



# Developing an end-to-end Windows Store app using JavaScript: Hilo

Derick Bailey  
Christopher Bennage  
Larry Brader  
David Britch  
Mike Jones  
Poornimma Kaliappan  
Blaine Wastell

December 2012



patterns & practices

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

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

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

© 2012 Microsoft. All rights reserved.

Microsoft, DirectX, Expression Blend, Internet Explorer, Visual Basic, Visual Studio, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

## Table of Contents

|   |    |
|---|----|
| Developing an end-to-end Windows Store app using JavaScript: Hilo.....                                | 7  |
| Download.....   | 7  |
| Applies to .....  | 7  |
| Prerequisites .....   | 7  |
| Table of contents at a glance .....   | 8  |
| Learning resources.....   | 8  |
| Getting started with Hilo (Windows Store apps using JavaScript and HTML) .....                        | 9  |
| Download.....   | 9  |
| Building and running the sample .....   | 9  |
| Designing the UX.....   | 10 |
| Projects and solution folders .....   | 10 |
| Development tools and languages .....   | 11 |
| Async programming patterns and tips in Hilo (Windows Store apps using JavaScript and HTML) .....      | 13 |
| You will learn.....   | 13 |
| A brief introduction to promises.....   | 13 |
| How to use a promise chain.....   | 14 |
| Using the bind function.....  | 15 |
| Grouping a promise .....  | 17 |
| Nesting in a promise chain.....   | 18 |
| Wrapping values in a promise.....   | 19 |
| Handling errors .....   | 20 |
| Using a separated presentation pattern in Hilo (Windows Store apps using JavaScript and HTML) .....   | 22 |
| You will learn.....   | 22 |
| MVP and the supervising controller pattern .....  | 22 |
| Mediator pattern .....  | 24 |
| Separating responsibilities for the AppBar and the page header.....                                   | 27 |
| Separating responsibilities for the ListView.....   | 28 |
| Using the query builder pattern in Hilo (Windows Store apps using JavaScript and HTML).....           | 30 |
| You will learn.....   | 30 |
| The builder pattern .....   | 30 |
| The query object pattern .....  | 31 |
| Code walkthrough.....   | 32 |
| Making the file system objects observable .....   | 39 |
| Working with tiles and the splash screen in Hilo (Windows Store apps using JavaScript and HTML) ..... | 41 |

|  |    |
|--|----|
| You will learn.....  | 41 |
| Why are tiles important? .....   | 41 |
| Choosing a tile strategy.....  | 41 |
| Designing the logo images .....  | 42 |
| Placing the logos on the default tiles.....  | 43 |
| Updating tiles .....   | 43 |
| Adding the splash screen .....   | 50 |
| Creating and navigating between pages in Hilo (Windows Store apps using JavaScript and HTML) ..... | 51 |
| You will learn.....  | 51 |
| Adding New Pages to the Project .....  | 51 |
| Designing pages in Visual Studio and Blend .....   | 52 |
| Navigation model in Hilo.....  | 53 |
| Navigating between pages in the single-page navigation model .....                                 | 54 |
| Creating and loading pages.....  | 55 |
| Loading the hub page.....  | 58 |
| Establishing the Data Binding .....  | 59 |
| Supporting portrait, snap, and fill layouts .....  | 60 |
| Using controls in Hilo (Windows Store apps using JavaScript and HTML).....                         | 62 |
| You will learn.....  | 62 |
| Common controls used in Hilo.....  | 62 |
| ListView .....   | 64 |
| FlipView.....  | 67 |
| AppBar.....  | 69 |
| SemanticZoom .....   | 70 |
| Canvas .....   | 70 |
| Img .....  | 72 |
| Styling controls.....  | 74 |
| Touch and gestures.....  | 74 |
| Testing controls.....  | 74 |
| Working with data sources in Hilo (Windows Store apps using JavaScript and HTML).....              | 75 |
| You will learn.....  | 75 |
| Data binding in the month page .....   | 75 |
| Choosing a data source for a page.....   | 77 |
| Querying the file system .....   | 78 |
| Implementing group support with the in-memory data source.....                                     | 78 |

|   |     |
|---|-----|
| Other considerations: Using a custom data source.....   | 81  |
| Considerations in choosing a custom data source.....  | 81  |
| Implementing a data adapter for a custom data source .....  | 82  |
| Binding the ListView to a custom data source.....   | 85  |
| Using touch in Hilo (Windows Store apps using JavaScript and HTML).....                               | 88  |
| You will learn.....   | 88  |
| Tap for primary action .....  | 89  |
| Slide to pan .....  | 90  |
| Swipe to select, command, and move .....  | 91  |
| Pinch and stretch to zoom .....   | 92  |
| Turn to rotate.....   | 93  |
| Swipe from edge for app commands.....   | 96  |
| Swipe from edge for system commands .....   | 97  |
| Handling suspend, resume, and activation in Hilo (Windows Store apps using JavaScript and HTML) ..... | 99  |
| You will learn.....   | 99  |
| Tips for implementing suspend/resume.....   | 99  |
| Understanding possible execution states .....   | 100 |
| Code walkthrough of suspend .....   | 101 |
| Code walkthrough of resume .....  | 102 |
| Code walkthrough of activation.....   | 102 |
| Other ways to close the app .....   | 105 |
| Improving performance in Hilo (Windows Store apps using JavaScript and HTML) .....                    | 107 |
| You will learn.....   | 107 |
| Performance tips.....   | 107 |
| Limit the start time .....  | 107 |
| Emphasize responsiveness.....   | 108 |
| Use thumbnails for quick rendering .....  | 108 |
| Retrieve thumbnails when accessing items.....   | 108 |
| Release media and stream resources when they're no longer needed .....                                | 109 |
| Optimize ListView performance.....  | 109 |
| Keep DOM interactions to a minimum .....  | 110 |
| Optimize property access .....  | 110 |
| Use independent animations.....   | 110 |
| Manage layout efficiently .....   | 110 |
| Store state efficiently.....  | 111 |

|   |     |
|---|-----|
| Keep your app's memory usage low when it's suspended .....                | 111 |
| Minimize the amount of resources that your app uses.....                  | 111 |
| Understanding performance .....   | 111 |
| Improving performance by using app profiling .....                        | 112 |
| Other performance tools .....   | 113 |
| Testing and deploying Windows Store apps: Hilo (JavaScript and HTML)..... | 114 |
| You will learn.....   | 114 |
| Ways to test your app.....  | 114 |
| Using Mocha to perform unit and integration testing.....                  | 115 |
| Testing asynchronous functionality .....                                  | 117 |
| Testing synchronous functionality .....                                   | 118 |
| Using Visual Studio to test suspending and resuming the app.....          | 119 |
| Using the simulator and remote debugger to test devices .....             | 120 |
| Using pseudo-localized versions for localization testing .....            | 120 |
| Security testing .....  | 120 |
| Using performance testing tools.....                                      | 121 |
| Making your app world ready.....  | 121 |
| Using the Windows App Certification Kit to test your app.....             | 122 |
| Creating a Windows Store certification checklist .....                    | 122 |
| Meet the Hilo team (Windows Store apps using JavaScript and HTML) .....   | 124 |
| Meet the team .....   | 124 |

## Developing an end-to-end Windows Store app using JavaScript: Hilo

The JavaScript version of the Hilo photo sample provides guidance to JavaScript developers who want to create a Windows 8 app using HTML, CSS, JavaScript, the Windows Runtime, and modern development patterns. Hilo comes with source code and documentation.

### Download

Download Hilo sample

After you download the code, see [Getting started with Hilo](#) for instructions.

Here's what you'll learn:

- How to use HTML, CSS, JavaScript, and the Windows Runtime to create a world-ready app for the global market. The Hilo source code includes support for three languages.
- How to implement tiles, pages, controls, touch, navigation, file system queries, suspend/resume.
- How to implement the Model-View-Presenter and query builder patterns.
- How to test your app and tune its performance.

### Note

- If you're just getting started with Windows Store apps, read [Tutorial: Create your first Windows Store app using JavaScript](#) to learn how to create a simple Windows Store app with JavaScript. Then download Hilo to see a complete app that demonstrates recommended implementation patterns.
- To learn about creating Hilo as a Windows Store app using C++ and XAML, see [Developing an end-to-end Windows Store app using C++ and XAML: Hilo](#).

### Applies to

- Windows Runtime for Windows 8
- Windows Library for JavaScript
- JavaScript

### Prerequisites

- Windows 8
- Microsoft Visual Studio 2012
- An interest in JavaScript programming

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

## Table of contents at a glance

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

- [Getting Started with Hilo](#)
- [Async programming patterns and tips](#)
- [Using a separated presentation pattern](#)
- [Using the query builder pattern](#)
- [Working with tiles and the splash screen](#)
- [Creating and navigating between pages](#)
- [Using controls](#)
- [Working with data sources](#)
- [Using touch](#)
- [Handling suspend, resume, and activation](#)
- [Improving performance](#)
- [Testing and deploying Windows Store apps](#)
- [Meet the Hilo team](#)

**Note** This content is available on the web as well. For more info, see [Developing an end-to-end Windows Store app using JavaScript: Hilo \(Windows\)](#).

## Learning resources

If you're new to JavaScript programming for Windows Store apps, read [Roadmap for Windows Store app using JavaScript](#). If you're new to the JavaScript language, see [JavaScript fundamentals](#) before reading this guidance.

The topic [Writing code for Windows Store apps \(JavaScript\)](#) contains important information for web developers learning how to write Windows Store apps.

You might also want to read [Index of UX guidelines for Windows Store apps](#) and [Blend for Visual Studio](#) to learn more about how to implement a great UX. The document [Designing the UX](#) explains how we designed the Hilo UX for both C++ and JavaScript.

## Getting started with Hilo (Windows Store apps using JavaScript and HTML)

Here we explain how to build and run the Hilo sample, how the source code is organized, and what tools and languages it uses.

### Download

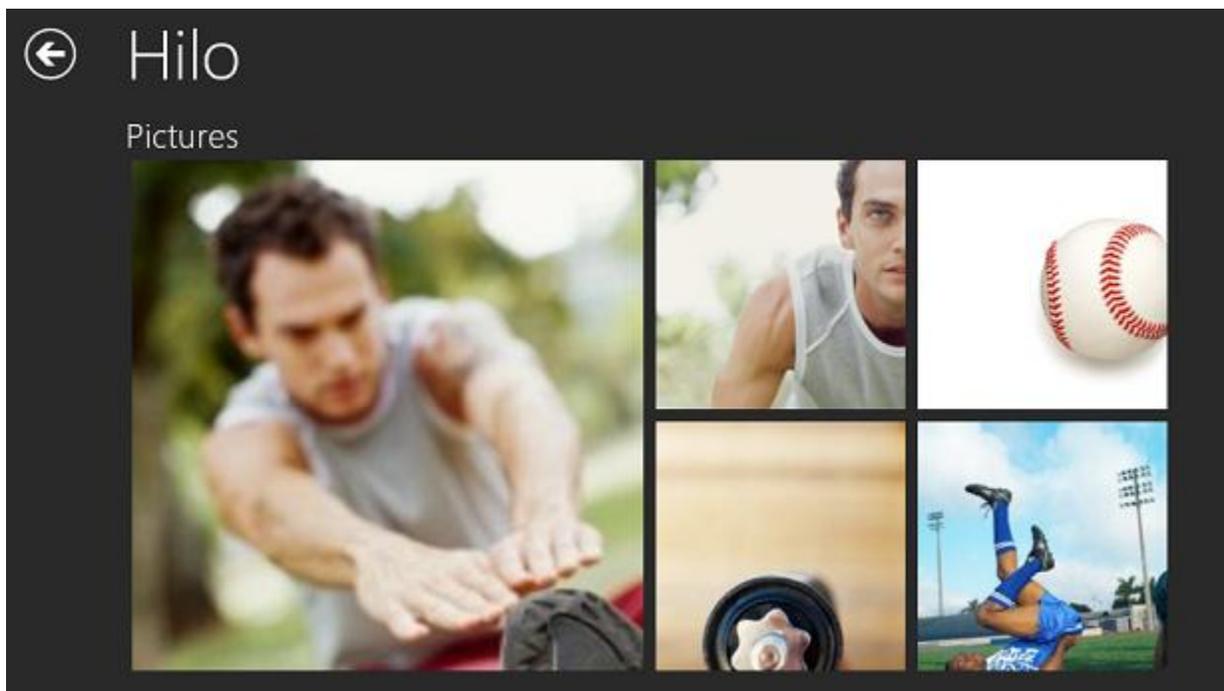
[Download Hilo sample](#)

After you download the code, see the next section, "Building and running the sample," for instructions.

### Building and running the sample

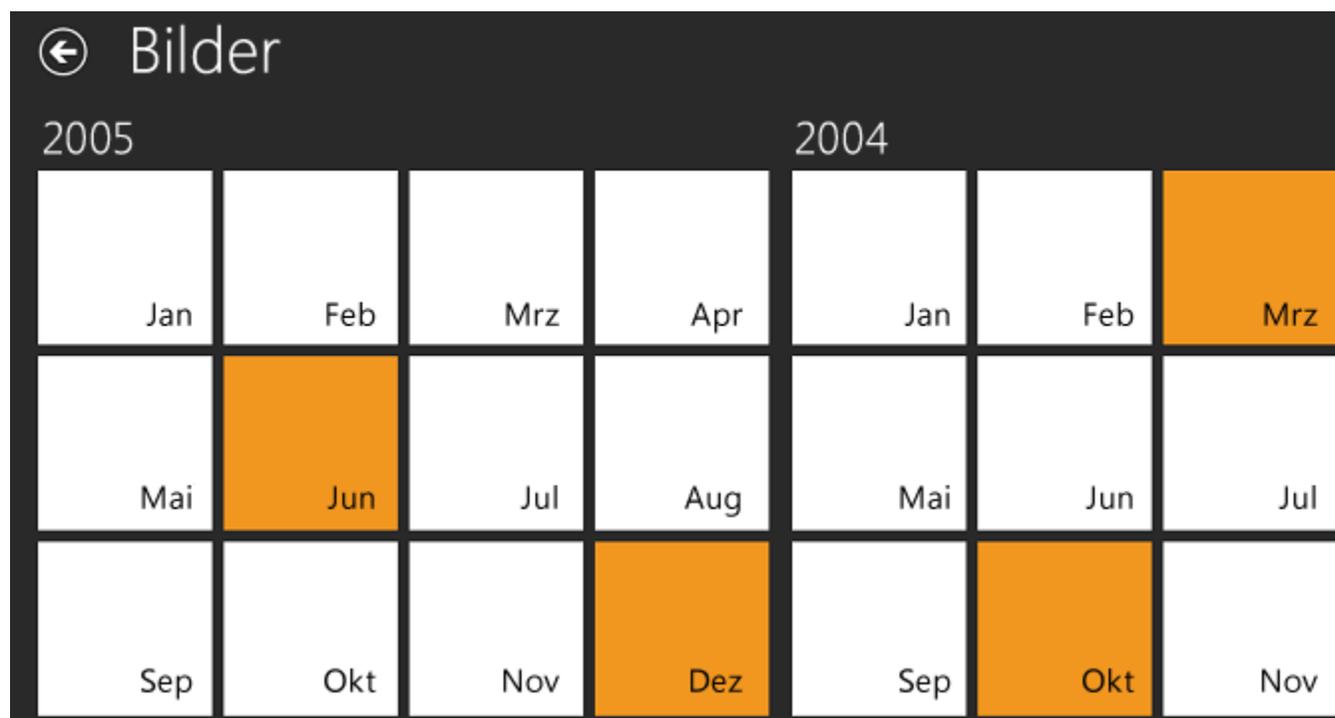
Build the Hilo project as you would build a standard project.

1. On the Microsoft Visual Studio menu bar, choose **Build > Build Solution**. This step compiles the code and also packages it for use as a Windows Store app.
2. After you build the solution, you must deploy it. On the menu bar, choose **Build > Deploy Solution**. Visual Studio also deploys the project when you run the app from the debugger.
3. After you deploy the project, pick the Hilo tile to run the app. Alternatively, from Visual Studio, on the menu bar, choose **Debug > Start Debugging**. Make sure that Hilo is the startup project. When you run the app, the hub page appears.



You can run Hilo in any of the languages that it supports. Set the desired language and calendar in Control Panel. For example, if you set the preferred language to German and the system calendar (date,

time, and number formats) to German before you start Hilo, the app will display German text and use the locale-specific calendar. Here's a screen shot of the year groups view localized for German.



The source code for Hilo includes localization for English (United States), German (Germany), and Japanese (Japan).

## Designing the UX

Hilo C++ and Hilo JavaScript use the same UX design. For more info on the UX for Hilo JavaScript, see the C++ topic [Designing the UX](#). You might want to read [Index of UX guidelines for Windows Store apps](#) before you read the UX document.

For general info on designing Windows Store apps, see [Planning Windows Store apps](#).

## Projects and solution folders

The Hilo Visual Studio solution contains two projects: Hilo and Hilo.Specifications. The version of Hilo that contains the Hilo.Specifications project is available at [patterns & practices: HiloJS: a Windows Store app using JavaScript](#).

The Hilo project uses Visual Studio solution folders to organize the source code files into these categories:

- The **References** folder, created automatically, contains the Windows Library for JavaScript SDK reference.

- The **Hilo** folder contains subfolders that represent a grouping of files by feature, like the Hilo\hub and the Hilo\Tiles folders. Each subfolder contains the HTML, CSS, and JavaScript files that are associated with a specific page or feature area.
- The **images** folder contains the splash screen, the tile, and other images.
- The **strings** folder contains subfolders named after supported locales. Each subfolder contains resource strings for the supported locales (in .resjson files).

The Hilo.Specifications project contains unit tests for Hilo. It shares code with the Hilo project and adds source files that contain unit tests.

You can reuse some of the components in Hilo with any app with little or no modification. We found that organizing files by feature was a helpful pattern. For your own app, you can adapt the organization and ideas that these files provide. When we consider a coding pattern to be especially applicable to other apps, we note that.

## Development tools and languages

Hilo is a Windows Store app that uses JavaScript. This combination isn't the only option, and a Hilo app in C++ is also available for download. You can write Windows Store apps in many ways, using the language of your choice.

With the changes in HTML, CSS, and JavaScript, JavaScript is now a great choice for a high-performing app such as Hilo that requires loading and manipulating lots of images. In Windows Store apps, you benefit automatically from improvements to the Internet Explorer 10 JavaScript engine (which you can read about [here](#)), and can take advantage of HTML5 features like running hardware-accelerated CSS animations. With JavaScript, you can also take advantage of the Internet Explorer F12 debugging tools that are now available in Visual Studio. And, of course, when you develop in HTML, CSS, and JavaScript, you can use the vast coding resources that are available for web-based apps.

When we considered which language to use for Hilo, we also asked these questions:

- **What kind of app did we want to build?** If you're creating a food, banking, or photo app, you might use HTML5/CSS/JavaScript or XAML with C#, C++, or Visual Basic because the Windows Runtime provides enough built-in controls and functionality to create these kinds of apps. But if you're creating a three-dimensional app or game and want to take full advantage of graphics hardware, you might choose C++ and DirectX.
- **What was our current skillset?** If you know web development technologies such as HTML, JavaScript, or CSS, then JavaScript might be a natural choice for your app.
- **What existing code could we use?** If you have existing code, algorithms, or libraries that work with Windows Store apps, you might be able to use that code. For example, if you have existing app logic written using jQuery, you might consider creating your Windows Store app with JavaScript. For info about differences between these technologies and Windows Store apps that you would need to know about, see [HTML, CSS, and JavaScript features and differences](#) and [Windows Store apps using JavaScript versus traditional Web apps](#).

**Tip** Apps written in JavaScript also support reusable Windows Runtime components built using C++, C#, or Visual Basic. For more info, see [Creating Windows Runtime Components](#).

The document [Getting started with Windows Store apps](#) orients you to the language options for creating Windows Store apps.

**Note** Regardless of which language you choose, you'll want to ensure that your app is *fast and fluid* to give your users the best possible experience. A fast and fluid app responds to user actions quickly, and with matching energy. The Windows Runtime enables this speed by providing asynchronous operations. Each programming language provides a way to implement these operations. See [Async programming patterns and tips](#) to learn more about how we used JavaScript to implement a fast and fluid UX.

## Async programming patterns and tips in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Use Windows Library for JavaScript promises for asynchronous operations in your Windows Store app.
- Use a promise chain to construct a series of asynchronous tasks. Avoid the use of nested promises in your promise chains. Include an error handler in the last then or done clause in your promise chain.
- Use **WinJS.Promise.join** to group non-sequential asynchronous tasks.

### Important APIs

- [WinJS.Promise.then](#)
- [WinJS.Promise.join](#)

Hilo uses promises to process the results of asynchronous operations. Promises are the required pattern for asynchronous programming in Windows Store apps using JavaScript. Here are some tips and guidance for using promises, and examples of the various ways you can use promises and construct promise chains in your app.

### You will learn

- How to compose asynchronous code.
- How to use promises.
- How to chain and group promises.
- How to code promise chains to avoid nesting.
- How to wrap non-promise values in a promise.
- How to handle errors in a promise.

### A brief introduction to promises

To support asynchronous programming in JavaScript, Windows Runtime and the Windows Library for JavaScript implement the [Common JS Promises/A](#) specification. A promise is an object that represents a value that will be available later. Windows Runtime and Windows Library for JavaScript wrap most APIs in a promise object to support asynchronous method calls and the asynchronous programming style.

When using a promise, you can call the [then](#) method on the returned promise object to assign the handlers for results or errors. The first parameter passed to **then** specifies the callback function (or completion handler) to run when the promise completes without errors. Here is a simple example in Hilo, which specifies a function to run when an asynchronous call, stored in **queryPromise**, completes. In this example, the result of the asynchronous call gets passed into the completion handler, **\_createViewModels**.

**JavaScript: Hilo\Hilo\imageQueryBuilder.js**

```

if (this.settings.bindable) {
    // Create `Hilo.Picture` objects instead of returning `StorageFile` objects
    queryPromise = queryPromise.then(this._createViewModels);
}

```

You can create a promise without invoking [then](#) immediately. To do this, you can store a reference to the promise and invoke **then** at a later time. For an example of this, see [Grouping a promise](#).

Because the call to [then](#) itself returns a promise, you can use **then** to construct promise chains, and pass along results to each promise in the chain. The return value may or may not be ignored. For more info on promises and the other methods that promises support, see [Asynchronous programming in JavaScript](#).

**How to use a promise chain**

There are several places in Hilo where we constructed a promise chain to support a series of asynchronous tasks. The code example here shows the **TileUpdater.update** method. This code controls the process that creates the thumbnails folder, selects the images, and updates the tile. Some of these tasks require the use of asynchronous Windows Runtime functions (such as [getThumbnailAsync](#)). Other tasks in this promise chain are synchronous, but we chose to represent all the tasks as promises for consistency.

**JavaScript: Hilo\Hilo\Tiles\TileUpdater.js**

```

update: function () {
    // Bind the function to a context, so that `this` will be resolved
    // when it is invoked in the promise.
    var queueTileUpdates = this.queueTileUpdates.bind(this);

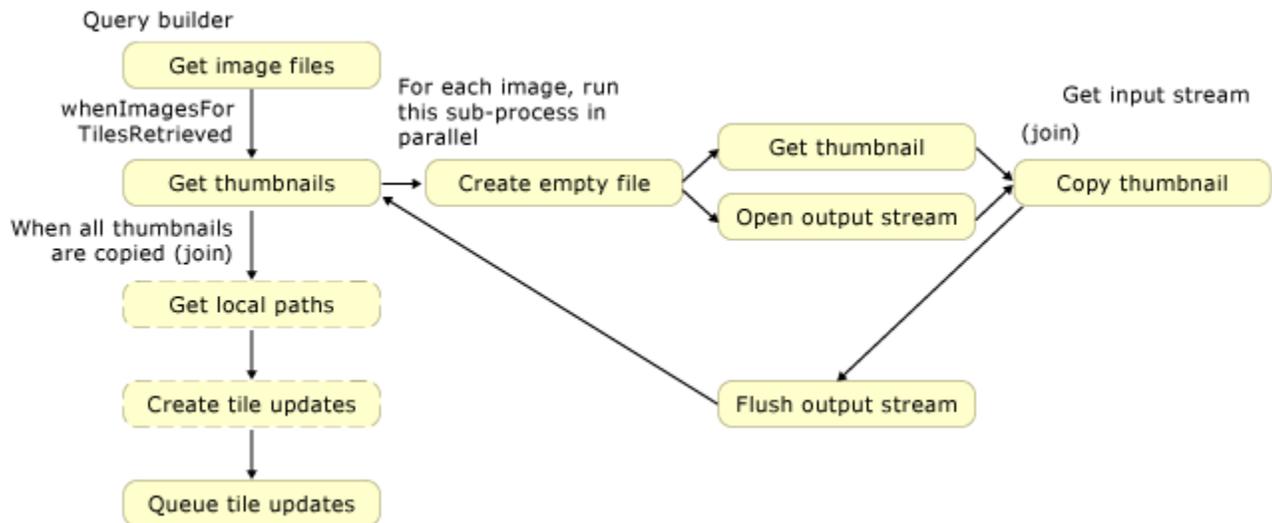
    // Build a query to get the number of images needed for the tiles.
    var queryBuilder = new Hilo.ImageQueryBuilder();
    queryBuilder.count(numberOfImagesToRetrieve);

    // What follows is a chain of promises. These outline a number of
    // asynchronous operations that are executed in order. For more
    // information on how promises work, see the readme.txt in the
    // root of this project.
    var whenImagesForTileRetrieved = queryBuilder.build(picturesLibrary).execute();
    whenImagesForTileRetrieved
        .then(Hilo.Tiles.createTileFriendlyImages)
        .then(this.getLocalImagePaths)
        .then(Hilo.Tiles.createTileUpdates)
        .then(queueTileUpdates);
}

```

Each function in the promise chain passes its result as input to the next function. For example, when **whenImagesForTileRetrieved** completes, it invokes the completion handler, **createTileFriendlyImages**. The calling function automatically passes the completion handler an array of files returned from the asynchronous function call. In this example the asynchronous function is **queryBuilder.build.execute**, which returns a promise.

The next diagram shows the operation flow in the tile updater's promise chain. This flow creates thumbnail images and tile updates. Creating a thumbnail from a picture file requires asynchronous steps that don't follow a straight-line. In the diagram, the solid ovals are asynchronous operations in the Windows Runtime. The dashed ovals are tasks that call synchronous functions. The arrows are inputs and outputs.



For more information about promise chains, see [Chaining promises](#).

## Using the bind function

The JavaScript **bind** function creates a new function with the same body as the original function, in which the **this** keyword resolves to the first parameter passed into **bind**. You can also pass additional parameters to the new function in the call to **bind**. For more info, see the [bind](#) function.

In coding asynchronous operations in Hilo, the [bind](#) function helped us with a couple of scenarios. In the first scenario, we used [bind](#) to preserve the evaluation of **this** in the execution context, and pass along local variables to the closure (a typical use in JavaScript). In the second scenario, we wanted to pass multiple parameters to the completion handler.

The code in `TileUpdater.js` provides an example of the first scenario where [bind](#) was useful. The last completion handler in the tile updater's promise chain, **queueTileUpdates**, gets bound in the tile updater's **update** method (shown previously). Here, we create a bound version of **queueTileUpdates** by using the **this** keyword, at which point **this** contains a reference to the **TileUpdater** object.

**JavaScript: Hilo\Hilo\Tiles\TileUpdater.js**

```
var queueTileUpdates = this.queueTileUpdates.bind(this);
```

By using [bind](#), we preserve the value of **this** for use in the **forEach** loop in **queueTileUpdates**. Without binding the **TileUpdater** object, which we later assign to a local variable (`var self = this`), we would get an exception for an undefined value if we tried to call `this.tileUpdater.update(notification)` in the **forEach** loop.

**JavaScript: Hilo\Hilo\Tiles\TileUpdater.js**

```
queueTileUpdates: function (notifications) {
    var self = this;
    notifications.forEach(function (notification) {
        self.tileUpdater.update(notification);
    });
},
```

The second scenario where [bind](#) is useful, for passing multiple parameters, is shown in the tile updater's promise chain. In the first completion handler in the tile updater's promise chain, **createTileFriendlyImages**, we use [bind](#) to partially apply two functions. For more info on partial function application, see [Partial application](#) and this [post](#). In Hilo, the **copyFilesToFolder** and **returnFileNamesFor** functions are bound to the context (null, in this case) and to the array of files as a parameter. For **copyFilesToFolder**, we wanted to pass in two arguments: the array of files (thumbnail images), and the result from [createFolderAsync](#), which is the target folder for the thumbnails.

**JavaScript: Hilo\Hilo\Tiles\createTileFriendlyImages.js**

```
function createTileFriendlyImages(files) {
    var localFolder = applicationData.current.localFolder;

    // We utilize the concept of [Partial Application][1], specifically
    // using the [`.bind`][2] method available on functions in JavaScript.
    // `bind` allows us to take an existing function and to create a new
    // one with arguments that been already been supplied (or rather
    // "applied") ahead of time.
    //
    // [1]: http://en.wikipedia.org/wiki/Partial_application
    // [2]: http://msdn.microsoft.com/en-us/library/windows/apps/ff841995

    // Partially apply `copyFilesToFolder` to carry the files parameter with it,
    // allowing it to be used as a promise/callback that only needs to have
    // the `targetFolder` parameter supplied.
    var copyThumbnailsToFolder = copyFilesToFolder.bind(null, files);

    // Promise to build the thumbnails and return the list of local file paths.
    var whenFolderCreated = localFolder.createFolderAsync(thumbnailFolderName,
        creationCollisionOption.replaceExisting);
```

```

    return whenFolderCreated
        .then(copyThumbnailsToFolder);
}

```

**Tip** We can bind to null because the bound functions do not use the **this** keyword.

The bound version of **copyFilesToFolder** is assigned to **copyThumbnailsToFolder**. When **copyFilesToFolder** is invoked, the previously-bound array of files is passed in as the first input parameter, **sourceFiles**. The target folder, which was stored in the previous code example as the result from [createFolderAsync](#), is passed into **copyFilesToFolder** automatically as the second parameter, instead of the only parameter.

#### JavaScript: Hilo\Hilo\Tiles\createTileFriendlyImages.js

```

function copyFilesToFolder(sourceFiles, targetFolder) {

    var allFilesCopied = sourceFiles.map(function (fileInfo, index) {
        // Create a new file in the target folder for each
        // file in `sourceFiles`.
        var thumbnailFileName = index + ".jpg";
        var copyThumbnailToFile = writeThumbnailToFile.bind(this, fileInfo);
        var whenFileCreated = targetFolder.createFileAsync(thumbnailFileName,
creationCollisionOption.replaceExisting);

        return whenFileCreated
            .then(copyThumbnailToFile)
            .then(function () { return thumbnailFileName; });
    });

    // We now want to wait until all of the files are finished
    // copying. We can "join" the file copy promises into
    // a single promise that is returned from this method.
    return WinJS.Promise.join(allFilesCopied);
};

```

## Grouping a promise

When you have non-sequential, asynchronous operations that must all complete before you can continue a task, you can use the [WinJS.Promise.join](#) method to group the promises. The result of **Promise.join** is itself a promise. This promise completes successfully when all the joined promises complete successfully. Otherwise, the promise returns in an error state.

The following code copies tile images to a new folder. To do this, we wait until we have opened a new output stream (**whenFileIsOpen**) and we have obtained the input file that contains a tile image (**whenThumbnailsReady**). Then we start the task of actually copying the image. We pass in the two returned promise objects to the [join](#) function.

**JavaScript: Hilo\Hilo\Tiles\createTileFriendlyImages.js**

```
var whenFileIsOpen = targetFile.openAsync(fileAccessMode.readWrite);
var whenThumbnailIsReady = sourceFile.getThumbnailAsync(thumbnailMode.singleItem);
var whenEverythingIsReady = WinJS.Promise.join({ opened: whenFileIsOpen, ready:
whenThumbnailIsReady });
```

When you use [join](#), the return values from the joined promises are passed as input to the completion handler. Continuing with the tiles example, `args.opened`, below, contains the return value from **whenFileIsOpen**, and `args.ready` contains the return value from **whenThumbnailIsReady**.

**JavaScript: Hilo\Hilo\Tiles\createTileFriendlyImages.js**

```
whenEverythingIsReady.then(function (args) {
    // `args` contains the output from both `whenFileIsOpen` and
    `whenThumbnailIsReady`.
    // We can identify them by the order they were in when we joined them.
    outputStream = args.opened;
    var thumbnail = args.ready;
```

## Nesting in a promise chain

In earlier iterations of Hilo, we created promise chains by invoking [then](#) inside the completion handler for the preceding promise. This resulted in deeply nested chains that were difficult to read. Here is an early version of the **writeThumbnailToFile** function.

**JavaScript**

```
function writeThumbnailToFile(fileInfo, thumbnailFile) {
    var whenFileIsOpen = thumbnailFile.openAsync(readWrite);

    return whenFileIsOpen.then(function (outputStream) {

        return fileInfo.getThumbnailAsync(thumbnailMode).then(function (thumbnail)
{
            var inputStream = thumbnail.getInputStreamAt(0);
            return randomAccessStream.copyAsync(inputStream,
outputStream).then(function () {
                return outputStream.flushAsync().then(function () {
                    inputStream.close();
                    outputStream.close();
                    return fileInfo.name;
                });
            });
        });
    });
}
```

We improved code like this by calling [then](#) directly on each returned promise instead in the completion handler for the preceding promise. For a more detailed explanation, see [post](#). The following code is semantically equivalent to the preceding code, apart from the fact that we also used [join](#) to group promises, but we found it easier to read.

#### JavaScript: Hilo\Hilo\Tiles\createTileFriendlyImages.js

```
function writeThumbnailToFile(sourceFile, targetFile) {
    var whenFileIsOpen = targetFile.openAsync(fileAccessMode.readWrite);
    var whenThumbnailIsReady = sourceFile.getThumbnailAsync(thumbnailMode.singleItem);
    var whenEverythingIsReady = WinJS.Promise.join({ opened: whenFileIsOpen, ready:
whenThumbnailIsReady });

    var inputStream,
        outputStream;

    whenEverythingIsReady.then(function (args) {
        // `args` contains the output from both `whenFileIsOpen` and
`whenThumbnailIsReady`.
        // We can identify them by the order they were in when we joined them.
        outputStream = args.opened;
        var thumbnail = args.ready;
        inputStream = thumbnail.getInputStreamAt(0);
        return randomAccessStream.copyAsync(inputStream, outputStream);

    }).then(function () {
        return outputStream.flushAsync();

    }).done(function () {
        inputStream.close();
        outputStream.close();
    });
}
```

In the preceding code, we need to include an error handling function as a best practice to make sure that the input and output streams are closed when the function returns. For more info, see [Handling errors](#).

It is worth noting that in the first implementation, we passed along some values via the closure (e.g., **inputStream** and **outputStream**). In the second, we had to declare them in the outer scope because it was the only common closure.

## Wrapping values in a promise

When you create your own objects that make asynchronous calls, you may need to explicitly wrap non-promise values in a promise using [Promise.as](#). For example, in the query builder, we wrap the return value from **Hilo.Picture.from** in a promise because the from function ends up calling several other asynchronous Windows Runtime methods ([getThumbnailAsync](#) and **retrievePropertiesAsync**).

**JavaScript: Hilo\Hilo\imageQueryBuilder.js**

```
_createViewModels: function (files) {
    return WinJS.Promise.as(files.map(Hilo.Picture.from));
}
```

For more info on **Hilo.Picture** objects, see [Using the query builder pattern](#).

## Handling errors

When there's an error in a completion handler, the [then](#) function returns a promise in an error state. If you don't have an error handler in the promise chain, you may not see the error. To avoid this situation, the best practice is to include an error handler in the last clause of the promise chain. The error handler will pick up any errors that happen along the line.

**Important** The use of [done](#) is recommended at the end of a promise chain instead of [then](#). They are syntactically the same, but with **then** you can choose to continue the chain. **done** does not return another promise, so the chain cannot be continued. Use **done** instead of **then** at the end of a promise chain to make sure that unhandled exceptions are thrown. For more info, see [How to handle errors with promises](#).

The following code example shows the use of [done](#) in a promise chain.

**JavaScript: Hilo\Hilo\month\monthPresenter.js**

```
return this._getMonthFoldersFor(targetFolder)
    .then(this._queryImagesPerMonth)
    .then(this._buildViewModelsForMonths)
    .then(this._createDataSources)
    .then(function (dataSources) {
        self._setupListViews(dataSources.images, dataSources.years);
        self.loadingIndicatorEl.style.display = "none";
        self.selectLayout();
    });
```

In Hilo, we implement an error handling function when attempting to save a cropped image on the crop page. The error handling function must be the second parameter passed to either [then](#) or [done](#). In this code, the error handling function corrects the photo orientation, if an EXIF orientation is returned but not supported.

**JavaScript: Hilo\Hilo\crop\croppedImageWriter.js**

```
var decoderPromise = getOrientation
    .then(function (retrievedProps) {

        // Even though the EXIF properties were returned,
        // they still might not include the `System.Photo.Orientation`.
        // In that case, we will assume that the image is not rotated.
```

```
    exifOrientation = (retrievedProps.size != 0)
        ? retrievedProps["System.Photo.Orientation"]
        : photoOrientation.normal;

}, function (error) {
    // The file format does not support EXIF properties, continue
    // without applying EXIF orientation.
    switch (error.number) {
        case Hilo.ImageWriter.WINCODEC_ERR_UNSUPPORTEDOPERATION:
        case Hilo.ImageWriter.WINCODEC_ERR_PROPERTYNOTSUPPORTED:
            // The image does not support EXIF orientation, so
            // set it to normal. this allows the getRotatedBounds
            // to work properly.
            exifOrientation = photoOrientation.normal;
            break;
        default:
            throw error;
    }
});
```

For info on best practices related to unit testing and error handling for asynchronous calls, see [Testing and deploying the app](#).

## Using a separated presentation pattern in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Use the supervising controller pattern (an MVP pattern) to help isolate view responsibilities from the presenter.
- Use the mediator pattern to help isolate and manage presentation responsibilities.

In Hilo, we used the supervising controller pattern (a Model-View-Presenter, or MVP, pattern) to get HTML templating support. The supervising controller pattern helped to separate the view from the presenter responsibilities. In most Hilo pages, we also used the mediator pattern to separate and coordinate presentation responsibilities. This separation allowed the app to be tested more easily and the code is also easier to understand.

### You will learn

- How Windows Store apps using JavaScript can benefit from MVP.
- Recommended techniques for applying the MVP pattern: the supervising controller and the mediator pattern.
- How to isolate responsibilities of the presenter classes.
- How to share views across pages for common UI elements such as the Back button and title.

### MVP and the supervising controller pattern

The MVP pattern separates business logic, UI, and presentation behavior.

- The model represents the state and operations of business objects that your app manipulates.
- The view (HTML and CSS) defines the structure, layout, and appearance of what the user sees on the screen. The view manages the controls on the page and forwards user events to a presenter class.
- The presenter contains the logic to respond to events, update the model and, in turn, manipulate the state of the view.

As you make the UI for a JavaScript app more declarative in nature, it becomes easier to separate the responsibilities of the view from presenter classes, and the view can interact directly with the data of the application (the model) through data binding. These are the main features of the supervising controller pattern, which is a type of MVP pattern. For more info, see [Supervising controller](#) on Martin Fowler's Website. For more info on MVP, see [Model-View-Presenter pattern](#).

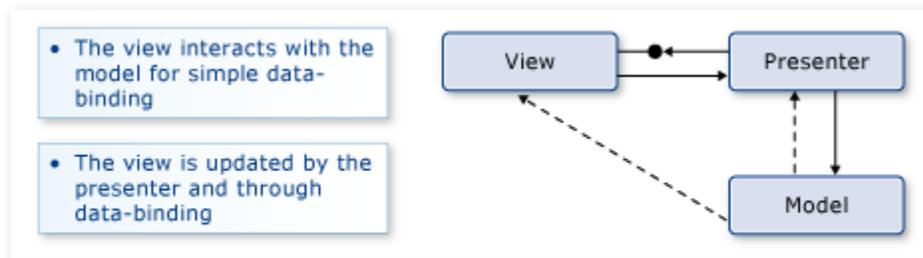
In Hilo, the controller corresponds to the presenter in MVP. To clarify meaning, we will use the term *supervising presenter* when discussing this pattern.

In the supervising presenter pattern, the view directly interacts with the model to perform simple data binding that can be declaratively defined, without presenter intervention. The model has no knowledge

of the view. The presenter updates the model. It manipulates the state of the view only when complex UI logic that cannot be specified declaratively is required.

**Tip** If your app needs to support changes to the model, you may need to implement an observer in your app. Windows Library for JavaScript has the ability to update the view for changes in the model through binding, but not vice versa. You can use the [observableMixin](#) object to implement this.

Here are the relationships between the view, the model, and the presenter in this pattern.



We implemented the supervising presenter pattern in Hilo. We chose this implementation instead of the passive view (another MVP pattern) to get HTML templating support. This made it easier to separate the view and the presenter responsibilities. In this instance, we chose simplicity of code over testability. Nevertheless, testability was an important concern for the project. For example, we also opted to create a presenter for every control on a page. This gave us more testable code, and we also liked the clean separation of concerns that this choice provided. We felt that the assignment of clear, explicit roles made the extra code that was required worthwhile.

**Note** We did not implement the Model-View-ViewModel pattern (MVVM) because two-way data binding is not supported.

The presenter classes in Hilo contain the logic to respond to the events, update the model and, in turn, manipulate the state of the view if needed. In Hilo, presenter classes are specified by using file name conventions. For example, the presenter class for the hub page's [ListView](#) control is implemented in `listViewPresenter.js` and the presenter class for the [FlipView](#) control is implemented in `flipviewPresenter.js`. The presenter class names correspond to the file names.

In Hilo, we used Windows Library for JavaScript templates for declarative binding, such as the template example shown here. This template is attached to a [ListView](#) control (not shown) to display images in the hub page. The `url`, `name`, and `className` properties for each image are declaratively bound to the model. For more info on templates, see [Using Controls](#).

#### HTML: Hilo\Hilo\hub\hub.html

```
<div id="hub-image-template" data-win-control="WinJS.Binding.Template">
  <div data-win-bind="style.backgroundImage: url.imageUrl; alt: name;
className: className" class="thumbnail">
    </div>
  </div>
```

To facilitate testing of the presenter classes, the presenter in the MVP pattern has a reference to the view interface instead of a concrete implementation of the view. In this pattern, you can easily replace the real view with a mock implementation of the view to run tests. This is the approach we take in unit testing Hilo. For more info, see [Testing and deployment](#). In the test code below, we create a **ListViewPresenter** with a reference to mock control (**Specs.WinControlStub**) instead of an actual UI control.

#### JavaScript: Hilo\Hilo.Specifications\specs\hub\ListViewPresenter.spec.js

```
describe("when snapped", function () {

    var el;

    beforeEach(function () {
        var appView = {};
        el = new Specs.WinControlStub();
        el.winControl.addEventListener = function () { };

        var listViewPresenter = new Hilo.Hub.ListViewPresenter(el, appView);

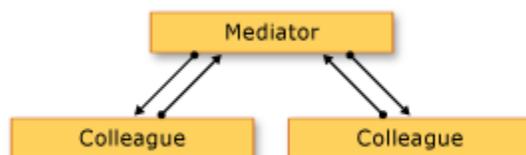
        listViewPresenter.setViewState(Windows.UI.ViewManagement.ApplicationViewState.snapped);
    });

    it("the ListView should use a ListLayout", function () {
        expect(el.winControl.layout instanceof WinJS.UI.ListLayout).equal(true);
    });

});
```

## Mediator pattern

In most Hilo pages, the mediator pattern is used to separate and coordinate presentation concerns. In the mediator pattern, we use a mediator to coordinate behavior of multiple objects (colleagues) that communicate with each other only indirectly, through the mediator object. This enables a loose coupling of the presenters. Each colleague object has a single responsibility.



Each presenter is responsible for a specific part of the page (a control) and the associated behaviors. One page-specific presenter (the mediator) coordinates the other presenters (control-specific) and receives forwarded events. In most of the Hilo pages, we obtain DOM elements in the page's [ready](#) function, by using **document.querySelector**. We then pass the DOM elements to the control-specific presenter. For example, the UI for the detail page contains an [AppBar](#), a filmstrip (based on a [ListView](#)

control), and a [FlipView](#) control. In the **ready** function for detail.js, we obtain the DOM elements for each control, and then pass each element to a new instance of each respective presenter.

#### JavaScript: Hilo\Hilo\detail\detail.js

```
ready: function (element, options) {

    var query = options.query;
    var queryDate = query.settings.monthAndYear;
    var pageTitle = Hilo.dateFormatter.getMonthFrom(queryDate) + " " +
Hilo.dateFormatter.getYearFrom(queryDate);
    this.bindPageTitle(pageTitle);

    var hiloAppBarEl = document.querySelector("#appbar");
    var hiloAppBar = new Hilo.Controls.HiloAppBar.HiloAppBarPresenter(hiloAppBarEl,
WinJS.Navigation, query);

    var filmstripEl = document.querySelector("#filmstrip");
    var flipviewEl = document.querySelector("#flipview");

    var flipviewPresenter = new Hilo.Detail.FlipviewPresenter(flipviewEl);
    var filmstripPresenter = new Hilo.Detail.FilmstripPresenter(filmstripEl);

    var detailPresenter = new Hilo.Detail.DetailPresenter(filmstripPresenter,
flipviewPresenter, hiloAppBar, WinJS.Navigation);
    detailPresenter.addEventListener("pageSelected", function (args) {
        var itemIndex = args.detail.itemIndex;
        options.itemIndex = itemIndex;
    });

    detailPresenter
        .start(options)
        .then(function () {
            WinJS.Application.addEventListener("Hilo:ContentsChanged",
Hilo.navigator.reload);
        });
},
```

Once the control-specific presenters are created, we pass them to a new instance of the page-specific presenter.

**Note** The [ready](#) function is part of the navigation model, and is called automatically when the user navigates to a page control. In Hilo, we use the page controls to obtain DOM elements and instantiate dependencies, but the page controls do not correspond to a specific part of MVP. For more info, see [Creating and navigating between pages](#).

[WinJS.Namespace.define](#) exposes the presenter objects for use in the app. Here is the code that does this for the detail page presenter.

#### JavaScript: Hilo\Hilo\detail\detailPresenter.js

```
WinJS.Namespace.define("Hilo.Detail", {
    DetailPresenter: WinJS.Class.mix(DetailPresenter, WinJS.Utilities.eventMixin)
});
```

When the detail page presenter is created, its constructor function assigns the control-specific presenters to detail page presenter properties. This pattern is typical for the various Hilo presenter classes.

#### JavaScript: Hilo\Hilo\detail\detailPresenter.js

```
function DetailPresenterConstructor(filmstripPresenter, flipviewPresenter,
hiloAppBar, navigation) {
    this.flipview = flipviewPresenter;
    this.filmstrip = filmstripPresenter;
    this.hiloAppBar = hiloAppBar;
    this.navigation = navigation;

    Hilo.bindFunctionsTo(this, [
        "bindImages"
    ]);
},
```

We invoke the presenter's **start** function from [ready](#). The **start** function executes a query to obtain the images needed, and then calls **bindImages**, passing it the query results. For more info on file system queries in Hilo, see [Using the query builder pattern](#).

#### JavaScript: Hilo\Hilo\detail\detailPresenter.js

```
start: function (options) {
    var self = this;
    this.query = options.query;

    return this.query.execute()
        .then(function (images) {

            var result = findImageByIndex(images, options.itemIndex,
options.itemName);
            var storageFile = images[options.itemIndex];
            // If the file retrieved by index does not match the name associated
            // with the query, we assume that it has been deleted (or modified)
            // and we send the user back to the last screen.
            if (isNaN(result.actualIndex)) {
                self.navigation.back();
            }
        });
}
```

```

        } else {
            self.bindImages(images);
            self.gotoImage(result.actualIndex, options.picture);
        }
    });
},

```

The **bindImages** function in `detailPresenter.js` registers event handlers for events such as **imageInvoked**. The handler receives events dispatched from the control-specific presenter classes. In the details page, the behavior is the same whether the user clicks an image in the filmstrip or the [FlipView](#) control, so the event handling is coordinated here in the page's mediator.

#### JavaScript: Hilo\Hilo\detail\detailPresenter.js

```

bindImages: function (images) {

    this.flipview.bindImages(images);
    this.flipview.addEventListener("pageSelected", this.imageClicked.bind(this));

    this.filmstrip.bindImages(images);
    this.filmstrip.addEventListener("imageInvoked", this.imageClicked.bind(this));

    this.hiloAppBar.enableButtons();
},

```

To bind the images to the control, **bindImages** calls the **bindImages** function in the control-specific presenters. The images are bound to the control by using the control's [itemDataSource](#) property.

#### JavaScript: Hilo\Hilo\detail\flipViewPresenter.js

```

bindImages: function (images) {
    this.bindingList = new WinJS.Binding.List(images);
    this.winControl.itemDataSource = this.bindingList.dataSource;
},

```

For more info on binding to data sources, see [Using Controls](#).

## Separating responsibilities for the AppBar and the page header

For common page elements, such as the [AppBar](#) and the page header (an [HTMLControl](#)), we implemented re-usable controls that could be used on multiple pages. We moved the files for these controls out of the page-related subfolders into the `\Hilo\controls` folder. The **AppBar** controls are defined in the `Hilo\controls\HiloAppBar` folder, and the page header control is defined in the `Hilo\controls\Header` folder. The page header includes a Back button control in addition to a page header. Moving the presenter classes for these controls away from page-related code helped us to cleanly separate concerns.

Both the [AppBar](#) and the page header are defined as page controls that support the recommended navigation model for a Windows Store app. To define them as page controls, we specify the use of the [HTMLControl](#) in the HTML code and make a call to [WinJS.UI.Page.define](#) in the associated JavaScript.

In the Hilo implementation, the HTML for a particular page only needs a section reference to the [AppBar](#) or Back button control. For example, here is the HTML used for the **AppBar** in the hub page and details page.

#### HTML: Hilo\Hilo\hub\hub.html

```
<section id="image-nav" data-win-control="WinJS.UI.HtmlControl" data-win-
options="{uri: '/Hilo/controls/HiloAppBar/hiloAppBar.html'}"></section>
```

The main view for the [AppBar](#) is contained in the HTML page referenced here, hiloAppBar.html. The HTML code here specifies two buttons for the **AppBar**, one for the crop command, and one for the rotate command.

#### HTML: Hilo\Hilo\controls\HiloAppBar\hiloAppBar.html

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="{sticky:
false}">
  <button
    data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'rotate', icon:'rotate', section: 'selection',
disabled: true}"
    data-win-bind="{disabled: isCorrupt}"
    data-win-res="{winControl: {label: 'RotateAppBarButton.Name'}}">
  </button>
  <button
    data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'crop', icon:'crop', section: 'selection', disabled:
true}"
    data-win-bind="{disabled: isCorrupt}"
    data-win-res="{winControl: {label: 'CropAppBarButton.Name'}}">
  </button>
</div>
```

The CSS and JavaScript files associated with hiloAppBar.html are contained in the same project location, the Hilo\controls\HiloAppBar folder.

## Separating responsibilities for the ListView

For pages with [ListView](#) controls, like the hub view, we couldn't easily separate the view associated with the **ListView** from the page itself, because the **ListView** is generally closely tied to whole page behavior and UI. Instead, we created a **ListView** presenter for each page to help separate concerns. For example, the hub page has a **ListViewPresenter** class specified in listViewPresenter.js, which is found in the \Hilo\hub folder along with other hub-related files.

The **ListViewPresenter** class handles all state and events associated with the [ListView](#) control for a particular page. For example, we set view state and we set the data source for the **ListView** in the **ListViewPresenter**. Events received by the **ListView** are forwarded to the page's presenter class. In the hub page, we use the **imageNavigated** event handler to handle the **ListView**'s **iteminvoked** event. We then raise a new event to be handled generically in the **HubPresenter**.

#### JavaScript: Hilo\Hilo\hub\listviewpresenter.js

```
imageNavigated: function (args) {
    var self = this;
    args.detail.itemPromise.then(function (item) {
        self.dispatchEvent("itemInvoked", {
            item: item,
            itemIndex: args.detail.itemIndex
        });
    });
},
```

The raised event gets handled by the **itemClicked** function in the **HubPresenter** (not shown).

## Using the query builder pattern in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Use the builder pattern to simplify the task of complex object creation for Windows file system access.
- Use the query object pattern to encapsulate a file system query.
- Test prefetching properties to optimize performance.

### Important APIs

- [Windows.Storage.Search.QueryOptions](#)
- [Windows.Storage.StorageFolder.createFileQueryWithOptions](#)

Hilo implements a combination of the builder pattern and the query object pattern to construct query objects for data access to the Windows file system.

### You will learn

- How to implement the builder pattern for a Windows Store app built using JavaScript.
- How to implement the query object pattern for a Windows Store app built using JavaScript.
- How to prefetch properties to optimize performance.
- How to create and execute a file system query.

### The builder pattern

The builder pattern is an object creation pattern in which you provide input to construct a complex object in multiple steps. Depending on input, the object may vary in its representation. In Hilo, we construct a query builder object whose only function is to build a query object, which encapsulates a query. All of the pages in Hilo need access to the local picture folder to display images, so the query builder settings are particular to a Windows file system query.

#### JavaScript: Hilo\Hilo\month\month.js

```
this.queryBuilder = new Hilo.ImageQueryBuilder();
```

When we create the query builder, default builder options specific to a file system query are set in the constructor. After you create a query builder, you can call its particular members to set options that are specific to the page and query. The following code specifies that the query builder object will create a query object that

- Is bindable to the UI
- Includes a prefetch call on a file property
- Is limited to six items in the query result

**JavaScript: Hilo\Hilo\hub\hubPresenter.js**

```

this.queryBuilder
    .bindable(true)
    .prefetchOptions(["System.ItemDate"])
    .count(maxImageCount);

```

The chained functions in the preceding code demonstrate the fluent coding style, which is supported because the builder pattern used in Hilo implements the fluent interface. The fluent interface enables a coding style that flows easily, partly through the implementation of carefully named methods. In the query builder object, each function in the chain returns itself (the query builder object). This enables the fluent coding style. For information on the fluent interface, see [FluentInterface](#).

Once you have set desired options on the query builder, we use the settings to create a [Windows.Storage.Search.QueryOptions](#) object. We then build the query by passing the query options object as an argument to [createFileQueryWithOptions](#). Finally, to execute the query we call [getFilesAsync](#). We will show an example of building and executing a query in the [Code Walkthrough](#).

For general information on the builder pattern, see [Builder pattern](#).

## The query object pattern

In the query object pattern, you define a query by using a bunch of properties, and then execute the query. You can change options on the query builder and then use the builder to create a new query object. The following code shows the constructor function for the internally-defined query object. The passed-in settings are used to build a file query ([\\_buildQueryOptions](#) and [\\_buildFileQuery](#)).

**JavaScript: Hilo\Hilo\imageQueryBuilder.js**

```

function QueryObjectConstructor(settings) {
    // Duplicate and the settings by copying them
    // from the original, to a new object. This is
    // a shallow copy only.
    //
    // This prevents the original queryBuilder object
    // from modifying the settings that have been
    // sent to this query object.
    var dupSettings = {};
    Object.keys(settings).forEach(function (key) {
        dupSettings[key] = settings[key];
    });

    this.settings = dupSettings;

    // Build the query options.
    var queryOptions = this._buildQueryOptions(this.settings);
    this._queryOptionsString = queryOptions.saveToString();
}

```

```

    if (!this.settings.folder.createFileQueryWithOptions) {
        var folder = supportedFolders[this.settings.folderKey];
        if (!folder) {
            // This is primarily to help any developer who has to extend Hilo.
            // If they add support for a new folder, but forget to register it
            // at the head of this module then this error should help them
            // identify the problem quickly.
            throw new Error("Attempted to deserialize a query for an unknown folder:
" + settings.folderKey);
        }
        this.settings.folder = folder;
    }

    this.fileQuery = this._buildFileQuery(queryOptions);
},

```

The query object pattern also makes it easy to serialize and deserialize the query. You do this by using methods on the query object, typically with corresponding names. For general info on the query object pattern, see [Query Object](#).

In developing Hilo, we changed our initial implementation of the repository pattern to the query object pattern. The main difference between the repository pattern and the query object pattern is that in the repository pattern you create an abstraction of your data source and call methods such as **getImagesByType**. Whereas, with the query object pattern, you set properties on the query builder, and then simply build a query based on the current state of the builder.

We changed to the query object pattern mainly for two reasons. First, we found we were building different query objects with a lot of the same code and default values in different pages. Second, we were adding repository functions that were no longer generic. The following code shows an example of one specialized repository function that we removed from Hilo.

#### JavaScript

```

getQueryForMonthAndYear: function(monthAndYear){
    var options = this.getQueryOptions();
    options.applicationSearchFilter = 'taken: ' + monthAndYear;
    return options.saveToString();
},

```

We felt that adding page-specific functions to the repository made the code too brittle, so we chose instead to streamline the implementation by using a query builder.

## Code walkthrough

The code defines the query builder class in `imageQueryBuilder.js`. The code also exposes the query builder object, **Hilo.ImageQueryBuilder**, for use in the app by using [WinJS.Namespace.define](#).

**JavaScript: Hilo\Hilo\imageQueryBuilder.js**

```
WinJS.Namespace.define("Hilo", {
  ImageQueryBuilder: ImageQueryBuilder
});
```

In Hilo, a particular page can request a query builder by first creating a new **ImageQueryBuilder** object. Typically, we create the query builder in each page's [ready](#) function.

**JavaScript: Hilo\Hilo\month\month.js**

```
this.queryBuilder = new Hilo.ImageQueryBuilder();
```

Once we create the query builder, we pass it to the page's presenter. For more info on the Model-View-Presenter pattern we used to create different views, see [Using a separated presentation pattern](#).

**JavaScript: Hilo\Hilo\month\month.js**

```
this.presenter = new Hilo.month.MonthPresenter(loadingIndicator, this.semanticZoom,
this.zoomInListView, zoomOutListView, hiloAppBar, this.queryBuilder);
```

When you create the query builder object, the constructor for the query builder calls **reset** to set the default values.

**JavaScript: Hilo\Hilo\imageQueryBuilder.js**

```
function ImageQueryBuilderConstructor() {
  this.reset();
},
```

In the **reset** function of the query builder object, we set all the default values, such as the source file folder and the supported files types.

**JavaScript: Hilo\Hilo\imageQueryBuilder.js**

```
reset: function () {
  this._settings = {};
  this._set("fileTypes", [".jpg", ".jpeg", ".tiff", ".png", ".bmp", ".gif"]);
  this._set("prefetchOption",
fileProperties.PropertyPrefetchOptions.imageProperties);

  this._set("thumbnailOptions", fileProperties.ThumbnailOptions.useCurrentScale);
  this._set("thumbnailMode", fileProperties.ThumbnailMode.picturesView);
  this._set("thumbnailSize", 256);

  this._set("sortOrder", commonFileQuery.orderByDate);
  this._set("indexerOption", search.IndexerOption.useIndexerWhenAvailable);
```

```

    this._set("startingIndex", 0);
    this._set("bindable", false);

    return this;
},

```

The values are added to the array in the `_set` method.

#### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```

_set: function (key, value) {
    this._settings[key] = value;
}

```

After you create a query builder, you can set additional options on the builder to match the desired query. For example, in the hub page, we set options in the following code. These options specify the type of builder as bindable, set a prefetch option for the item date ([System.ItemDate](#)), and set the count to six. The count value indicates that the hub page will display only six images total.

#### JavaScript: Hilo\Hilo\hub\hubPresenter.js

```

this.queryBuilder
    .bindable(true)
    .prefetchOptions(["System.ItemDate"])
    .count(maxImageCount);

```

When we call **bindable**, we simply add a new "bindable" setting to the array of the query builder's settings. We will use this property later to create bindable objects that wrap the returned file system objects. For more info, see [Making the file system objects observable](#).

#### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```

bindable: function (bindable) {
    // `!!` is a JavaScript coercion trick to convert any value
    // in to a true boolean value.
    //
    // When checking equality and boolean values, JavaScript
    // coerces `undefined`, `null`, `0`, `""`, and `false` into
    // a boolean value of `false`. All other values are coerced
    // into a boolean value of `true`.
    //
    // The first ! then, negates the coerced value. For example,
    // a value of "" (empty string) will be coerced in to `false`.
    // Therefore `!""` will return `true`.
    //
    // The second ! then inverts the negated value to the
    // correct boolean form, as a true boolean value. For example,

```

```
// `!!""` returns `false` and `!!"something"` returns true.
this._set("bindable", !!bindable);
return this;
},
```

When you perform file operations with the Windows Runtime, it can be helpful to instruct Windows Runtime to optimize the retrieval of specific file properties. You do this by setting prefetch options on a query. Here, we take the passed in [System.ItemDate](#) parameter and set the query builder's **prefetchOption** property to the same value. Initially, we just set the prefetch properties on the query builder, as shown here. We will use this value later when we build the query.

#### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```
prefetchOptions: function (attributeArray) {
    this._set("prefetchOption", fileProperties.PropertyPrefetchOptions.none);
    this._set("prefetchAttributes", attributeArray);
    return this;
},
```

Once the query builder's options are set, we call **build** to build the actual query. We pass in a [Windows.Storage.KnownFolders](#) object to specify the file system folder set.

#### JavaScript: Hilo\Hilo\hub\hubPresenter.js

```
var query = this.queryBuilder.build(this.folder);
```

The **build** method creates a new internally defined query object. In the query object's constructor function, **QueryObjectConstructor**, we attach the query builder's settings.

#### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```
function QueryObjectConstructor(settings) {
    // Duplicate and the settings by copying them
    // from the original, to a new object. This is
    // a shallow copy only.
    //
    // This prevents the original queryBuilder object
    // from modifying the settings that have been
    // sent to this query object.
    var dupSettings = {};
    Object.keys(settings).forEach(function (key) {
        dupSettings[key] = settings[key];
    });

    this.settings = dupSettings;

    // Build the query options.
```

```

var queryOptions = this._buildQueryOptions(this.settings);
this._queryOptionsString = queryOptions.saveToString();

if (!this.settings.folder.createFileQueryWithOptions) {
    var folder = supportedFolders[this.settings.folderKey];
    if (!folder) {
        // This is primarily to help any developer who has to extend Hilo.
        // If they add support for a new folder, but forget to register it
        // at the head of this module then this error should help them
        // identify the problem quickly.
        throw new Error("Attempted to deserialize a query for an unknown folder:
" + settings.folderKey);
    }
    this.settings.folder = folder;
}

this.fileQuery = this._buildFileQuery(queryOptions);
},

```

From the constructor, the query object calls **\_buildQueryOptions**. This function turns the query builder settings into a valid [Windows.Storage.Search.QueryOptions](#) object. It is here that we set prefetch options to optimize performance. For the Hub page, we pass the [ItemDate](#) property as the second parameter in [setPropertyPrefetch](#).

#### JavaScript: Hilo\Hilo\hub\hub.js

```

_buildQueryOptions: function (settings) {
    var queryOptions = new search.QueryOptions(settings.sortOrder,
settings.fileTypes);
    queryOptions.indexerOption = settings.indexerOption;

    queryOptions.setPropertyPrefetch(settings.prefetchOption,
settings.prefetchAttributes);

    if (this.settings.monthAndYear) {
        queryOptions.applicationSearchFilter =
translateToAQSFilter(settings.monthAndYear);
    }

    return queryOptions;
},

```

The prefetched item date is not used directly in the hub view. The hub view doesn't care about dates, and just displays the first six images. But in hub.js we will use the item date to set the correct index value on an image before passing to other pages, such as the details page. Windows Runtime handles retrieval of the item date on demand in a separate asynchronous operation. This is a relatively slow operation that is performed one file at a time. By prefetching the item dates before we get the actual images, we could improve performance.

For the month page, we also set a prefetch option on the thumbnails ([setThumbnailPrefetch](#)). For more info on prefetching thumbnails, see [Improving performance](#).

When creating your own apps, you will want to test prefetch options when you interact with the file system, to see whether you can improve performance.

After calling `_buildQueryOptions`, the query object constructor then calls `_buildFileQuery`. This function converts the [QueryOptions](#) object to a Windows Runtime file query by using [Windows.Storage.StorageFolder.createFileQueryWithOptions](#)

#### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```
_buildFileQuery: function (queryOptions) {
    return this.settings.folder.createFileQueryWithOptions(queryOptions);
},
```

We then execute the query. In the Hub page, the code to execute the query looks like this.

#### JavaScript: Hilo\Hilo\hub\hubPresenter.js

```
return query.execute()
    .then(function (items) {
        if (items.length === 0) {
            self.displayLibraryEmpty();
        } else {
            self.bindImages(items);
            self.animateEnterPage();
        }
    });
```

In the execute call, we use [getFilesAsync](#) to actually retrieve the images. We then check whether the `bindable` property was set in the query. If `bindable` is set, this means we need to wrap returned [StorageFile](#) objects to make them usable for binding to the UI. For info on wrapping the file system object for binding to the UI, see [Making the file system objects observable](#).

#### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```
execute: function () {
    var start, count;
    var queryPromise;

    switch (arguments.length) {
        case (0):
            start = this.settings.startingIndex;
            count = this.settings.count;
            break;
        case (1):
            start = arguments[0];
```

```

        count = 1;
        break;
    case (2):
        start = arguments[0];
        count = arguments[1];
        break;
    default:
        throw new Error("Unsupported number of arguments passed to
`query.execute`.");
    }

    if (count) {
        // Limit the query to a set number of files to be returned, which accounts
        // for both the `count(n)` and `imageAt(n)` settings from the query builder.
        queryPromise = this.fileQuery.GetFilesAsync(start, count);
    } else {
        queryPromise = this.fileQuery.GetFilesAsync();
    }

    if (this.settings.bindable) {
        // Create `Hilo.Picture` objects instead of returning `StorageFile` objects
        queryPromise = queryPromise.then(this._createViewModels);
    }

    return queryPromise;
},

```

When the query result is returned in the completed promise, we call **bindImages**. The main job of this function is to associate the ungrouped index value for each returned image to a group index value that is month-based. It then requests binding to the UI by using **setDataSource**.

#### JavaScript: Hilo\Hilo\hub\hubPresenter.js

```

bindImages: function (items) {
    this.dataSource = items;

    if (items.length > 0) {
        items[0].className = items[0].className + " first";
    }

    // We need to know the index of the image with respect to
    // to the group (month/year) so that we can select it
    // when we navigate to the detail page.
    var lastGroup = "";
    var indexInGroup = 0;
    items.forEach(function (item) {
        var group = item.itemDate.getMonth() + " " + item.itemDate.getFullYear();
        if (group !== lastGroup) {

```

```

        lastGroup = group;
        indexInGroup = 0;
    }

    item.groupIndex = indexInGroup;
    indexInGroup++;
});

this.listView.setDataSource(items);
},

```

The Hub page uses a Windows Library for JavaScript [Binding.List](#) for its data source. For info on data sources used in Hilo, see [Working with data sources](#).

## Making the file system objects observable

Objects returned from the file system, aren't in a format that's bindable (observable) to controls such as [ListView](#) and [FlipView](#). This is because Windows Runtime is not mutable. There is a Windows Library for JavaScript utility for making an object observable, but this utility tries to change the object by wrapping it in a proxy. When returned [StorageFile](#) objects need to be bound to the UI, they are set as "bindable" in the query builder. For bindable objects, the **execute** function in the query object calls **\_createViewModels**. This maps the file system objects to **Hilo.Picture** objects.

### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```

_createViewModels: function (files) {
    return WinJS.Promise.as(files.map(Hilo.Picture.from));
}

```

The code here shows the constructor function for a **Picture** object. In this function, string values are added to the object as properties. After the properties are set, they can be used for binding to the UI. For example, the **url** property is set by calling the app's URL cache (not shown). The cache calls [URL.createObjectURL](#). **createObjectURL** takes as input a blob of binary data (a passed-in thumbnail) and returns a string URL.

### JavaScript: Hilo\Hilo\Picture.js

```

function PictureConstructor(file) {
    var self = this;

    this.addUrl = this.addUrl.bind(this);

    this.storageFile = file;
    this.urlList = {};
    this.isDisposed = false;

    this._initObservable();
}

```

```
this.addProperty("name", file.name);
this.addProperty("isCorrupt", false);
this.addProperty("url", "");
this.addProperty("src", "");
this.addProperty("itemDate", "");
this.addProperty("className", "thumbnail");

this.loadFileProperties();
this.loadUrls();
},
```

**Tip** Before navigating to a new page, you need to release objects that you created by using [URL.createObjectURL](#). This avoids a potential memory leak. You can do this by using [URL.revokeObjectURL](#), which in Hilo is called in `urlCache.js` (not shown). For info on finding memory leaks and other performance tips, see [Improving performance](#) and [Analyzing memory usage data](#).

## Working with tiles and the splash screen in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Create tiles for your app early in development.
- Every app must have a square tile. Consider when to also enable the wide tile.
- Use a tile template to update the contents of your app's tile.

### Important APIs

- [TileUpdateManager](#)
- [TileUpdater](#)

When you use tiles and the splash screen effectively, you can give your users a great first-impression of your Windows Store app.

### You will learn

- How we incorporated an app tile that displays the user's photos.
- How we added the splash screen.
- What considerations to make in your own app.

### Why are tiles important?

Traditional Windows desktop apps use icons. Icons help to visually connect an app or file type with its brand or use. Because an icon is a static resource, you can often wait until the end of the development cycle to incorporate it. However, tiles are different from icons. Tiles add life and personality and can create a personal connection between the app and the user. The goal of tiles is to keep your users coming back by offering a personal connection.

**Tip** For Hilo, we knew early that we wanted to display the user's photos on the tile. But for other apps, it might not be apparent until later what content will keep your users coming back. We recommend that you add support for tile updates when you first create your project, even if you're not yet sure what the tile will show. When you decide later what to show on the tile, the infrastructure will already be in place. This way, you won't need to retrofit your code to support tile updates later.

### Choosing a tile strategy

You have a few options when you choose a tile strategy. You must provide a square tile and optionally you can provide a wide tile. If you provide a wide tile, the user can decide to display the square tile instead of the wide tile. You can also display [badges](#) and [notifications](#) on your tile.

**Note** Because Hilo is a photo app, we wanted to show the user's photos on the tile. You can show images on both the square and wide tile. The wide tile enables us to show multiple images, so we decided to support both. See [Choosing between a square and wide tile size](#) for info on how to choose the right tiles for your app.

We settled on the following rules for the tile behavior:

- Display a wide default tile that shows the Hilo logo before the app is ever launched.
- Each time the app is launched or resumed, update the square and wide tiles according to these rules:
  - If the user has fewer than five pictures in the Pictures folder, display the default tile that shows the Hilo logo.
  - Otherwise, randomly choose 15 pictures from the most recent 30 and set up the notification queue to cycle among three batches of five pictures. If the user chooses the square tile, it will show only the first picture from each batch.

Read [Guidelines and checklist for tiles](#) to learn how tile features relate to the different tile styles. Although you can provide notifications to the square tile, we wanted to also enable the wide tile so that we could display multiple images.

### Tip

You can also enable secondary tiles for your app. A secondary tile enables your users to pin specific content or experiences from an app to the Start screen to provide direct access to that content or experience. For more info about secondary tiles, read [Pinning secondary tiles](#).

## Designing the logo images

Our UX designer created the small, square, and wide logos according to the pixel size requirements for each. The designer suggested a theme that fitted the Hilo brand. Choosing a small logo that represents your app is important so that users can identify your app when the tile displays custom content. This is especially important when the contents of your tile change frequently—you want your users to be able to easily find and identify your app. The small Hilo logo has a transparent background, so that it looks good when it appears on top of a tile notification or other background.

The **images** folder contains the small, square, and wide logo images. For more info about working with image resources, see [Quickstart: Using file or image resources](#) and [How to name resources using qualifiers](#).



30 x 30 pixels



150 x 150 pixels



310 x 150 pixels

**Important** Because the small logo appears on top of a tile notification, consider the color scheme that you use for the foreground color versus the background color. For Hilo, this decision was challenging because we display users' photos and we cannot know what colors will appear. We experimented with several foreground colors and chose white because we felt it looked good when displayed over most pictures. We also considered the fact that most photos do not have white as the dominant color.

## Placing the logos on the default tiles

The Visual Studio manifest editor makes the process of adding the default tiles relatively easy. To learn how, read [Quickstart: Creating a default tile using the Visual Studio manifest editor](#).

## Updating tiles

You use tile templates to update the tiles. Tile templates are an XML-based approach to specify the images and text that customize the tile. The [Windows.UI.Notifications](#) namespace provides classes to update Start screen tiles. For Hilo, we used the [TileUpdateManager](#) and [TileUpdater](#) classes to get the tile template and queue the notifications. Hilo defines the **Hilo.Tiles** namespace and a **TileUpdater** object to choose the images to show on the tile, and form the XML that is provided to Windows Runtime.



**Note** Each tile template enables you to display images, text, or both. We chose **TileTemplateType.TileWideImageCollection** for the wide tile because it shows the greatest number of images. We also chose it because we did not need to display additional text on the tile. We also use **TileSquareImage** to display the first image in the user's collection when they choose to show the square tile. For the complete list of options, see [TileTemplateType](#).

Tile updates occur in the app's **TileUpdater.update** method, which is called during app initialization in `default.js`.

#### JavaScript: Hilo\default.js

```
if (currentState.kind === activation.ActivationKind.launch) {
    if (currentState.previousExecutionState !==
activation.ApplicationExecutionState.terminated) {
        // When the app is started, we want to update its tile
        // on the start screen. Since this API is not accessible
        // inside of Blend, we only invoke it when we are not in
        // design mode.
        if (!Windows.ApplicationModel.DesignMode.designModeEnabled) {
            var tileUpdater = new Hilo.Tiles.TileUpdater();
            tileUpdater.update();
        }
    }
}
```

**Note** We considered updating the tiles when the app suspends, instead of when the app initializes, to make the updates more frequent. However, every Windows Store app should suspend as quickly as possible. Therefore, we did not want to introduce additional overhead when the app suspends. For more info, read [Optimizing your app's lifecycle](#).

The **TileUpdater.update** method performs the following steps to update the tile in the background:

- Create a local folder to store a copy of the thumbnails displayed on the tile (implemented in the **createTileFriendlyImages** function).
- If there are at least 30 photos in the user's collection:
  - Generate thumbnails for the 30 photos in the local app folder. (This is implemented in the **createTileFriendlyImages** function.)
  - Randomly select 15 of the user's 30 most recent photos. (This is implemented in the **createTileUpdates** function).
  - Create 3 batches of 5 photos. (This is implemented in the **createTileUpdates** function).

- Create the notification and update the tile. (This is implemented in the **createTileUpdates** and **queueTileUpdates** functions).

[Verify your URLs](#) describes the ways you can reference images that appear in your tile notification. We use `ms-appdata:///local/` for Hilo because we copy thumbnails to a local app folder.

**Note** The number of bytes consumed by the thumbnails is small, so even if very large pictures are chosen for the tile, copying the thumbnails doesn't take much time or disk space.

We copy photos to a local app folder for two reasons. First, we did not want to store thumbnail versions of the images in the user's personal folder. Second, we did not want to reference the user's personal folder directly, in case the user wants to delete a picture that appears on the tile.

The following example shows the **TileUpdater.update** method. This code controls the process that creates the thumbnails folder, selects the images, and updates the tile. We create a chain of tasks that perform the update operations. Some of the tasks here operate in a synchronous fashion, but we use the asynchronous programming model, which is a promise chain. We do this because we also have embedded calls to Windows Runtime asynchronous functions, such as [getThumbnailAsync](#). Windows Runtime wraps all APIs in a Promise object to support asynchronous method calls, so we use the asynchronous model. We felt this made the code more consistent and easier to read.

#### JavaScript: Hilo\Hilo\Tiles\TileUpdater.js

```
update: function () {
    // Bind the function to a context, so that `this` will be resolved
    // when it is invoked in the promise.
    var queueTileUpdates = this.queueTileUpdates.bind(this);

    // Build a query to get the number of images needed for the tiles.
    var queryBuilder = new Hilo.ImageQueryBuilder();
    queryBuilder.count(numberOfImagesToRetrieve);

    // What follows is a chain of promises. These outline a number of
    // asynchronous operations that are executed in order. For more
    // information on how promises work, see the readme.txt in the
    // root of this project.
    var whenImagesForTileRetrieved = queryBuilder.build(picturesLibrary).execute();
    whenImagesForTileRetrieved
        .then(Hilo.Tiles.createTileFriendlyImages)
        .then(this.getLocalImagePaths)
        .then(Hilo.Tiles.createTileUpdates)
        .then(queueTileUpdates);
}
```

For more info on the promise chain that we use here, see [Async programming patterns and tips](#).

**Note** We considered multiple options for how to choose pictures. Two alternatives we considered were to choose the most recent pictures, or enable the user to select them. We went with randomly choosing

15 of the most recent 30 to get both variety and recent pictures. We felt that having users choose the pictures would be inconvenient for them, and might not entice them to come back to the app later.

The first function that is called in the tile updater's promise chain, **createTileFriendlyImages** is exported by `createTileFriendlyImages.js`. With that function we set the local folder where we will copy the images. We use [createFolderAsync](#) to create a subfolder in the local folder to store the tile images, and then call the **copyFilesToFolder** function. In this code, we use JavaScript [bind](#) function to create partially applied functions. For more information on the use of [bind](#) and partially applied functions in Hilo, see [Async programming patterns and tips](#).

#### JavaScript: Hilo\Hilo\Tiles\createTileFriendlyImages.js

```
function createTileFriendlyImages(files) {
    var localFolder = applicationData.current.localFolder;

    // We utilize the concept of [Partial Application][1], specifically
    // using the [`.bind`][2] method available on functions in JavaScript.
    // `bind` allows us to take an existing function and to create a new
    // one with arguments that been already been supplied (or rather
    // "applied") ahead of time.
    //
    // [1]: http://en.wikipedia.org/wiki/Partial_application
    // [2]: http://msdn.microsoft.com/en-us/library/windows/apps/ff841995

    // Partially apply `copyFilesToFolder` to carry the files parameter with it,
    // allowing it to be used as a promise/callback that only needs to have
    // the `targetFolder` parameter supplied.
    var copyThumbnailsToFolder = copyFilesToFolder.bind(null, files);

    // Promise to build the thumbnails and return the list of local file paths.
    var whenFolderCreated = localFolder.createFolderAsync(thumbnailFolderName,
creationCollisionOption.replaceExisting);

    return whenFolderCreated
        .then(copyThumbnailsToFolder);
}
```

We let the Windows Runtime handle the task of generating the actual thumbnail images. The runtime scales the thumbnail as needed to fit on the tile.

To get the thumbnail from Windows Runtime, we need to set [ThumbnailMode](#) on the **FileProperties** object, and then call [getThumbnailAsync](#) to retrieve the actual thumbnail images.

#### JavaScript: createTileFriendlyImages.js

```
thumbnailMode = Windows.Storage.FileProperties.ThumbnailMode
// . . .
var whenThumbnailIsReady = fileInfo.getThumbnailAsync(thumbnailMode.singleItem);
```

[getThumbnailAsync](#) is called in the `writeThumbnailToFile` function. In `writeThumbnailToFile`, we use [WinJS.Promise.join](#) to make sure that target files are open and thumbnails are available. When the joined promise has completed, we construct a promise chain that calls `getInputStreamAt` to open a file stream, copy the files to a `RandomAccessStream` file stream using [copyAsync](#), flush the output stream, and then shut everything down.

#### JavaScript: Hilo\Hilo\Tiles\createTileFriendlyImages.js

```
function writeThumbnailToFile(sourceFile, targetFile) {
    var whenFileIsOpen = targetFile.openAsync(fileAccessMode.readWrite);
    var whenThumbnailIsReady = sourceFile.getThumbnailAsync(thumbnailMode.singleItem);
    var whenEverythingIsReady = WinJS.Promise.join({ opened: whenFileIsOpen, ready:
whenThumbnailIsReady });

    var inputStream,
        outputStream;

    whenEverythingIsReady.then(function (args) {
        // `args` contains the output from both `whenFileIsOpen` and
`whenThumbnailIsReady`.
        // We can identify them by the order they were in when we joined them.
        outputStream = args.opened;
        var thumbnail = args.ready;
        inputStream = thumbnail.getInputStreamAt(0);
        return randomAccessStream.copyAsync(inputStream, outputStream);

    }).then(function () {
        return outputStream.flushAsync();

    }).done(function () {
        inputStream.close();
        outputStream.close();
    });
}
```

When the `TileUpdater` object is created, its constructor function uses the [Windows.UI.Notification.TileUpdateManager](#) class to create a `TileUpdater` object. The job of the `TileUpdater` class is to update the content of the app's tile. Use the [TileUpdater.enableNotificationQueue](#) method to queue notifications if more than one image is available.

#### JavaScript: Hilo\Hilo\Tiles\TileUpdater.js

```
function TileUpdaterConstructor() {
    this.tileUpdater = tileUpdateManager.createTileUpdaterForApplication();
    this.tileUpdater.clear();
    this.tileUpdater.enableNotificationQueue(true);
},
```

We added the call to [TileUpdater.clear](#) to support the case where the user deleted everything in the library. This method deletes the tile images.

After the app creates the thumbnails and copies them to the right place, **createTileFriendlyImages**, then the **update** method in `TileUpdater.js` calls **createTileUpdates** to build the live tile notifications. This task includes formatting the XML for the tile templates by selecting a specific XML template for both square and wide tiles. In the running app, wide tiles are used by default because the Hilo package manifest includes an image for a wide tile. The **buildTileNotification** function passes the paths for the thumbnail images to **populateTemplateFor.wideTile**.

#### JavaScript: Hilo\Hilo\Tiles\createTileUpdates.js

```
function buildTileNotification(thumbnailPaths) {
    // The square tile will just display the first image used for wide tile.
    var squareTileFile = thumbnailPaths[0];

    var squareTile = populateTemplateFor.squareTile(squareTileFile);
    var wideTile = populateTemplateFor.wideTile(thumbnailPaths);

    var compositeTile = buildCompositeTile(wideTile, squareTile);
    var notification = new Windows.UI.Notifications.TileNotification(compositeTile);

    return notification;
}
```

The **buildWideTile** function formats the XML for the tile update, using **tileWideImageCollection** as the tile template type. This template specifies XML for five images on the tile. The template builds upon the [TileWideImageCollection](#) tile template by inserting the provided list of thumbnails into the XML content template, to produce a **TileNotification** object.

#### JavaScript: Hilo\Hilo\Tiles\populateTemplate.js

```
function buildWideTile(thumbnailFilePaths) {

    // For more information about the `TileWideImageCollection` template, see:
    // http://msdn.microsoft.com/en-us/library/windows/apps/hh761491.aspx#TileWideImageCollection
    var template = templates.tileWideImageCollection;

    var xml = tileUpdateManager.getTemplateContent(template);
    var images = xml.getElementsByTagName("image");

    thumbnailFilePaths.forEach(function (thumbnailFilePath, index) {
        images[index].setAttribute("src", thumbnailFilePath);
    });

    return xml;
}
```

When the notifications are built, then the **update** method in `TileUpdater.js` calls **queueTileUpdates** to update the tile. For each notification, **queueTileUpdates** calls **TileUpdater.update** to actually update the tiles in the UI.

#### JavaScript: Hilo\Hilo\Tiles\TileUpdater.js

```
queueTileUpdates: function (notifications) {
    var self = this;
    notifications.forEach(function (notification) {
        self.tileUpdater.update(notification);
    });
},
```

The XML for a typical tile notification, which you don't normally see when developing apps, looks like this:

#### XML

```
<tile>
  <visual>
    <binding template="TileWideImageCollection">
      <image id="1" src="ms-appdata:///local/thumbnails/thumbImage_0.jpg"/>
      <image id="2" src="ms-appdata:///local/thumbnails/thumbImage_1.jpg"/>
      <image id="3" src="ms-appdata:///local/thumbnails/thumbImage_2.jpg"/>
      <image id="4" src="ms-appdata:///local/thumbnails/thumbImage_3.jpg"/>
      <image id="5" src="ms-appdata:///local/thumbnails/thumbImage_4.jpg"/>
    </binding>
    <binding template="TileSquareImage">
      <image id="1" src="ms-appdata:///local/thumbnails/thumbImage_0.jpg"/>
    </binding>
  </visual>
</tile>
```

**Note** The wide tile template also contains a template for the square tile. This way, both the square and wide tiles are covered by a single template.

If you use text with your tile, or your images are sensitive to different languages and cultures, read [Globalizing tile and toast notifications](#) to learn how to globalize your tile notifications.

Use these resources to learn more about tile templates:

- [The tile template catalog](#)
- [Tile schema](#)

## Adding the splash screen

All Windows Store apps must have a splash screen, which is a composite of a splash screen image and a background color, both of which you can customize. The splash screen is shown as your app loads. As with tiles, our designer created the splash screen image. We chose an image that resembled the default tile logos and fits the Hilo brand. It was straightforward to add the splash screen to the app. Read [Quickstart: Adding a splash screen](#) to learn how.



If your app needs more time to load, you might need to display the splash screen for a longer duration, to show real-time loading information to your users. To do this, create a page that mimics the splash screen by showing the splash screen image and any additional info. For Hilo, we considered using an extended splash screen, but because the app loads the hub page very quickly, we didn't have to add it. For more info about extending the duration of the splash screen, see [How to extend the splash screen](#).

Use these resources to learn more about splash screens:

- [Adding a splash screen](#)
- [Guidelines and checklist for splash screens](#)

## Creating and navigating between pages in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Use [Blend for Visual Studio 2012](#) to edit and debug CSS. Use Microsoft Visual Studio to edit and debug code.
- Use and modify `navigator.js` to provide navigation between pages in your app. Consider creating a base object for your page controls to centralize common tasks, such as processing localized string resources and handling deserialization.
- Use the **ApplicationView** class for view management. Design your views to support **FullScreenLandscape**, **Filled**, **FullScreenPortrait**, and **Snapped** layouts.

In Windows Store apps such as Hilo, there is one page for each screen of the application that a user can navigate to. The app loads a home page (the main hub page) on startup and loads additional pages in response to navigation requests. We created pages by using page controls, which provide built-in support for navigation, and we modified a project template file, `navigator.js`, to implement the navigation model.

### You will learn

- How pages were designed in the Hilo app.
- How the Hilo app loads pages and their data sources at run time.
- How the Hilo app uses a customized version of the template navigation control.

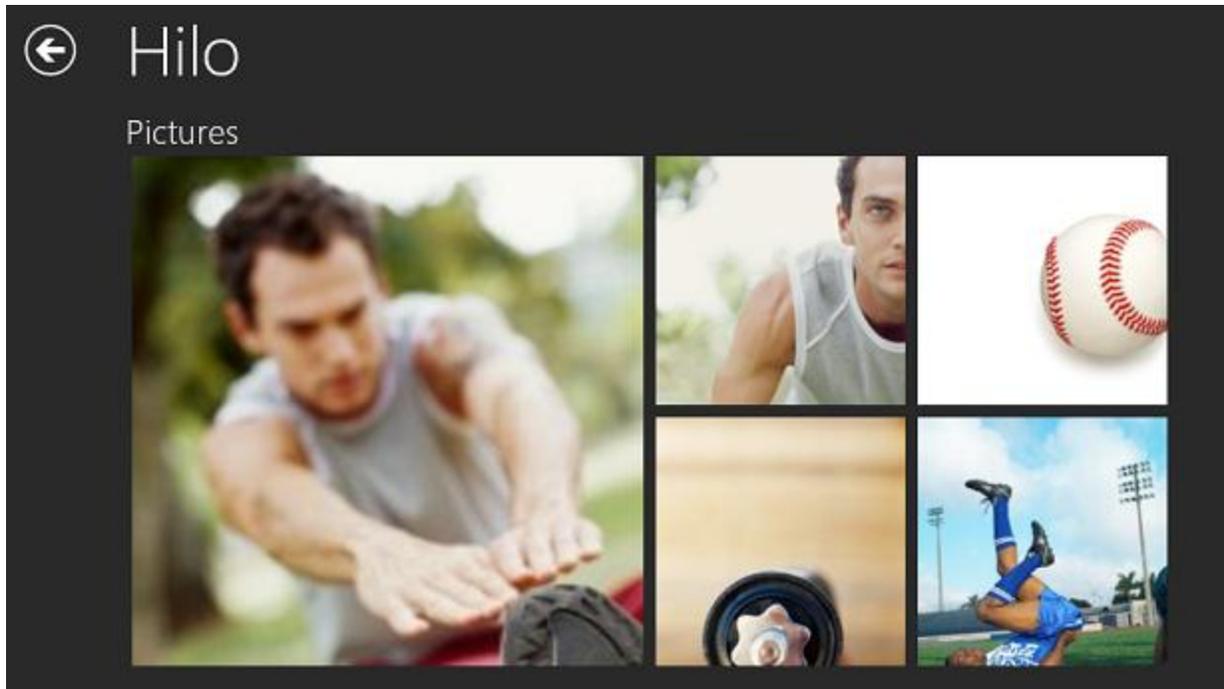
### Adding New Pages to the Project

There are five pages in the Hilo app. These are:

- The main hub page, which shows a small subset of photos and provides links to other views.
- The month view page, where you can view photos by group.
- The details page, where you can select a photo to crop or rotate.
- The crop page, which allows you to edit a photo.
- The rotate page, which allows you to rotate a photo.

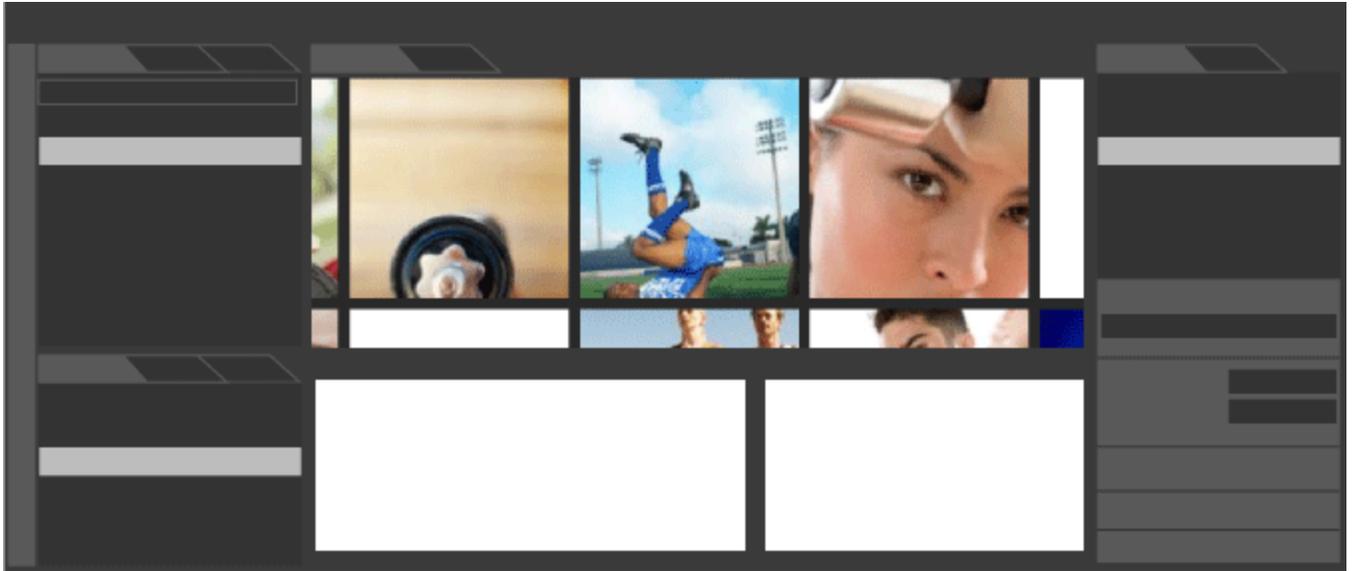
When we built the app, Hilo developers added each page to the project by using the **Add New Item** dialog box in Visual Studio. Each page is a separate [PageControl](#) with an associated HTML, CSS, and JavaScript file.

The hub page is the start page you see when you run the app. Here is what the main hub page looks like.



## Designing pages in Visual Studio and Blend

Hilo C++ developers used Blend for Microsoft Visual Studio 2012 for Windows 8 to fine-tune the shared UX design for Hilo C++ and Hilo JavaScript. Blend helped with the design of pages and controls. For more info on the shared UX design, see [Designing the UX](#). For Hilo JavaScript, we used Blend primarily as a versatile CSS editor. In addition to providing a live DOM view, Blend supports a feature called Interactive Mode that allows you to run the app and pause it in a particular state. You can view the paused app on the Design Surface, edit CSS, and see the effects immediately. We found this feature useful to fine-tune the page presentation. The following illustration shows a paused app running in the Design Surface. The CSS properties pane on the right provides information about how the styles are applied to the page and allows you to change values.



For more info on using Blend, see [Blend for Visual Studio 2012](#).

We recommend that you use Visual Studio to work with the code-focused aspects of your app. Visual Studio works best for writing JavaScript, running, and debugging your app. The **Refresh Windows app** button  in Visual Studio allowed us to iterate quickly through code modifications by reloading updated HTML, CSS, and JavaScript code without restarting the debugger.

For more info on designing an app with Visual Studio, see [Create your first Windows Store app using JavaScript](#). That topic provides an introduction that shows you how to design a simple page in a Windows Store app.

## Navigation model in Hilo

Pages in the Hilo app are page controls that support a single-page navigation model, which is the preferred navigation model for Windows Store apps using JavaScript. Each page control represents content that the user can navigate to. Pages contain other controls. The single-page navigation model provides a smoother, app-like transition between pages, and also makes it easier to manage state, because scripts are never unloaded. When we created Hilo, we found that the fastest route to a functional navigation model was to use the single-page model. In this way, we could benefit by re-using existing, tested navigation code. When you use single-page navigation in very large applications, you may need to more actively manage memory to improve performance.

In this navigation model, HTML pages (the page controls) are loaded into a single content host, a DIV element declared in default.html. In default.html, the content host DIV element is declared as a control of type **PageControlNavigator** using the HTML [data-win-control](#) attribute that is provided by Windows Library for JavaScript.

**HTML: Hilo\Default.html**

```
<div id="contenthost" data-win-control="Hilo.PageControlNavigator" data-win-
options="{home: '/Hilo/hub/hub.html'}">
</div>
```

The **PageControlNavigator** is implemented in PageControlNavigator.js. In Hilo, PageControlNavigator.js is a modified version of the navigator.js file that is included with Visual Studio project templates, such as Grid and Split. For more information about the single-page navigation model used in some Windows Store apps, see [QuickStart: using single-page navigation](#).

We made a few modifications to navigator.js for Hilo, such as renaming the file and changing the namespace to fit the Hilo naming conventions. We also replaced some DOM level-0 style event registrations with the use of the DOM level 2 **addEventListener**.

We also implemented a **reload** function (not shown) in the modified navigator.js file. We added the **reload** function to support scenarios in which a user makes a change in the file system's Pictures folder while the app is running. We capture the file system change by using the file query's [contentschanged](#) event. The listener for the event is implemented in the contentChangedListener.js file.

**Navigating between pages in the single-page navigation model**

To load a page into the content host DIV element, the app calls [WinJS.Navigation.navigate](#), passing in the page URL. The **navigate** function is used to navigate to a page control. In Hilo pages, you can find a typical example of using the **navigate** function in **itemClicked**, which is invoked when a picture is clicked or tapped. In this code, **nav** contains a reference to **WinJS.Navigation**.

**JavaScript: Hilo\Hilo\hub\hubPresenter.js**

```
itemClicked: function (args) {

    // Get the `Hilo.Picture` item that was bound to the invoked image,
    // and the item index from the list view control.
    var picture = args.detail.item.data;

    // Build the query that can find this picture within it's month group.
    var options = this.buildQueryForPicture(picture);

    // Navigate to the detail view, specifying the month query to
    // show, and the index of the individual item that was invoked.
    this.nav.navigate("/Hilo/detail/detail.html", options);
},
```

[WinJS.Navigation.navigate](#) does not directly navigate to a page, but invokes the [WinJS.Navigation.onnavigated](#) event. The handler for **WinJS.Navigation.onnavigated** is implemented in PageControlNavigator.js. The main job of the handler is to call a Windows Library for JavaScript page

control function ([render](#)) that loads the actual page. The code in the handler, not shown, is unmodified from Navigator.js.

Each page control implements a [ready](#) function that automatically runs when the page loads. The implementation of the **ready** function is different for each page. For more information on feature implementation in the pages, see [Using Controls](#).

## Creating and loading pages

To create a new Hilo page, we used Visual Studio to add a Page Control item template (**Add > New Item**). By using the template, in one step we added an HTML, CSS, and JavaScript file to the project. These files included default content and were already wired up to support the single-page navigation model. We then modified the template to work as a Hilo page.

Hilo pages use a custom base object for all page controls. We implemented the base object so that we could combine common page logic and centralize some tasks more easily than we could in the default item template. In the base object, we included code to handle the following tasks:

- Register pages, such as "Hilo/hub/hub.html".
- Process localized string resources.
- Process clickable subtitles that are present in most of the pages.
- Handle deserialization when an app resumes after being suspended.

To make a new page use our base page object, we replace the call to [WinJS.UI.Pages.define](#) in the item template code with a call to **Hilo.controls.pages.define**. We moved the call to **WinJS.UI.Pages.define**, which is required to create a page control, to the base page object. The arguments passed to **define** specify the name of the page and the methods implemented on the page. The following code example is from hub.js. In this code, the methods implemented on the page include [ready](#), **updateLayout**, and **unload**.

### JavaScript: Hilo\Hilo\hub\hub.js

```
Hilo.controls.pages.define("hub", {  
  
    ready: function (element, options) {  
  
        // Handle the app bar button clicks for showing and hiding the app bar.  
        var appBarEl = document.querySelector("#appbar");  
        var hiloAppBar = new Hilo.Controls.HiloAppBar.HiloAppBarPresenter(appBarEl,  
WinJS.Navigation);  
  
        // Handle selecting and invoking (clicking) images.  
        var listViewEl = document.querySelector("#picturesLibrary");  
        this.listViewPresenter = new Hilo.Hub.ListViewPresenter(listViewEl,  
Windows.UI.ViewManagement.ApplicationView);  
  
        // Coordinate the parts of the hub page.  
        this.hubViewPresenter = new Hilo.Hub.HubViewPresenter(  

```

```

        WinJS.Navigation,
        hiloAppBar,
        this.listViewPresenter,
        new Hilo.ImageQueryBuilder()
    );

    this.hubViewPresenter
        .start(knownFolders.picturesLibrary)
        .then(function () {
            WinJS.Application.addEventListener("Hilo:ContentsChanged",
Hilo.navigato.r.reload);
        });
    },

    updateLayout: function (element, viewState, lastViewState) {
        this.listViewPresenter.setViewState(viewState, lastViewState);
    },

    unload: function () {
        WinJS.Application.addEventListener("Hilo:ContentsChanged",
Hilo.navigato.r.reload);
        Hilo.UrlCache.clearAll();
        this.hubViewPresenter.dispose();
        this.hubViewPresenter = null;
    }
});

```

The **define** function for the base object, implemented in `pages.js`, is exposed for use in the app with [WinJS.Namespace.define](#).

#### JavaScript: Hilo\Hilo\controls\pages.js

```

WinJS.Namespace.define("Hilo.controls.pages", {
    define: define
});

```

In the `define` function, we set the URL for the page and then pass the page-specific ready function into a call to **wrapWithCommonReady**.

#### JavaScript: Hilo\Hilo\controls\pages.js

```

function define(pageId, members) {

    var url = "/Hilo/" + pageId + "/" + pageId + ".html";

    members.ready = wrapWithCommonReady(members.ready);
    members.bindPageTitle = bindPageTitle;
}

```

```
return WinJS.UI.Pages.define(url, members);
}
```

**wrapWithCommonReady** takes the page's [ready](#) function as input, and returns a new wrapper function. In the wrapper function, we added additional code that is common to all the Hilo pages. This includes, for example, the call to [WinJS.Resources.processAll](#), which processes the localized string resources (.resjson files). It also includes a call to handle deserialization if the app is resuming after being suspended. The wrapper function includes a call to the page's **ready** function in this line: `return ready(element, options)`. By centralizing these tasks, we were able to reduce page dependencies, avoid repeating code on different pages, and make page behavior more consistent.

#### JavaScript: Hilo\Hilo\controls\pages.js

```
function wrapWithCommonReady(pageSpecificReadyFunction) {
    pageSpecificReadyFunction = pageSpecificReadyFunction || function () { };
    return function (element, options) {
        processLinks();

        // Handle localized string resources for the page.
        WinJS.Resources.processAll();

        // Ensure that the `options` argument is consistent with expectations,
        // for example, that it is properly deserialized when resuming.
        Hilo.controls.checkOptions(options);

        // We need to bind the `pageSpecificReadyFunction` function explicitly,
        // otherwise it will lose the context that the developer expects (that is,
        // it will not resolve `this` correctly at execution time.
        var ready = pageSpecificReadyFunction.bind(this);

        // Invoke the custom `ready`.
        return ready(element, options);
    };
}
```

When **wrapWithCommonReady** returns, we use its return value (the wrapper function) to set the new [ready](#) function for the page. In **define**, we then call a function to set the page title and finally call [WinJS.UI.Pages.define](#). We pass this method the URL and the page members. The **WinJS.UI.Pages.define** function is required to define a page control.

#### JavaScript: Hilo\Hilo\controls\pages.js

```
members.ready = wrapWithCommonReady(members.ready);
members.bindPageTitle = bindPageTitle;
```

```
return WinJS.UI.Pages.define(url, members);
```

When the app runs, the new [ready](#) function gets called automatically after a call to [navigate](#). Then, the new **ready** calls the page's **ready** function. Here's the **ready** function for the hub page.

#### JavaScript: Hilo\Hilo\hub\hub.js

```
ready: function (element, options) {

    // Handle the app bar button clicks for showing and hiding the app bar.
    var appBarEl = document.querySelector("#appbar");
    var hiloAppBar = new Hilo.Controls.HiloAppBar.HiloAppBarPresenter(appBarEl,
WinJS.Navigation);

    // Handle selecting and invoking (clicking) images.
    var listViewEl = document.querySelector("#picturesLibrary");
    this.listViewPresenter = new Hilo.Hub.ListViewPresenter(listViewEl,
Windows.UI.ViewManagement.ApplicationView);

    // Coordinate the parts of the hub page.
    this.hubViewPresenter = new Hilo.Hub.HubViewPresenter(
        WinJS.Navigation,
        hiloAppBar,
        this.listViewPresenter,
        new Hilo.ImageQueryBuilder()
    );

    this.hubViewPresenter
        .start(knownFolders.picturesLibrary)
        .then(function () {
            WinJS.Application.addEventListener("Hilo:ContentsChanged",
Hilo.navigato.r.reload);
        });
},
```

The **ready** function here creates presenters for the view. Hilo implements a Model-View-Presenter (MVP) pattern. For more info, see [Using a separated presentation pattern](#).

## Loading the hub page

When the Hilo app starts up, the hub page gets loaded into the content host DIV element declared in default.html. The hub page is specified as the home page in the content host DIV using the Windows Library for JavaScript [data-win-options](#) attribute.

**HTML: Hilo\default.html**

```
<div id="contenthost" data-win-control="Hilo.PageControlNavigator" data-win-
options="{home: '/Hilo/hub/hub.html'}">
</div>
```

In PageControlNavigator.js, the [data-win-options](#) value gets passed into the constructor function for the **PageControlNavigator**. The home page is assigned to the navigator object. In addition, this code specifies **Hilo** as the namespace for the navigator object.

**JavaScript: PageControlNavigator.js**

```
WinJS.Namespace.define("Hilo", {
  PageControlNavigator: WinJS.Class.define(

    // Define the constructor function for the PageControlNavigator.
    function PageControlNavigator(element, options) {
      var body = document.body;

      // . . .

      this.home = options.home;
      // . . .
      Hilo.navigator = this;
    },
```

To load the hub page into the content host DIV element, the app calls [navigate](#). The app passes in the home value of the **PageControlNavigator**. The following code is found in the **activated** event handling function in default.js.

**JavaScript: Hilo\default.js**

```
if (nav.location) {
  nav.history.current.initialPlaceholder = true;
  return nav.navigate(nav.location, nav.state);
} else {
  return nav.navigate(Hilo.navigator.home);
}
```

## Establishing the Data Binding

Each page of the Hilo app includes data binding. Data binding links the model (data source) to the view. In Hilo, presenter classes are responsible for assigning the data source to the view. For more information see [Using a separated presentation pattern](#) in this guide. For additional info on declarative binding used in Hilo, see [Using Controls](#).

## Supporting portrait, snap, and fill layouts

We designed Hilo to be viewed full-screen in the landscape orientation. Windows Store apps such as Hilo must adapt to different application view states, including both landscape and portrait orientations. Hilo supports [fullScreenLandscape](#), **filled**, **fullScreenPortrait**, and **snapped** layouts. Hilo uses the [Windows.UI.ViewManagement.ApplicationView](#) class to get the current view state. It uses the [resize](#) event to handle changes to the visual display to support each layout.

The **PageControlNavigator** registers for the [resize](#) event in its constructor.

### JavaScript: Hilo\Hilo\PageControlNavigator.js

```
function PageControlNavigator(element, options) {
    // . . .

    window.onresize = this._resized.bind(this);

    // . . .
},
```

When a [resize](#) event occurs, the handler for the event calls the **updateLayout** function that is implemented in Hilo pages. The **appView** variable contains a reference to the [ApplicationView](#) object, the value of which gets passed as the view state.

### JavaScript: Hilo\Hilo\PageControlNavigator.js

```
_resized: function (args) {
    if (this.pageControl && this.pageControl.updateLayout) {
        this.pageControl.updateLayout.call(this.pageControl, this.pageElement,
appView.value, this._lastViewState);
    }
    this._lastViewState = appView.value;
},
```

The following example shows the implementation of **updateLayout** for the hub page. In the hub page, **updateLayout** calls the [ListView](#) presenter's **setViewState** function. It then passes along the view state.

### JavaScript: Hilo\Hilo\hub\hub.js

```
updateLayout: function (element, viewState, lastViewState) {
    this.listViewPresenter.setViewState(viewState, lastViewState);
},
```

In the [ListView](#) presenter, **setViewState** sets the current layout of the hub page's **ListView** control by using the value returned from **selectLayout**. It then forwards the view state to **selectLayout**.

**JavaScript: Hilo\Hilo\hub\listViewPresenter.js**

```
setViewState: function (viewState) {  
    this.lv.layout = this.selectLayout(viewState);  
},
```

**selectLayout** checks whether the current view state is in snapped mode or not. If it is, it returns a [ListLayout](#) object, which represents a vertical list. If the view state is not snapped, it returns a [GridLayout](#) object, which represents a horizontal list.

**JavaScript: Hilo\Hilo\hub\listViewPresenter.js**

```
selectLayout: function (viewState, lastViewState) {  
  
    if (lastViewState === viewState) { return; }  
  
    if (viewState === appViewState.snapped) {  
        return new WinJS.UI.ListLayout();  
    }  
    else {  
        var layout = new WinJS.UI.GridLayout();  
        layout.groupInfo = function () { return listViewLayoutSettings; };  
        layout.maxRows = 3;  
        return layout;  
    }  
},
```

Some pages don't need to do anything special to handle view states, so **updateLayout** is not implemented for all pages.

## Using controls in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Use binding to connect your UI to data.
- Use templates to format and display multiple instances of data.
- Use standard controls so that your app is touch and pointer enabled.

Controls are the core UI objects of the Windows Library for JavaScript. You can use data binding with controls to display data on the UI at run time. Here we'll look at the [ListView](#), [FlipView](#), [AppBar](#), and [SemanticZoom](#) Windows Library for JavaScript controls, and the [canvas](#) and [img](#) HTML elements.

### You will learn

- How to use binding to declaratively connect your UI to data.
- How to apply templates to conveniently format and display multiple instances of data.
- How to use some of commonly used Windows Library for JavaScript controls.

### Common controls used in Hilo

Here are the main controls that we used in Hilo. You can also refer to the HTML files in the Hilo folder in the Microsoft Visual Studio project.

- [ListView](#)
- [FlipView](#)
- [AppBar](#)
- [SemanticZoom](#)
- [canvas](#)
- [img](#)

The [ListView](#), [FlipView](#), [AppBar](#), and [SemanticZoom](#) controls are Windows Library for JavaScript controls. The [canvas](#) and [img](#) controls are HTML elements.

**Tip** See [Controls list](#) and [Controls by function](#) for all available controls. Bookmark these pages so that you can come back to them when you want to add another feature to your app. Also use the **Toolbox** window in Visual Studio and the **Assets** tab in Blend for Microsoft Visual Studio 2012 for Windows 8 to browse and add controls to your pages.

All of these controls display run time data on the UI through binding. Binding provides a data-bound property value, which defers the value until run time. Binding is key to effectively using the Windows Library for JavaScript, because it enables you to declaratively connect your UI to data. Binding also helps you to cleanly separate your UI from your data when you design your app, because data is resolved at run time. For Hilo, binding data at run time is critical because we don't know anything about the user's Pictures library at design time.

In order to minimize memory leaks when you bind data to the UI, you must set the [optimizeBindingReferences](#) property to true in your startup code. Here's the code.

#### JavaScript: Hilo\default.js

```
WinJS.Binding.optimizeBindingReferences = true;
```

**Note** When you perform declarative binding, you should always set the [optimizeBindingReferences](#) property to true in your startup code. If you do not do that, the bindings in your app may leak memory. For more info see [JavaScript project templates for Windows Store apps](#).

Hilo typically binds text content to the UI through a [span](#) element. Here's the code that uses binding to display the page title on the detail page, with the page title being the month and year of the photo.

#### HTML: Hilo\Hilo\detail\detail.html

```
<span id="pageTitle" class="pagetitle" data-win-bind="innerText: title">[Month Year]</span>
```

The [data-win-bind](#) attribute binds a property of an element to a property of a data source, as a one-time binding.

**Note** When specifying a [data-win-bind](#) attribute, you can specify multiple sets of element/data source property pairs by separating them with a semicolon.

To display the data specified in the binding you must call the [WinJS.Binding.processAll](#) function. Here's the code.

#### JavaScript: Hilo\Hilo\controls\pages.js

```
function bindPageTitle(title) {
    // Bind the title based on the query's month/year.
    var pageTitleEl = document.querySelector("#pageTitle");
    WinJS.Binding.processAll(pageTitleEl, { title: title });
}
```

Here, two arguments are specified in the call to the [processAll](#) function. The first argument represents the *rootElement* parameter, which is the element you traverse to find elements for binding. The second argument represents the *dataContext* parameter, which is the object to use for data binding. In the second argument, the first use of *title* denotes the property of the data source specified in the [data-win-bind](#) attribute above. The second use of *title* represents the function parameter.

**Note** The *rootElement* parameter can be omitted from the call to the [processAll](#) function. This has the effect of searching the entire document to find elements to bind to.

The function binds the value of the data from the *dataContext* parameter to the elements that are descendants of the *rootElement* parameter, when those descendants have the [data-win-bind](#) attribute

specified. The function then returns a promise that completes when every item that contains binding declarations has been processed.

For more info about binding, see [Quickstart: binding data and styles](#), [How to bind a complex object](#), and see [Using a separated presentation pattern](#) in this guide.

## ListView

The [ListView](#) control displays data from an [IListDataSource](#) in a customizable list or grid. Items in a **ListView** control can contain other controls, but they can't contain a [FlipView](#) or another **ListView**. In Hilo, the data displayed by **ListView** controls are typically specified by a template.

Windows Library for JavaScript templates are a convenient way to format and display multiple instances of data. Hilo uses templates in conjunction with [ListView](#) and [FlipView](#) controls to specify the way to display data in the controls.

A template is commonly defined declaratively by creating a DIV element for the template, and adding a [data-win-control](#) attribute that has a value of [WinJS.Binding.Template](#). This makes the DIV element host the specified Windows Library for JavaScript control. Here's the code for the template used by the [ListView](#) control on the hub page.

### HTML: Hilo\Hilo\hub\hub.html

```
<div id="hub-image-template" data-win-control="WinJS.Binding.Template">
  <div data-win-bind="style.backgroundImage: url.backgroundUrl; alt: name;
className: className" class="thumbnail">
    </div>
  </div>
```

A template object must have a single root element, which can also serve as a parent for the template's contents. Here, the root element sets the [data-win-bind](#) attribute to bind the value of the **style.backgroundImage** property to the value of the **url.backgroundUrl** property, along with some additional property bindings. The **class** attribute of the DIV element is set to use the **thumbnail** styling from the **hub.css** file.

**Note** Templates must be defined before they are used.

The [ListView](#) control can then use the template to display data. Here's the code.

### HTML: Hilo\Hilo\hub\hub.html

```
<div id="picturesLibrary"
  data-win-control="WinJS.UI.ListView"
  data-win-options="{
  itemTemplate: select('#hub-image-template'),
  selectionMode: 'single'}" />
```

The **itemTemplate** property associates the [ListView](#) control with the previously defined template, named **hub-image-template**. You set the **itemTemplate** property in the [data-win-options](#) attribute.

In the **ready** function of the page control, `hub.js`, the [ListView](#) control is returned using CSS query selector syntax. Instances of the **ListViewPresenter** and **HubViewPresenter** classes are initialized, and the **start** function of the **HubViewPresenter** instance is then called. Here's the code.

#### JavaScript: Hilo\Hilo\hub\hub.js

```
ready: function (element, options) {

    // Handle the app bar button clicks for showing and hiding the app bar.
    var appBarEl = document.querySelector("#appbar");
    var hiloAppBar = new Hilo.Controls.HiloAppBar.HiloAppBarPresenter(appBarEl,
WinJS.Navigation);

    // Handle selecting and invoking (clicking) images.
    var listViewEl = document.querySelector("#picturesLibrary");
    this.listViewPresenter = new Hilo.Hub.ListViewPresenter(listViewEl,
Windows.UI.ViewManagement.ApplicationView);

    // Coordinate the parts of the hub page.
    this.hubViewPresenter = new Hilo.Hub.HubViewPresenter(
        WinJS.Navigation,
        hiloAppBar,
        this.listViewPresenter,
        new Hilo.ImageQueryBuilder()
    );

    this.hubViewPresenter
        .start(knownFolders.picturesLibrary)
        .then(function () {
            WinJS.Application.addEventListener("Hilo:ContentsChanged",
Hilo.navigato.r.reload);
        });
},
```

The **start** function executes a query to return images to populate the [ListView](#) control with. It then calls the **loadImages** function, which in turn calls the **bindImages** function. The **bindImages** function calls the **setDataSource** function of the **ListViewPresenter** class, in order to bind the **ListView** to the data source. Here's the code for the **ListViewPresenter.setDataSource** function.

#### JavaScript: Hilo\Hilo\hub\listViewPresenter.js

```
setDataSource: function (items) {
    this.lv.itemDataSource = new WinJS.Binding.List(items).dataSource;
},
```

The [ListView](#) control is referenced through the **lv** variable. For binding, the **ListView** control requires a data source that implements [IListDataSource](#). For in-memory arrays of data, Hilo uses [WinJS.Binding.List](#). The [itemDataSource](#) property takes an **IListDataSource** object. However, the **List** object is not an **IListDataSource**, but it does have a [dataSource](#) property that returns an **IListDataSource** version of itself.

When the instance of the **ListViewPresenter** is created in the **ready** function of the page control, `hub.js`, the **ListViewPresenter** constructor calls the **setup** method. This sets the [layout](#) property of the [ListView](#) control to the data returned by the **selectLayout** method. The **selectLayout** method defines the layout of the photo thumbnails on the hub page. It creates a new instance of the [WinJS.UI.GridLayout](#) class, sets the [groupInfo](#) property of the instance to the settings contained in the **listViewLayoutSettings** variable, and sets the [maxRows](#) property of the instance to 3. Then it returns the instance. Here's the code.

#### JavaScript: Hilo\Hilo\hub\listViewPresenter.js

```
selectLayout: function (viewState, lastViewState) {

    if (lastViewState === viewState) { return; }

    if (viewState === appViewState.snapped) {
        return new WinJS.UI.ListLayout();
    }
    else {
        var layout = new WinJS.UI.GridLayout();
        layout.groupInfo = function () { return listViewLayoutSettings; };
        layout.maxRows = 3;
        return layout;
    }
},
```

#### JavaScript: Hilo\Hilo\hub\listViewPresenter.js

```
var listViewLayoutSettings = {
    enableCellSpanning: true,
    cellWidth: 200,
    cellHeight: 200
};
```

**Note** The property settings in the **listViewLayoutSettings** variable must correspond to the height and width values specified in the CSS for the items. They need to be the greatest common denominator of the different widths and heights.

The CSS for the hub page specifies that each photo thumbnail will have a width and height of 200 pixels, with the exception of the first thumbnail which will have a width and height of 410 pixels. 410 pixels is the width and height of a standard thumbnail doubled, plus 10 pixels to account for the margin between the grid cells. Here's the code.

**JavaScript: Hilo\Hilo\hub\hub.css**

```
.hub section[role=main] .thumbnail {
  width: 200px;
  height: 200px;
}

.hub section[role=main] .thumbnail.first {
  width: 410px;
  height: 410px;
}
```

For more info about binding by using templates, see [How to use templates to bind data](#). For more info about the [ListView](#) control, see [Quickstart: Adding a ListView](#). For more info about how we used the **ListView** control to help navigate between large sets of pictures, see [Working with data sources](#) in this guide.

**FlipView**

The [FlipView](#) control displays a collection of items, such as a set of photos, and lets you flip through them one at a time. In Hilo, the data displayed by the **FlipView** control on the detail page is specified by a template.

Windows Library for JavaScript templates are a convenient way to format and display multiple instances of data. Hilo uses templates in conjunction with [ListView](#) and [FlipView](#) controls to specify the way to display data in the controls.

A template is commonly defined declaratively by creating a DIV element for the template, and adding a [data-win-control](#) attribute that has a value of [WinJS.Binding.Template](#). This makes the DIV element host the specified Windows Library for JavaScript control. Here's the code for the template used by the [FlipView](#) control.

**HTML: Hilo\Hilo\detail\detail.html**

```
<div id="image-template" data-win-control="WinJS.Binding.Template">
  <div class="flipViewBackground">
    <div class="flipViewImage" data-win-bind="backgroundImage: src
Hilo.Picture.bindToImageSrc">
      </div>
    </div>
  </div>
</div>
```

A template object must have a single root element, **flipViewBackground**, to serve as a parent for the template's contents. Here, the parent element contains only one child element, **flipViewImage**. This element's [data-win-bind](#) attribute binds the **backgroundImage** property to the **src** property. The **src** property is set by the **Hilo.Picture.bindToImageSrc** function.

The [FlipView](#) control can then use the template to display data. Here's the code.

#### HTML: Hilo\Hilo\detail\detail.html

```
<div id="flipview"
  data-win-control="WinJS.UI.FlipView"
  data-win-options="{
    itemTemplate: select('#image-template')}">
</div>
```

The **itemTemplate** property, which is set in the [data-win-options](#) attribute, associates the [FlipView](#) control with the previously-defined template, named **image-template**.

**Note** The [FlipView](#) control does not dynamically adjust its height to fit the content. For a **FlipView** to render, you must specify an absolute value for its height.

In the **ready** function of the page control, detail.js, the [FlipView](#) control is returned using CSS query selector syntax. An instance of the **DetailPresenter** class is initialized, and the **start** function of the **DetailPresenter** instance is then called. Here's the code.

#### JavaScript: Hilo\Hilo\detail\detail.js

```
ready: function (element, options) {

    var query = options.query;
    var queryDate = query.settings.monthAndYear;
    var pageTitle = Hilo.dateFormatter.getMonthFrom(queryDate) + " " +
Hilo.dateFormatter.getYearFrom(queryDate);
    this.bindPageTitle(pageTitle);

    var hiloAppBarEl = document.querySelector("#appbar");
    var hiloAppBar = new Hilo.Controls.HiloAppBar.HiloAppBarPresenter(hiloAppBarEl,
WinJS.Navigation, query);

    var filmstripEl = document.querySelector("#filmstrip");
    var flipviewEl = document.querySelector("#flipview");

    var flipviewPresenter = new Hilo.Detail.FlipviewPresenter(flipviewEl);
    var filmstripPresenter = new Hilo.Detail.FilmstripPresenter(filmstripEl);

    var detailPresenter = new Hilo.Detail.DetailPresenter(filmstripPresenter,
flipviewPresenter, hiloAppBar, WinJS.Navigation);
    detailPresenter.addEventListener("pageSelected", function (args) {
        var itemIndex = args.detail.itemIndex;
        options.itemIndex = itemIndex;
    });

    detailPresenter
```

```

        .start(options)
        .then(function () {
            WinJS.Application.addEventListener("Hilo:ContentsChanged",
Hilo.navigator.reload);
        });
    },

```

The **start** function executes a query to return images to populate the [FlipView](#) control. Then the **DetailPresenter.bindImages** function is called to create a new instance of the **FlipviewPresenter** class. The **FlipviewPresenter** constructor in turn calls the **FlipviewPresenter.bindImages** function to bind the **FlipView** control to the data source. Here's the code for the **FlipviewPresenter.bindImages** function.

#### JavaScript: Hilo\Hilo\detail\flipviewPresenter.js

```

bindImages: function (images) {
    this.bindingList = new WinJS.Binding.List(images);
    this.winControl.itemDataSource = this.bindingList.dataSource;
},

```

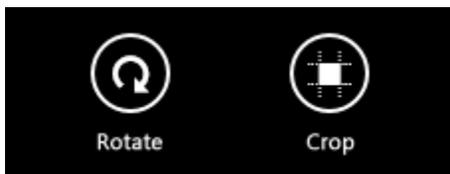
You reference the [FlipView](#) control through the **winControl** variable. For binding, the **FlipView** control requires a data source that implements [IListDataSource](#). For in-memory arrays of data, Hilo uses [WinJS.Binding.List](#). The [itemDataSource](#) property takes an **IListDataSource** object. The **List** object is not an **IListDataSource**, but it does have a [dataSource](#) property that returns an **IListDataSource** version of itself.

For more info about binding using templates, see [How to use templates to bind data](#), and see [Working with data sources](#) in this guide. For more info about the [FlipView](#) control, see [Quickstart: Adding a FlipView](#).

## AppBar

The [AppBar](#) control is a toolbar for displaying app-specific commands. Hilo displays a bottom app bar on every page, and a top app bar on the image view page that displays a filmstrip view of all photos for the current month.

Here's what the buttons look like on the bottom app bar on the image view page.



Place buttons that enable navigation or critical app features on a page. Place buttons in an app bar if they are not critical to the navigation and use of your app. For example, on the month view page, we enable the user to click on the text for a given month to jump to all photos for that month, because we

felt that it was crucial to navigation. We added the rotate and crop commands to the app bar because these are secondary commands and we didn't want them to distract the user.

The location of the app bar on a page is controlled by its [placement](#) property. Hilo has an image navigation control that provides a re-usable implementation of the bottom app bar. This control can be used to navigate to the rotate and crop pages. Here's the HTML for the bottom app bar that appears on the image view page.

#### HTML: Hilo\Hilo\detail\detail.html

```
<section id="image-nav" data-win-control="WinJS.UI.HtmlControl" data-win-options="{uri: '/Hilo/controls/HiloAppBar/hiloAppBar.html'}"></section>
```

Per UI guidelines for app bars, the app bar will automatically hide labels and adjust padding when in snapped or portrait orientation. For more info about app bars, see [Quickstart: adding an app bar with commands](#), and [Guidelines and checklist for app bars](#). For more info about how we used the [AppBar](#) control in Hilo, see [Swipe from edge for app commands](#) in this guide.

## SemanticZoom

The [SemanticZoom](#) control lets the user zoom between two views of a collection of items. For more info about how we use this control to help navigate between large sets of pictures, see [Pinch and stretch to zoom](#) in this guide.

## Canvas

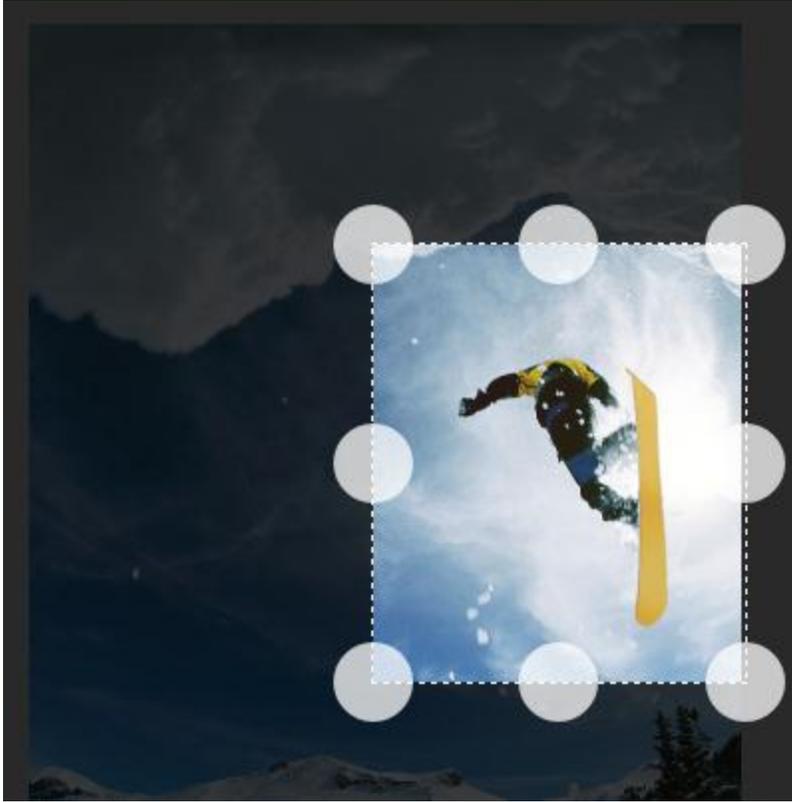
The [canvas](#) HTML element provides an object that can be used for drawing, rendering, and manipulating images and graphics on a document. We used this element to perform visual cropping of the photo, to show what the crop result will look like when the file is saved.

Here's the HTML for the [canvas](#).

#### HTML: Hilo\Hilo\crop\crop.html

```
<div class="canvas-wrapper">  
  <canvas id="cropSurface"></canvas>  
</div>
```

Here's what the Hilo crop UX looks like.



Manipulation of the canvas element occurs in JavaScript. The code obtains the canvas-based crop selection, and then calculates the selected area of the original image by scaling the canvas-based selection to the original image dimensions. The on-screen image is then cropped, to show what the crop result will look like when the file is saved. Multiple cropping operations are supported by storing an offset for the starting location of the crop on the original image, rather than relative to the canvas size. Here's the code.

#### JavaScript: Hilo\Hilo\crop\imageView.js

```
cropImage: function () {
    var selectionRectScaledToImage = this.getScaledSelectionRectangle();
    // Reset image scale so that it reflects the difference between
    // the current canvas size (the crop selection) and the original
    // image size, then re-draw everything at that new scale.
    this.imageToScreenScale =
this.calculateScaleToScreen(selectionRectScaledToImage);
    this.drawImageSelectionToScale(selectionRectScaledToImage,
this.imageToScreenScale);

    // Remember the starting location of the crop on the original image
    // and not relative to the canvas size, so that cropping multiple times
    // will correctly crop to what has been visually selected.
    this.image.updateOffset({ x: selectionRectScaledToImage.startX, y:
selectionRectScaledToImage.startY });
}
```

```
return this.canvasEl.toDataURL();
},
```

For more info about the crop UX, see [Using touch](#) in this guide.

## Img

We use the [img](#) HTML element to display photos on the rotate and crop pages of the app.



Here's the HTML for the [img](#) element on the rotate page.

### HTML: Hilo\Hilo\rotate\rotate.html

```

```

In the **ready** function of the page control, rotate.js, the [img](#) element is returned using CSS query selector syntax. An instance of the **RotatePresenter** class is initialized, and the **start** function of the **RotatePresenter** instance is then called. Here's the code.

### JavaScript: Hilo\Hilo\rotate\rotate.js

```
var imgEl = document.querySelector("#rotate-image");
this.presenter = new Hilo.Rotate.RotatePresenter(imgEl, this.appBarPresenter,
fileLoader, expectedName, touchProvider);
```

```
this.presenter.start();
```

The **start** function in turn calls the **\_loadAndShowImage** internal function. This takes the query result from the image query and displays the image that is loaded, in the **img** element. Here's the code for the **\_loadAndShowImage** function.

#### JavaScript: Hilo\Hilo\rotate\rotatePresenter.js

```
_loadAndShowImage: function (queryResult) {
    var self = this;

    if (queryResult.length === 0) {
        this.navigation.back();
    } else {
        var storageFile = queryResult[0].storageFile;

        if (storageFile.name !== this.expectedFileName) {
            this.navigation.back();
        } else {
            this.hiloPicture = new Hilo.Picture(storageFile);
            this.el.src = this.hiloPicture.src.url;

            return storageFile.properties
                .getImagePropertiesAsync()
                .then(function (props) {
                    self.imageProperties = props;
                    self.adjustImageSize();
                });
        }
    }
},
```

The **img** element is referenced through the **el** variable. Therefore, the code sets the **src** property of the **img** element to the URL of the image to be displayed.

**Note** The **img** element displays photos on the rotate and crop pages of the app. On the other pages, photos are displayed using the **backgroundImage** CSS property of a DIV element.

For more info about images, see [Displaying images, graphics, and thumbnails](#).

## Styling controls

Hilo's appearance was customized by styling the controls used in the app. To customize the appearance of controls you use CSS. A Windows Store app using JavaScript also supports some advanced control styling features, with the Windows Library for JavaScript providing a set of styles that give your app the Windows 8 look and feel.

For more info, see [Adding and styling controls](#), and [Quickstart: styling controls](#).

## Touch and gestures

The Windows Library for JavaScript runtime provides built-in support for touch. Because the runtime uses a common event system for many user interactions, you get automatic support for mouse, pen, and other pointer-based interactions. The exception to this is the rotate page, which uses the [GestureRecognizer](#) class to listen for and handle pointer and gesture events that enable the user to rotate the displayed photo.

**Tip** Design your app for the touch experience, and mouse and pen support come for free.

For more info about how we used touch in Hilo, see [Using touch](#).

## Testing controls

When you test your app, ensure that each control behaves as you expect in different configurations and orientations. When we used a control to add a new feature to the app, we ensured that the control behaved correctly in snap and fill views and under both landscape and portrait orientations.

If your monitor isn't touch-enabled, you can use the simulator to emulate pinch, zoom, rotation, and other gestures. You can also work with different screen resolutions. For more info, see [Running apps in the simulator](#).

You can test your app on a computer that doesn't have Visual Studio but has hardware you need to test. For more info, see [Running apps on a remote machine](#).

For more info on how we tested Hilo, see [Testing and deploying apps](#).

## Working with data sources in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Use a [ListView](#) when you need a flexible way to group and display data.
- When working with the [ListView](#), consider using a `WinJS.Binding.List`, an in-memory data source, as long as it provides a responsive UI.
- When working with the [ListView](#), consider using a custom data source for access to very large data sets.

### Important APIs

- [WinJS.Binding.List](#)
- [IListDataAdapter](#)

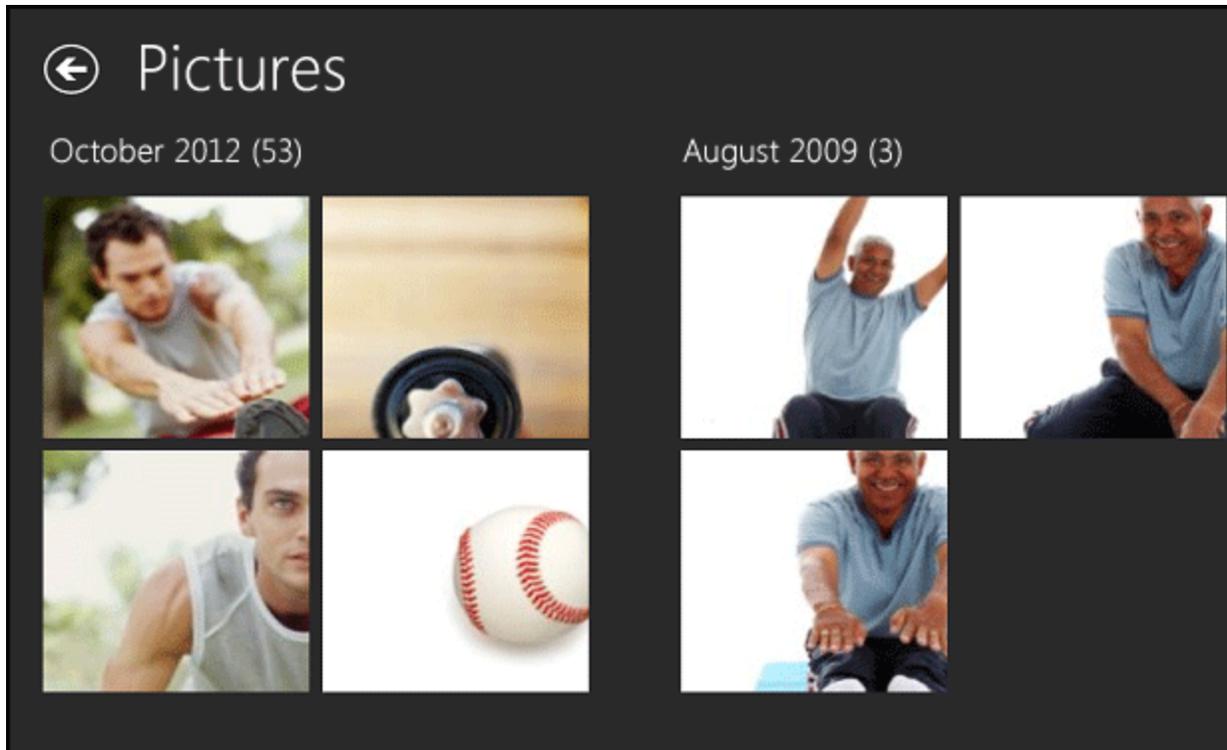
The pages in Hilo use an in-memory data source to bind data to the Windows Library for JavaScript [ListView](#) control. While developing the month page for Hilo, we initially implemented a custom data source instead, but later changed the implementation to use an in-memory data source because our user experience (UX) requirements changed.

### You will learn

- Tips on choosing the type of data source to implement in your app.
- How to manage grouping in both an in-memory data source ([WinJS.Binding.List](#)) and a custom data source.
- How to implement an in-memory data source and a custom data source.

### Data binding in the month page

In this topic, we'll take a look at the [ListView](#) control in the month page as an example of data binding with different types of data sources. For more info about declarative binding for other controls used in Hilo and non-array data sources, see [Using Controls](#). For more info about binding, see [Quickstart: binding data and styles](#). To meet UX requirements for the month page, we needed a flexible method to group and display a very large data set consisting of the images in the local pictures library. The month page, shown in the following screenshot, displays images in month-based and year-based groups.



The [ListView](#) provides the flexibility we need to group and display data. We considered whether it would make sense to implement our own custom control, but the **ListView** is the best choice for us because it provides appearance and behavior consistent with Windows Store apps, built-in support for the cross-slide gesture, and it's performance optimized. For example, the **ListView** handles UI virtualization, recycling elements when they go out of view without destroying objects.

The month page includes a [ListView](#) control that's used for the main view. The second [data-win-control](#) element is used for the zoomed-out view (showing more months) that's provided by the [SemanticZoom](#) control.

#### HTML: Hilo\Hilo\month\month.html

```
<div class="semanticZoomContainer" data-win-control="WinJS.UI.SemanticZoom">

  <div id="monthgroup" data-win-control="WinJS.UI.ListView" data-win-options="{
    layout: {type: WinJS.UI.GridLayout},
    selectionMode: 'single'}">
  </div>

  <div id="yeargroup" data-win-control="Hilo.month.YearList">
  </div>

</div>
```

Windows Library for JavaScript templates are used to format and display multiple instances of data. The month page uses several templates—three for the normal and snapped views (`id="month*"`) and two

for the zoomed-out view (`id="year*"`) provided by the [SemanticZoom](#) control. (For more info about data binding and Windows Library for JavaScript templates used in Hilo controls, see [Using controls](#).) For the month page, we'll associate the [ListView](#) to the templates programmatically in JavaScript code. For more info about binding the data source to these properties, see [Implementing group support with the in-memory data source](#).

Here's the HTML code for the Windows Library for JavaScript templates used for group items and group headers in the normal view.

#### HTML: Hilo\Hilo\month\month.html

```
<div id="monthItemTemplate" data-win-control="WinJS.Binding.Template">
  <div data-win-bind="style.backgroundImage: url.backgroundUrl; className:
className"></div>
</div>

<div id="monthGroupHeaderTemplate" data-win-control="WinJS.Binding.Template">
  <a class="monthLink" href="#"><span data-win-bind="innerHTML:
title"></span>&nbsp;<span data-win-bind="innerText: count"></span></a>
</div>

<div id="monthSnappedTemplate" data-win-control="WinJS.Binding.Template">
  <span data-win-bind="innerHTML: title"></span>&nbsp;<span data-win-
bind="innerText: count"></span>
  <div data-win-bind="style.backgroundImage: imageUrl;"
class="thumbnail"></div>
</div>
```

In the preceding code, the [data-win-bind](#) attribute values specify declarative binding for DOM properties like the DIV element's **backgroundImage** (a URL), **title**, for the group header name (a month name), and **count**, for the number of items in the group.

**Tip** You can also use the [data-win-options](#) attribute, specifically the [itemTemplate](#) and [groupHeaderTemplate](#) properties, to declaratively associate the [ListView](#) control with a specific Windows Library for JavaScript template.

## Choosing a data source for a page

The pages in Hilo use a [WinJS.Binding.List](#) for binding data to either the [ListView](#) or [FlipView](#) control.

For most pages, the [WinJS.Binding.List](#) is sufficient for our requirements, and is the only data source that type we implement. The **WinJS.Binding.List** provides a mechanism for creating groups from the primary data. The main limitation is that it's a synchronous data source, by which we mean that it is an in-memory data source (an array) and all its data must be available for display. Most Hilo pages display a somewhat limited file set. (The details page provides a set of files that belong to a single month, for example, and the hub page provides only six images in total.)

The month page uses a [ListView](#) to display images for all months in the pictures library. For the month page, we wanted to ensure high performance for a very large data set, so we initially implemented a custom data source. Using a custom data source, we could obtain data for the **ListView** one page at a time. Later, our UX requirements for the month page changed to match those of the C++ version of Hilo. Hilo C++ displays only eight pictures per month in the month page. The **ListView** interface for the custom data source expected contiguous data, in order for it to support groups, so we decided to switch to a [WinJS.Binding.List](#) implementation. This UX change also diminished our need to support a very large data set, which made the decision easier. For more info about using a custom data source, see [Other considerations: Custom data sources](#).

## Querying the file system

In the month page, we execute queries on Hilo's underlying data source, the file system, to obtain folders and files in the pictures library. For more info about the query builder used to execute queries, see [Using the query builder pattern](#).

To get images contained in a group, the month page uses the same query builder as other pages in Hilo. But to generate groups for the month page, Hilo uses a folder query instead of a file system query and passes a parameter ([groupByMonth](#)) to the [Windows.Storage.Search.QueryOptions](#) object. This query creates virtual folders based on the months read from each item's [System.ItemDate](#) property. In this code, **folder** contains a reference to the [Windows.Storage.KnownFolders.picturesLibrary](#).

### JavaScript: Hilo\Hilo\month\monthPresenter.js

```
_getMonthFoldersFor: function (folder) {
    var queryOptions = new search.QueryOptions(commonFolderQuery.groupByMonth);
    var query = folder.createFolderQueryWithOptions(queryOptions);
    return query.getFoldersAsync(0);
},
```

In the MonthPresenter, we will use the returned [StorageFolder](#) objects to create file queries for each month (not shown).

## Implementing group support with the in-memory data source

For ungrouped data, the use of the [WinJS.Binding.List](#) is fairly simple. After returning [StorageFile](#) objects from a file query, you need to pass in the array of items to a new instance of the list and assign its **dataSource** property to the [ListView itemDataSource](#) property.

### JavaScript: Hilo\Hilo\hub\listViewPresenter.js

```
setDataSource: function (items) {
    this.lv.itemDataSource = new WinJS.Binding.List(items).dataSource;
},
```

To use grouped data (pictures grouped into months), you need to assign the data to the [ListView](#) by using [itemDataSource](#), but you also need to do some extra work beforehand. For each picture returned from the month page's file query, we set a **groupKey** property. This is used by the [WinJS.Binding.List](#) to group data internally. In the month page, we set **groupKey** to a unique value by using a numerical representation of the year that has the month appended, as shown in the following code. This value enables us to implement sorting as required by the **List**, and to sort items in the correct order.

#### JavaScript: Hilo\Hilo\month\monthPresenter.js

```
var buildViewModels = foldersWithImages.map(function (folder) {
    promise = promise.then(function () {
        return folder.query
            .getFilesAsync(0, maxImagesPerGroup)
            .then(function (files) {
                filesInFolder = files;
                // Since we filtered for zero count, we
                // can assume that we have at least one file.

                return
files.getAt(0).properties.retrievePropertiesAsync([itemDateProperty]);
            })
            .then(function (retrieved) {
                var date = retrieved[itemDateProperty];
                var groupKey = (date.getFullYear() * 100) + (date.getMonth());
                var firstImage;

                filesInFolder.forEach(function (file, index) {
                    var image = new Hilo.Picture(file);
                    image.groupKey = groupKey;
                    self.displayedImages.push(image);

                    if (index === 0) {
                        firstImage = image;
                    }
                });
            });
    });
});
```

Prior to setting the group key, we also wrap each [StorageFile](#) in a **Hilo.Picture** object to make it bindable to the [ListView](#). (They are not bindable because Windows Runtime objects such as **StorageFile** are not mutable.) For more info, see [Using the query builder pattern](#).

To implement groups by using the [WinJS.Binding.List](#), we create a new instance of the **List** and call the [createGrouped](#) method. For grouping to work correctly, we pass **createGrouped** three functions:

- **groupKey**, to obtain the group key for each picture.
- **groupData**, to obtain an array of group data for each picture.
- **groupSort**, to implement sorting of the data.

This code shows the implementation of these functions, the creation of the [List](#) object, and the call to [createGrouped](#).

#### JavaScript: Hilo\Hilo\month\monthPresenter.js

```

_createDataSources: function (monthGroups) {
    var self = this;

    function groupKey(item) {
        return item.groupKey;
    }

    function groupData(item) {
        return self.groupsByKey[item.groupKey];
    }

    function groupSort(left, right) {
        return right - left;
    }

    var imageList = new
WinJS.Binding.List(this.displayedImages).createGrouped(groupKey, groupData,
groupSort);

```

In the preceding code, **groupData** calls the presenter's **groupsByKey** function to get the group information required by the [List](#). The group information has already been assigned to the month page presenter's **groupsByKey** property. Here's the code in which we set the group information properties and assign it to **groupsByKey**. (This code runs once for each month's file query, before creating the **List**.)

#### JavaScript: Hilo\Hilo\month\monthPresenter.js

```

var monthGroupViewModel = {
    itemDate: date,
    name: firstImage.name,
    backgroundImage: firstImage.src.backgroundUrl,
    title: folder.groupKey,
    sortOrder: groupKey,
    count: folder.count,
    firstItemIndexHint: firstItemIndexHint,
    groupKey: date.getFullYear().toString()
};

firstItemIndexHint += filesInFolder.size;
self.groupsByKey[groupKey] = monthGroupViewModel;
groups.push(monthGroupViewModel);

```

If you don't call the [createSorted](#) function of the [List](#) in your code, the **List** sorts data by passing the **groupKey** values to your implementation of the **groupSort** function. (See the preceding code.) In Hilo, this means that the month/year numerical value is used for sorting data.

It is also worth noting that **firstItemIndexHint** will be used to specify the raw index for the first item in each month group. This value is used to build a query specific to a month when a picture is selected (not shown), which causes the app to navigate to the details page.

## Other considerations: Using a custom data source

This section provides details about the working implementation of a custom data source that we initially created for Hilo. For Hilo source code that implements the custom data source, see [this version of Hilo](#).

In this section, we'll describe the following:

- Considerations in choosing a custom data source
- Implementing a data adapter for a custom data source
- Binding the `ListView` to a custom data source

### Considerations in choosing a custom data source

If we didn't need to match the Hilo C++ UX, which shows only eight images for each month in the month page, we would have used a custom data source in Hilo. (The [ListView](#) interface for the custom data source expected contiguous data to support groups, so we decided to switch to a [WinJS.Binding.List](#) implementation, which uses an in-memory array of data.) A custom data source would have provided data virtualization, enabling us to display images one page at a time and avoid any delay resulting from the use of **WinJS.Binding.List**.

For data sources other than the in-memory list, choices include a [StorageDataSource](#) object or a custom data source. A **StorageDataSource** includes built-in support for the Windows file system, but it doesn't support grouping of items, so we chose to implement a custom data source.

For grouped data, two data sources are required—one for the groups and one for the items.

A custom data source must implement the [IListDataAdapter](#) interface. Both data sources in the month page use data adapters that implemented this interface. When you implement this interface, the [ListView](#) infrastructure can make item requests at run time, and the data adapter responds by returning the requested information or by providing information stored in a cache. If images aren't loaded yet, the **ListView** displays placeholder images (which are specified in `month.css`).

For more info about the [ListView](#) and data sources, see [Using a ListView](#). For more info about custom data sources, and an example that uses a web-based data source, see [How to create a custom data source](#).

## Implementing a data adapter for a custom data source

A custom data source must implement the [IListDataAdapter](#) interface. In Hilo, the **DataAdapter** class implemented **IListDataAdapter**. Most of the challenges of implementing a custom data source had to do with basing the data source on the file system, and in particular, creating groups with a correspondence to image files only. Objects returned from the file system—[StorageFolder](#) and [StorageFile](#) objects—aren't in a format that's bindable to the [ListView](#), so you have to do some work to generate usable data. For the groups and items required in the month page, we needed to provide data in the special formats required by the **IListDataAdapter** interface.

**Important** The final version of Hilo did not implement a custom data source. For Hilo code that implements the custom data source, see [this version of Hilo](#).

For all pages in Hilo, we also wrap returned images in **Hilo.Picture** objects, which are implemented in `Picture.js`. For more info, see [Using the query builder pattern](#).

The key requirements to implement [IListDataAdapter](#) include adding implementations for [getCount](#), [itemsFromIndex](#), and [itemsFromKey](#).

### JavaScript: Hilo\Hilo\month\groups.js (in version of Hilo with a custom data source)

```
var DataAdapter = WinJS.Class.define(function (queryBuilder, folder,
getMonthYearFrom) {
    // . . .
    };

}, {

    itemsFromIndex: function (requestIndex, countBefore, countAfter) {
        // . . .
    },

    getCount: function () {
        // . . .
    },

    itemsFromKey: function (key, countBefore, countAfter) {
        // . . .
    },
});
```

The [getCount](#) method returns the total count of all items currently in the data source. This number must be either be correct or return **null**, or the [ListView](#) might not work correctly, and it might not indicate a reason for failure.

During our investigation period, we believed that implementing [itemsFromKey](#) was optional, but at run time the [ListView](#) infrastructure required an internal object derived from the **itemsFromKey** implementation, so we added a simple implementation of **itemsFromKey** to the group data adapter.

**Important** A similar data adapter for items is implemented in `members.js`. Because the data adapter for items was easier to implement and didn't require extra code to handle the grouping of data, we focus on the groups in this topic. The data adapter for items didn't require an implementation of [itemsFromKey](#) either.

At run time, when the [ListView](#) needs a set of items (groups, in this case) to populate the **ListView**, it calls [itemsFromIndex](#), passing it a request for a specific group index value along with suggested values for the number of groups to return before and after the requested index value.

#### JavaScript: `Hilo\Hilo\month\groups.js` (in version of Hilo with a custom data source)

```
itemsFromIndex: function (requestIndex, countBefore, countAfter) {
    var start = Math.max(0, requestIndex - countBefore),
        count = 1 + countBefore + countAfter,
        offset = requestIndex - start;

    // Check cache for group info. If not complete,
    // get new groups (call fetch).

    // . . .

    var buildResult = this.buildResult.bind(this, offset, start);

    return collectGroups.then(buildResult);
},
```

The return value for [itemsFromIndex](#) needs to be an object that implements the [IFetchResult](#) interface. In Hilo, we create an object with the following properties to fulfill the interface requirements:

- [items](#), which must be an array of items (groups) that implement the [IItems](#) interface.
- [offset](#), which stores the index for the requested group relative to the items array.
- [absoluteIndex](#), which stores the index of the requested group relative to the custom data source.
- [totalCount](#), which stores the total number of groups in the items array.

To get these values, we calculate the [offset](#) and [absoluteIndex](#) values by using **requestIndex**, **countBefore**, and **countAfter**, which are the requested values passed in by the [ListView](#). (See the preceding code.) We retrieve the [totalCount](#) from the data adapter's cached value.

#### JavaScript: `Hilo\Hilo\month\groups.js` (in version of Hilo with a custom data source)

```
buildResult: function (offset, absoluteIndex, items) {
    // Package the data in a format that the consuming
    // control (a list view) can process.
    var result = {
        items: items,
        offset: offset,
        absoluteIndex: absoluteIndex
```

```

    };

    if (this.totalCount) {
        result.totalCount = this.totalCount;
    }

    return WinJS.Promise.as(result);
},

```

To get the array of requested groups (the [items](#) property), we make file system queries by calling functions like [getFoldersAsync](#), which returns an array of [StorageFolder](#) objects. We call **getFoldersAsync** in the **fetch** function (called from [itemsFromIndex](#)), and then use the JavaScript array mapping function to convert the **StorageFolder** objects to usable data.

#### JavaScript: Hilo\Hilo\month\groups.js (in version of Hilo with a custom data source)

```

return this.query
    .getFoldersAsync(start, count)
    .then(function (folders) {
        return WinJS.Promise.join(folders.map(convert));
    })

```

In **convert**, the callback function for array mapping, we make additional file system queries by using the passed in [StorageFolder](#) object, and then pass the query results to **buildMonthGroupResult** in a joined promise.

#### JavaScript: Hilo\Hilo\month\groups.js (in version of Hilo with a custom data source)

```

var query = this.queryBuilder.build(folder);

var getCount = query.fileQuery.getItemCountAsync();

var getFirstFileItemDate = query.fileQuery
    .getFilesAsync(0, 1)
    .then(retrieveFirstItemDate);

return WinJS.Promise
    .join([getCount, getFirstFileItemDate])
    .then(this.builldMonthGroupResult.bind(this));

```

In **buildMonthGroupResult**, we create a group object that implements the [IItems](#) interface, and that can be set as the [items](#) property value of the [IFetchResult](#). To implement the **IItems** interface, we use the query results to set group properties like **groupKey**, **data**, **index**, and **key**. For example, the query results from `getFilesAsync(0, 1)` are used to set group properties based on the item date of the first file in the group.

**JavaScript: Hilo\Hilo\month\groups.js (in version of Hilo with a custom data source)**

```

buildMonthGroupResult: function (values) {

    var count = values[0],
        firstItemDate = values[1];

    var monthYear = this.getMonthYearFrom(firstItemDate);

    var result = {
        key: monthYear,
        firstItemIndexHint: null, // We need to set this later.
        data: {
            title: count ? monthYear : 'invalid files',
            count: count
        },
        groupKey: count ? monthYear.split(' ')[1] : 'invalid files'
    };

    return result;
}

```

**Important** The **key** property must be unique or the [ListView](#) may appear to fail without providing an error message.

The data properties, **title** and **count**, are declaratively bound to the UI in the Windows Library for JavaScript template for the group header defined in month.html.

**Important** This function also creates properties like **firstItemIndexHint**, which is used to specify the first index value in each group. Its value is set later.

For more info on the use of promises in Hilo, see see [Async programming patterns and tips](#).

### Binding the ListView to a custom data source

When the month page is loaded, the [ready](#) function for the [page control](#) is called, in the same way that it's called for other Hilo pages. In the **ready** function, two [IListAdapter](#) objects are created:

- A data adapter object that represents the groups, which is instantiated by calling **Hilo.month.Groups**.
- A data adapter that represents the items, which is instantiated by calling **Hilo.month.Items**.

In both cases, we pass the query builder to the [IListAdapter](#) along with several other parameters.

**Important** The final version of Hilo didn't implement a custom data source. For Hilo code that implements the custom data source, see [this version of Hilo](#).

**JavaScript: Hilo\Hilo\month\month.js (in version of Hilo with a custom data source)**

```

ready: function (element, options) {

    // I18N resource binding for this page.
    WinJS.Resources.processAll();

    this.queryBuilder = new Hilo.ImageQueryBuilder();

    // First, set up the various data adapters needed.
    this.monthGroups = new Hilo.month.Groups(this.queryBuilder,
this.targetFolder,
    this.getMonthYearFrom);
    this.monthGroupMembers = new Hilo.month.Members(this.queryBuilder,
    this.targetFolder, this.getMonthYearFrom);

    var yearGroupMembers = this._setYearGroupDataAdapter(this.monthGroups);
    var yearGroups = yearGroupMembers.groups;

    // Then provide the adapters to the list view controls.
    this._setupMonthGroupListView(this.monthGroups, this.monthGroupMembers);
    this._setupYearGroupListView(yearGroups, yearGroupMembers);
},

```

Any data source that binds to a [ListView](#), including a [WinJS.Binding.List](#), must implement [IListDataSource](#). To bind to a **ListView** (or a **FlipView**) by using a custom data source, the data source you create must derive from [VirtualizedDataSource](#), which implements **IListDataSource**. The **VirtualizedDataSource** is the base class for an [IListDataAdapter](#).

The **DataAdapter** class implements [IListDataAdapter](#). In this code, called from the [ready](#) function, we create the group data adapter—an object derived from [VirtualizedDataSource](#)—and pass a new **DataAdapter** object to the constructor for the newly derived class. To instantiate the new object correctly, we also need to pass the **DataAdapter** object to [\\_baseDataSourceConstructor](#), the base class constructor for a **VirtualizedDataSource**.

**JavaScript: Hilo\Hilo\month\groups.js (in version of Hilo with a custom data source)**

```

var Groups = WinJS.Class.derive(WinJS.UI.VirtualizedDataSource,
    function (queryBuilder, folder, getMonthYearFrom) {
        this.adapter = new DataAdapter(queryBuilder, folder, getMonthYearFrom);
        this._baseDataSourceConstructor(this.adapter);
    }, {
    getFirstIndexForGroup: function (groupKey) {
        var groupIndex = this.adapter.cache.byKey[groupKey];
        var group = this.adapter.cache.byIndex[groupIndex];
        return group.firstItemIndexHint;
    }
}

```

```
});
```

After the data adapters for groups and members are created, data is returned asynchronously from the file system upon request by the [ListView](#). For more info, see [Implementing a data adapter for a custom data source](#).

In the preceding code, the **ready** function calls **\_setupMonthGroupListView** and **\_setupYearGroupListView** to bind the data sources to the [ListView](#). In the following code, the **monthgroupListView** control declared in `month.html` is retrieved, and then the [groupDataSource](#) and [itemDataSource](#) properties are used to bind the groups and members from the Hilo data adapters to the **ListView**.

#### JavaScript: Hilo\Hilo\month\month.js (in version of Hilo with a custom data source)

```
_setupMonthGroupListView: function (groups, members) {  
    var listview = document.querySelector("#monthgroup").winControl;  
    listview.groupDataSource = groups;  
    listview.itemDataSource = members;  
    listview.addEventListener("iteminvoked", this._imageInvoked.bind(this));  
},
```

## Using touch in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Understand how each user interaction works on touch-enabled devices.
- When possible, use the standard touch gestures and JavaScript controls that Windows 8 provides.
- To implement Semantic Zoom, use the [ListView](#) control or develop your own control that implements [IZoomableView](#).

Hilo provides examples of tap, slide, swipe, pinch and stretch, and turn gestures. Here we explain how we applied the Windows 8 touch language to Hilo to provide a great experience on any device.

### You will learn

- How the Windows 8 touch language was used in Hilo.

Part of providing a great experience is ensuring that an app is accessible and intuitive to use on a traditional desktop computer and on a small tablet. For Hilo, we put touch at the forefront of our UX planning because it adds an important experience by providing a more engaging interaction between the user and the app.

As described in [Designing the UX \(Hilo C++\)](#), touch is more than simply an alternative to using a mouse. We wanted to make touch an integrated part of the app because touch can add a personal connection between the user and the app. Touch is also a natural way to enable users to crop and rotate their photos. In addition, we use Semantic Zoom to highlight how levels of related complexity can easily be navigated. When the user uses the pinch gesture on the month page, the app switches to a calendar-based view. Users can then browse photos more quickly.

Hilo uses the Windows 8 touch language. We use the standard touch gestures that Windows provides for these reasons:

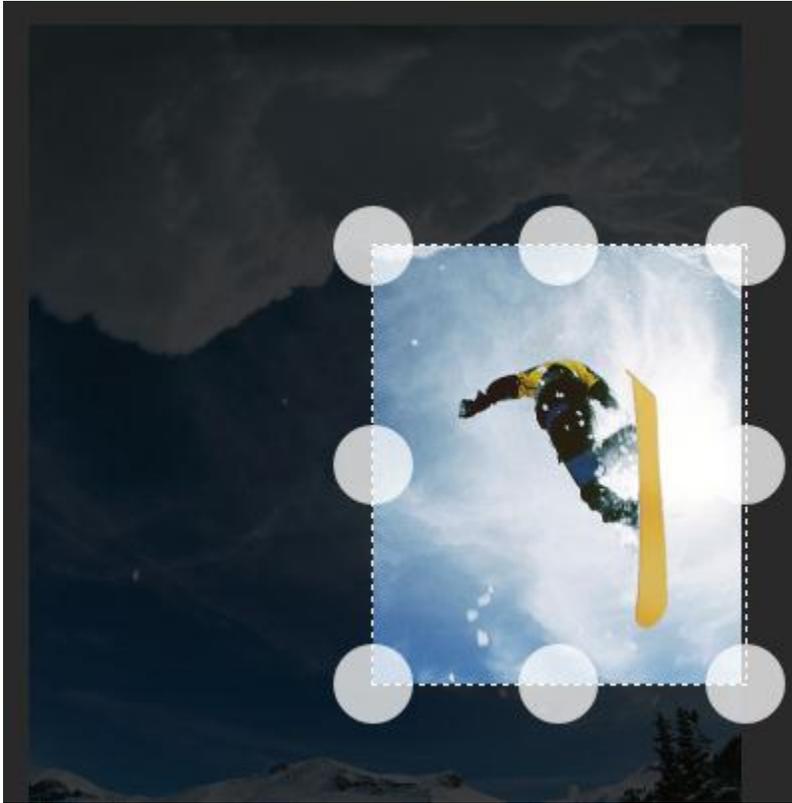
- The Windows Runtime provides an easy way to work with them.
- We don't want to confuse users by creating custom interactions.
- We want users to use the gestures that they already know to explore the app, and not need to learn new gestures.

We also wanted Hilo to be intuitive for users who use a mouse or similar pointing device. The built-in controls work as well with a mouse or other pointing device as they do with touch. So when you design for touch, you also get mouse and pen functionality. For example, you can use the left mouse button to invoke commands. In addition, mouse and keyboard equivalents are provided for many commands. For example, you can use the right mouse button to activate the app bar, and holding the Ctrl key down while scrolling the mouse wheel controls Semantic Zoom interaction.

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

## Tap for primary action

Tapping an element invokes its primary action. For example, in crop mode, you tap the crop area to crop the image.



To implement crop, we used a [canvas](#). The **click** event of the **canvas** initiates the crop operation, with the event firing for both pointer devices and touch. In the **ready** function of the page control, `crop.js`, the **setupImageView** function is called. This function creates an instance of the **ImageView** class. Here's the code.

### JavaScript: Hilo\Hilo\crop\setupImageView.js

```
this.imageView = new Hilo.Crop.ImageView(image, this.cropSelection, canvasEl, imageEl);
```

We added a listener in the **ImageView** constructor. The listener specifies the function that executes when a click is received on the canvas. [WinJS.Class.mixin](#) is used to define the **ImageView** class. The [eventMixin](#) mixin is used.

### JavaScript: Hilo\Hilo\crop\imageView.js

```
canvasEl.addEventListener("click", this.click.bind(this));
```

The **click** function simply raises the preview event.

#### JavaScript: Hilo\Hilo\crop\imageView.js

```
this.dispatchEvent("preview", {});
```

We added a listener in the **start** function in the **CropPresenter** class. This listener specifies the function that executes when the preview event is received from the **ImageView** class.

#### JavaScript: Hilo\Hilo\crop\cropPresenter.js

```
this.imageView.addEventListener("preview", this.cropImage.bind(this));
```

The **cropImage** function calls the **cropImage** function of the **ImageView** class. This function obtains the canvas-based crop selection, and then the selected area of the original image is calculated. We perform this calculation by scaling the canvas-based selection to the original image dimensions. The on-screen image is then cropped, to show what the crop result will look like when the file is saved. To support multiple cropping operations, we store an offset that represents the starting location of the crop on the original image, rather than relative to the canvas size. Here's the code.

#### JavaScript: Hilo\Hilo\crop\imageView.js

```
cropImage: function () {
    var selectionRectScaledToImage = this.getScaledSelectionRectangle();
    // Reset image scale so that it reflects the difference between
    // the current canvas size (the crop selection) and the original
    // image size, then re-draw everything at that new scale.
    this.imageToScreenScale =
this.calculateScaleToScreen(selectionRectScaledToImage);
    this.drawImageSelectionToScale(selectionRectScaledToImage,
this.imageToScreenScale);

    // Remember the starting location of the crop on the original image
    // and not relative to the canvas size, so that cropping multiple times
    // will correctly crop to what has been visually selected.
    this.image.updateOffset({ x: selectionRectScaledToImage.startX, y:
selectionRectScaledToImage.startY });

    return this.canvasEl.toDataURL();
},
```

## Slide to pan

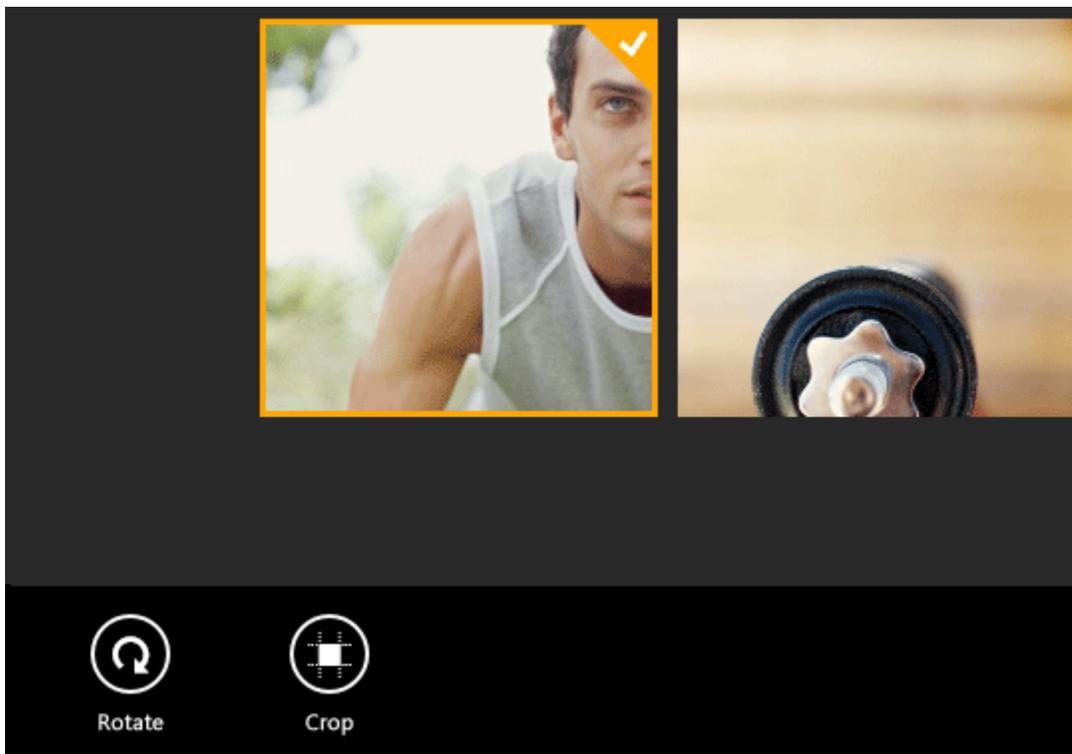
Hilo uses the slide gesture to pan among images in a collection. For example, when you view an image, you can also quickly pan to any image in the current collection by using the [FlipView](#), or by using the filmstrip that appears when you display the app bar. We used the [ListView](#) control to implement the filmstrip.



A benefit of using the [ListView](#) object is that it has touch capabilities built in, removing the need for additional code.

### Swipe to select, command, and move

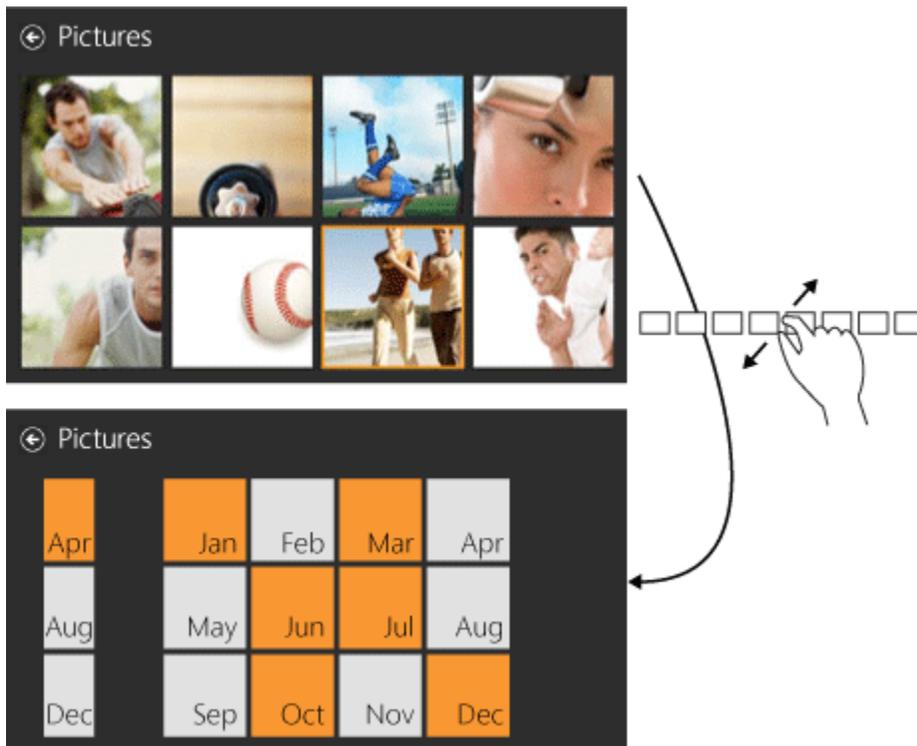
With the swipe gesture, you slide your finger perpendicular to the panning direction to select objects. In Hilo, when a page contains multiple images, you can use this gesture to select one image. When you display the app bar, the commands that appear apply to the selected image. [ListView](#) and other controls provide built-in support for selection. You can use the [selection](#) property to retrieve the selected item.



## Pinch and stretch to zoom

Pinch and stretch gestures are not just for magnification, or performing "optical" zoom. Hilo uses *Semantic Zoom* to help users navigate between large sets of pictures. Semantic Zoom enables you to switch between two different views of the same content. You typically have a main view of your content and a second view that allows users to quickly navigate through it. (Read [Quickstart: adding a SemanticZoom](#) for more info about Semantic Zoom.)

On the month page, when you zoom out, the view changes to a calendar-based view. The calendar view highlights those months that contain photos. You can then zoom back in to the image-based month view. The following diagram shows the two views that the Semantic Zoom switches between.



To implement Semantic Zoom, we used the [SemanticZoom](#) object. We then provide controls for the zoomed-in and zoomed-out views. The controls must implement the [IZoomableView](#) interface. Hilo uses the [ListView](#) control to implement Semantic Zoom.

**Note** The Windows Library for JavaScript provides one control that implements [IZoomableView](#)—the [ListView](#) control. You can also create your own custom controls that implement **IZoomableView** or augment an existing control to support **IZoomableView** so that you can use it with [SemanticZoom](#).

For the zoomed-in view, we display a [ListView](#) that binds to photo thumbnails that are grouped by month. The **ListView** also shows a title (the month and year) for each group. For more info about the zoomed-in view implementation, see [Working with data sources](#).

For the zoomed-out view, we display a [ListView](#) that binds to photo thumbnails that are grouped by year. Here's the HTML for the zoomed-out view.

#### HTML: Hilo\Hilo\month\month.html

```
<div id="yeargroup" data-win-control="Hilo.month.YearList">
</div>
```

The DIV element specifies a [data-win-control](#) attribute that has a value of **Hilo.month.YearList**. This sets the DIV element as the host for the **YearList** control. The **YearList** class, which implements the control, creates and initializes a new [ListView](#) object. Here's the code.

#### JavaScript: Hilo\Hilo\month\yearList.js

```
var listViewEl = document.createElement("div");
element.appendChild(listViewEl);

var listView = new WinJS.UI.ListView(listViewEl, {
    layout: { type: WinJS.UI.GridLayout },
    selectionMode: "none",
    tapBehavior: "none"
});

this.listView = listView;
listView.layout.maxRows = 3;
```

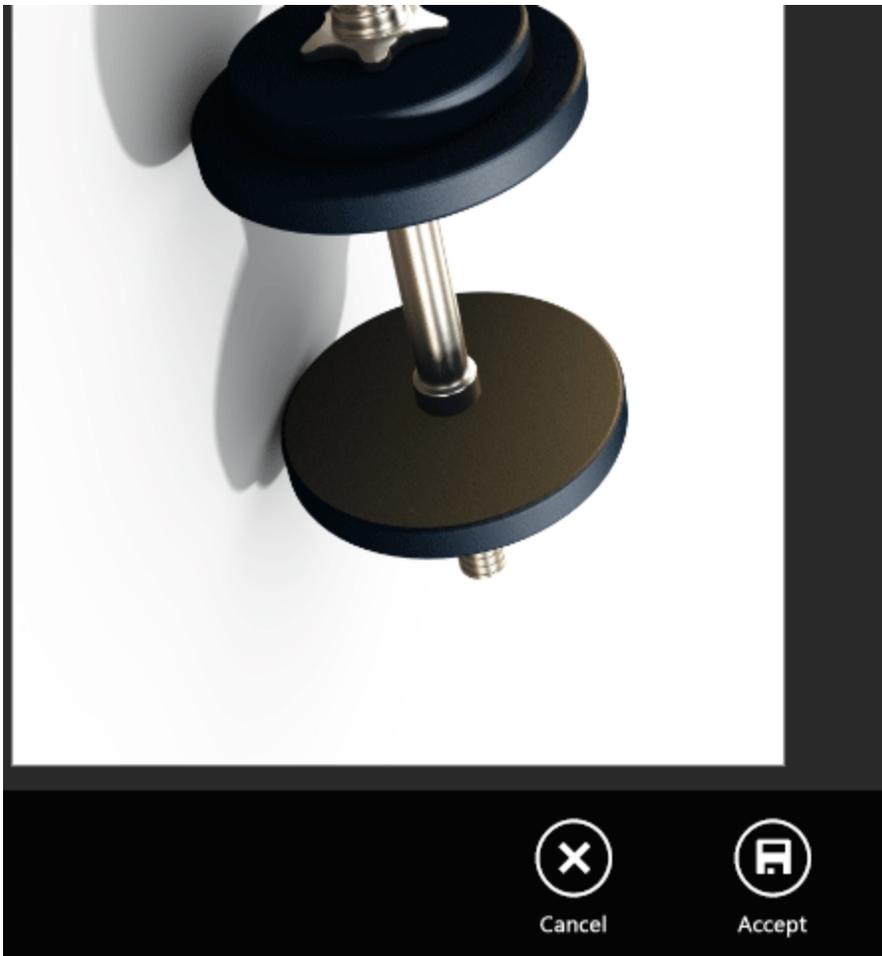
The **YearList** class also creates a template to format and display the multiple instances of data in the year group. We create this template by setting the [itemTemplate](#) property of the [ListView](#) to a function that generates DOM elements for each item in the year group.

For more info about setting up a data source, see [Working with data sources](#).

For more info about Semantic Zoom, see [Quickstart: adding a SemanticZoom](#) and [Guidelines for Semantic Zoom](#).

## Turn to rotate

In the rotate image view, you can use two fingers to rotate the image. When you release your fingers, the image snaps to the nearest 90-degree rotation.



The **ready** function of the page control class for the rotate page, `rotate.js`, creates a new instance of the **TouchProvider** class. Here's the code.

#### JavaScript: `Hilo\Hilo\rotate\rotate.js`

```
var touchProvider = new Hilo.Rotate.TouchProvider(element);
```

The **TouchProvider** class provides logic to show the photo rotating as the rotate gesture is being applied. The constructor creates an instance of the [GestureRecognizer](#) class, which listens for and handles all pointer and gesture events. The gesture recognizer is configured to only process the rotate gesture. Event listeners are then registered for the [MSPointerDown](#), [MSPointerMove](#), and [MSPointerUp](#) DOM pointer events, along with the [manipulationupdated](#) and [manipulationcompleted](#) events. Functions are registered for these DOM pointer events, which simply pass the received [PointerPoint](#) to the gesture recognizer.

#### JavaScript: `Hilo\Hilo\rotate\TouchProvider.js`

```
function TouchProviderConstructor(inputElement) {
    var recognizer = new Windows.UI.Input.GestureRecognizer();
```

```

recognizer.gestureSettings = Windows.UI.Input.GestureSettings.manipulationRotate;

this._manipulationUpdated = this._manipulationUpdated.bind(this);
this._manipulationCompleted = this._manipulationCompleted.bind(this);

inputElement.addEventListener("MSPointerDown", function (evt) {
    var pp = evt.currentPoint;
    if (pp.pointerDevice.pointerDeviceType === pointerDeviceType.touch) {
        recognizer.processDownEvent(pp);
    }
}, false);

inputElement.addEventListener("MSPointerMove", function (evt) {
    var pps = evt.intermediatePoints;
    if (pps[0] && pps[0].pointerDevice.pointerDeviceType ===
pointerDeviceType.touch) {
        recognizer.processMoveEvents(pps);
    }
}, false);

inputElement.addEventListener("MSPointerUp", function (evt) {
    var pp = evt.currentPoint;
    if (pp.pointerDevice.pointerDeviceType === pointerDeviceType.touch) {
        recognizer.processUpEvent(pp);
    }
}, false);

recognizer.addEventListener("manipulationupdated", this._manipulationUpdated);
recognizer.addEventListener("manipulationcompleted",
this._manipulationCompleted);

this.displayRotation = 0;
},

```

The [manipulationupdated](#) and [manipulationcompleted](#) events are used to handle rotation events. The **\_manipulationUpdated** method updates the current rotation angle and the **\_manipulationCompleted** method snaps the rotation to the nearest 90-degree value. Here's the code.

#### JavaScript: Hilo\Hilo\rotate\TouchProvider.js

```

_manipulationUpdated: function (args) {
    this.setRotation(args.cumulative.rotation);
},

_manipulationCompleted: function (args) {
    var degrees = args.cumulative.rotation;
    var adjustment = Math.round(degrees / 90) * 90;
    this.animateRotation(adjustment);
}

```

**Caution** These event handlers are defined for the entire page. If your page contains more than one item, you need additional logic to determine which object to manipulate.

The page control class for the rotate page, `rotate.js`, passes the instance of the **TouchProvider** class to the **RotatePresenter** class, where the **setRotation** and **animateRotation** methods of the **TouchProvider** class are set to internal methods in the **RotatePresenter** class.

#### JavaScript: Hilo\Hilo\rotate\rotatePresenter.js

```
touchProvider.setRotation = this._rotateImageWithoutTransition;
touchProvider.animateRotation = this._rotateImage;
```

Calls in the **TouchProvider** class to the **setRotation** and **animateRotation** methods call the **\_rotateImageWithoutTransition** and **\_rotateImage** internal methods of the **RotatePresenter** class respectively, in order to display the image rotated on-screen by setting the CSS rotation of the image element.

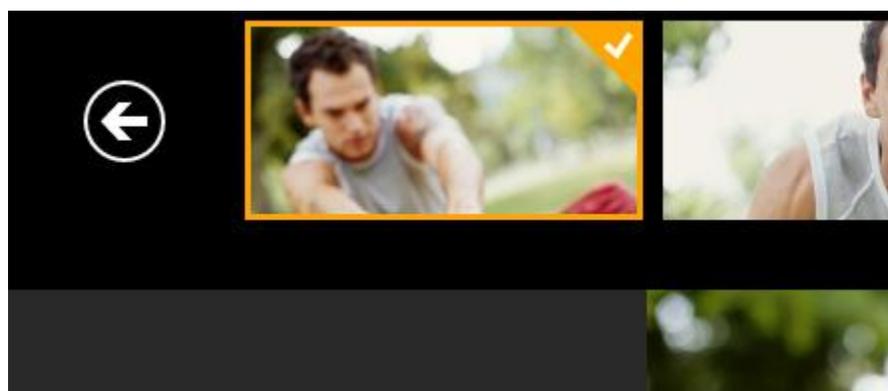
For more info, see [Quickstart: Pointers](#), [Quickstart: DOM gestures and manipulations](#), [Quickstart: Static gestures](#), and [Guidelines for rotation](#).

## Swipe from edge for app commands

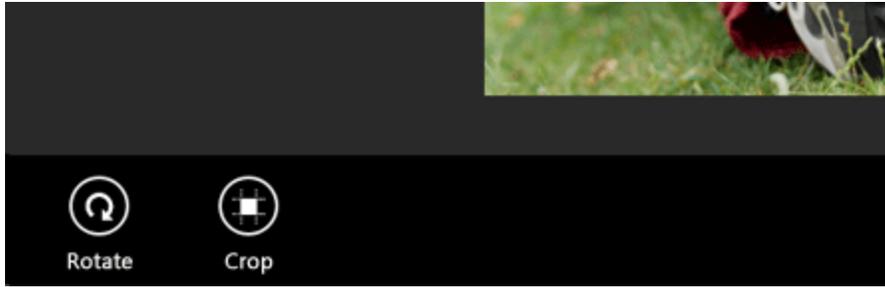
When there are relevant commands to display, Hilo displays the app bar when the user swipes from the bottom or top edge of the screen.

Every page can define a top app bar, a bottom app bar, or both. For instance, Hilo displays both when you view an image and activate the app bar.

Here's the top app bar.



Here's the bottom app bar for the same photo.



The location of the app bar on a page is controlled by its [placement](#) property.

Hilo implements an image navigation control that provides a re-usable implementation of the bottom app bar that can be used to navigate to the rotate and crop pages. There are two parts to this control:

- The `hiloAppBar.html` file, which can be included in any page that needs the navigation app bar.
- The `hiloAppBarPresenter.js` file, which is the controller that's used to provide the functionality of the app bar.

This control is used on the hub, month, and detail pages. To add the control, we add a reference to it in the page's markup and in the page control file. Here's the code for adding it to a page's markup.

#### HTML: Hilo\Hilo\detail\detail.html

```
<section id="image-nav" data-win-control="WinJS.UI.HtmlControl" data-win-
options="{uri: '/Hilo/controls/HiloAppBar/hiloAppBar.html'}"></section>
```

In the page control file, the **HiloAppBarPresenter** requires a reference to the HTML element that's used to place the control on the screen. Here's the code.

#### JavaScript: Hilo\Hilo\detail\detail.js

```
var hiloAppBarEl = document.querySelector("#appbar");
var hiloAppBar = new Hilo.Controls.HiloAppBar.HiloAppBarPresenter(hiloAppBarEl,
WinJS.Navigation, query);
```

**Caution** In most cases, you shouldn't programmatically display an app bar if there are no relevant commands to show. For example, the main hub page shows the app bar whenever a photo is selected by calling the **HiloAppBarPresenter.setNavigationOptions** method. Calling the **HiloAppBarPresenter.clearNavigationOptions** method will hide the app bar.

For more info about app bars, see [Adding app bars](#).

## Swipe from edge for system commands

Because touch interaction can be less precise than other pointing devices, such as a mouse, we maintained a sufficient distance between the app controls and the edges of the screen. Because we

maintained this distance, the user can easily swipe from the edge of the screen to reveal the app bars and charms, or to display previously used apps. For more info see [Laying out an app page](#).

## Handling suspend, resume, and activation in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Save application data when the app is being suspended.
- Release exclusive resources and file handles when the app is being suspended.
- Use the saved application data to restore the app when needed.

Hilo provides examples of how to suspend and activate a Windows Store app that uses JavaScript. You can use these examples to write an app that fully manages its execution life cycle. Suspension can happen at any time, and when it does you need to save your app's data so that the app can resume correctly.

### You will learn

- How Windows determines an app's execution state.
- How the app's activation history affects its behavior.
- How to implement support for suspend, resume, and activation by using JavaScript.

### Tips for implementing suspend/resume

You should design your app to suspend correctly when the user moves away from it, or when there is a low power state. You should also design the app to resume correctly when the user moves back to it, or when Windows leaves the low power state. Here are some things to remember:

- Save application data when the app is being suspended.
- Resume your app in the state that the user left it in.
- When navigating away from a page, save the page state to minimize the time required to suspend the app.
- Allow views and presenters to save and restore state that's relevant to each. For example, if the user has typed text into a text box but hasn't yet tabbed out of the text box, you might want to save the partially entered text as view state. In Hilo, only presenters need to save state.
- Release exclusive resources when the app is being suspended.
- When the app resumes, update the UI if the content has changed.
- When the app resumes after being terminated, use the saved application data to restore the app state.

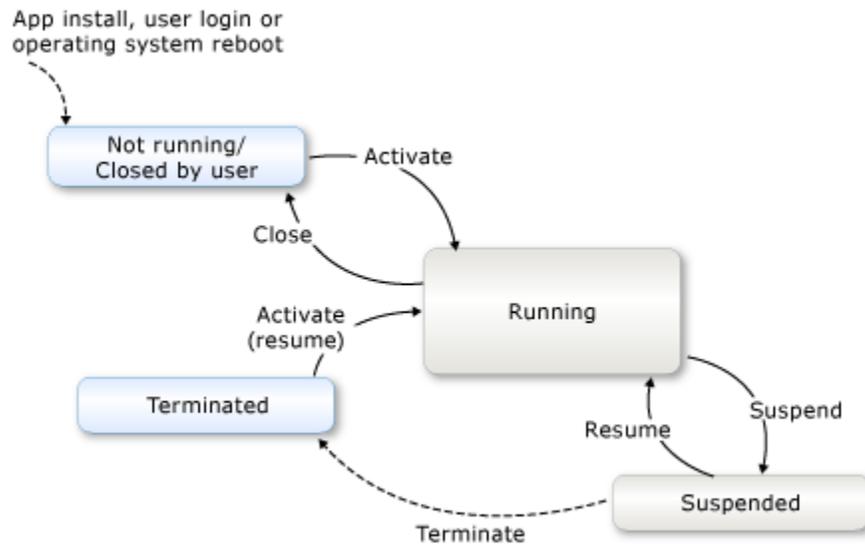
For more info, see [Guidelines for app suspend and resume \(Windows Store apps\)](#).

## Understanding possible execution states

Which events occur when you activate an app depends on the app's execution history. There are five cases to consider. The cases correspond to the values of the [Windows.ActivationModel.Activation.ApplicationExecutionState](#) enumeration.

- **notRunning**
- **terminated**
- **closedByUser**
- **suspended**
- **running**

Here's a diagram that shows how Windows determines an app's execution state. In the diagram, the blue rectangles indicate that the app isn't loaded into system memory. The white rectangles indicate that the app is in memory. The dashed arcs are changes that occur without any notification to the running app. The solid arcs are actions that include app notification.



The execution state depends on the app's history. For example, when the user starts the app for the first time after installing it or after restarting Windows, the previous execution state is **notRunning**, and the state after activation is **running**. When activation occurs, the activation event arguments include a [previousExecutionState](#) property that indicates the state the app was in before it was activated.

If the user switches to a different app or if the system enters a low power mode of operation, Windows notifies the app that it's being suspended. At this time, you must save the navigation state and all user data that represents the user's session. You should also free exclusive system resources, like open files and network connections.

Windows allows 5 seconds for an app to handle the **suspending** event. If the **suspending** event handler doesn't complete within that amount of time, Windows behaves as though the app has stopped responding and terminates it.

After the app responds to the [oncheckpoint](#) (or [suspending](#)) event, its state is **suspended**. If the user switches back to the app, Windows resumes it and allows it to run again.

Windows might terminate an app (without notification) after it has been suspended. For example, if the device is low on resources it might reclaim resources that are held by suspended apps. If the user launches your app after Windows has terminated it, the app's previous execution state at the time of activation is **terminated**.

You can use the previous execution state to determine whether your app needs to restore the data that it saved when it was last suspended, or whether you must load your app's default data. In general, if the app stops responding or the user closes it, restarting the app should take the user to the app's default initial navigation state. When an app is activated after being terminated, it should load the application data that it saved during suspension so that the app appears as it did when it was suspended.

**Note** When an app is suspended but hasn't yet been terminated, you can resume the app without restoring any state. The app will still be in memory. In this situation, you might need to reacquire resources and update the UI to reflect any changes to the environment that occurred while the app was suspended. For example, Hilo updates its app tile when resuming from the suspended state.

For a description of the suspend/resume process, see [Application lifecycle \(Windows Store apps\)](#). For more info about each of the possible previous execution states, see [ApplicationExecutionState](#) enumeration. You might also want to consult [Guidelines for app suspend and resume \(Windows Store apps\)](#) for info about the recommended user experience for suspend and resume.

## Code walkthrough of suspend

When Hilo starts, the bootstrapper registers a handler for the [oncheckpoint](#) (or [suspending](#)) event. Windows invokes this event handler before it suspends the app. Here's the code.

### JavaScript: Hilo\default.js

```
app.addEventListener("checkpoint", function (args) {
    // The app is about to be suspended, so we save the current
    // navigation history.
    app.sessionState.history = nav.history;
}, false);
```

Hilo uses the event handler to synchronously save the app's navigation history to the [sessionState](#) object. The **sessionState** object is used for storing data that can be used to restore the app's state after it resumes from suspension. Any data stored in the **sessionState** object is automatically serialized to disk when the app is suspended.

**Note** Because Windows allows 5 seconds for the app to handle the **suspending** event, any asynchronous events handled by the [oncheckpoint](#) handler must complete within this time frame.

When the app is suspended when the detail, rotate, or crop page is active, the **QueryObject** for that page is serialized so that when the app resumes, the correct photo can be loaded. Here's the code.

**JavaScript: Hilo\Hilo\imageQueryBuilder.js**

```
toJSON: function () {
    return this;
},
```

The **toJSON** method is called automatically when the **QueryObject** is serialized. Though the implementation doesn't add anything beyond the in-built logic, it's included to demonstrate where the serialization could be customized if required.

## Code walkthrough of resume

When an app resumes from the **suspended** state, it enters the **running** state and continues from where it was when it was suspended. No application data is lost, because it was stored in memory.

It's possible that an app being resumed has been suspended for hours or even days. So, if the app has content or network connections that might need to be updated, these should be refreshed when the app resumes. When an app is suspended, it doesn't receive network or file event items that it registered to receive. In this situation, your app should test the network or file status when it resumes.

If an app registered an event handler for the [resuming](#) event, it is called when the app resumes from the **suspended** state. You can refresh your content by using this event handler. Hilo subscribes to the **resuming** event to refresh the thumbnails on the app's tile. Here's the code.

**JavaScript: Hilo\default.js**

```
Windows.UI.WebUI.WebUIApplication.addEventListener("resuming", function (args) {
    var tileUpdater = new Hilo.Tiles.TileUpdater();
    tileUpdater.update();
}, false);
```

## Code walkthrough of activation

When an app is started, the bootstrapper registers a handler for the [onactivated](#) event. A user can activate an app through a variety of contracts and extensions, but in Hilo we only needed to handle normal startup. So the **onactivated** event receives an object of type [WebUILaunchActivatedEventArgs](#). This object contains an [ApplicationExecutionState](#) enumeration that indicates the app's previous execution state. Here's the code.

**JavaScript: Hilo\default.js**

```
app.addEventListener("activated", function (args) {

    var currentState = args.detail;

    if (currentState.kind === activation.ActivationKind.launch) {

        if (currentState.previousExecutionState !==
```

```

activation.ApplicationExecutionState.terminated) {

    // When the app is started, we want to update its tile
    // on the start screen. Since this API is not accessible
    // inside of Blend, we only invoke it when we are not in
    // design mode.
    if (!Windows.ApplicationModel.DesignMode.designModeEnabled) {
        var tileUpdater = new Hilo.Tiles.TileUpdater();
        tileUpdater.update();
    }

    // Begin listening for changes in the `picturesLibrary`.
    // If files are added, deleted, or modified, update the
    // current screen accordingly.
    Hilo.contentChangeListener
        .listen(Windows.Storage.KnownFolders.picturesLibrary);

} else {
    // This app has been reactivated from suspension.
    // Restore app state here.
}

// If any history is found in the `sessionState`, we need to
// restore it.
if (app.sessionState.history) {
    nav.history = app.sessionState.history;
}

// After we process the UI (search the DOM for data-win-control),
// we'll navigate to the current page. These are async operations
// and they will return a promise.
var processAndNavigate = WinJS.UI
    .processAll()
    .then(function () {

        if (nav.location) {
            nav.history.current.initialPlaceholder = true;
            return nav.navigate(nav.location, nav.state);
        } else {
            return nav.navigate(Hilo.navigators.home);
        }
    });

args.setPromise(processAndNavigate);
}
}, false);

```

The code checks whether the application is being started, and if it is, checks its [previousExecutionState](#) to determine whether the previous state was **terminated**. If the app wasn't terminated, the handler treats it as though it's being launched for the first time, so the tiles on the Start screen are updated. A handler is then registered for the [contentschanged](#) event, so that if the user's Pictures library changes, the app is notified and responds accordingly. The handler is kept registered while the app is suspended, although the app doesn't receive **contentschanged** events while it's suspended. When the app resumes, it receives a single event that aggregates all the file system changes that occurred while the app was suspended.

Then, regardless of whether the previous state was **terminated**, any navigation history contained in the [sessionState](#) object is restored. The [processAll](#) function then processes the page and activates any Windows Library for JavaScript controls that are declared in the page markup. Finally, if navigation history exists, the app navigates to the page that was displayed when the app was suspended. Otherwise, it navigates to the hub page.

If the previous state of the app was **terminated**, and when the app was suspended the detail, rotate, or crop page was active, the **QueryObject** for the page is deserialized so that the photo that was displayed when the app was suspended is reloaded.

#### JavaScript: Hilo\Hilo\imageQueryBuilder.js

```
deserialize: function (serializedQuery) {
    // Even though we pass in the entire query object, we really only care
    // about the settings. They allow us to reconstruct the correct query.
    var settings = serializedQuery.settings;

    if (typeof settings.monthAndYear === "string") {
        settings.monthAndYear = new Date(settings.monthAndYear);
    }

    return new QueryObject(settings);
}
```

The **deserialize** function takes a query object and uses the query object settings to reconstruct the query. The **checkOptions** function in `pageControlHelper.js`, which is invoked when each page is loaded, is responsible for calling the **deserialize** function. Here's the code.

#### JavaScript: Hilo\Hilo\controls\pageControlHelper.js

```
function checkOptions(options, deserialize) {
    if (!options) { return; }

    deserialize = deserialize || Hilo.ImageQueryBuilder.deserialize;

    if (options.query) {
        var original = options.query;
        var query = deserialize(original);
    }
}
```

```

    // Copy any properties that were not produced
    // from the deserialization.
    Object.keys(original).forEach(function (property) {
        if (!query[property]) {
            query[property] = original[property];
        }
    });

    options.query = query;
}
}

```

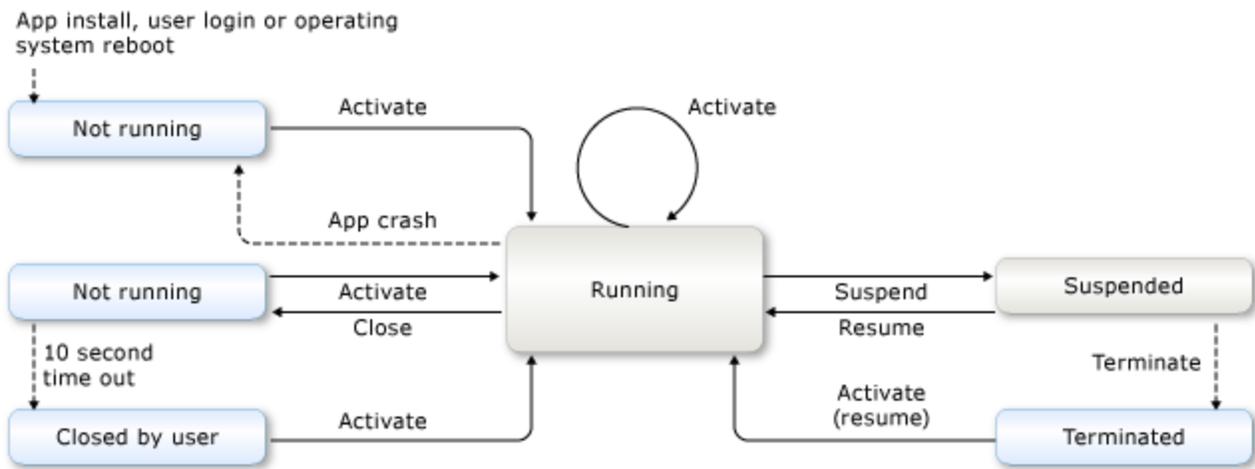
The **checkOptions** function is invoked from `page.js`, with only the **options** parameter being passed. So, provided that the **options** variable contains data, the **deserialize** variable will be set to the **deserialize** function in the **ImageQueryBuilder** class. Then, provided that the **options** variable contains a JSON serialized query, and provided that the query isn't being executed, the **deserialize** function will be invoked to reconstruct the query object. Finally, any properties that weren't deserialized are copied to the query object.

## Other ways to close the app

Apps don't contain UI for closing the app, but users can choose to close an app by pressing `Alt+F4`, dragging the app to the bottom of the screen, or selecting the **Close** context menu for the app when it's in the sidebar. When an app is closed by any of these methods, it enters the **notRunning** state for approximately 10 seconds and then transitions to the **closedByUser** state.

Apps shouldn't close themselves programmatically as part of normal execution. When you close an app programmatically, Windows treats this as an app crash. The app enters the **notRunning** state and remains there until the user activates it again.

Here's a diagram that shows how Windows determines an app's execution state. Windows takes app crashes and user close actions into account, in addition to the suspend or resume state. In the diagram, the blue rectangles indicate that the app isn't loaded into system memory. The white rectangles indicate that the app is in memory. The dashed lines are changes that occur without any modification to the running app. The solid lines are actions that include app notification.



## Improving performance in Hilo (Windows Store apps using JavaScript and HTML)

### Summary

- Use bytecode caching to improve the load time of your app.
- Use asynchronous APIs that execute in the background and inform the app when they've completed.
- Use performance tools to measure, evaluate, and target performance-related issues in your app.

The Hilo team spent time learning what works and what doesn't work to create a fast and fluid Windows Store app. Here are some tips and coding guidelines for creating a well-performing, responsive app.

### You will learn

- Tips that help create a fast and fluid app.
- The differences between performance and perceived performance.
- Recommended strategies for profiling an app.

### Performance tips

A *fast and fluid* app responds to user actions quickly, and with matching energy. The Hilo team spent time learning what works and what doesn't work to create a fast and fluid app. Here are some things to remember:

- Limit the start time
- Emphasize responsiveness
- Use thumbnails for quick rendering
- Retrieve thumbnails when accessing items
- Release media and stream resources when they're no longer needed
- Optimize ListView performance
- Keep DOM interactions to a minimum
- Optimize property access
- Use independent animations
- Manage layout efficiently
- Store state efficiently
- Keep your app's memory usage low when it's suspended
- Minimize the amount of resources that your app uses

### Limit the start time

It's important to limit how much time the user spends waiting while your app starts. You can dramatically improve the loading time of an app by packaging its contents locally. In turn, this leads to other performance benefits, such as bytecode caching. *Bytecode caching* is a technique in which the system creates bytecode for each JavaScript file once, rather than re-creating the bytecode each time it starts the app. This technique improves load time by approximately 30 percent in a larger app. To

benefit from bytecode caching, you must ensure that all JavaScript files are UTF8 encoded with a byte-order mark, and ensure that all JavaScript files are statically referenced in the root of your HTML start page. For more info, see [Reducing your app's loading time](#).

### Emphasize responsiveness

Don't block your app with synchronous APIs, because if you do the app can't respond to new events while the API is executing. Instead, use asynchronous APIs that execute in the background and inform the app when they've completed by raising an event. You should also break intensive processing operations into a series of smaller operations, allowing the app to respond to user input in between these operations. For more info, see [Executing code](#).

### Use thumbnails for quick rendering

The file system and media files are an important part of most apps, and also one of the most common sources of performance issues. File access is traditionally a key performance bottleneck for apps that display gallery views of files, such as photo albums. Accessing an image can be slow because it takes memory and CPU cycles to store, decode, and display the image.

Instead of scaling a full-size image to display as a thumbnail, use the Windows Runtime thumbnail APIs. The Windows Runtime provides a set of APIs backed by an efficient cache that enables an app to quickly get a smaller version of an image to use for a thumbnail. These APIs can improve code execution times by a few seconds and improve the visual quality of the thumbnail. And because these APIs cache the thumbnails, they can speed up subsequent launches of your app. For more info, see [Accessing the file system efficiently](#).

### Retrieve thumbnails when accessing items

In addition to providing an API for retrieving thumbnails, the Windows Runtime also includes a [setThumbnailPrefetch](#) method in its API. This method specifies the type and size of thumbnails that the system should start loading immediately when items are accessed, instead of retrieving them on a case-by-case basis.

In Hilo JavaScript, the **MonthPresenter** class queries the file system for photos to display on the month page that meet a specific date criteria, and returns any photos that meet that criteria. The **\_getImageQueryOptions** function uses the [setThumbnailPrefetch](#) method to return thumbnails for the files in the query result set. Here's the code:

#### JavaScript: Hilo\Hilo\month\monthPresenter.js

```
_getImageQueryOptions: function () {
    var queryOptions = new search.QueryOptions(search.CommonFileQuery.orderByDate,
[".jpg", ".jpeg", ".tiff", ".png", ".bmp", ".gif"]);
    queryOptions.setPropertyPrefetch(fileProperties.PropertyPrefetchOptions.none,
[itemDateProperty]);
    queryOptions.setThumbnailPrefetch(fileProperties.ThumbnailMode.picturesView, 190,
fileProperties.ThumbnailOptions.useCurrentScale);
    queryOptions.indexerOption = search.IndexerOption.useIndexerWhenAvailable;
```

```
return queryOptions;
},
```

In this case, the code retrieves thumbnails that display a preview of each photo, at a requested size of 190 pixels for the longest edge of the thumbnail, and increases the requested thumbnail size based on the pixels per inch (PPI) of the display. Using the [setThumbnailPrefetch](#) method can result in improvements of 70 percent in the time it takes to show a view of photos from the user's Pictures library.

## Release media and stream resources when they're no longer needed

Media file resources can greatly increase the size of your app's memory footprint. So it's important to release the handle to media as soon as the app is finished using it. Releasing media streams that are unused can significantly reduce the memory footprint of your app and help keep it from being closed when it's suspended.

For example, Hilo releases instances of the [InMemoryRandomAccessStream](#) class when it no longer needs them by calling the [close](#) method on the objects. Here's the code:

### JavaScript: Hilo\Hilo\imageWriter.js

```
if (memStream) { memStream.close(); }
if (sourceStream) { sourceStream.close(); }
if (destStream) { destStream.close(); }
```

For more info, see [Accessing the file system efficiently](#).

## Optimize ListView performance

When you use a [ListView](#) control, there are a number of ways to optimize performance. You should optimize the following experiences:

- **Initialization.** The time interval starting when the control is created and ending when items are shown on screen.
- **Touch panning.** The ability to pan the control by using touch and ensure that the UI doesn't lag behind the touch gestures.
- **Scrolling.** The ability to use a mouse to scroll through the list and ensure that the UI doesn't lag behind the mouse movement.
- **Interaction for selecting, adding and deleting items.**

The [ListView](#) control depends on the app to supply the data sources and templating functionality to customize the control for the app. The way these are configured in the **ListView** implementation has a large impact on its overall performance. For more info, see [Using ListView](#) and [Working with data sources](#).

## Keep DOM interactions to a minimum

In the Windows Store app using JavaScript platform, the DOM and the JavaScript engine are separate components. Any JavaScript operation that involves communication between these components has a performance impact in comparison to operations that can be carried out completely in the JavaScript runtime. So it's important to keep interactions between these components to a minimum.

Use DOM objects only to store information that directly affects how the DOM lays out or draws elements. Using DOM objects can result in a 700 percent increase in access time as compared to accessing variables that aren't attached to the DOM. For more info, see [Writing efficient JavaScript](#).

## Optimize property access

The flexibility of being able to add properties to and remove properties from individual objects on the fly results in a significant performance impact because property-value retrieval requires a dictionary lookup. You can speed up property access for certain programming patterns by using an internal inferred type system that assigns a type to objects that have the same properties. To take advantage of this optimization, you should:

- Use constructors to define properties.
- Add properties to objects in the same order, if you create multiple instances of an object.
- Don't delete properties, because doing so can greatly degrade the performance of operations on the object that contained the property.
- Don't define default values on prototypes or conditionally added properties. Though doing so can reduce memory consumption, such objects receive different inferred types, so accessing their properties requires dictionary lookups.

For more info, see [Writing efficient JavaScript](#).

## Use independent animations

Windows Store apps built for Windows using JavaScript allow certain types of animations to be offloaded from the UI thread to a separate, GPU-accelerated system thread. This offloading creates smoother animations because it ensures that the animations aren't blocked by the actions in the UI thread. This type of animation is called an *independent animation*.

For example, Hilo uses CSS3 transitions and animations to independently animate the [transform](#) property to rotate a photo. For more info, see [Animating](#).

## Manage layout efficiently

To render an app, the system must perform complex processing that applies the rules of HTML, CSS, and other specifications to the size and position of the elements in the DOM. This process is called a *layout pass*, and it can have a performance impact.

A number of API elements trigger a layout pass, including [window.getComputedStyle](#), [offsetHeight](#), [offsetWidth](#), [scrollLeft](#), and [scrollTop](#). One way to reduce the number of layout passes is to combine API

calls that cause a layout pass. The more complex your app's UI, the more important following this tip becomes. For more info, see [Managing layout efficiently](#).

### Store state efficiently

Store session data in the [sessionState](#) object. This object is an in-memory data structure that's good for storing values that change often but need to be maintained even if Windows closes the app. It's automatically serialized to the file system when the app is suspended, and automatically reloaded when the app is reactivated. By using this object, you can help to reduce the number of file operations that your app performs. For more info, see [Storing and retrieving state efficiently](#).

### Keep your app's memory usage low when it's suspended

When your app resumes from suspension, it reappears nearly instantly. But when your app restarts after being closed, it might take longer to appear. So preventing your app from being closed when it's suspended can help to manage the user's perception and tolerance of app responsiveness. You can accomplish this by keeping your app's memory usage low when it's suspended.

When your app begins the suspension process, it should free any large objects that can be easily rebuilt when it resumes. Doing so helps to keep your app's memory footprint low, and reduces the likelihood that Windows will terminate your app after suspension. For more info, see [Optimizing your app's lifecycle](#) and [Handling suspend, resume and activation](#).

### Minimize the amount of resources that your app uses

Windows has to accommodate the resource needs of all Windows Store apps by using the Process Lifetime Management (PLM) system to determine which apps to close in order to allow other apps to run. A side effect of this is that if your app requests a large amount of memory, other apps might be closed, even if your app then frees that memory soon after requesting it. Minimize the amount of resources that your app uses so that the user doesn't begin to attribute any perceived slowness in the system to your app.

## Understanding performance

Users have a number of expectations for apps. They want immediate responses to touch, clicks, and key presses. They expect animations to be smooth. They expect that they'll never have to wait for the app to catch up with them.

Performance problems show up in various ways. They can reduce battery life, cause panning and scrolling to lag behind the user's finger, or make the app appear unresponsive for a period of time.

Optimizing performance is more than just implementing efficient algorithms. Another way to think about performance is to consider the user's perception of app performance. The user's app experience can be separated into three categories – perception, tolerance, and responsiveness.

- **Perception.** User perception of performance can be defined as how favorably they recall the time it took to perform their tasks within the app. This perception doesn't always match reality.

Perceived performance can be improved by reducing the amount of time between activities that the user needs to perform to accomplish a task.

- **Tolerance.** A user's tolerance for delay depends on how long the user expects an operation to take. For example, a user might find cropping an image intolerable if the app becomes unresponsive during the cropping process, even for a few seconds. You can increase a user's tolerance for delay by identifying tasks in your app that require substantial processing time and limiting or eliminating user uncertainty during those tasks by providing a visual indication of progress. And you can use async APIs to avoid making the app appear frozen.
- **Responsiveness.** Responsiveness of an app is relative to the activity being performed. To measure and rate the performance of an activity, you must have a time interval to compare it against. The Hilo team used the guideline that if an activity takes longer than 500 milliseconds, the app might need to provide feedback to the user in the form of a visual indication of progress.

## Improving performance by using app profiling

One technique for determining where code optimizations have the greatest effect in reducing performance problems is to perform app profiling. The profiling tools for Windows Store apps enable you to measure, evaluate, and find performance-related issues in your code. The profiler collects timing information for apps by using a sampling method that collects CPU call stack information at regular intervals. Profiling reports display information about the performance of your app and help you navigate through the execution paths of your code and the execution cost of your functions so that you can find the best opportunities for optimization. For more info, see [How to profile JavaScript code in Windows Store apps on a local machine](#). To learn how to analyze the data returned from the profiler, see [Analyzing JavaScript performance data in Windows Store apps](#).

When profiling your app, follow these tips to ensure that reliable and repeatable performance measurements are taken:

- At a minimum, take performance measurements on hardware that has the lowest anticipated specifications. Windows 8 runs on a wide variety of devices, and taking performance measurements on one type of device won't always show the performance characteristics of other form factors.
- Make sure that you profile the app on the device that's capturing performance measurements when it is plugged in, and when it is running on a battery. Many systems conserve power when running on a battery, and so operate differently.
- Make sure that the total memory use on the system is less than 50 percent. If it's higher, close apps until you reach 50 percent to make sure that you're measuring the impact of your app, rather than that of other processes.
- When you remotely profile an app, we recommend that you interact with the app directly on the remote device. Although you can interact with an app via Remote Desktop Connection, doing so can significantly alter the performance of the app and the performance data that you collect. For more info, see [How to profile JavaScript code in Windows Store apps on a remote device](#).
- Avoid profiling your app in the simulator because the simulator can distort the performance of your app.

## Other performance tools

In addition to using profiling tools to measure app performance, the Hilo team also used the Performance Analyzer for HTML5 Apps, and the Windows Reliability and Performance Monitor (perfmon).

The Performance Analyzer for HTML5 Apps is a tool that enables you to identify common performance issues in your HTML5 apps. The tool examines an app for common performance measures, like activation time, UI responsiveness, memory footprint, memory leaks, and memory growth. For more info, see [Performance Analyzer for HTML5 Apps](#).

Perfmon can be used to examine how programs you run affect your device's performance, both in real time and by collecting log data for later analysis. The Hilo team used this tool for a general diagnosis of the app's performance. For more info, see [Windows Reliability and Performance Monitor](#).

## Testing and deploying Windows Store apps: Hilo (JavaScript and HTML)

### Summary

- Use multiple modes of testing for best results.
- Use unit tests and integration tests to identify bugs at their source.
- Apps must be certified before they can be published in the Windows Store.

Testing helps you to ensure that your app is of high quality. We designed Hilo for testability and recommend that you do the same. Testing considerations for a Windows Library for JavaScript app are different from the testing considerations for a JavaScript-based web browser app. For Hilo we performed unit testing, integration testing, suspend and resume testing, localization testing, performance testing, and testing with the Windows App Certification Kit.

### You will learn

- How the various modes of testing contribute to the reliability and correctness of an app.
- How to write unit tests that test Windows Library for JavaScript functionality.

### Ways to test your app

You can test your app in many ways. Here's what we chose for Hilo.

- **Unit testing** tests individual functions in isolation. The goal of unit testing is to check that each unit of functionality performs as expected so that errors don't propagate throughout the app. Detecting a bug where it occurs is more efficient than observing the effect of a bug indirectly at a secondary point of failure.
- **Integration testing** verifies that the components of an app work together correctly. Integration tests examine app functionality in a manner that simulates the way the app is intended to be used. Normally, an integration test will drive the layer just below the user interface. In Hilo, there is no explicit distinction between unit and integration tests. But you can implicitly recognize this kind of test by the interactions of functions and objects working together to form view models for a supervising controller.
- **User experience (UX) testing** involves direct interaction with the user interface. This type of testing often needs to be done manually. Automated integration tests can be substituted for some UX testing but can't eliminate it completely.
- **Security testing** focuses on potential security issues. It's based on a threat model that identifies possible classes of attack.
- **Localization testing** makes sure that an app works in all language environments.
- **Accessibility testing** makes sure that an app supports touch, pointer, and keyboard navigation. It also makes sure that different screen configurations are supported.
- **Performance testing** identifies how an app spends its time when it's running. In many cases, performance testing can locate bottlenecks or routines that take a large percentage of an app's CPU time.

- **Device testing** ensures that an app works properly on the range of hardware that it supports. For example, it's important to test that an app works with various screen resolutions and touch-input capabilities.

The rest of this article describes the tools and techniques that we used to achieve these testing approaches in Hilo. For more info on test automation, see [Testing for Continuous Delivery with Visual Studio 2012](#).

## Using Mocha to perform unit and integration testing

You should expect to spend about the same amount of time writing unit and integration tests as you do writing the app's code. The effort is worth the work because it results in much more stable code that has fewer bugs and requires less revision.

In Hilo, we used Mocha for unit and integration tests. Mocha is a non-Microsoft test framework that you can use to test both synchronous and asynchronous code. We also paired Mocha with Chai, a non-Microsoft assertion testing library. The Hilo.Specifications project of the Hilo Visual Studio solution contains all of the code that supports Hilo testing.

**Note** The version of Hilo that contains the Hilo.Specifications project is available at [patterns & practices - HiloJS: a Windows Store app using JavaScript](#).

You can run the tests by setting the Hilo.Specifications project as the startup project, adding Mocha.js and Chai.js to the lib folder of that project, and then running the project. Here's the output from running the unit tests for the image query builder.

## Image Query Builder

when building a query

- ✓ should return a query object that can be executed

when serializing and then deserializing a query object

- ✓ should restore all of the options for the query

when deserializing a query object for an unsupported folder

- ✓ should throw an error explaining that the folder is unknown

when specifying a month and year for images

- ✓ should configure the query for the specified month and year

when executing a query that specifies the number of images to load

- ✓ should load the specified number of images

when executing a query that does specify the number of images to load

- ✓ should load all images in the folder

when specifying the index of a specific image to load

- ✓ should only load that one image when executing

when specifying the images should be bindable

- ✓ should return instances of bindable Picture objects 202ms

when building a query with an image index

- ✓ should load the one specified image [\(view source\)](#)

when executing an already built query and specifying an image index

- ✓ should load the one specified image

For more info about unit testing with Mocha, see [Testing asynchronous functionality](#) and [Testing synchronous functionality](#).

These tests specify a **beforeEach** hook to act as setup function that's invoked at the start of each unit test. Here's the code for the **beforeEach** hook:

### JavaScript: Hilo.Specifications\specs\queries\imageQueryBuilder.spec.js

```
describe("Image Query Builder", function () {

  var queryBuilder, storageFolder;

  beforeEach(function (done) {
    queryBuilder = new Hilo.ImageQueryBuilder();

    var whenFolder =
Windows.Storage.ApplicationData.current.localFolder.getFolderAsync("Indexed");
    whenFolder.then(function (folder) {
      storageFolder = folder;
      done();
    });
  });
});
```

```
    });
  });
```

The **beforeEach** hook simply creates an instance of the **ImageQueryBuilder** object for the folder named Indexed, which is in the root folder of the local app data store. This instance is then used in each unit test. The **beforeEach** hook accepts an optional single-parameter function known as the **done** function from the calling unit test. The **done** function is an anonymous function. This function, when present in your callback function, notifies Mocha that you're writing an asynchronous test. If the **done** function is omitted, the unit test runs synchronously. In the asynchronous callback function in the **beforeEach** hook, either the **done** function is invoked or a configurable time-out is exceeded, which both cause Mocha to be notified to continue running the rest of the unit test that invoked the **beforeEach** hook.

### Testing asynchronous functionality

You can test asynchronous functionality by specifying a function callback to be run when the test is complete. Here's the code for a unit test that checks whether a file system query specifies the number of images to return:

#### JavaScript: Hilo.Specifications\specs\queries\imageQueryBuilder.spec.js

```
describe("when executing a query that specifies the number of images to load",
function () {
  var queryResult;

  beforeEach(function () {
    queryResult = queryBuilder
      .count(1)
      .build(storageFolder)
      .execute();
  });

  it("should load the specified number of images", function (done) {
    queryResult.then(function (images) {
      expect(images.length).equals(1);
      done();
    }).done(null, done);
  });
});
```

In this unit test, the **beforeEach** hook is invoked to create an instance of the **ImageQueryBuilder** object. This stores a promise in a variable that provides access to the array of objects that was loaded by the query. The unit test then checks to ensure that only one image was returned before invoking the callback function to complete the execution of the unit test.

When the promise resolves, it provides a value as a parameter to the callback function. This resolution enables you to asynchronously return the data that was requested. You can then test the value of the

asynchronous callback function against an assertion or expectation. Here, we put the asynchronous test and the **done** call in the expectation's **it** function.

**Note** Although the **it** functions in these unit tests are asynchronous, the **beforeEach** hooks are not.

When you call an **expect** function and the expectation fails, an exception is thrown. Normally the Mocha test runner intercepts this exception and reports it as a test failure. But Windows Library for JavaScript promises intercept the expectation's error, preventing Mocha from receiving it. Another result of this interception is that the **done** function is never called, so the test will go into a wait state and Mocha will time out after 2 seconds, providing a report of the testing time out instead of the message from the failed expectation.

In Windows Library for JavaScript promises, when an exception is thrown in a **then** function, the error is forwarded to the error handling function of the promise's **done** function. So you can access the error that an expectation threw by chaining a **done** call onto the end of the **then** call.

To complete the asynchronous expectation, you must call Mocha's **done** callback function, passing the error that was caught as a parameter. When you pass any parameter to Mocha's **done** function, Mocha is notified that the test failed. Mocha assumes that the parameter you pass is the error from the expectation or the reason that the test failed. This means that you can call Mocha's **done** function with the error parameter of the error handling callback function.

A promise's **done** function takes 3 callback parameters—a *completed* callback function, an *error* callback function, and a *progress* callback function. If the value of the *completed* callback function is **null**, the fulfilled value is returned. Because the error handling callback function of the promise has the same method signature as Mocha's **done** callback function, the **done** function can be passed directly as the callback function.

## Testing synchronous functionality

You can test synchronous functionality by omitting a callback function. Here's the code for a unit test that checks whether the **ImageQueryBuilder** class can configure a file system query for a specified month and year:

### JavaScript: Hilo.Specifications\specs\queries\imageQueryBuilder.spec.js

```
describe("when specifying a month and year for images", function () {
    var queryOptions;

    beforeEach(function () {
        var query = queryBuilder
            // Query for January 2012.
            .forMonthAndYear(new Date(2012, 0))
            .build(storageFolder);

        queryOptions = new Windows.Storage.Search.QueryOptions();
        queryOptions.loadFromString(query._queryOptionsString);
    });
});
```

```

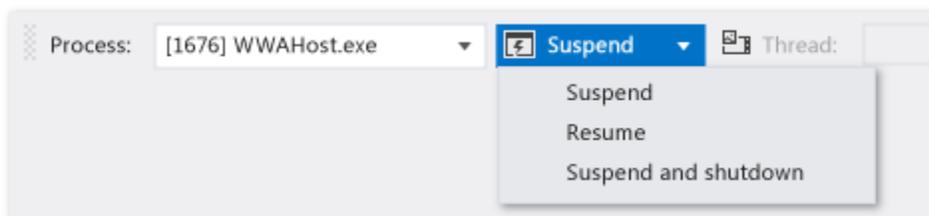
it("should configure the query for the specified month and year", function () {
    // The resulting query will always be against the local time zone.
    // This means that we need to adjust our spec to test for a local time.
    // Start with January 1st 2012 (local time).
    var start = new Date(2012, 0, 1, 0, 0, 0).toISOString().replace(/\.d\d\dZ$/,
    "Z");
    //End on one second before February 1st 2012 (local time).
    var end = new Date(2012, 0, 31, 23, 59,
59).toISOString().replace(/\.d\d\dZ$/, "Z");
    expect(queryOptions.applicationSearchFilter).equals("System.ItemDate:" +
start + ".." + end);
    });
});

```

In this unit test, we invoke the **beforeEach** hook to create an instance of the **ImageQueryBuilder** object. After execution of the **beforeEach** hook completes, we invoke the **forMonthAndYear** function on the **ImageQueryBuilder** instance, specifying a month and year as the function parameter. We then build the query for the specified month and year by calling the **build** function, passing a [StorageFolder](#) object as the function parameter. An instance of the [QueryOptions](#) class is then created and initialized with the query options from the **ImageQueryBuilder** instance. The unit test then checks that the [ApplicationSearchFilter](#) property of the **QueryOptions** instance contains the right month and year, in a correctly formatted string.

## Using Visual Studio to test suspending and resuming the app

When you debug a Windows Store app, the **Debug Location** toolbar contains a drop-down menu that enables you to suspend, resume, or suspend and shut down (terminate) the running app. You can use this feature to ensure that your app behaves as expected when Windows suspends or resumes it, or activates it after a suspend and shutdown sequence. Here's the drop-down menu that enables you to suspend the running app:

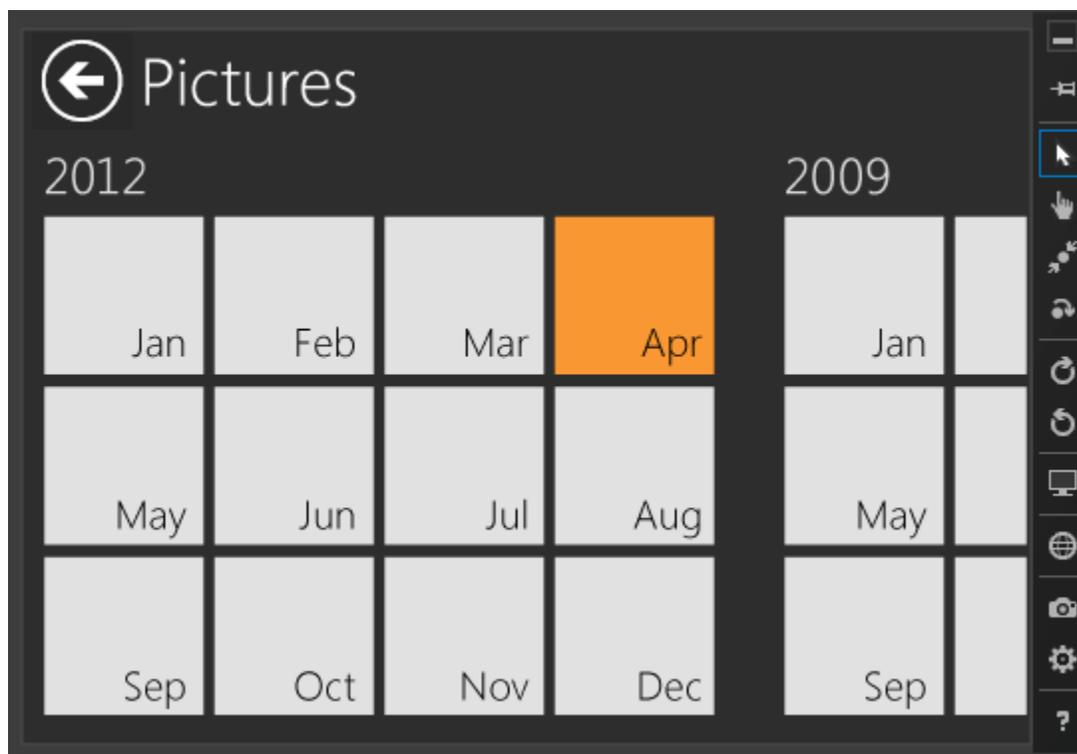


If you want to demonstrate suspending from the debugger, run Hilo in the Visual Studio debugger and set breakpoints in Hilo's [oncheckpoint](#) (or [suspending](#)) and [onactivated](#) event handlers. Then select **Suspend and shutdown** from the **Debug Location** toolbar. The app closes. Restart the app in the debugger, and the app follows the code path for resuming from the **terminated** state. For more info, see [Handling suspend, resume and activation](#).

## Using the simulator and remote debugger to test devices

Visual Studio includes a simulator that you can use to run your Windows Store app in various device environments. For example, you can use the simulator to check whether your app works correctly with a variety of screen resolutions and with a variety of input hardware. You can simulate touch gestures even if you're developing the app on a computer that doesn't support touch.

Here's Hilo running in the simulator:



To start the simulator, click **Simulator** in the drop-down menu on the **Debug** toolbar in Visual Studio. The other choices in this drop-down menu are **Local machine** and **Remote machine**.

In addition to using the simulator, we also tested Hilo on a variety of hardware. You can use remote debugging to test your app on a computer that doesn't have Visual Studio installed on it. For more info about remote debugging, see [Running Windows Windows Store apps on a remote machine](#).

## Using pseudo-localized versions for localization testing

We used pseudo-localized versions of Hilo for localization testing. See [Localizability Testing](#) for more info.

## Security testing

We used the STRIDE methodology for threat modeling as a basis for security testing in Hilo. For more info about that methodology, see [Uncover Security Design Flaws Using The STRIDE Approach](#).

## Using performance testing tools

In addition to using profiling tools to measure app performance, we also used the [Performance Analyzer for HTML5 Apps](#), [Windows Reliability and Performance Monitor](#), and the JavaScript Memory Analyzer. For more info, see [Analyzing memory usage in Windows Store apps](#) and [Improving performance](#).

## Making your app world ready

Preparing your app for international markets can help you reach more users. [Globalizing your app](#) provides guides, checklists, and tasks to help you create a user experience that reaches users by helping you to globalize and localize your Windows Store app. Hilo supports all Gregorian calendar formats. Its resource strings have been localized in 3 languages: English (United States), German (Germany), and Japanese.

Here are some of the issues we had to consider while developing Hilo:

- **Think about localization early.** We considered how flow direction and other UI elements affect users across various locales. We also incorporated string localization early to make the entire process more manageable.
- **Separate resources for each locale.** We maintain separate folders in the project for each locale. For example, `strings > en-US > resources.rejson` defines the strings for the en-US locale. For more info, see [Quickstart: Using string resources](#) and [How to name resources using qualifiers](#).
- **Localize the app manifest.** We followed the steps in [Localizing the package manifest](#), which explains how to use the **Manifest Designer** to localize the name, description, and other identifying features of an app.
- **Ensure that each piece of text that appears in the UI is defined by a string resource.** For example, here's the HTML that defines the app title that appears on the main page:

### HTML: Hilo\Hilo\hub\hub.html

```
<div data-win-control="WinJS.UI.HtmlControl"
  data-win-options="{ uri: '/Hilo/Controls/Header/Header.html',
titleResource: 'AppName.Text' }">
</div>
```

For the en-US locale, in the resource file we define **AppName.Text** as "Hilo."

- **Add contextual comments to the app resource file.** Comments in the resource file help localizers more accurately translate strings. For example, for the **AppName.Text** string, we provided the comment "Page title on hub page" to give the localizer a better idea of where the string is used. For more info, see [How to prepare for localization](#).
- **Define the flow direction for all pages.** We followed the steps in [How to adjust layout for RTL languages and localize fonts](#) to ensure that Hilo includes support for right-to-left layouts.

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

## Using the Windows App Certification Kit to test your app

To give your app the best chance of being certified, validate it by using the Windows App Certification Kit. The kit performs a number of tests to verify that your app meets certain certification requirements for the Windows Store. These tests include:

- Examining the app manifest to verify that its contents are correct.
- Inspecting the resources defined in the app manifest to ensure that they're present and valid.
- Testing the app's resilience and stability.
- Determining how quickly the app starts and how fast it suspends.
- Inspecting the app to verify that it calls only APIs for Windows Store apps.
- Verifying that the app uses Windows security features.

You must run the Windows App Certification Kit on a Release build of your app. If you run it on a Debug build, validation fails. For more info, see [How to: Set Debug and Release Configurations](#).

It's possible to validate your app whenever you build it. If you use Team Foundation Build, you can configure your build computer so that the Windows App Certification Kit runs automatically every time an app is built. For more info, see [Validating a package in automated builds](#).

For more info, see [How to test your app with the Windows App Certification Kit](#).

## Creating a Windows Store certification checklist

You'll use the Windows Store as the primary method to sell your apps or make them available. For info about how to prepare and submit your app, see [Selling apps](#).

As you plan your app, we recommend that you create a publishing-requirements checklist to use later when you test your app. This checklist can vary depending on the kind of app you're creating and on how you plan to monetize it. Here's our checklist:

1. **Open a developer account.** You must have a developer account to upload apps to the Windows Store. For more info, see [Registering for a Windows Store developer account](#).
2. **Reserve an app name.** You can reserve an app name for one year. If you don't submit the app within the year, the reservation expires. For more info, see [Naming and describing your app](#).
3. **Acquire a developer license.** You need a developer license to develop a Windows Store app. For more info, see [Getting a Developer License for Windows 8](#).
4. **Edit your app manifest.** Modify the app manifest to set the capabilities of your app and provide items such as logos. For more info, see [Manifest Designer](#).
5. **Associate your app with the Store.** When you associate your app with the Windows Store, your app manifest file is updated to include data that's specific to Windows Store.
6. **Copy a screenshot of your app to the Windows Store.**
7. **Create your app package.** The simplest way to create an app package is by using Microsoft Visual Studio. For more info, see [Packaging your app using Visual Studio 2012](#). An alternative way is to create your app package at the command prompt. For more info, see [Building an app package at a command prompt](#).

8. **Upload your app package to the Windows Store.** During the upload process, your app package is checked for technical compliance with the [certification requirements](#). If your app passes these tests, you'll see a message that indicates that the upload succeeded. If a package fails an upload test, you'll see an error message. For more info, see [Resolving package upload errors](#).

Though we didn't actually upload Hilo to the Windows Store, we did perform the necessary steps to ensure that it would pass validation.

Before you create your app package to upload it to the Windows Store, be sure to do these things:

- Review the app-submission checklist. This checklist indicates what information you must provide when you upload your app. For more info, see [App submission checklist](#).
- Ensure that you have validated a release build of your app with the Windows App Certification Kit. For more info, see [Testing your app with the Windows App Certification Kit](#).
- Take some screen shots that show off the key features of your app.
- Have other developers test your app. For more info, see [Sharing an app package locally](#).

Also, if your app collects personal data or uses software that's provided by others, you must include a privacy statement or additional license terms.

## Meet the Hilo team (Windows Store apps using JavaScript and HTML)

The goal of patterns & practices is to enhance developer success through guidance on designing and implementing software solutions. We develop content, reference implementations, samples, and frameworks that explain how to build scalable, secure, robust, maintainable software solutions. We work with community and industry experts on every project to ensure that some of the best minds in the industry have contributed to and reviewed the guidance as it develops. Visit the [patterns & practices Developer Center](#) to learn more about patterns & practices and what we do.

### Meet the team

This guide was produced by:



- **Program Management:** Blaine Wastell
- **Development:** Derick Bailey (Muted Solutions LLC), Christopher Bennage
- **Written guidance:** David Britch (Content Master Ltd.), Mike Jones
- **Test:** Larry Brader, Poornimma Kaliappan (VanceInfo)
- **UX and graphic design:** Deborah Steinke (Graphic Designer, adaQuest, Inc.), Howard Wooten
- **Editorial support:** Mick Alberts, John Osborne

We want to thank the customers, partners, and community members who have patiently reviewed our early content and drafts. We especially want to recognize Jose M. Alarcon (CEO, campusMVP.net), Srđan Božović (MFC Mikrokomerc), Pablo Cibraro (AgileSight), Andrew Davey, Robert Fite (Application Delivery Architect), Paul Glavich (Chief Technology Officer, Saasu.com), Ian Green (www.hardcore-games.tk), Adrian Kulp (CTO, Aeshen LLC), Alvin Lau (Solutions Consultant, Dimension Data (Singapore)), Thomas Lebrun, Chris Love, Yasser Makram (Independent Consultant), Ming Man, Chan (Section Manager, AMD), Christopher Maneu (UI Engineer, Deezer), Paulo Morgado (MVP, paulomorgado.net), Caio Proiete (Senior Trainer, CICLO.pt), Carlos dos Santos (Product Manager, CDS Informática Ltda.), Mitchel Sellers (CEO, Director of Development, IowaComputerGurus Inc.), Darren Sim (Partner, Similton Group LLP), Perez Jones Tsisah (Freelance Software Developer), Dave Ward (Encosia), and Tobias Zimmergren (Microsoft MVP) for their technical insights and support throughout this project.

We hope that you enjoy working with the Hilo source files and this guide as much as we enjoyed creating it. Happy coding!