9.

Screen size

The biggest impact that screen size has on Windows Store apps is actually the screen resolution. On larger screens, there is a higher screen resolution, and therefore more available space or screen real-estate for your application. Users expect to see more content and more functionality on larger screens.

Managing this screen real estate takes consideration by the app developer and designer. The parameters that you define for the layout at design and development time determine how the app looks on large screens.

The platform, controls, and templates have all been designed to accommodate different screen sizes. Although much of your app's layout scales with little additional effort or code, you must give some consideration to your top-level layout, content regions, app navigation, and commands. This ensures that they are placed predictably and intuitively on larger screens.

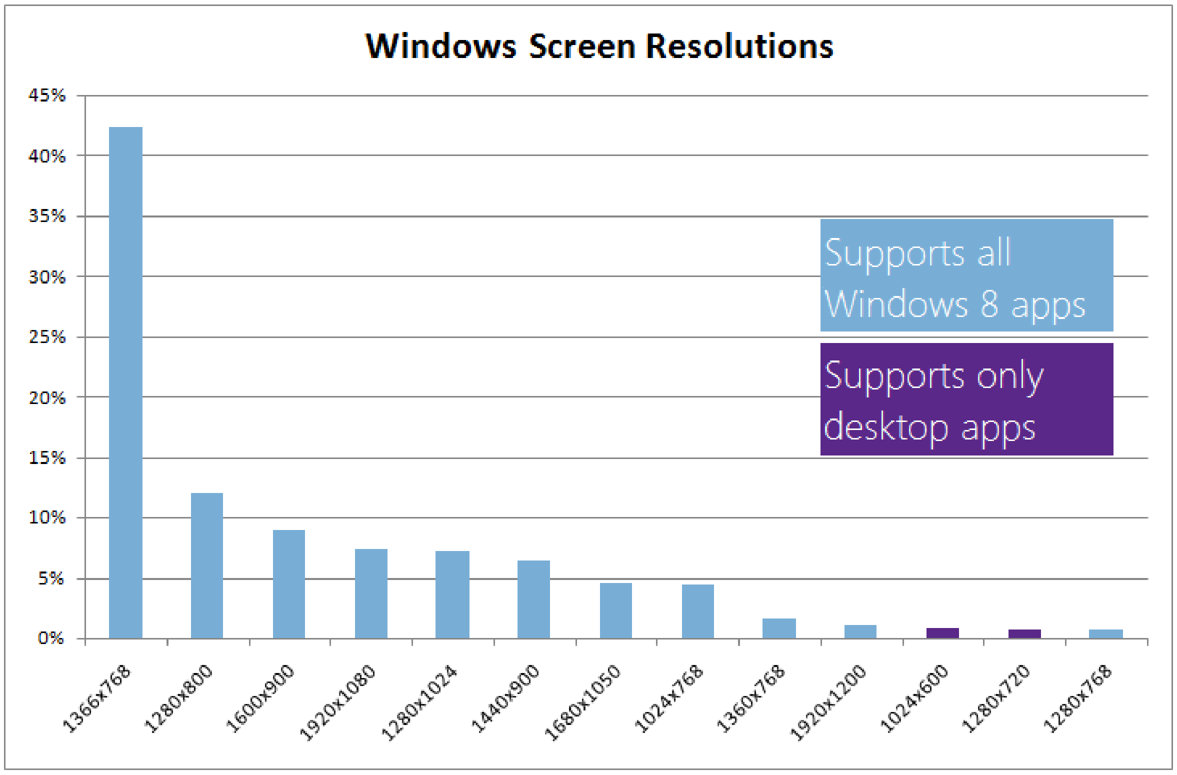
The following image demonstrates the large empty regions caused by not building an adaptable layout for large screens.

Minimum screen resolutions

There are two primary screen resolutions that your app should support:

- The minimum resolution for Windows Store apps is 1024x768.
- The minimum resolution required to support all the features of Windows 8 (including multitasking with snap) is 1366x768.

The following image presents all screen resolutions that can support Windows 8 applications. The vertical column represents the percentage of Windows 8 apps that are supported at the resolution.



The following table presents the practices recommended for the primary screen resolutions.

Design for:	To ensure that:
A minimum resolution of 1024x768.	All of your UI (such as, navigation, controls, and content) fits on the screen without clipping.
An optimal resolution of 1366x768	All of your UI (such as navigation, controls, and content) fits on the screen without blank regions.

Designing for larger screens

When designing an app for larger screens, consider how your app’s layout, aesthetic, proportions, and controls can be applied to a larger canvas. Additionally, any app can have any number of layouts of varying complexity. Each layout can be considered individually for larger screens.

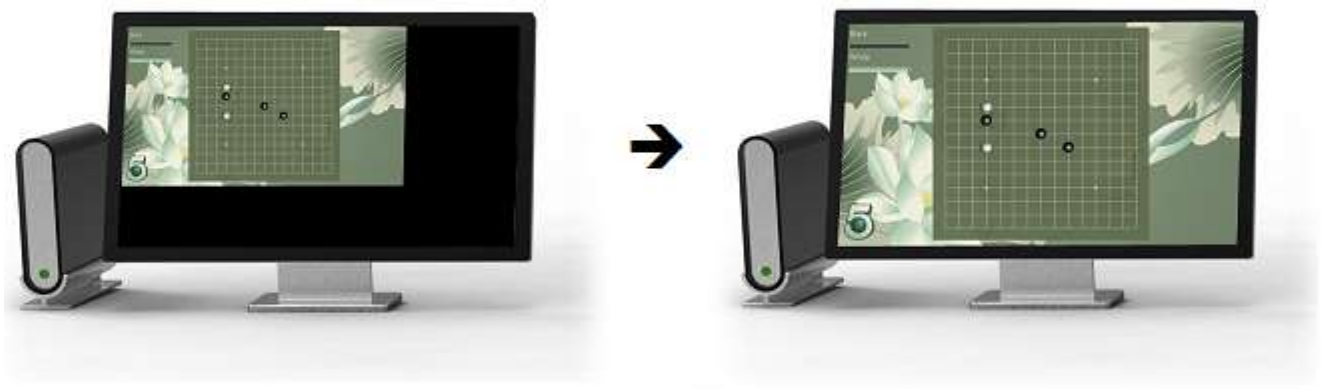
Fill the screen with content	This provides a user experience that is as immersive as possible regardless of the screen size. Apps should appear to fill the screen to the best of their ability and should appear to be thoughtfully designed for varying screen sizes. Users who buy larger monitors expect that their apps continue to look great on those large screens and fill the screen with more content, where possible.
Determine whether your layout should be fixed or adaptive	There are two techniques that can be used to build an app that scales to different screen sizes. The choice depends on the complexity of the layout and the usage scenarios.

Fixed layout

A fixed layout is most often seen in games or game-like apps that are composed primarily of bitmap images. These types of layouts have a fixed amount of content (for example, a game board) where showing more content is not possible or may not add any value. These layouts cannot, or will not, dynamically change or adapt to different screen sizes because they are designed with fixed pixel values. Windows 8 accommodates these apps with a "scale to fit" approach that is built into the platform.

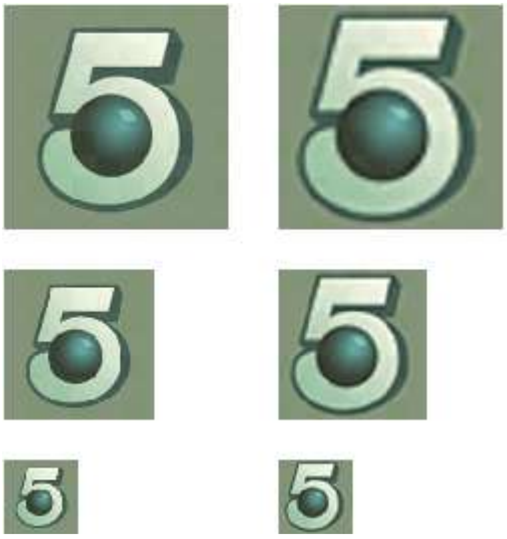
Fixed layout: Scale to Fit

If you determine that your app requires a fixed layout that can't adapt to different screen sizes, you can use a scale-to-fit approach. This makes your fixed layout fill the screen on different screen sizes, as shown in the following image.



The following table presents the practices recommended for apps that use scale-to-fit functionality.

You should:	**Description**
Start with the base resolutions.	When designing a fixed layout, start by designing your layout for the baseline resolutions: 1024x768 and 1366x768. This is the layout that Windows 8 scales to the larger screens.
Place your fixed content within a **ViewBox** control.	The **ViewBox** control scales a fixed layout to fit the screen. • Ensure that the **ViewBox** control is sized to 100% width and height. • Define the fixed size properties of the **ViewBox** to the fixed pixel sizes of your layout (for example, 1366x768).
Avoid putting adaptive controls (such as an app bar) in the **ViewBox**.	These controls automatically adapt to different screen sizes.
Define letterboxing style and color.	Fixed app layouts do not dynamically change in response to aspect ratio or screen size changes. So, the scale-to-fit technique automatically centers and letterboxes (horizontally, or vertically) your app's content. Defining a letterboxing style and color that complements the app or hardware color can provide a great experience. Letterboxing color is defined by your top-level app layout's background color. We recommend choosing a dark color like black that blends in with the hardware, a neutral color like grey that looks intentional, or a color that matches your app content color.
Provide vector or high-resolution assets.	The scale-to-fit technique scales your application to varying sizes up to 200% of the design size for your app on a large desktop monitor. Vector assets like Scalable Vector Graphics (SVG), XAMLExtensible Application Markup Language (XAML), or design primitives scale without scaling artifacts or blurriness. If raster assets (such as bitmap images) are required, provide images that are twice as large as the design size so they can be scaled down instead of scaled up.

	The following images demonstrate how scalar images (right) degrade when scaled up in size compared to vector images (left). 

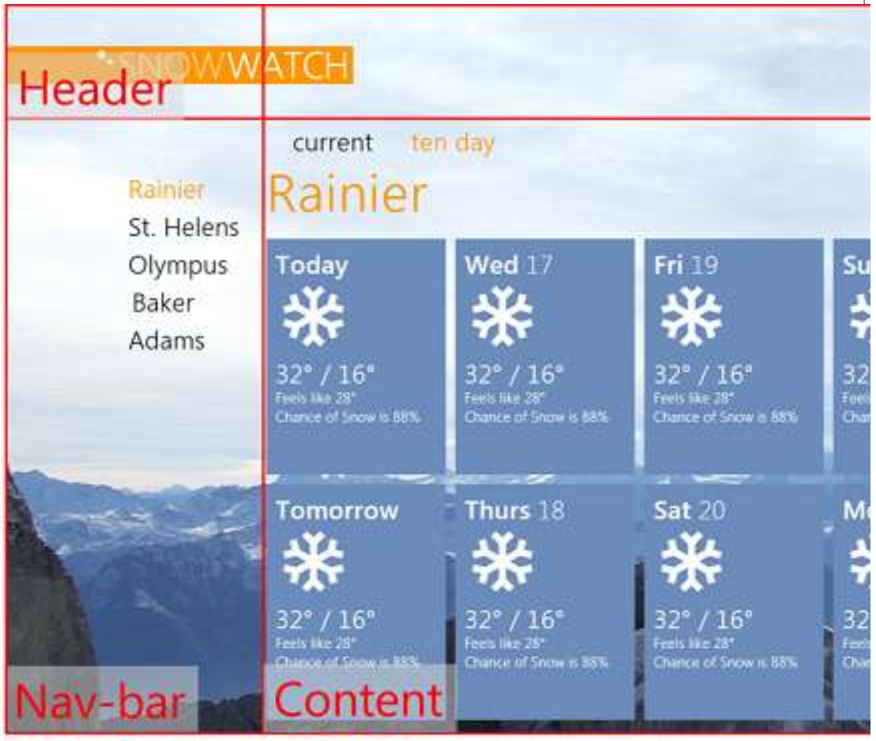
Adaptive layout

Adaptive layouts are most often seen in content consumption, management, and creation apps. These layouts are most often made up of defined proportional elements with a top-level wireframe. For example, a news reader app has a header, footer, and a content region in the center. These layouts dynamically change and adapt to different screen sizes and bring in more content and dynamically fill the space according to the rules of the layout. Windows 8 accommodates these apps with adaptive layout techniques discussed here in more depth.

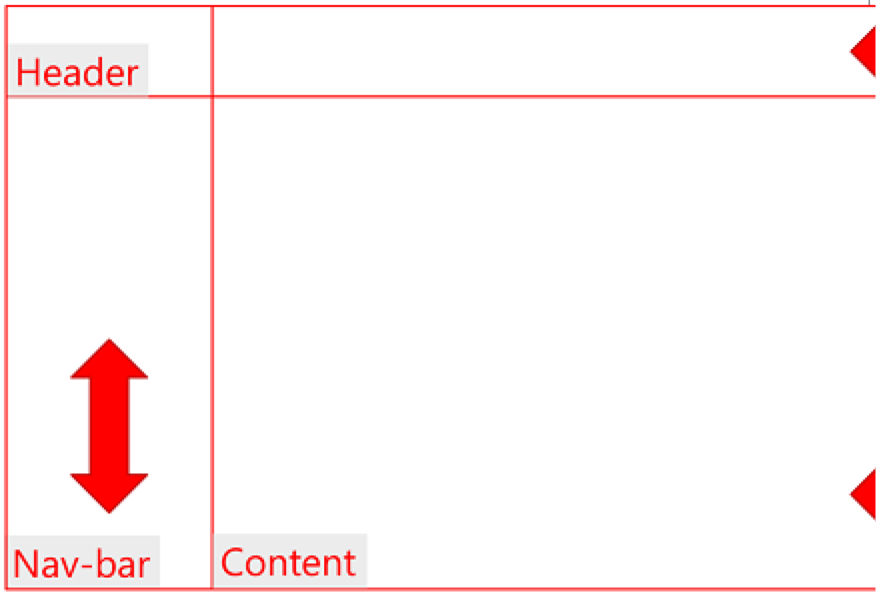
Adaptive layout: manage the space

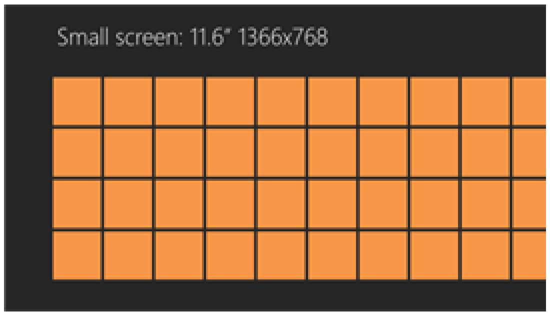
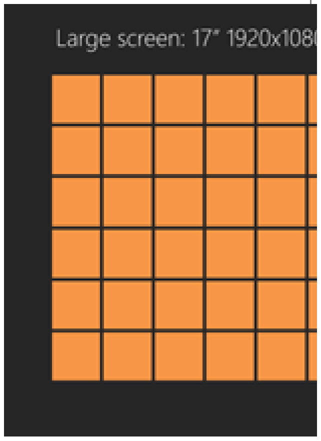
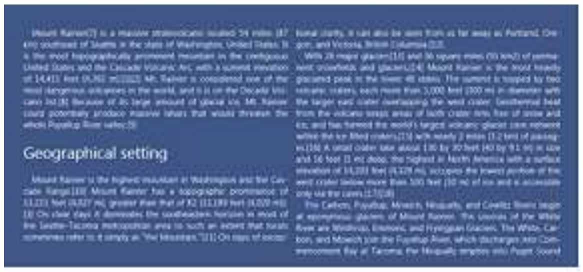
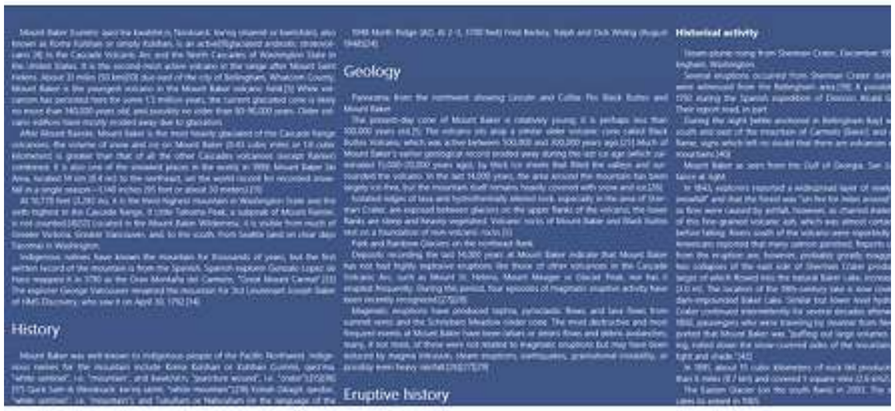
If you determine that your app layout can support adaptive layout to accommodate different screen sizes, use the following considerations to determine how your app can use all available screen space.

The following table presents the practices recommended for apps that use adaptive layout.

Practice	Description
Determine how each adaptive region can use the available space.	For each cell that you've identified as being adaptive in the horizontal or vertical direction, determine how your app layout uses that space on a larger screen.
Determine the top-level layout wireframe.	This wireframe should describe what the top-level regions of your app are. This wireframe should include where your header, navigation, and content regions are. The following image demonstrates a top-level wireframe.  SNOWWATCH Header current ten day Rainier Rainier St. Helens Olympus Baker Adams Today Wed 17 Fri 19 Tomorrow Thurs 18 Sat 20 32° / 16° Feels like 28° Chance of Snow is 88% Nav-bar Content

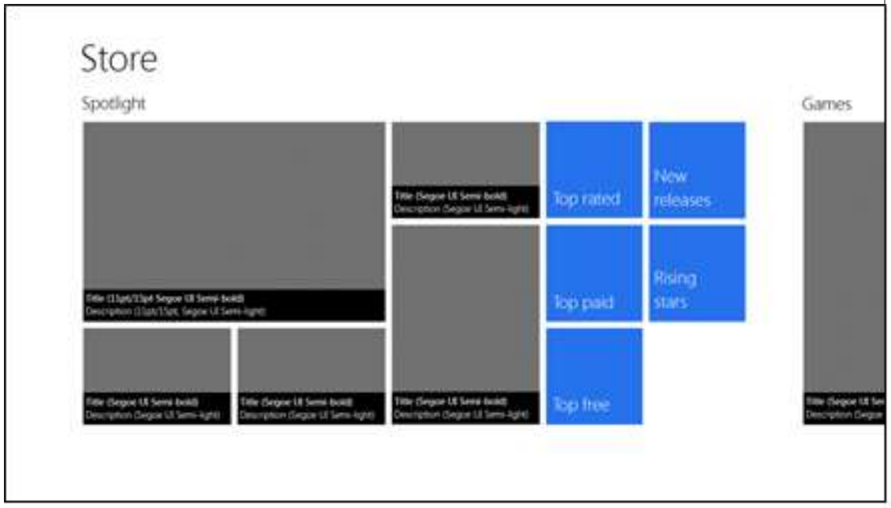
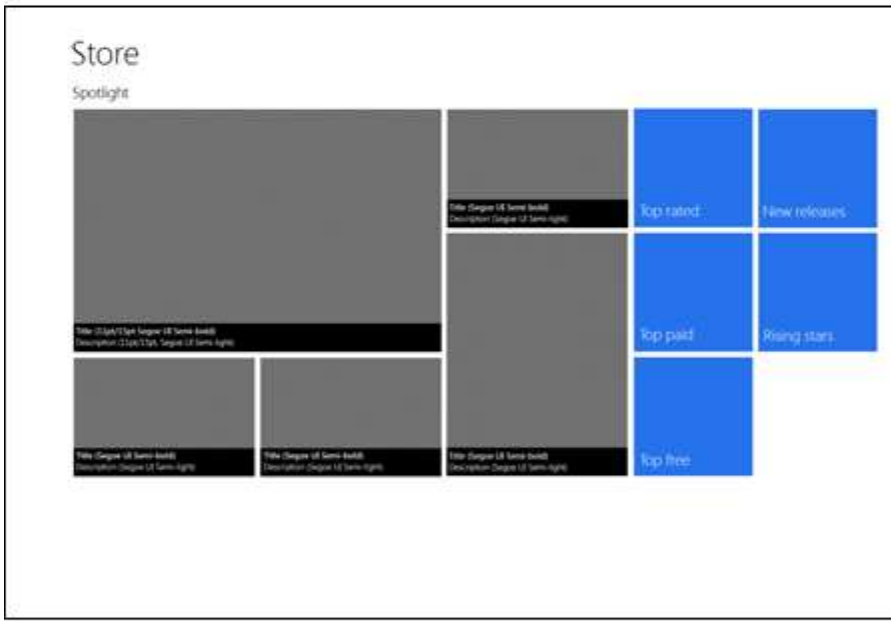
View state design

Determine which parts of the layout are fixed vs. adaptive.	For each cell in your top-level wireframe, determine how each cell should size itself for both the vertical and horizontal dimension when your app is shown at different sizes. This top-level wireframe description and the sizing behavior can be used to define the parameters for a **GridLayout**. 
Determine in which dimensions each adaptive region will adapt.	For each cell that you've identified as being adaptive in the horizontal or vertical direction, determine how your app layout uses that space on larger screens.
Determine how your app uses space in adaptive dimensions.	After you've determined how each region of your app with adapts to different sizes, the next step is to consider how your app uses the space. There are many techniques that your app can use and combine to use the space. The Windows 8 platform and controls support all of them.

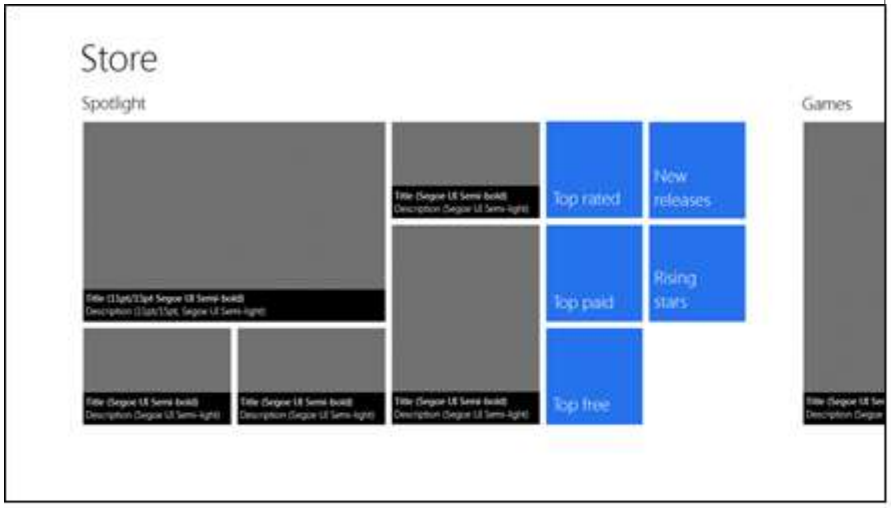
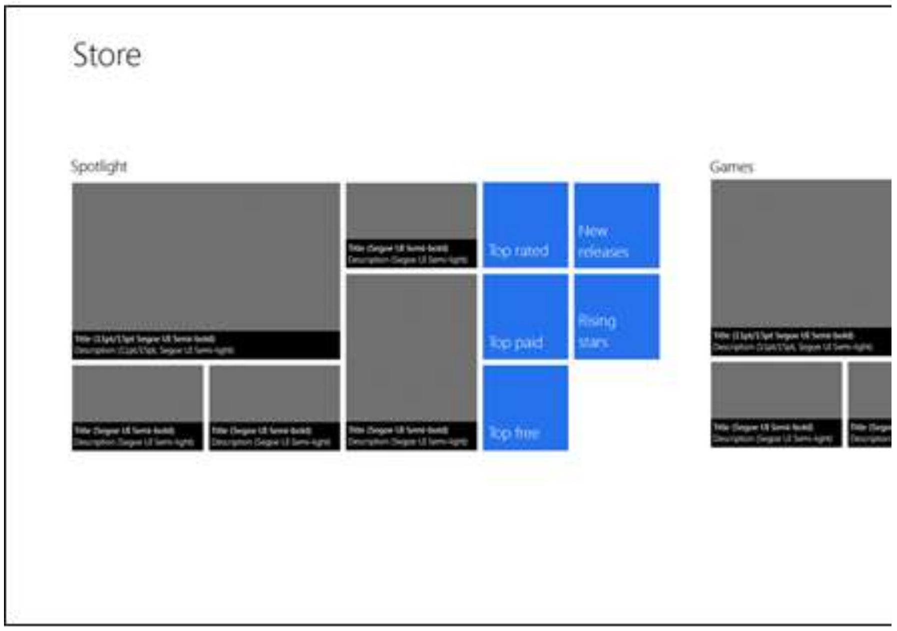
Ensure the width and height of all collection views are sized to 100%.	A **ListView** control automatically fills available space with more items.  
Use Multi-column Layout for text, where appropriate.	Multi-column Layout automatically adds columns for readability and flows content to fill available space.   Geology History Eruptive history Historical activity

Use a **canvas** for images, where appropriate.	A **canvas** automatically expands to fill available space.

View state design

Show more whitespace.	 Store Spotlight Games Title (11pt/15pt Segoe UI Semi-bold) Description (11pt/15pt, Segoe UI Semi-light) Title (Segoe UI Semi-bold) Description (Segoe UI Semi-light) Top rated New releases Top paid Rising stars Top free  Store Spotlight Title (11pt/15pt Segoe UI Semi-bold) Description (11pt/15pt, Segoe UI Semi-light) Title (Segoe UI Semi-bold) Description (Segoe UI Semi-light) Top rated New releases Top paid Rising stars Top free

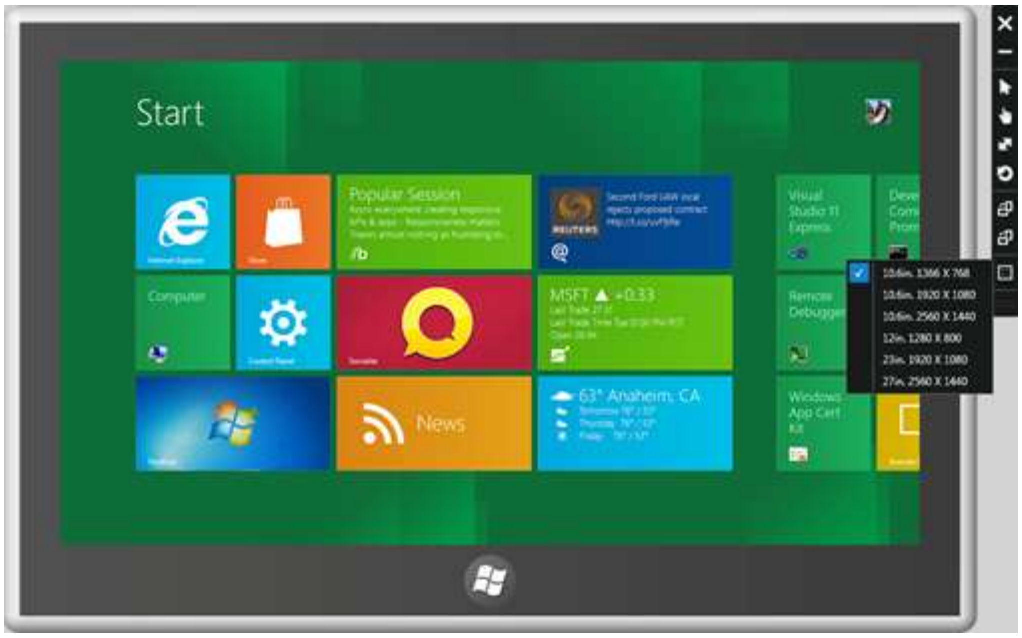
View state design

Show more app.	 

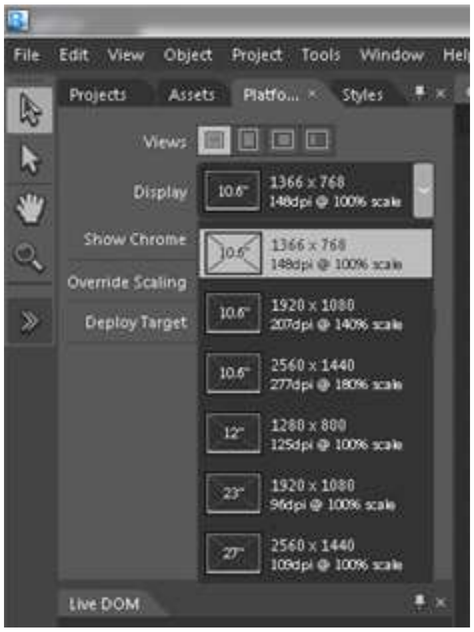
Testing your app layout

It’s important to test apps on different screen sizes. We realize that most people don’t have many devices at their disposal, so we built support for testing apps on different screens into tools such as Visual Studio 2012. The Windows Simulator lets you run your apps on a variety of screen sizes, orientations, and pixel densities, as shown in the following image.

View state design



You can use the platform menu in Blend for Microsoft Visual Studio 2012, shown in the next image, to design your app on different screen sizes and pixel densities. The Blend canvas updates dynamically based upon the screen option chosen from the platform menu.


File Edit View Object Project Tools Window Hel
Projects
Assets
Platfo...
Styles
Views
Display
Show Chrome
Override Scaling
Deploy Target
10.6" 1366 x 768 148dpi @ 100% scale
10.6" 1366 x 768 148dpi @ 100% scale
10.6" 1920 x 1080 207dpi @ 140% scale
10.6" 2560 x 1440 277dpi @ 180% scale
12" 1280 x 800 125dpi @ 100% scale
23" 1920 x 1080 96dpi @ 100% scale
27" 2560 x 1440 109dpi @ 100% scale
Live DOM

Guidelines for snapped and fill views

View state design

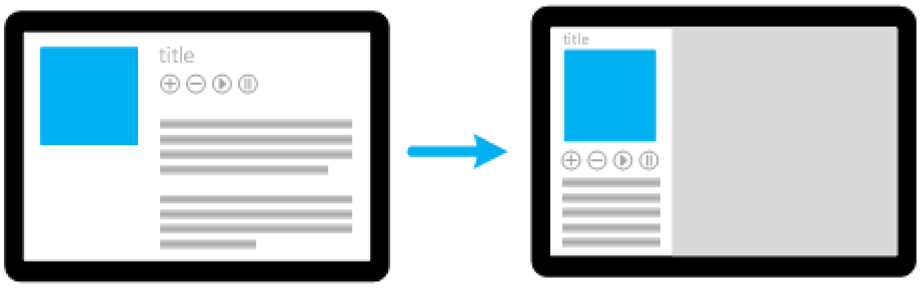
Because people can snap every app, design your app for the snapped view state. If you don't, the system resizes your app anyway and might crop your content or add scrollbars.

Note Snapped and fill views are only available on displays with a horizontal resolution of 1366 relative pixels or greater.

Here are some general principles you should follow:

- **Maintain state and preserve continuity.** Maintain the state in the snapped view, even if it means showing less content or reducing functionality. Prioritize maintaining state over showing the user a more attractive, but otherwise out-of-context, snapped page.
- **Have feature parity across states.** Remember that snapping is simply resizing your app. The user 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.
- **Use media queries for your layout logic.** This allows your application to respond appropriately when your application layout changes.
- **Use application view API.** This ensures your app is notified of layout changes, especially during app launch and relaunch from a suspended state.
- **Put the user in control.** Don't programmatically unsnap your app to get the user's attention. Unsnapping should be used when the user tries to use a feature that is not available in the snapped state. If your app has snapped views for all pages in the app, you shouldn't need to programmatically unsnap at all.
- **Don't add UI controls to programmatically unsnap your app.** The splitter between the apps is always present and lets the user unsnap whenever they want to.

Remember, the snapped app is your app resized! It is not a gadget or a minimized window. You want to maintain state, context, and interactivity for your users. Snapping and unsnapping should never destroy the user's work or state.

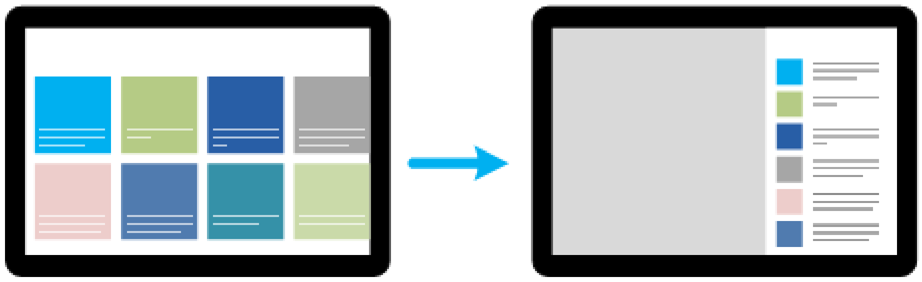


When should you use media queries and when should you use JavaScript layout change events?

Use:	**To drive:**
Media queries	Changes in layout and manipulation of properties specified through CSS styles: • Element size • Element layout (inline, block) • Visibility of various elements
JavaScript events	Changes in behavior and manipulation of properties that cannot be specified through CSS styles: • Scroll direction of the ListView control • Control changes, such as going from a list of buttons to a single drop-down select control

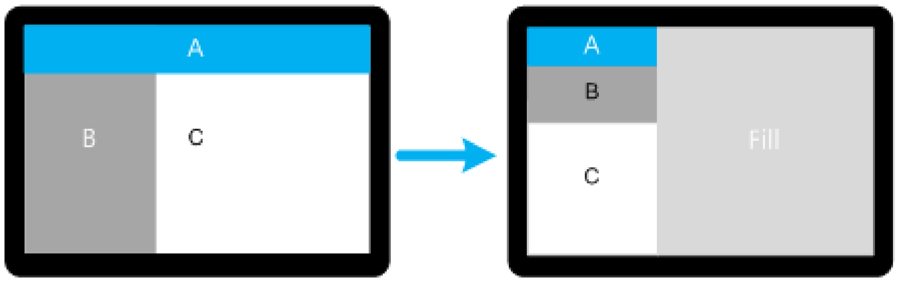
Snapped layouts

Panning direction Take advantage of the snapped layout's "tall and skinny" aspect ratio to pan vertically. If your app changes its panning direction between snapped and fullscreen or fill states, make it clear in the UI that the panning direction has changed.



Column layouts Given the narrow width of the snapped state, we recommended that you change multi-column layouts to a single column layout when the app is snapped. The following example has a multi-column layout in its unsnapped state. When the app is snapped, elements of the layout are stacked to create a single column layout.

View state design



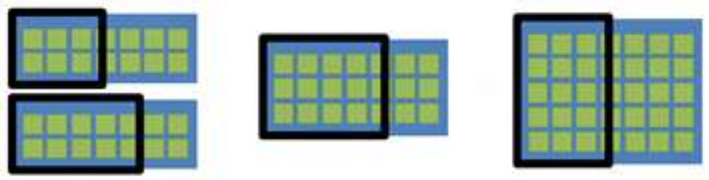
View State and Window Dimensions

Depending on the needs of your app, you can base your app's UI layout on view states, the app's window dimensions, a combination of view states and window dimensions, or neither.

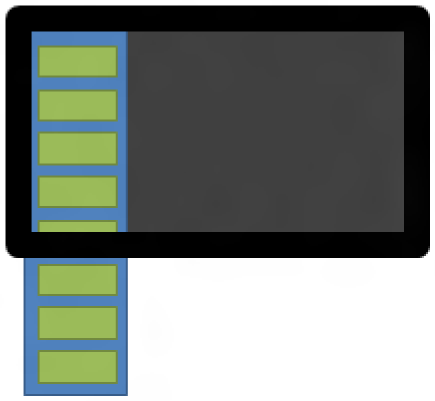
View states give you high-level information about the "shape" of the app and context around the app. Window dimensions give you finer control over the layout of UI elements, which is useful if specific widths and heights matter for how elements are laid out.

Whether you use view states or window dimensions depends on how your app "treats" its content, as described in the following sections:

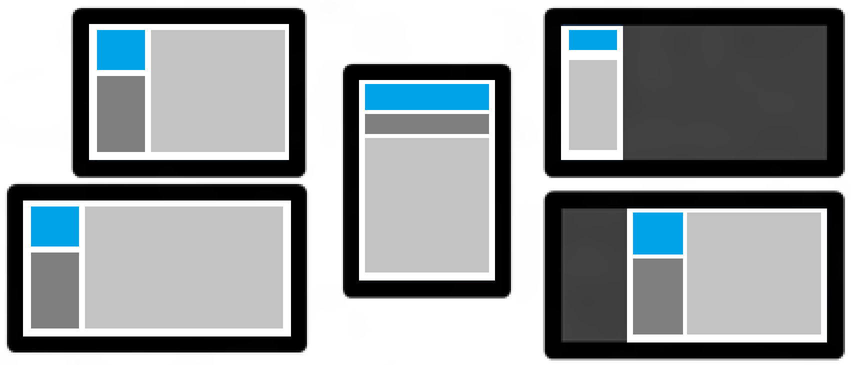
Completely fluid content The app content flows off the screen horizontally, adding more rows if there is sufficient height.



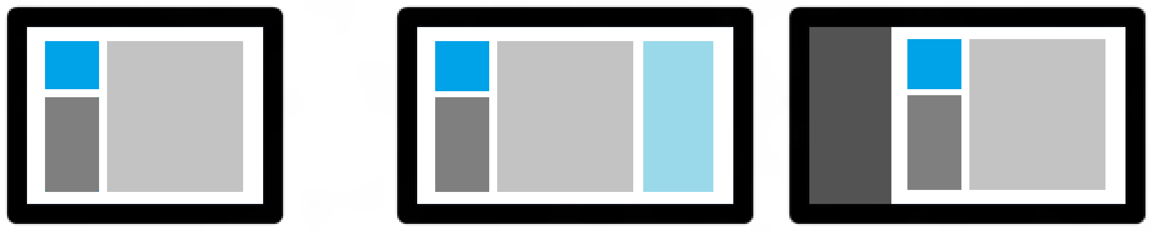
For fluid content, the width of the screen and the window dimensions don’t matter for layout purposes. However, you might need the view state to determine whether the app is snapped so you can convert content for vertical panning.



Shape dependent The app lays out UI elements based on the "shape" of the app, regardless of its exact dimensions. For example, the app might stack UI elements when the app is taller than it is wide. The view state gives sufficient information for this scenario.

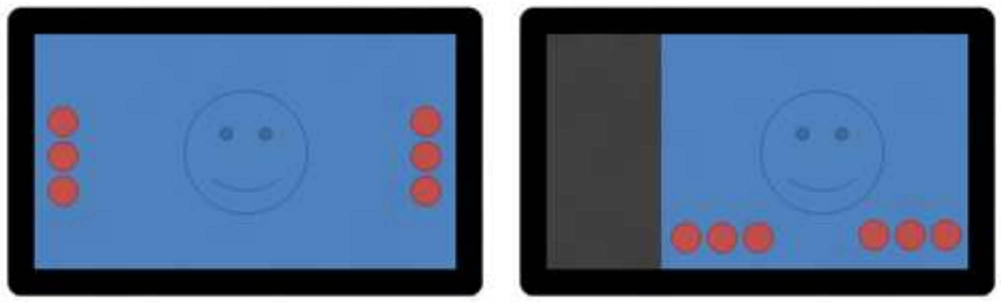


Exact dimensions matter The app has optional UI that appears only when there is sufficient space. For example, a social networking site might show a real-time news feed only when there is enough width.



When exact dimensions matter, the view state alone doesn't give enough information to determine whether to show the optional UI. There might be a filled state that is wide enough to show the optional UI, or a full screen state that is not wide enough. In cases like these where exact widths matter, use window dimensions to lay out your app.

Full screen matters The app needs to know whether it has all of the pixels on the device screen. For example, a game might place controls at the sides when it is full screen, and reposition them at the bottom when not full screen.



Window dimensions don't tell you whether your app is full screen. You'll need to get the filled view state to determine how to position controls.

Guidelines for resizing

These guidelines explain how to ensure that your app looks great when Windows needs to resize it. Windows automatically resizes your app when, for example, people call up the soft keyboard.

- **Use the default behavior.** To reduce the development effort on your part, let Windows handle the resize and reflow of content whenever possible.
- **Use custom handling.** If the default behavior is inadequate, you can implement custom handling. For example, let's say you have an email app and you want to ensure a good user experience when Windows resizes the app when users are typing a new message with the soft keyboard. You can customize the resize of the input field to take up the entire viewable area, allowing users to see the maximum amount of their message as they create it. Such behavior is not automatically performed by Windows.
- **Specify a minimum size for all input fields.** This ensures that the input fields do not disappear when the windows are resized.
- **Test your application's resize behavior thoroughly.** In particular, test that the soft keyboard doesn’t cover your app's input fields.
- Do NOT perform custom handling if the resulting layout is identical to what Windows would have provided.

Branding design

A brand defines the qualities that a business wants to be known for. When designing your Windows Store apps, make thoughtful decisions to ensure that your apps incorporate the essence of your brand.

Visual elements of your brand guideline

Every brand, whether it's new or well-known, conforms to a brand guideline. Think of the brand guideline as a toolbox of visual elements with rules for applying them. A distinctive color palette, graphics, layout, and photography style all work together to create a repeatable and recognizable brand. The branding will be recognizable in broadcast, print media, or a Windows Store app.

You use a number of visual elements to design your brand. Think of these visual elements as the knobs and dials that you manipulate through code to create a unique look and feel in your Windows Store app.

Colors	Color is one of the key attributes of any brand. Apply the primary color associated with your brand in ways that remind customers that this app comes from your business. • Be selective in how color is used throughout your app. A carefully chosen color palette makes a simple, clean, and brand-appropriate presentation of content. • Secondary colors should complement your brand's primary color. • To ensure legibility, choose colors that maintain a 4.5:1 contrast ratio, when applied to typography.
Icons	Windows Store apps leave behind the "icons represent everything everywhere" approach to UI design. • Icons in Windows Store apps are an important part of navigating a touch-first system. Think of icons as touch targets. • Icons aren't bullet points, embellishments, or detailed illustrations. • Icons are graphic in nature, flat in perspective, and monochromatic. This approach reinforces the principle of "content over chrome."
Images	As most brands have a distinct color palette, they also use a unique set or style

	of images. The images in your Windows Store apps must speak to your brand. • Images should communicate what products and services your business is about. • Images can be staged or modified to capture the emotions, feelings, and memories associated with your brand. • Images can range from hand-drawn art to studio-quality photography.
Grid	The grid system is a design tool that helps achieve visual unity across different apps and features while providing a structure for variation and maintaining user interest. • Adhere to the grid system. This helps align the UX of your apps with the rest of Windows. • The grid is the underlying, and somewhat invisible, blueprint for how the visual elements of your Windows Store app will all come together.
Layout	A layout is the arrangement of visual elements on an app page. • Your brand needs to shine through in both portrait and landscape orientations and in all three view states: full screen, snapped, and fill.
Logo	Use a logo as a means to facilitate quick identification and public recognition. • Your logo helps customers locate your app in the Windows Store and on the Start screen. • Your logo is an anchor point throughout your app's user experience.
Typography	Well-crafted typefaces are a key part of Windows Store apps. • If typography is not an important part of your visual brand, then benefit from the capabilities, grace, and clarity of Segoe, or the alternate typeface, Cambria. • If typography is important, you can also use your own distinctive typefaces.

Examples that evoke unique brands

Here are some examples that show how each visual element helps to evoke different emotions and feelings. Descriptions for each visual element provide specific details about how the layout was created, and why it's important for leveraging the new Windows 8 UI style.

Contoso French Bakery brand

The Contoso French Bakery brand is known as the destination people seek out to satisfy their sweet tooth and guilty pleasures. Since there are only a few Contoso French Bakeries in the area, every visit is memorable, and every customer is more than happy to place the diet on hold. This brand is not about Mom’s home-baked goods or sugary treats for the kids. The Contoso French Bakery brand is indulgent, heavenly, and perhaps even a bit devilish.

	Colors A palette of just two colors - brown and pink - is enough to make customers think of a bakery. Both colors are used in a purposeful way. Brown is the primary color, rich but neutral enough to let full-color photography to stand out, and pink is the accent color that pulls the eye in without being too contrasty or distracting.
	Icons Small, pink flag icons call attention to certain content. Icons aren't used for layout or commanding.
	Images Photographs of bakery items are the focus of this design. Images are used pragmatically to categorize the content and subtly reinforce the decadent side of the brand.
	Grid The standard grid is used throughout this design. For example, the logo is top-left aligned and margins are maintained. The design suggests a specific brand without compromising the new Windows 8 UI silhouette.
	Layout Compared to some Windows Store app designs, this example has a different feel. The most unique difference is the absence of square tiles. The tiles and grid still exist, but the images have been stylized. They mimic the items that are likely to be found in a bakery, for example, the circular and scalloped shapes of a cupcake tin.
	Logo The logo serves two purposes. First, it identifies the business and labels the app. Second, it serves as an anchor point by denoting the top and left margins of the app. Simple alignment details like this help to frame your app's content.

Typography Cambria, a serif font, feels more personal and less technical than a sans serif font. Cambria is the only typeface used throughout the app, which reinforces an open, clean layout. Also, the typeface is applied with restraint, which results in a look and feel that avoids being over stylized or busy.

Contoso Sandwich Truck brand

The Contoso Sandwich Truck brand is known for being an inviting place to meet with friends and family. The hand-crafted entrees and specialty drinks make Contoso Sandwich Truck more than a great place for regulars. It's a popular destination for out-of-town guests. This brand is not about being a tourist hot spot. It's one of the city's original food trucks. This brand is authentic.

Colors Warm dark gray, off-white, and copper colors give this design a rich feel. The warm palette evokes some of the same colors you'd find in a homemade bread crust, soup, or pastry.

Icons No icons are used in the design, because they're not needed. If icons were added, they'd appear as nothing more than ornamentation. Instead, typography does the heavy-lifting when it comes to presenting content in a clear way.

Images No images are used in the design. Images could be used to accompany the text, but this layout relies entirely on typography to communicate the products offered and the brand itself.

Grid If not for the grid, the typography would feel as if it were floating haphazardly on the page. In this example, content intentionally "breaks" the grid. The top margin is reduced to provide more room for menu content. The left margin has been adjusted so the company name, intro text, and location information feel more centered, giving the content room to breathe.

Layout The design is intended to feel like a menu. Like the sandwiches and other menu items, the layout is intended to feel more handcrafted than the other examples that appear here.

Branding design

Logo A company logo isn't used in this example. The company name appears where the logo would, in text. The company name in text is the logotype. A logotype is a mark made of words, not a graphic or symbol, that's used to identify a company or event.

Typography Copperplate Gothic Bold is the primary typeface used in this design. It gives the company name its look and feel. Script MT Bold and Segoe UI are the accompanying typefaces. The script font is used sparingly in the menu header. Segoe UI is used as the body text.

Alternate Contoso Sandwich Truck brand

This is an alternative to the design of the Contoso Sandwich Truck brand. It's a brand known for cooking up great street food and being available almost everywhere. This brand is backed-up by a crew of excellent chefs and a fleet of trucks. They are all geared toward people on the go, like people downtown during the weekdays, and crowds at weekend festivals. This brand is not about fast food or fine dining. The Contoso Food Trucks brand is urban, social, and location-aware.

Colors Full-color photography adds an array of colors to the app. To avoid competition between the color palette and the photography, black and yellow are used as accent colors. From a personality perspective, the black and yellow are also colors found in urban settings, such as street signs, and road markings.

Icons Icons appear only in star shapes, which indicate a customer rating system. Menu choices remain the focus and aren't cluttered by an excessive use of icons.

Images For this business to get customers to visit them, enticing and persuasive images of the product are critical. All of the food photography is professionally staged. This makes it possible to showcase close-up images in the app.

Grid The standard grid anchors all aspects of this design. Even though a company logo isn't used, the company name appears near the top-left and is aligned with the left margin.

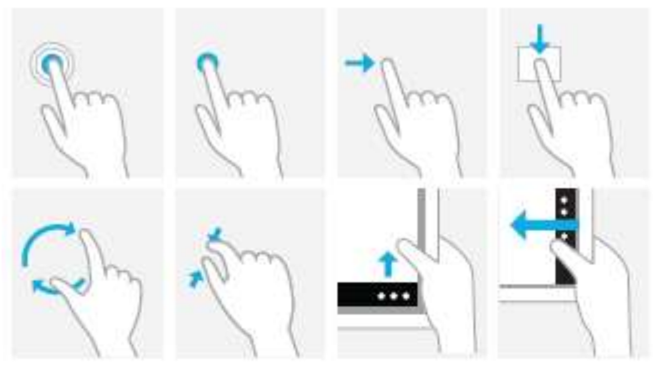
Layout The organization of content has the same structure that you'd see in an overhead view of a city street map. To challenge this structure, full-bleed and overlapping content

	sets this example apart. The overall presentation of content is bold and direct, based on the need to appeal to people who are on the go and need to make quick decisions.
	Logo A company logo is not used in this example. The company name appears where the logo would, and in text. One could argue the company name in text is the logotype. A logotype is a mark made of words, and not a graphic or symbol, used to identify a company or event.
	Typography Trebuchet is a sans serif font, and it feels clean and technical. It's not difficult to imagine this typeface appearing on municipal properties and signage. Only three point sizes are used, and only two colors, which minimizes the visual noise. Full-color photography is the focus, followed by the text box shapes, which add personality.

TOUCH, COMMANDING, AND CONTROLS

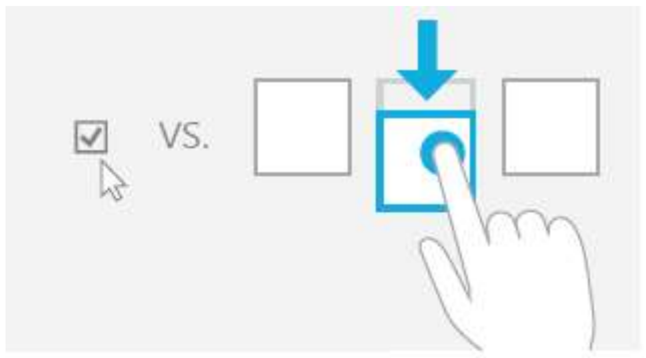
Touch interaction design

1. Use the Windows 8 touch language.



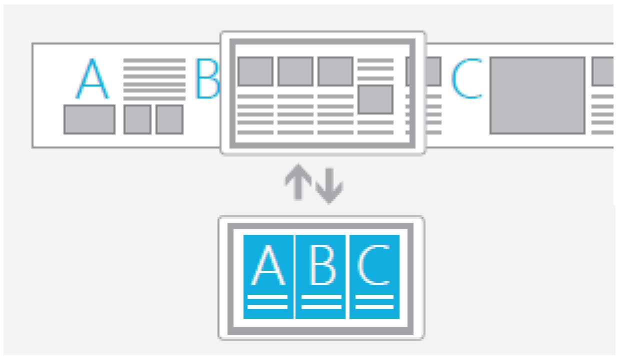
Windows 8 provides a concise set of touch interactions used consistently throughout the system. Applying this language consistently makes your app feel familiar to what users already know. This increases user confidence by making your app easier to learn and use.

2. Use fingers for what they're good at.



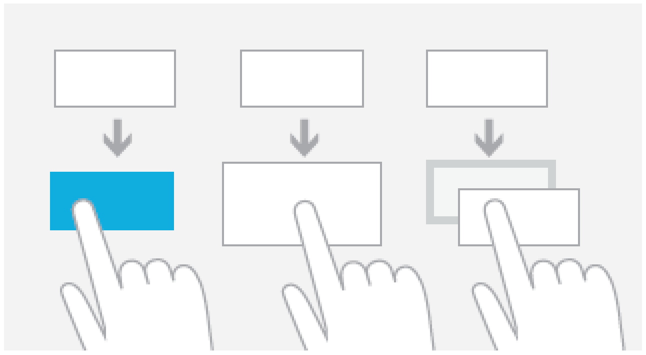
A mouse and pen are precise, while fingers aren't, and small targets require precision. Use large targets that support direct manipulation and provide rich touch interaction data. Swiping down on a large item is quick and easy because the entire item is a target for selection.

3. Browse content with touch.



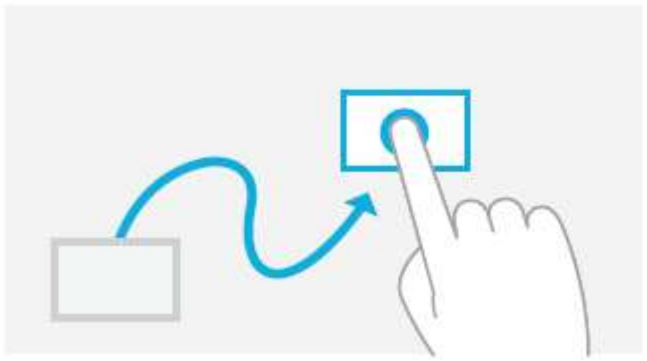
Semantic Zoom and panning make navigation fast and fluid. Instead of putting content in multiple tabs or pages, use large canvases that support panning and Semantic Zoom.

4. Provide feedback.



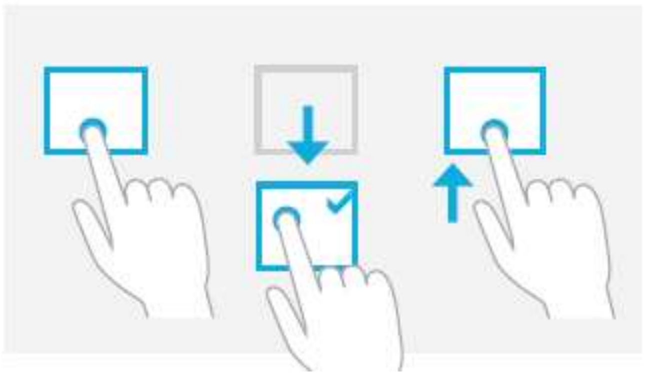
Increase user confidence by providing immediate visual feedback whenever the screen is touched. Interactive elements should react by changing color, changing size, or by moving. Items that are not interactive should show system touch visuals only when the screen is touched.

5. Content follows finger.



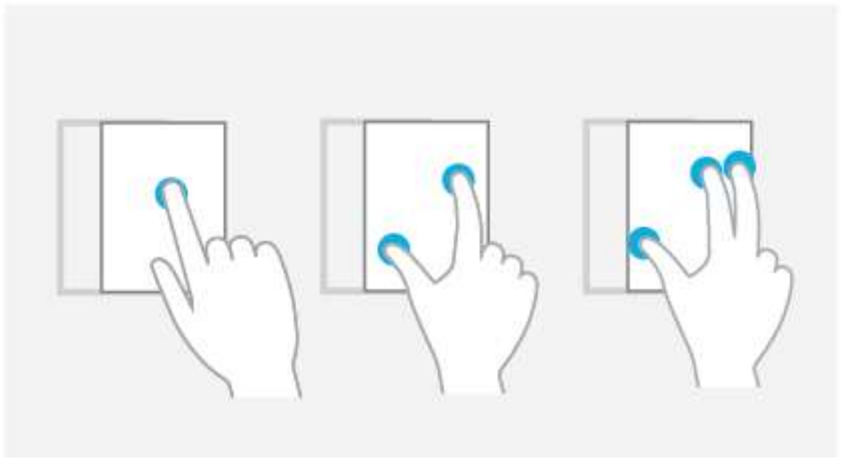
Elements that a user can move or drag, such as a canvas or a slider, should follow the user's finger when moving. Buttons and other elements that do not move should return to their default state when the user slides or lifts their finger off the element.

6. Keep interactions reversible.



If you pick up a book, you can put it back down where you found it. Touch interactions should behave in a similar way—they should be reversible. Provide visual feedback to indicate what will happen when the user lifts their finger. This will make your app safe to explore using touch.

7. Allow any number of fingers.



People often touch with more than one finger and don’t even realize it. That's why touch interactions shouldn’t change radically based on the number of fingers that are touching the screen. Just like the real world, sliding something with one or three fingers shouldn't make a difference.

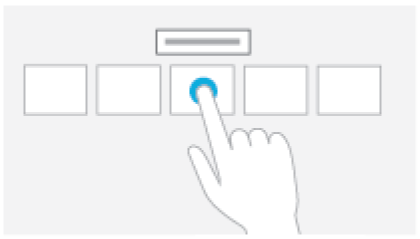
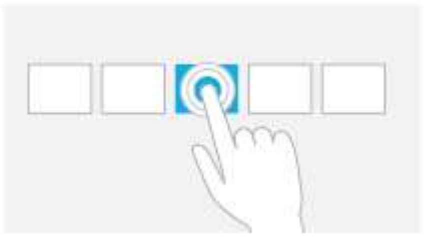
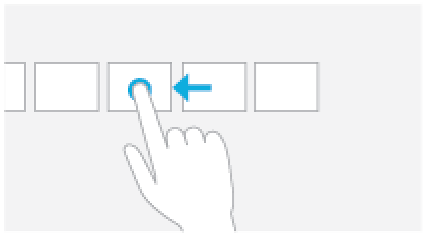
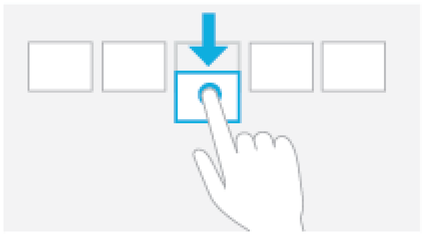
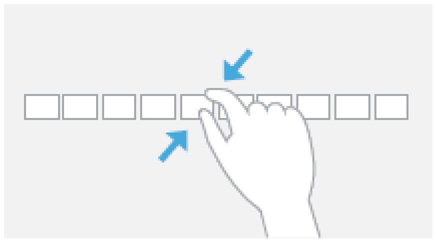
8. Keep interactions untimed.

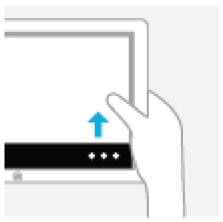
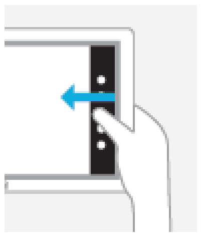
Interactions that require compound gestures, such as double tap or press and hold, need to be performed within a certain amount of time. Avoid timed interactions like these because they are often triggered accidentally and are difficult to time correctly.

Windows 8 touch language

This list describes the standard touch-related terms used in Windows 8.

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

	Press and hold to learn	This touch interaction causes detailed information or teaching visuals (for example, a tooltip or context menu) to be displayed without a commitment to an action. Anything displayed this way should not prevent users from panning if they begin sliding their finger.
	Tap for primary action	Tapping on an element invokes its primary action, for instance launching an app or executing a command.
	Slide to pan	Slide is used primarily for panning interactions but can also be used for moving, drawing, or writing. Slide can also be used to target small, densely packed elements by scrubbing (sliding the finger over related objects such as radio buttons).
	Swipe to select, command, and move	Sliding the finger a short distance, perpendicular to the panning direction, selects objects in a list or grid (**ListView** and **GridLayout** controls). Display the **app bar** with relevant commands when objects are selected.
	Pinch and stretch to zoom	While the pinch and stretch gestures are commonly used for resizing, they also enable jumping to the beginning, end, or anywhere within the content with Semantic Zoom. A **SemanticZoom** control provides a zoomed out view for showing groups of items and quick ways

		to dive back into them.
	Turn to rotate	Rotating with two or more fingers causes an object to rotate. Rotate the device itself to rotate the entire screen.
	Swipe from edge for app commands	App commands are revealed by swiping from the bottom or top edge of the screen. Use the **app bar** to display app commands.
	Swipe from edge for system commands	Swiping from the right edge of the screen reveals the charms that expose system commands. Swiping from the left edge cycles through currently running apps. Sliding from the top edge toward the bottom edge of the screen closes the current app. Sliding from the top edge down and to the left or right edge snaps the current app to that side of the screen.

Note Users can perform direct manipulations like the slide-to-pan, pinch-to-zoom, and turn-to-rotate interactions simultaneously and with any number of touch points.

Windows 8 Touch posture

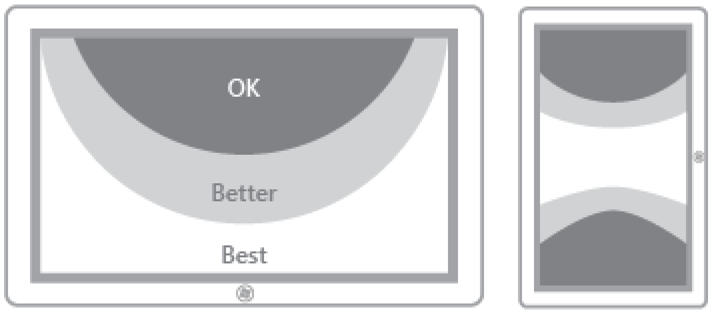
Designing for touch is more than designing what's displayed on the screen. It also requires designing for how the device will be held (grip).

Typically, different people have a few favorite grips when holding a tablet.

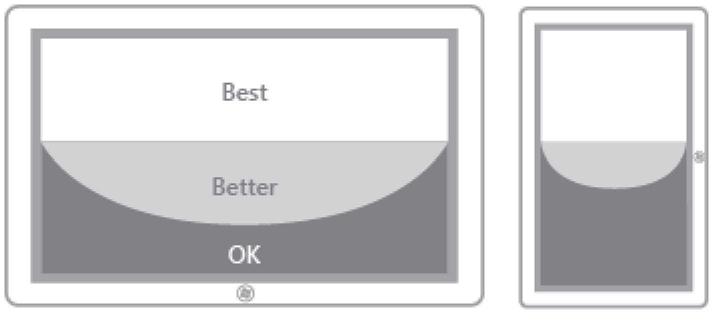
The current task and how it's presented usually determines which grip is used. However, the immediate environment and physical comfort also affect how long a grip is used and how often it's changed.

Try optimizing your app for different kinds of grips. But if an interaction naturally lends itself to a specific grip, optimize for that.

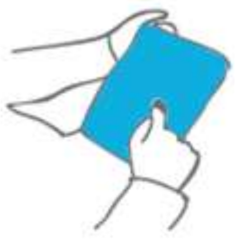
Interaction areas: Because slates are most often held along the side, the bottom corners and sides are ideal locations for interactive elements.



Reading areas: Content in the top half of the screen is easier to see than content in the bottom half, which is often blocked by the hands or ignored.



Four most common grips: While there are many ways to hold a tablet, these four grips are most commonly used.

Grip	Grip and interaction	Design considerations
	One hand holding, one hand interacting with light to medium interaction	• Right or bottom edges offer quick interaction. • Lower right corner might be occluded by hand and wrist. • Limited reaching makes

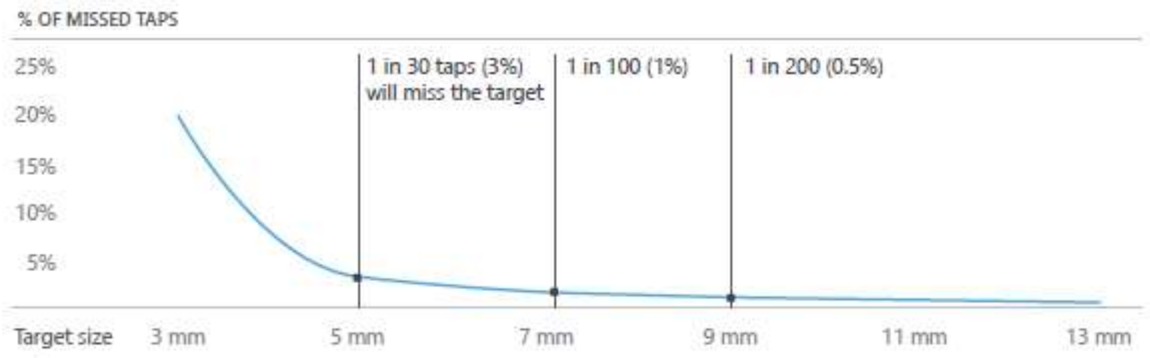
		touching more accurate. • Reading, browsing, email, and light typing.
	Two hands holding, thumbs interacting with light to medium interaction	• Lower left and right corners offer quick interaction. • Anchored thumbs increase touching accuracy. • Anything in the middle of the screen is difficult to reach. • Touching middle of screen requires changing posture. • Reading, browsing, light typing, gaming.
	Device rests on table or legs, two hands interacting with light to heavy interaction	• Bottom of the screen offers quick interaction. • Lower corners might be covered by hands and wrists. • Reduced need for reaching makes touching more accurate. • Reading, browsing, email, heavy typing.
	Device rests on table or stand, with or without interaction	• Bottom of screen offers quick interaction. • Touching top of the screen occludes content. • Touching top of screen might knock a docked device off balance. • Interaction at a distance

		reduces readability and accuracy. • Increase target size to improve readability and precision. • Watching a movie, listening to music.

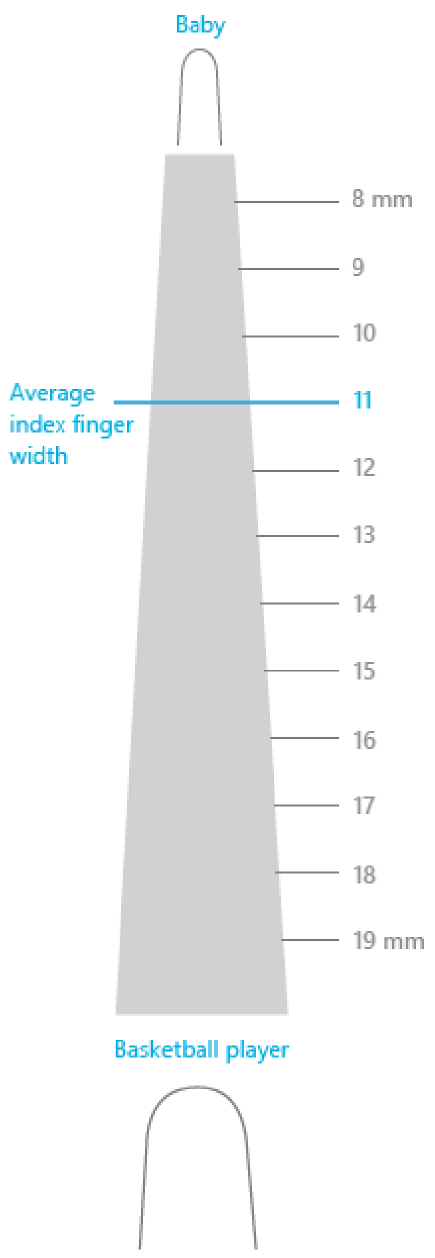
Windows 8 Touch targets

Size vs. efficiency: Target size influences error rate

There's no perfect size for touch targets. Different sizes work for different situations. Actions with severe consequences (such as delete and close) or frequently used actions should use large touch targets. Infrequently used actions with minor consequences can use small targets.



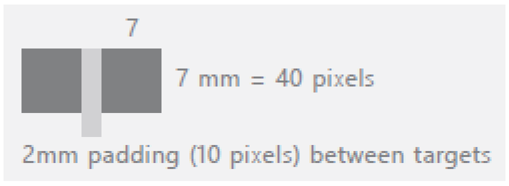
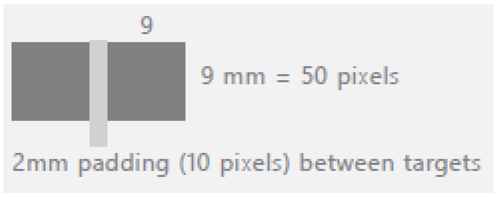
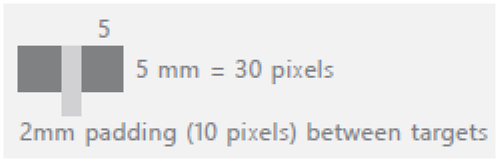
Fat fingers?



People often blame themselves for having "fat fingers." But even baby fingers are wider than most touch targets.

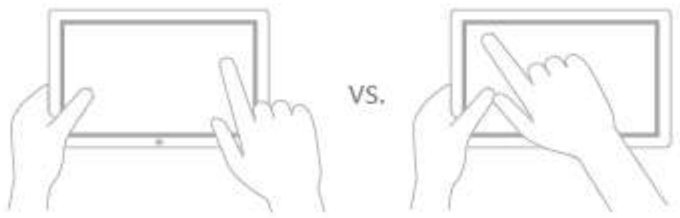
The image on the left shows that the width of the average adult finger is about 11 millimeters (mm) wide, while a baby's is 8 mm, and some basketball players have fingers wider than 19 mm!

Target size guidelines: Here are some guidelines for deciding how large or small to make your touch targets.

 7 7 mm = 40 pixels 2mm padding (10 pixels) between targets	7x7 mm: Recommended minimum size 7x7 mm is a good minimum size if touching the wrong target can be corrected in one or two gestures or within five seconds. Padding between targets is just as important as target size.
 9 9 mm = 50 pixels 2mm padding (10 pixels) between targets	When accuracy matters Close, delete, and other actions with severe consequences can't afford accidental taps. Use 9x9 mm targets if touching the wrong target requires more than two gestures, five seconds, or a major context change to correct.
 5 5 mm = 30 pixels 2mm padding (10 pixels) between targets	When it just doesn't fit If you find yourself cramming things to fit, it's okay to use 5x5 mm targets as long as touching the wrong target can be corrected with one gesture. Using 2 mm of padding between targets is extremely important in this case.

Most people are right handed

Most people hold a slate with their left hand and touch it with their right. In general, elements placed on the right side are easier to touch, and putting them on the right prevents occlusion of the main area of the screen.



Guidelines for touch input

How you design your user interface can influence how easy your app is to use with touch input. To ensure that your app is touch optimized, follow these guidelines:

- Be sure to accommodate the size difference between mouse pointers and fingertips. Touch requires larger UI elements to ensure accuracy and to prevent fingers from obscuring important information.
- Always provide immediate, direct visual feedback for touch interactions. For example, you can use highlighting or tool tips to indicate the current touch target and prevent the accidental activation of other targets.
- Use physics effects such as acceleration and inertia to provide a natural feel in interactions such as panning.
- Use snap points and other constraints to help guide people to the most useful states.

Guidelines for cross-slide

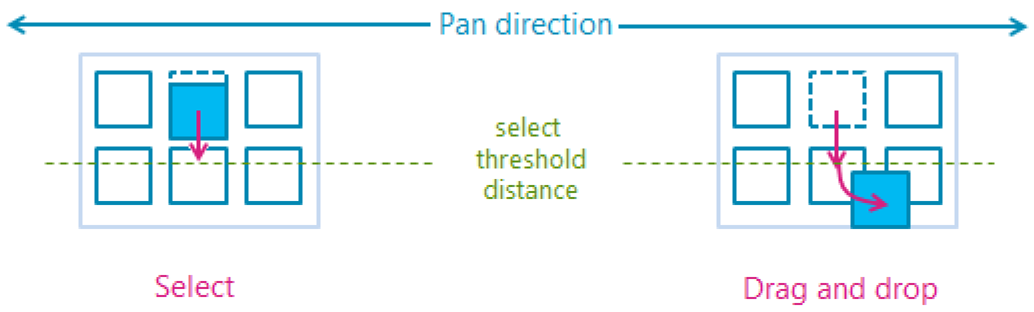
Cross-slide is the touch-optimized technique used by Windows Store apps in Windows 8 for selecting or dragging-and-dropping an item within a content area that is pannable in one dimension (vertical or horizontal).

Both the slide and swipe gestures use cross-slide functionality to manipulate an item by moving it some distance and, based on a distance threshold, provide the selection or drag-and-drop interaction. The gesture must be performed in a direction perpendicular to the panning direction.

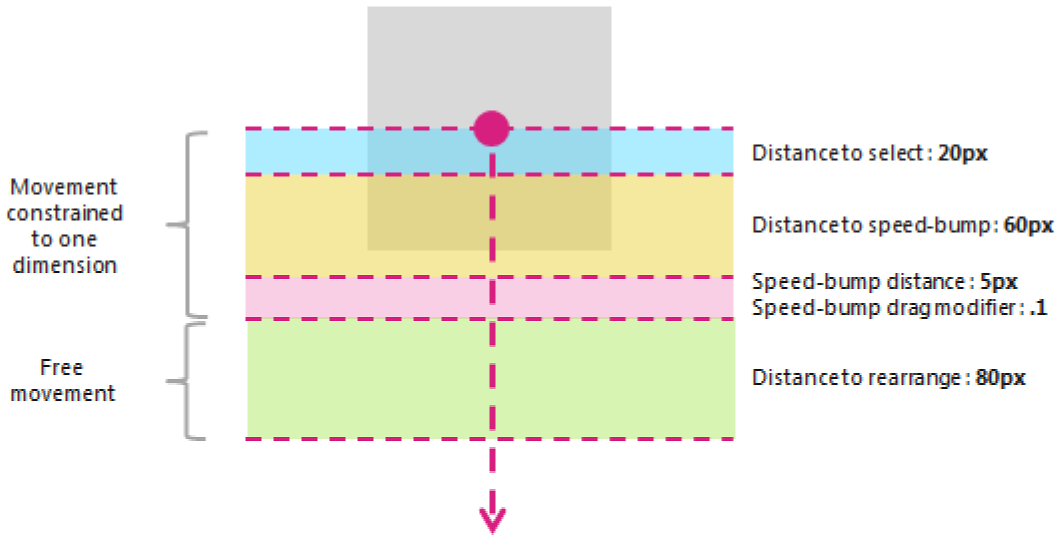
When to use cross-slide

Use cross-slide to let people select an item or drag-and-drop an item, if a distance threshold is crossed, using a slide or swipe gesture perpendicular to the single panning direction of a content area.

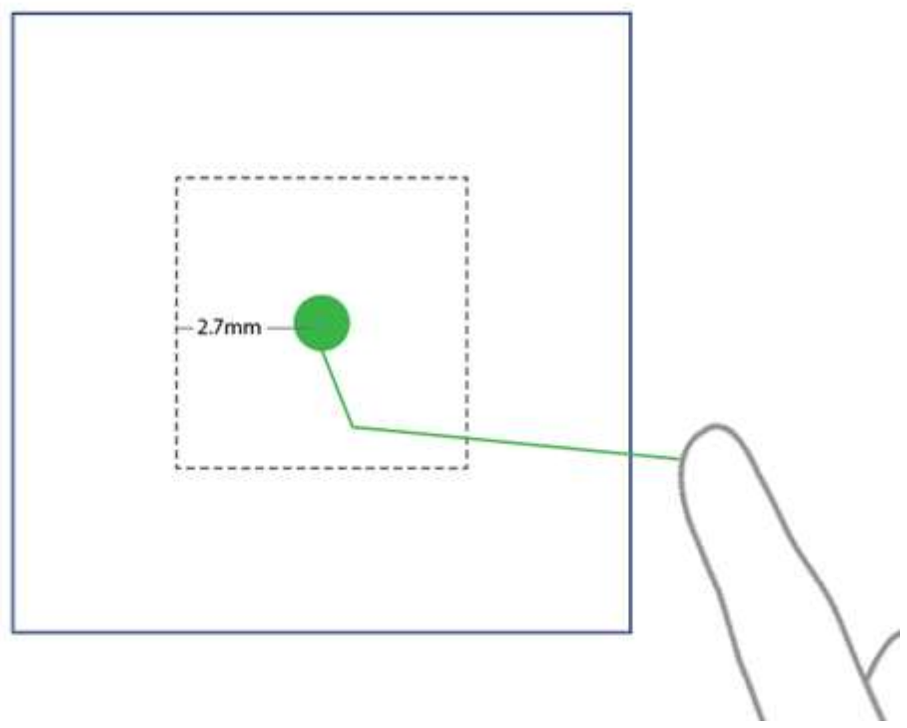
The following diagram demonstrates both selecting and moving an object by using cross-slide. The image on the left shows how an item is selected if a swipe gesture does not cross a distance threshold before the contact is lifted and the object released. The image on the right shows how a sliding gesture crosses a distance threshold and results in movement of the object.



The threshold distances used by the cross-slide interaction are shown in the following diagram.

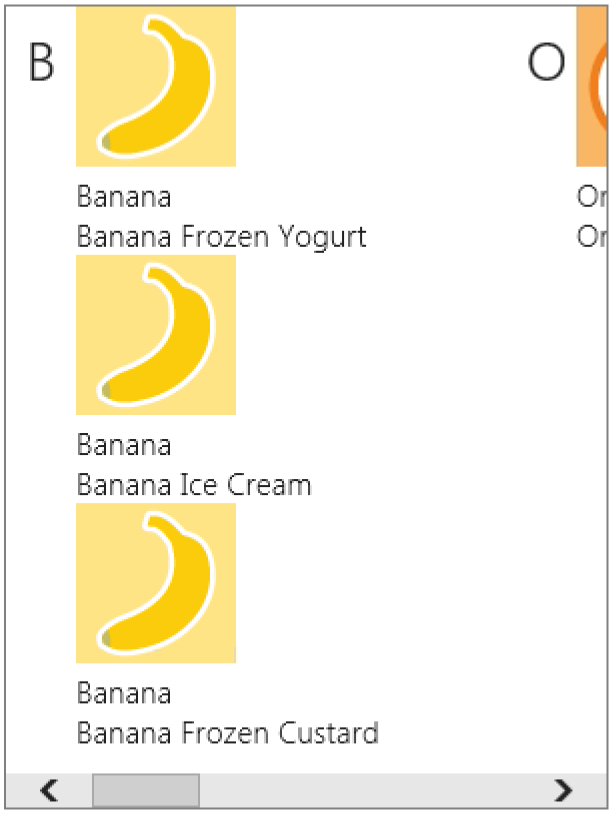
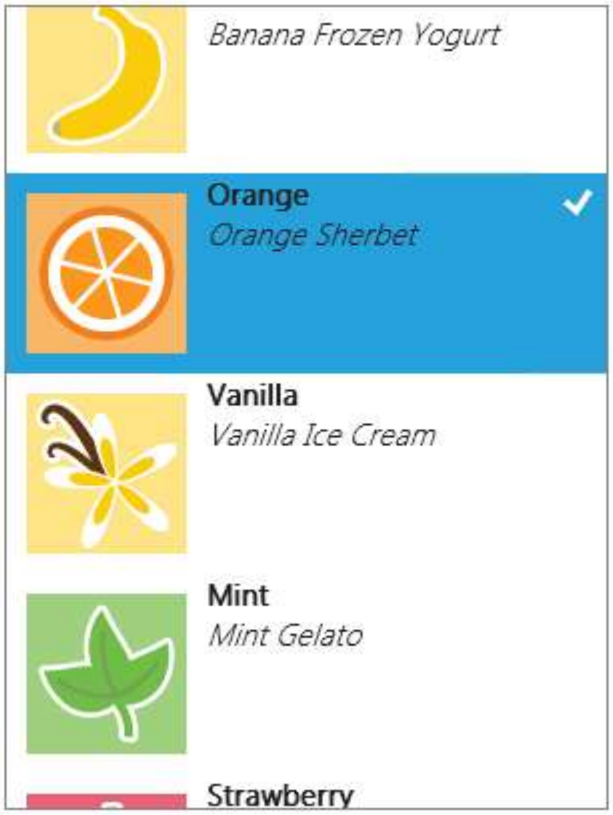

Movement constrained to one dimension
Free movement
Distance to select: 20px
Distance to speed-bump: 60px
Speed-bump distance: 5px
Speed-bump drag modifier: .1
Distance to rearrange: 80px

To preserve scrolling functionality, a small threshold of 2.7mm (approximately 10 pixels at target resolution) must be crossed before either a select or drag-and-drop interaction is activated. This small threshold helps the system to differentiate cross-sliding from panning, and also helps ensure that a tap gesture is distinguished from both cross-sliding and panning. The following diagram illustrates this technique.



This diagram illustrates how a user intends to scroll horizontally, but moves their finger down slightly at touch down. With no threshold, the interaction would be interpreted as a cross-slide because of the initial vertical movement. With the threshold, the movement is interpreted correctly as horizontal panning.

We strongly recommend cross-slide for lists or collections that scroll in a single direction.

 B Banana Banana Frozen Yogurt Banana Banana Ice Cream Banana Banana Frozen Custard O	 Banana Frozen Yogurt Orange Orange Sherbet Vanilla Vanilla Ice Cream Mint Mint Gelato Strawberry
A horizontally panning two-dimensional list. Drag vertically to select or move an item.	A vertically panning one-dimensional list. Drag horizontally to select or move an item.

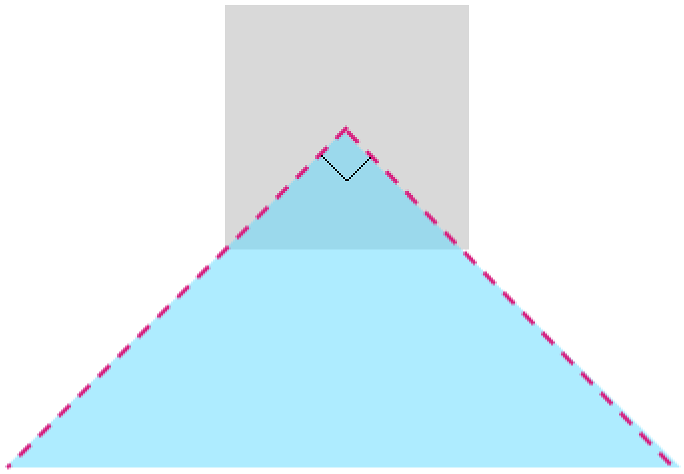
In cases where the content area can be panned in two directions, such as web browsers or e-readers, use the press-and-hold timed interaction to invoke the context menu .

Selecting

Selection is the marking, without launching or activating, of one or more objects. This action is analogous to a single mouse click, or Shift key and mouse click, on one or more objects.

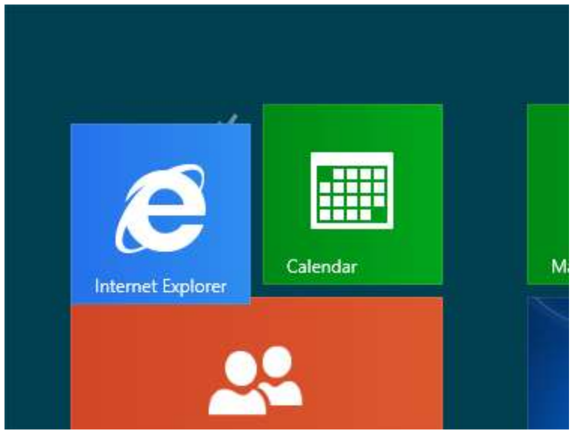
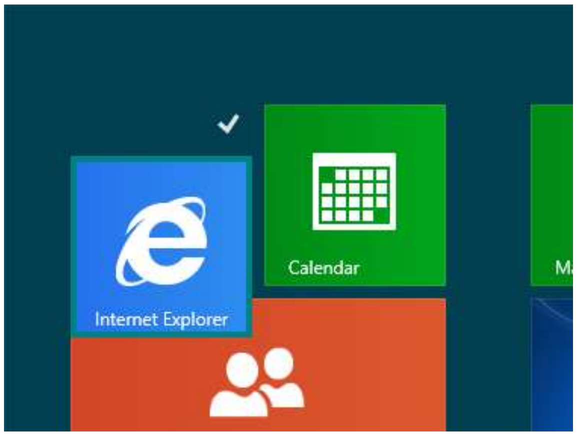
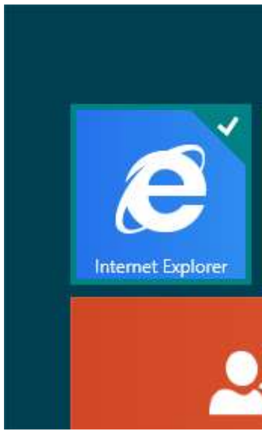
Cross-slide selection is achieved by touching an element and releasing it after a short dragging interaction. The cross-slide method of selection eliminates both the dedicated selection mode and the press-and-hold timed interaction required by other touch interfaces. The cross-slide method does not conflict with the tap interaction for activation.

In addition to the distance threshold, cross-slide selection is constrained to a 90° threshold area, as shown in the following diagram. If the object is dragged outside of this area, it is not selected.



The cross-slide interaction is supplemented by a press-and-hold timed interaction, also referred to as a "self-revealing" interaction. This supplemental interaction activates an animation that indicates what action will be performed when the interaction is released.

The following screen shots demonstrate how the self-revealing interaction animation works.

		
Unselected state. Press finger down to start cross-slide interaction.	Drag down to select. Self-revealing interaction demonstrates what action will be performed.	Continue to drag do revealing image cha object can now be c

Note In addition to the cross-slide, people can single tap for selection in apps where selection is the only primary action that can be performed. The cross-slide self-revealing animation is displayed to disambiguate this functionality from the standard tap interaction for activation and navigation.

Selection basket

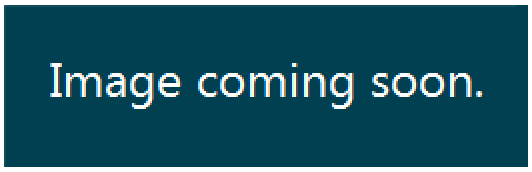
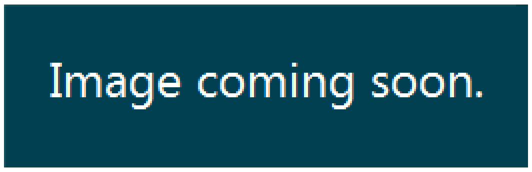
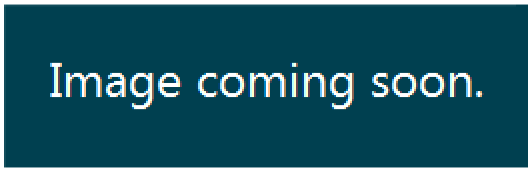
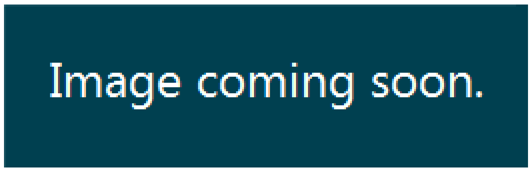
The selection basket is a visually distinct and dynamic representation of items that have been selected from the primary list or collection in the app. This feature is useful for tracking selected items. Apps should use it where:

- Items can be selected from multiple locations.
- Many items can be selected.
- An action or command relies upon the selection list.

Items in the selection basket can be removed with a cross-slide selection interaction on the item in the basket. This, in turn, cancels the selection of the corresponding item in the primary list. Canceling the selection in the primary list removes the item from the basket.

The selection basket also enables a user to select and cancel all items in the current list or collection at once by using a single interaction.

The following images demonstrate how the selection-basket works.

		
No items are selected.	Item is selected with cross-slide interaction.	Item is represented in sele
		
Additional items are selected.	Item is removed from selection basket with cross-slide interaction.	Item selection is canceled.

The content of the selection basket persists across actions and commands. For example, if you select a series of photographs from a gallery, apply a color correction to each photograph, and share the photographs in some fashion, the items remain selected.

If no selection basket is used in an app, the current selection should be cleared after an action or command. For example, if you select a song from a play list and rate it, the selection should be cleared.

The current selection should also be cleared when no selection basket is used and another item in the list or collection is activated. For example, if you select an inbox message, the preview pane is updated. Then, if you select a second inbox message, the selection of the previous message is canceled and the preview pane is updated.

Queues

A queue is not equivalent to the selection basket list and should not be treated as such. The primary distinctions include:

- The list of items in the selection basket is only a visual representation; the items in a queue are assembled with a specific action in mind.
- Items can be represented only once in the selection basket but multiple times in a queue.
- The order of items in the selection basket represents the order of selection. The order of items in a queue is directly related to functionality.

For these reasons, the cross-slide selection interaction should not be used to add items to a queue. Instead, items should be added to a queue through a drag-and-drop action.

Moving (drag and drop)

Drag-and-drop actions move an object from one location to another.

A cross-slide drag-and-drop action is achieved by touching an object and dragging it past a distance threshold. Windows Touch requires this distance threshold to differentiate between the drag-and-drop action and the selection action.

If more than one object needs to be moved, select each item with cross-slide selection and then drag them.

Guidelines for optical zoom and resizing

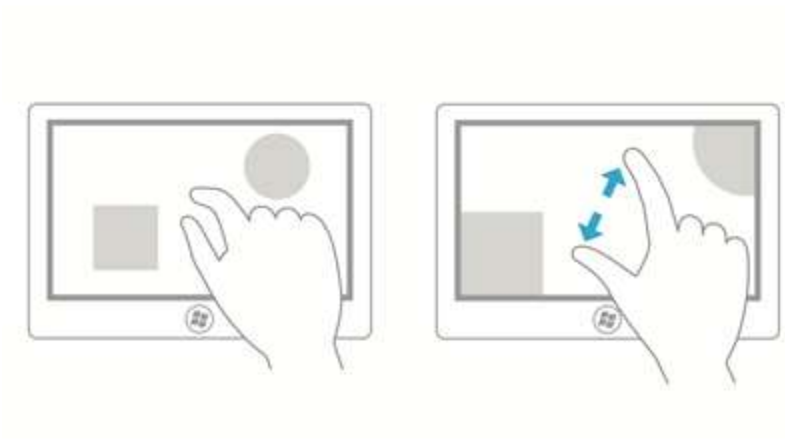
Optical zoom lets people magnify their view of the content within a content area (it is performed on the content area itself). Resizing, on the other hand, lets people change the relative size of one or more objects without changing the view into the content area (it is performed on the objects within the content area).

Optical zoom should not be confused with the Semantic Zoom interaction. While they share the same gestures, Semantic Zoom refers to the presentation and navigation of structured data or content within a single view. The view can be, for example, the folder structure of a computer, a library of documents, or a photo album.

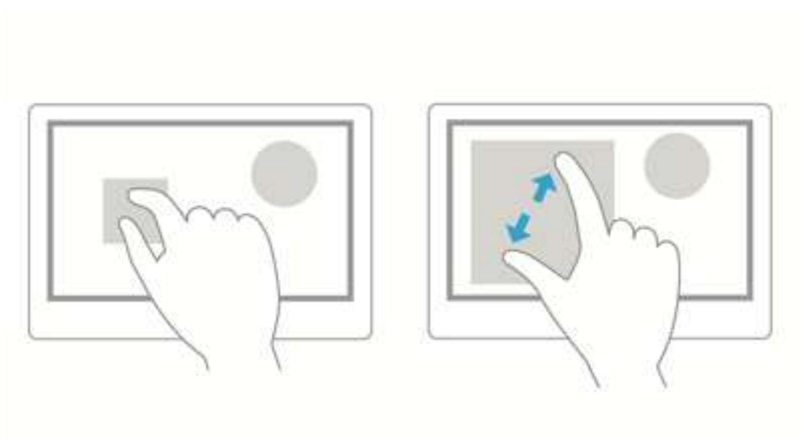
Both optical zoom and resizing interactions are performed through the pinch and stretch gestures (moving the fingers farther apart zooms in and moving them closer together zooms out), or by holding the Ctrl key down while scrolling the mouse scroll wheel, or by holding the Ctrl key down (with the Shift key, if no numeric keypad is available) and pressing the plus (+) or minus (-) key.

The following diagrams demonstrate the differences between resizing and optical zooming.

Optical zoom: User selects an area, and then zooms into the entire area.



Resize: User selects an object within an area, and resizes that object.



Use the following guidelines for apps that support either resizing or optical zooming:

- If maximum and minimum size constraints are defined, visual feedback should demonstrate when the user has reached those limits. When the user releases their touch, the content should snap to the minimum or maximum size.
- Resizing physics should be turned on. These include the following:
    - Acceleration: Repeated pinching accelerates resizing. This helps the user to reach the maximum or minimum size quickly.
    - Deceleration: Resizing decelerates when the user stops pinching. This is similar to sliding to a stop on a slippery surface.
    - Absorption: Resizing momentum during deceleration causes a slight bounce-back effect if a resizing constraint is reached.
- You can use snap points to influence zooming behavior by providing logical points within the content at which to stop zooming and ensure that a specific subset of content is displayed in the viewport. Provide snap points for common zoom levels or logical views to make it easier for a user to select those levels. For example, photo apps might provide a snap point at 100% for image resizing or, in the case of mapping apps, snap points might be useful for the transitions from city to state to country views.

Snap points enable users to be imprecise and still achieve their goals.

There are two types of snap-points:

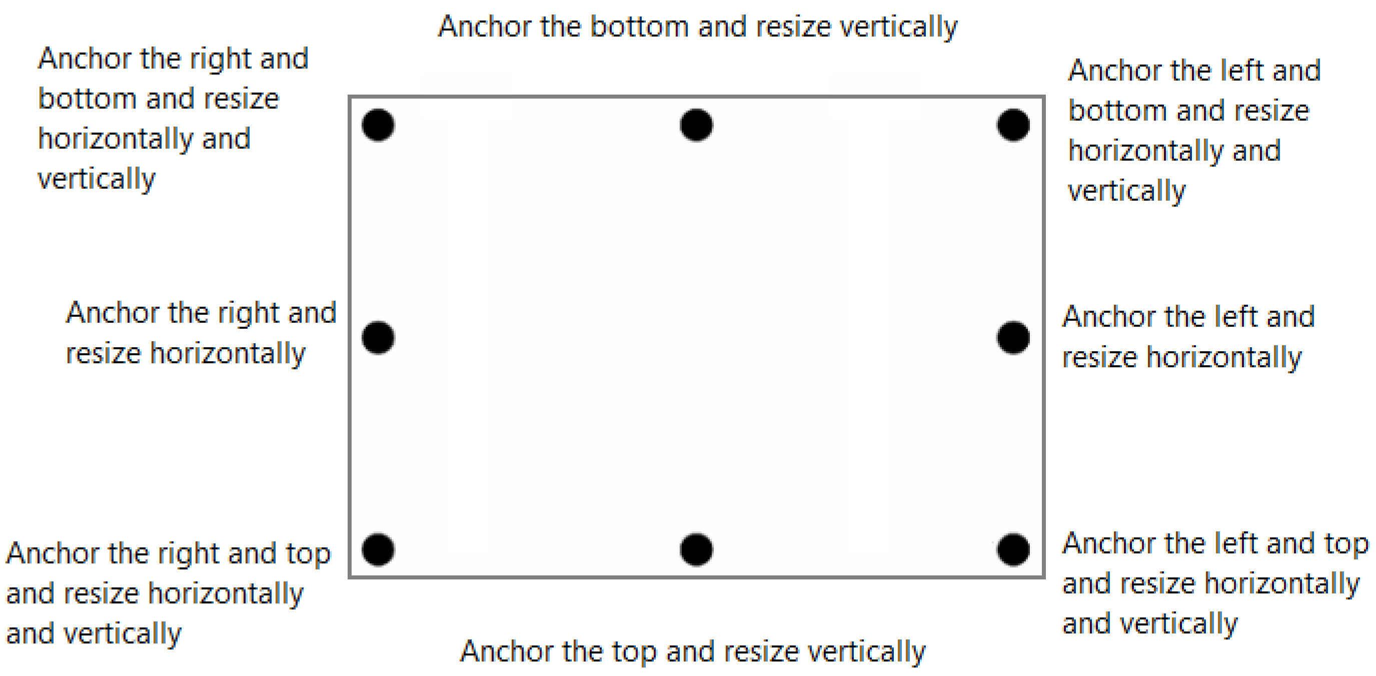
    - Proximity - After the contact is lifted, a snap point is selected if inertia stops within a distance threshold of the snap point.
    - Mandatory - The snap point selected is the one that immediately precedes or succeeds (depending on the direction of the gesture) the last snap point crossed before the contact was lifted. A snap point can't be skipped due to inertia.

- Space controls according to the guidelines for targeting.

- Don't use zooming to navigate the UI or expose additional controls within your app, use a panning region instead.

Use the following guidelines for apps that support resizing.

- Scaling handles, as shown in the following diagram, should be provided for constrained resizing. Isometric, or proportional, resizing is the default if the handles are not specified.


Anchor the bottom and resize vertically
Anchor the right and bottom and resize horizontally and vertically
Anchor the left and bottom and resize horizontally and vertically
Anchor the right and resize horizontally
Anchor the left and resize horizontally
Anchor the right and top and resize horizontally and vertically
Anchor the left and top and resize horizontally and vertically
Anchor the top and resize vertically

- In most cases, you should not put resizable objects within a resizable content area. Exceptions to this include:
  - Drawing apps where resizable items can appear on a resizable canvas or art board.
  - Webpages with an embedded object such as a map.

Note

In all cases, the content area is resized unless all touch points are within the resizable object.

Guidelines for panning

Panning or scrolling helps people navigate within a single view, to display the content of the view that does not fit (horizontally or vertically) within the viewport. The view can include, for example, the folder structure of a computer, a library of documents, or a photo album. Depending on the input device, panning or scrolling is performed within a pannable region by using:

- A mouse or active pen/stylus to click the scroll arrows, drag the scroll box, or click within the scroll bar.
- The wheel button of the mouse to emulate dragging the scroll box.
- The extended buttons (XBUTTON1 and XBUTTON2), if supported by the mouse.
- The keyboard arrow keys to emulate dragging the scroll box or the page keys to emulate clicking within the scroll bar.
- Touch or passive pen/stylus to slide or swipe the fingers in the desired direction.

Sliding (moving the fingers slowly in the panning direction) results in a one-to-one scrolling relationship where the content scrolls at the same speed and distance as the fingers. Swiping (rapidly sliding and lifting the fingers) results in the following physics being applied to the panning animation:

- Deceleration (inertia): Lifting the fingers causes panning to start decelerating. This is similar to sliding to a stop on a slippery surface.
- Absorption: Panning momentum during deceleration causes a slight bounce-back effect if either a snap point or a content area boundary is reached.

Note Panning physics are not exposed programmatically.

The scroll control

The scroll control provides assistance to people for panning or scrolling a view. It is exposed to developers of Windows Store apps using JavaScript at design time through Cascading Style Sheets (CSS) only. This control has two modes:

- Panning indicators when using touch.
- Scroll bars when using other input methods that include mouse, keyboard, and stylus.

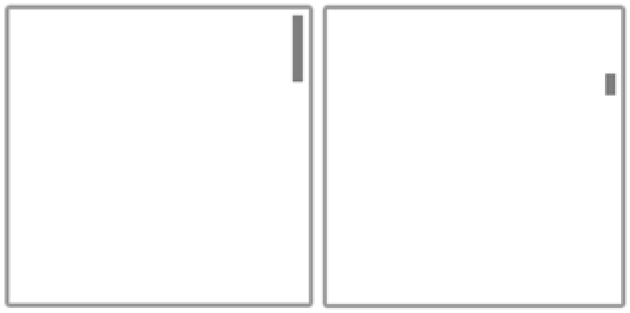
The control manages the modes , without any work by the developer.

Note The scroll control is a just-in-time control: the panning indicator is only visible when the touch contact is within the pannable region. Similarly, the scroll bar is only visible when the mouse cursor or keyboard focus is within the scrollable region.

Panning indicators

Panning indicators are similar to the scroll box in a scroll bar. They indicate the proportion of displayed content to total pannable area and the relative position of the displayed content in the pannable area.

The following diagram shows two pannable areas of different lengths and their panning indicators.



Types of panning

Windows 8 supports three types of panning:

- Single axis - panning is supported in one direction only (horizontal or vertical).
- Rails - panning is supported in all directions. However, once the user crosses a distance threshold in a specific direction, then panning is restricted to that axis.
- Freeform - panning is supported in all directions.

Snap points

Snap points can be used to influence panning behavior by providing logical points within the content at which to stop panning and ensure that a specific subset of content is displayed in the viewport. Cognitively, snap points act as a paging mechanism for the user.

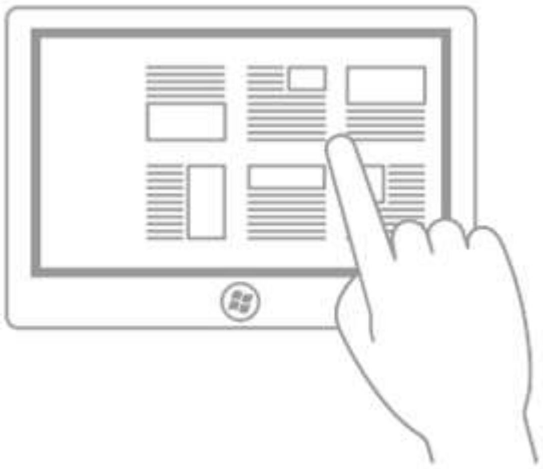
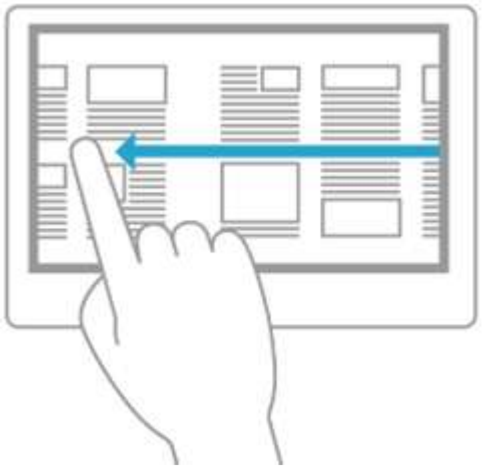
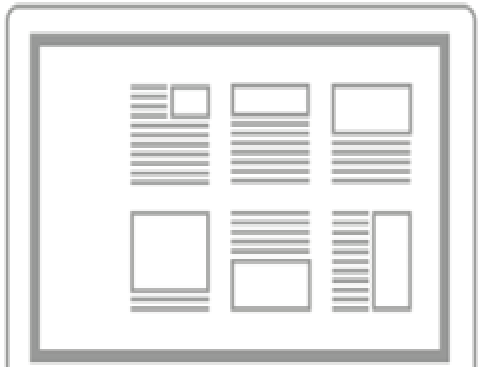
In addition, people tend to physically tire of excessive sliding or swiping in large pannable areas. Snap points enable people to be imprecise and still achieve their goals.

There are two types of snap-points:

- Proximity - After the contact is lifted, a snap point is selected if inertia stops within a distance threshold of the snap point.
- Mandatory - The snap point selected is the one that immediately precedes or succeeds (depending on the direction of the gesture) the last snap point crossed before the contact was lifted. A snap point cannot be skipped due to inertia.

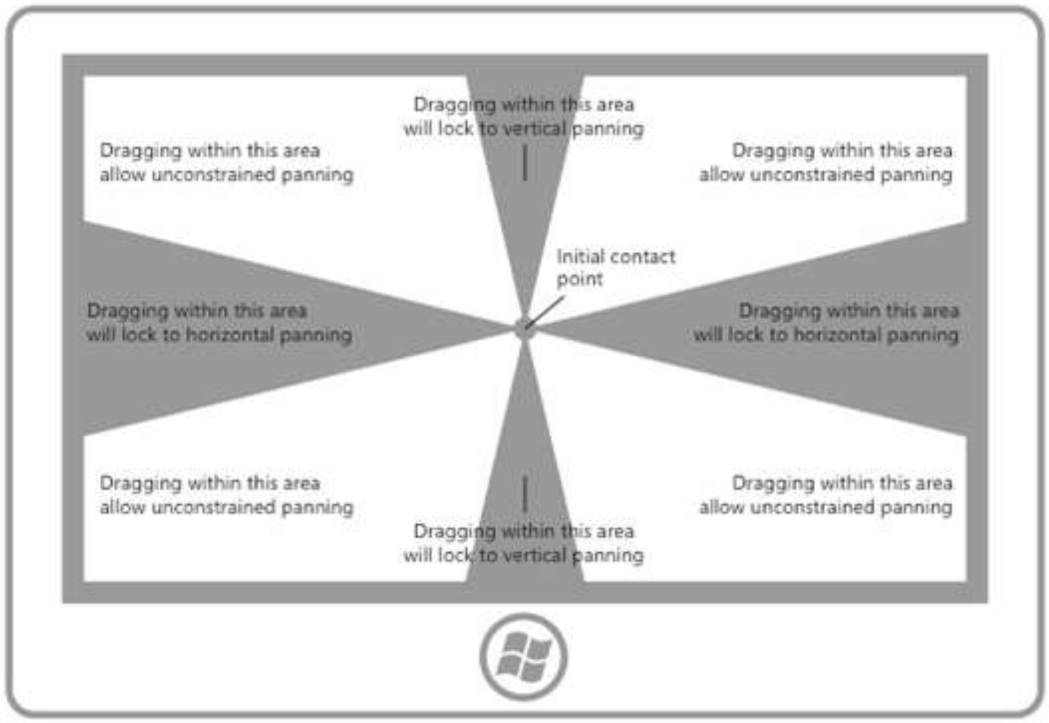
Panning snap-points are useful for apps such as web browsers and photo albums that emulate paginated content or have logical groupings of items that can be dynamically regrouped to fit within a viewport or display.

Panning to a certain point and releasing causes the content to pan to a logical location.

Begin panning content	
Pan to a midway point and release	
Pannable area moves to the snap point	

Rails

It is common for webpages to be wider and taller than the screen of a display device. For this reason, two-dimensional panning (horizontal and vertical) is often necessary. To improve the user experience in these cases, rails encourage panning in one dimension while locking out panning in the other dimension. This is accomplished by providing horizontal and vertical boundaries (rails) where panning is constrained unless the user drags a finger to exit the rail.


Dragging within this area will lock to vertical panning
Dragging within this area allow unconstrained panning
Dragging within this area allow unconstrained panning
Initial contact point
Dragging within this area will lock to horizontal panning
Dragging within this area will lock to horizontal panning
Dragging within this area allow unconstrained panning
Dragging within this area allow unconstrained panning
Dragging within this area will lock to vertical panning

User experience guidelines

Use the following guidelines for apps that support panning:

One-dimensional overflow

Use one-dimensional panning when the content area extends beyond the viewport along a single axis.

- Use vertical panning for a one-dimensional list of items.
- Use horizontal panning for a grid of items.

Don’t use mandatory snap-points if a user should be able to pan and stop between snap-points. Mandatory snap-points guarantee that the user will “stop” on a snap-point. Use proximity snap-points instead.

Two-dimensional overflow

Use two-dimensional panning when the content area extends beyond the viewport along both axes.

- Override the default rails behavior and use freeform panning for unstructured content where the user is likely to move in multiple directions. For example, freeform panning is typically suited to photographs or maps.

Paged view

Use mandatory snap-points when the content is composed of discrete elements or you want to display an entire element. This can include pages of a book or magazine, a column of items, or individual images.

- A snap-point should be placed at each logical boundary.
- Each element should be sized to fit the view.

Logical and key points

Use proximity snap-points if there are key points or logical places in the content that a user will likely stop.

- A snap-point should be placed at each section header.

Panning indicators and scroll bars

Display panning indicators and scroll bars to provide location and size cues. Hide them if the app provides a custom navigation feature.

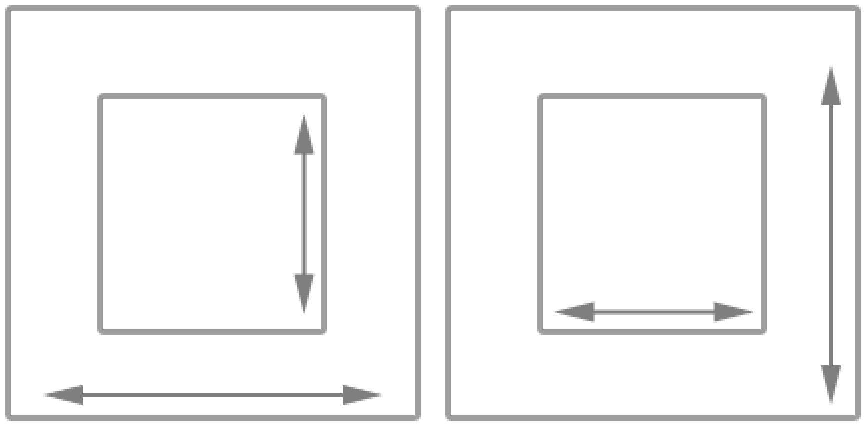
Note Unlike standard scroll bars, panning indicators are purely informative. They are not exposed to input devices and cannot be manipulated in any way.

Chaining embedded or nested content

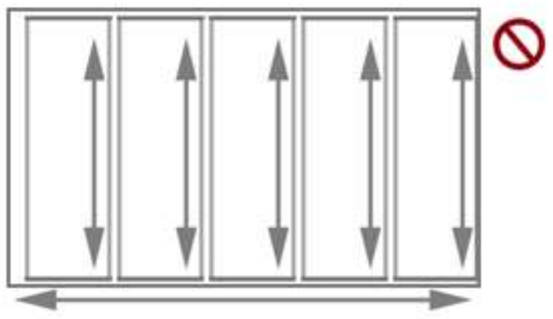
Chaining is used for panning within a single-axis content area that contains one or more single-axis or freeform panning regions (when the touch contact is within one of these child regions). When the panning boundary of the child region is reached in a specific direction, panning is then activated on the parent region in the same direction.

For example, a photo album could support freeform panning within each individual image while also supporting single-axis panning within the album to the previous or next images.

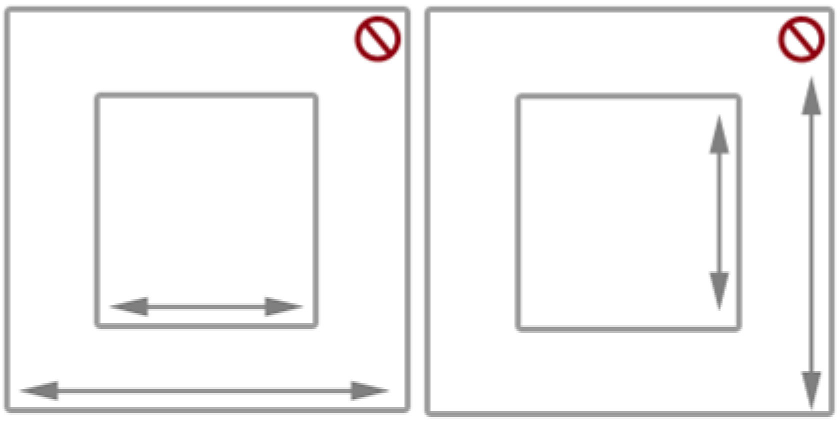
When a pannable region is placed inside another pannable region it's important to specify enough space between the container and the embedded content. In the following diagrams, one pannable region is placed inside another pannable region, each going in perpendicular directions. There is plenty of space for people to pan in each region.



Without enough space, as shown in the following diagram, the embedded pannable region can interfere with panning in the container. This can result in unintentional panning in one or more of the pannable regions.



Don't chain or place one pannable region within another pannable region if they both pan in the same direction, as shown in the following diagram. This can result in the parent area being panned unintentionally when a boundary for the child area is reached. Consider making the panning axis perpendicular.



Guidelines for rotation

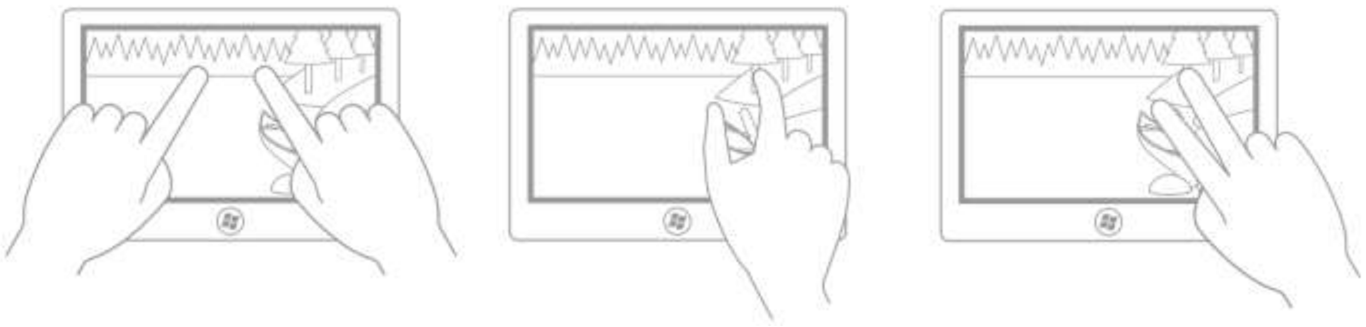
Rotation is the touch-optimized technique used by Windows Store apps in Windows 8 to enable people to turn an object in a circular direction (clockwise or counterclockwise).

Depending on the input device, the rotation interaction is performed by using:

- A mouse or active pen/stylus to move the rotation gripper of a selected object.
- Touch or passive pen/stylus to turn the object in the desired direction using the rotate gesture.

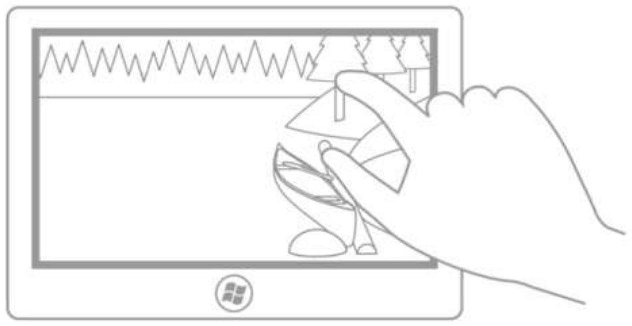
When to use rotation

Use rotation to help people directly rotate UI elements. The following diagrams show some of the supported finger positions for the rotation interaction.

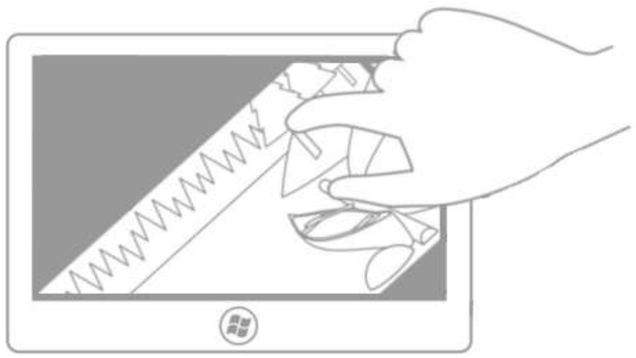


Note Intuitively, and in most cases, the rotation point is one of the two touch points unless the user can specify a rotation point unrelated to the contact points (for example, in a drawing or layout app). The following images demonstrate how the user experience can be degraded if the rotation point is not constrained in this way.

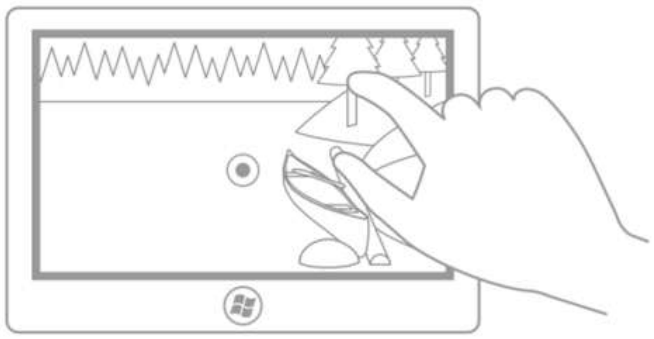
This first picture shows the initial (thumb) and secondary (index finger) touch points: the index finger is touching a tree and the thumb is touching a log.



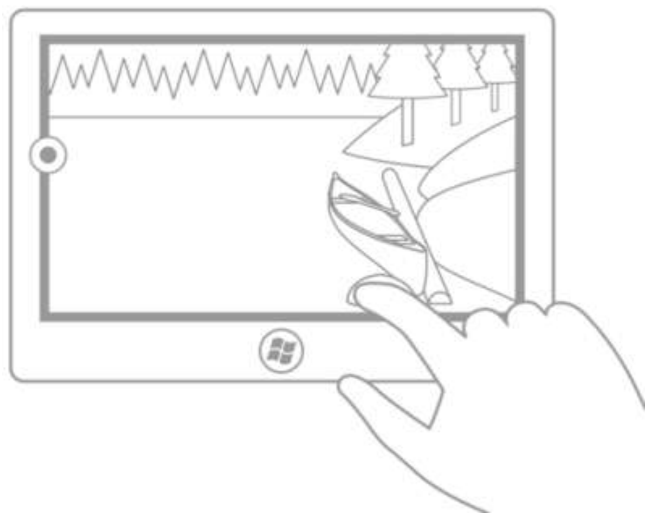
In this second picture, rotation is performed around the initial (thumb) touch point. After the rotation, the index finger is still touching the tree trunk and the thumb is still touching the log (the rotation point).



In this third picture, the center of rotation is defined by the app (or set by the user) to be the center point of the picture. After the rotation, because the picture did not rotate around one of the fingers, the illusion of direct manipulation is broken (unless the user has chosen this setting).



In this last picture, the center of rotation is defined by the app (or set by the user) to be a point in the middle of the left edge of the picture. Again, unless the user has chosen this setting, the illusion of direct manipulation is broken in this case.



Windows 8 supports three types of rotation: free, constrained, and combined.

Free rotation

Free rotation enables a user to rotate content freely anywhere in a 360 degree arc. When the user releases the object, the object remains in the chosen position. Free rotation is useful for drawing and layout apps such as Microsoft PowerPoint, Word, Visio, and Paint; and Adobe Photoshop, Illustrator, and Flash.

Constrained rotation

Constrained rotation supports free rotation during the manipulation but enforces snap points at 90 degree increments (0, 90, 180, and 270) upon release. When the user releases the object, the object automatically rotates to the nearest snap point.

Constrained rotation is the most common method of rotation, and it functions in a similar way to scrolling content. Snap points let a user be imprecise and still achieve their goal. Constrained rotation is useful for apps such as web browsers and photo albums.

Combined rotation

Combined rotation supports free rotation with zones (similar to rails in panning) at each of the 90 degree snap points enforced by constrained rotation. If the user releases the object outside of one of 90 degree zones, the object remains in that position; otherwise, the object automatically rotates to a snap point.

Note A user interface rail is a feature in which an area around a target constrains movement towards some specific value or location to influence its selection.

Guidelines for Semantic Zoom

Semantic Zoom is a touch-optimized technique used by Windows Store apps in Windows 8 for presenting and navigating large sets of related data or content within a single view. The single view may include, for example, a photo album, app list, or address book.

Note

This functionality is analogous to panning and scrolling (which can be used in conjunction with Semantic Zoom) within a single view.

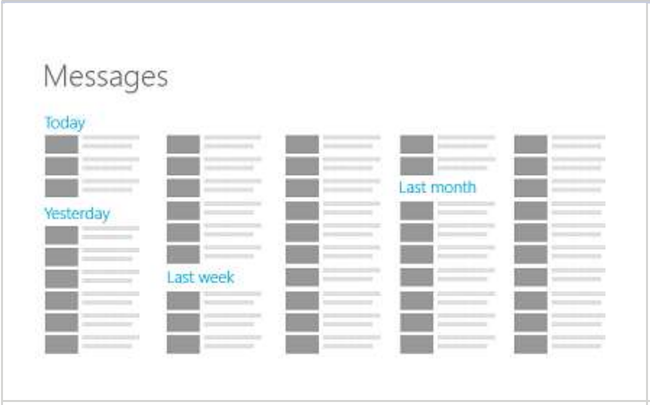
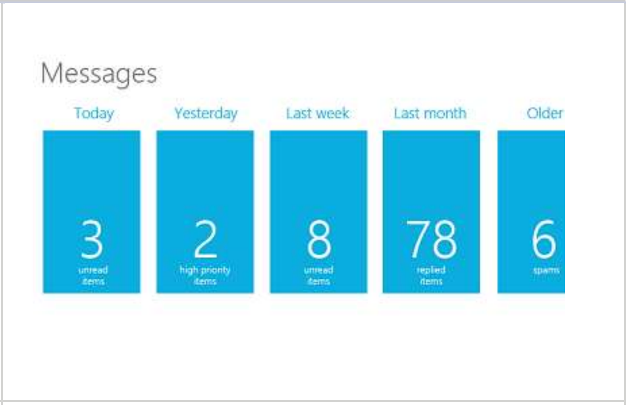
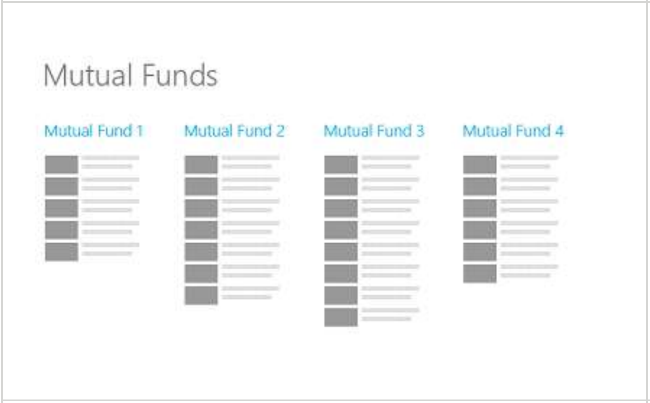
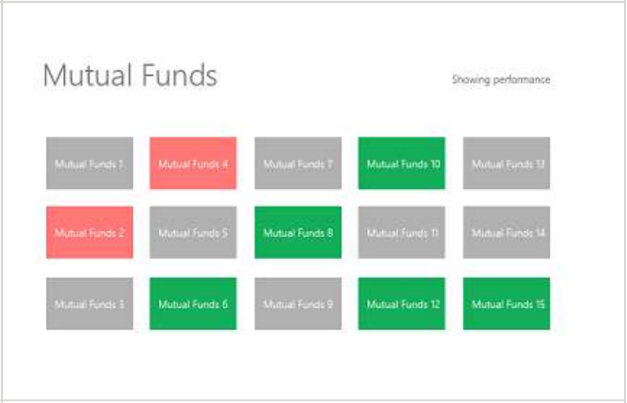
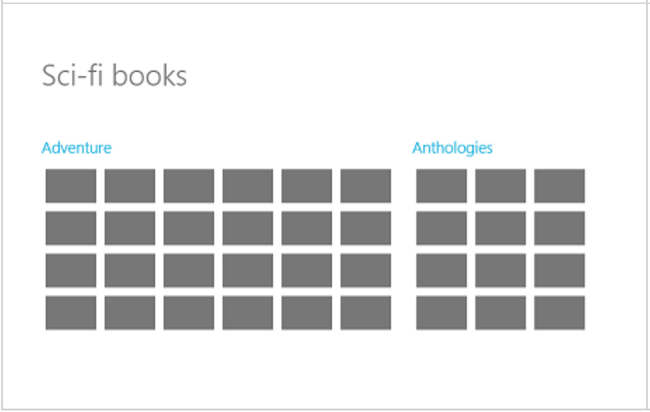
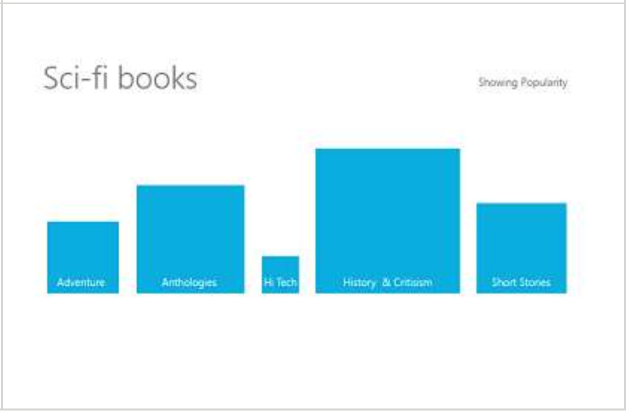
Semantic Zoom uses two distinct modes of classification (or zoom levels) for organizing and presenting the content: one low-level (or zoomed in) mode that is typically used to display items in a flat, all-up structure and another, high-level (or zoomed out) mode that displays items in groups and enables a user to navigate quickly and browse through the content.

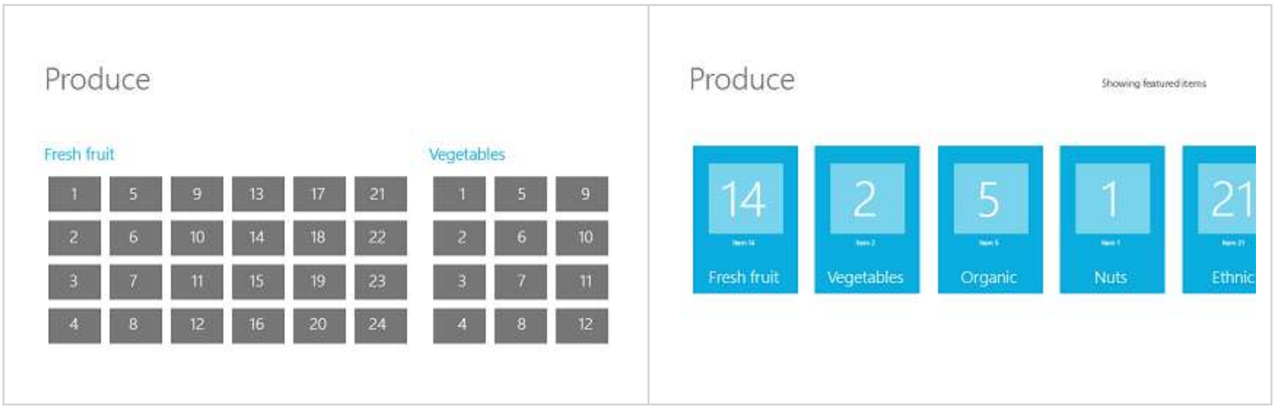
The Semantic Zoom interaction is performed with the pinch and stretch gestures (moving the fingers farther apart zooms in and moving them closer together zooms out), or by holding the Ctrl key down while scrolling the mouse scroll wheel, or by holding the Ctrl key down (with the Shift key, if no numeric keypad is available) and pressing the plus (+) or minus (-) key.

Examples of apps that could use Semantic Zoom include:

- An address book that organizes contacts alphabetically (or by some other means) and presents the data by using the letters of the alphabet. The user could then zoom in on a letter to see the contacts associated with that letter.
- A photo album that organizes images by metadata (such as date taken). The user could then zoom in on a specific date to display the collection of images associated with that date.
- A product catalog that organizes items by category.

Other examples of Semantic Zoom layouts:

Zoomed in	Zoomed out
	 Messages Today Yesterday Last week Last month Older 3 unread items 2 high priority items 8 unread items 78 replied items 6 spams
 Mutual Funds Mutual Fund 1 Mutual Fund 2 Mutual Fund 3 Mutual Fund 4	 Mutual Funds Showing performance Mutual Funds 1 Mutual Funds 4 Mutual Funds 7 Mutual Funds 10 Mutual Funds 13 Mutual Funds 2 Mutual Funds 5 Mutual Funds 8 Mutual Funds 11 Mutual Funds 14 Mutual Funds 3 Mutual Funds 6 Mutual Funds 9 Mutual Funds 12 Mutual Funds 15
	 Sci-fi books Showing Popularity Adventure Anthologies Hi Tech History & Criticism Short Stories



Navigating with Semantic Zoom

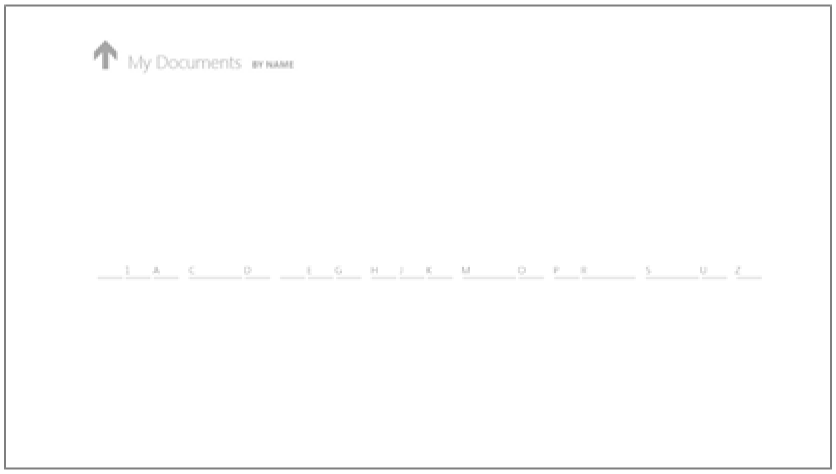
While navigating content is possible through panning and scrolling alone, powerful navigational and organizational capabilities are possible when paired with Semantic Zoom.

Panning and scrolling are useful for small sets of content and short distances. However, navigation quickly becomes cumbersome for large sets of content. Semantic Zoom greatly reduces the perception of traveling long distances when navigating through large amounts of content and provides quick and easy access to locations within the content.

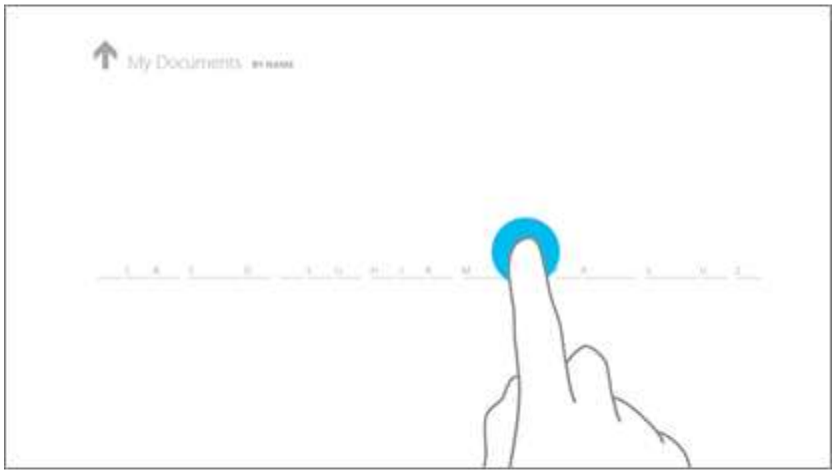
Note Semantic Zoom should not be confused with optical zoom. They do share interaction and basic behavior, displaying more or less detail based on a zoom factor. However, optical zoom refers to the adjustment of magnification for a content area or object such as a photograph.

Scroll jump

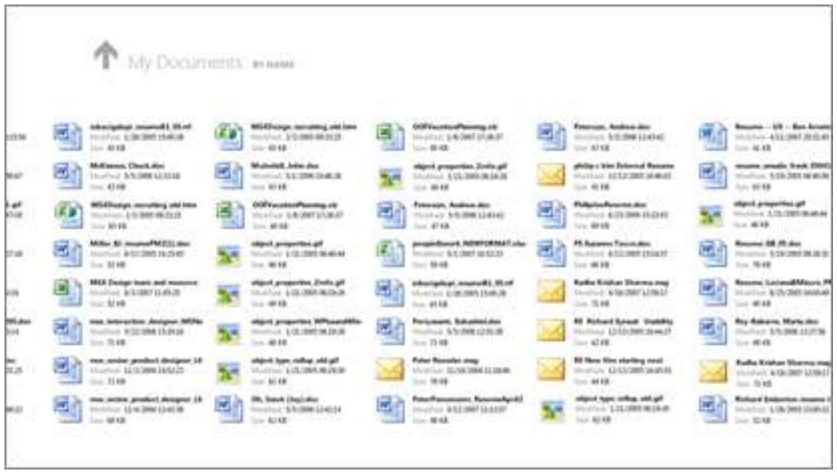
Tapping the content in zoomed-out mode zooms the view and pan to the tapped point, as shown in these three diagrams.



Zoomed out, the entire content can be a touch target.



A tap on a section of the content.



Zoomed in and panned to the tapped area.

Transitions

A smooth cross-fade and scale animation is used for the transition from one semantic zoom level to another. This is the default Windows Touch behavior and cannot be customized.

Considerations and recommendations

You are responsible for defining the two semantic levels in your apps.

Consider these questions when designing the zoomed-out mode:

- How should the structure and presentation of the info change based on the zoom level?
- Would hints, or "signposts," be helpful for navigating the data?
- What amount of content provides a useful semantic view while minimizing panning and scrolling?

These considerations are often in conflict with each other. You want rich content with lots of info so that people know where they are jumping to. But you need to balance this info with the total length of the semantic level. If people need to pan a lot in the zoomed-out mode, you lose the main benefit provided by Semantic Zoom—quick navigation.

Dos and Don'ts

The following dos and don'ts ensure a successful Semantic Zoom experience for your customers.

Dos

- Use the correct touch target size for elements that are useable or interactive.
- Provide a suitable and intuitive Semantic Zoom region.
- People often initiate Semantic Zoom from within the area that surrounds the displayed items. Make the Semantic Zoom region large enough to encompass this area. For example, the Windows Store provides a large amount of white space around the app list where the user can place their fingers and zoom in or out.
- Use structure and semantics that are intrinsic to the view.
- Use the group names for items in a grouped collection.
- Use sort ordering (such as chronological ordering for dates or alphabetical ordering for a list of names) for an ungrouped, but sorted, collection.
- Use pages to represent a document collection.
- Ensure that the item layout and panning direction does not change based on zoom level.
- Layouts and panning interactions should be consistent and predictable across zoom levels.
- Limit the number of pages (or screens) in the zoomed-out mode to three.
- Semantic Zoom enables a user to jump quickly to content. Introducing excessive panning destroys this benefit.

Don'ts

- Don't use Semantic Zoom to change the scope of the content.
- For example, a photo album should never switch to a folder view in File Explorer.
- Don't set a border on the SemanticZoom control's child controls.
- If you set borders on both the **SemanticZoom** and its child controls, the **SemanticZoom** border and the border of the child control that is in view are both visible. When zooming in or out, the child control's borders are scaled along with the content and don't look good. Set a border only on the **SemanticZoom** control.

Guidelines for selecting text and images

Touch optimization

Text selection and manipulation are particularly susceptible to user experience challenges introduced by touch interactions. Mouse, pen/stylus, and keyboard input are highly granular: a mouse click or pen/stylus contact is typically mapped to a single pixel, and a key is pressed or not pressed. Touch input is not granular; it's difficult to map the entire surface of a fingertip to a specific x-y location on the screen to place a text caret accurately.

To accommodate the less precise targeting behavior of touch interactions, the visuals for the selection and manipulation user experience have been completely redesigned for Windows 8. These visuals include "grippers", which both indicate that a selection can be adjusted and identify the interaction target.

Note

Grippers are used for text selection, media controls, resizing, and image cropping.

With touch, selection interactions are performed primarily through gestures such as a tap to set an insertion cursor or select a word, and a slide to modify a selection. As with other Windows 8 touch interactions, timed interactions are limited to the press and hold gesture to display a context menu.

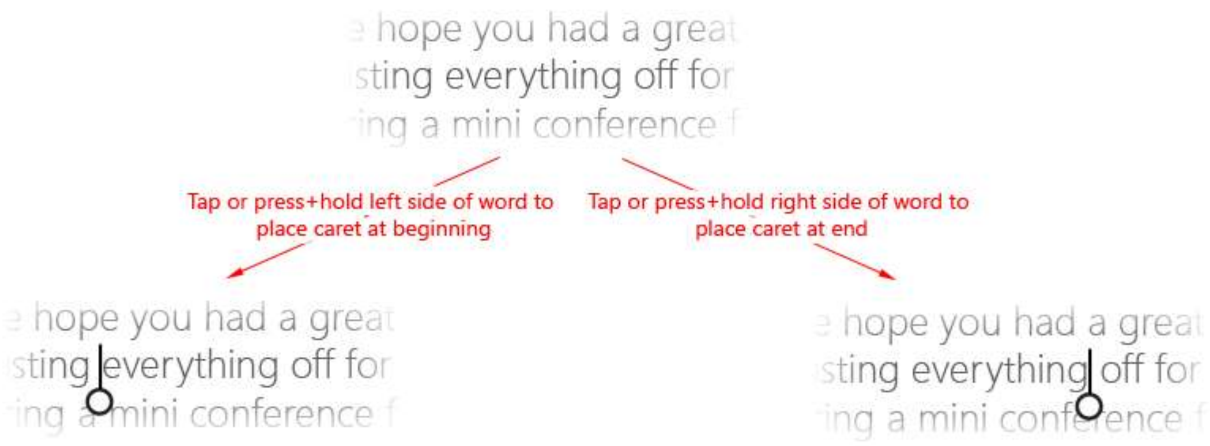
Interactions through mouse, pen/stylus, and keyboard all behave as expected with the new selection visuals.

Editable and non-editable content

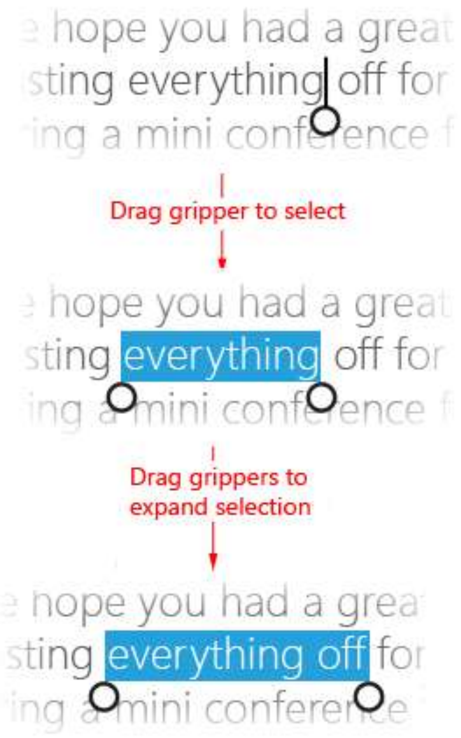
Windows 8 recognizes two possible states for selection interactions, editable and non-editable, and adjusts selection UI, feedback, and functionality accordingly.

Editable content

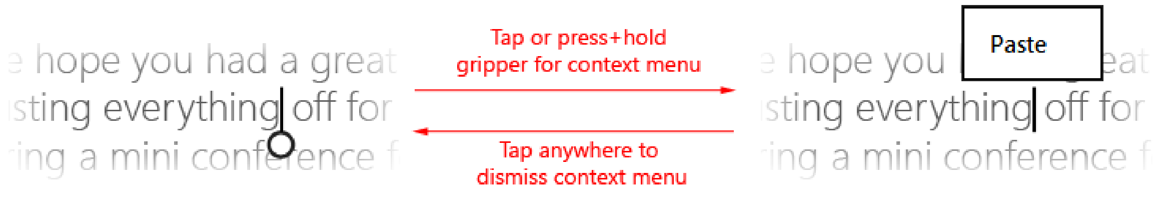
The following image demonstrates how to place an initial insertion cursor with gripper by tapping near the beginning or ending of a word.

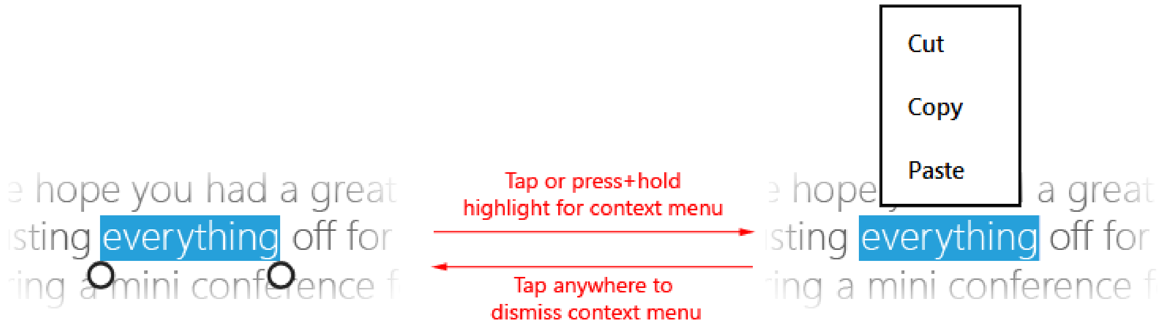

hope you had a great
sting everything off for
ing a mini conference f
Tap or press+hold left side of word to place caret at beginning
Tap or press+hold right side of word to place caret at end

The following image demonstrates how to adjust a selection by dragging the gripper.



The following images demonstrate how to invoke the context menu by tapping within the selection or on a gripper (press and hold can also be used).

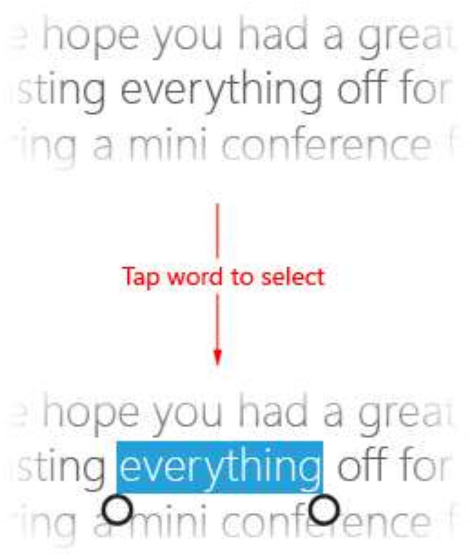

e hope you had a great
sting everything off for
ing a mini conference f
Tap or press+hold gripper for context menu
Tap anywhere to dismiss context menu
Paste



Note These interactions vary somewhat in the case of a misspelled word. Tapping a word that is marked as misspelled will both highlight the entire word and invoke the suggested spelling context menu.

Non-editable content

The following image demonstrates how to select a word by tapping within the word (no spaces are included in the initial selection).



Follow the same procedures as for editable text to adjust the selection and display the context menu.

Guidelines for targeting

Touch targeting in Windows 8 uses the full contact area of each finger that is detected by a touch digitizer. The larger, more complex set of input data reported by the digitizer is used to increase precision when determining the user's intended (or most likely) target. This ensures a much more satisfying experience for the user by improving accuracy and instilling confidence in their touch interactions.

The following recommendations describe how to optimize your app for touch targeting.

Measurements and scaling

To be consistent across different screen sizes and pixel densities, all target sizes are represented in physical units (millimeters). Physical units can be converted to pixels by using the following equation:

Pixels = Pixel Density × Measurement

The following example uses this formula to calculate the pixel size of a 9 mm target on a 135 pixel per inch (PPI) display at a 1x scaling plateau:

Pixels = 135 PPI × 9 mm

Pixels = 135 PPI × (0.03937 inches per mm × 9 mm)

Pixels = 135 PPI × 0.35433 inches

Pixels = 48 pixels

This result must be adjusted according to each scaling plateau defined by the system.

Thresholds

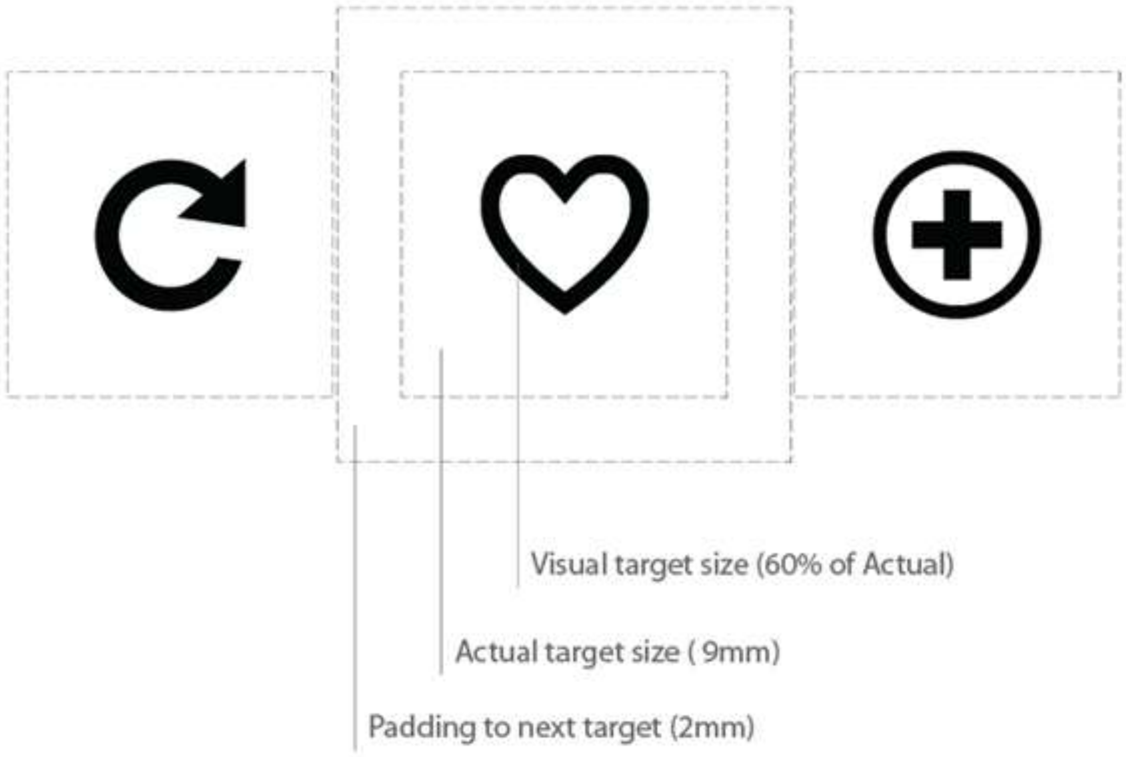
Distance and time thresholds may be used to determine the outcome of an interaction.

For example, when a touch-down is detected, a tap is registered if the object is dragged less than 2.7 mm from the touch-down point and the touch is lifted within 0.1 second or less of the touch-down. Moving the finger beyond this 2.7 mm threshold results in the object being dragged and either selected or moved. Depending on your app, holding the finger down for longer than 0.1 second may cause the system to perform a self-revealing interaction.

Target sizes

There are no definitive recommendations for how large a target should be or where it should be placed within your app. The size and target area of an object depend on various factors, including the user experience scenarios and interaction context.

The following diagram shows how target size is typically a combination of a visual target, actual target size, and any padding between the actual target and other potential targets.


Visual target size (60% of Actual)
Actual target size (9mm)
Padding to next target (2mm)

The following table lists the minimum and recommended sizes for all components of a touch target.

Target component	Minimum size	Recommended size
Padding	2 mm	Not applicable.
Visual target size	< 60% of actual size	90-100% of actual size
Actual target size	9 x 9 mm (48 x 48 px @ 1x) For elements smaller than recommended, the distance from the center of one element to the center of the other	Not applicable

	element should be at least 9mm.	
Total target size	11 x 11 mm (approximately 60 px: three 20-px grid units @ 1x)	13.5 x 13.5 mm (72 x 72 px @ 1x) This implies that the size of the actual target and padding combined should be larger than their respective minimums.

These target size recommendations can be adjusted as required by your particular scenario. Some of the considerations that went into these recommendations include:

- Frequency of Touches: Consider making targets that are repeatedly or frequently pressed larger than the minimum size.
- Error Consequence: Targets that have severe consequences if touched in error should have greater padding and be placed further from the edge of the content area. This is especially true for targets that are touched frequently.
- Position in the content area
- Form factor and screen size
- Finger posture
- Touch visualizations
- Hardware and touch digitizers

Targeting assistance

Windows provides targeting assistance to support scenarios where the minimum size or padding recommendations presented here are not applicable. These scenarios include, for example, hyperlinks on a webpage, calendar controls, drop down lists and combo boxes, or text selection.

These targeting platform improvements and user interface behaviors work together with visual feedback (disambiguation UI) to improve user accuracy and confidence.

If a touchable element must be smaller than the recommended minimum target size, the following techniques can be used to minimize the targeting issues that result.

Tethering

Tethering indicates to a user that they are connected to, and interacting with, an object even though the input contact isn't directly in contact with the object. Tethering is a visual cue, shown as a connector from a contact point to the bounding rectangle of an object. Tethering can occur when:

- A touch contact was first detected within some proximity threshold to an object, and this object was identified as the most likely target of the contact.
- A touch contact was moved off an object but the contact is still within a proximity threshold.

This feature is not exposed to Windows Store app using JavaScript developers.

Scrubbing

Scrubbing means to touch anywhere within a field of targets and slide to select the desired target without lifting the finger until it is over the desired target. Scrubbing is also referred to as "take-off activation", where the object that is activated is the one that was last touched when the finger was lifted from the screen.

Use the following guidelines when you design scrubbing interactions:

- Scrubbing is used in conjunction with disambiguation UI.
- The recommended minimum size for a scrubbing touch target is 20 px (3.75 mm @ 1x size).
- Scrubbing takes precedence when performed on a pannable surface, such as a webpage.
- Scrubbing targets should be close together.
- An action is canceled when the user drags a finger off a scrubbing target.
- Tethering to a scrubbing target is specified if the actions performed by the target are non-destructive, such as switching between dates on a calendar.
- Tethering is specified in a single direction, horizontally or vertically.

Guidelines for visual feedback

You should be sure that your app provides people with visual feedback on their touch interactions. Visual feedback helps people recognize how Windows and your app interpret their touch input. Visual feedback can indicate successful interactions, relay system status, improve the sense of control, reduce errors, encourage interaction, and help people understand, learn, and adapt to the system and input device. Touch visualizations are critical for activities that require accuracy and precision.

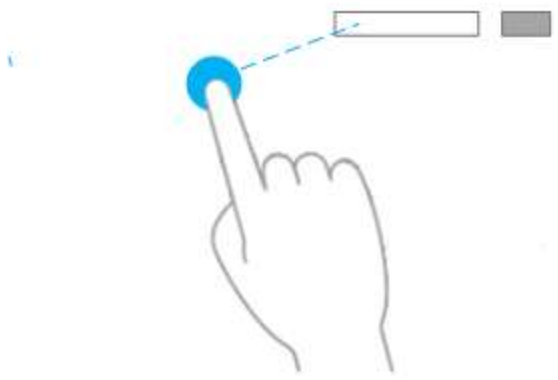
For example, if a user attempts to tap a control but misses, the location of the tap should be clearly identified so people know how far their tap was from the target.

If you're using the standard Windows UI and controls we provide, you get that visual feedback for free. If you create custom UI, then you need to consider when and how to provide appropriate feedback. Visual feedback might be inappropriate for games or drawing apps where the feedback distracts from the user task without providing much added information.

Visual feedback

If you need to implement custom visual feedback, also called touch visualizations, follow these guidelines.

- All controls must provide touch feedback.
- Touch events should provide feedback no matter how brief the contact:
    - To confirm that the touch screen is working.
    - To show that a target was not touched and the user must try again.
    - To show that a target is not touch-enabled.
    - To show that a control or app is not responding.
- Feedback must be immediate for all touch events.
- Feedback should be shown only for the control interpreted as touched.
- Feedback should consist of subtle, intuitive cues that do not distract people from their intended action.
- Disambiguation, or informational, UI should identify the control, show available functionality, and provide guidance where necessary. The next section describes informational UI in greater detail.
- Do not show visual feedback during panning or dragging; the actual movement of the object on the screen is sufficient. However, if the content area does not pan or scroll, then you should use touch visualizations to indicate the boundary conditions.
- Touch targets should cling to the fingertip during any manipulation.
- Show the "tether" visualization when the finger is dragged off the element, but not lifted, to demonstrate that the element is still the active element. The following screen shot shows how tethering maintains the visual and mental connection to the active element.



An app can opt out of touch visualizations, but this is recommended only for situations where the visualizations interfere with the use of the app. These situations can occur, for example, with a game or drawing app.

Informational user interface

Informational UI, also known as disambiguation UI, helps overcome fingertip occlusion, displays information about an object, and describes functionality and how to access it. Tooltips, rich tooltips, and context menus are used to implement this touch feature.

A timed interaction, touch-and-hold, is dedicated to the display of informational UI. The touch-and-hold interaction involves touching the screen without lifting the finger for a specified amount of time until an informational UI is displayed. A timed interaction is acceptable in this case as it acts as a tool for learning and exploration.

The recommended amount of time depends on the type of informational UI being presented, as described in the following table.

Informational UI type	Timing	Activation	Description
Occlusion tooltip (for scrubbing and small targets)	0 ms	Yes	Intended for rapid clarification of actions. Typically used for commands.
Occlusion tooltip (for actions)	200 ms	Yes	

Informational pop-up (rich tooltip)	~2000 ms	No	Intended for slower, more deliberate exploration and learning. Typically used with collection items.
Self-revealing interaction	~2000 ms	No	
Context menu	~2000 ms	No	Exposes a limited set of commands related to the selected object.

Occlusion tooltips for scrubbing and small targets

These tooltips should describe the occluded target. These tooltips are useful when targeting and activating items smaller than a standard touch target size, such as hyperlinks on a webpage.

You can replace these tooltips by an informational pop-up after a certain time threshold has passed. After this initial threshold, the tooltip should follow the behavior of the informational pop-up. For example, this would be used in a browser where the occlusion tooltip shows the occluded text and, after the time threshold, then shows the full URL.

Occlusion tooltips for actions and commands

These tooltips should describe the action that occurs when the finger is released from an element. These tooltips are useful when targeting and activating a button or similar control.

It is acceptable for a small-target tooltip to be followed by an action tooltip after a certain time threshold has passed. In this case, the small-target tooltip should expand to include the additional information in the action tooltip.

Rich tooltip or informational pop-up

These tooltips should reveal secondary information about an element. For example, a rich tooltip could be a text description of an image, the full text of a truncated title, or other information relevant to the target.

Rich tooltips or information pop-ups typically contain information that does not need to be made available immediately and, in some cases, might be distracting if shown too quickly. A longer time threshold lets people be more deliberate about obtaining the information.

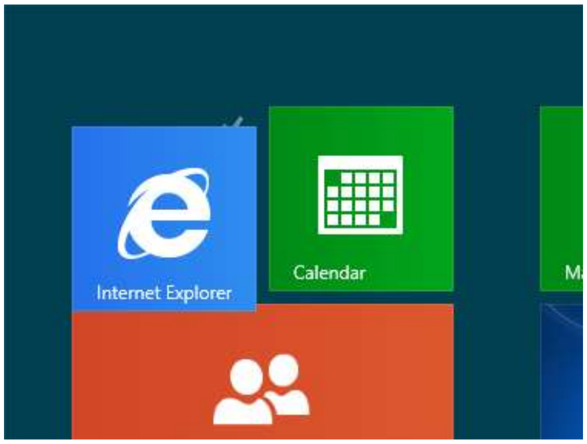
After a rich tooltip is displayed, the object is no longer activated when the user lifts their finger. The reason for this is that information gleaned from the tooltip might influence the user to not to activate the item.

We recommend that the visual design and information in the rich tooltip be distinct and more substantial than that of a standard tooltip.

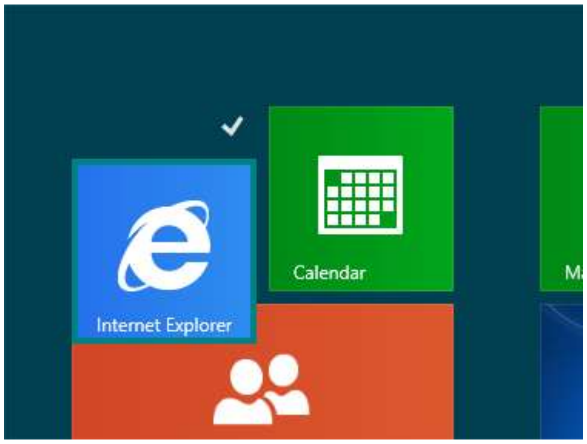
Self-revealing interactions

A self-revealing interaction is an informative visual cue (or animation) that demonstrates how to perform an action with a target object. It provides a preview of the result of that action.

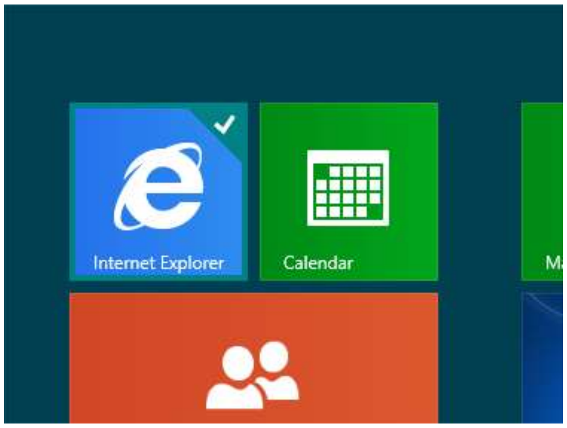
The following images show the self-revealing interaction for a cross-slide selection on the Start screen. When a user touches an app tile (without dragging the tile) the tile slides down (as if being dragged) to reveal the selection check mark that would appear if the app were actually selected.



Unselected state. Press finger down to start cross-slide interaction.



Drag down to select. Self-revealing interaction demonstrates what action will be performed.



Continue to drag down and the self-revealing image changes to show that the object can now be dragged and dropped.

After a self-revealing interaction is displayed, the object is no longer activated when the user lifts their finger.

Context menu

The touch-optimized context menu is composed of two parts. A visual cue, the hint, is displayed as a result of a hold interaction. Then, the context menu itself is displayed after the hint disappears and the finger is lifted.

Important The context menu should be used only where selection is not possible.

Guidelines for touch keyboard

The Windows 8 touch keyboard enables text entry for form factors that don't have a hardware keyboard or other peripheral keyboard devices. The touch keyboard is invoked when a user taps on an editable input field, and is dismissed when the input field loses focus. The touch keyboard is used for text entry only.

The following table presents the practices recommended for programming Windows Store apps that use the touch keyboard for text entry.

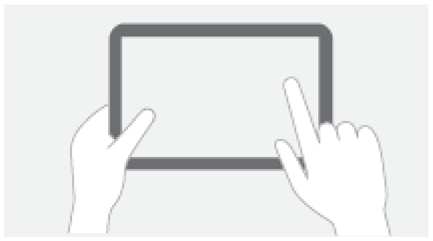
Practice	Description
Use the touch keyboard for text input only, not for commands or keyboard shortcuts	The touch keyboard is present on touch-enabled systems when the user needs to input text. To dismiss the touch keyboard, either move focus to another control or set the input field to read-only.
Maintain keyboard presence throughout a user flow.	If you're creating custom UI, make sure your custom controls have the proper UI Automation ControlType. This ensures keyboard persistence when focus moves from a text input field while in the context of text entry. For example, if you have a menu that's opened in the middle of a text-entry scenario, and you want the keyboard to persist, the menu must have the ControlType Menu.
Ensure that people always can see the input field that they're typing into.	The touch keyboard occludes half of the screen. Windows Store apps provide a default experience for managing UI when the touch keyboard appears, by ensuring that the input field with focus scrolls into view. Handle the **Showing** and **Hiding** events exposed by the **InputPane** object to customize your app's reaction to the keyboard's appearance.
Implement UI Automation properties for custom controls that have text input.	Standard controls for Windows Store apps have these properties, but custom controls require you to implement TextPattern. For the keyboard to persist contextually as focus changes to different controls, a custom control must have one of the following properties: • Button • Check box

	• Combo box • Radio button • Scroll bar • Tree item • Menu • Menu item
Don't use the touch keyboard as a commanding and controlling device.	The touch keyboard doesn't provide many of the accelerators or command keys found on a hardware keyboard, such as alt, the function keys, or the Windows Logo key. Don't make people navigate their app by using the keyboard.
Don't keep the keyboard displayed only to keep the touch keyboard on the screen.	When you don't expect text entry to occur, don't display the touch keyboard.
Don't manipulate UI Automation properties to control the touch keyboard.	Other accessibility tools rely on the accuracy of UI Automation properties.

Commanding design

You have several surfaces you can place commands and controls on in your Windows Store app, including the app window, pop-ups, dialogs, and bars. Choosing the right surface at the right time can mean the difference between an app that's a breeze to use and one that's a burden.

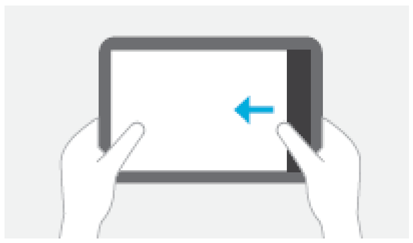
Use the canvas



Users should be able to complete the core scenarios just by using the canvas. Whenever possible, let users directly manipulate the content on the app's canvas, rather than adding commands that act on the content.

For example, in a Restaurant browsing app, finding and viewing restaurant details should be done on the canvas by tapping, panning, or selecting content.

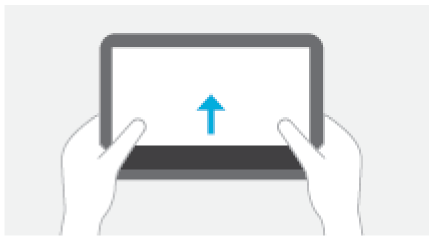
Use the charms



Leverage the charm and app contracts to enable common app commands. Avoid duplicating the functionality of app contracts on your app's canvas or in the app bar.

- **Search:** Let your users quickly search through your app's content from anywhere in the system, including other apps. And vice versa.
- **Share:** Let your users share content from your app with other people or apps, and receive shared content.
- **Devices:** Let your users enjoy audio, video, or images streamed from your app to other devices in their home network.
- **Settings:** Consolidate all of your settings under one roof and let users configure your app with a common mechanism they're already familiar with.

Use the app bar



Use the app bar to display commands to users on demand. The app bar shows commands relevant to the user's context, usually the current page, or the current selection.

The app bar is not visible by default. It appears when a user swipes a finger from the top or bottom edge of the screen. The app bar can also appear programmatically on object selection or on right click.

The app bar is transient, going away after the user taps a command, taps the app canvas, or repeats the swipe gesture. If needed, you can keep the app bar visible to ease multi-select scenarios.

Use context menus

You can use context menus for clipboard actions (like cut, copy, and paste), or for commands that apply to content that cannot be selected (like an image on a web page).

The system provides apps with default context menus for text and hyperlinks. For text, the default context menu shows the clipboard commands. For hyperlinks, the default menu shows commands to copy and to open the link.

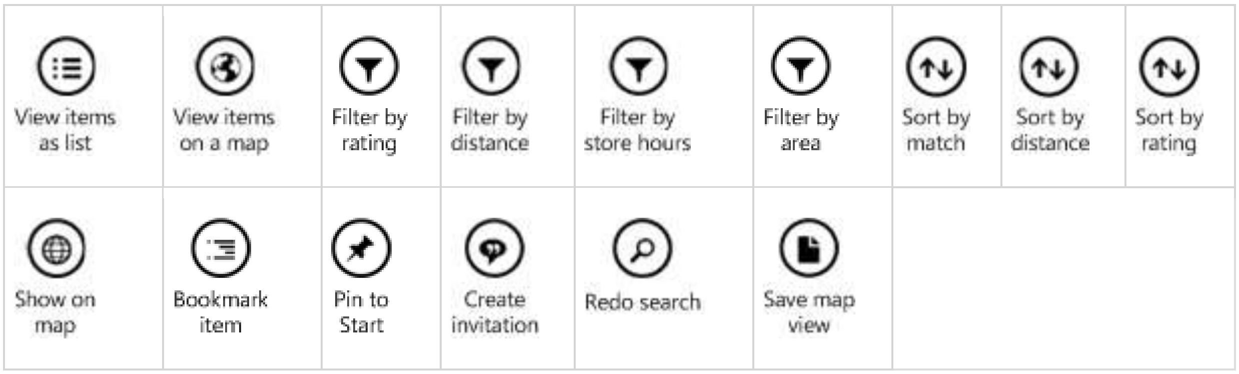
Command placement

Let's use a fictional restaurant app to illustrate the process of organizing commands for the app bar, focusing on a browsing scenario.

Organize commands

The first step is to identify all the app commands and organize them by the scenario or location. The following list of commands are commonly used when browsing for a restaurant.

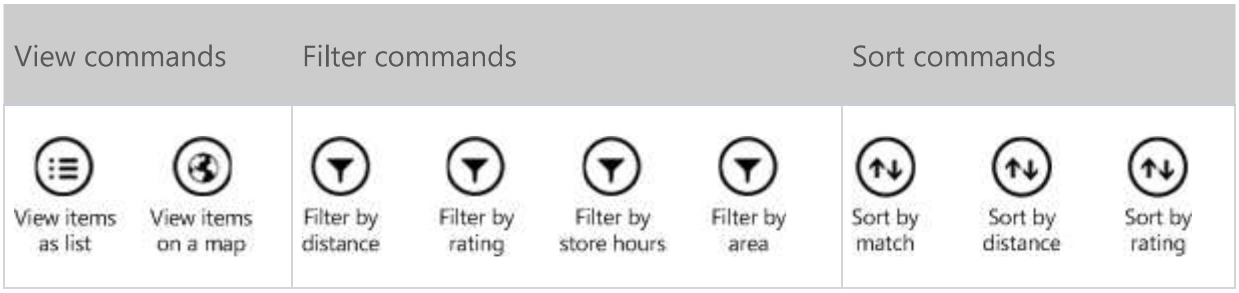
- What commands should appear throughout the entire app?
- What commands should show only on certain pages?
- What commands should use charms or go in settings?



Create command sets

Next we group commands into command sets. The app bar displays command sets as a unit, with a divider between the sets.

- What commands are functionally related?
- What commands toggle different view types?
- What commands should appear when a selection is made?

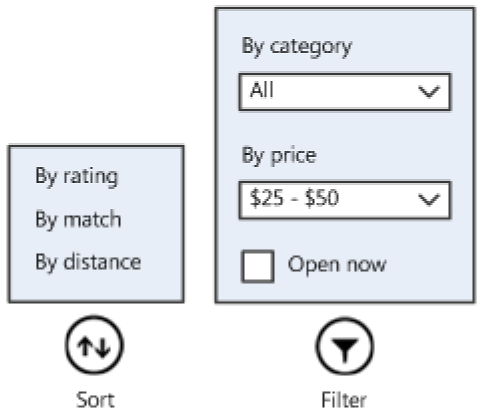


Selection commands	Map view commands	New Item commands

Create menus

Next, consider whether your command sets would work better in a command menu.

- Is the app bar too crowded or are there too many commands to fit?
- Is there a set that would benefit from longer labels or interactive controls?



Menus let you present more options in less space and include interactive controls.

In this example, the Sort menu pops up a simple list that makes choosing options easy. The Filter menu pops up a set of controls that lets users filter items by more complex criteria.

Place commands on the app bar

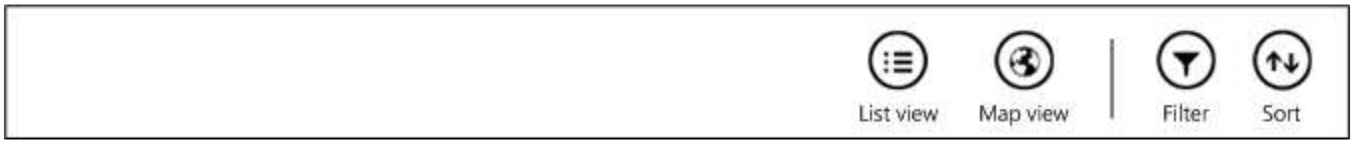
There are a few ways to position commands within the app bar, and variations may occur depending upon certain circumstances. Follow these command placement rules whenever possible.

Predictability To the extent possible, use consistent interaction and command placement across all views of your app.

Ergonomics Consider how the placement of specific commands can improve the speed or ease with which a command can be acted upon.

Aesthetics Limit the number of commands to avoid the app bar from looking complicated. Choose icons that are easy to understand or predict. Keep text labels short.

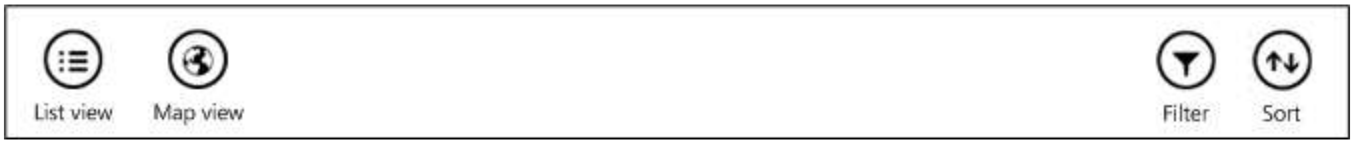
1. Place persistent commands on the right



Start by placing default commands on the right side of the app bar. If there are only a few commands, the app bar may end up with commands only on the right.

In this example for the Browse commands, the view command set and the filter/sort set are persistent.

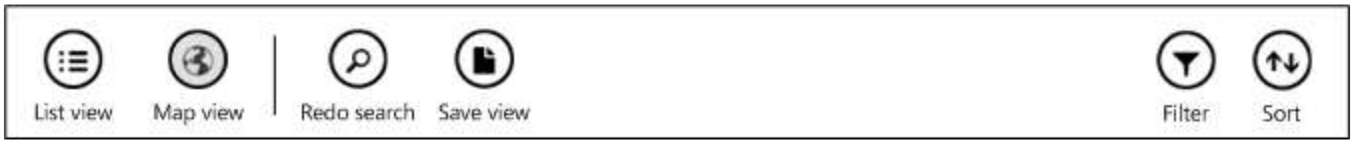
2. Use the edges



If there is a larger number of commands, separate the distinct sets of commands on the left or the right to balance out the app bar and to make commands more ergonomically accessible.

Here we decide to move the view command set to the left and keep the filter/sort set on the right. In this example, when map view is active the map view commands appear to the right of the view command set.

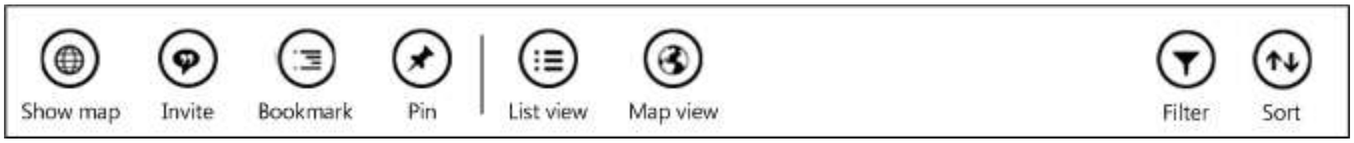
3. Show/hide disabled commands



Commands that are not relevant in certain circumstances should be hidden. When they do appear, they should not disrupt the ordering of persistent commands.

In this example, when map view is active the map view commands appear to the right of the view command set.

4. Insert selection commands



Commanding design

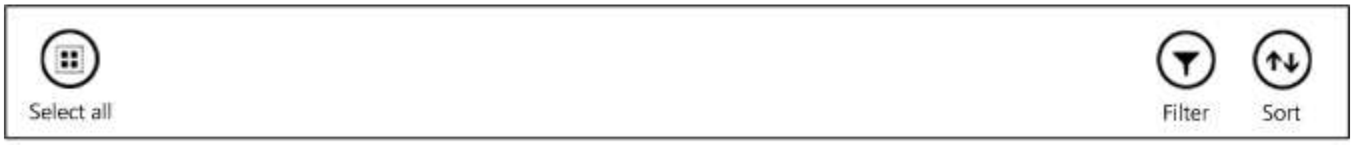
Commands that appear when the user makes a selection go on the far left, sliding over any commands that may have been there. This makes selection commands more noticeable and easier to access.

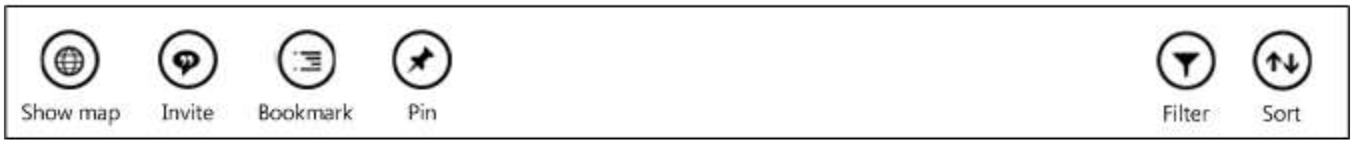
Here the view command set slides over to the right to make room for the selection command set.

Use standard placement for common commands

Some commands are common and appear in many apps. To create consistency and instill confidence, follow these guidelines when deciding where to place commands in the app bar.

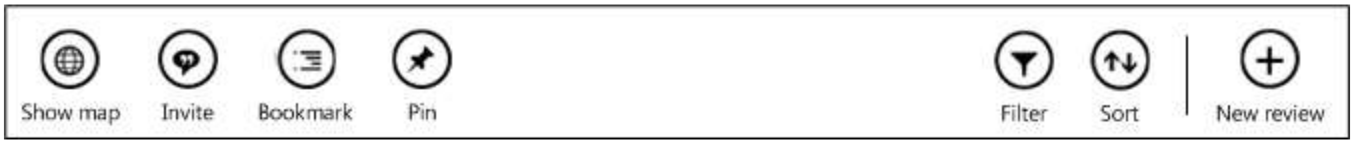
Selection commands Commands related to your selection always appear on the far left, whether they are contextual commands that appear on selection, or commands that affect your selection.





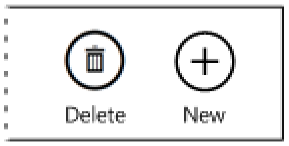
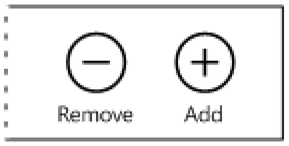
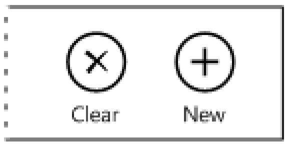
In this example, before users select anything, a "Select all" command appears on the left. After users select something, the other selection commands appear on the left.

New Item command If your app calls for a "New" command, where any new type of entity is created (add, create, compose), place that command against the right edge of the bar. This gives every "New" command, regardless of the specific app or context, consistent placement and makes it easily accessible with thumbs.



In this example, the "New review" command lets users create a new restaurant review. Other commands, related to "New review," are placed next to it to the left.

The + glyph should only be used to represent the "New" command, and it should not appear anywhere else in an app bar.

Delete commands Use Delete/New if your app is about managing individual entities that may persist outside of your particular app, like in a mail or camera app. Delete/New should always appear in this order.	
Remove commands Use Remove/Add if your app is about managing a list, such as a to-do list, a list of cities in a weather app, or a list of bookmarked restaurants. Remove should always appear to the left of Add.	
Clear commands Use clear if you are taking a destructive action on all possible items. Use the command label to be explicit about what the command will act on, such "Clear selection."	

Controls

Guidelines for text input

Is this the right control?

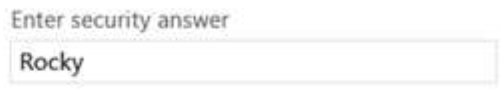
Text input controls let people enter and edit a text or numeric value. Consider these questions when deciding whether to use a text input control.

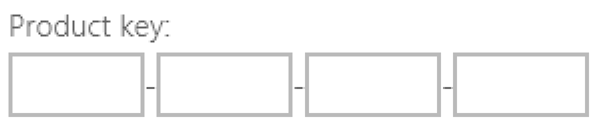
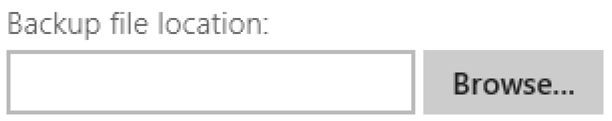
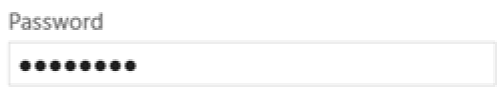
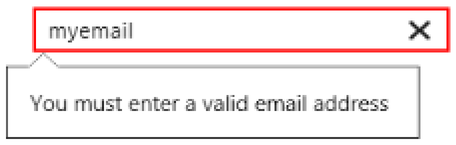
- **Is it practical to enumerate all the valid values efficiently?** If so, consider using one of the selection controls instead.
- Is the valid data completely unconstrained? Or is the valid data constrained only by format (constrained length or character types)? If so, use a text input control.
- **Does the value represent a data type that has a specialized common control?** If so, use the appropriate control instead of a text input control. For example, use a **DatePicker** instead of a text input control to accept a date entry.
- If the data is numeric:
    - Do people perceive the setting as a relative quantity? If so, use a slider.
    - Would the user benefit from instant feedback on the effect of setting changes? If so, use a slider, possibly along with an accompanying control.

There are single-line and multi-line text input controls. The next section describes when to use single-line text input controls, and later sections describe multi-line text input controls.

Choosing the right single-line text input control

For short strings, use a single-line text input control. This table describes when to use the different types of text input controls.

Basic data input	Use single-line text input controls to gather small pieces of text from people. The following example shows a single-line text box to capture an answer to a security question. The answer is expected to be short, and so a single-line text box is appropriate here. Because the information collected does not match any of the specialized input types that Windows recognizes, the generic "Text" type is appropriate.  Enter security answer Rocky

Formatted data input	Use a set of short, fixed-sized, single-line text input controls to enter data with a specific format. 
Assisted data input	Use a single-line, unconstrained text input control to enter or edit strings, combined with a command button that helps people select valid values. 
Numeric input	Use a single-line, number input control to enter or edit numbers.
Password and PIN input	Use a single-line password input control to enter passwords and PINs securely. 
Email input	Use the single-line email input control to enter an email address. When you use an email input control, you get the following for free: • When people navigate to the text box, the touch keyboard appears with an email-specific key layout. • When people enter an invalid email format, a dialog appears to let them  know.

URL input	Use the URL input control for entering web addresses.
Telephone number input	Use the telephone number input control for entering telephone numbers.

Dos and don'ts for single-line input boxes

Do	Use several single-line text boxes to capture many small pieces of text information. If the text boxes are related, group them together.
	Provide placeholder text in your single-line text boxes if you believe people need instructions for entering a value.
	Make the size of your single-line text boxes slightly wider than the longest anticipated input. If doing so makes the control too wide, separate it into two controls; for example, you could split a single address input into "Address line 1" and "Address line 2".
	Set a maximum length. If the backing data source doesn't allow a long input string, limit the input and use a validation popup to let people know when they reach the limit.
Don't	**Don't use a text area with a row height of 1 to create a single-line text box.** Instead, use the **input type="text"** element.
	Don't use placeholder text to pre-populate the text control. Text boxes clear placeholder text when people use the control. Use the "value" attribute instead.
	Don't use a text box as a search box. It's common practice in web pages to use an **input** element to create a search box. However, you create a much better and more consistent experience when you use the Search charm instead. The Search charm provides a consistent searching experience that your app can plug into.

	Don't put another control right next to a password input box. The password input box has a password reveal button for people to verify the passwords that they have typed. Having another control right next to it might make people accidentally reveal their passwords when they try to interact with the other control. To prevent this from happening, put some spacing between the password in put box and the other control, or put the other control on the next line.

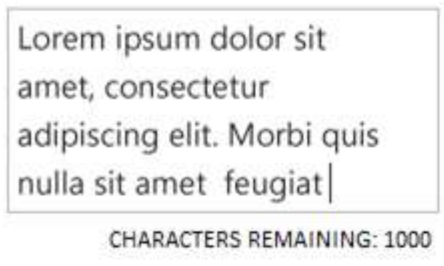
Choosing the right multi-line text input control

When people need to enter or edit long strings, use a multi-line text control. There are two types of multi-line text input control: the plain text input control (the **textarea** element) and the rich text control (an element, such as a **div**, that has its **contenteditable** attribute set to true).

- Use a rich text box if the primary purpose of the multi-line text box is to create documents (such as blog entries or the contents of an email message), and those documents require rich text.
- Use a rich text box if you want people to be able to format their text.
- When you capture text that is only consumed, and not redisplayed later, use a plain text input control. For example, a user completes the survey and the data is sent to a server, but the user doesn't see the data again. It is unnecessary to allow people to style this text.
- For all other scenarios, use a plain text input control.

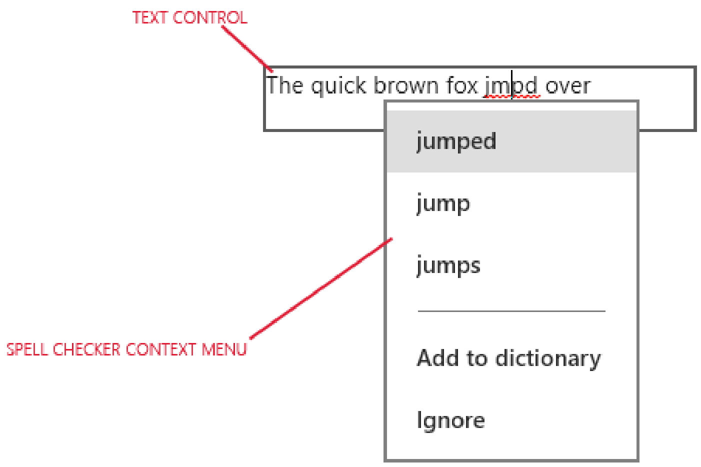
Dos and don'ts for multi-line text input controls

Do	**When you create a rich text box, provide styling buttons and implement their actions.** (Windows Store apps using JavaScript don't automatically provide these controls for you.)
	Use a font that represents the feel of your app.
	Make the height of the text control tall enough to accommodate typical entries.
	When you capture long spans of text where users are expected to keep their word count or character count below some maximum, use a plain text box. Also, provide a live-running counter to show the user how many characters or words they have left

	before they reach the limit. You will need to create the counter yourself. Place it near the text box and update it dynamically as the user enters each character or word.  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi quis nulla sit amet feugiat CHARACTERS REMAINING: 1000
	Don't let your text input controls grow in height while people type.
	Don't use a multi-line text box when people need only a single line.
Don't	Don't use a rich text control if plain text is adequate.

Guidelines for spell checking

Windows Store apps provide a built-in spell checker for multiline and single text input boxes. Here's an example of the built-in spell checker:



Use spell checking with text input controls for these two purposes:

- To auto-correct misspellings

  The spell checking engine automatically corrects misspelled words when it's confident about the correction. For example, the engine automatically changes "teh" to "the."

- To show alternate spellings

  When the spell checking engine is not confident about the corrections, it adds a red line under the misspelled word. It displays the alternates in a context menu when you tap or right-click the word.

Spell checking is turned on by default for multiline text input controls and turned off for single-line controls.

Dos and Don'ts

Do	Use spell checking to help people as they enter words or sentences into text input controls. Spell checking works with touch, mouse, and keyboard input.
Don't	**Don't use spell checking where a word is not likely to be in the dictionary or where people wouldn't value spell checking.** For example, don't turn it on for input boxes of passwords, telephone numbers, or names. (Spell checking is disabled by default for these controls.) Telephone numbers, passwords, and names are rarely in the dictionary, so spell checking them doesn't do any good and might be distracting.

Don’t disable spell checking just because the current spell checking engine doesn't support your app language. When the spell checker doesn't support a language, it doesn't do anything, so there's no harm in leaving the option on. Also, some people might use an Input Method Editor (IME) to enter another language into your app, and that language might be supported. For example, when building a Chinese app, although the spell checking engine doesn’t recognize Chinese now, don’t turn spell checking off. The user may switch to an English IME and type English into the app. Then, if spell checking is enabled, the English gets spell checked.

Guidelines for thumbnails

If you want people to browse files with your app, let them see previews of those files as they browse by displaying thumbnail images. For example, when people use a file picker to browse the file system, Windows use thumbnails to let the user preview all the files in a location as they browse. We also recommend that you use a thumbnail to give the user a preview of a single file to display alongside other, more detailed file information.

Using thumbnails to preview files in this way helps keep the look and feel of your app consistent with Windows Store app design as used in Windows 8.

Appropriate use of thumbnails

- Displaying previews for many items (like files and folders)

  For example, a photo gallery app would use thumbnails to give users a small view of each picture as the users browse their photos.

- Displaying a preview for an individual item (like a file)

  For example, the user may want to see more information about a file, including a larger thumbnail for a better preview, before deciding whether to open the file.

Inappropriate use of thumbnails

- Don't request thumbnails larger than 1024 pixels (on the longest side)

  Thumbnail images should be small; sizes over 1024 pixels are not supported.

User experience guidelines

Get thumbnail images that give users the best previews for the kinds of files they are browsing

Users often want to browse for particular kinds of files—they might want to look through their photos or music. Provide thumbnail images that are optimized for displaying the kinds of files that your user wants to browse. To show thumbnail images for files, specify a thumbnail mode.

This table shows the thumbnail modes that we recommend using for various kinds of files.

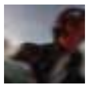
Display previews for	Thumbnail modes	Features of the retrieved thumbnail images
pictures	**picturesView** **videosView**	**Size:** Medium, preferably at least 190 x 130 pixels

videos		**Aspect ratio:** Uniform, wide aspect ratio of about .7 Cropped for higher-quality previews
documents music	**documentsView** **musicView** **listView**	**Size:** Small, preferably at least 40 x 40 pixels **Aspect ratio:** Uniform, square aspect ratio Good for previewing album art because of the square aspect ratio Documents look the same as they look in a file picker window (it uses the same icons)
any single item	**singleItem**	**Size:** Large, at least 256 pixels on the longest side **Aspect ratio:** Variable, uses the original aspect ratio of the file

Tip The features of thumbnail images for different modes might become even more specific in the future. To account for this, we recommend that you specify the thumbnail mode that most closely describes the kinds of files you want to display previews for. For example, if you want to display video files you should use the **videosView** thumbnail mode.

This table shows examples of thumbnail images that you might retrieve for different kinds of items if you specified a particular thumbnail mode.

	Specified thumbnail mode(s)		
Item	• **picturesView** • **videosView**	• **documentsView** • **musicView** • **listView**	• **singleItem**
Picture file	Retrieved thumbnail image:	Retrieved thumbnail image: The thumbnail was cropped to the square aspect ratio. 	Retrieved thumbnail image: The thumbnail image uses the original aspect ratio of the file.

Video file	Retrieved thumbnail image: that has an adornment: The thumbnail has an adornment that differentiates it from pictures. 	Retrieved thumbnail image: The thumbnail was cropped to the square aspect ratio. 	Retrieved thumbnail image: The thumbnail image uses the original aspect ratio of the file.
Music file	Retrieved thumbnail image: The thumbnail is an icon on a background of appropriate size. The app that is associated with the file determines the background color. **Note** If the associated app is a Windows Store app, the app's tile background color is	Retrieved thumbnail image: • If the file has album art, the thumbnail is the album art. • Otherwise, the thumbnail is an icon on a background of appropriate size. The app that is associated with the file determines the background color. **Note** If the associated app is a Windows Store app, the	Retrieved thumbnail image: If the file has album art, the thumbnail is the album art and uses the original aspect ratio of the file. Otherwise, the thumbnail is an icon.

	used.	app's tile background color is used.	
Document file	Retrieved thumbnail image: The thumbnail is an icon on a background of appropriate size. The app that is associated with the file determines the background color. **Note** If the associated app is a Windows Store app, the app's tile background color is used.	Retrieved thumbnail image: The thumbnail is an icon on a background of appropriate size. The app that is associated with the file determines the background color. **Note** If the associated app is a Windows Store app, the app's tile background color is used.	Retrieved thumbnail image: • The document thumbnail, if one exists. Tigers Season Highlights • Otherwise, the thumbnail is an icon.
Folder	Retrieved thumbnail image: • If there is a picture file in the folder, the picture thumbnail is used. • Otherwise, no	No thumbnail image is retrieved.	Retrieved thumbnail image: The thumbnail is an icon that represents a folder.

	thumbnail image is retrieved.		
File group	Retrieved thumbnail image: • If there is a picture file among the files in the group, the picture thumbnail is used. • Otherwise, no thumbnail image is retrieved.	Retrieved thumbnail image: • If there is a file that has album art among the files in the group, the thumbnail is the album art. • Otherwise, no thumbnail image is retrieved.	Retrieved thumbnail image: • If there is a file that has album art among the files in the group, the thumbnail is the album art and uses the original aspect ratio of the file. • Otherwise, the thumbnail is an icon that represents a group of files.

Displaying previews of pictures or videos

- Get thumbnails by using picturesView or videosView thumbnail mode

  Retrieved thumbnail images are cropped to make good previews. Windows tries to preserve the most meaningful part of the picture to use as the thumbnail image. For example, a portrait-orientated photo is cropped at 1/3 of the height in an attempt to preserve people's faces that might be in the picture.

  Size: Medium, preferably at least 190 x 130 pixels.

  Aspect ratio: Uniform, wide aspect ratio of about .7. The uniform aspect ratio is good for aligning thumbnail images in a grid.

- Display multiple items (like files and/or folders)

  Thumbnail images that are retrieved using the **picturesView** or **videosView** thumbnail modes are intended for letting users browse many files. Avoid using them to represent a single, individual file.

- Represent files using previews, only

  Avoid displaying additional file information alongside thumbnails. This lets users browse the files by seeing just the previews (supplied by the thumbnail images) without the additional clutter of unneeded details about each file.

  This lets users browse files in a layout that is similar to the way that files are displayed in a file picker when it is called in thumbnail display mode.

- Differentiate folders and file groups from individual files by superimposing a text label over the thumbnail image

  This text label should be either the name of the folder or the criteria used to form the group of files.

- Display placeholder images while thumbnail images load

  A placeholder image is a generic representation that you should display in place of a thumbnail image while the thumbnail image loads. Using placeholders in this way helps your app seem more responsive because users can interact with items even before the previews load.

  We recommend that a placeholder be:

  - Specific to the kind of item that it stands in for. For example, folders, pictures, and videos should all have their own specialized placeholders that use different icons, text, and/or colors.
  - The same size and aspect ratio as the thumbnail image that it stands in for.

    - Displayed until the thumbnail image is loaded, if a thumbnail can be retrieved.

- If you can't retrieve a thumbnail for an item (like a file, folder, or file group), display a placeholder

  There might be a problem with retrieving the thumbnail for an item. Or, the item (like a folder or a group of files) might not have a thumbnail. In either case, having a placeholder that you can fall back on helps make sure that users can browse smoothly and uninterrupted.

Displaying previews of documents or music files

- Get thumbnails by using musicView, documentsView or listView thumbnail mode
- Display multiple items (like files and/or folders)

  The thumbnails that are retrieved by using one of these modes are intended for creating views that browse multiple files. Avoid using them to represent a single, individual file.

- Represent files by using previews and relevant file information

  In addition to previews of the files, which are supplied by thumbnail images, you should display relevant file information for users while they browse. This lets users identify key information about a file that may not be readily available from a thumbnail image alone. For example, you might display the name of the artist for a music file, which users may not know just by looking at the album art for the file.

  This lets users browse files in a layout that is similar to the layout used by a file picker when it is called in thumbnail display mode.

- Represent folders and file groups by using a placeholder image and by superimposing a text label over the placeholder

  We recommend that you display a placeholder image with superimposed text to differentiate system constructs like folders and file groups from actual files. A visual distinction between these types of items will help make it easier for users who are browsing files with your app.

  We recommend that a placeholder should have:

    - The same size and aspect ratio as the thumbnail image that it stands in for.
    - A text label that is either the name of the folder or the criteria that was used to form the group of files

- Display placeholder images while thumbnail images load

  A placeholder image is a generic representation that you should display in place of a thumbnail image while the thumbnail image loads. Using placeholders in this way helps your app seem more responsive because users can interact with items even before the previews load.

We recommend that a placeholder be:

- Specific to the kind of item that it stands in for. For example, folders, pictures, and videos should all have their own specialized placeholders that use different icons, text, and/or colors.
- The same size and aspect ratio as the thumbnail image that it stands in for.
- Displayed until the thumbnail image is loaded, if a thumbnail can be retrieved.

- If you can't retrieve a thumbnail for an item (like a file, folder, or file group), display a placeholder

  There might be a problem with retrieving the thumbnail for an item. Or, the item (like a folder or a group of files) might not have a thumbnail. In either case, having a placeholder that you can fall back on helps make sure that users can browse smoothly and uninterrupted.

Displaying a preview for a single item

- Get the thumbnail by using singleItem thumbnail mode
- Display a single item (like one file or one folder)

  The thumbnail that is retrieved using this mode is intended for creating a detailed view of a single item. This detail view could be similar to the way that Windows Explorer displays a single file. Avoid using a thumbnail that is retrieved for single item to represent that item in a view that displays multiple items.

- Display a placeholder image while the thumbnail image loads

  A placeholder image is a generic representation that you should display in place of a thumbnail image while the thumbnail image loads. Using a placeholder in this way helps your app seem more responsive because users can interact with the item even before the preview loads.

  We recommend that a placeholder be:

  - Specific to the kind of item that it stands in for. For example, folders, pictures, and videos should all have their own specialized placeholders that use different icons, text, and/or colors.
  - The same size and aspect ratio as the thumbnail image that it stands in for.
  - Displayed until the thumbnail image is loaded, if a thumbnail can be retrieved.

- If you can't retrieve a thumbnail for an item (like a file, folder, or file group), display a placeholder.

There might be a problem with retrieving the thumbnail for an item. Or, the item (like a folder or a group of files) might not have a thumbnail. In either case, having a placeholder that you can fall back on helps make sure that users can browse smoothly and uninterrupted.

Guidelines for Flyouts

Flyouts are great at showing UI that you don't want on the screen all the time. The user can close a Flyout at any time by simply tapping or clicking outside of it, or by pressing ESC. If users are in control of bringing up new UI, they must also be in control of dismissing it. When the user makes a selection in the Flyout, the Flyout should be dismissed.

Do not dismiss a Flyout programmatically unless the user has pressed a command button or selected a menu item in the Flyout. A Flyout should not be dismissed automatically if the user has simply toggled a setting, for instance.

Flyouts are useful in your app for any of a number of reasons. Typical uses of a Flyout are:

- **Collecting information:** If the user selects an action that requires more input, such as choosing an option or typing information, then that UI can be placed in a Flyout to keep the user in their original context. For example, let's say that in a map app, users can label the locations they tag. Users tap the location to tag it, and the app presents a Flyout so that users can enter their label.

  Example: In the browser, to pin an item to the Start screen, the user taps the Pin icon on the app bar. The user then enters the name of the new tile in a Flyout.

- **Warnings and confirmations:** Warn the user before they take a potentially destructive action.

  Example: In a photo app, the user presses a delete icon in the toolbar. Next to the toolbar button, a Flyout appears that warns the user that the photos will be permanently deleted, and provides the delete command. The user can easily press the revealed delete command that appears, or dismiss the Flyout if they pressed the delete icon by accident.

  Note The only warnings or errors that should go in a Flyout are those that can be shown immediately and are a direct result of user action.

- **Drop-down menus:** If a button in an app bar has more than one option, then display a Flyout to let the user pick the option.

  Example: In an email app, the user presses Respond on the app bar, and a menu is displayed to let the user choose among ways to respond: Reply, Reply All, or Forward. If the user presses the Cancel button on the app bar, then a menu is displayed to let the user choose between the ways to Cancel: Discard or Save Draft.

  Note Context menus are meant for contextual actions when selecting text; Flyouts should be used to create drop-down menus from UI elements such as buttons.

- **Displaying more info:** Show more details about an item on the screen that the user is interested in.

Example: In the browser, while browsing InPrivate, the user selects the InPrivate icon. A Flyout then appears to give the user more information about InPrivate mode. Most of the time the browser UI is kept clean, but if requested can provide more detail for users that are interested.

When not to use a Flyout

Avoid Flyouts in the following scenarios:

- If a message, error, warning, or other piece of UI is not invoked directly by the user at that moment, then it should not be a Flyout

  Example: Notifications that updates are available, a trial has expired, or the Internet is not available, should not be displayed in Flyouts.

- If an experience is one that requires prolonged interaction, multiple screens, or lots of UI, then you should integrate the UI into the canvas of the app. For example:
  - The user is working through a wizard with a lot of text entry.
  - The user is changing a long list of settings.
- Avoid using a Flyout for the primary list of commands for your app. Use the app bar for that.
- If a menu is required solely for commands about a text selection, then use a context menu instead. See Guidelines for context menus.

Designing a Flyout

The key to designing a good Flyout is to keep it as simple as possible. Don't include parts of a Flyout that are not necessary for the situation.

Size: The Flyout should be as small as possible given its content. It doesn't need extra padding beyond what the Flyout provides. If a control isn't absolutely necessary, then don't include it. For example, if there are no actions for the user to take, then don't include any buttons. There is no need for Close or OK buttons. Relying on light-dismiss (in which the Flyout disappears when the user touches anywhere on the screen outside of the Flyout) is enough. Similarly, if a title isn't absolutely necessary, then don't include a title.

Position: The Flyout should always be positioned near its point of invocation. If the user tapped on a toolbar button to bring up a Flyout, then the Flyout should show above or below the toolbar button. If showing the Flyout above or below the control would obscure important content, then it can be placed to the left or right of it.

The Flyout is positioned by specifying the object to anchor it to and the side of the object that it should appear on. Flyouts should be center-aligned to their anchor unless the anchor is on the very edge of the screen (such as the user-tile Flyout on the Start screen).

Flyouts should never be positioned in non-contextual places such as the center of the screen, for several reasons:

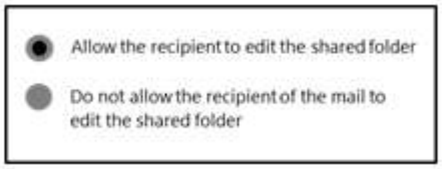
- When UI is shown in a position disconnected from the action that invoked it, the user needs to go searching for this UI and is slowed down. The overall experience is disrupted, creating less pleasant and fluid UI.
- The user may not notice that the Flyout has appeared and may accidentally dismiss it by continuing to tap, making your app feel unresponsive.
- Users expect that centered (or other arbitrarily positioned windows) contain a Close or Cancel button. They would use the buttons even if a light-dismiss option was present, undermining your goal of lightweight UI.

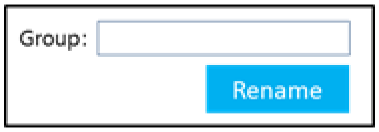
Parts of a Flyout

A Flyout has three components: the title, the main content, and command buttons.

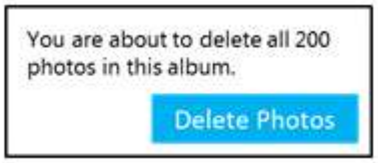
Here are recommendations for when to use each component for each common use of the Flyout.

Collecting information:

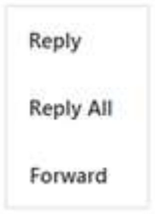
Title	None
Main content	Include just the controls you need. Keep any instructions or "Learn More" links to a minimum. If the user is changing a setting or toggling an on/off switch, for example, then the change should commit as soon as it is made. Interacting with custom content should not dismiss the Flyout; unless there is a command button, the user should be in control of dismissing the Flyout manually.  ● Allow the recipient to edit the shared folder ● Do not allow the recipient of the mail to edit the shared folder
Controls	If a button is meant only to commit the user's changes, then it isn't required and those changes should be committed automatically. If the button begins some action (such as Login, or Save Document), or the user has entered text that they want to commit, then a button is appropriate. In that case, the flyout should be dismissed when the user presses the button. But the user can cancel without committing by light-dismissing the Flyout.

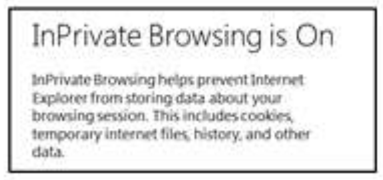
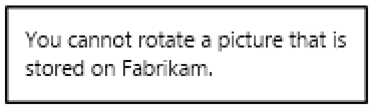
Warnings and confirmations:

Title	None
Main content	State the warning that the user should consider before they take the action. **Do not** phrase it as a question.  You are about to delete all 200 photos in this album.
Controls	Include just the action that the user initiated, such as Delete. Do not include the opposite action or a Cancel button; that can be achieved by dismissing the Flyout.

Menus:

Title	None
Main content	List the menu items that the user can interact with. 
Controls	No buttons are necessary because the user makes a direct selection in the list.

Displaying more info:

Title	Optional title to relate status, or a description of an icon that invoked it.  InPrivate Browsing is On InPrivate Browsing helps prevent Internet Explorer from storing data about your browsing session. This includes cookies, temporary internet files, history, and other data.
Main content	Include the information.  You cannot rotate a picture that is stored on Fabrikam.
Controls	Put optional buttons to do more with the information in the Flyout.

Guidelines for message dialogs

Appropriate use of message dialogs

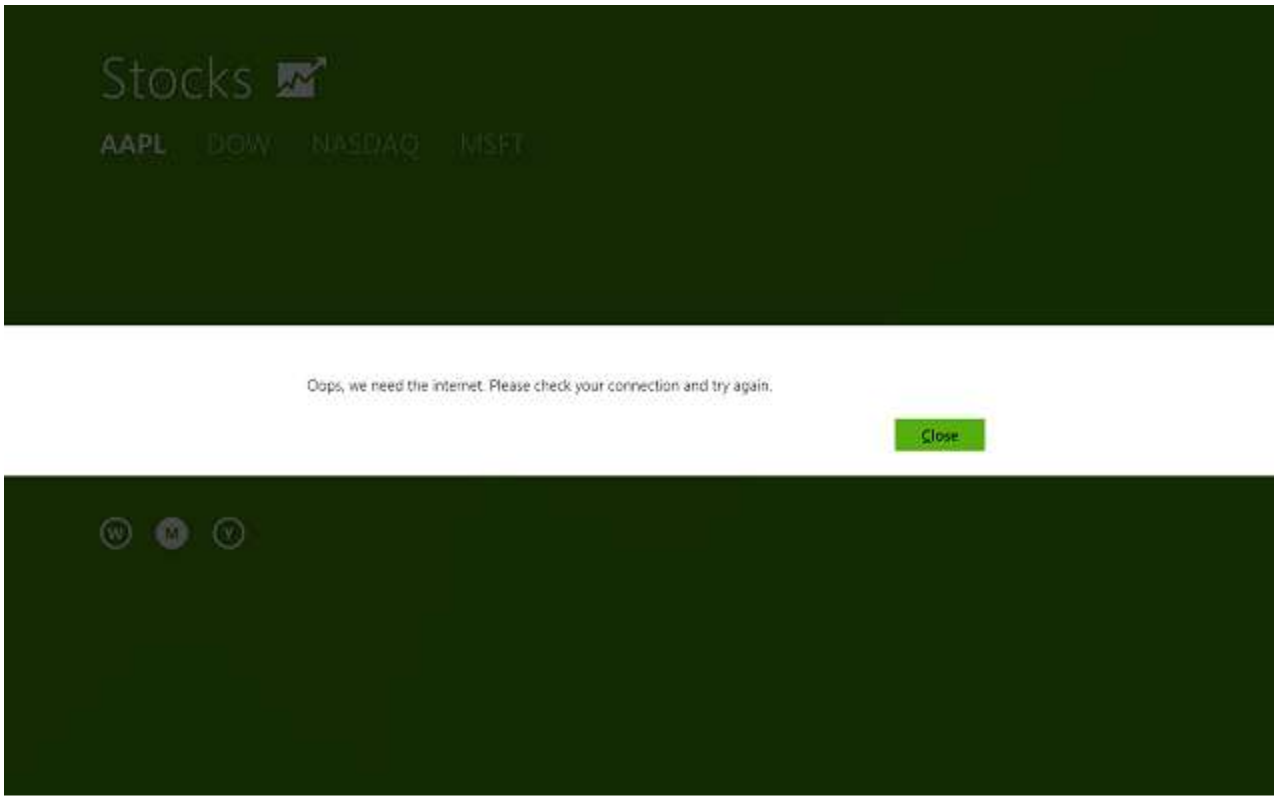
These are scenarios where it is appropriate to take the user out of their immersive experience and present them with a dialog:

Urgent information

Use message dialogs to convey urgent information that the user must see and acknowledge before continuing. An example is, "Your trial period for advanced features has expired."

Errors

Error messages that apply to the overall app context use message dialogs. These are different than error messages that can be conveyed inline. An appropriate example is a message dialog that shows a connectivity error. This critically affects the value that the user can get from the app:



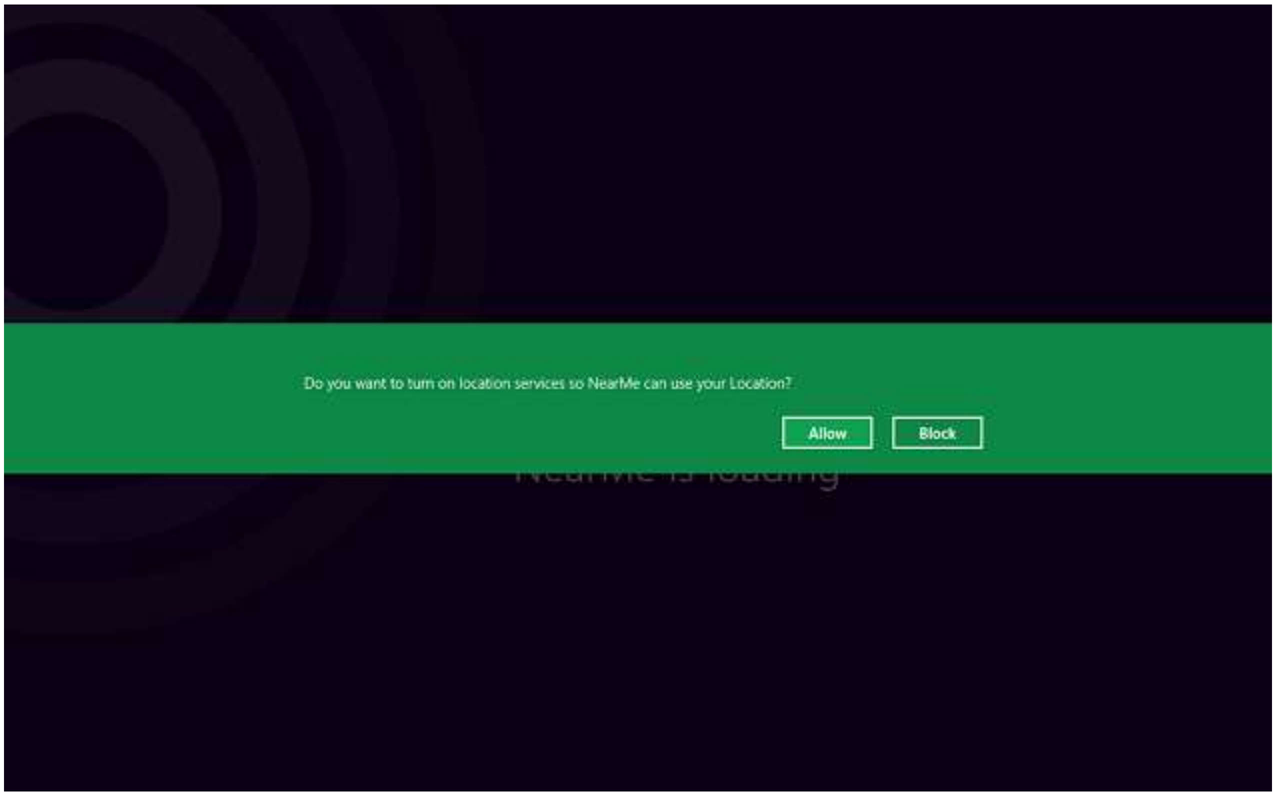
Oops, we need the internet. Please check your connection and try again.

Questions

Use message dialogs to present blocking questions that require the user's input.

A blocking question is a question where the app cannot make a choice on the user's behalf, and cannot continue to fulfill its value proposition to the user. A blocking question should present clear choices to the user. It is not a question that can be ignored or postponed.

Here's an example of a message dialog from the Windows device consent broker. The dialog asks for consent to use location services:


Do you want to turn on location services so NearMe can use your Location?

Inappropriate use of message dialogs

- When the app needs to confirm the user's intention for an action that the user has taken, a Flyout is the appropriate surface.
- For errors that are contextual to a specific place on the page, such as validation errors in password fields, use the app's canvas to show inline errors.

Guidelines for errors

Errors within an app can be communicated to the user through three main surfaces. The app developer chooses the right surface for an error based on the content and consequences of the error.

To show:	Use this surface:
A non-critical error specific to an element in the app. Your app cannot fix the problem, but users can. User interaction: Users can continue to interact with the app, system components, and other apps without dismissing the error. Example: The user enters an invalid string in a text box and then retypes it.	Text inline on the canvas • Text only • Dismissed by app • Appears inline near the source of the error
A non-critical error that applies to the whole app. Your app cannot fix the problem, but users can. User interaction: Users can continue to interact with the app, system components, and other apps without dismissing the error. Example: Mail cannot sync at the moment.	Text at the top of the page • Text only • Dismissed by app • Appears at the top of the page
A significant but non-critical error that applies to the whole app and your app can suggest a solution. User interaction: Users can respond to your prompt or continue to interact with the app, system components, and other apps without dismissing the error.	Error and warning bar • Text, two buttons • Dismissed by user • Appears near the top of the page
A critical error that applies to the whole app and prevents the user from using the app. User interaction: Users cannot continue interacting with the app unless they dismiss the error. Users can still interact with system components and use other apps.	Message dialog • Text, 1 to 3 buttons, title (optional) • Dismissed by user • Appears centered across the app

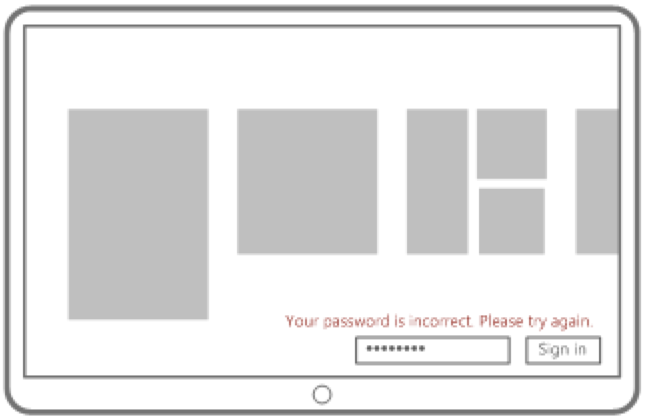
Do not use flyouts, toasts, or custom UI surfaces to display errors.

Errors: Inline text

In general, the inline error is the first choice of surface. An inline text error delivers messages in the context of the user's current actions or the current app page itself. An inline error does not require an explicit user action to dismiss the message. The message goes away automatically when it no longer applies.

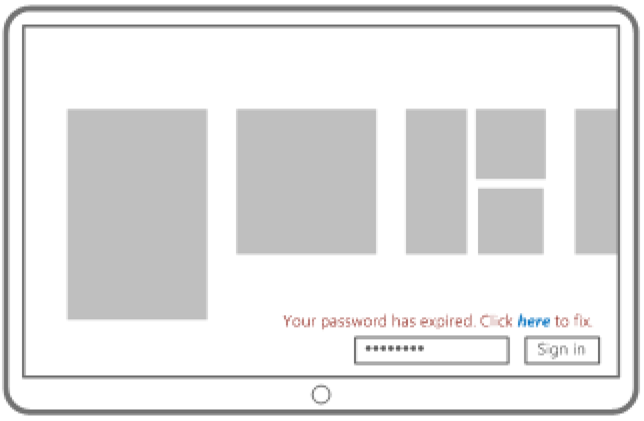
Do	Align the message with the control or element that the message relates to. Lay out the message with ample surround space to increase its focal strength.

The following example shows an inline error message associated with a specific text box.



Don't Include actions or commands in the message.

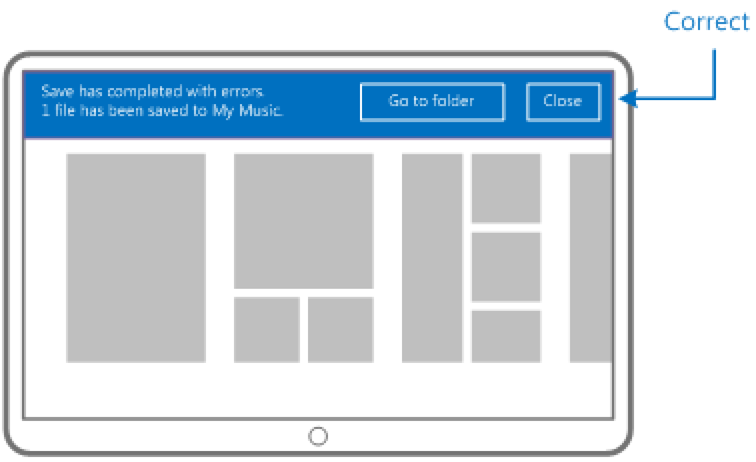
In the following example, an Error and Warning bar would be a better choice.



Errors: Error or warning bar

Use an error or warning bar to notify users of important errors and warnings and to encourage the user to do something. Error messages inform users that a problem occurred, explain why it happened, and provide a solution so users can fix the problem. Warning messages alert a user of a condition that might cause a problem in the future.

Do	Position the bar at the top of the screen, encouraging the user to notice and do something. Color the bar with a color from the app's palette. Use the same color and layout for all your error and warning bars.

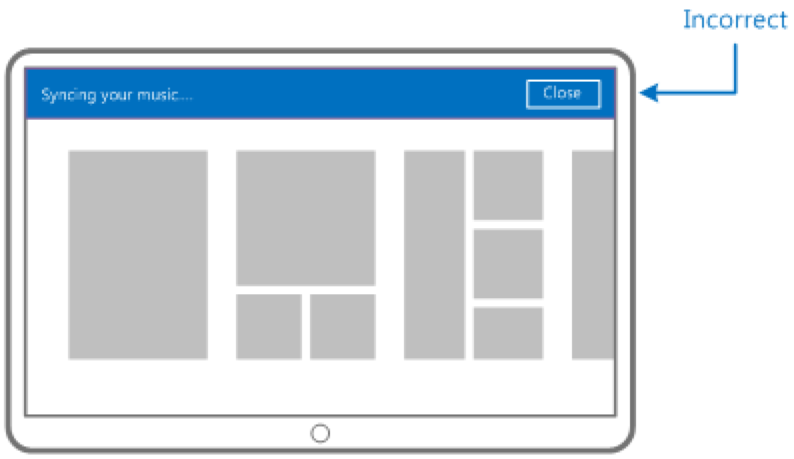

Correct
Save has completed with errors.
1 file has been saved to My Music.
Go to folder
Close

Don't Display bars with different colors or glyphs (such as a shield or exclamation point) based on perceived severity.

Use an 'X' glyph to close the bar; instead, use a labeled Close button.

Use an error and warning bar for information-only message.

The message in the following example is purely informational and no action is required. In this case, an inline message at the top of the screen should have been used.

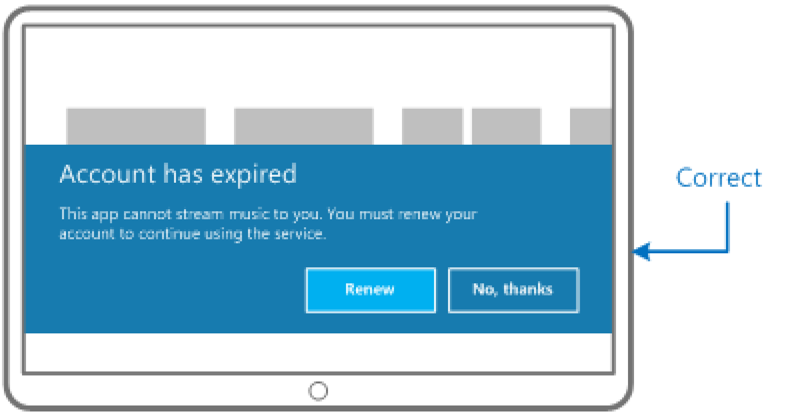


Errors: Message dialogs

Use a message dialog only if a modal message is required, blocking the user from interacting with the app.

Do Use a message dialog if the user must do something before using the app any further.

The following example is an appropriate use of an error message dialog because users cannot use the app unless they have an active account.

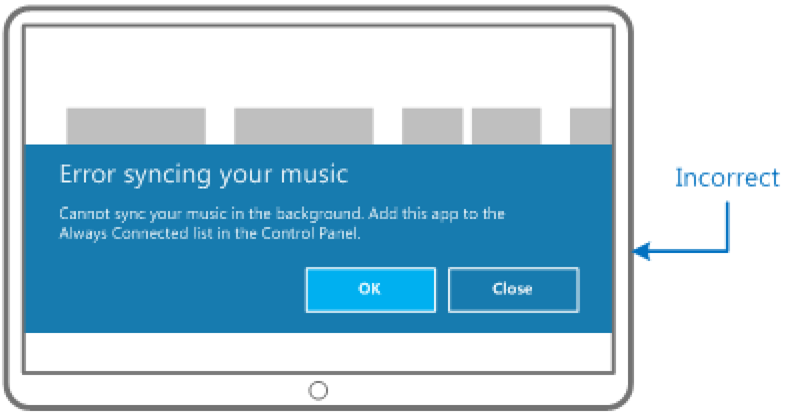


Account has expired

This app cannot stream music to you. You must renew your account to continue using the service.

Don't use a dialog if the user can ignore the message.

In the following example, there is nothing about the error that would require you to block users until they address it. An error or warning bar would have been a better choice.



Error syncing your music

Cannot sync your music in the background. Add this app to the Always Connected list in the Control Panel.

Guidelines for buttons

Is this the right control?

A button lets the user initiate an immediate action, such as submitting a form.

Don't use a button when the action is to navigate to another page; use a link instead. Exception: For wizard navigation, use buttons labeled "Back" and "Next".

Choosing the right type of button

There are three types of button controls: submit, reset, and normal buttons. Follow these guidelines to choose the right button type:

- Submit button

  Use the submit button to send a user input to a server or perform an action, such as a "next" button that saves the form data and goes to the next app page.

- Reset: form reset

  Use the reset button to clear a form or page of user input.

- Normal button: customized action

  Use the normal button to trigger an action.

A button element without any attribute acts as submit button if it is the first button inside a form.

Dos and Don'ts

Do	Use a concise, specific, self-explanatory text that clearly describes the action that the button performs.
	Customize the Normal and Submit buttons with text or images to make it clear to users what happens when they tap or click the button.
	When using AJAX to submit a form, use a submit button and override the form submit function. Then users can commit by pressing the enter key regardless of where the focus is in the form.
Don't	Don't change the Reset button text, unless you need to change it for localization. The default English text for the reset button is "Reset".

	Don't swap the default styles of the submit, reset, and normal buttons.
	Don't put too much content inside a button. Although the button element can contain almost any other HTML elements, such as tables and check boxes, putting too much content inside the button confuses users. Make the content inside a button concise and easy to understand. A button should not contain anything more than a picture and some text.

Guidelines for login controls

Many apps provide a personalized or premiere experience to users when they're logged in. Some apps require the user to log in to a registered account to get value from the app. Other apps provide a rich baseline experience for all users but enable enhanced features when the user is logged in.

Login settings

The Settings charm is the best place for users to find and manage their account settings.

If your app offers a way for users to log in to the app or to create an account, enable users to swipe from the edge and change the login settings in the Settings flyout. This design ensures predictability and ease of access, regardless of where the user might be in the app's workflow. Also, this design frees room on the app's canvas that would otherwise be dedicated to login-related UI.

Login scenarios

Your app may require that the user is logged in, or login may be optional. The login user experience depends on the app.

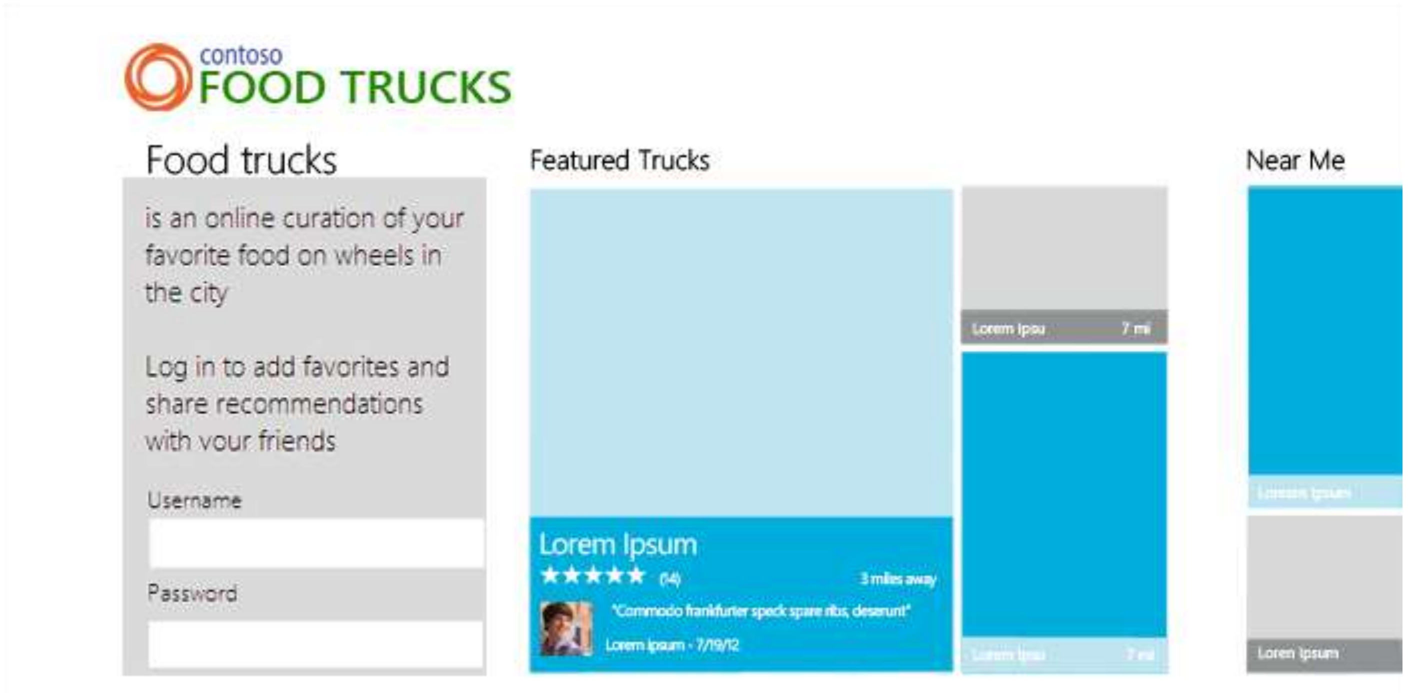
Required login

When your app requires users to log in or create an account when the app first runs, the first screen of the app should feature the login UI prominently. Once the user is logged in, there is no need for an onscreen login UI. Users log out by using the Settings charm.

Recommended login

If your app needs to elevate the login UI to the app's canvas, provide the controls inline in your content. This design ensures that users see the login option on the landing page when they first launch the app, but the login UI doesn’t get in the way of the overall experience.

As users browse the app’s content or views, the logon UI scrolls out of view, but still has a presence in the app. For example, place a login UI as the first section of the ListView control in the app’s landing page. For consistency, the user should always be able to find the login UI in the Settings flyout.

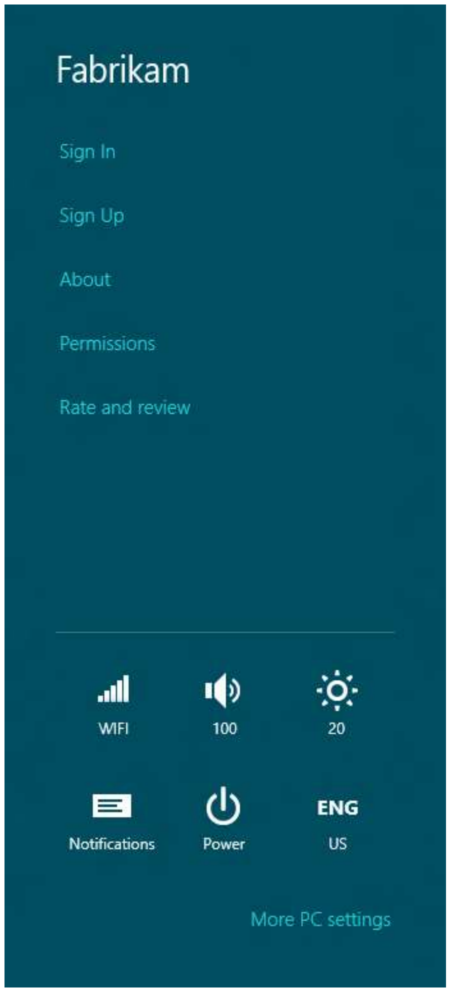
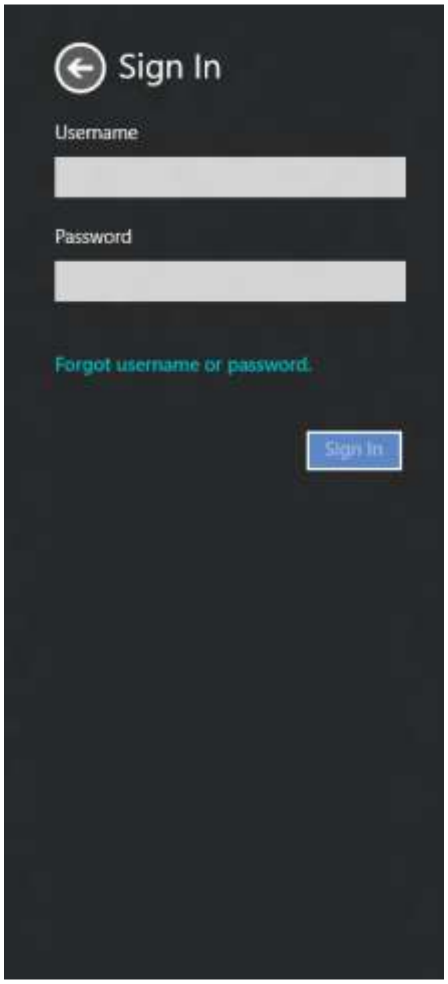

contoso
FOOD TRUCKS
Food trucks
is an online curation of your favorite food on wheels in the city
Log in to add favorites and share recommendations with your friends
Username
Password
Featured Trucks
Lorem Ipsum
(14)
3 miles away
"Commodo frankfurter speck spare ribs, deserunt"
Lorem ipsum - 7/19/12
Lorem Ipsu
7 mi
Near Me
Loren Ipsum

App with login UI hosted as the first section of the ListView control

Optional login

Some apps provide great value without requiring that users are logged in. For example, a news app may offer an initial view of news articles that are interesting to many different readers. User can gain value from the app without logging in.

You can host an optional login UI in the Settings flyout, so the login UI doesn't distract from the content in the app or consume space on the canvas.

	
Settings charm flyout login	Settings flyout login

Contextual login

Sometimes, an app may require a login UI that's specific to some content in the app. For example, if the user wants to post a comment on a news article, the app may require login.

Indicate the need for a user login by putting a contextual login button on the page. The button launches the Settings flyout that hosts the login UI.

Logout UI

Once users have logged into the app, they should have a familiar and reliable place in the system where they can log out of the app, if necessary.

Avoid putting a persistent logout UI on the app's canvas. The Settings flyout for the app is the right place to enable the user to log out. Once the user has logged in to the app, logging out happens rarely, if the app is delivering meaningful personal content.

Personalizing the app on login

When a user logs in to Windows, the Start screen conveys the user's identity and personalization in numerous ways. User preferences for color, layout, apps, and organization of content are personal and customizable.

Once the user has logged in, your app should update its content based on the user's preferences, instead of showing generic content. Each app has a unique way of showing and delivering personal content, which enhances the user's experience in a Windows Store app. When you think about your app's logged in experience, focus on the content that makes your app personal and connected.

There are times when the user who logs in to the app is different than the user who logged in to Windows. For example, a friend who is using someone else's laptop or slate may want to log in to their social network. Avoid putting a persistent UI on the app's canvas that shows identity, because having different identities across the Start screen and the app is more confusing than rewarding for users.

The Settings Accounts flyout is the most intuitive and reliable location for showing user identity in a way that doesn't disrupt the rich, immersive, content-first experience.

Guidelines for app bars

App bars provide the user with easy access to commands when they need them. The user can swipe the bottom edge of the screen to make app bars appear and can interact with their content to make app bars disappear. App bars can also be used to show commands or options that are specific to the user's context, such as photo selection or drawing mode.

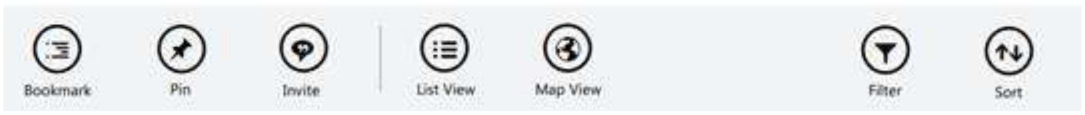
If you have a command that is necessary for a user to complete a workflow (such as buying a product), place those commands on the canvas instead of in app bars.

Guidelines for commands in the app bar

Follow these guidelines when placing your commands on the app bar.

- Do place commands consistently, and organize them into command sets.
    1. Start with your commands on the right.
    2. If you have distinct sets of commands, then divide those sets up on either side of the screen. For example, you can have a set for creating new content and a set for filtering the view.
    3. If you have more than two sets, then put the command sets most like each other on the same side, separated by a separator.
    4. Ensure that commands always show in the same relative position, and on the same side of the screen, whenever they appear inside of your app.
- Do place contextual commands on an app bar, and show that bar programmatically when an item has been selected without changing views.

  Group commands that show on selection (such as Crop/Delete/Pin photo) with Select all/Clear selection (if those buttons exist) on the left. If there are already commands on the left, then they should "bump" those commands over, separated by a separator.

- Do set the app bar's dismissal mode to sticky when displaying contextual commands.

  If you have contextual commands on an app bar, set the mode to sticky while that context exists and turn off the sticky mode when the context is no longer present (such as when a photo is deselected). In sticky mode, the bar does not automatically hide when the user interacts with the app. This is useful for multi-select scenarios or when the context involves interaction such as manipulating cropping handles. The bar stays visible while the user performs the actions. The user can still hide the bar by swiping the top or bottom edge of the screen, and they can show it again with an edge swipe.

- Do use menus when you have too many commands.

  If you are unable to fit all of your commands in an app bar as separate buttons, then group commands together. Place those commands in menus that are opened from app bar buttons.

  Use logical groupings for the commands, such as placing Reply, Reply All, and Forward in a Respond menu.

  Don't create a menu such as "More" or "Advanced" for unrelated, miscellaneous commands. These types of generic commands tend to make an app feel more complicated, and only a small subset of users explore these menus. If you find yourself needing an overflow and there aren't any logical groupings available, consider simplifying your app.

- Do design your app bar for snap and portrait view.

  If you have ten app bar commands or less, your bar will automatically hide labels and adjust padding so that those ten commands still fit in snapped or portrait orientation. If you do not want two rows of commands to appear in snapped view and you have more than five commands, you can group commands together into menus. Or, provide a more focused experience that requires fewer commands on each screen.

  Don't have more than ten commands on your app bar, because that will wrap to two lines for many users.

  Because labels are hidden by default in snapped and portrait view, use icons that are easily identifiable for your commands and provide tooltips for all of your commands.

- Do design for horizontal scrolling.

  The app bar covers scrollbars when an app has a horizontal scrolling area that appears at the bottom of the app. The user may need to click in the app to dismiss the app bar in order to use the scrollbar, or they can use a mouse wheel to scroll.

  If your app bar is in sticky mode, such as when content is selected, reduce the height of your scrolling area so that the scrollbar is flush with the top edge of the app bar.

- Do use the default styles for commands, menus, and flyouts.

  If you want to customize the look of an app bar, we recommend customizing the colors of the background, icons, and labels, but not the size or padding of the buttons. The layout is carefully designed for touch, as well as to fit ten commands at all supported screen widths. If you change the layout you may get undesirable behavior.

- Do use the bottom app bar for commands and the top app bar for navigation.

  Use the bottom app bar for commands that act on the current page.

Use the top app bar for navigational elements that move the user to a different page. Do not put navigation on the bottom bar.

- Don't put critical commands on the app bar.

  Don't place commands that are essential for the user to complete their task on an app bar. For example, do not place the Buy button on the app bar of a store's product page, because it is impossible to complete the core scenario without that button.

  If the user feels that they're opening the bar frequently just to accomplish their task, then consider direct manipulation, Semantic Zoom, or on-canvas commands.

- Don’t put login, logout, or other account management commands in the app bar.

  All account management commands, like login, logout, account settings, or create an account should go in a Settings flyout. If it's critical that the user logs in on a particular page, provide a button in the main app window to allow the user to log in.

- Don't put clipboard commands for text on the app bar.

  Place Cut, Copy, and Paste commands in a context menu rather than in an app bar.

Handling the right mouse button

To keep your app's UI consistent with other Windows Store apps, users must click the right mouse button to trigger the app bar that you provide. If you have an app that must use the right mouse button for another purpose, like secondary fire in a game or a virtual trackball in a 3-D viewer, the app can ignore the events that raise the app bar. But still consider the role of the app bar, or a similar context menu, in your game's control model. It's an important part of the Windows Store app experience.

Follow these guidelines when designing the controls for your app:

- If your app needs to use right mouse button for an important function, use it for that function directly. Don't activate any contextual UI or the app bar if it isn't important to workflow.
- If there are regions of the DirectX surface that don't need app-specific contextual right-click actions, like border menus, show the app bar when the user right-clicks these regions.
- If support for a right mouse button is needed everywhere on the canvas, consider showing the app bar when the user right-clicks the top-most horizontal row of pixels, the bottom-most horizontal row of pixels, or both.
- If none of these solutions suffice, place a custom control on the DirectX surface to enable mouse gestures to open the app bar.

- If your app supports touch controls, remember that a long press, or press-and-hold, is the same as a click on a right mouse button. Handle both events in a similar way.
- Don't provide an alternate behavior for the Win+Z keypress combination in your app. Develop an app bar or similar context menu, and display it when the user presses the Windows key with the Z key.

Guidelines for context menus

The context menu is a lightweight menu that gives users immediate access to actions on text (like clipboard commands) or UI objects in apps. The system provides apps with default context menus for text and hyperlinks. You can replace the default context menus with menus that show custom commands for text or hyperlinks. Or, you can create your own context menus that act on other UI objects (like thumbnails).

Use context menus only to show commands that are directly relevant to users and that cannot be readily accessed through the app toolbar or direct manipulation (like touch rotation).

Appropriate use of context menus

- Showing clipboard commands

  Use a context menu to show clipboard commands (Cut, Copy, and Paste) for objects such as selected text. By default, the system shows Cut, Copy, and Paste commands for selected text. Common paste menu commands are Select All, Paste, and Undo. You can override these commands by showing a customized context menu. Where possible, mimic system behavior in your app by preserving the default commands.

- Showing custom commands

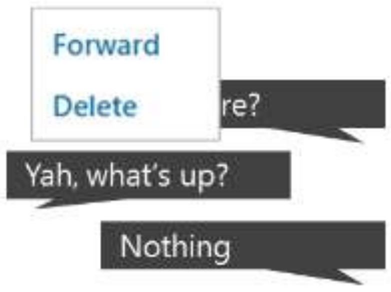
  The system comes with default context menus for text and hyperlinks. Apps can replace these context menus with their own context menus.

  A custom command is appropriate to include in a context menu when it is not found in the app's toolbar and cannot be performed by using direct interaction (like rotation).

- Showing commands that operate on objects that cannot be selected

  Use the context menu to show commands for an object that needs to be acted upon but that cannot be selected or otherwise indicated.

  For example: A chat conversation may not be appropriate to add selection to; in this case a context menu could make commands available for each message in a chat conversation.



Inappropriate use of context menus

- Don't add a command to the context menu when direct manipulation or selection is possible

  Users should execute commands primarily by directly manipulating a UI element or by selecting a UI element and using a command that is on the app bar. This helps ensure that commands are in predictable and discoverable screen locations.

  For example, users should be able to rotate a picture by manipulating the image directly with their fingers instead of using a "Rotate" command in the context menu.

- Don't duplicate commands in the context menu

  If there is already a clear way to accomplish the action like direct manipulation or an existing app bar command , do not add a context menu command for that same action. Instead of relying on duplication, trust that users find commands by selecting an item or navigating to an item to act on it.

  However, there is an exception for some actions that can be executed by keyboard shortcuts. If the action can only be executed by a keyboard shortcut, like CRTL+C to copy, it is okay to duplicate that action by adding a command to the context menu.

- Don't show a context menu for the background of a page or for a large object

  Instead, when you have commands that act on the background of a page or on an object that takes up the whole screen, use the app bar. Or, add a command to your app's canvas to act on the page or object.

User experience guidelines

- Keep command names short

  The context menu has a maximum width that equates to approximately 50 characters. When command names are too long, they are truncated automatically and an ellipse ("...") replaces the missing characters. Avoid truncation of your commands in the context menu by keeping command names short.

- Use sentence capitalization for each command name

  The first character of the command should be uppercase and all of the remaining characters should be lowercase.

- Use a separator to distinguish groups of related commands

  Use separators in the context menus to group sets of commands together. Use a separator to divide app-specific or view-specific commands from a predictable set of commands, such as clipboard commands, that are shown together.

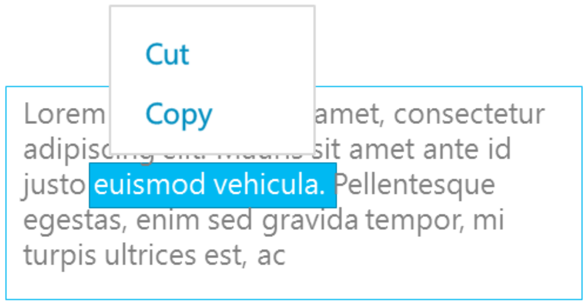
- Don't use item accelerators

  Item accelerators cannot be used with context menu commands. Do not use ampersands (&) at the beginning of command names in your menu.

- Don't show a command in a context menu unless it is contextually relevant to the selection or object

  The context menu does not have a disabled state; the context menu doesn't appear if there are no commands in it.

  For example, if there is no text in the clipboard that can be pasted into text that the user can edit, the default context menu does not show the **Paste** command. Instead, it shows only the **Cut** and **Copy** commands.


Cut
Copy
Lorem amet, consectetur adipisc sit amet ante id justo euismod vehicula. Pellentesque egestas, enim sed gravida tempor, mi turpis ultrices est, ac

- Don't show a command that would result in an error

  For example, do not show the paste command if the paste location cannot be edited.

- Show the fewest number of commands possible, up to the six-command limit

  If you're struggling to fit commands into the menu, ask yourself the following questions:

  - Can this command be represented with direct manipulation?
  - What would your user's experience be like without this command?
  - Is this command accessible in another way? What value, if any, is gained by duplicating the command in the context menu?
  - Does the item have a page that represents it? If so, the command could be in the app bar or on the canvas on that page instead on the context menu in the collection.
  - Does this command always need to be shown, or only in certain contexts?

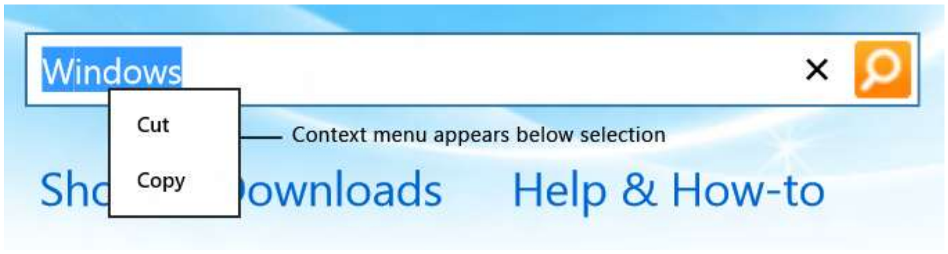
- Order custom commands in the context menu by importance, with the most important commands at the bottom

- Order clipboard commands in the standard Cut, Copy, Paste order and place them at the bottom of the menu

If you want to show only two clipboard commands (for example, Cut and Paste) simply omit the unused command and otherwise preserve the order.

- Position the context menu close to the object your user wants to act on.

  The context menu should be positioned close to the object or selection that it is acting on. When you show the context menu, provide a rectangle for the object that it will act on, and the context menu will position itself near it. By default, a context menu should be positioned below the object that it acts on.



- Dismiss the context menu

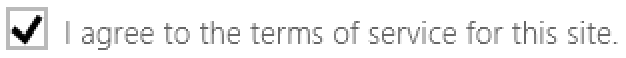
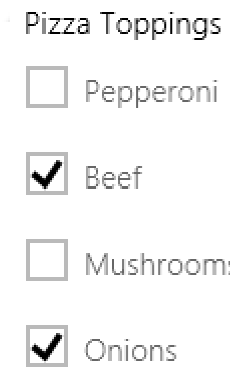
  Programmatically dismiss the context menu when the context it was shown for does not exist anymore. This can be done by using cancel in the standard async pattern.

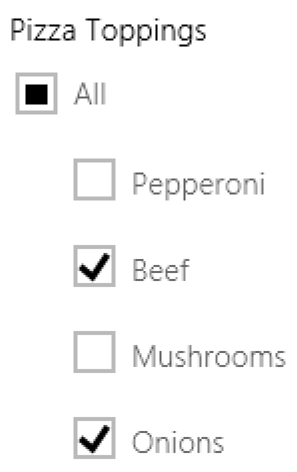
Guidelines for check boxes

A check box is a control that the user can check or uncheck by tapping, clicking, or pressing the space bar on the keyboard. Most check boxes have two states, checked and unchecked, but some check boxes support a third, indeterminate state. Follow these guidelines for adding check boxes to your Windows Store app.

Is this the right control?

Use check boxes to present users with a binary choice, one or more options that are not mutually exclusive, or a mixed choice.

A binary choice Use a single check box for a yes or no choice.	 ✓ I agree to the terms of service for this site.
One or more options that are not mutually exclusive Create a group of check boxes when users can select any combination of options.	 Pizza Toppings ☐ Pepperoni ✓ Beef ☐ Mushrooms ✓ Onions

Mixed choice When an option applies to more than one object, you can use a check box to indicate whether the option applies to all, some, or none of those objects. When the option applies to some, but not all, of those objects, use the check boxes intermediate state to represent a mixed choice. One example of a mixed choice check box is a "Select all" check box that becomes indeterminate when a user selects some, but not all, sub-items.	 Pizza Toppings All Pepperoni Beef Mushrooms Onions

For a binary choice, the main difference between a check box and a toggle switch is that the check box is for status and the toggle switch is for action. You can delay committing a check box interaction (as part of a form submit, for example) while you should immediately commit a toggle switch interaction. Also, only check boxes allow for multi-selection.

When there is more than one option but only one can be selected, use a radio button instead.

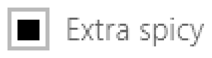
Don't use a check box as an on/off control; use a toggle switch instead.

Dos and don'ts

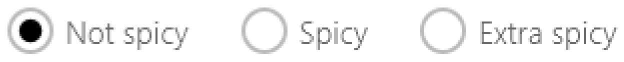
	Enclose the check box within a label with a checkbox so that clicking the label toggles the check box. Doing so increases the size of the selection area and makes the check box more accessible to touch users.
	Use the indeterminate state to indicate that an option is set for some, but not all, child objects.
Do	When using indeterminate state, use subordinate check boxes to show which options are selected and which are not.
Don't	Don't put two check box groups next to each other, or users won't be able to tell which options belong with which group. Use group labels to separate the groups.

Don't use the indeterminate state to represent a third state. The indeterminate state is used to indicate that an option is set for some, but not all, child objects. So, don't allow users to set an indeterminate state directly.

For an example of what not to do, this check box uses the indeterminate state to indicate medium spiciness:



Instead, use a radio button group that has three options: Not spicy, Spicy, and Extra spicy.



Guidelines for DatePickers

These are recommended practices for using Windows Library for JavaScript DatePicker controls.

Practice	Description
Use the DatePicker to display dates on forms or when you need to use space efficiently.	This practice provides the following benefits: • The default inline display of the DatePicker makes it well suited for these situations.
The DatePicker uses style classes on its outer container element and on the parts for the year, month, and day.	This practice provides the following benefits: • You can modify those style rules, or provide more specific selectors and rules to provide custom styling or layouts.
The DatePicker can also be displayed vertically.	This practice provides the following benefits: • This is useful when the width of the app is narrow, for example when it's docked.
Don't set the year range to more than 200 years. Instead, define a useful range of years by using the minYear and maxYear properties.	This ensures that your users don't have to scroll through hundreds of entries.
The DatePicker is not a calendar control. It does not support a pop-up grid style display.	Avoiding this practice ensures that your users do not expect a pop-up calendar.

Guidelines for TimePickers

These are recommended practices for using Windows Library for JavaScript TimePicker controls.

Practice	Description
Use a TimePicker to enable the selection of times when you need to use space efficiently.	This practice provides the following benefits: • The default inline display of the TimePicker makes it well suited for these situations.
If you don't want to display the AM and PM element, hide it by using CSS.	This practice provides the following benefits: • The entire time is still represented internally, but users can't see or change the hidden element.
The TimePicker can be configured to display minute values in 15-minute increments.	This practice provides the following benefits: • Users don't have to scroll through 60 items.
Consider displaying the TimePicker vertically.	This practice provides the following benefits: • This is useful when the width of the app is narrow, for example, when it's docked.
Don't use a TimePicker to display the current time. It's a static display that's meant to be set by the user.	Avoiding this practice ensures that your users don't expect a real-time display.

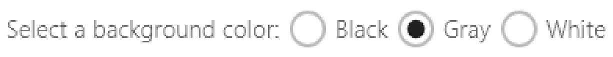
Guidelines for radio buttons

Radio buttons let users select one option from two or more choices. Each option is represented by one radio button. A user can select only one radio button in a radio button group. A radio button has two states: checked and not checked.

Radio buttons are so called because they function like the channel presets on radios. Follow these guidelines for adding **radio buttons** to your Windows Store app.

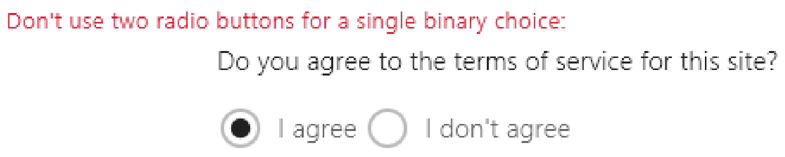
Is this the right control?

Use **radio buttons** to present users with two or more mutually exclusive options, as here.



Select a background color: Black [✓ Gray] White

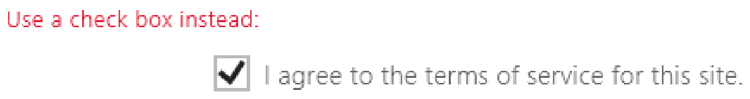
Don’t use radio buttons if there are only two mutually exclusive options that you can combine into a single checkbox. For example, use a checkbox for "I agree" instead of two radio buttons for "I agree" and "I don't agree."



Don't use two radio buttons for a single binary choice:

Do you agree to the terms of service for this site?

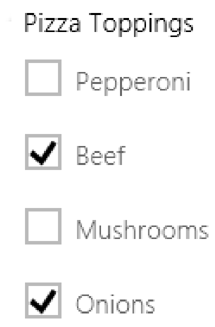
[✓ I agree] I don't agree



Use a check box instead:

[✓ I agree to the terms of service for this site.]

When the user can select multiple options, use a checkbox or select control instead.

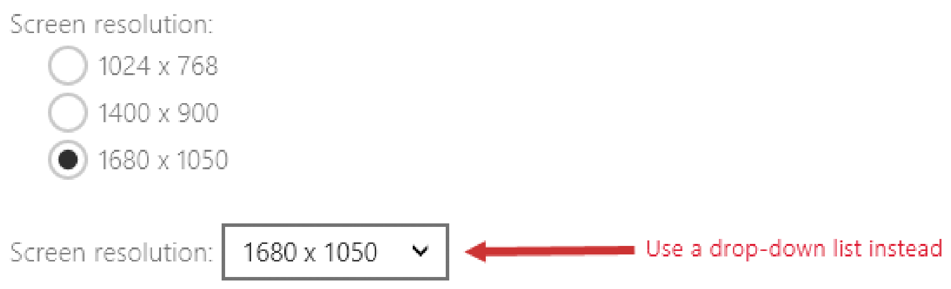


Pizza Toppings

Pepperoni [✓ Beef] Mushrooms [✓ Onions]

Use radio button when you want to draw attention to the selection options by making them all visible. If the default option is recommended for most users in most situations, radio buttons might draw more attention to the options than necessary. If you don't want to call attention to the options or need to save space, use a drop-down list (the select control) instead.

For example, use a drop-down list instead of radio buttons to display available screen resolutions because the user only cares about the current resolution.



Don't use radio buttons when the options are numbers that have fixed steps, like 10, 20, and 30. Use a slider control instead.

Dos and don'ts

	Enclose the radio button in a label element so that tapping the label selects the radio button. Enclosing the radio button in a label makes it more accessible to touch users. Place the label text after the radio button control.
Do	Place the radio button's label text after the radio button, not before or above it.
	Don't put more than eight options in a radio button group. Because the screen space used is proportional to the number of options, keep the number of options in a group between 2 and 7. When you need to present more options, use a select control as a drop-down list or a **ListView** instead.
Don't	**Don't put two radio button groups next to each other.** When two radio button groups are right next to each other, it's difficult to determine which buttons belong to which group. Use group labels to separate them.

Guidelines for the Select control

With the **select** control users can select a value, or multiple values, from a set of items with text labels. The **select** control has two modes of operation: drop-down list mode and list box mode.

Is this the right control?

Use the **select** control to let users select one or more values from a set of items that can be adequately represented using single lines of text.

Don't use the **select** control to display items that contain multiple lines of text or images. Use a **ListView** instead.

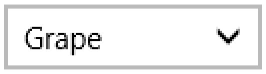
When there are fewer than eight items, consider using radio buttons (if only one item can be selected) or check boxes (if multiple items can be selected) instead.

Choose the right mode

The **select** control has two modes: *Drop-down list mode* and *List box mode*.

When to use the drop-down list mode

In this mode, the select control conserves on-screen space by showing only the currently-selected item.



Users must tap a faceplate, which opens a drop-down, to see other selectable items. Users can select only one item in this mode.

- Use this mode when the selection items are of secondary importance in the flow of your app.

  In some situations, showing all the items by using the list box mode draws more attention to the options than necessary. You can save space and minimize distraction by using the drop-down list mode.

- Use this mode when you need to save space.

  Use the **select** control's drop-down list mode to conserve on-screen real estate. The drop-down list mode lets you provide users with various choices, but uses a small footprint.

When to use the list box mode

In this mode, the **select** control is always open and displays all its items without any additional interaction. This mode supports both single selection and multiple selections.

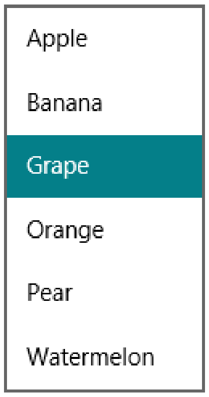
When using this mode, set the **select** control's size so that it is large enough to display all its items without requiring the user to pan or scroll.

- Use this mode when there are fewer than 10 items and the items are important enough to display prominently.
- Use this mode for multi-selection.

  If there are fewer than 10 items and they should be prominently displayed, use the list box mode. If you have more than ten items and want to enable multi-selection, use a **ListView**.

Here's an example of an appropriate use of the **select** control in list box mode:

In this ticket ordering app, there are only a handful of ticket types. They are simple items, but they're also an important aspect of the app itself. By using the inline mode, the selectable options are visible at all times.



Guidelines for sliders

Is this the right control?

Use a slider when you want your users to be able to set defined, contiguous values (such as volume or brightness) or a range of discrete values (such as screen resolution settings).

A slider is a good choice when you know that users think of the value as a relative quantity, not a numeric value. For example, users think about setting their audio volume to low or medium—not about setting the value to 2 or 5.

Don't use a slider for binary settings. Use a **toggle switch** instead.

Here are some additional factors to consider when deciding whether to use a slider:

- **Does the setting seem like a relative quantity?** If not, use **radio** buttons or **select** controls.
- **Is the setting an exact, known numeric value?** If so, use a numeric text box.
- **Would a user benefit from instant feedback on the effect of setting changes?** If so, use a slider. For example, users can choose a color more easily by immediately seeing the effect of changes to hue, saturation, or luminosity values.
- **Does the setting have a range of four or more values?** If not, use **radio buttons**.
- **Can the user change the value?** Sliders are for user interaction. If a user can't ever change the value, use read-only text instead.

If you are deciding between a slider and a numeric text box, use a numeric text box if:

- Screen space is tight.
- The user is likely to prefer using the keyboard.

Use a slider if:

- Users will benefit from instant feedback.

Choosing the right layout: horizontal or vertical

You can layout your slider horizontally or vertically. Use these guidelines to determine which layout to use.

- Use a natural orientation. For example, if the slider represents a real-world value that is normally shown vertically (such as temperature), use a vertical orientation.
- If the control is used to seek within media, like in a video app, use a horizontal orientation.

- When using a slider in page that can be panned in one direction (horizontally or vertically), use a different orientation for the slider than the panning direction. Otherwise, users might swipe the slider and change its value accidentally when they try to pan the page.
- If you're still not sure which orientation to use, use the one that best fits your page layout.

Guidelines for the range direction

The range direction is the direction you move the slider when you slide it from its current value to its **max** value.

- For vertical slider, put the largest value at the top of the slider, regardless of reading direction. For example, for a volume slider, always put the maximum volume setting at the top of the slider. For other types of values (such as days of the week), follow the reading direction of the page.
- For horizontal styles, put the lower value on the left side of the slider for left-to-right page layout, and on the right for right-to-left page layout.
- The one exception to the previous guideline is for media seek bars: always put the lower value on the left side of the slider.

Guidelines for steps and tick marks

- Use **step** points if you don't want the slider to allow arbitrary values between **min** and **max**. For example, if you use a slider to specify the number of movie tickets to buy, don't allow floating point values. Give it a **step** value of 1.
- If you specify **steps** (also known as snap points), make sure that the final **step** aligns to the slider's max value.
- Use tick marks when you want to show users the location of major or significant values. For example, a slider that controls a zoom might have tick marks for 50%, 100%, and 200%.
- Show tick marks when users need to know the approximate value of the setting.
- Show tick marks and a value label when users need to know the exact value of the setting they choose, without interacting with the control. Otherwise, they can use the value tooltip to see the exact value.
- Always show tick marks when **step** points are obvious. For example, if the slider is 200 pixels wide and has 200 snap points, you can hide the tick marks because users won't notice the snapping behavior. But if there are only 10 snap points, show tick marks.

Guidelines for labels

Slider labels

The slider label indicates what the slider is used for.

- Use a label with no ending punctuation.
- Position labels above the slider when the slider is in a form that places its most of its labels above their controls.
- Position labels to the sides when the slider is in a form that places most of its labels to the side of their controls.
- Avoid placing labels below the slider because the user's finger might occlude the label when the user touches the slide.

Range labels

The range, or fill, labels describe the slider's minimum and maximum values.

- Label the two ends of the slider range, unless a vertical orientation makes this unnecessary.
- Use only one word, if possible, for each label.
- Don't use ending punctuation.
- Make sure these labels are descriptive and parallel. Examples: Maximum/Minimum, More/Less, Low/High, Soft/Loud.

Value labels

A value label displays the current value of the slider.

- If you need a value label, display it below the slider.
- Center the text relative to the control and include the units (such as pixels).

Dos and don'ts

Do	**Size the control so that users can easily set the value they want.** For settings with discrete values, make sure the user can easily select any value using the mouse.
	Give immediate feedback while or after a user makes a selection (when practical). For example, the Windows volume control beeps to indicate the selected

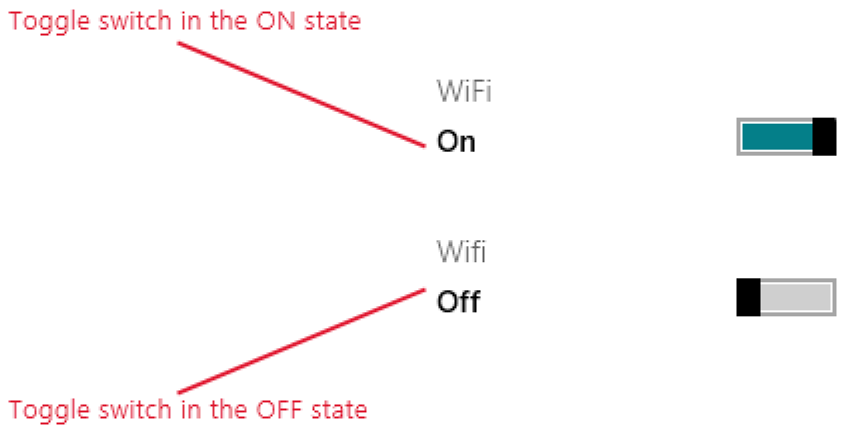
	audio volume.
	Use labels to show the range of values. Exception: If the slider is vertically oriented and the top label is Maximum, High, More, or equivalent, you can omit the other labels because the meaning is clear.
	Disable all associated labels when you disable the slider.
	Don't use a slider as a progress indicator.
	Don't change the size of the slider thumb from the default size.
Don't	Don't create a continuous slider if the range of values is large and users will most likely select one of several representative values from the range. Instead, use those values as the only steps allowed. For example if time value might be up to 1 month but users only need to pick from 1 minute, 1 hour, 1 day, or 1 month, then create a slider with only 4 step points.

Guidelines for toggle switches

The toggle switch mimics a physical switch that allows users to turn things on or off. The control has two states: on and off. Use these guidelines when adding toggle switch controls to your Windows Store app.

Is this the right control?

Use a toggle switch for binary operations that become effective immediately after the user changes it. For example, use a toggle switch to turn services or hardware components on or off.



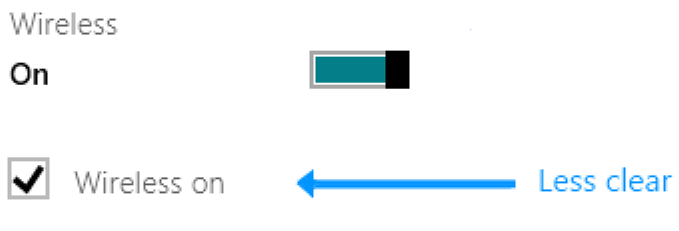
A good way to test whether you should use toggle switch is to think about whether you would use a physical switch to perform the action in your context.

After the user toggles the switch on or off, you perform the corresponding action immediately.

Choosing between toggle switch and check box

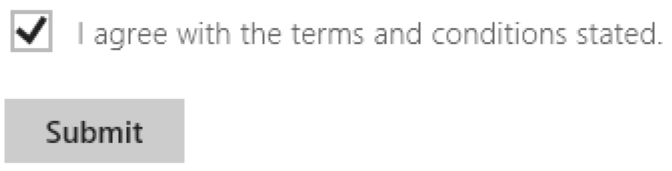
In some cases, you could use either a toggle switch or check box. Follow these guidelines to choose between the two.

- Use a toggle switch for binary settings when changes become effective immediately after the user changes them.

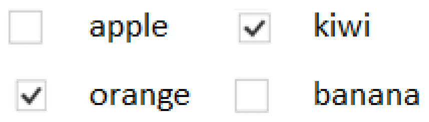


It's clear in the toggle switch case that the wireless is set to on. But in the checkbox case, users need to think about whether the wireless is on now or whether they need to check the box to turn wireless on.

- Use a checkbox when the user has to perform extra steps for changes to be effective. For example, if the user must click a "submit" or "next" button to apply changes, use a check box.



- Use check boxes or a **ListView** when the user can select multiple items:



Dos and Don'ts

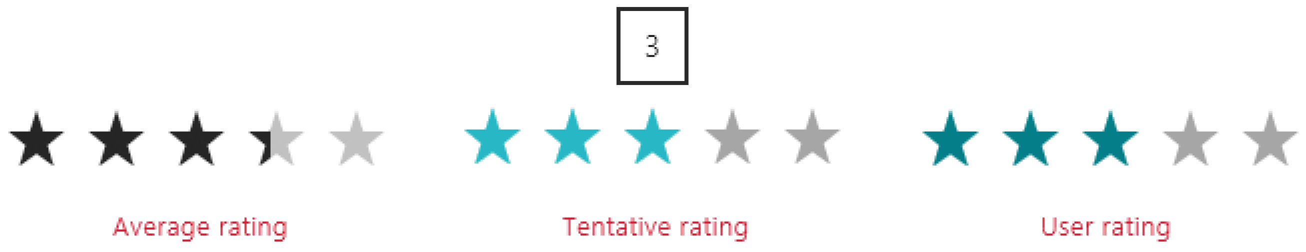
Do	**Replace the On and Off labels when there are more specific labels for the setting.** If there are short (3-4 characters) labels that represent binary opposites that are more appropriate for a particular setting, use them. For example, you might use "Show/Hide" if the setting is "Show images." Using more specific labels can help when localizing the UI.
	Don't replace the On or Off label unless you must. Use the default labels unless there are labels that are more specific for the setting.
Don't	Don't use labels longer than 3 or 4 characters.

Guidelines for the Rating control

The **Rating** control lets users rate something by clicking an icon that represents a rating. It can display 3 types of ratings: an average rating, a tentative rating, and the user's rating. Use these guidelines when adding **Rating** controls to your Windows Store app.

Is this the right control?

The **Rating** control lets users rate something by clicking an icon that represents a rating. It can display three types of ratings: an average rating, a tentative rating, and the user's rating.



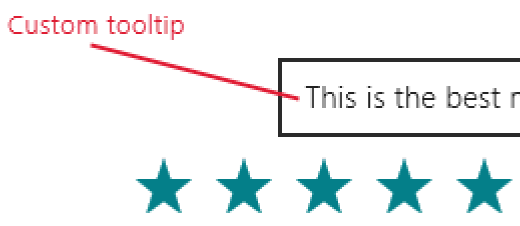
Use the rating control to show the degree to which users like, appreciate, or are satisfied with an item or service. For example, use the rating control to let users rate a movie. Don't use it for other types of data that have a continuous range, such as screen brightness (use a slider for that).

Don't use the rating control as a filter control. For example, if you want to let users filter search results to show restaurants with five stars, use a slider. Using the ratings control could mislead users into thinking that they are giving a new rating to the restaurant.

Don't use a one-star rating control as a like/dislike control. Use a check box instead. The rating control is not designed for a binary rating, for example, tapping on the control doesn't toggle the star on and off.

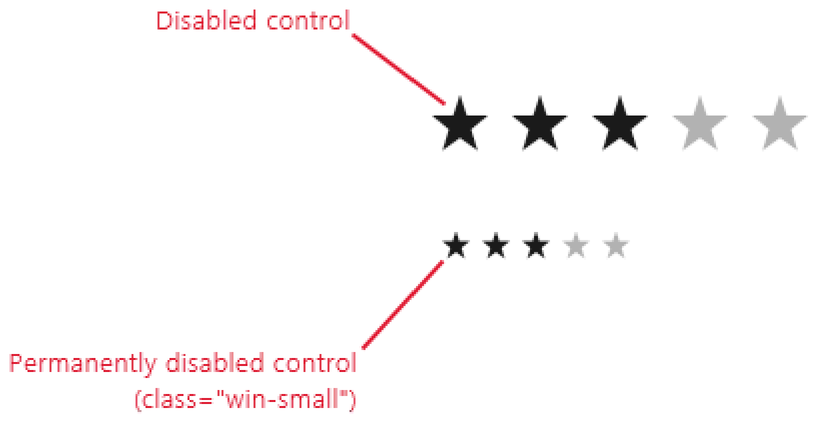
Dos and don'ts

Do	**Use tooltips to give users more context.** You can customize the tooltip to show more meaningful words for each star, like "excellent, "very good," or "not bad," as shown here:



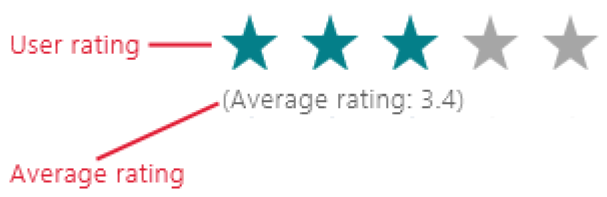
Disable the rating control when you want to prevent the user from adding or modifying the rating. A disabled rating control continues to display the rating (if one is set), but doesn't allow the user to add or modify it. If you want to restrict ratings to logged-in users, you can disable the rating control. When users tap the control, you can send them to a log-in page.

If the control cannot be enabled (it is read-only for the life of the app), make it smaller than other rating controls by setting the **class** attribute of the control host element to "win-small". Making the control smaller helps distinguish it from the other controls and also discourages interaction.



Show the average rating and user rating at the same time. After the user provides a rating, the rating control displays the user's rating instead of the average rating. Whenever it's meaningful to your users, show them the average user rating in addition to their rating. Here are two ways you can display average rating:

- Display the average in an accompanying text string (such as "average: 3.5").
- Use two rating controls together, one to show the user rating, and one that shows the average rating and doesn't allow user input.

	
	Don't change the default number of stars (the max rating) unless you must. By default, the rating control has 5 stars, with 1 being the lowest or worst rating and 5 being the highest or best. If your app follows this convention, it will be easy for users to interpret what the rating means.
Don't	Don't disable the "Clear your rating" feature unless you must prevent users from deleting their ratings.

Guidelines for progress controls

Follow these guidelines for adding **progress** controls to your Windows Store app.

Is this the right control?

The **progress** control shows users the progress of an operation that takes more than 2 seconds to complete. A progress control can show an approximate percentage of completion (determinate progress), or indicate that an operation is ongoing (indeterminate progress).

It's not always necessary to show a progress control; sometimes a task's progress is obvious enough on its own or the task completes so quickly that showing a progress control would be distracting. Here are some points to consider when you determine whether you should show a progress control.

- Does the operation take more than two seconds to complete?

  If so, show a progress control as soon as the operation starts. If an operation takes more than two seconds to complete most of the time, but sometimes completes in under two seconds, wait 500ms before you show the control. This avoids flickering.

- Is the operation waiting for the user to complete a task?

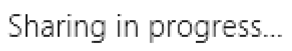
  If so, don't use a progress bar. Progress bars are for computer progress, not user progress.

- Does the user need to know that something is happening?

  For example, if the app is downloading something in the background, and user didn't initiate the download, the user doesn't need to know about it.

- Is the operation a background activity that doesn't block user activity and is of minimal (but still some) interest to the user?

  Use text and ellipses when your app is performing tasks that don't have to be visible all the time, but you still need to show the status.


Sharing in progress...

  Use the ellipses to indicate that the task is ongoing. If there are multiple tasks or items, you can indicate the number of remaining tasks. When all tasks complete, dismiss the indicator.

- Can you use the content from the operation to visualize progress?

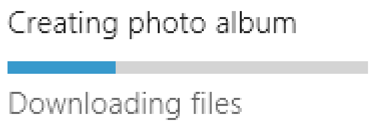
If so, don't show progress control. For example, when displaying images loaded from the disk, images appear on the screen one-by-one as they are loaded. Displaying progress control would provide no benefit. It would clutter the UI.

Don't use the "wait cursor" to indicate activity. Users who use touch to interact with the system won't see it. Users who use the mouse don't need two ways to visualize activity (the cursor and the progress control).

Choosing the right type of progress control style

There are 3 progress control styles:

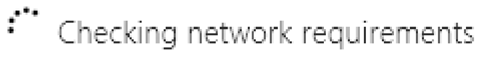
- The determinate progress bar style



Use the determinate progress bar style when a task is determinate, that is when it has a well-defined duration or a predictable end. Here are some examples of determinate tasks:

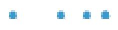
  - The app is downloading a 500k photo and has received 100k so far.
  - The app is displaying a 15 second advertisement and 2 seconds have elapsed. If the task is not determinate, use the indeterminate progress bar or ring.

- The indeterminate progress ring style



Use this style for tasks that are not determinate and are modal (block user interaction).

- The indeterminate progress bar style



Use this style for tasks that are not determinate that are non-modal (don't block user interaction).

Treat partially modal tasks as non-modal. Some tasks block interaction until some progress has been made, and then user can start interacting with the app again. For example, when the user performs a search query, interaction is blocked until the first result is displayed. Treat tasks such as these as non-modal and use the indeterminate progress bar style if modal state lasts less than 2 seconds. If modal state can last more than 2 seconds, use indeterminate progress ring for the modal phase of the task, and use the indeterminate progress bar for the non-modal phase.

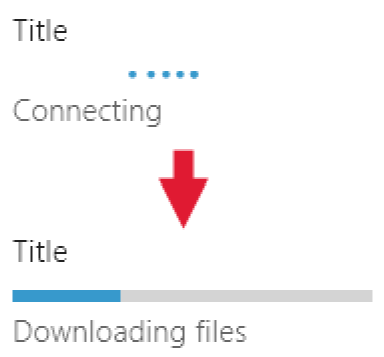
General guidelines

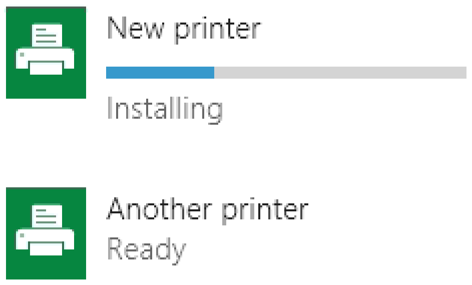
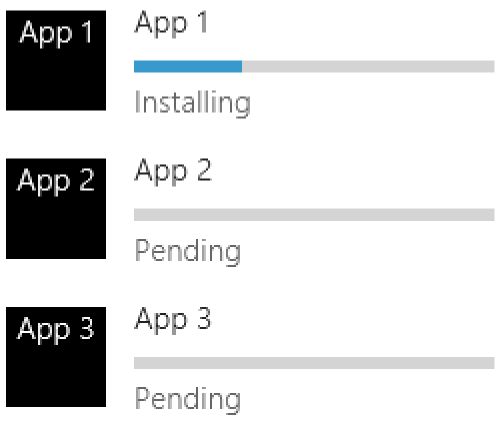
Here are some general guidelines to follow, regardless of which progress control style you use.

- **Show a single progress control for multiple active related tasks.** If there are multiple related items on the screen that are all simultaneously performing some kind of activity, don't show multiple progress controls. Instead, show one that ends when the last task completes. For example, if the app downloads multiple photos, show a single progress control, instead of showing one for every photo.
- Don't change the location or size of the progress control while the task is running.

Guidelines for determinate tasks

If you can estimate remaining amount of work in time, bytes, files, or some other quantifiable units of measure, use a determinate progress bar. Here are some guidelines for using the determinate progress bar.

Switching from indeterminate to determinate If some time (or action) is needed to start providing determinate progress, use the indeterminate bar first, and then switch to the determinate bar. For example, if the first step of a download task is connecting to a server, you can't estimate how long that takes. After the connection is established, switch to the determinate progress bar to show the download progress. Keep the progress bar in the same place and with the same size after the switch.	

Showing progress and status inline Suppose you have a list of items, such as a list of printers, and certain actions can initiate an operation on items in that list (such as installing a driver for one of the printers). When this happens and the operation is determinate, show a determinate progress bar next to the item. Show the subject (label) of the task above the progress bar and status underneath. Don't provide status text if what's happening is obvious. After the task completes, hide the progress bar. Use the status text to communicate the new state of an item.	
Showing multiple operations When you want to show a list of tasks, align the content in a grid so users can see the status at a glance. Show progress bars for all items, even those that are pending. Because the purpose of this list is to show ongoing operations, remove operations from the list when they complete.	 App 1 App 1 Installing App 2 App 2 Pending App 3 App 3 Pending
Showing app-modal determinate progress in the app bar If a user initiated a task from the app bar and it blocks user interaction, show the progress control in the app bar. If it's clear what the progress bar is showing progress for, you can align progress bar to the top of the app bar and omit the label and status; otherwise, provide a label and status	

text. Disable interaction during the task by disabling controls in the app bar and ignoring input in the content area.	

Dos and don'ts for determinate tasks

	If the operation is modal (blocks user interaction), and takes longer than 10 seconds, provide a way to cancel it.
	Space progress updates evenly. Avoid situations where progress increases to over 80% and then stops for a long period. You want to speed up progress towards the end, not slow it down. Avoid drastic jumps, such as from 0% to 90%.
	After setting progress to 100%, wait until the determinate progress bar finishes animating before hiding it.
	If your task is stopped (by a user or an external condition), but a user can resume it, visually indicate that progress is paused by using the win-paused CSS style. Provide status text under the progress bar that tells the user what's going on.
Do	If the task is stopped and can’t be resumed, or it must be restarted from scratch, visually indicate that there's an error by using the win-error CSS style. Replace the status text (underneath the bar) with a message that tells the user what happened and how to fix the issue (if possible).

	Don’t decrement progress. Always increment the progress value. If you need to reverse an action, show the progress of reversal as you would show progress of any other action.
Don't	**Don’t restart progress (from 100% to 0%), unless it’s obvious to the user that a current step or task is not the last one.** For example, suppose a task has two parts: downloading some data, and then processing and displaying the data. After the

	download is complete, reset the progress bar to 0% and begin showing the data processing progress. If it's unclear to users that there are multiple steps in a task, collapse the tasks into a single 0-100% scale and update status text as you move from one task to the next.

Guidelines for indeterminate tasks

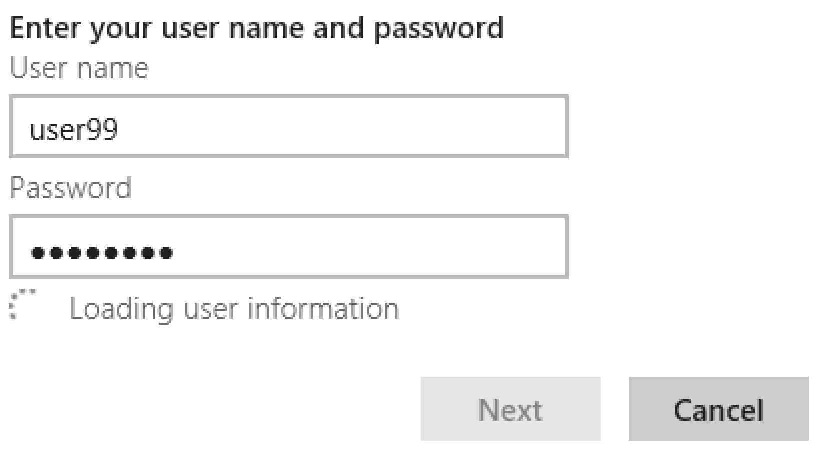
If the task is modal—it blocks interaction until its completion—use the indeterminate progress ring style. If the task is not modal, use the indeterminate progress bar style.

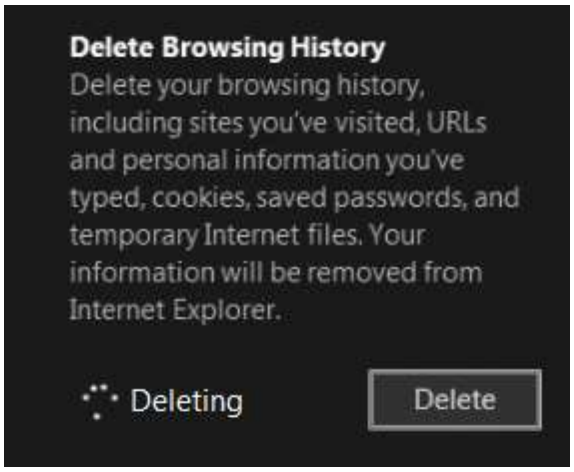
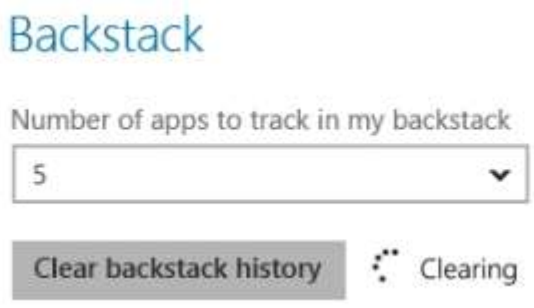
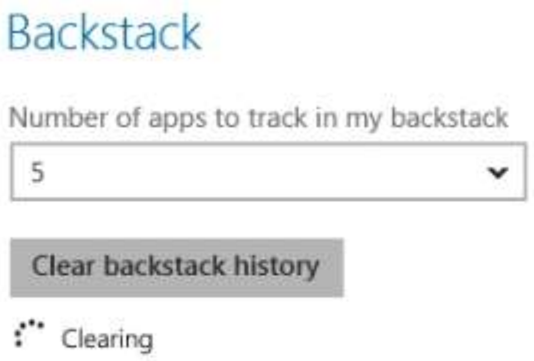
Indeterminate progress ring

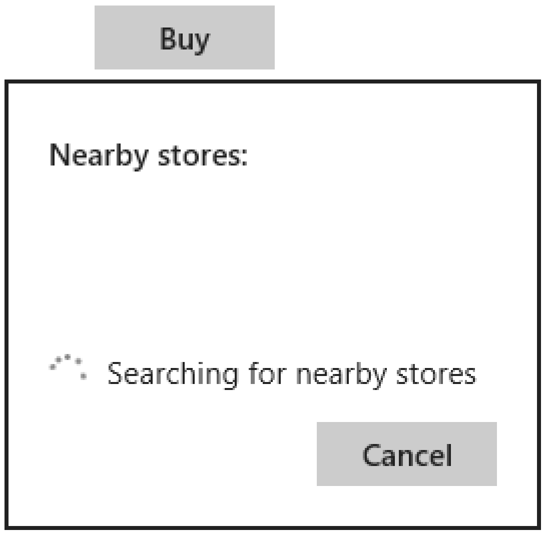
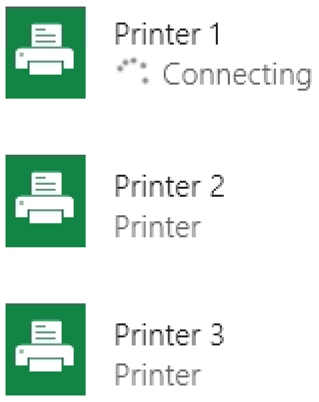
Follow these guidelines for displaying the progress ring:

- Display the progress ring in the context of the action: show it near the location where the user initiated the action or where the resulting data will display.
- Provide status text to the right of the progress ring.
- Make the progress ring the same color as its status text.
- Disable controls that user shouldn't interact with while the task is running.
- If the task results in an error, hide the progress indicator and status text and display an error message in their place.

Here are some guidelines for specific situations that involve the progress ring.

In a dialog, an action occurs before you move to the next screen Place the progress ring just above the button area, left-aligned with the content of the dialog.	 Enter your user name and password User name user99 Password ●●●●●●●●● Loading user information

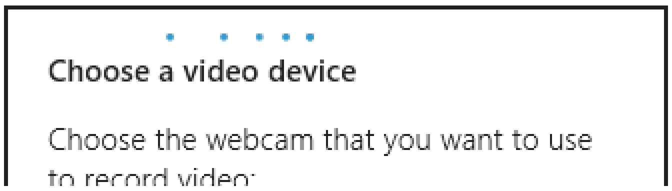
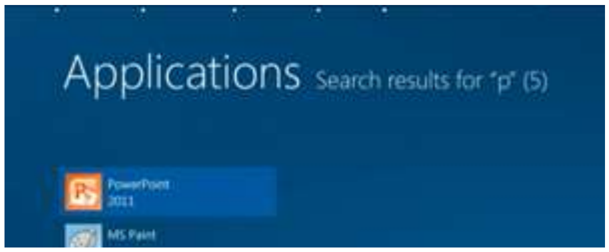
Showing progress in an app window with right-aligned controls Place the progress ring to the left or just above the control that caused the action. Left-align the progress ring with related content.	 **Delete Browsing History** Delete your browsing history, including sites you've visited, URLs and personal information you've typed, cookies, saved passwords, and temporary Internet files. Your information will be removed from Internet Explorer.
Showing progress in an app window with left-aligned controls If a control that starts the modal action is aligned to the left, place the progress ring to the right of that control. Or you can place it underneath the control.	 Backstack Number of apps to track in my backstack - or -  Backstack Number of apps to track in my backstack

Showing progress in a flyout Use a flyout if activity can proceed in the background when the user dismisses the flyout by tapping outside it.	 Buy Nearby stores: Searching for nearby stores Cancel
Showing multiple items Place the progress ring and status text underneath the title of the item. If an error occurs, replace the progress ring and status with error text.	 Printer 1 Connecting Printer 2 Printer Printer 3 Printer

Indeterminate progress bar

Use indeterminate progress bar for tasks that don't block user interaction (non-modal).

Here are some guidelines for specific indeterminate progress bar situations.

Showing progress in a flyout Place the indeterminate progress bar at the top of the flyout and set its width so that it spans the entire flyout. This placement minimizes distraction but still communicates ongoing activity. Don't give the flyout a title, because a title prevents you from placing the progress bar at the top of the flyout.	 Choose a video device Choose the webcam that you want to use to record video:
In an app window Place the indeterminate progress bar at the top of the app window, spanning the entire window.	 Applications Search results for "p" (5)

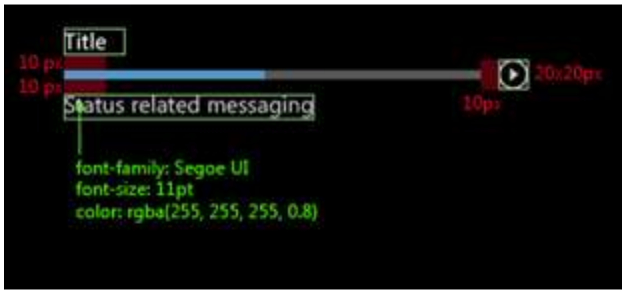
Guidelines for status text

- When you use the determinate progress bar, don't show the progress percentage in the status text. The control already provides that info.
- If you use text to indicate activity without a progress control, use ellipsis to convey that the activity is ongoing.
- If you use a progress control, don't use ellipsis in your status text, because the progress control already indicates that the operation is ongoing.

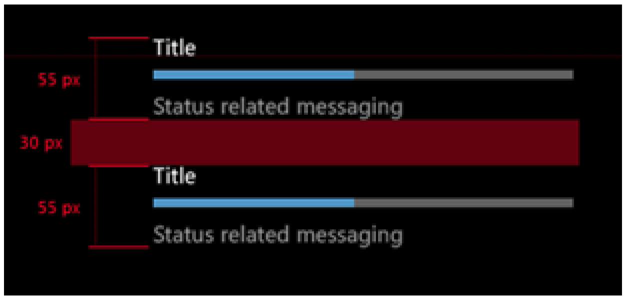
Layout patterns

Here are layout guidelines for several common patterns of progress control usage.

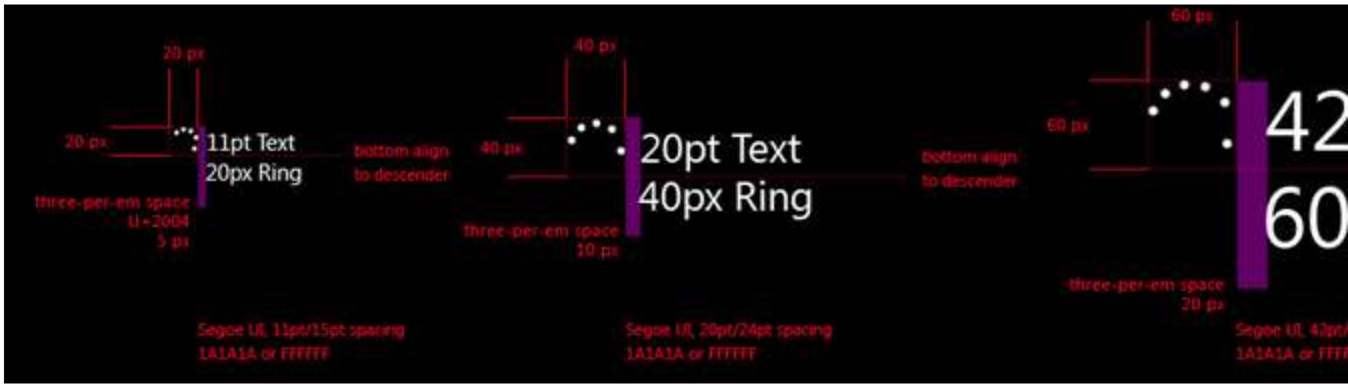
- Determinate progress bar with label and status



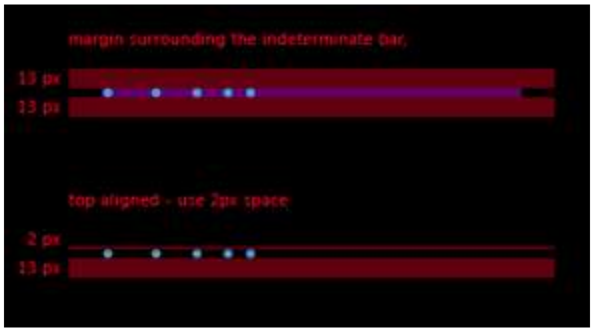
- Multiple progress bars



- Indeterminate progress ring with status text



- Indeterminate progress bar


margin surrounding the indeterminate bar,
13 px
13 px
top aligned - use 2px space
2 px
13 px

Guidelines for tooltips

Is this right control?

A tooltip is a short description that is linked to another control or object. Tooltips help users understand unfamiliar objects that aren't described directly in the UI. They display automatically when the user presses and holds or hovers the mouse pointer over a control. The tooltip disappears when the user moves the finger, the mouse pointer, or a pen pointer.

Use a tooltip to reveal more info about a control before asking the user to do something. You can also use a tooltip to show the item under the finger during touchdown, so that users know where they are touching. (You should try to find other ways to disambiguate first, such as use a larger control, more spacing, or styling the control's active/hover state.)

When should you use a tooltip? To decide, consider these questions:

- Is the info displayed based on pointer hover?

  If not, use another control. Display tips only as the result of user interaction—never display them on their own.

- Does a control have a text label?

  If not, use a tooltip to provide the label. It is a good programming practice to label most controls and for these you don't need tooltips. Toolbar controls and command buttons with graphic labels need tooltips.

- Does an object benefit from a more detailed description or further info?

  If so, use a tooltip. But the text must be supplemental—that is, not essential to the primary tasks. If it is essential, put it directly in the UI so that users don't have to discover or hunt for it.

- Is the supplemental info an error, warning, or status?

  If so, use another UI element, such as a flyout.

- Do users need to interact with the tip?

  If so, use another control. Users can't interact with tips because moving the mouse makes them disappear.

- Do users need to print the supplemental info?

  If so, use another control.

- Will users find the tips annoying or distracting?

  If so, consider using another solution—including doing nothing at all. If you do use tips where they might be distracting, allow users to turn them off.

Here are some examples of good ways to use tooltips:

- Showing the day of the week when users touch a date in a calendar.
- Showing a preview of the linked website when users touch a hyperlink.

Dos and don'ts

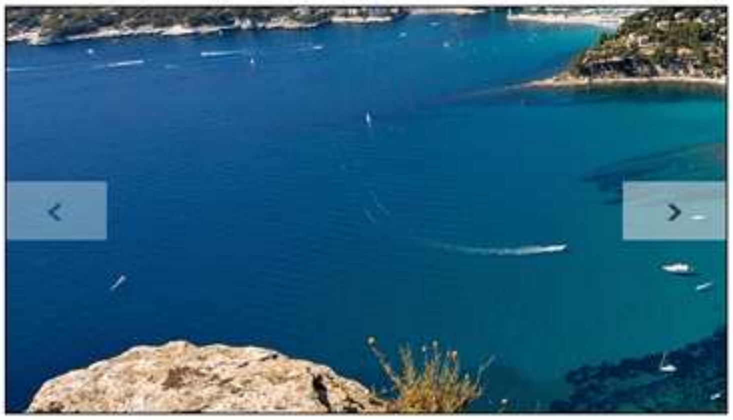
Do	**Keep the tooltip text concise.** Tooltips are perfect for short sentences and sentence fragments. Large blocks of text are difficult to read and overwhelming.
	Create helpful, supplemental tooltip text. Tooltip text must be informative. Don't make it obvious or just repeat what is already on the screen. Because tooltip text isn't always visible, it should be supplemental info that users don't have to read. Communicate important info by using self-explanatory control labels or in-place supplemental text.
	Use images when appropriate. Sometimes it's better to use an image in a tooltip. For example, when the user touches a hyperlink, you can use a tooltip to show a preview of the linked page.
Don't	**Don't use a tooltip to display text already visible in the UI.** For example, don't put a tooltip on a button that shows the same text of the button unless touching the button blocks its text.
	Don't put interactive controls inside the tooltip.
	Don't put images that look like they are interactive inside the tooltip.

Guidelines for FlipView controls

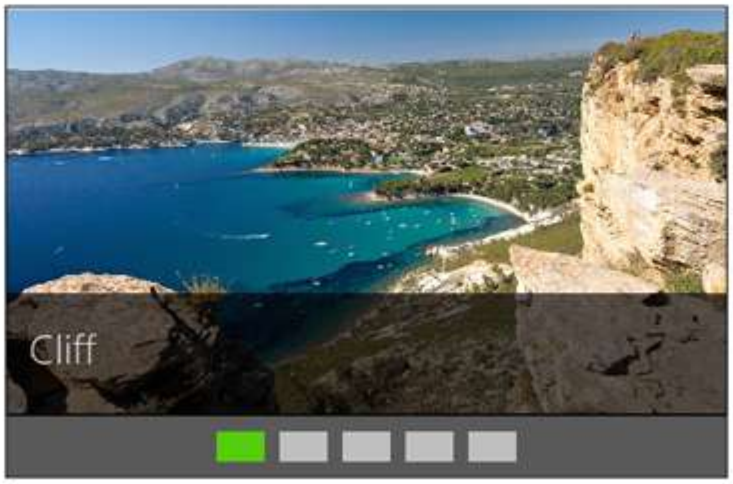
The **FlipView** control lets people flip through views in an app or through a collection of items, like photos in a photo album, one at a time. It offers people a way to look at each individual item while navigating through the collection.

Is this the right control?

The **FlipView** control lets people flip through views or items one at a time. Flip buttons appear on mouse hover and let people flip to the next or previous item.



You can also add a context indicator control so users can jump directly to a particular item.



Use a **FlipView** control to:

- Flip between items in a small collection (< 10) of related items

For example, in a shopping app, you might want to show several images of a product in a product details page. You can use the context indicator control for a small collection, but this is often not necessary.

- Flip between items in a medium-sized list (10 to 25) of related items

  For example, in a real estate app, you might want to show a number of images of each house, showcasing the rooms and views. For these medium-sized lists, include a context indicator control, like a filmstrip of thumbnails, that lets users to jump to specific photos.

- Flip between views of an app

  For example, an app launcher can have one view that shows a user's favorites in a grid and another that shows the favorites in a list. You can add a FlipView control to let users navigate from one view to another.

Don't use a **FlipView** control for large collections. The repetitive motion of flipping through each item becomes tedious for users. A ListView control is a better choice.

Dos and Don'ts

	Use a context indicator when the items in the collection don't provide enough contextual information to help users know where they are in the collection. A FlipView for days of the week would not need a context indicator, whereas a FlipView for product images would.
	Give users an indication of where the current item is in the collection. Use a context indicator to help users know how large the collection is and where the current item is in relation to the others in the collection.
	Tailor the indicator for the number of items and the specific scenario. For small collections, you can show the entire collection in the context indicator. For medium-sized collections, you might want to show only 5 at a time, for example. For purely visual collections, you might display thumbnails; for text-based collections, you might prefer to display the alphabet so users can jump to the right letter.
Do	**Allow users to jump to specific items.** Make sure you always help users feel in control of where they are and where they want to go.
Don't	Don't use a FlipView for large collections. Use a ListView instead.

Guidelines for ListView controls

Is this the right control?

The ListView control offers a way to present collections of data, referred to as “items”, to users of your app. You can use ListView to produce various layouts, but it may not necessarily be the best option.

Here are some points to consider when determining whether you should use a ListView control.

- How are lists of items organized?

  Use ListView for apps that show a list of items as tiles, such as television shows, music, books, or emails. Visually you can display items in groups and variable-sized items within a group, and interactively there is full touch support, including semantic zoom.

  ListView is for items that expose multiple pieces of data, such as a picture with an overlay and a textual description. If your items are just a set of short strings, consider using the Select control.

  Don’t use ListView to display one large item per screen. For items where the user is meant to dwell on each item one-at-a-time, use a FlipView control instead.
- Is the number of columns within one category, fixed? Or does it vary based on the data and can be large?

  Although ListView offers a number of formatting features, is not meant to be used solely for layout. ListView is used to manage items that are fed by a common data source, and share some level of common interaction such as being selectable or tappable.
- Is the number of groups fixed or can it grow based on data?

  The more the number of items is dynamic, the more it makes sense to use ListView. The less dynamic they are, that is, the number is fixed, the less it makes sense and a simple template would be better.

The interaction model

It is helpful to think about what the purpose of your ListView is. Once this is established, setting the interaction model is easier. ListViews are generally used for one of the following purposes:

Static mode In this mode, the ListView is acting solely as a way to present content, and most types of interactivity are disabled. This is useful for collections of items that are read-only, and cannot be even activated or navigated into.

Content library mode This mode should be used when displaying a collection, or library, of content. It is typically used when presenting media such as pictures and videos.

In a Content Library, the primary user action is to tap on an item to invoke an action. Some (if not most) content libraries will also support the selection of items, using a new touch gesture we call cross slide.

Split view mode In this mode, tapping an item in the overview pane changes what is displayed in the details pane.

Picker mode is best used when the primary user action is selection (and the ability to invoke an item on tap is not important). In this interaction model, tap is used for selection in addition to cross-slide.

Choosing the right type of ListView control style

The ListView comes with two "layouts" built into the control: list layout, and grid layout.

List layout

List Layouts are single-column only with a top to bottom reading order, and they always pan or scroll vertically. List Layouts do not support item grouping.

Use the List Layout if the ListView is not the primary focus of your app.

For example, in common email apps, you'll often see a bunch of folders on the left side of the screen. This is a good candidate for a List Layout form of ListView.

Grid layout

The grid layout always pans horizontally, and items are laid out following a top-to-bottom, then left-to-right reading order. Grid layout supports item grouping.

Use the Grid Layout if the ListView is the primary point of focus in your app.

For example, a media rental app that displays movies should use a Grid Layout on its main screen, since the movie list is the primary point of focus of the app.

Dos and don'ts for list layout

Do	ListView is intended for items that expose multiple pieces of data. If your items are just a set of short strings, consider using the Select control.

Guidelines for grid layout

Use the Grid Layout if the ListView is the primary point of focus in your app and you want to emphasize content over all other things on the screen. The grid layout is useful for the following:

- Use grouping to organize collections of items

  There are two main situations where grouping is beneficial in a Grid Layout: when the collection of items becomes large, and/or when the items are naturally grouped—such as when viewing a collection of songs grouped into various albums.

  Grouping organizes the collection, and enables users to better find their way around.

- Use static mode to present content with no interactivity

  In this mode the ListView is acting solely as a way to present content, and most types of interactivity are disabled. This is useful for collections of items that are read-only, and cannot be even activated or navigated into.

- Use content library mode to display collections of content.

  This mode should be used when displaying a collection, or library, of content. It is typically used when presenting media such as pictures and videos.

  In a Content Library, the primary user action is to tap on an item to invoke an action. Some (if not most) content libraries will also support the selection of items, using a new touch gesture we call cross slide.

- Use the picker mode for user selection

  Picker Mode is best used when the primary user action is selection, and the ability to invoke an item on tap is not important. In this interaction model, tap is used for selection in addition to cross-slide.

Dos and don'ts for grid layout

Do	If possible, size the grid so that it takes up the full screen width. This places greater emphasis on the content.
	Size the content items so that they are easy to interact with touch.
	When you display images in your grid, generously size the items to allow those images to be clearly visible.
	Limit the amount of additional UI built around the grid, such as extra buttons. Some additional UI is fine, but overwhelming numbers of buttons can be distracting.
	When your grid contains many items, consider using the Semantic Zoom control. The

Semantic Zoom control is a transport control that allows users to zoom in and out of larger areas of UI, such as a very wide ListView. Use Semantic Zoom to assist users in navigating ListViews with large numbers of items.

General guidelines

Here are some general guidelines to follow, regardless of which list view control you use.

- Do not use ListView primarily for layout

  Although ListView offers some nice layout features, it is intended to manage a set of related items that are related by data source and interactivity. Items that aren't a part of a list, but rather discrete blocks of content that you want in the overall layout region are best displayed by using other layout options.

- Consider managing large amounts of hierarchical ListView content with Sematic Zoom

  For an app that contains many categories, with many items under each category, and where each category may contain different content types, semantic zoom is useful in managing the amount of content and categories in these views.

  One example of this is the people picker app. It is laid out like the below, where the alphabet list is one ListView, and favorites is laid out separately.

- Use a ListView to expose app content, not commands

  It is appropriate to put buttons inside of ListView items; however, don't use a ListView to expose a toolbar of buttons for Cut / Copy / Paste. In this case it is better just to use a bunch of Button controls laid out side-by-side.

- Suppress selection checkmarks in single-selection scenarios

  For list and grid items a checkmark should be suppressed where there is only a single-selection.

- Follow the Windows 8 type and layout grid

  Windows 8 features and apps follow a common type and layout grid that provides a consistent personality throughout the system—more details about this can be found here. Your ListView should present items that are sized and aligned to this type and layout grid. There are three ways you can ensure that this is done correctly:

  - You can use a set of pre-defined item templates that are properly designed to do this. These item templates can be retrieved here.

- You can use a Visual Studio Project Template with a ListView in Grid Layout already included in the app.
- Finally, if neither of these options works for you, you can read the article on the type and layout grid, and once fully understood, create your own item templates.

Managing the ListView layout when going to a snapped view

In Windows 8, apps can be snapped, which limits their width to 320px. Likewise, when unsnapped, the width of the app is much greater, allowing more and more content to be shown. Use the following guidance to help you determine what your ListView should do in these situations:

- If your ListView's content is not central to your app, consider using a Select control in the snapped view.

  For example, if your ListView contains a bunch of categories that a user can select from, but those categories are not the primary content of your app, might consider using a select control in the snapped view. The select control only supports a simplistic visual rendering of items, but if your content is not important, then this de-emphasis is appropriate.
- If your ListView's content is critical even in the snapped view, then switch between grid layout and list layout when your app is snapped / unsnapped.

  This allows your ListView's contents to be emphasized even when in the snapped state.

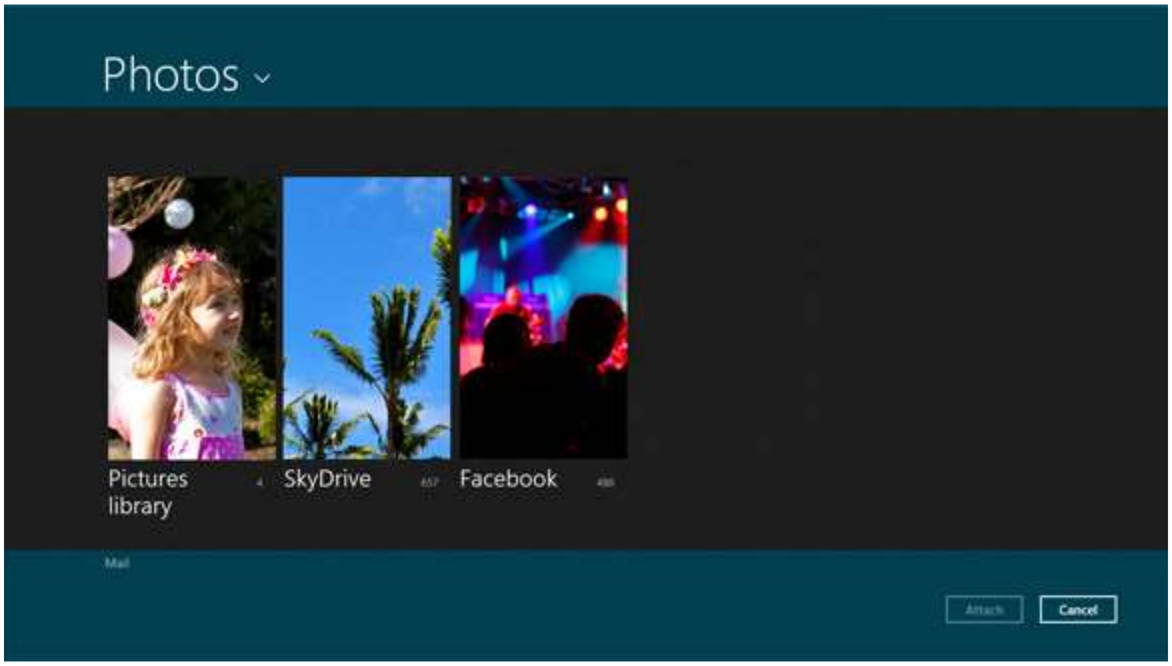
Guidelines for file pickers

Follow these guidelines to customize the file picker when you access and save files and folders. They also apply when you use the file picker to provide files, a save location, or file updates for other apps.

Appropriate use of file pickers in apps

- Access files and folders.

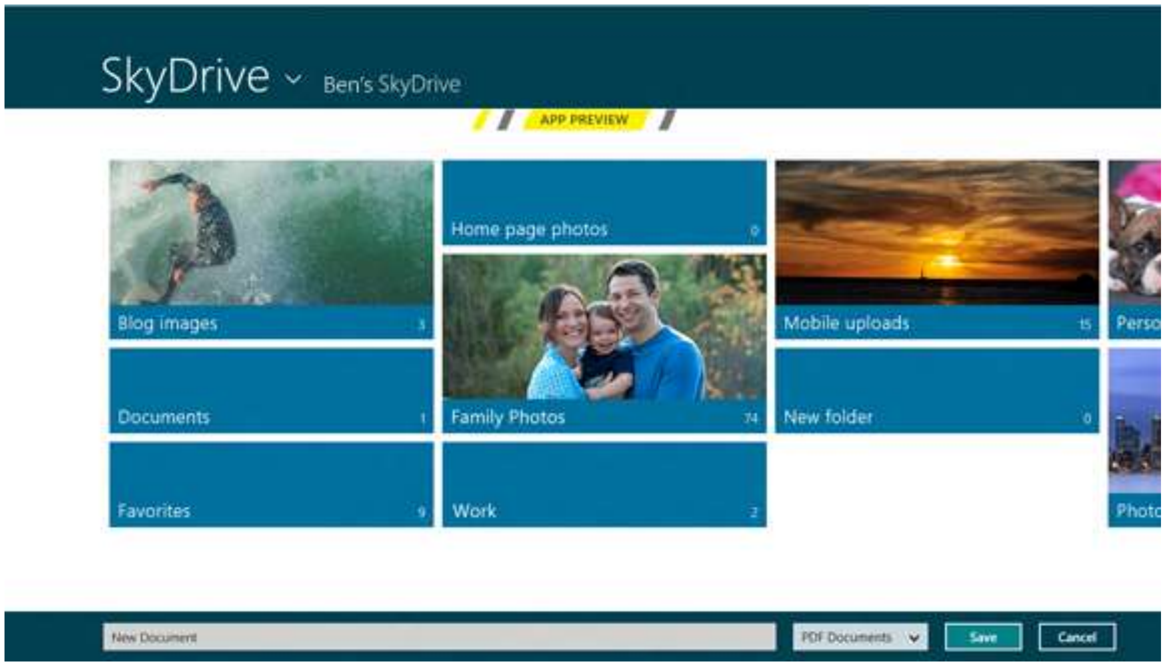
  Add a control to your app that calls the file picker to let the user pick files for your app to operate on. The user can then pick files through the file picker's UI as shown in the screen shot.

  The user can pick files from any location (including from other apps) that is listed in the drop-down list at the upper left in the file picker letterbox.

- Add "save as" to your app.

  Add a control to your app's UI that calls the file picker so that the user can specify the name, file type, and/or save location (like another app) of the file to save. The user can then navigate and save their file through the file picker's UI, as shown in the screen shot. Learn more about calling the file picker to save a file in How to save files through file pickers.



Inappropriate use of file pickers in apps

- Don't use the file picker to explore, consume, or manage file content.

  Instead, we recommend that you let users explore, consume, and/or manage file content by creating dedicated pages and UI in your app. This helps users focus on their current task and helps ensure that when users pick files, their experience is uncluttered by unnecessary functionality.

  For example, a photo gallery app should provide a customized, dedicated page and UI that lets users organize and view picture files within the app. The app can then customize this UI to best suit the user's needs. When the user wants to add files to the gallery, it would call the file picker which provides an experience that is specialized for picking.

- Don't use the file picker to save a file if a unique, user-specified file name or location is not needed.

  If the user doesn't have to specify a file name, file type, or location to save to, we recommend that your app save the file automatically in the background (without launching a file picker). This helps eliminate unnecessary user interaction, making the process of saving a file faster and less intrusive.

User experience guidelines: accessing and saving files and folders

- Set the file types to ensure that users can pick or save only file types that your app can handle.

Whether picking or saving files and folders, customize the file picker to display only the file types that your app supports and that are relevant to the user’s current task. For example, if the user is picking or saving a video, set the file types so that the user can select or save only a video file that uses a format that your app can handle.

This also applies to folder picking where the user is using files displayed in the file picker to help them determine which folder to select. By filtering the view to the proper file type, you help the user identify the correct folder faster.

- When accessing files or folders, set the view mode based on the kinds of items that the user is picking from.

  For example, if the user is picking pictures or videos, set the view mode to **Thumbnail**. If the user is picking any other kind of files or folders, set the view mode to **PickerViewMode.List**.

- Set the commit button text to match the user's current task.

  Whether picking or saving files and folders, customize the file picker by setting the commit button text appropriately for the user's current task. For example, if the user wants to pick a set of files to upload to your app, set the commit button text to "Upload".

- Set the suggested start location to the most relevant location possible based on the user's current task.

  Whether picking or saving files and folders, customize the file picker to suggest the most relevant start location possible based on the user's current task and the list of possible start locations provided by the **PickerLocationId** enumeration. For example, if the user is picking pictures, you may want to set the suggested start location to the user’s Pictures library.

- When accessing files, let the user pick a single file or multiple files based on the current task.

  For example, if the user is picking a profile picture, call the file picker for picking a single file. If the user is picking photos to send to a friend, call the file picker for picking multiple files.

- When saving files, set a default file name for the file to save.

  If users accept the default file name that you provide, they don't have to take the time to enter a different name and they can complete the "save as" task faster. You can use the **FileSavePicker.SuggestedFileName** property to set the default file name.

Guidelines for find-in-page

The following table presents the practices recommended for implementing find-in-page.

Practice	Description
Implement find-in-page to enable people to find matches in the current body of text.	Find-in-page enables people to find matches in the current body of text. • Document viewers and readers are the most likely type of apps to provide find-in-page. • The primary purpose of these apps is to enable the user to have a full screen viewing/reading experience. • Find-in-page functionality is secondary and should be located in an app bar along other functionality provided for the user which they will use when needed.
Don't use the Search charm to find text in the current body of text.	Apps that use the Search charm provide an experience which searches over a set of items, like web pages, movies, and events. The user enters a query into the search box, and a related result set is returned. Often, these results are sorted by using a relevance algorithm.

User experience guidelines

The following table presents the practices recommended for adding find-in-page to an app.

Practice	Description
Use an app bar to let people find text with your app.	Find-in-page functionality is secondary and should be located in an app bar. • Apps that provide find-in-page should have all necessary controls in an app bar. • If your app includes a lot of functionality beyond find-in-page, you can provide a **Find** button in the top-level app bar as an entry point to another app bar that contains all

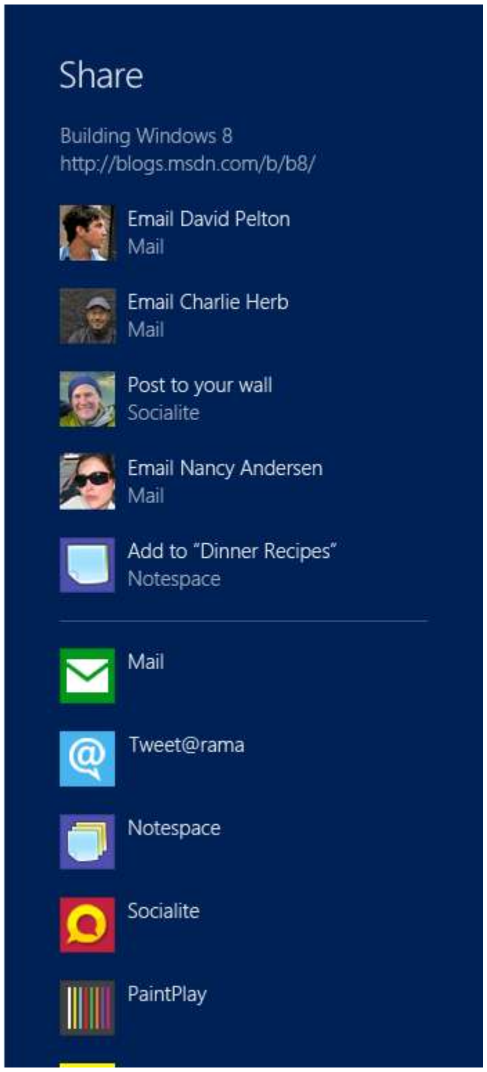
	of your find-in-page controls.
The find-in-page app bar should remain visible when interacting with the touch keyboard.	The touch keyboard appears when a user taps the input box. The find-in-page app bar should move up, so it's not occluded by the touch keyboard.
Find-in-page should remain available while the user interacts with the view.	People need to interact with the in-view text while using find-in-page. For example, people may want to zoom in or out of a document or pan the view to read the text. Once the user starts using find-in-page, the app bar should remain available with a **Close** button to exit find-in-page.
Enable the keyboard shortcut (CTRL+F).	Implement the keyboard shortcut CTRL+F to enable the user to invoke the find-in-page app bar quickly.
Include the basics of find-in-page functionality.	These are the UI elements that you need to implement find-in-page: • Input box • Previous and Next buttons • A match count • Close
The view should highlight matches and scroll to show the next match on screen.	People can move quickly through the document by using the **Previous** and **Next** buttons and by using scroll bars or direct manipulation with touch.
Find-and-replace functionality should work alongside the basic find-in-page functionality.	For apps that have find-and-replace, ensure that find-in-page doesn't interfere with find-and-replace functionality.
Avoid summary panes.	Summary panes are duplicative and should not be included in find-in-page functionality.

	• The easiest way for a user to identify the match they are looking for is to see it in the document, because seeing the match inline gives much more context than a summary of results pane. • People can move quickly through the document by using the **Previous** and **Next** buttons and by using scroll bars or direct manipulation with touch. • If you choose to include a summary pane, people should be able to access it by using a toggle button. • When the summary pane is toggled on, it should be visible whenever text is in the input box. • If the user closes the find-in-page app bar, then the summary pane should not be visible until the next time that the user enters text in the input box. • If the summary pane is toggled off it should not show again until the user toggles it back to the on state.

Charms, contracts, and devices

Guidelines for sharing content

People often come across information they're excited to share with someone or use in another app. The share charm is a lightweight, in context, easy way to share stuff. Follow these guidelines to let your users share content from your app with other people or apps, and receive shared content.



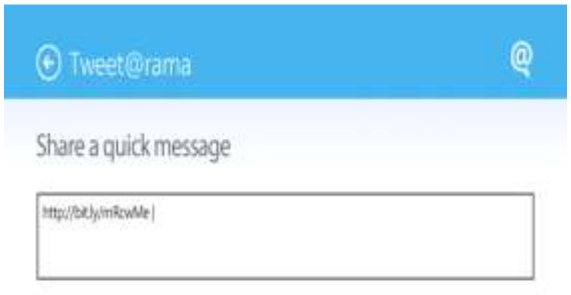
When people swipe from the side of the screen and tap the share charm, the share pane appears with a list of apps people can share their content with. This list includes any apps that are "share targets" for a particular data format.

If your app has content to share, your app is a share source. If your app can receive content from other apps, then it's a share target. Of course, apps can be both at once!

Note: If your app is both source and target for a particular data format, then it appears by default in the list of share targets each time people share from your app. Sometimes this is great, and sometimes it's a little silly to share with yourself. If it's silly, then display an error message that prompts the user to select a different target app.

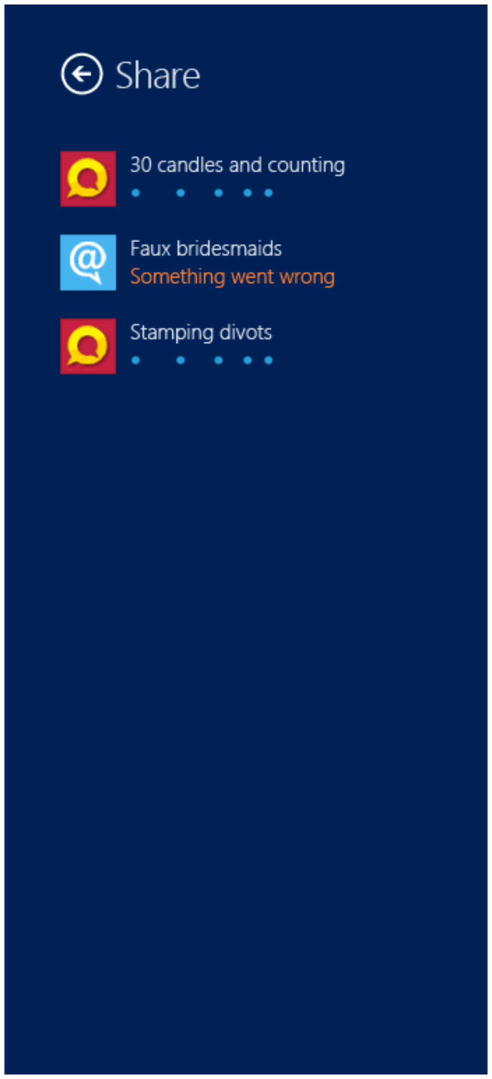
When people choose to share content, source apps provide the requested content in a shareable format, and display the metadata in the content preview. The chosen target app launches, reads the shared content, and displays whatever UI is appropriate.

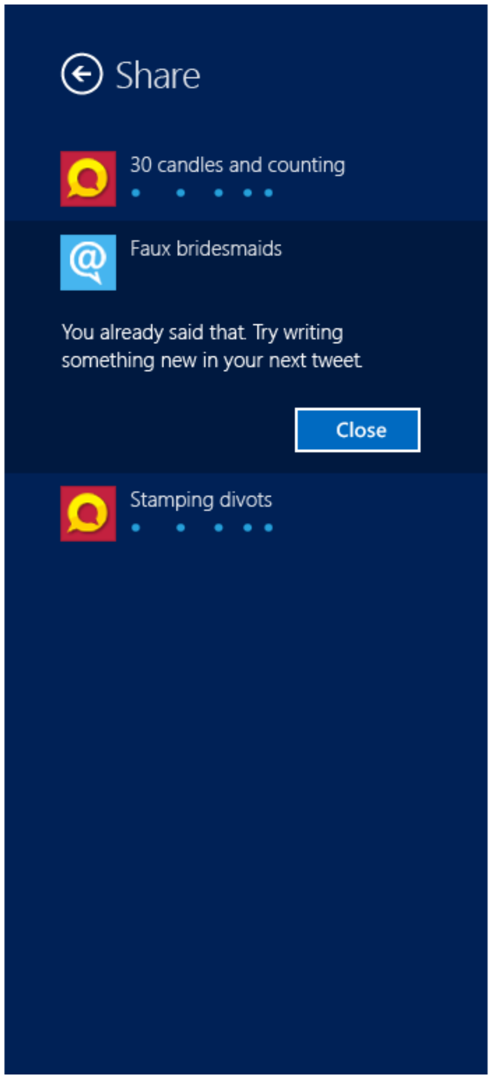
When the target app reports that the sharing operation has completed, it can return a QuickLink to be displayed in the share pane.



User experience with lengthy operations

Many sharing operations can take time to complete. People can start sharing a large amount of content and, while that is ongoing, they can switch to another app. Then, they can check the progress of the share operation by re-opening the share window.





You already said that. Try writing something new in your next tweet

If a sharing operation fails, Windows displays an informative message from the target app with steps to correct the problem when possible.

Best practices

We expect most, if not all, Windows Store apps to support some share tasks. Use the following best practices as you implement sharing in your app.

Source apps

- When possible, include links to online versions of local content.

  If an app supports downloading content that is also available to everyone on the web, it could share links to the online content rather than copies of the downloaded content. For example, suppose that a news site provides a rich reader app but also publishes the same articles on the website. If a user wants to share an article with a social networking site, the reader app can share links to the online version of the article that the user is currently viewing.

  If you do not provide a website that enables everyone to view content, your app must share a copy of the content. For example, suppose that a photo viewing app does not have a corresponding website. The photo viewer can share the photos with a target app that can upload those to its own website.

- Respect user selections

  When preparing content for sharing, your app can support the content in multiple formats. It's important that your app respect the selection the user makes. For example, don't include a link to a web page if the user has only selected a portion of that page.

- Set properties and use them to supply useful information

  When you package data for sharing, you have the option to supply a variety of properties that provide additional information about the content that is being shared. Taking advantage of these properties can help target apps improve the user experience. For example, providing a title and description that conveys what the user is sharing can help when the user is sharing content with more than one app. Adding a thumbnail when sharing an image or a link to a web page can provide a visual reference to the user.

- Provide a message to the user when sharing cannot be completed

  If your app supports sharing but a particular sharing operation cannot be completed for some reason, provide a message to be displayed in the share window that describes the steps that the user must take to correct the problem.

- Don't display a message that sharing is not supported by your app.

Windows displays a standard message to the user if your app does not support the sharing contract.

- Don’t provide alternate ways to invoke sharing

  Rely on the share charm and share window. Do not create a Share command on your app bar, or create a Share button in your app window or context menus.

- Preserve user selections

  Your app should preserve the user's selection even after the share flyout closes. This can help users if they want modify their selection, or share the same content to multiple targets.

- Provide a string that indicates what the user is sharing

  Provide a string that indicates what the user is sharing. For example, if the user is sharing a web page, include a string that has the URL of the page. If it's an image, include a description if possible.

- Support sharing of copied data

  If your app supports a way to copy data in the app, provide a way to share that same data.

Target apps

- Keep the look and feel the same between your target app and your primary app.

  Align your target app with the design for your primary app, including elements like fonts, colors, and controls. The target app should feel familiar to people who use your primary app frequently.

- Keep interactions simple

  Avoid time-consuming or complex interactions in your target app. In most cases, actions like text formatting, tagging people in photos, or setup tasks like connecting to a data source, are best handled outside of the Share charm.

- Keep navigation to a minimum

  When a user selects your app to share content, Windows 8 automatically provides a back button for navigating back to the app list. You shouldn't depend on this back button—a target app can't use this button for its navigation. Also, avoid adding your own back button, or making your users navigate back and forth between multiple pages in your target app. Instead, use inline controls, such as progressive disclosure controls, select controls, and inline error messages.

- Don't use light dismiss flyouts

The Share UI already uses light dismiss. Including another light dismiss element in your target app can cause confusion with your users.

- Acknowledge user actions

  When a user taps the Share charm or invokes the Share UI, let them know that the system is responding to their action—for example, through an inline message—before closing the share pane. This helps give the user confidence that their share started successfully.

- Put important buttons where they can be easily reached

  Put share buttons where people can reach them easily. We recommend putting share buttons on the right side of the screen, so people can reach them with their right thumbs.

- Remove links that lead people away from the sharing experience

  When a user is sharing content, take steps to ensure they remain in the sharing context. For example, if your app has links that lead to other areas of your app (such as to a home page), remove or hide them so the user doesn't leave the sharing experience accidentally.

- Previews should match the actual content whenever possible

  If your app includes a preview of what the user is sharing, that preview should match what will actually be shared as much as possible.

- Use QuickLinks well

  QuickLinks act as links to your app that are customized for a specific set of user actions. Take advantage of these QuickLinks if there are specific actions that might save the user time and encourage them to share content with your app in the future. These actions might include sending an email to a specific person, or doing something with a photo.

Guidelines for creating custom data formats

People share a variety of information when they're online. Successful apps take the time to analyze the types of information that people are most likely to share with others. Then they package that information so that the receiving apps can process it correctly. In many cases, the information people want to share falls into one of the six standard formats that Windows 8 supports. However, there are many instances in which having a more targeted data type can create a better user experience. For these situations, your app can support custom data formats.

Why custom formats?

The share feature in Windows 8 supports six standard data formats:

- Text
- HTML
- Bitmap
- StorageItems
- URI
- RTF

These formats are versatile, which makes it easy for your app to quickly support sharing and receiving shared content. The drawback to these formats is that they don't provide a lot of context for the receiving app. To illustrate this, consider the following string, which represents a postal address:

```
1234 Main Street, New York, NY 98208
```

An app can share this string by using **DataPackage.setText**. But because the app that receives the text string doesn’t know exactly what the string represents, it is limited in what it can do with that data. By using a custom data format, the source app can define the data that is being shared as a postal address. This gives the receiving app some additional information that it can use to process the information in way the user expects.

Custom formats can also help you provide more efficient ways of sharing data. For example, a user has a collection of photos stored on Microsoft SkyDrive, and decides to share some of them on a social network. This scenario is tricky to implement using the standard formats, for a few reasons:

- You can only use the URI format to share one item at a time.
- The StorageItems format is intended for sharing local files. To use StorageItems in this scenario would require that your app download each picture, and then share them.

- Text and HTML let you provide a list of links, but the meaning of those links is lost—the receiving app won't know that these links represent the pictures the user wants to share.

Using a custom format to share these pictures provides two main benefits:

- Your app can share the pictures faster, because you could create a collection of URIs, instead of downloading all the images locally.
- The receiving app would understand that these URIs represent images, and could process them accordingly.

Defining a custom format

If you decide that your app can benefit from defining a custom format, there are a few things you should consider:

- Consider the experiences you want to enable. It's important that you think about what actions your users want to take, and what data format best supports those actions.
- Understand the standard data formats, so you don't create a custom format unnecessarily.
- Don't rely on format combinations. For example, don't expect an app to understand that if sees one format, it should also look for a second format. Each format must be self-contained.
- If you create your own custom format, make the definition of that format available to other app developers.
- After you publish a custom format, don't change it. Consider it like an API: elements might get added or deprecated, but backward-compatibility and long-term support are important.
- When you name your format, match the name to the contents of the format. For example, `UriCollection` indicates that any URI is valid, while `WebImageCollection` indicates that it contains only URIs that point to online images.
- Carefully consider the meaning of the format. Have a clear understanding of what the format represents and how it should be used.
- Review the structure of the format. Think through whether the format supports multiple items or serialization, and what limitations the format has.

Choosing a data type

One of the most important decisions you'll make when defining a custom format is the WinRT data type used for transferring it between source and target apps. The **DataPackage** interface supports several data types for a custom format:

- Any scalar type (integer, string, **DateTime**, and so on)
- **IRandomAccessStream**
- **IUri**
- **IStorageItem**
- A collection of any of the above items

When defining a format, select the type appropriate for that data. It is important that all consumers and recipients of this format use the same data type—even if there are different options. Otherwise, it may lead to unexpected data type mismatch failures in target apps.

Custom format example: WebFileItems

To better illustrate how to think about creating a custom format, consider a fictional custom format, WebFileItems.

At the fictional company, Fabrikam, a developer writes an app that shares files stored online. One option would be to download the items to the local computer, but that could be time-consuming and inefficient. Instead, the developer decides to create a custom format for use with these file types.

First, the developer considers the definition of the new format. In this case, it's a collection of any file type (document, image, and so on) that is stored online. Because the files are on the web instead of the local machine, the developer decides to name the format WebFileItems.

Next, the developer needs to decide on the specifics of the format, and decides on the following:

- The format should consist of an IPropertyValue that contains an InspectableArray that represents the URIs.
- The format must contain at least 1 item, but there is no limit as to how many items it can contain.
- Any valid URI is allowed.
- URIs that are not accessible outside of the source app's boundary (such as authenticated URIs) are discouraged.

With this information mapped out, the developer now has enough information to create and use a custom format.

Adding custom formats to your app

After you define a custom format, use these tips for adding the format to your app:

- Test the format with other apps. Make sure that the data is processed correctly from the source app to the target app.
- Stick to the intended purpose of the format. Don't use it in unintended ways.
- If you're writing a source app, use at least one standard format as well. This ensures that people can share data with apps that don't support the custom format. The experience may not be as ideal for the user, but it is better than not letting them share data with the apps they want.
- If you're writing a target app, consider supporting at least one standard format. This way, your app can receive data from source apps—even if they don't use the custom format you prefer.

Guidelines for clipboard commands

Clipboard commands—copy, paste, and cut—provide people with a familiar way to transfer content from one location to another. With these commands you can help people transfer content:

- Within the same app
- Between Windows Store apps
- Between desktop apps
- Between Windows Store apps and desktop apps

Although Windows 8 supports other ways for apps to exchange information—such as through sharing—copy and paste commands remain an expected part of the Windows experience. Your app should support them whenever possible.

Where and how to support copy and paste

In general, support copy and paste for any editable content that a user can explicitly select, such as a subset of a document or an image. Also consider supporting copy and paste commands for content that people might want to use somewhere else. For example:

- Images in a photo gallery app
- Computation results in a calculator
- Restaurant address in a restaurant-search app

As always, be aware of rights management and other factors that might restrict the use of copy and paste commands. For example, if your app supports viewing rights-managed mail, a policy might restrict the user from copying all or parts of such content.

After you decide where to support copy and paste, think about how to add it to your app. Here are a few guidelines to help you:

- Make sure it's clear what a user is copying, or where a user can paste content.
- Provide support for paste only on editable regions and canvases in your app.
- Consider implementing an undo command, as copy and paste can lead to content being deleted or replaced.
- If a control already supports copy and paste, use the control's implementation. If you need to build your own implementation of copy and paste, make the experience consistent with these controls.
- Consider supporting sharing if you are also supporting copy.

Where not to use copy and paste

Here are a few considerations for where you shouldn't use copy and paste.

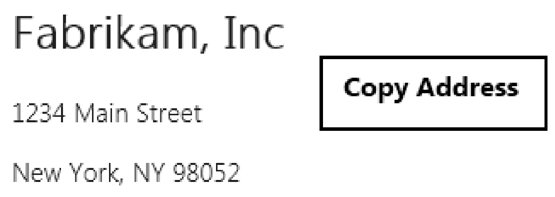
- Don't provide support for copying content that can't be selected—either explicitly, or through a context menu.
- Don't provide support for copying text that is not part of the core content of your app. Do not copy titles, headers, and button text.

Accessing copy and paste commands in your app

Before implementing support for copy and paste commands, consider how people access copy and paste commands. In general, people access copy and paste commands by one of three methods: a context menu, the app bar, or keyboard shortcuts.

Use a context menu:

- For items that people can select only through tap-and-hold gestures—such as hyperlinks or embedded images. For example, let's say your app displays an address to the user, and you want the user to be able to copy that address. A great user experience would be to create a Copy Address command that people can access when they either right-click or tap-and-hold the address. This command would then copy the address to the clipboard, from which the user can paste it into the app of their choice.



- For text selection (both editable and read-only).
- For paste operations where the target is well defined, such as a cursor location or a table cell.

If the preceding guidelines don't apply to your app, then you can likely use the app bar. Some examples include:

- When your app supports the selection of multiple items.
- When the user can select a portion of an image.
- When the target of a paste command is clear—such as pasting a screen shot on a canvas.

We strongly encourage you to always support keyboard shortcuts. Also, if your app supports the paste command, disable it when the clipboard is empty or if it contains content that your app doesn't support.

Guidelines for search

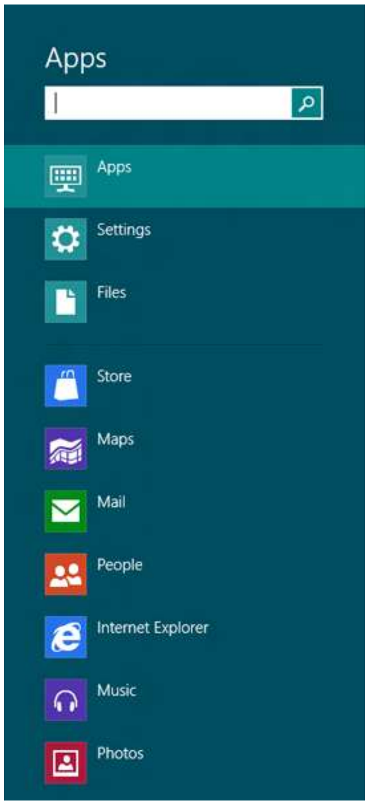
Adding search through the Search charm lets people search your app's content from anywhere in their system at any time. If your app is the main app on screen, people can search its content immediately by selecting the Search charm. Otherwise, people can select the Search charm and then select your app from the list of apps in the search pane to search your app.

Using the Search charm helps you:

- Take advantage of muscle memory that people have built by using the Search charm in the system and in other apps.
- Ensure that people have a consistent and predicable experience when they search and when they change search settings.
- Make your app more visible by placing it at the top of the list of searchable apps if the user searches your app frequently.

Appropriate use of the Search charm

- Search for content in your app



Your app's content could include content on the local file system, or content that's accessible through a web service. You can set up the Search charm so that you can respond to a user's search query and display search results in a page that you design for your app.

When a user selects the Search charm, a search pane is opened containing a search box where they can enter a query, as shown in the screen shot.

If your app is the main app on screen, it is automatically highlighted in the list of apps in the search pane, like **Apps** is highlighted in the screen shot. For example, if you open the Search charm while in the Store app, the Store app would be automatically highlighted in the list of apps.

Tip You can use shortcut keys to access the charms, including the Search charm. The Windows Logo Key + C lets you select any of the charms, and the Windows Logo Key + Q selects the Search charm.

Inappropriate use of the Search charm

- Don't add any UI controls for search in your app.

  Your app’s position in the list of apps in the search pane is determined by how often the user searches your app by using the Search charm. As a result, adding an additional control for search in your app UI could actually make your app more difficult to search when your app is not the main app on screen, because it may not be as visible in the search pane's app list.

  Additionally, adding your own controls for search unnecessarily duplicates features and might confuse people. For example, one way that app-specific search UI might confuse people is by causing the user to have more than one search history with a particular app. One search history would be based on use of the Search charm, which the system tracks and maintains. Another independent search history would be based on the use of the app-specific search UI.

- Don’t place search UI in the app bar.

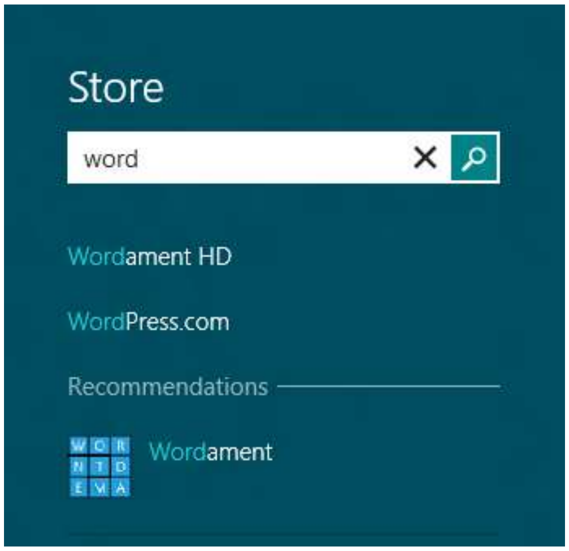
  Maintain the app bar as a place to show commands that are unique and specific to your app.

- Don’t use the Search charm to add a "find-in-page" feature to your app.

  When people use "find-in-page", they want to stay on the current app page. But the Search charm navigates them away, and replaces the current page with another app page that displays search results.

  Instead, add a control to your app bar that lets people "find-in-page" and group the "find-in-page" feature with related features like "replace".

Customizing suggestions and placeholder text in the search pane



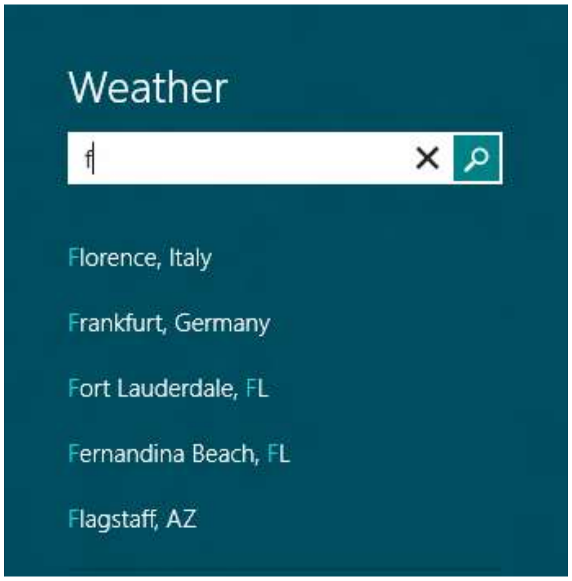
When the user selects the Search charm and starts typing a query into the search box, search suggestions are shown just below the box in the search pane. These suggestions are supplied by the main app on screen if it has implemented the Search contract.

There are two types of suggestions an app can provide: query suggestions and result suggestions. Query suggestions are auto-completions of the user's query text, and represent possible queries that the user might want to search for. Result suggestions are strong or exact matches to the user's query that the user may want to see immediately.

In the screen shot of the search pane, the Store app provides two query suggestions and one result suggestion for the user's query, "word".

- Always provide query suggestions to help the user search quickly.

  Use query suggestions as way to auto-complete query text that people can search for in your app. This is a great way to help people search quickly by reducing the amount of typing needed to complete a search. Instead of entering the entire query, people can select one of the suggested queries and immediately execute the search.



Your query suggestions should contain the user's current query text.

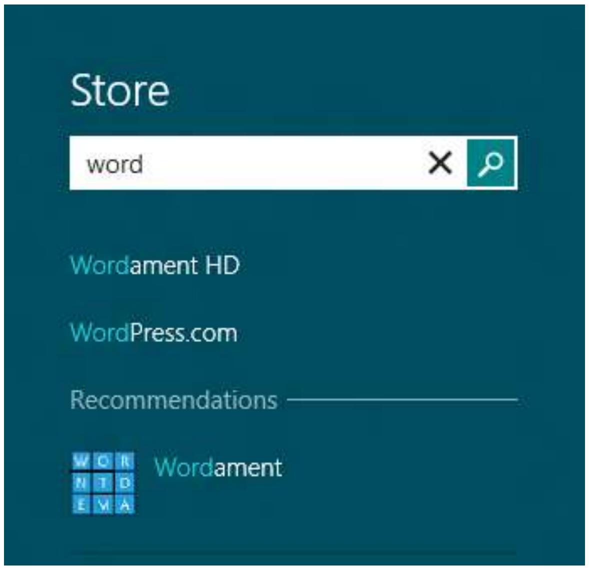
Because your query suggestions should be auto-completions that are based on the current query text, they should actually contain the current query text. For example, the suggested queries provided by the Weather app in the screen shot all contain the current query text, "f".

Your query suggestions should directly reflect the results that your app can provide.

By reflecting the results your app can provide, your query suggestions can help the user figure out what they can search for in your app. For example, the Weather app in the screen shot automatically completes the user's query to suggest cities for which the app can provide weather reports.

- If the user selects a query suggestion, immediately take the user to a search results page for the selected query.
- If you want to recommend strong or exact matches for the user's query, provide result suggestions.

  Use result suggestions to let the user go directly to the details of a particular result without the need to navigate to a search results page.



A result suggestion should consist of an appropriate image or thumbnail, a relevant title or label, and a brief description.

The image, title, and description help the user to determine quickly whether the suggested result is what they were searching for, as shown in the screen shot.

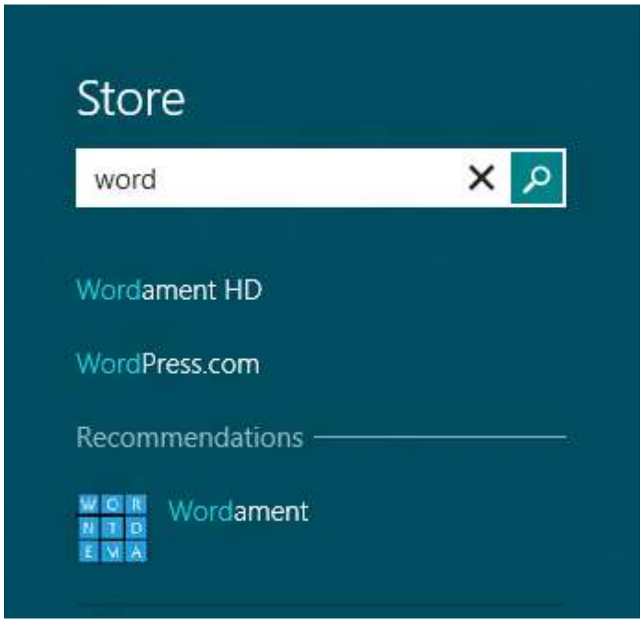
- If you want to supply multiple result suggestions, use and label separators to help people distinguish between results.

  For example, when providing more than one suggestion for results with different content types (like movies vs. TV shows), use labeled separators to provide a meaningful distinction between the content types of the result suggestions.

  Separators can be added to the list of suggestions that your app supplies to the search pane, but each separator counts towards the five-suggestion limit.

- If the user selects a result suggestion, immediately take the user to the details of that result without first taking them to a search results page.
- If you provide both types of search suggestions (queries and results), provide only one result suggestion. It should be displayed last, at the bottom of the list of suggestions.

While the user enters a search query, Windows automatically provides suggestions for possible queries in the search pane. These suggestions are based on the user's search history with your app. They are shown first, before suggestions that your app provides are displayed. Showing your suggested result last helps visually distinguish it from suggested queries, as shown in the screen shot.



- Supply no more than five search suggestions.

  The search pane will show only the first five suggestions (for queries and/or results) that are supplied by your app.

- Don't use suggestions to filter or scope search results for your app.

  Filtering and scoping refine and manipulate the set of search results that is associated with a specific query. They should be placed with the results in the app's search results page. In contrast, suggestions are directly related to the query text that the user enters in the search box.

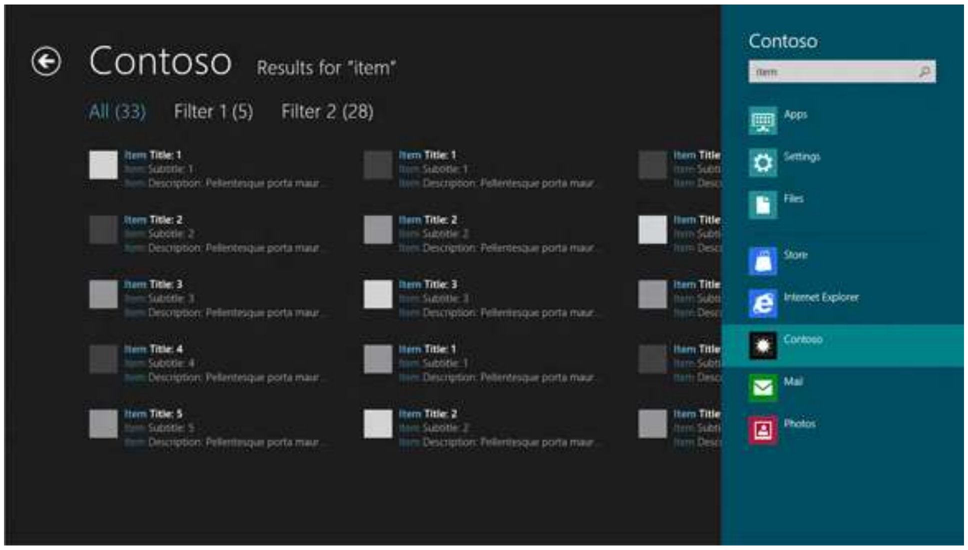
- Use placeholder text in the search box to describe what people can search for in your app.

  We recommend that you set the placeholder text of the search box to a brief description of the content in your app that a user can search for. For example, a music app that supports searching by album name, song name, or artist name should set the placeholder text to be: "Album, artist, or song name".

  Placeholder text is shown only when the search box is empty and is cleared if the user starts typing into the box.

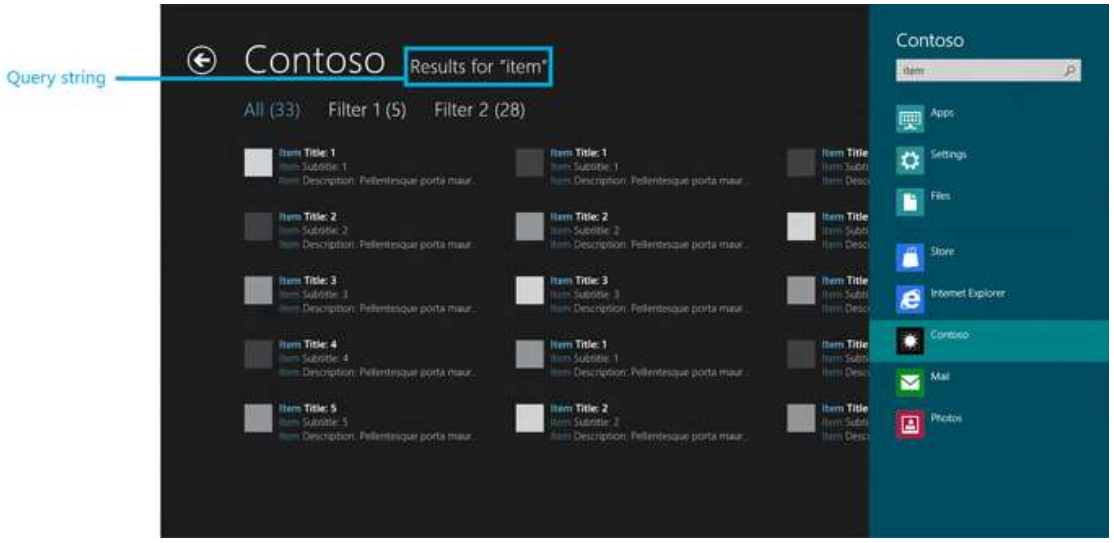
Designing a search results page

When the user submits a search query to your app, you display a page that shows search results for the query. Because you design the search results page for your app, you can ensure that the results presented to your user are useful and have an appropriate layout. For example, the screen shot shows the search results page created for the Contoso app which displays example search results for the "item" query.



- Let people see what they searched for (their query text).

  For example, in the Contoso app's search results page, the "Results for" string next to the "Contoso" app name indicates that the current set results is for the "item" query.



- Use the ListView control and Search contract templates to bring the Windows 8 look and feel to your app.

  Using the **ListView** control for displaying search results helps produce a consistent and predictable user experience and reinforces what people have already learned in the system. The Microsoft Visual Studio Express 2012 for Windows 8 templates provided for the Search contract use the **ListView** control. Learn more about the **ListView** control in Adding ListView and GridView controls.

  The **ListView** control lets you use a grid layout or a list layout to display your search results. Search results are ranked and therefore typically have a strong order from best match to weakest, which should be reflected in how the results are ordered in both layouts. Grid layouts in Windows 8 use horizontal scrolling. Therefore, make a grid of search results that is ordered from top to bottom and then from left to right. Because list layouts use vertical scrolling, search results that are displayed in a list should be ordered from top to bottom.

- Avoid putting important or relevant results or information on the right edge of the screen.

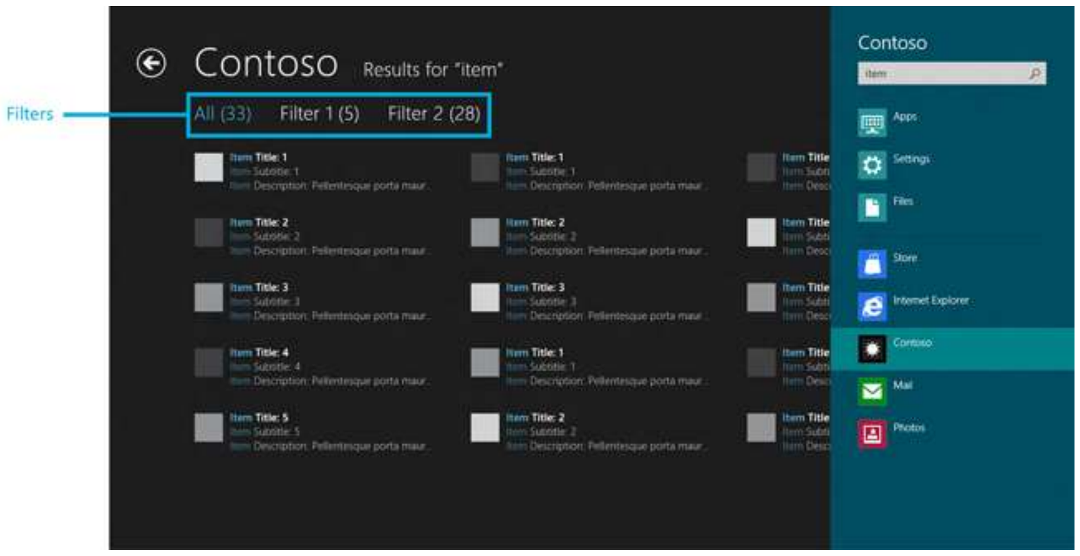
  The Search charm is lightweight and quickly gets out of the way whenever the user interacts with the app canvas. However, to help people quickly glance at results, avoid showing the most important or relevant result on the right side. This side might be temporarily covered up when the search pane is visible.

- Let people filter and/or scope search results from the search results page.

  You can improve your app's search results page by letting people set filters and scopes to refine the set of search results. Best practices for letting people filter and scope search results in your app's search results page include:

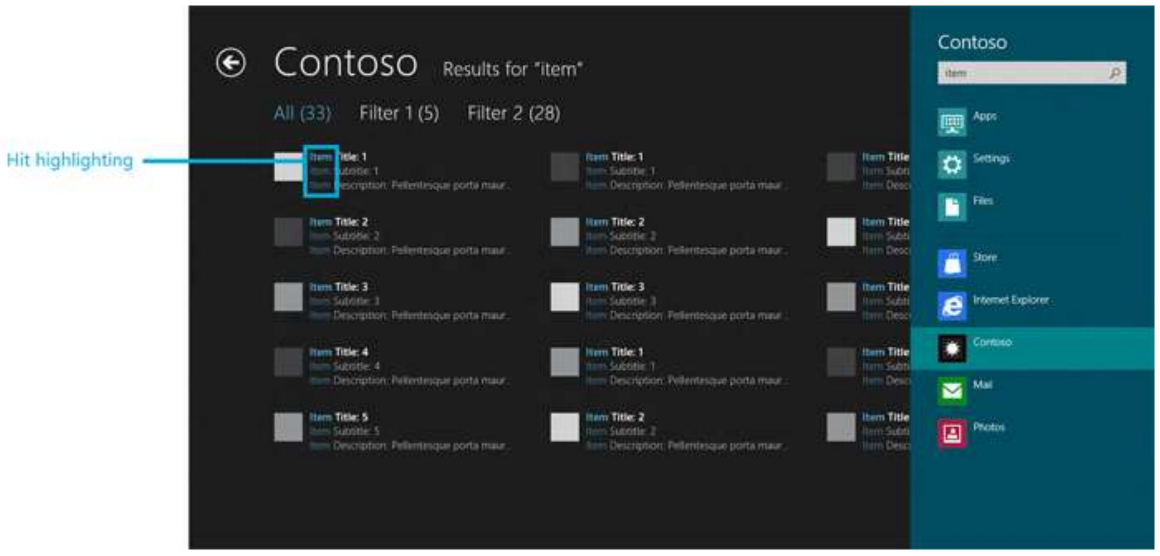
  - Indicate the number of results available with each filter or in each scope. This helps people understand whether they are effectively refining their search.
  - Provide a way to clear the filters and see all results.

  For example, the Contoso app's search results page provides a set of filters above the results, with counts next to each filter.



- Indicate why a search result matches the query.

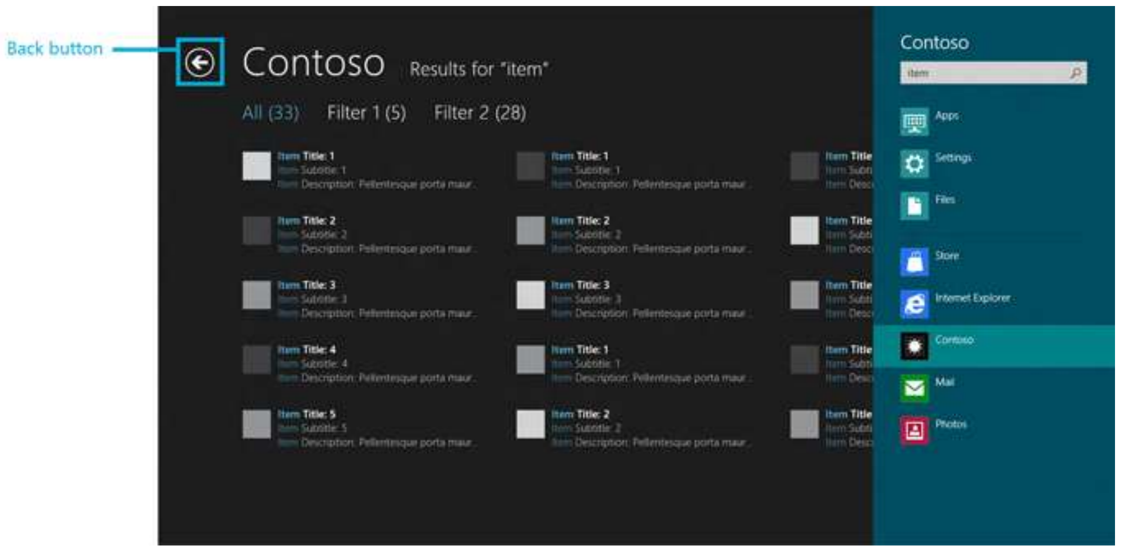
  For example, the Contoso app's search results page in the screen shot highlights the user's query ("item") in each result. This is called *hit highlighting*.



- Let people navigate back to the last-viewed page after they look at the details for a result.

  When searching, people are often going to be looking at several results as they gather information. Looking at a specific result and then getting back to the search results page should be easy.

  A common way to accomplish this is to include a back button in the app UI as shown in the screen shot. This back button should be used to go to the previous page that the user was interacting with before submitting their search.



Tips for activation from search

- Be sure to handle a search activated event for an empty query.

  When people open the search pane (through the Search charm), they can choose an app to search with when they enter a query into the search box. In your activated event handler, when your app is activated for search, you should also check to see if your app was activated with an empty **queryText** string.

  If your app is activated with an empty **queryText** string and your app is already running or is suspended, return to the app's last-viewed page. If your app isn't running or suspended, take the user to a landing page appropriate for this search.

  Generally, your app's default, home page is an appropriate landing page when the **queryText** is an empty string, but you can also design an app page specifically for this purpose.

- Save the previous state of the app when the app is activated for search.

  The power of the Search contract enables people to invoke an app for search from anywhere in the system by using the Search charm. The app must be able to respond to a search activation event at any time. Apps should save their state and provide people with a way to get back to that previous state where appropriate.

  Example: In a mail app, the user might have a partially composed message. Later, if the user switches to the mail app to search in it, the mail app should save the message as a draft and provide people with a way to get back to it.

- Navigate to your search results page while responding to Search activation when your app is snapped.

If your app is snapped when the user uses it to search (causing the search activated event to fire) your app automatically unsnaps (causing the unsnap event to fire) and becomes the main app on screen.

The search activated event and the unsnap event are fired in this order:

1. Search activated event

   In response to this event, your app should display its search results page for the query.

2. View state changed event

In response to this event, check whether your app has unsnapped, and if it has, adjust your app to become the main app on screen. Do not take the user away from your app's search results page, because if your app doesn't load the search results page, the user might feel that the app is broken and is not searching properly.

- Save the search results page, the filters, and scope for the last query in case your app is activated to search for that query again.

  The Search charm lets people switch between multiple apps quickly to compare results for the same search query. The user might submit a search query to your app and set filters for your app's search results. The user might then switch to another app, search that app using the same query, and then come back to view your app's search results again.

  When this happens, your app is activated for search (again) and your app's search results page should show the same filters that the user set the first time that your app displayed results for this query. If the current query is the same as the last query, you should not get a new set of search results, but instead load your previous search results page, including the filters and/or scope that the user had applied, and the exact location where the user was focused.

Guidelines for file picker contracts

Follow these guidelines to customize the file picker when you use the file picker to provide access to your app's content, a save location, or file updates for other apps.

Appropriate use of file pickers contracts in apps

- Provide files.

  Integrating with the File Open Picker contract lets your app provide users and other apps access to your app's content through the file picker.

- Provide a save location.

  Integrating with the File Save Picker contract lets your app provide users and other apps with a save location through the file picker.

  If your app provides a save location, also provide access to your app's content by integrating with the File Open Picker contract.

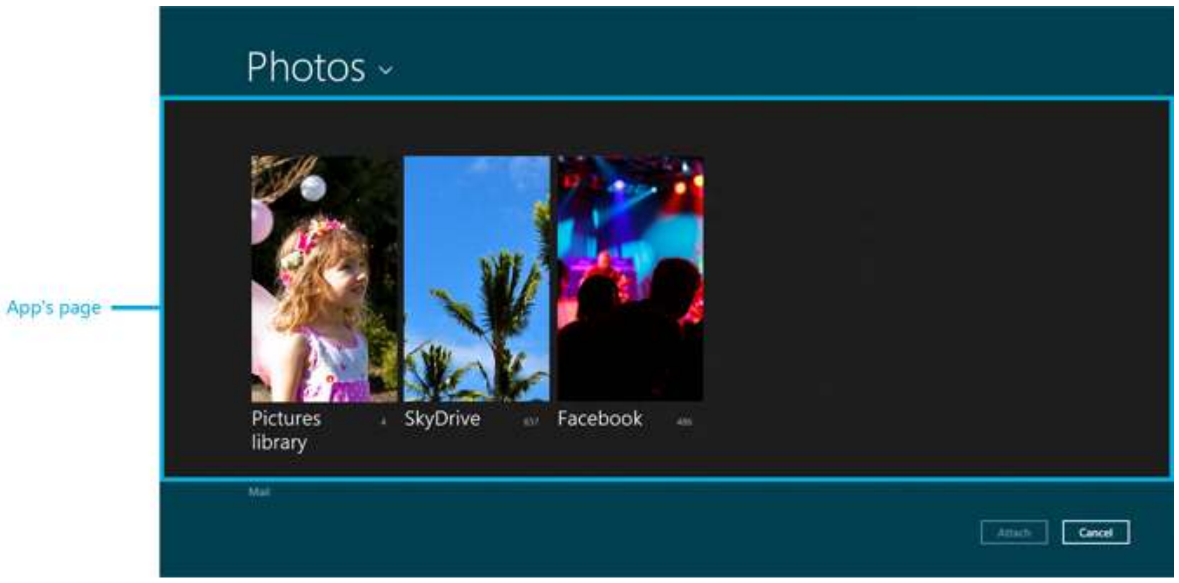
- Provide real-time file updates.

  Integrating with the Cached File Updater contract lets your app perform updates on files in your app's repository and provide updates to local versions of the files in your repository. From the users' perspective, this lets them to operate on a remote file that your app maintains in its repository as though that file were local. For example, the user could use a text editor app to edit a file and Microsoft SkyDrive could update the version of that file in its repository).

  If your app updates files, it should also provide a save location and access to files. It can do this by integrating with the File Save Picker contract and the File Open Picker contract, respectively.

User experience guidelines: providing files, a save location, and file updates

If your app provides files, a save location, or file updates through file pickers, you need to design a page for your app that displays files (or other UI) to the user. This page is displayed in the center area of the file picker.



This screen shot has been modified to emphasize and label the center area of a file picker window to show where your app's page will be loaded

- Design the page to display in the file picker (your file picker page) based on an existing page that your app uses to display files.

  If your app is providing files for the user to pick through a file picker, your app should have an existing page that lets users view files. We recommend that you design your file picker page so that it is consistent with this existing file-view page. Making these two pages consistent with each other helps users feel comfortable and familiar with how your app displays files in the file picker.

  To further ensure that users feel comfortable with your app's file picker page, use the same (or similar) navigation UI and error reporting for your file picker page as you use for your existing file-view app page. Especially in the case of navigation, users expect similar commands and locations to be available in both the file picker page and the existing file-view page.

- Design your file picker page around your user's current task.

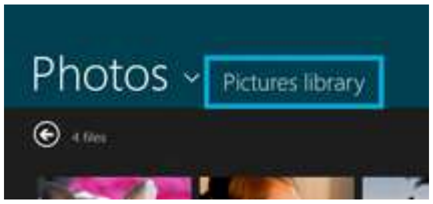
  Keep the UI for your file picker page focused on the user's current task. For example, help users pick, save, or update files, by stripping out UI that is not directly related. This helps make sure that using the file picker is a quick, in-and-out experience that gets users back into the app they were using (the calling app or caller).

  For example, if a file picker is being used to access files that are provided by your app, remove UI that supports complex or detailed navigation, search, or information that cannot be picked.

If you want to let the user perform other tasks like consumption, modification, and file management, add controls or other UI for those tasks to your main app.

- Set the title of the file picker to the name of the user's current location.

  This gives users a predictable way to orient themselves as they use your app from the file picker. The title, which is highlighted in this screen shot, appears in top bar of the file picker letterbox.

  In this screen shot, the title is `Pictures library`, which lets the user know where they are in their system. Update this title whenever the user navigates to a different location.

- All file locations that are accessible to your app should be accessible from your file picker page.

  If your app can normally access files in a particular location, your file picker page should also give access to files in that location. Access to locations should also be consistent across all file picker pages, if your app has more than one page. This ensures that users have predictable access to files and locations.

- Use the UI templates and controls available in Microsoft Visual Studio.

  Visual Studio has built-in templates that you can use to help create the file picker view for your Windows Store apps.

Additional UX guidelines: providing files

- Display files in your file picker page in a unique and relevant way.

  Organize and display files in a way (or ways) that is unique to your app and ensures that your page is both convenient and relevant to users. This should still be consistent with what users see in the view that your app uses to display files within the app.

- Display files on your file picker page that are not accessible by using Windows or other apps.

  Differentiate your app from Windows and other apps by providing access to files in locations that are not accessible from other apps or from Windows. These locations can include your app's storage folders or remote servers.

- Design the UI of your file picker page to respond to the selection mode of the calling app.

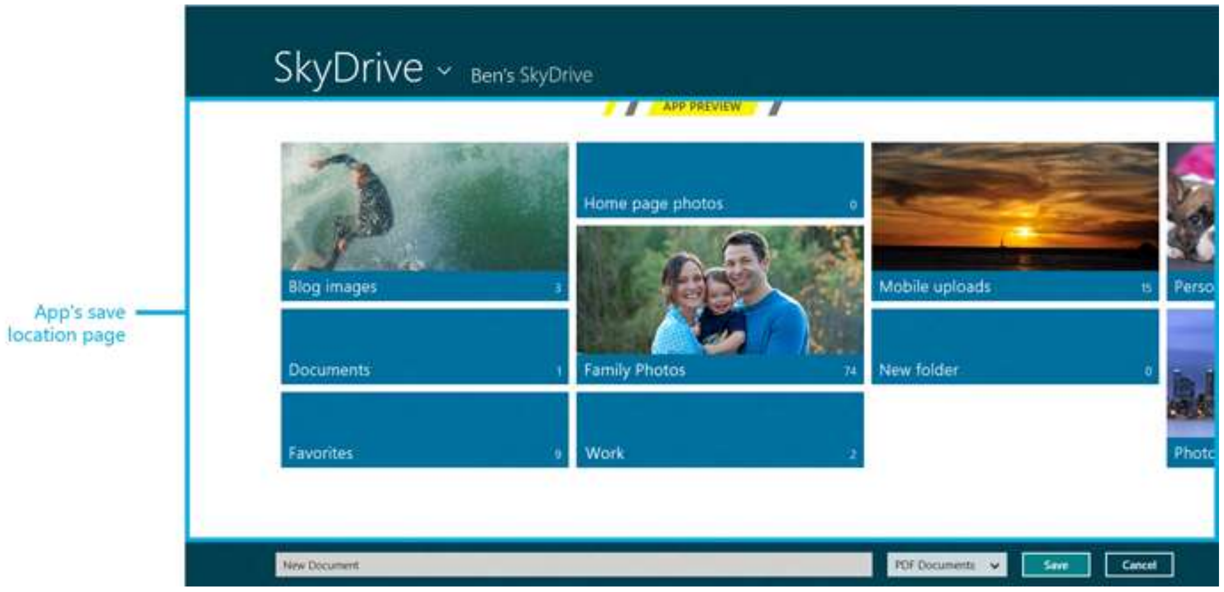
When an app calls a file picker to access files, the calling app specifies whether the user can pick a single item or multiple items. We recommend that you design your app page to indicate selected files appropriately and differently for each selection mode. For example, if the user is trying to select a profile picture (a single item selection) from files provided by your app, they might tap or click more than one photo while they try to decide which to pick. In this situation, your app UI would only allow one item to be selected at a time. Otherwise, if the user is trying to select multiple files to share with their friends (multiple item selection), your app UI would allow multiple items to be selected simultaneously.

- For webcam, photography, and camera apps, design the UI of your file picker page around taking pictures.

  Make sure users can get back into the app they were using (the calling app or caller) by simplifying your app's UI for your file picker page. Limit the controls you provide on your file picker page to controls that let the user take a picture, and let the user apply a few pre-processing effects (like toggling the flash and zooming).

  All available controls must be visible on your file picker page because your app bar is not accessible to the user from the file picker. We recommend that you organize these controls on your file picker page similarly to the way they are organized in your app bar. Position them on your file picker page as close as possible to where they appear in your app bar, at the top or bottom of the page.

Additional UX guidelines: providing a save location



This modified screen shot emphasizes the center area of a file picker window where the page that displays your app's save location will be loaded.

- Provide save locations that are not accessible to users through Windows or other apps.

  Let users save files to locations that are not easily accessible through Windows or other apps, like your app's storage folders or remote storage locations.
- Change the files displayed on your file picker page based on the file type selected.

  If the user changes file type in the file picker's file-type drop-down list, update your view to display only files that match the selected file type. Filtering displayed files by type provides the user with an easy, consistent method of identifying the types of files they're interested in.
- Allow the user to replace a file easily by selecting the file in your app's file picker page

  If the user selects a file in your file picker page, automatically replace the file name in the file picker file name box so users can easily replace existing files.

Additional UX guidelines: providing file updates

- Provide a repository that can track and update files for users.

  If users use your app as a primary storage location where they regularly save and access files, you may want your app to track some files to provide real-time updates for users.
- Design your app and file picker page to present a robust repository.

  If users use your app as a primary storage location for their files, design your app and your associated file picker view to protect against data loss, which could be caused by frequent file updates or conflicting file versions.
- Let users resolve issues encountered during updates.

  To help ensure a successful update, your app should to notify users in real time (using **UIRequested**) when a file is being updated or saved and user intervention is needed to resolve an issue. It is especially important that your app help users resolve issues with credentials, file version conflicts, and disk capacity. The UI you create should be lightweight and focused specifically on resolving the issue. If more than one step is required (like login), all the steps should be handled in your app's file picker page. Once complete, your app can enable the file picker commit UI. In addition, your app should update the file picker title to gives users context about where they are.

  If the problem cannot be solved in real time by the user, or if you simply need let the user know what happened (perhaps an error occurred that the user can't resolve), we recommend that you notify the user of the problem the next time your app is launched instead of immediately when the problem occurs via **UIRequested**.
- Provide additional information about update and save operations from your normal app pages.

Your main app UI should let users manage settings for in-progress and future operations, get information about in-progress and previous operations, and get information about any errors that have occurred.

Guidelines for app settings

The Settings charm is always available and provides a single access point for all settings. Settings include system settings that always apply, app-related settings that are brokered by the app with permissions, and settings that are specific to the current app.

People swipe from the side of the screen to display the charms and tap the Settings charm to display the settings window. The settings window includes both app and system settings.

The app may provide **SettingsCommand** entry points, which appear at the top of the settings window. Two entry points, **Permissions**, and **Rate and review**, are provided by the system. The bottom of the settings window includes PC settings provided by the system, like volume, brightness, and power.

Note Only apps that are installed through the Windows Store have the **Rate and review** entry point. Side-loaded enterprise apps don't have this entry point.

The entry points open settings Flyouts, which contain the app's settings, help, About info, and any secondary commands or information that your users would access infrequently. An app can have multiple entry points for settings, and each settings Flyout can have multiple options.

User experience guidelines

Choose which app features are accessed in app settings

- Choose features that affect the behavior of the app as a whole and that are adjusted only occasionally, like choosing between Celsius and Fahrenheit as temperature default units in a weather app, or changing account settings for a mail app.
- Use the Settings charm to provide access to app info that's not needed very often, such as privacy statements, help, app version, or copyright info.
- Don't include features that are part of a typical app workflow, like changing the brush color in an art app. These features belong on an app bar or on the canvas.

Add your entry points for app settings to the settings windows

- Group similar or related settings together under each entry point. Avoid adding more than four entry points.
- Combine less-used settings into a single entry point so that more common settings can each have their own entry point.
- Expose the same entry points regardless of the app context. If some settings are not relevant in a certain context, handle them in the app settings Flyout, not by changing the entry points presented in the settings window.

- Be specific in naming entry points. Reflect the information or settings that are disclosed to the user when they select the entry point. When there is only one settings entry point that covers multiple categories, avoid repeating the term "Settings" or using a synonym. Instead, use a qualifier of settings, like "Defaults".
- Promote more frequently used settings to have their own entry point in the top-level charm. For example, if you need to have About, terms of use, privacy statement, and support information represented in your settings, place them under the About entry point rather than having an entry point for each setting.

Create settings Flyouts

The **SettingsFlyout** control implements all of the UI requirements in this section.

- Launch a consistent UI surface, by using a settings Flyout, from all of the settings entry points specified in the settings window.
- Settings Flyouts should be narrow (346 pixels) or wide (646 pixels) and should have a header that includes a back button, the name of the entry point that opened the Flyout, and the app's icon.
- Use the **showPanel** and **hidePanel** animations, so that settings Flyouts slide in from the side of the screen that they're closest to and slide out toward the same side.
- Place your settings Flyout on the same side of the screen as the charms. The charms and the settings window may be on the left side of the screen if the system's text direction is right-to-left.
- Use a light-dismiss surface, so that the settings Flyout disappears when the user touches anywhere on the screen outside of the surface. This enables the user to quickly change a setting and get back to the app. The UI created through the **SettingsFlyout** classes is a light-dismiss surface. If you create your own UI, make sure that your settings Flyout is dismissed when the user touches the app (on contact down), when the app loses activation, such as when the charms are opened, and when the app is snapped.

Add settings to settings Flyouts

- Aim for simplicity.
    - Provide well considered defaults for your features and add as few settings as possible.
    - Avoid settings hierarchies deeper than two levels.
    - Present content from top to bottom in a single column, scrollable if necessary, but limit scrolling to a maximum of three times the screen height.

- When a user changes a setting, the app should reflect the change immediately. A new setting value is applied as soon as the user stops interacting with a particular setting control. This is recommended for light-dismiss surfaces. If your app doesn't apply the new values to the settings store by the time the settings Flyout closes, the new values might be lost, because a settings Flyout shouldn't have a commit button. Handle the **beforeshow** and **afterhide** events to do control initialization and state serialization.
- Use an additional level of UI or an expand/collapse model to repeat settings exposed under multiple objects listed in the UI. For example, a weather app that provides per-city settings should provide the list of cities that can be configured in the first settings level and let the user tap on the city to get to the next level to configure options there. A mail app that supports multiple accounts should show the list of accounts and let the user tap on an account to open the corresponding second-level settings UI.
- Use the provided app UI style sheets and common controls. The style sheets provide standard layout, style, and spacing for common controls.
    - Use class="win-label" or <name of style> for the title of the settings Flyout.
    - Use H3, or <name of style> for headers above sections of settings.
    - Use <p> or <name of style> for descriptive text above controls.
    - Use built-in control styles for control labels when available or <label> otherwise.
    - Use built-in control styles or <name of style> for control status (such as "On" for a toggle switch).
- Use controls consistently for settings of the same type. Add a descriptive message if one of the controls is disabled.
- Each control should have a simple, explanatory label.

Here is a list of basic controls recommended for settings:

- Toggle switch: Use toggle switches to let users set values on or off.
- Button: Use buttons to let users initiate an immediate action without dismissing the current settings UI.
- Hyperlink: Use a hyperlink when the action takes the user to another UI surface and dismisses the current settings UI.
- Text input box: Use an input box to let users enter text. Use the type of text input box that corresponds to the type of text you're capturing from the user, such as email or password.
- Radio button group: Use a radio button group to let users choose one item from a set of up to 3 mutually exclusive, related options.

- Select control: Use a select control to let users choose one item from a set of 6 or more text-only items.

Inappropriate use of settings

- Don't add to the settings area any commands that are associated with common app workflow. These commands should be placed in the app bar or on the app canvas.
- Don't use an entry point in the settings window to do something directly without launching another UI surface.
- Don't use the settings window to navigate into another part of the app. When the settings window closes, the user should be in the same place in the app that they were when they entered settings. Controls such as the app bar are a more appropriate place for navigation.
- Don't use the **SettingsFlyout** class as a general-purpose control. It's intended only for settings UI launched from the Settings charm.

Guidelines for app help

Help content should be a single page and can include text, links, and images. To provide the most current content, include a link to your support or home page or embed an online page into your help page.

Appropriate use of Help

Before including Help content for your app, consider whether your app actually needs it. For example, if your app has proven to be easy to use, you might decide that Help content isn't necessary. If there are one or two UI elements in your app that are a bit tricky for people to understand, try integrating tips into the UI, creating a simple in-app demo, or redesigning those elements to avoid creating help that only addresses one or two simple fixes.

Dos

If you've decided to add a Help page for your app, here are some things to consider:

- Do label the entry point in the Settings pane **Help** to clearly identify it. Although the label is configurable, it's better to stay consistent with all other apps.
- Do open to your Help page from the Settings pane. The Help entry point shouldn't link directly to a website.
- Do keep the Help page short and easily scanned.
- If your Help content does not fit into a single topic, or if you want to include information that requires updating, then add links to your support website from the Help content. But keep in mind that linking to a web page takes your customer out of the app experience and should be used sparingly.
- Do write clearly and use a conversational voice, but avoid idioms and colloquialisms.

Don'ts

Here are some things to avoid when creating Help content:

- Don't use the Help entry point to link directly to a website.
- Because Help should feel like it is inside the app, don't launch a web browser when the user taps **Help**.
- Avoid technical terms and jargon whenever possible.
- Don't create Help content in the app that lists every known issue or documents every single feature of your app. Remember that you can link to your support web page, which might be a better location for information like that.

- Don't use Help to notify customers that a newer version of the app is available. This information is available in the first-level of the Settings user interface.

Guidelines for devices that access personal data

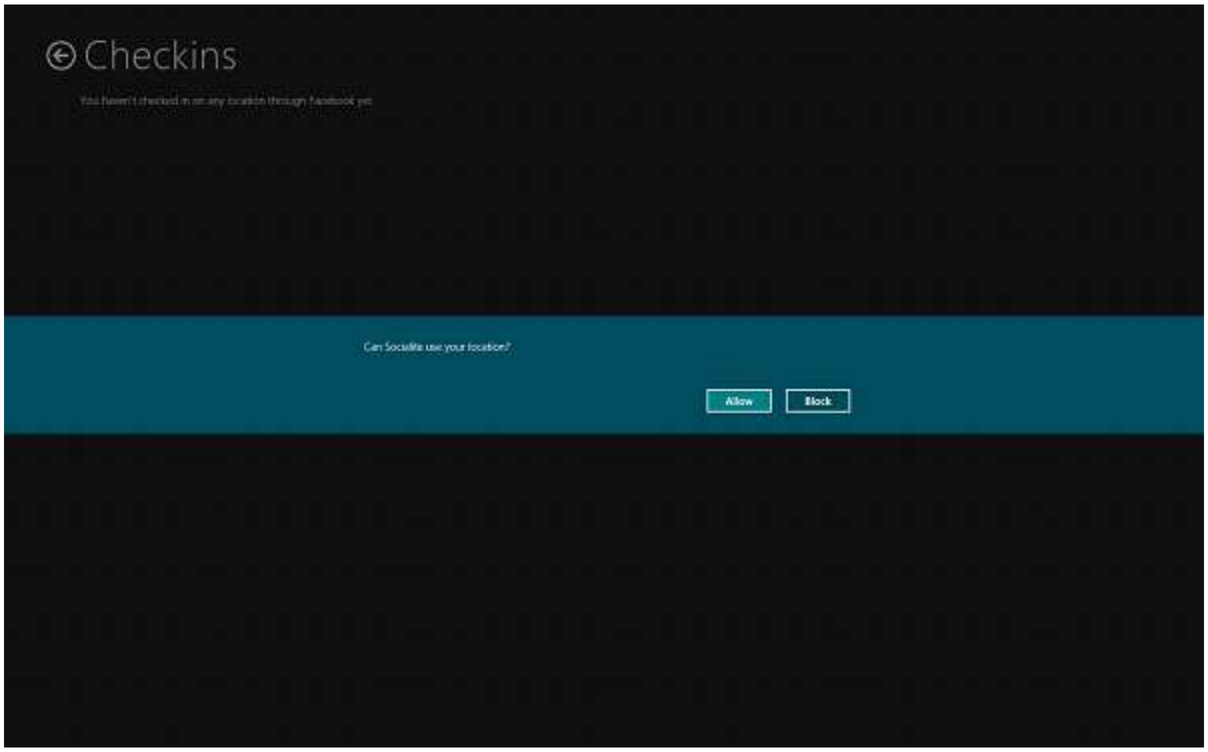
Microphones, cameras, location providers, and text messaging services can access the user's personal data or cost the user money, so they are considered *sensitive devices*. Windows Store apps have features to ensure the user has control over which apps may access these sensitive devices.

How the user controls the app's use of sensitive devices.

Permissions for Windows Store apps to use sensitive devices are controlled on a per-app, per-user level. The user controls permissions by using the consent prompt, or by using the Settings charm

The consent prompt

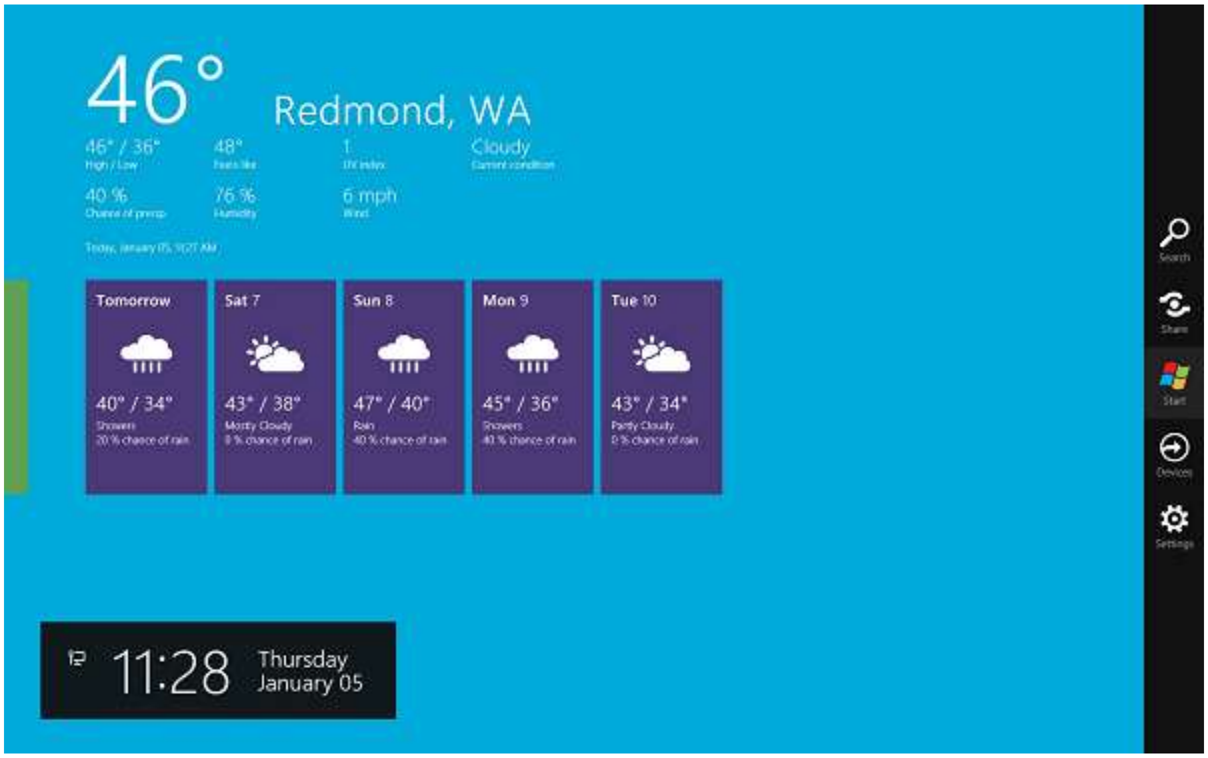
The following screen shot shows the consent prompt as it appears in a Windows Store apps. The prompt appears the first time an app accesses a sensitive device. It gives the user options to block or allow the app's access to the device capability. Windows remembers the response, so that the user is not prompted again for the same app.


Checkins
You haven't checked in on any location through facebook yet
Can Socialite use your location?
Allow
Block

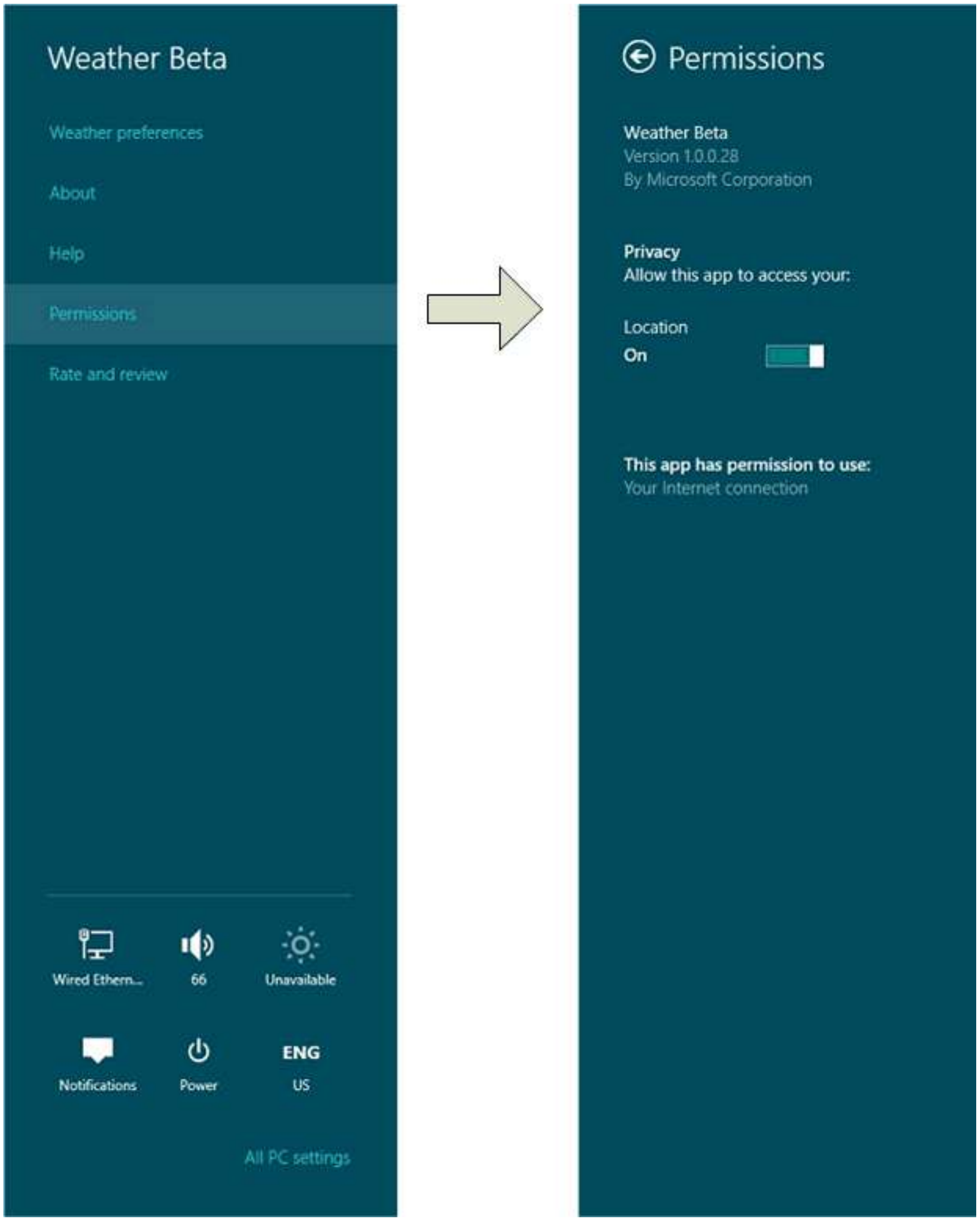
The Settings charm

Users of Windows 8 can also control each app's access to sensitive devices by using the Settings charm. The user taps on the Settings charm to open a settings flyout. They can then enable or disable the app's access to sensitive device capabilities.

The Settings charm is pictured here on the right side of the app:



The Settings charm provides flyouts that let the user enable or disable sensitive capabilities. An example of a flyout is shown here:



Permissions

Weather Beta
Version 1.0.0.28
By Microsoft Corporation

Privacy
Allow this app to access your:

Location
On

This app has permission to use:
Your Internet connection

Start using the device only if it's needed

The first API call that accesses the device triggers the consent prompt. If an app's primary purpose does not require access to a sensitive device, it can confuse the user if a prompt for permission to use the device appears as soon as the app starts. Follow these guidelines for a good user experience.

Guideline	Example
If use of the sensitive device is not essential to your app, don't access it until the user specifically requests it.	A social networking app has buttons for "Check in with my location" and "Take a profile picture". This app should not access location or the camera until the user clicks the corresponding button.
If an app requires device access for its main function, then it can access the device when the app starts.	An app for capturing live videos from an attached camera requires the camera for its main purpose. It's OK for this app to use the camera when it starts, to show a video preview right away.

First use of the device must be on the main UI thread

The first call to start using the device must be made on the UI thread so that the consent prompt can be shown to the user. If the consent prompt can't be shown, the user can't grant device access to the app.

To make sure you first access the device on the UI thread do the following:

- Don't use a background task for the first use of the device.
- In an app using JavaScript, the first use of the object that accesses the device should not be in the activation handler for the app.
- In an app using C# or C++ with XAML, the first use of the object that accesses the device should typically be done in MainPage.xaml.cs instead of App.xaml.cs.

Note In apps that use C# or C++ in Windows 8, the first use of the device object should be on the STA thread. Calls from an MTA thread may result in an error code of E_ILLEGAL_METHOD_CALL.

What if access to a device is turned off?

An app's access to a sensitive device may be disabled for one of these reasons:

- The user blocked access by using the consent prompt.
- The user disabled access to the device in the Settings charm.

- The device is not present on the system.

To provide a good user experience when access is disabled, do the following:

- Handle the error from the API that occurs when your app attempts to access a disabled device capability.
- Display a message to the user that informs them that the device capability is disabled. Also tell them how they can re-enable it by using the Settings charm, and by trying again to use the capability in the app. Follow the Guidelines for notifying the user to provide this message.
- Provide UI for the user to re-initiate access to the device if the device is re-enabled. Reinstantiate or reinitialize the object that accesses the device by using this UI. For example, a mapping app may provide a button for refreshing the current location. The button must instantiate a new **Geolocator** object.

Guidelines for notifying the user of device revocation

The following guidance describes how apps should react when a user revokes a sensitive capability. The purpose of this guidance is to properly notify the user of loss of functionality, and guide them towards turning a capability back on without taking away from the app's experience.

Your app should tell the user that the capability is turned off, and that they can enable the capability by using the app's settings.

Reason the device is disabled	Sample error message format
The user blocked access by using the consent prompt or the Settings charm.	"Your <*device capability*> is currently turned off. To change your <*device capability*> setting, open the settings charm and tap permissions. Then <*enable action*> to start using <*device capability*> again. " • Replace <*device capability*> with webcam, microphone, location, or text messaging. • Replace <*enable action*> with the action the user needs to take in the UI to reinitialize access the capability, like clicking a button.
The device capability isn't present on the system.	“You do not have the required <*device capability*> present on your system.”

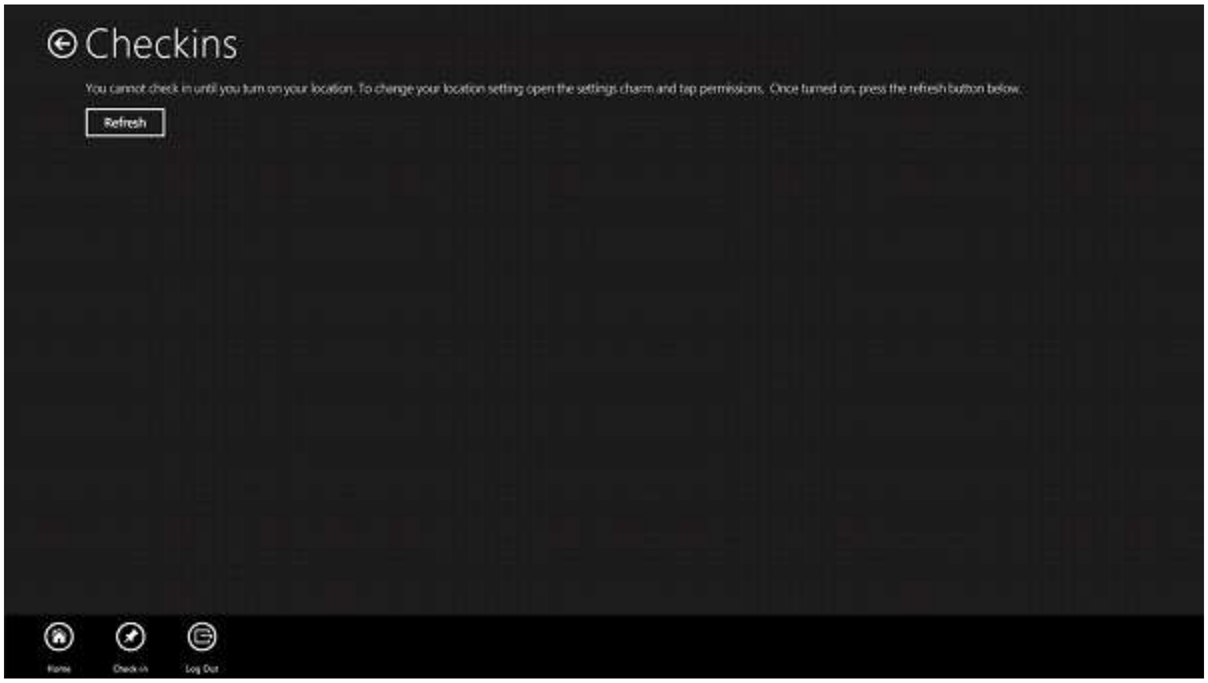
The UI you use to present the message about a disabled device capability depends on whether the device capability is essential to the app.

Display a flyout or inline text if the device is non-essential

If the sensitive device is not essential to your app, display the message in a flyout at the point of invocation, or in unobtrusive inline text.

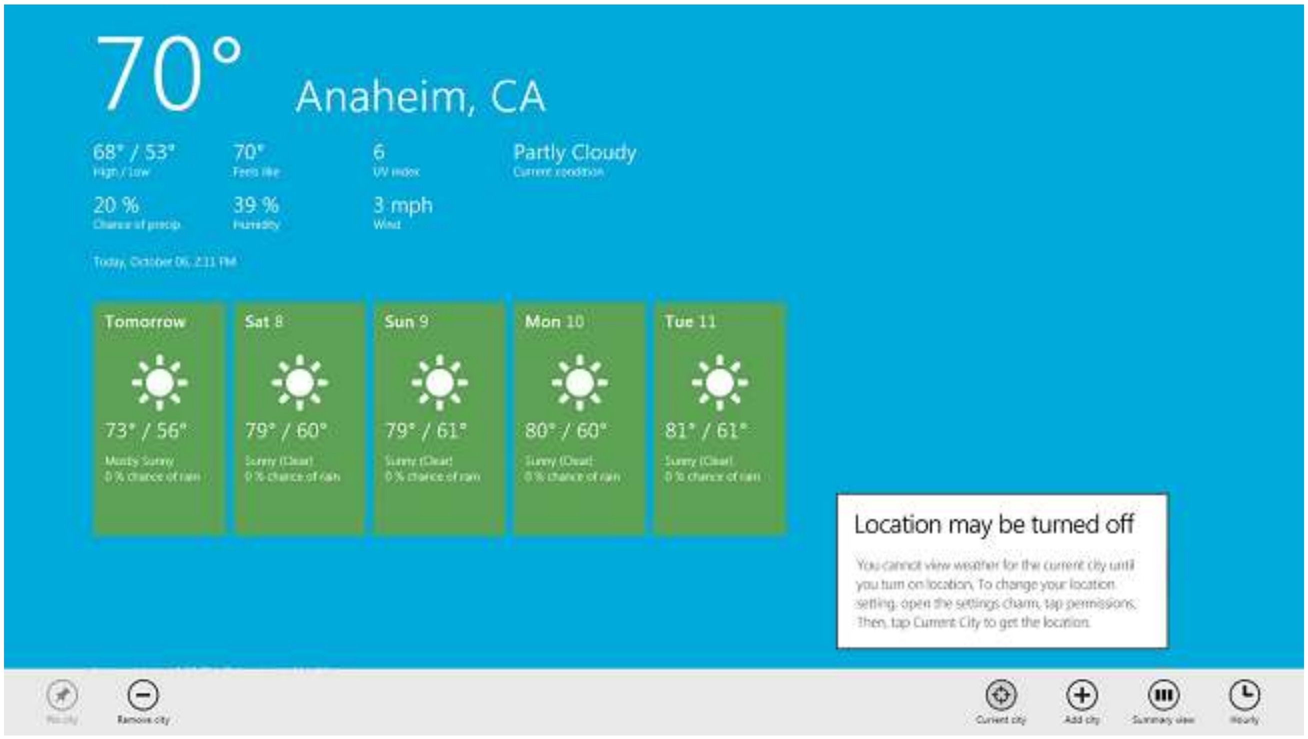
For example, if a social networking app has buttons for "Check in with my location", and the user clicks the button when the required device capability is disabled, the app should show the error message in a flyout near the button, or in inline text.

The following screen shot shows an app that displays the message in unobtrusive inline text. The message reads: "You cannot check in until you turn on your location. To change your location settings open the settings charm and tap permissions. Once turned on, tap the refresh button below". The refresh button in this example needs to reinstantiate the object that accesses the device capability, in this case, location.


Checkins

You cannot check in until you turn on your location. To change your location setting open the settings charm and tap permissions. Once turned on, press the refresh button below.

The following screen shot shows an app that displays the message in a flyout near the button. The message reads: "You cannot view the weather for the current city until you turn on location. To change your location setting, open the settings charm and tap Permissions. Then tap Current city to get the location." In this example, the Current city button should reinstantiate the object that accesses location.



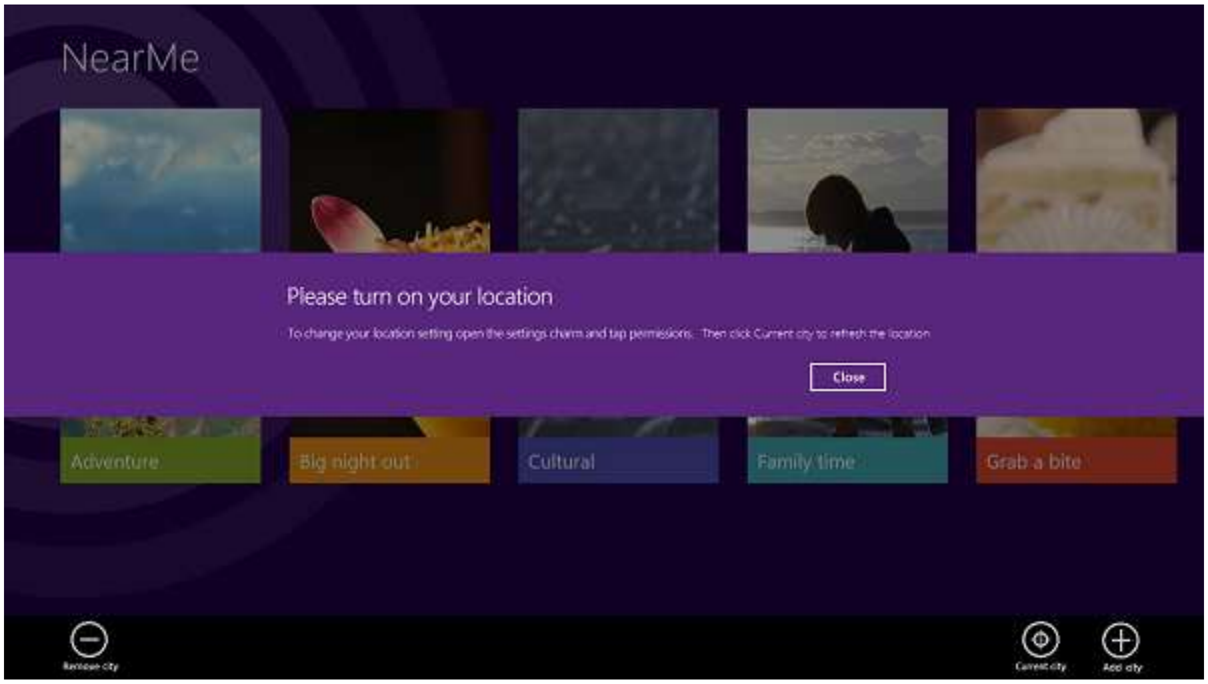
Location may be turned off

You cannot view weather for the current city until you turn on location. To change your location setting, open the settings charm, tap permissions. Then, tap Current City to get the location.

Display a dialog if the device capability is essential

If an app requires device access for its main function, display the error by using an error dialog.

In the screen shot that follows, an app for displaying nearby points of interest requires the location capability for its main function. So it uses a dialog to display a message that instructs the user to re-enable the location capability.


Please turn on your location

To change your location setting open the settings charm and tap permissions. Then click Current city to refresh the location

Other guidelines for revocation messages

Here are some more guidelines for informing the user that a device capability is disabled:

- Notify the user of an unavailable capability if the user attempts to use a device capability that's not essential to the app. The user should be aware of the loss of functionality.
- Make the device revocation message clearly visible to the user.
- Use a flyout when a capability is not available, a control to use the capability is provided, and the capability is not essential to app functionality.
- Don't let the text interrupt the app's flow.
- Don't use notifications to notify the user of device capability unavailability.
- Don't programmatically try to launch the **Permissions** page in the Settings charm
- Don't show an error message for a device capability that has not yet been requested by the user. For example, if a social networking site has an option to include location when the user posts messages, but the user has not chosen to share location, don't show an error message when posting messages.

Guidelines for location-aware apps

Performance guidelines

This section describes several ways to ensure that your app gets the location data it needs, without spending more resources than necessary.

Use one-time location requests when updates aren't needed

Some apps need to acquire location data only once, and don't need to receive location updates. For example, an app that adds a geolocation tag to a photo or email does not need to receive location update events.

Adjust the movement threshold

Some apps need location updates only when the user has moved a large distance. For example, an app that provides local news or weather updates may not need location updates unless the user's location has changed to a different city.

Set the report interval

Most apps, other than real-time navigation apps, don't require a highly accurate constant stream of location updates. Some apps need to be updated on a regular interval. For example, a weather app may only desire a data update every 15 minutes.

Location-aware devices can track the report intervals requested by all installed apps, and provide data reports at the smallest requested interval. Then, the app with the greatest need for accuracy receives the data it needs. Therefore, your app might get updates more often than your report interval.

Note It isn't guaranteed that the location source will honor the request for the given report interval. Not all location provider devices track the report interval, but you should still provide it for those that do.

Set the desired accuracy

To help conserve power, your app should set the desired accuracy level to indicate whether your app needs high accuracy data. If no apps require high-accuracy data, the system can save power by not turning on GPS providers.

Note Set desired accuracy to HIGH if GPS data is important for your app.

Use geocoordinate accuracy

Apps that have specific accuracy requirements, such as navigation apps, should use the geocoordinate accuracy level to determine whether the available location data meets the app's requirements.

Consider startup delay

The first time an app requests location data, there might be a short delay of one to two seconds while the location provider starts up. Consider this in the design of your app's user interface. For instance, you may want to avoid blocking other tasks while you wait.

Consider background behavior

Windows Store apps won't receive location update events while suspended. If your app tracks location updates by logging them, be aware of this. When the app is resumed, it will receive only new events. It will not get any updates that occurred when it was inactive.

If your app needs location data when it is suspended, you must use a background task. Follow these guidelines when using geolocation within a background task or a background audio exception:

- Use only one-shot lookups from within background tasks.
- Do not register for position or state change notifications in your background task or in a background audio exception.

Note JavaScript background location tasks are not supported. The background task must be written in C++ or C#.

User experience guidelines

Start using the location object only when the app requires location data

The app's first access to geolocation will trigger the consent prompt. This occurs the first time an app calls **getGeopostionAsync** or registers an event handler for the **PositionChanged** event. If an app's primary purpose does not require access to geolocation, it can be confusing to the user if a prompt for permission to use the device appears as soon as the app starts. Follow these guidelines for a good user experience.

Guideline	Example
If location is not essential to your app, don't access it until the user	A social networking app has a button for "Check in with my location". This app should not access location until

specifically requests it.	the user clicks the button.
If an app requires location for its main function, then it can access the device when the app starts.	An app for placing the user on a map requires location for its main purpose. It's OK for this app to use location when it starts, to show the user's location right away.

Use the main UI thread for the first use of Geolocation

The first use of the **Geolocator** object to get location data must be made on the main UI thread, to show a consent prompt to the user. The first use of the **Geolocator** can be either the first call to **GetGeopositionAsync**, or the first registration of a handler for the PositionChanged event. The consent prompt is described further in Guidelines for using sensitive devices.

This means that:

- In an app using JavaScript, the first use of the **Geolocator** object should not be done in an activation handler.
- In an app using C# or C++, the first use of the **Geolocator** object to get location data must be on the main UI thread. The main UI thread is typically code in MainPage.xaml.cs instead of App.xaml.cs.

Provide UI for indicating accuracy

Since different location providers can provide varied accuracy, use the accuracy parameter to communicate accuracy to the user. For example, in a mapping app, a circle around the user's location can indicate the accuracy range.

Provide UI for manually entering location

Your app should provide an alternate way to enter location, if the current location data is not available.

Provide UI for manually refreshing location

Your app should provide a UI control to allow the user to refresh their current location.

Guidelines for responding to changes in location settings

- The user should be able to turn off location by using the **Settings** charm or **Control Panel**.

- To respond appropriately if the user disables or re-enables location, do the following:
    - Handle the **StatusChanged** event. The **status** property of the argument to the **StatusChanged** event has the value **Disabled** if the user turns off location.
    - Check the error codes returned from **GetGeopositionAsync**. If the user has disabled location, calls to **GetGeopositionAsync** will fail with an ACCESS_DENIED error. The **LocationStatus** property will have the value **Disabled**.

- If you have an app, such as a mapping app, for which location is essential, ensure that you do the following:
    - Handle the **PositionChanged** event to get updates if the user's location changes.
    - Handle the **StatusChanged** event as described here.

Show an appropriate error message or dialog when location is disabled or not available

When access to location data is revoked by the user, or data is not available to the app, present the user with appropriate error messages. If you need to notify the user that location is turned off, follow the error guidelines described in Guidelines for using sensitive devices, as well as these:

- We suggest this for your message: "Your location is currently turned off. Change your settings through the Settings charm to turn it back on."
- Don't let error messages interrupt the app's flow. If location is not essential for your app, display the message as inline text. Social networking or gaming apps fall into this category.
- If location is essential for your app's functionality, display the message as a flyout or a dialog. Mapping and navigation apps fall into this category.
- Don't try to programmatically launch the Settings charm.

Discard the location object and event listeners when location is disabled

The **Geolocator** object becomes non-functional when the user revokes access to location.

Remove any event listeners for the **PositionChanged** event when location is disabled.

Clear cached location when location is disabled

If your app saves or caches location data, clear any cached data when the user revokes access to location.

Provide UI for re-enabling location

Your app should provide UI for re-enabling location, such as a refresh button that reinstantiates the **Geolocator** object and reattempts to get location.

- If the user re-enables location access after disabling it, there is no notification to the app. The **status** property does not change, and there is no **StatusChanged** event. Your app should reattempt access by creating a new **Geolocator** object and calling **GetGeopositionAsync** to get updated location data, or resubscribing to **PositionChanged** events. If the status then indicates that location has been re-enabled, clear any UI that it previously displayed that notified the user that location was disabled, and respond appropriately to the new status.
- Your app should also reattempt to get location on activation, when the user explicitly attempts to use functionality that requires location, or at any other scenario-appropriate time.

Privacy considerations

A user's geographic location is personally identifiable information (PII). The Microsoft Privacy site provides guidance for protecting user privacy

Guidelines for developing using proximity

Proximity allows users to connect two devices with the human gesture of a *tap*, or by browsing for devices within wireless range. You do not need to be connected to a network. You can simply tap two computers together, or connect using Wi-Fi Direct. You can use proximity to connect apps on two computers, for a multi-player game experience or to share content, such as photos or links.

Proximity Connections

There are several ways to communicate by using **Proximity**:

Term	Description
Out-of-band sessions	You can establish a session using the **PeerFinder** object, which connects devices over an out-of-band transport (Bluetooth, Infrastructure network, or Wi-Fi Direct). While the range for a tap is limited to 3-4 centimeters, the range for out-of-band transport options is much larger. You may not need to include proximity in your app to share resources. If Windows supports your sharing scenario, then enable the Sharing contract and use built-in Windows functionality to share resources by using tapping.
Browse for Peers	You can establish a session by using the **PeerFinder.FindAllPeersAsync()** method. This method finds all remote peers that are running the same app, if they have also called the **PeerFinder.Start()** method to advertise that they are available for a peer session. Browsing for peers does not use a tap gesture, but instead uses Wi-Fi Direct to discover the remote peer establish a connection.
Publishing and subscribing for messages	You can send or receive messages during a tap gesture by using the **ProximityDevice** object.

If an app calls the **ConnectAsync** method to create a connection with a peer, the app will no longer advertise for a connection. The app will not be found by the **FindAllPeersAsync()** method until the app calls the **StreamSocket.Close** method to close the socket connection.

You will only find peers where the computer is within wireless range and the peer app is running in the foreground. If a peer app is running in the background, proximity does not advertise for peer connections.

If you open a socket connection by calling the **ConnectAsync** method, only one socket connection can be open at a time for the computer. If your app, or another app calls the **ConnectAsync** method, then the existing socket connection will be closed.

Each app on your computer can have one open connection to a peer app on another computer if that connection was established using a tap gesture. You can open socket connections from one app to a peer app on multiple computers by tapping each computer. If you create a connection by using a tap gesture, a new tap gesture does not close the existing connection. You must call the **StreamSocket.Close** method of the socket object to create a new connection to the same peer app on the same peer computer by using a tap gesture.

Best practices

The overall feel of a proximity experience should be easy, lightweight, and intuitive. Users should go through minimal setup to participate in a proximity experience. Once a proximity experience is over, exiting the experience should be as sleek as entering it. Proximity is intended for an app that just wants a connection with another instance of itself, without being concerned about the details of the connection. If an app needs constant updates about the connection (for example: bandwidth usage, speed), then do not use proximity.

Term	Description
Finding Peers	When your app browses for other peers that are running the same app, do not continuously browse for peers. Instead, provide the user an option to browse for peers within Wi-Fi range.
Ask for consent	Always ask for user consent to start a connected proximity experience, that is, to put an app in multi-user mode. While users are running an app, asking for consent should be forward and dismissible. For example, two people playing a game could be given a chance to provide consent before they decide to play together. In cases where a tap occurs as the app launches, users should be given a chance to provide consent in the start menu or lobby of the app.
Show the state of the	When a user puts an app in multi-user mode, the UI should

connection	reflect one of three connection states: • Waiting for a tap • Connecting devices (show progress) • Devices now connected, or connection failed
Revert to single-user mode after a failed connection	If a connection breaks or fails to establish, convey this information to users by putting the app back into single user mode and displaying a message to indicate that the connection has failed.
Keep the user in control	Ensure that users can easily navigate out of a proximity experience.
Proximity only creates **StreamSocket** objects for network connections	If your app requires a different type of connection object than a **StreamSocket** object, you cannot use Proximity to connect.

Guidelines for developing print-capable apps

The following table explains these recommended practices.

Don't add an in-app print button unless it is required.	This ensures that the users invoke the print experience through the Devices Charm. If printing is the natural completion of a particular workflow, it is appropriate to add an in-app print button at an appropriate location. For example, it would be practical to add a button for printing a boarding pass after an airline check-in workflow.
Do keep error messages to two lines max.	When a user makes an invalid entry in the Print window, you can display an error message. Alert the user about the invalid entry, and let them know what to do. As a best practice, we recommend that you keep the information in the error message window to a maximum of two lines.

Best practices for deciding how to create your printed content

You have a fair amount of flexibility when deciding how to generate your print content. In most cases it is easiest to print by using the same technology in which you are developing the rest of your app.

However, you may find that different printing technologies are better for different things. While you may use HTML5 and JavaScript with a custom print template, you may find that the layout functionality of XAML printing is easier to manage. Or you may discover that you want to lay out the printed content pixel by pixel, for which the Direct2D print control provides richer support.

Best practices for using a custom print template with HTML Content

A Windows Store app can use a custom print template to customize the preview and print layout of a document. The following table shows the recommended practices for using a custom print template.

Do make sure that the preview and print layout dimensions are correct.	Changes in printer settings can cause a change in paper size. Therefore, read the current values for **pageWidth** and **pageHeight** before pagination. This ensures a correct preview and print layout.
Do make sure that preview	Because you can use code to control the preview content that is

content shows up correctly in the print window.	shown to the user, you must make sure that the preview content shows correctly for different combinations of paper size and orientation values.

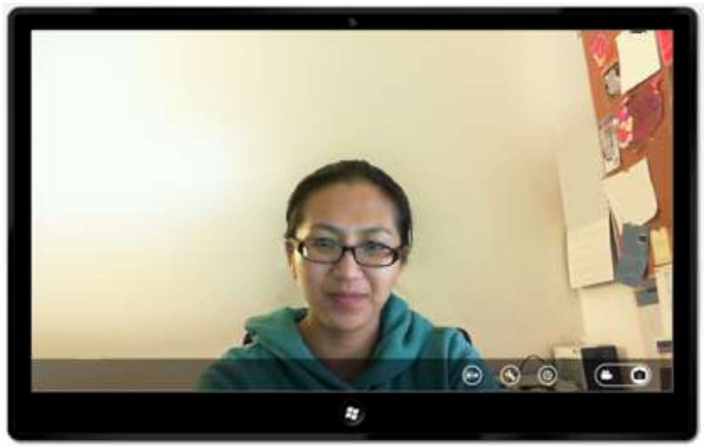
Best practices for customizing settings in the print window

Customizing the settings in the print window gives the app developer the flexibility to change how the printer settings (and options) are presented to the user. The following table explains these recommended practices.

Practice	Description
Do not change the order of the settings shown in the print window unless it is necessary.	The order of the settings shown to the user is customizable. However, to maintain consistency in the experience, you are encouraged to retain the default order of the settings and add more settings to the list if necessary. For example, the **Copies** settings are listed first in the default print experience, and users expect this listing order in your app's print experience too.
Do not add more printer settings to the print window unless it is necessary.	Some printer settings and their behavior can be specific to the printer. It is better for the printer manufacturer to handle the addition of such settings. If the printer manufacturer made printer-specific settings available for your printer, then users of your app can click **More Settings** in the print window to invoke the Windows Store device app that displays those additional settings (if this Windows store device app has been installed).

Guidelines for the camera UI

The camera dialog is a touch-optimized, full-screen experience for capturing photos and videos from an embedded or attached camera. People can crop their captured photos and trim their captured videos before returning them to your calling app. They can also adjust some of the camera settings such as brightness, contrast, and exposure.



Appropriate use of the camera UI

Use the camera UI if your app requires live photo or video. For instance, you can use the camera UI to let users take a photo of themselves for their profile pic.

Inappropriate use of the camera UI

- Don't use the camera dialog if you need to have real-time feedback or control over the image that is being captured. For example, a barcode reader might want to immediately let the user know that a pic of a bar code isn't readable. In this case the camera dialog would not be the right option because it doesn't provide direct control over the captured video stream. Use Media Capture instead.
- Don't use the camera dialog if you need to add custom controls to the user interface. Instead, use Media Capture if you need to add UI customizations beyond what the camera dialog provides.
- Don't turn on cropping or trimming in the camera dialog if your app provides those features. Video or photo editing apps, for example, should turn off trimming and cropping so that your apps features aren't redundant with what the camera dialog provides.

Guidelines for developing audio-aware apps

The following table presents the practices recommended when you add and configure media buttons in a Windows Store app.

Practice	Description
Design your button-press event handlers to respond with the least amount of delay possible.	This makes sure that the user immediately gets verification of their button input. Long delays in response cause the user to press buttons multiple times, resulting in unexpected app behavior.
Make sure that the media buttons are used in their standard ways.	This makes sure that the user has a familiar experience with the use of the media buttons.
Make sure that you do not use track and artist name strings that are longer than 127 characters.	When you configure your app to display track metadata, make sure that the strings for track name and artist name have a maximum length of 127 characters. If you use track and artist name strings that are longer than 127 characters, it will result in an error condition. If your app does not handle this error properly, it could cause the app to stop functioning.

Best practices for using playback manager in an app

The following table presents the practices recommended for configuring audio/video (AV) streams, and working with playback manager.

Practice	Description
You must only consider using the msAudioCategory assignment if you need audio to play in the background.	Audio playback drains the battery, so unless there is a clear need for background audio (media playback designed for longer term listening, for example) do not declare an audio category. Alternatively, you can use the "Other" category. Otherwise, your app will be muted and then suspended.

Use low-latency audio only when it is needed for specific apps that require it (including multi-track recorders and low-latency video capture).	Low-latency audio is automatically invoked when you select the "Communications" audio category. For any other category, consider leaving low-latency settings at their default (which is OFF). Low-latency buffers use significantly more CPU and battery resources, and are typically reserved for foreground apps on which the user is focused.
You can use ForeGroundOnly media to mute playback of background media.	If you develop a game that plays its own audio soundtrack, the soundtrack will be muted if the user was already playing an audio track in the background when the game started. If you believe that the game's audio soundtrack is essential for the function of the game, you can choose ForeGroundOnlyMedia as the msAudioCategory for this soundtrack. This mutes the currently playing background audio. SoundEffects that are mixed in with background media will still be heard in either case. ForeGroundOnlyMedia can also be selected for video apps that should stop background media when the video starts, and should not run at all when they are in the background.

Best practices for managing call control

The following table presents the practices recommended for managing call control on the default Bluetooth communications device.

Practice	Description
You must make call control functionality predictable for communications apps.	This ensures that the user has an experience that is familiar and seamless. If your app uses call buttons in an unfamiliar way, make it very obvious to the user.
Track call tokens carefully. Make sure you end the call correctly for the device, and also end the audio/video stream.	Doing this helps you make sure that when the call is completed, you send an "endcall" notification back to the device, with the appropriate call token. That way the user has an indication from the device that the call has ended.

Animations

List animations

List animations let you insert or remove single or multiple items from a collection such as a photo album or a list of search results.

Appropriate use of list animations

- Use list animations to add a single new item to an existing set of items. For example, use them when a new email arrives or a new photo is imported into an existing set.
- Use list animations to add several items to a set at one time. For example, use them when you import a set of new photos into an existing collection. The addition or deletion of multiple items together should happen all at the same time, with no delay between the action on the individual objects.

Inappropriate use of list animations

- Don't use the list animations to display or remove a container. These animations are for members of a collection or set that is already being displayed. Use one of the UI animations to show or hide a transient container on top of the app surface. Use the transition animations to display or replace a container that is part of the app surface.
- Don't use the list animations on an entire set or collection of items. Use the transition animations to add or remove an entire collection within your container.

Transition animations

Transition animations move either entire pages into or out of view or sets of content onto or out of pages.

Appropriate use of content transitions

- Use transition animations when there is a set of new items to bring into an empty container. For example, after the initial load of an app, part of the app's content might not be immediately available for display. After that content is ready to be shown, use a content transition to bring that late content into the view.
- Use transitions to replace one set of content with another set of content that already resides in the same container within a view. For example, when the user switches from a "Latest Issues" view to a "My Issues" view in this magazine app, use **enterContent** for the content replacement.

- Slide the content into the view against the general page flow or reading order. For instance, if the animation is to bring new content to a document that flows from left to right, then the incoming content should move in from right to left.
- If you have more than one container with content to be updated, trigger all of the transition animations simultaneously without any staggering or delay.

Inappropriate use of content transitions

- Don't use content transitions to bring in an entire new container. If the content's container is not visible on the screen, use one of the transient UI animations, either **showPopup** (for floating UI) or **fadeIn** (for UI on the main app surface) to bring on the container. After the container is in place, use **enterContent** for content inside that container.

Appropriate use of page transitions

- Use page transitions to bring in your initial content when your app is launched.
- Slide the content into the view against the general page flow or reading order. For instance, if the animation is to bring a new "page" in an app that flows from left to right, then the incoming content should move in from right to left.
- If there is more than one content region in your new page, choose the transition order so that new content appears in each region in the order in which it would be read. For instance, if the reading order is left to right, the content regions should be ordered top to bottom, then left to right.
- If some of the content on the incoming page is not ready to display immediately, use a page transition to bring in the empty container. Then use a content transition to bring the content into the page.

Inappropriate use of page transitions

- Don't use page transitions when there is already content on the screen. If there is already content on the page, use content transitions instead.

Drag-and-drop animations

Drag and drop animations help you create great animations when people start to move a drag-able UI item, drag the object between other UI items, and drop the UI item.

Appropriate use of drag and drop animations

- Include in the animation only those UI items that people can drag

- Allow some object movement before triggering the animation to start a drag-and-drop sequence. This prevents the user from accidentally dragging an object that they meant to only tap or select. The recommended threshold is 20 touch independent pixels (TIPs).
- When dropping an item to reorder a list, combine animations to reposition existing items in the list to make room for the item that is being dropped. For example, after using a drag and drop animation to drop an item into a list, use a list animation, like **createAddToListAnimation** or **AddDeleteThemeTransition**, to move all elements into their proper positions.
- Similarly, you can use an animation like **fadeOut** when a file is dropped into a folder to make the file disappear from the screen.
- When the user drags an object to an area where it can be dropped between two other objects, ensure that the other objects shift enough to communicate the drop target area to the user. For JavaScript apps, **dragBetweenEnter** is the preferred animation.
- Create a drop target area that is large enough. This area should not be so small that it is difficult for the user to position the drag source.

Inappropriate use of drag and drop animations

- Do not use drag and drop animations if the drag source cannot be dropped in an area.

Transient UI animations

Transient UI is meant to appear and disappear to notify users or help users complete a contextually relevant task. Common transient UI include popups and UI items that fade in and fade out.

Appropriate use of transient UI animations

- Use the popup animations for custom context menus, dialog boxes, flyouts, tooltips, or other contextual UI that not a part of app page itself.
- Use fade in and out animations to show or hide controls on the app page or inside existing containers on the page. For example, fade in a scroll bar when the cursor nears the bottom of the screen.
- Use fade in and out animations to show or hide UI elements that don't have a specific recommended animation.

Inappropriate use of transient UI animations

- Don't use pop-up animations for UI that is part of the main app surface. The pop-up animations are used to show UI that floats above the main app surface, such as a dialog box. If your UI is part of the main app surface instead of above it, use one of these alternatives:

Action	Animation
To change the contents of a UI container	Content transitions
For controls or if no other animation applies	Fade in and out transient animations

- Don't use pop-up animations for common controls provided by Windows such as the dialog, flyout, tooltip, or context menu controls. These common controls have the pop-up animations built in. You do not need to provide the animations yourself. You only need the pop-up animations if you are creating a custom control.
- Don't use fade in and out animations for UI elements that have another recommended animation. The following table shows specific JavaScript animations recommended for specific UI elements. XAML apps have similar animations. These elements should not use fade animations.

UI type	Recommended animation
Dialog box	**showPopup** and **hidePopup**
Flyout	**showPopup** and **hidePopup**
Tooltip	**showPopup** and **hidePopup**
Context menu	**showPopup** and **hidePopup**
Command bar	**showEdgeUI** and **hideEdgeUI**

Task pane or edge-based panel	**showPanel** and **hidePanel**
Contents of any UI container	**enterContent** and **exitContent**

- Common controls provided by Windows, such as dialogs, flyouts, tooltips, context menus, and the app bar, have the recommended animations already included. You do not need to provide the animations yourself.

Edge-based UI animations

Edge-based animations help bring in UI from the edge of the screen. The most common uses are for custom app bars and message bars.

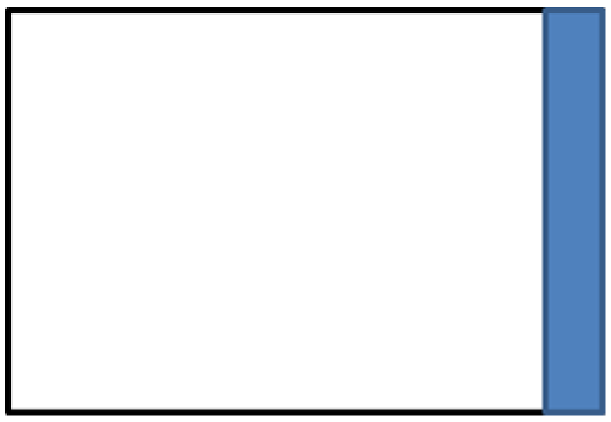
Appropriate use of edge UI and panel animations

- Use edge animations like **showEdgeUI** to show a message bar or a custom command bar that does not extend far into the screen.
- Use panel animations like **showPanel** to show UI that slides a significant distance into the screen, as in the case of a task pane or a custom keyboard.
- Slide the UI in from the same edge that it is attached to.
- Slide the UI out to the same edge that it came in from.
- If the contents of the app need to resize in response to the UI sliding in or out, use an animation like **crossFade** to create a seamless animation. For example:

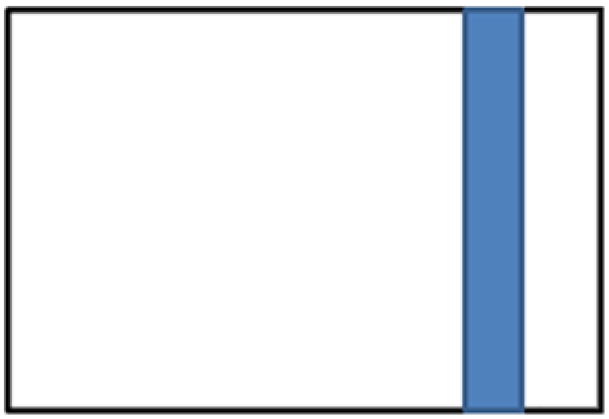
Inappropriate use of edge UI and panel animations

- Don't use edge or panel animations for any UI container or control that is not at the edge of the screen. These animations are used only for showing, resizing, and dismissing UI at the edges of the screen. To move other types of UI, use the reposition animations.

Animations



Use showEdgeUI/hide EdgeUI



Use reposition

APP ACTIVATION

People activate your app in a number of ways. They can tap on your app tile and secondary tiles, or tap on items that can be activated, like files or protocols. Tiles are like the front door to your app. You want to engage people with your tile so they are more likely to launch your app. Each time people do tap your tile, you launch your app and briefly display a splash screen so you can load their last app state by using roaming app data. That way, people are taken to right where they left off.

The guidelines in this section help you create a great experience around launching, suspending, and resuming your app from tiles and file and protocol activation.

Tiles

A tile is an app's representation on the Start screen. A tile allows you to present rich and engaging content to your user on Start when your app is not running. Tapping or clicking the tile launches the app. Tiles come in two sizes: square and wide. Several template variations are provided for each size, with text, images, or a combination of text and images. Templates can have a single frame or two stacked frames, which is called a peek template. A tile based on a peek template animates between the two frames within the tile space. Single-frame templates do not animate.

In one corner, a tile can optionally show branding, as the app's name or a small logo image.

Tiles can be live (updated through notifications) or you can leave them static. Tiles begin as a default tile, defined in the app's manifest. A static tile always displays the default content, which is generally a full-tile logo image. A live tile can update the default tile to show new content, but if the update expires or is removed, it can return to the default. In the corner opposite to its branding, a tile can also display a status badge, which can be a number or a glyph.

Two important points to always remember:

- If you use a wide tile, the user can resize the tile from wide to square or square to wide at any time. You don't know which size is displayed.
- The user can turn tile notifications off and on at any time.

User experience guidelines

This section talks about design considerations that you should keep in mind when you plan the look and use of your tiles

- Tile design philosophy
- Choosing between a square and wide tile size

- Using default tiles
- Using peek templates
- Other design considerations
- Using tile update notifications

Tile design philosophy

Your goal is to create an appealing tile for your app. If you use a live tile, your goal is to present engaging new content that the user wants to see on their Start screen, and that invites them to launch the app. To that end, avoid the overuse of loud colors. Simple, clean, elegantly designed tiles are more successful than tiles that scream for attention like a petulant child.

When designing your app, you might ask yourself "Why should I invest in a live tile?" There are several reasons:

- Tiles are the "front door" to your app. A compelling live tile can draw users into your app when your app is not running. A user increasingly values an app that they use frequently.
- A live tile is a selling point that differentiates your app from apps on operating systems that allow only static tiles and icons on their Start screen.
- A live tile is a selling point that differentiates your app from other apps in the Windows Store. Users are likely to prefer the app with the great live tile to a similar app with a static tile.
- If users like your live tile, a prominent placement of that tile in Start drives re-engagement with your app. Chance discovery of cool content in the app through the tile makes users happy.
- If users don't like your tile, they might place it at the end of Start or unpin it altogether, turn off updates, or even uninstall your app.

Some characteristics that make a live tile compelling are:

- Fresh, frequently updated content that makes users feel that your app is active even when it's not running.

  Example: Showing the latest headlines or a count of new e-mails.
- Personalized or tailored updates that use what you know about the user, such as interests that you allow the user to specify through app settings.

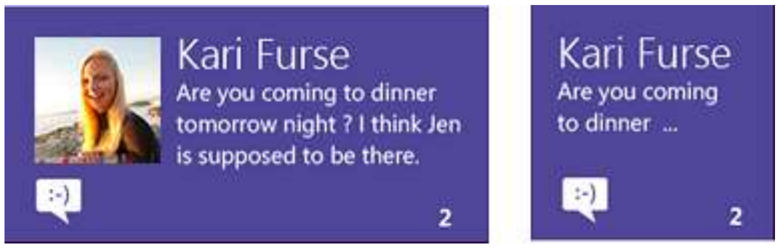
  Example: Deals of the day tailored to the user's hobbies.
- Content relevant to the user's current context.

  Example: A traffic condition app that uses the user's current location to display a relevant traffic map.

Choosing between a square and wide tile size

A square tile is required. You must decide whether you want to allow for a wide tile as well. This choice is made by providing a wide logo image when you define your default tile in your manifest. If you do not include a wide logo image, your tile is always square. The user cannot resized it and it cannot accept wide update notifications.

- Use only a square tile if your app does not use tile notifications to send updates to the user. Wide tile content should always be fresh and regularly updated. If you aren't using a live tile, do not provide a wide logo in the manifest.
- Use a square tile with a badge if your app supports only scenarios around short summary notifications. These are notifications that can be expressed through only a badge image or a single number. For instance, an SMS app that plans to use notifications to communicate only the number of new texts received would fit this scenario. Do not provide a wide logo in the manifest.
- Use the square size if your app sends updates that should not be shown in detail on the Start screen. For instance, a paystub app could simply say that a new paystub is available instead of mentioning specifics such as the amount. Do not provide a wide logo in the manifest.
- Use the wide size tile only if your app has new and interesting content to display to the user, and those notifications are updated frequently (at least weekly).
- Square tiles show less content than wide tiles, so prioritize your content. Don't try to fit everything that you can show in a wide tile into a square tile.


Kari Furse
Are you coming to dinner tomorrow night ? I think Jen is supposed to be there.
2
Kari Furse
Are you coming to dinner ...
2

If you have wide tile content that consists of an image plus text, you can use a square peek template to break the content into two frames.

Using default tiles

An app's default tile is defined in its manifest. It is static, square, or wide, and simple in design. For some apps, the default tile is all that you ever need. When an app is installed, the default tile is shown on the Start screen until that tile receives an update notification. If a wide logo image is provided, the wide tile is used. An update can expire or be explicitly removed, in which case the tile reverts to the default content until the next notification arrives.

Appropriate use of default tiles

- Use the default tile image to reflect your app's brand, essentially as a canvas for your app's logo.
- If you are including a wide logo, consider the design relationship between the wide and square tile images that you provide. Always remember that the user can display your tile as either square or wide, and can change that option at any time. Here are some general rules:
    - Center the logo horizontally in the tile.
    - Keep the same vertical placement of the logo in both the square and wide tiles, which are of equal height.
    - Include the app name at the bottom of the tile if your logo does not include it. The following examples show both situations.

  Tiles using the app name element defined in the manifest:

Tiles that include the app's name in the logo image:

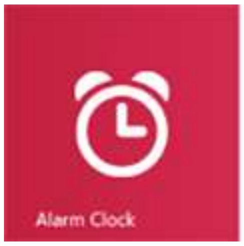
Treehouse Stampede!

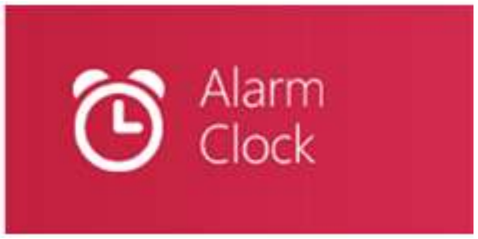
Treehouse Stampede!

  - For apps with longer names, and because the name can wrap over two lines, make sure that your logo image and the name do not overlap. A safe approach is to restrict your logo to about 80x80 pixels in the 100% image resource.
  - If you make the space around the logo itself transparent in your image, your app's brand color (declared in the manifest) shows through. A gradient is applied to it as part of the Windows 8 look. This tactic would be used with a logo such as the mail app tile shown earlier.

Inappropriate use of default tiles

- Don't design the default tile to include an explicit text call to launch the app, such as a tile that says "Click Me!"
- If your logo contains your app's name, don't repeat that name in the name field. Use one or the other, as shown here:





Using peek templates

Peek templates supply tile content which cycles between two frames of information within the tile space. The upper frame is an image or image collection, and the lower frame is text or text plus an image.

Appropriate use of peek templates

- Use peek templates if your scenario has both primary and supplementary content and contains both images and text. Good examples include notifications for an e-mail that included a photo or a news story with a picture/header/body layout.
- A peek template grabs the user's attention when it animates, so be sure that it provides desirable content. Otherwise, you will annoy your user.
- When you use a peek template, its display can start at either end (frame) of its cycle—text fully lowered or text fully raised. It can also animate up or down to the other frame. Therefore, make sure that the contents of each of your frames can stand alone.

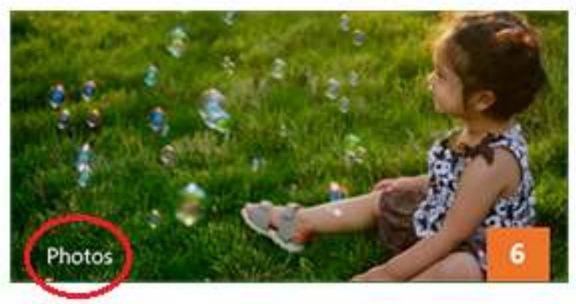
Inappropriate use of peek templates

- Don't use peek templates to display information about things the user already knows about. For example, a paused video notification shown on a tile shouldn't use a peek template.
- Don't use peek templates for notifications that are not conceptually grouped. For example, a peek template should never be used to send a news story if the photo or photos are not part of that story.
- Don't use peek templates if the most important part of your notification could be off-screen due to the peek animation. For example, a weather app may display the temperature and an accompanying image, such as a smiling sun or a cloud. Using a peek

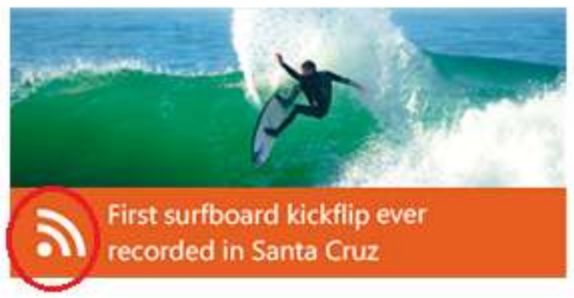
template would mean that the temperature (the purpose of the notification) isn't always visible. A static template that shows the image and temperature at the same time would be more useful to the user.

Other design considerations

- When determining how to convey an app's brand information in a tile, choose either the app's name, as shown here:

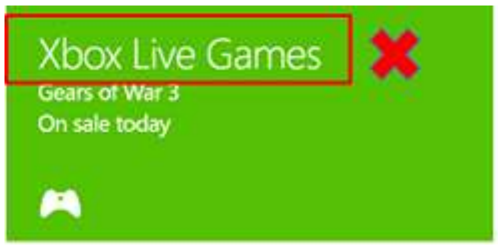
  or logo image, as shown here:

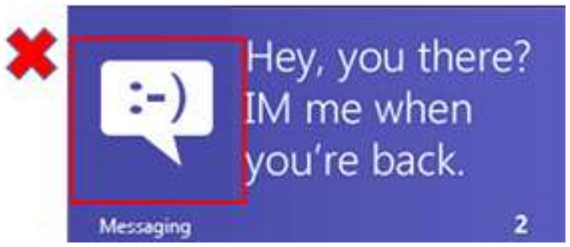
  

  These items are originally defined in the app manifest and the developer can choose which of the two to display in each subsequent notification. However, after you make the choice of name or logo, stick with it for the sake of consistency.

- Don't use the image or text elements to display app branding information in a tile update notification. To reinforce your app's brand and provide consistency to the user, branding should be provided through the template's elements that are provided for that purpose: the app name (short name), or a logo image. A live tile can change its appearance considerably from notification to notification, but the location of the name/logo is consistent. This consistency ensures that users can find their favorite apps through a quick scan, seeing that information in the same place on each tile. If your app doesn't leverage the provided branding elements (name and logo), then it can be harder for users to quickly identify your app's tile.

The following images show tiles that use the template's text and image elements to inappropriately convey branding. In both cases the tiles are also using the name or logo as designed, so the additional branding is redundant information.





- Don't use branding as one of the items in the notification queue or as one of the frames in a peek template. Both of these scenarios involve animated changes to the tile, which catches the user's eye. Calling the user's attention through an animation simply to display your brand instead of interesting new content only annoys that user.
- If your app's name does not fit in the space provided by the optional "short name", use a shorter version or a meaningful acronym. For example, you could use "Contoso Game" for the very addictive "Contoso Fun Game Version 3". Names that exceed the maximum number of pixels are truncated with an ellipsis. The maximum name length is approximately 40 English characters over two lines, but that varies with the specific letters involved. We encourage shorter app names from a design standpoint. You can also specify a longer name for your app (the "display name") in your manifest. This name is used in the "All apps" view and in the tooltip, though not on the tile.
- Don't use tiles for advertisements.
- Avoid the overuse of loud colors in tiles. Simple, clean, elegantly designed tiles are more successful than those that scream for attention like a petulant child.
- Don't use images with text on them; use a template with text fields for any text content. Text in an image does not look as sharp as rendered tile text. If an image asset isn't provided that is appropriate to the current display, the image might be scaled, which can further degrade its legibility.
- Don't rely on tiles to send urgent, real-time information to the user. For instance, a tile is not the right surface for a communication app to inform the user of an incoming call. Toast notifications are a better medium for messages of a real-time nature.
- Avoid image content that looks like a hyperlink, button, or other control. Tiles do not support those elements and the entire tile is a single click target.
- Don't use relative time stamps or dates (for instance, "two hours ago") on tile update notifications because those are static while time moves on, which makes the message inaccurate. Use an absolute date and time such as "11:00 A.M.".

Choosing the right notification method to update your tile

There are several mechanisms which can be used to update a live tile:

- Local API calls
- One-time scheduled notifications, using local content
- Push notifications, sent from a cloud server
- Periodic notifications, which pull information from a cloud server at a fixed time interval

The choice of which mechanism to use largely depends on the content you want to show and how frequently that content should be updated. Most apps will probably use a local API call to update the tile when the app is launched or the state changes within the app. This local call makes sure that the tile is up-to-date when it launches and exits. The choice of using local, push, scheduled, or polling notifications, alone or in some combination, completely depends upon the app. For example, a game can use local API calls to update the tile when a player reaches a new high score the player. At the same time, that same game app could use push notifications to send that same user new high scores achieved by their friends.

How often should your tile update? If you choose to use a live tile, consider how often the tile should be updated.

- For personalized content, such as message counts or telling whose turn it is in a game, we recommend that you update the tile as the information becomes available. This is particularly true if the user would notice that the tile content was lagging, incorrect, or missing.
- For non-personalized content, such as weather updates, we recommend that the tile be updated no more than once every 30 minutes. This frequency allows your tile to feel up-to-date without overwhelming your user.

Appropriate use of tile notifications

- Use what you know about the user to send personalized notifications to them through the tile. Tile notifications should be relevant to the user. The information you have to work with is largely internal to your particular app and could be limited by a user's privacy choices. For example, a television streaming service can show the user updates about their most-watched show. Or, a traffic condition app can use the user's current location (if the user allows that to be known) to show the most relevant map.
- Send frequent updates to the tile so the user feels that the app is connected and receiving fresh, live content. The cadence of tile notifications depends on your specific app scenario. For example, a busy social media app might update every 15 minutes, a weather app every two hours, a news app a few times a day, a daily offers app once a

day, and a magazine app monthly. If your app updates less than once a week, consider using a simple square tile with a badge to avoid the appearance of stale content.

- Provide engaging and informative tile notifications so that users can make an informed decision about whether they need to launch your app. In general, a notification is an invitation to the user to launch the app for more details or to do something. For example, a notification might cause the user to want to respond to a social media post, read a full news story, or get the details about a sale.
- Send notifications about content hosted on the home or landing page of your app. That way, when the user launches your app in response to your notification, they can easily find the content that the notification was about.

Inappropriate use of tile notifications

- Don't use live tiles if you don't have interesting, new, personalized content for the user. A calculator app, for instance, just isn't going to have that.
- Don't use live tiles if the only interesting thing to communicate is the user's last state. Utility apps, developer tools like Microsoft Visual Studio, and browsers that would only show thumbnails of the user's last session should not use live tiles.
- Don't use live tiles to spam the user or show advertisements. That will get you kicked out of the store.

Choosing a badge image

A badge can display either a number from 1-99 or a status glyph.

Note The badge catalog is not extendable. Only the Windows-provided images listed here can be used on a tile.

Numeric badges

A number from 1 to 99	1 99

Glyph badges

Status	Glyph

none	No badge shown
activity	⟳
alert	✱
available	🟢
away	🟠
busy	🔴
newMessage	✉
paused	⏸
playing	▶
unavailable	⚪
error	⊗
attention	❗

Guidelines for badges

Appropriate use of badges

- Use the square tile size with badges if your app supports only scenarios with short summary notifications. For instance, a short message service (SMS) app that plans to show only the number of new texts received.
- Display a number on your badge when the number is small enough to be meaningful in your scenario. If your badge is likely to always display a number of 50 or higher, then consider using a system glyph. Strategies to make a badge number less overwhelming include showing the count since the user last launched the app rather than the absolute count. For instance, showing the number of missed calls since the user last launched the app is more useful than showing the total number of missed calls since the app was installed.
- Use one of the provided system glyphs to indicate a change in cases where a number would be unhelpful or overwhelming. For instance, the number of new unread articles on a high volume RSS feed can be overwhelming. Instead, use the newMessage system glyph.
- Use a glyph if a number is not meaningful. For instance, if the tile shows a "paused" notification for a playlist, it should use the paused glyph because a number doesn't make any sense for this scenario.
- Use the newMessage glyph in cases where a number is ambiguous. For instance, "10" in a tile badge for social media could mean 10 new requests, 10 new messages, 10 new notifications, or some combination of them all.
- Use the newMessage glyph in high-volume scenarios, such as mail or some social media, where the tile's badge could be continually displaying the maximum value of 99. It can be overwhelming for the user to always see the maximum value and it conveys no useful information by remaining constant.

Inappropriate use of badges

- Don't repeat badge numbers elsewhere in a wide tile's body content, because the two instances could be out of sync at times.
- Don't use a glyph if what the glyph tells the user never changes. Glyphs represent notifications and transient state, not any sort of permanent branding or state.

Guidelines for secondary tiles

Only users can create a secondary tile; apps cannot create secondary tiles programmatically. Users also have explicit control over secondary tile removal, either on the Start screen or through the parent app.

Appropriate use of secondary tiles

- Secondary tiles, like all tiles on the Start screen, are dynamic outlets that should be frequently updated with new content. Secondary tiles can surface notifications and updates by using the same mechanisms as any other tile. To update the tile when the app is not running, the secondary tile must request and open a channel Uniform Resource Identifier (URI) with the Windows Push Notification Services (WNS).
- While the choice to create a secondary tile is entirely the user's, the developer determines the areas in the app that are offered to them to pin. The developer should follow these guidelines:
    - If the content in focus is already pinned, the app bar should show and enable the pin button (as the "unpin button") so that the user can use it to unpin the pinned content.
    - If the content in focus is not pinnable, the app bar pin button should not be shown. If the app exposes the pin command outside of the app bar, the app bar's pin button should either not be shown or be shown in a disabled state. The choice between disabling or not showing the pin button depends on the UI surface and scenario where the pin button appears when it is enabled.
    - Use the system-provided glyphs for pin and unpin.
    - Developers can also add contextual interactions specific to their app that create secondary tiles.
- The app should use meaningful, recreatable IDs for secondary tiles. This type of ID is important for the following reasons:
    - Users can reacquire secondary tiles when the app is installed on a second computer. When you use predictable secondary tile IDs that are meaningful to an app, it helps the app understand what to do with these tiles when they are seen in a fresh installation on a new computer.
    - At run time, the app can query whether a specific tile exists.
    - The secondary tile platform can be asked to return the set of all secondary tiles that belong to a specific app. Using meaningful IDs for these tiles can help the app to examine the set of secondary tiles and perform appropriate actions. For

instance, for a social media app, IDs could identify individual contacts for whom tiles were created.

Inappropriate use of secondary tiles

- Don't use secondary tiles as shortcuts to discrete files that cannot change, or to other static content.
- Don't use a secondary tile as a virtual command button to interact with the parent app, such as a "skip to next track" tile.

Guidelines for lock screen apps

Lock screen basics

To determine whether your app is a good candidate for a lock screen presence, you must understand the operation and limitations of the lock screen.

- A maximum of seven app badges can appear on the lock screen. The badge information reflects the badge information on the app's Start screen tile. The badge (either a glyph or a number) is accompanied by a monochrome icon (logo image) to identify the app the badge is associated with.
- Only one of those seven apps can occupy a detailed status slot, which allows it to display the text content of the app's most recent tile update.
- The lock screen's detailed status tile does not show images included in that tile update.
- The user is in charge of which apps can display information on the lock screen, and which one of those apps can display detailed status.
- All apps that have a lock screen presence can also run background tasks. All apps that can run background tasks have a lock screen presence. An app cannot use background tasks without also claiming a slot on the lock screen.
- The notification queue is not supported by the lock screen's detailed status tile. Only the latest update is shown.
- If an app has a lock screen presence and has set the **Toast Capable** option to "Yes" in its manifest, it displays its received toast notifications on the lock screen when the lock screen is showing. Toast shown on the lock screen is identical to toast shown elsewhere.
- Tile updates, badge updates, and toast notifications are not designed for or sent to the lock screen. You, as the sender, don't know if the device is locked. For an app with a lock screen presence, any notification is reflected both on the Start screen and on the lock screen.

Characteristics of a good lock screen presence

The only way that your app can have a lock screen presence is if the user gives their explicit permission. They can give this permission either in response to a request from your app (and you can ask only once), or manually through the **Personalize** page of **PC Settings**. By giving that permission, the user declares that the information coming from your app is important to them. Your app must then be worthy. Therefore, you must consider whether your app is a good candidate to have a lock screen presence at all.

A good candidate for a lock screen presence has these attributes:

- The information is quickly digestible
- The information is always up-to-date
- The information is understandable without additional context
- The information should be personal and useful to the user
- The information should only display when there is a change
- Only toast notifications should play a sound on arrival

The information is quickly digestible

If the lock screen is displayed, the user isn't currently interacting with the device. Therefore, any update information that your app displays on the lock screen should be something that the user can take in and understand at a glance. As an analogy, think of an incoming call on a cell phone. You glance at the phone to see who's calling and either answer or let it go to voice mail. Information displayed on the lock screen should be as easy to take in and deal with as the cell phone display. All of the other characteristics support this one.

The information is always up-to-date

Good badge updates, tile updates, and toast notifications, whether they're shown on the Start screen or the lock screen, are all potentially actionable. Based on the information those notifications provide, the user can decide whether they want to launch the app in response. For example, to read a new e-mail or comment on a social media post. From the lock screen, that also means unlocking the device. Therefore, the information needs to be up-to-date so that the user is making an informed decision. If users begin to see that your app's information on the lock screen is not up-to-date, then you've lost their trust. They'll probably find a more reliably informative app to occupy that lock screen slot.

Good examples: up-to-date information

- A messaging app sends a notification when a new message arrives. If that notification is ignored, the app updates its badge with a count of missed messages. If the user is present, they can turn on the screen to assess the importance of the message, and choose to respond promptly or let it wait. If the user isn't present, they will see an accurate count of missed messages when they return.

- A mail app uses its badge to display a count of its unread mail. It updates the badge immediately when a new mail arrives. A user can quickly turn on their screen to check how many unread emails they have, and they can be assured that the count is accurate. They have the information to decide if they want to unlock their device and read mail.

Bad examples: out-of-date information

- A messaging app updates its badge with its count of missed messages only once every half hour. The user can't rely on the badge count in deciding whether they want to unlock the device.
- A weather app that uses the detailed status slot continues to show a severe weather alert after the alert has expired. This gives the user incorrect information. And, if the text specifies when the alert ends, it makes it obvious to the user that this is old information. The user loses confidence that the app is capable of keeping them properly informed. The app should have cleared this information when it expired.
- A calendar app continues to display an appointment that has passed. Again, the app should have cleared this information when it expired.

The information is understandable without more context

This contextual information is not present on the lock screen:

- The tile that goes with the badge, when the app is not allowed to display detailed status. Even when detailed status is shown, the badge is physically separate from the tile. The logo image next to a badge is the only identification of the app it represents.
- Images in tile updates. Only the text portion of the update is shown in the detailed status slot.
- The notification queue. Only the most recent update is shown in the detailed status slot.

Therefore, your updates must be understandable to the user without the additional context available to you on the Start screen. Again, keep in mind that notifications cannot be targeted at the lock screen. Therefore, all of your app's update communications must fall under the "understandable without more context" rule.

Note Unlike the detailed tile, toast includes both image (if present) and text. Toast displayed on the lock screen is identical to toast displayed elsewhere, so it does not lose context.

Good examples: understandable without more context

- A mail app uses its badge to display the count of its unread mail. Its Start screen tile might display more information, such as text snippets from the most recent mails or pictures of the senders. But what the badge communicates is understandable without that extra information.
- A social networking app uses the detailed status slot to inform the user of their friends' recent activity. When a friend sends them a message, that friend's name is included in the notification text (for instance "Kyle sent you a new message!"). On the Start screen, the user can see a rich experience with their friend's picture in the update notification. Meanwhile, on the lock screen, even though there is no image, the text still makes it clear who sent the message.

Bad examples: not understandable without more context

- A messaging app updates its tile with the latest received message, and shows only the sender's picture and the message text. On the Start screen, it is clear to the user who the message is from. In the lock screen, without the sender's picture, the user has no idea who sent the message.
- A social networking app updates its tile with a collage of photos, with no text. On the Start screen this is a pleasant, lively tile. On the lock screen, because there is no text in the tile update, nothing is displayed at all.

The information should be personal and useful to the user

Two of the main purposes of the lock screen are to provide a personalized surface for the user and to display app updates. Consider both of these purposes when you judge whether your app is a good candidate for a lock screen presence.

Apps with a lock screen presence are special—only seven can ever be on the lock screen at a time. By giving an app one of those precious lock screen slots, the user is stating that information coming from that app is important enough to be seen even when the user isn't actively using their device. Therefore, the app should provide information that is both personal and useful to the user.

Note By definition, the lock screen is displayed when the device is locked. No login or other security hurdle is required to see the contents of the lock screen. Therefore, while the information displayed there is ideally personalized, keep in mind that anyone can see it.

Good examples: information personalized to the user

- A mail app displays the number of unread emails in the user's account.
- A messaging app displays the number of missed messages sent to the user.

- A news app displays the number of new articles in categories that a user has flagged as favorites.

Bad examples: impersonal information

- A news app displays the total number of new articles that come from its service, not taking into account the user's stated preferences.
- A shopping app sends a notification about a sale, but not based on any item or category preference that the user has given.

The information should display only when there is a change

As we said earlier, the goal is that information on the lock screen can be taken in at a glance. To that end, if an app is not currently displaying a badge, a gap is left on the lock screen where that badge would otherwise appear. This gap increases the ability of a user to notice something that needs their attention. The appearance of a badge and logo following an event is more noticeable than if it has been there all along.

Do not show status simply for the sake of showing status. Long-running or never-changing status just clutters the lock screen, obscuring more important information. A badge should display only when something has happened that the user should be aware of. The same is true for a tile update. Remove stale notification content from your tile, which causes the tile to revert to its default image on the Start screen and displays nothing on the lock screen.

Good examples: information displayed only when it's useful

- A mail app displays a badge only when there is unread mail. When new mail arrives, its badge is updated and shown.
- A messaging app displays its connection status only when the user is unable to receive messages. A "connected" status is the assumed default state of the app, so there is no point in conveying that information. "Everything is fine" is not an actionable notification. However, informing the user when they cannot receive messages is useful, actionable information.

Bad examples: long-running status

- A mail or messaging app has no count of unread mail to display and so shows a connection status until new mail or messages arrive. This ongoing display of connection status decreases the user's ability to see at a glance whether they have a new message, because the badge is always present.

- A calendar app displays a message that states that the user has no appointments. Again, this decreases the at-a-glance usability of the detailed status slot, since something is always displayed there.

Only toast notifications should play a sound on arrival

Do not include code in your app that plays a sound when your badge or tile updates. However, an arriving toast can play a sound as it is designed to do.

By following the guidance described in this article, you will be able to create apps that display the right information in the right way on the lock screen. You can thereby increase user satisfaction and confidence in your app.

When to use the lock screen request API

Call the lock screen request API (RequestAccessAsync) only if your app truly needs background privileges to function properly. Because there are only seven background slots available, users must distinguish which apps truly need background privileges to function properly and which work fine without them (even if they might gain additional functionality with the privileges).

If an app absolutely requires background privileges to meet user expectations, we recommend that it uses the request API to prompt the user to place the app on the lock screen.

However, if an app meets user expectations without having background privileges, we recommend that you do not explicitly prompt the user to place the app on the lock screen. Instead, let the user place their app on the lock screen through the **Personalize** page of **PC Settings**.

Examples of apps that should call the request API:

- A messaging app that requires background privileges to receive messages when the app is not in the foreground
- A mail app that requires background privileges to sync the user's inbox when the app is not in the foreground

Examples of apps that should not call the request API:

- A weather app that uses periodic notifications rather than background activity to update its forecast
- A news app that refreshes its badge count of new articles at a specific time of day

Basic app suspend and resume

Design your Windows Store app to suspend when the user switches away from it, and resume when the user switches back to it. Carefully consider the purpose and usage patterns of your app to ensure that your user has the best experience possible when your app is suspended and resumed. Most Windows Store apps should stop what they are doing when the user switches away from them. Very few apps should continue to run after the user switches away. For example, a music player should continue to play music even after the user switches to another app.

Use these guidelines to help you design the suspend and resume behavior of your app.

Do	Generally, resume your app as the user left it rather than starting it fresh It's a poor experience for users to have to start fresh every time they switch away to another app and back to your app. Examples of situations where you should resume the app as the user left it: • Web browsing session • Shopping cart • Unfinished e-mail • Paused movie or game
	Start the app fresh if a long period of time has elapsed since the user last accessed it If there's a good chance that users won't remember or care about what was happening when they last saw your app, launch the app from its default launch state. You must determine an appropriate period after which your app should start fresh. If there is any doubt about whether to resume or start fresh, you should resume the app right where the user left off. Examples of situations where it's better to start fresh: • Newsreader where the user would be brought back to a very old or stale article • Weather app where the data is stale
	Save app data when the app is being suspended Explicitly saving your app data helps ensure that the user can resume your app even if Windows terminates or suspends it. Suspended apps don't receive notification that they are being terminated. So, 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.
	Release exclusive resources and file handles when the app is being suspended Explicitly releasing exclusive resources and file handles helps ensure that other apps can access them while your app isn't using them. Suspended apps don't receive notification that they are being terminated. So, it's a best practice to have your app release its handles when it's suspended and open its handles when it's launched after termination. Examples of exclusive resources are webcams, I/O devices, external devices, and network resources.
	When resuming your app after it was suspended, update the UI if the content has changed since it was last visible to the user To the user, it should appear as though the app was running in the background.
	When resuming your app after it was terminated, use the saved app data to restore your app Users expect to find the app as they left it, whether it was suspended or terminated by Windows or closed by the user.
	Provide users with options if you can't predict whether they want to resume or start fresh It might not always make sense to bring the app back to where it left off. Instead, provide the user with a set of options for what to do next. For example, when the user switches back to your game, you could display a prompt so the user can decide whether to resume the game or start a new one.

Use these guidelines to avoid creating a poor user experience.

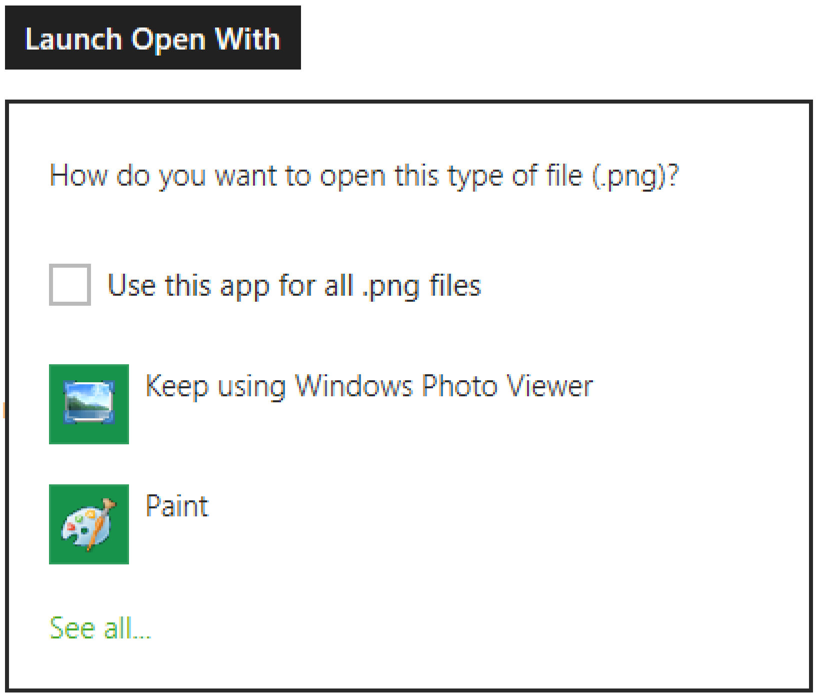
Don't	Don't terminate the app when it's moved off screen Windows ensures that there is a consistent way for the user to access and manage Windows Store apps. Your app is suspended when it's moved off screen. By leaving the app lifecycle to Windows, you ensure that your user can return to your app as efficiently as possible. Doing so also provides the best system performance and

	battery life from the device.
	Don't restore state for an app that was terminated as the result of a crash If your app was terminated unexpectedly, assume that stored app data is possibly corrupt. The app should not try to resume but rather start fresh. Otherwise, restoring corrupt app data could lead to an endless cycle of activation, crash, and termination.
	Don't offer users ways to close or terminate your app in its UI Users should feel confident that Windows is managing their apps for them. Windows Store apps should not display Close buttons or other ways to exit the app. Windows can terminate your app to ensure the best system performance and reliability. Also, users can choose to close apps through a gesture.

Launch with file types and protocols

When opening a file or protocol, the user sometimes needs to use the **Open With** list to select which app to use as the default. Windows 8 implements this list as a Flyout. Although you can't customize the contents of the **Open With** Flyout, you can control its position in your app. Be sure to follow the guidelines and position the Flyout near its point of invocation whenever possible.

Here's an example of an ideal way to use the Flyout. Notice that it's located right next to the button that called it.


Launch Open With
How do you want to open this type of file (.png)?
Use this app for all .png files
Keep using Windows Photo Viewer
Paint
See all...

You can present files and protocols however you see fit—typically as a thumbnail or a hyperlink. The primary action for these items should be **Open**. This option should invoke the default handler for the file or protocol, which might result in showing the **Open With** Flyout. (We recommend that you assume that the Flyout appears in some cases and position it accordingly.)

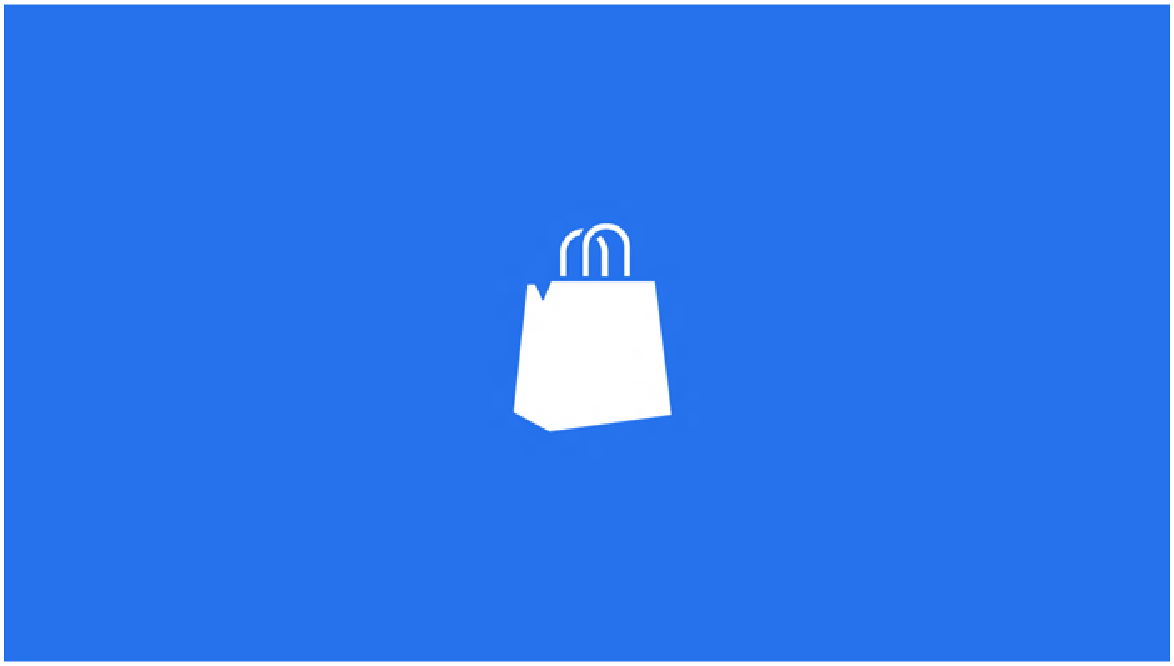
If you choose to implement any secondary actions for files or protocols in your app, such as Save As or Download, consider letting the user choose an alternate app from an **Open With** Flyout.

Remember, Windows Store apps can't set, change, or query default apps for file types and protocols, so you shouldn't try to add that functionality to your app.

Splash screens

Every Windows Store app must have a splash screen, which consists of a splash screen image and a background color. You can customize both of these features. Windows displays this splash screen immediately when the user launches an app. This quick display of the screen provides immediate feedback to users while app resources are initialized. As soon as your app is ready for interaction, Windows dismisses the splash screen.

A well designed splash screen can make your app more inviting. The Splash screen sample uses this simple, understated splash screen, shown here at 75% scale:



This splash screen is created by combining a blue background color with a transparent PNG.

Additionally, you can customize your app's launch experience by extending the splash screen and triggering entrance animations.

Appropriate use of the splash screen

- Customize the splash screen to differentiate your app.

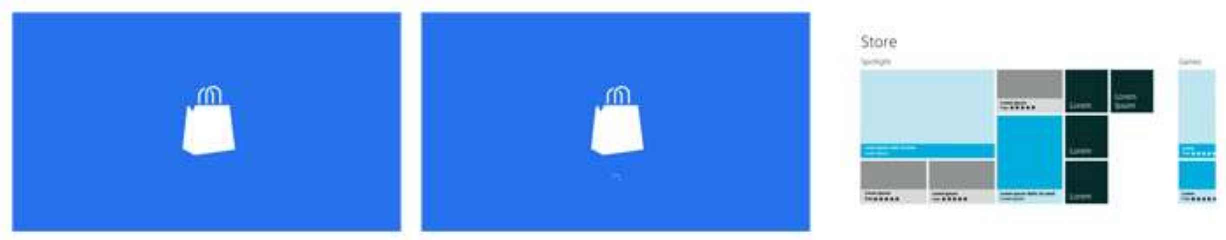
  Your splash screen consists of an image and a background color, both of which you can customize. A well designed splash screen can make your app more inviting.

  Putting an image and background color together to form the splash screen helps the splash screen look good, on any device form factor. When the splash screen is displayed, only the size of the background changes to compensate for a variety of screen sizes. Your image always remains intact.

- Create an extended splash screen so that you can complete more tasks before showing your app's landing page.

  You can further control the loading experience of your app by creating an extended splash screen that imitates the splash screen displayed by Windows. By imitating the splash screen displayed by the system, you can construct a smooth and informative loading experience for your users. For example, your app may need more time to prepare its UI or load network data. You can then use your extended splash screen to display more information as your app completes those tasks.

  You can see an example of the users' loading experience in these images, progressing from left to right. The screen shots show the transition from the splash screen displayed by Windows, to the app's extended splash screen, and lastly to the app's landing page.



To learn how to use the **SplashScreen** class to create and show an extended splash screen, see How to extend the splash screen.

Tip If you use fragment loading to load your extended splash screen, you see a flicker between the time when the Windows splash screen is dismissed and when your extended splash screen is displayed. You see this flicker because fragment loading begins to load your extended splash screen asynchronously, before the **activated** event handler finishes executing. You can avoid this unsightly flicker entirely by using the design pattern demonstrated by the Splash screen sample. Instead of loading the extended splash screen as fragments, you paint it on top of the app's UI. When your additional loading tasks are complete, you can then stop displaying your extended splash screen to reveal your app's landing page. Alternatively, you may wish to continue loading your extended splash screen as a fragment. In that case you can prevent the flicker by getting an activation deferral and responding to **activated** events asynchronously. Get a deferral for an activated event by calling the **activatedOperation**.**getDeferral** method.

Inappropriate use of the splash screen

- Don't use the splash screen or your extended splash screen to display advertisements.

  The purpose of the splash screen is to let users know, while the app is loading, that the app they wanted to start is starting. Introducing foreign elements into the splash screen reduces the user's confidence that they launched the correct app and makes the app harder to identify at a glance.

- Don't use your extended splash screen as a mechanism to display multiple, different splash screen images.

  The purpose of the splash screen and extended splash screen is to provide a smooth, polished loading experience for your users. Using your extended splash screen to display multiple, different splash screen images distracts from this purpose and could be jarring or confusing to your users. Instead, your extended splash screen should continue the current loading experience only while other tasks are completed.

- Don't use the splash screen or your extended splash screen to display an "about" page.

  The splash screen should not show version information or other app metadata. Display this information in your app's Windows Store description or within the app itself.

User experience guidelines: splash screen

Use the following guidelines to help you customize the splash screen that Windows displays for your app.

- Use an image that clearly identifies your app.

  Use an image and color scheme that clearly identify your app, so that users are confident that they launched the correct app. Making a unique screen also helps reinforce your brand.

- Use a transparent PNG as your splash screen image for best visual results.

  Using a transparent PNG lets the background color you chose show through your splash screen image. Otherwise, if the image has a different background color, your splash screen may look disjointed and unappealing.

- Provide a version of your splash screen image that is sized for all three scale factors.

  All apps must have a splash screen image that is 620 x 300 pixels, for when the device uses 1x scaling. We also recommend that you include more splash screen images for 1.4x and 1.8x scaling. Providing images for all three scale factors helps you create a clean and consistent launch experience across different devices.

Use the following table to determine the required size the splash screen image for each scale factor:

Scale	Image size (pixels)
1x	620 x 300
1.4x	868 x 420
1.8x	1116 x 540

- Select an image that fills the space allotted for the splash screen image (for each scale factor).

  This helps you produce a splash screen that looks professional and has a positive impact on your users' loading experience.

- Choose an image that uses the area allotted by the system for the splash screen image.

  When you choose a splash screen image, try to take advantage of the space allotted at each scale factor. To determine the size of the splash screen image for each scale factor, refer to the scale and image size table.

  This helps you produce a high-quality splash screen, by ensuring the quality of the image.

- Show system and event-related UI after the splash screen is dismissed.

  You can determine when it is safe to show system or event-related UI by listening for the splash screen **dismissed** event. Otherwise, the associated UI (like the search pane, a message dialog, or web authentication broker) might show while the splash screen is displayed. This might cause unwanted visual effects.

- Start entrance animations after the splash screen is dismissed.

  Many apps wish to show content entrance animations each time the app's landing page is loaded. You can determine when to start your animations by listening for the splash screen **dismissed** event.

User experience guidelines: extended splash screen

Use the following guidelines to help you create an extended splash screen that your app displays while it completes additional loading tasks.

- Make sure that your extended splash screen looks like the splash screen that Windows displays.

  Your extended splash screen should use the same background color and image as the Windows splash screen. Use a consistent image and background color to ensure that the transition from the Windows splash screen to your app's extended splash screen looks professional and is not jarring for your users.

- Position your extended splash screen image at the coordinates where Windows displayed the splash screen image.
- Adjust the position of your extended splash screen to respond to resize events like snap and rotation.

  Your extended splash screen should adjust the coordinates of its splash screen image, if the app is snapped or the device is rotated, by listening for the **onresize** event. This resizing helps ensure that your app's loading experience looks smooth and professional, regardless of how your users manipulate their device or change the layout of apps on their screen.

- If you show your extended splash screen for more than a few seconds, add a progress ring so users know that your app is still loading.

  Use the indeterminate progress ring control to help make your app seem more responsive, by letting users know that your app hasn’t crashed and will be ready soon. Additionally, consider using this control to display a single line of text alongside the progress ring, to briefly explain to your users what your app is doing.

  Making your app seem more responsive and keeping your users informed is a great way to create a positive loading experience for users.

Roaming app data

Windows 8 automatically transitions certain app data between user devices. No heavy lifting required from the app developer. Roaming app data provides a great end-user experience for apps where the user utilizes more than one device, such as a PC at work and a slate at home. To include app data roaming where appropriate, follow these guidelines when you design your app.

The user experience

If a user installs your app on a second device after using it on another device first, all settings and preferences made on the first device are automatically applied and made available on the second device. This provides a desirable user experience that involves a minimum of setup work for your app on the user's second device, since everything is already configured according to user preference. Any future changes to these settings and preferences will also transition automatically, providing a uniform experience across all devices, even if devices are being upgraded or replaced over time. Consider this example scenario: Peter just bought a new Windows 8 slate and opens his favorite calendar app. He is delighted to find that all his calendar accounts are already configured, and they show his calendar appointments in his familiar color preferences that he established on his laptop.

In addition to settings and preferences, Windows 8 also transitions session or state information. This allows end-users to continue to use an app session that was closed or abandoned on one device, when they transfer to a secondary device. Consider this example scenario: Susie was playing her favorite puzzle game just before she headed out to work. She takes out her Windows 8 slate on the bus, opens the puzzle game, and can continue playing from her last game state, which includes her new high score.

Design guidelines

Utilizing roaming app data in your app is easy and does not require significant changes to code. It is best to utilize roaming app data for all size-bound data and settings that are used to preserve a user’s settings preferences as well as app session state. To make sure that your app makes the best use of this feature, follow these design guidelines.

DO

Do use roaming for preferences and customizations Roam any app data that the end-user would choose to set on each device, such as user preferences. This could include info such as:

- Favorite sports team (sports news app)
- Favorite movie genre (media app)
- Background color customization
- App view preferences

Do use roaming to let users continue a task across devices

Roam any app data that enables users to pick up a task right where they left off on a different device. This could include information like:

- Last position in the app context (e.g. a page number in book-reader app)
- High score or game level information
- Navigation back stack
- Composing a to-do list
- Composing email

DON'T

Use these guidelines to avoid creating a poor user experience.

Don’t use roaming for information that is local to a device

Sometimes, there is information that is meaningful only on a specific device – for example, a path name to a local file resource on a PC. This information should not be part of roaming app data and must remain local to a device. You may still decide to roam local information. But ensure that the app is capable of gracefully recovering, in case the information is not valid on the secondary device.

Don't use roaming to move large datasets

There is a limit to the size of app data that each app can roam. If an app hits this limit, none of its app data can roam until the app’s total roamed app data is less than the limit again. For this reason, it is best to restrict roaming to user preferences, links, and small data files. As part of your app design it is important to consider how to put a bound on larger data, to not exceed the upper limit. For example, if saving a game state requires 10 KB per game, the app might only allow the user store up to 10 games.

Don't use roaming for instant syncing or for frequently changing information

Windows roams app data opportunistically and doesn't guarantee an instant sync. In scenarios where a user is offline or on a high latency network, roaming could be delayed significantly. Don't build a UI that depends on a sync to occur instantly. If your app frequently changes information – for example, with up-to-the-second position in a song or movie – do not use roaming app data. Instead pick a less frequent representation that still provides good user experience. For example, use the current song being played, or the current movie chapter being played. For important, time-critical settings, a special high priority settings unit is available that provides more frequent updates.

Additional considerations

Pre-requisites

Any user can benefit from roaming app data as long as they are using a Microsoft Account to log on to their device. App data must be written via the proper mechanisms for transition between devices. Apps can transition data between any devices that utilize the same Microsoft Account Connected account. Users or Group Policy Administrators can switch off roaming app data on a device altogether. Users who do not utilize a Microsoft Account, or operate devices where roaming app data has been switched off, can still use your app. But any app data will stay local to each device.

Device trust

Data stored in the credential vault will transition only if a user has made a device “trusted”.

Conflict resolution

Roaming app data is not intended for simultaneous use of apps on more than one device at a time, as it could lead to undesired and unexpected situations. In case a particular data unit has been changed on two devices, causing a conflict in the following synchronization, the system always favors the value that was written last. This ensures that the app utilizes the most up-to-date information. If settings use a settings container or composites, the conflict resolution occurs on the level of the container or composite. Then, the container or composite with the latest change is transitioned.

The right time for writing data

Depending on the expected life-time of the setting, data should be written at different times. Infrequent, slowly changing app data should be written immediately. On the other hand, app data that changes frequently should be written only periodically, at regular intervals like once every 5 minutes, and when the app is suspended. For example, a music app might write the "current song" settings whenever a new song starts to play. However, the actual position in the song should be written only on suspend.

The data changed event

Since roaming app data could change at any time, the system provides the developer with a Data Changed event. To properly use roaming app data, the app must listen to this event and then take the appropriate action to update with the current data.

Excessive usage protection

The system has various protection mechanisms in place to avoid inappropriate use of resources. In case app data does not transition as expected, it is likely that the device has been temporarily restricted. Waiting for some time usually resolves this situation automatically and no action is required.

Versioning

App data can utilize versioning of app data to upgrade from one data structure to another. The version number is different from the app version and can be set at will. You should use only increasing version numbers, since data loss occurs when an app transitions to a lower data version number that represents newer data. App data roams between apps only when they have the same version number. For example, devices on version 2 can transition data between each other, and devices on version 3 do the same. But there is no automatic transition between version 2 and version 3 devices. It is the responsibility of the app to track version number at the time of a version-number update. A newly installed app that has previously used various version numbers on other devices starts with app data that has the highest version number available.

Testing and tools

Developers can lock their device in order to trigger a synchronization of roaming app data. If it seems that the app data does not transition within a certain time frame, check the following items and make sure that:

- Your roaming app data does not exceed the maximum size.
- Your files are closed and released properly.
- There are at least two devices with the same data version for the app.

OTHER USER EXPERIENCES

Notifications

Appropriate use of toast notifications

- Notify the user of something personally relevant and time sensitive. Examples include:
    - New emails in a mail app
    - An incoming VOIP call
    - A new instant message
    - A new text message
    - A calendar appointment or other reminder
    - Notifications for which the user has explicitly opted in
- Navigate to an appropriate destination in your app when the user taps on a toast. Consider that notifications are an invitation to switch context rather than a strictly informational update.
- Consider that the user might not see a particular toast notification. Provide alternate ways for the user to get the same information if it is important. You may want to retain related information on your tile or within your app views.
- A running app can hide a toast notification if it is no longer valid. This happens, for example, with an incoming call where the other party has hung up or has already answered on another device.
- Combine multiple related updates that occur within a short period into a single toast notification. For instance, if you have three new e-mails that arrive at the same time, the app or app server should raise a single notification that states that there are three new e-mails, rather than three separate notifications.
- Present information in the simplest possible form. If your content doesn't require a headline, omit it. A message such as "Your download is complete." is complete and needs no additional presentation.
- Use images when they add clear value to the message, such as a photo of the sender of a message.

Inappropriate use of toast notifications

- Don't use toast notifications to notify the user of something that must be seen, such as a critical alert. To ensure that the user has seen your message, notify them in the context of your app with a Flyout, dialog, app bar, or other inline element.
- Don't include text that tells the user to "click here to..." It is assumed that all toast notifications have a click or tap action with a result that is made clear by the context of the notification.
- Don't use toast notifications to notify the user of transient failures or network events, such as a dropped connection.
- Don't notify the user of something they didn't ask to be notified about. For instance, don't assume that all users want to be notified each time one of their contacts appears online.
- Don't use toast notifications for anything with a high volume of notifications, such as stock price information.
- Don't use toast notifications to notify the user of routine maintenance events, such as the completion of an anti-virus scan.
- Don't raise a toast notification when your application is in the foreground. In that case, consider notifying the user in the context of your app with a Flyout, dialog, app bar, or other inline element. Listen for the **PushNotificationReceived** event to intercept push notifications when your application is running.
- Don't add generic images such as icons or your app logo in the image field of a notification.
- Don't place your app's name in the text of the notification. Users identify your application by your app's logo, which is automatically included in the toast notification.
- Don't use your app to ask users to enable toast notifications if they have chosen to disable them. Your app is expected to work without toast notifications.
- Don't automatically migrate your balloon notification scenarios to toast—consider that it may be more appropriate to notify the user when they aren't immersed in a full-screen experience (desktop style apps only).
- Don't use toast notifications for non-real time information, such as a picture of the day.
- Don't hide toast notifications unless necessary.

Guidelines for scheduled notifications

- Follow the recommendations in Guidelines for tiles and Guidelines for toast notifications. Consider that guidance when planning the content of your tile or toast notification, as well as how frequently each should be updated.
- Consider using **background tasks** to update the schedule periodically using the **MaintenanceTrigger** class. For example, your app can initially schedule notifications a week in advance and then use the **MaintenanceTrigger** class to continue to schedule successive weeks on an ongoing basis, even without the user launching your app during any given week.
- Consider using a **timeZoneChange** system trigger to respond to changes to the system clock, such as Daylight Savings Time. By default, scheduled notifications are triggered in Coordinated Universal Time (UTC) and are not updated automatically in response to system clock changes. For example, a reminder app would need to change the scheduled time of the reminders when the system time changes. To do so, your app can use a background task that responds to the **timeZoneChange** trigger, adjusting its timing appropriately.

Guidelines for periodic notifications

- Call the **StartPeriodicUpdate** or **StartPeriodicUpdateBatch** method each time your app is launched or brought into focus. These calls ensure that the tile content is updated each time the user launches or switches to the app.
- Follow the UX guidelines for tiles and badges when you consider what to place on the tile and how frequently the tile should be updated.
- Periodic tile updates support the tile notification tag used with the notification queue. When using **TileUpdater.startPeriodicUpdateBatch**, your service can set the tag on each notification by providing the X-WNS-Tag HTTP response header.
- Update your tile and badge XML content on your web service to match the polling frequency of your client. For instance, if the tile is set to poll at half-hour intervals, it is also a best practice to update the content on your web service every half an hour.
- Set the expiration on your tile or badge update to match the time period after which your notification would no longer be useful to the user. By default, all polled tile and badge content expires three days after the client receives them. But, if your cloud service becomes unreachable or the user disconnects from the network for an extended period of time, the content on your tile should not persist on the Start screen when it is no longer relevant. For example, a shopping deal that expires at midnight should set its expiration time to midnight.
- Do not use periodic updates for content that the user will expect to receive immediately, such as breaking news or weather alerts. Notifications of that type are best delivered through push notifications.

Guidelines for push notifications

- **Follow the overall tile and toast notification guidelines**. Whether a tile or toast notification is generated locally or through the cloud, it should respect the same user guidelines.
- **Register your app in the Dashboard**. You cannot use Windows Push Notification Services (WNS) unless your app is registered with the Dashboard. Your app server has to use the specific credentials provided by the Dashboard to authenticate and send notifications.
- **Respect your user's battery life**. Your users can receive notifications at any time, even when their device is in a low-power state. The more notifications that you send, the more resources it requires and the more frequently you wake up the device. Keep this battery impact in mind when you determine the frequency of your notifications—increasing the frequency of your notifications does not always increase the value of your app. For example, if your tile content is updated too frequently, some of your updates will never be seen by the user. Developers should choose the lowest frequency of notifications that will still achieve a great user experience.
- **Do not use push notifications for spam or with malicious intent**. If an app sends overly frequent notifications simply to be obnoxious or monopolize bandwidth, or sends notifications that are considered spam, WNS reserves the right to protect its users. WNS can block selected apps from using push notifications. In addition, if users report that an app is exhibiting malicious intent, that app could be subjected to Windows Store removal policies.
- **Implement channel renewal**. Channel URLs can expire and are not guaranteed to remain the same each time you request one. Therefore, your app should request a channel each time the app launches. If the returned channel URL is different than the URL that you had been using, update your reference in your app server.
- **Reuse your access token**. Your access token can be used to send multiple notifications. Therefore, your server should cache the access token so that it does not need to reauthenticate each time it wants to send a notification. If the token has expired, your app server receives an error. You should then authenticate your app server and retry the notification.
- **Be aware that WNS has no delivery guarantees**. Although WNS maintains high levels of availability and reliability, ultimately, the delivery of notifications cannot be guaranteed. Your app should not use WNS for critical notifications.

Security Considerations

- **Do not send confidential or sensitive data through push notifications**. Push notifications are not intended to carry highly sensitive or confidential information. For example, a bank account number or password should never be sent in a notification.
- **Always secure your channel registration callback to your app server**. When your app receives its channel URL and sends it to your app server, it should send that information securely. We recommend that the mechanism used to do so be authenticated and encrypted.
- **Validate that the channel URL is from WNS**. Never attempt to push a notification to a service that is not WNS. Always ensure that your channel URLs use the Windows.com domain.
- **Keep your app server credentials a secret**. Never share your Package Security Identifier (PKSID) and secret key with anyone. Store those credentials on your app server in a secure manner. If you believe that your secret key has been compromised, generate a new key. We recommended that you routinely generate a new secret key to present villains with a moving target.

Microsoft account sign-in

Your app can sign users in and out with their Microsoft accounts to access data in Microsoft cloud services like Hotmail, Microsoft SkyDrive, and Windows Live Messenger. If your app does this, make sure that it follows these user-experience guidelines.

Signing users in

To sign a user in, use one of the following approaches, depending on when your app needs the user to sign in.

- If your app doesn't work until after the user signs in, show the Microsoft account sign-in dialog immediately after the app starts.
- If your app works without needing the user to sign in first, and it needs the user to sign in later to enable specific scenarios or features, provide the following user experience. First, the user must click the Settings charm and then click **Accounts** (or its localized equivalent). After the user does that, display a custom sign-in control—for example, a label with the text "Sign in" (or its localized equivalent). After the user clicks this control, display the sign-in dialog. To support this experience, use the **label** property of the **SettingsCommand** class to display the **Accounts** command (or its localized equivalent). After the user clicks the command, use the **invoked** property to show a **SettingsFlyout** control that contains the custom sign-in control.
- If your app works without needing the user to sign in first, and it provides specific commands that integrate with Microsoft cloud services like Hotmail, SkyDrive, and Messenger, provide the following user experience. First, after the user executes the "save photo" command from the app's client area or its app bar, display the sign-in dialog. After the user is signed in, your code can complete the save task.

Signing users out

In Windows 8, if a user signs in to a device with a local or domain account that is not connected to a Microsoft account, and then signs in to your app with a Microsoft account, your app should provide a custom sign-out control. For example, the control could be a label with the text "Sign out" (or its localized equivalent).

Note If the user signs in to the device with a Microsoft account or a local or domain account that is connected to a Microsoft account, providing a custom sign-out control has no effect.

To determine whether your app should provide a custom sign-out control, your code must provide the following user experience. First, the user must click the Settings charm and then click **Accounts** (or its localized equivalent). After the user clicks the command, and if the user can sign out, display the custom sign-out control. To do this, use the **label** property of the **SettingsCommand** class to display the **Accounts** command (or its localized equivalent). After the user clicks the command, use the **invoked** property to show a **SettingsFlyout** control that contains the custom sign-out control. However, if the user cannot sign out, simply show the current user's name.

Things to avoid

Make sure your app:

- Doesn't display text, controls, or sign-in and sign-out dialogs other than those previously described here. Using the Microsoft account sign-in experience helps reassure your users that your app can't directly access their Microsoft account credentials.
- Doesn't display custom sign-in or sign-out controls anywhere other than the **SettingsFlyout** control or as part of the app's task-command flow.

App resources

Guidelines for creating resources

- Do not put resources, such as UI strings and images, in code. Instead, put them into resource files, such as .resjson or .resw files.
- Use qualifiers to support file and string resources that are tailored for different display scales, UI languages, or high contrast settings.
- Set the default language qualifier project property.
- String resources, even those in the default language, should have a file or folder named with the language tag.
- Add comments to your string resource for the localizer.

Guidelines for referring to resources

- Add unique resource identifiers in the code and markup to refer to resources.
- Refer to images in HTML, code, or manifests without the qualifiers.
- Listen for events that fire when the system changes and it begins to use a different set of qualifiers. Reprocess the document so that the correct resources can be loaded.

Globalization

Whether you plan to localize your UI or not, consider an international market when you select UI terms and images. This planning saves time and cost for the markets you do localize to. Customers who use a version that isn't localized in their native language will understand your UI more easily.

The following table presents the practices recommended for internationalizing your apps.

Practice	Description
Use the correct formats for numbers, dates, times, addresses, and phone numbers.	The formatting used for numbers, dates, times, and other forms of data varies between cultures, regions, languages, and markets. If you're displaying numbers, dates, times, or other data, use globalization APIs to get the format that the user prefers.
Support international paper sizes.	The most common paper sizes differ between countries, so if you include features that depend on paper size, like printing, be sure to support and test common international sizes.
Support international units of measurement and currencies.	Different units and scales are used in different countries, although the most popular are the metric system and the imperial system. If you deal with measurements, like length, temperature, or area, get the correct system measurement by using the **Globalization** namespace. If your app supports displaying currencies, use the correct formatting. You can also get the currency for the user's geographic region by using the **CurrenciesInUse** property.
Display text and fonts correctly.	The ideal font, font size, and direction of text vary between different markets.
Use Unicode for character encoding.	By default, recent versions of Microsoft Visual Studio use Unicode character encoding for all documents. If you're using a different editor, be sure to save source files in the appropriate Unicode character encodings. All Windows Runtime APIs return UTF-16 encoded strings.
Record the language of	When your app asks users for text input, record the language of

input.	input. This ensures that when the input is displayed later it's presented to the user with the appropriate formatting. Use the **CurrentInputMethodLanguage** property to get the current input language.
Don't use language to assume a user's location, and don't use location to assume a user's language.	In Windows, the user's language and location are separate concepts. A user can speak a particular regional variant of a language, like en-gb for English as spoken in Great Britain. But, the user can be located in an entirely different country or region. Consider whether your apps require knowledge about the user's language, like for UI text, or location, like for licensing issues.
Avoid colloquialisms and metaphors.	Language that's specific to a demographic group, such as culture and age, can be hard to understand or translate, because only people in that demographic group use that language. Similarly, metaphors might make sense to one person but mean nothing to someone else. For example, a "bluebird" means something specific to those who are part of skiing culture, but those who aren't part of that culture don't understand the reference. If you plan to localize your app and you use an informal voice or tone, be sure that you adequately explain to localizers the meaning and voice to be translated.
Don't use technical jargon, abbreviations, or acronyms.	Technical language is less likely to be understood by non-technical audiences or people from other cultures or regions, and it's difficult to translate. People don't use these kinds of words in everyday conversations. Technical language often appears in error messages to identify hardware and software issues. At times, this might be necessary, but you should rewrite strings to be non-technical.
Avoid images that might be offensive.	Images that are appropriate in your own culture are sometimes offensive or misinterpreted in other cultures. Avoid use of religious symbols, animals, or color combinations that are associated with national flags or political movements.
Avoid political offense in maps or when referring to regions.	Maps sometimes include controversial regional or national boundaries, and they're a frequent source of political offense. Be careful that any UI used for selecting a nation refers to it as a

	"country/region". Putting a disputed territory in a list labeled "Countries", like in an address form, could get you in trouble.
Don't use string comparison by itself to compare language tags.	BCP-47 language tags are complex. There are a number of issues when comparing language tags, including issues with matching script information, legacy tags, and multiple regional variants. The resource management system in Windows takes care of matching for you. You can specify a set of resources in any languages, and the system chooses the appropriate one for the user and the app.
Don't assume that sorting is always alphabetic.	For languages that don't use Latin script, sorting is based on things like pronunciation, number of pen strokes, and other factors. Even languages that use Latin script don't always use alphabetic sorting. For example, in some cultures, a phone book might not be sorted alphabetically. The system can handle sorting for you, but if you create your own sorting algorithm, be sure to take into account the sorting methods used in your target markets.

Guidelines for localization

The following table presents the practices recommended for adapting your app for a specific language, region, market, or audience.

Practice	Description
Separate resources such as UI strings and images from code.	Design your apps so that resources, like strings and images, are separated from your code. This enables them to be independently maintained, localized, and customized for different scaling factors, accessibility options, and a number of other user and machine contexts. Separate string resources from your app's code to create a single language-independent codebase. Always separate strings from app code and markup. And place them into a resource file, like a ResW or ResJSON file. Use the resource infrastructure in Windows to handle the selection of the most appropriate resources to match the user's

	runtime environment.
Isolate other localizable resource files.	Place other files that require localization, like images that contain text to be translated or that need to be changed due to cultural sensitivity, in folders tagged with language names.
Set your default language, and mark all of your resources, even the ones in your default language.	Always set the default language for your apps appropriately. The default language determines the language that's used when the user doesn't speak any of the supported languages of the app. Mark default language resources, for example en-us/Logo.png, with their language, so the system can tell which language the resource is in and how it's used in particular situations.
Determine the resources of your app that require localization.	What needs to change if your app must be localized for other markets? Text strings require translation into other languages. Images sometimes must be adapted for other cultures. Consider how localization affects other resources that your app uses, like audio or video.
Use resource identifiers in the code and markup to refer to resources.	Instead of having string literals or specific file names for images in your markup, use references to the resources. Be sure to use unique identifiers for each resource.
Enable text size to increase.	Allocate text buffers dynamically, since text size sometimes expands when translated. If you must use static buffers, make them extra-large (perhaps doubling the length of the English string) to accommodate potential expansion when strings are translated. There may also be limited space available for a user interface. To accommodate localized languages, ensure that your string length is approximately 40% longer than what you would need for the English language. For short strings, such as single words, you may need as much as 300% more space. In addition, enabling multiline support and text-wrapping in a control leaves more space to display each string.
Support mirroring.	Text alignment and reading order can be left-to-right, as in English, or right-to-left (RTL), as in Arabic or Hebrew. If you are localizing your product into languages that use a different

	reading order than your own, be sure that the layout of your UI elements supports mirroring. Even items such as back buttons, UI transition effects, and images sometimes need to be mirrored.
Reduce localization costs.	Reduce localization costs by avoiding use of text in images or speech in audio files. If you're localizing to a language with a different reading direction than your own, using symmetrical images and effects make it easier to support mirroring.
Use short strings.	Shorter strings are easier to translate and enable translation recycling. Translation recycling saves money because the same string isn't sent to the localizer twice. Strings longer than 8192 characters are not supported by some localization tools, so keep string length to 4000 or less.
Provide strings that contain an entire sentence.	Provide strings that contain an entire sentence, instead of breaking the sentence into individual words, because the translation of words often depends on their position in a sentence. Also, don't assume that a phrase with multiple parameters is translated with those parameters in the same order for every language.
Comment strings.	Ensure that strings are properly commented, and only the strings that must be translated are provided to localizers. Over-localization is a common source of problems.
Don't re-use strings in different contexts.	Don't re-use translated strings in different contexts, because even simple words like "on" and "off" may be translated differently, depending on the context.