

Making Everything Easier!™

Windows® 8

Application Development with HTML5

FOR
DUMMIES®

Learn to:

- **Build modern applications for the Windows Store**
- **Create user experiences that take advantage of the Windows 8 interface**
- **Sell your Windows apps in the Windows Store to users worldwide**
- **Use the new WinRT API with Visual Studio 2012**

Bill Sempf

Coauthor of C# 2010 All-in-One For Dummies

www.it-ebooks.info



Get More and Do More at Dummies.com®



Start with **FREE** Cheat Sheets

Cheat Sheets include

- Checklists
- Charts
- Common Instructions
- And Other Good Stuff!

To access the Cheat Sheet created specifically for this book, go to
www.dummies.com/cheatsheet/windows8applicationdevelopmentwithhtml5

Get Smart at Dummies.com

Dummies.com makes your life easier with 1,000s of answers on everything from removing wallpaper to using the latest version of Windows.

Check out our

- Videos
- Illustrated Articles
- Step-by-Step Instructions

Plus, each month you can win valuable prizes by entering our Dummies.com sweepstakes. *

Want a weekly dose of Dummies? Sign up for Newsletters on

- Digital Photography
- Microsoft Windows & Office
- Personal Finance & Investing
- Health & Wellness
- Computing, iPods & Cell Phones
- eBay
- Internet
- Food, Home & Garden

Find out “HOW” at Dummies.com

*Sweepstakes not currently available in all countries; visit Dummies.com for official rules.

www.it-ebooks.info



***Windows® 8
Application
Development
with HTML5***
FOR
DUMMIES®

***Windows® 8
Application
Development
with HTML5***
FOR
DUMMIES®

by Bill Sempf



John Wiley & Sons, Inc.

Windows® 8 Application Development with HTML5 For Dummies®

Published by
John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030-5774

www.wiley.com

Copyright © 2013 by John Wiley & Sons, Inc., Hoboken, New Jersey

Published by John Wiley & Sons, Inc., Hoboken, New Jersey

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Trademarks: Wiley, the Wiley logo, For Dummies, the Dummies Man logo, A Reference for the Rest of Us!, The Dummies Way, Dummies Daily, The Fun and Easy Way, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries, and may not be used without written permission. Microsoft and Windows are registered trademarks of Microsoft Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002.

For technical support, please visit www.wiley.com/techsupport.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2012950492

ISBN 978-1-118-17335-0 (pbk); ISBN 978-1-118-23995-7 (ebk); ISBN 978-1-118-26466-9 (ebk); ISBN 978-1-118-22704-6 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1



About the Author

Husband. Father. Author. Software composer. Brewer. Lockpicker. Ninja. Insurrectionist. Honorary blue teamer. Lumberjack. All words that have been used to describe **Bill Sempf** recently. I write some software, and this isn't my first book. Pleased to meet you.

Dedication

Thank you to my ever-patient family: my beautiful, brilliant wife, Gabrielle, amazing son, Adam, and incredible daughter, Kaelan. You are the reason I do stuff like this.

Author's Acknowledgments

This one has more than the usual number of suspects in the acknowledgements department. I have to first thank Katie Feltman, my acquisitions editor at Wiley, for seeing the promise of Windows 8 and agreeing to the book at all. She knows the technical world well, and me better. Thanks for your vision.

Linda Morris, the project editor, put up with innumerable changes in my personal writing style, as always happens during the writing of a book. How she made this collection of articles into a final product, I'll never know. I am just glad there are people like her out there.

My family, of course, had to put up with innumerable evenings and weekends of "Daddy has to write again?" I appreciate the patience, and am glad to be back watching Disney movies. No, really!

My local Microsoft evangelists have been instrumental in making sure I make a fool of myself as rarely as humanly possible. Jennifer Marsman, Brian Prince, and Jeff Blankenburg all played a tremendously important role in my growth in the Windows 8 arena, and I thank them.

The local Columbus development community never fails to make me think, and brings the reality of day-to-day development to my writing. Thanks to Samidip Basu, Rob Streno, and Leon Gersing especially for focusing my thought around the realities of app development.

Adam Kinney was my partner in arms in the two apps I built while writing this book. His experience at Microsoft notwithstanding, he provided an unusual clarity of thought about pragmatic JavaScript architecture, structuring apps, and focusing on the right features at the right time. Oscar Naim was the project owner of that project and drove us both to consider all angles of the user experience.

Publisher's Acknowledgments

We're proud of this book; please send us your comments at <http://dummies.custhelp.com>. For other comments, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002.

Some of the people who helped bring this book to market include the following:

Acquisitions and Editorial

Project Editor: Linda Morris

Acquisitions Editor: Katie Feltman

Copy Editor: Linda Morris

Technical Editor: Mike Spivey

Editorial Manager: Jodi Jensen

Editorial Assistant: Leslie Saxman

Sr. Editorial Assistant: Cherie Case

Cover Photo: © Marjan Veljanoski /
iStockphoto

Cartoons: Rich Tennant (www.the5thwave.com)

Composition Services

Project Coordinator: Katie Crocker

Layout and Graphics: Jennifer Creasey,
Joyce Haughey, Corrie Niehaus

Proofreaders: Cara Buitron, John Greenough

Indexer: Potomac Indexing, LLC

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Acquisitions Director

Mary C. Corder, Editorial Director

Publishing for Consumer Dummies

Kathleen Nebenhaus, Vice President and Executive Publisher

Composition Services

Debbie Stailey, Director of Composition Services

Contents at a Glance

<i>Introduction</i>	<i>1</i>
<i>Part I: Discovering Windows 8.....</i>	<i>7</i>
Chapter 1: Introducing Windows 8.....	9
Chapter 2: Embracing Windows 8 Style.....	25
Chapter 3: Getting a Grip on Windows 8 Development	49
Chapter 4: Setting Up a Windows 8 App.....	63
<i>Part II: Working with the Externals</i>	<i>81</i>
Chapter 5: Using Everyday Controls	83
Chapter 6: Laying Out the App.....	111
Chapter 7: Presenting Data.....	137
Chapter 8: Building Tiles and Using Notifications	159
<i>Part III: Digging into the Internals.....</i>	<i>187</i>
Chapter 9: Programming to Contract.....	189
Chapter 10: Talking to the Internet	209
Chapter 11: Managing the Process Lifecycle.....	227
Chapter 12: Keeping Local Storage.....	243
<i>Part IV: Getting Ready to Publish</i>	<i>261</i>
Chapter 13: Integrating with Hardware.....	263
Chapter 14: Preparing for the Store	279
Chapter 15: Going to the Cloud.....	299
Chapter 16: Making Money with Your App.....	311
<i>Part V: The Part of Tens</i>	<i>327</i>
Chapter 17: Ten App Ideas	329
Chapter 18: Ten Places to Find Samples and Info.....	335
<i>Index</i>	<i>341</i>

Table of Contents

<i>Introduction</i>	1
Who Should Read This Book	1
About This Book	2
Foolish Assumptions	3
How to Use This Book	3
How This Book Is Organized	3
Part I: Discovering Windows 8	4
Part II: Working with the Externals	4
Part III: Digging into the Internals	4
Part IV: Getting Ready to Publish	4
Part V: The Part of Tens	4
Icons Used in This Book	4
Conventions Used in This Book	5
Where to Go from Here	6
 <i>Part 1: Discovering Windows 8</i>	7
 Chapter 1: Introducing Windows 8	9
Finding the Path	10
Figuring out where Microsoft is headed	10
Building a simple Hello World app	12
Comparing programs from yesterday	17
Treating Users Right	20
Setting some design principles	20
Using the Design Checklist	21
Convention over configuration	22
Embracing the Developer Experience	22
Using the best developer tools on the planet	23
Folding in WinRT	24
 Chapter 2: Embracing Windows 8 Style	25
Designing with Windows 8 Style	26
Content before chrome	26
Layout	30
Interactions	33
Navigation	35
Keeping It Fast and Fluid	38
Use animation for a purpose	39
Design for touch first	39

Snapping and Scaling Beautifully.....	40
Design for multiple views.....	40
Design for multiple screen sizing.....	41
Deal with pixel density.....	41
Using the Right Contracts.....	42
Share.....	42
Search.....	42
Pickers.....	43
Settings.....	43
Investing in a Great Tile.....	44
Brand your tile.....	44
Drive traffic with secondary tiles.....	44
Keeping Your App Connected and Alive.....	45
Bring your tile to life.....	45
Use notifications.....	46
Roaming to the Cloud.....	46
Chapter 3: Getting a Grip on Windows 8 Development.....	49
Focusing on the Users.....	49
Immerse the user.....	50
Provide what the user needs.....	51
Design for touch.....	51
Enable multitasking.....	53
Assume connectivity.....	54
Taking Advantage of the Development Environment.....	55
Sticking with Visual Studio.....	55
Picking a language.....	56
Touring Expression Blend.....	57
Making the Change from Windows Development.....	60
Learning the new SDKs.....	60
Driving user confidence.....	60
Taking the next steps.....	61
Chapter 4: Setting Up a Windows 8 App.....	63
Getting Started in Visual Studio.....	63
Project types.....	64
Making a new project.....	65
Solution layout.....	67
Building Different Layouts.....	69
Grid layout.....	69
Split layout.....	70
Fixed layout.....	74
Getting Around.....	74
Navigation.....	75
Loading fragments.....	76
Adding fragments.....	77

Part 11: Working with the Externals..... 81**Chapter 5: Using Everyday Controls 83**

Using Basic HTML.....	83
Div	84
Text box	87
Select	91
Check box	93
ToggleSwitch	95
DatePicker and TimePicker	97
Informing the User.....	99
Determinate progress status.....	100
Indeterminate progress status.....	101
Ratings.....	103
Sliders.....	105
Playing Media Files with HTML5.....	107
Listening to audio	107
Watching video	109

Chapter 6: Laying Out the App 111

Building a Windows 8 Layout.....	112
Using typography.....	112
Making space.....	115
Keeping ergonomics in mind.....	117
Putting things in a grid.....	120
Interacting with Your App.....	121
Integrating navigation	122
Using the app bar.....	123
Making commands contextual	125
Making the App Fast and Fluid	128
Animating Windows.....	128
Snapping and scaling.....	133

Chapter 7: Presenting Data 137

Showing Single Bits of Data	137
Setting single values using the DOM	138
Using the declarative binding in WinJS.....	139
Listing Data.....	142
Dealing with collections of objects.....	143
Using the ListView	144
FlipView.....	148
Grouping, Sorting, and Selecting	149
Group control.....	149
Single and multiselect	151
Built-in animations.....	152
Working with Groups Using Semantic Zoom	153
Using Semantic Zoom.....	154
Considering the technical details	156

Chapter 8: Building Tiles and Using Notifications. 159

Using Basic Tiles	160
Core to the Windows 8 design language	161
Tile capabilities	161
Tile expectations	162
Building Live Tiles	163
Configuring a basic tile	163
Publishing with the templates	165
Rendering content in the app's personality	167
Getting Notified	168
Making notifications part of your tile	169
Getting a message on the queue	170
Creating Secondary Tiles	171
Getting the idea of a secondary tile	172
Pinning a secondary tile	172
Handling launch from secondary tiles	174
Getting the User's Attention with Toast	174
Deciding when to use Toast	175
Calling on Toast from your app	176
Giving the user control	179
Notifying Users When Your App Isn't Running	180
Designing for Windows Push Notification Service	181
Registering your app	182
Pushing notifications	183
Painting it Azure	185

Part III: Digging into the Internals 187**Chapter 9: Programming to Contract. 189**

Coding to a Whole New Tune	189
The Search Contract	190
Searching within your app	191
Returning results to other apps	194
The Share Contract	194
Loving sharing	195
Allowing sharing from your app	196
Becoming a share target	199
Settings	201
Playing to Devices	204
Getting the gist of Play To	204
Coding a Play To example	205
File Picking from App to App	206

Chapter 10: Talking to the Internet	209
Building Different Layouts	209
The Grid Application template	210
List layout	212
Communicating with the Internet	214
The original XMLHttpRequest	215
XHR: Microsoft's little secret	216
A promise kept	217
Handling Data	219
Setting the stage	219
Calling the feed	220
Formatting the results	222
Chapter 11: Managing the Process Lifecycle	227
Handling App Suspension	228
Switching tasks	228
Dealing with suspension and termination	231
Registering a resume event	233
Making Your App Look Alive	234
Dealing with application launch	235
Doing some things after suspension	236
Running background tasks	239
Using live tiles	242
Chapter 12: Keeping Local Storage	243
Putting Settings Where You Can Get Them	244
Organizing local settings	244
Shh! Roaming settings, too	246
Keeping more stuff with ApplicationDataContainer	246
Keeping settings in files	249
Giving the user access with the Settings charm	249
Filing Things Away for Later	252
Using the file system	252
The File Picker contract	256
Taking Advantage of Deployable Databases	258
Providing some relational power	258
Using the HTML5 database options	260
Part IV: Getting Ready to Publish	261
Chapter 13: Integrating with Hardware	263
Using the Camera and Microphone	264
Windows.Media API	264
Accessing the webcam from your app	266

Collecting Data from Sensors.....	269
Getting the user's location.....	269
Determining if the device is moved.....	269
Being aware of lighting.....	272
Touching the Surface	273
Comprehensive mouse and pointer features	273
Writing with the pen.....	273
Chapter 14: Preparing for the Store.....	279
Testing Your App.....	279
Old-fashioned testing	280
Getting a unit testing framework	280
Verifying runtime behavior	282
Verifying lifecycle	284
The Windows App Certification Kit.....	285
Manually Checking Windows Store Style	287
Pushing to the Store.....	289
Registering for a developer account	290
Submitting an app.....	291
Surviving testing	296
Managing Your App's Existence	297
Chapter 15: Going to the Cloud.....	299
Touring Azure	300
Announcing Windows Azure Mobile Services	301
Getting what you need to get started.....	301
Creating a new mobile service.....	302
Constructing a database.....	303
Connecting your Windows Store app.....	304
Running your app with a service	306
Taking it to the next level	307
Pushing Data to the User.....	309
Getting Authenticated.....	310
Chapter 16: Making Money with Your App	311
Adding Ads	312
Designing for advertising	312
The Windows 8 Ads SDK.....	314
Using other ad networks.....	318
Handling In-App Purchasing.....	320
Planning for app expansion.....	321
Getting your app ready for purchasing content	322
Adding the purchase UI.....	324
Setting up the Store for your app	325

Part V: The Part of Tens 327**Chapter 17: Ten App Ideas329**

Publishing Social Content.....	329
Integrating Planning Tools	330
Finding Your Way Around	330
Playing the Day Away.....	330
Reading What There Is to Read	331
Organizing Your Stuff.....	331
Painting with Photos	332
Managing Your Business	332
Keeping Healthy.....	332
Going Shopping.....	333

Chapter 18: Ten Places to Find Samples and Info 335

dev.windows.com.....	335
design.windows.com.....	336
social.msdn.microsoft.com/forums	336
www.stackoverflow.com.....	337
jQuery.com.....	337
www.buildwindows.com.....	337
jsonformatter.curiousconcept.com	338
www.w3schools.com/js	338
windowsteamblog.com	338
www.microsoftstore.com	339

Index 341

Introduction

In 2000, I received a phone call. An editor from a book publisher had read my writing about Microsoft .NET Live Services. She would like to know if I could write a chapter on web services for a new VB.NET book.

“Sure,” I said. “How different could it be from VB6, right?”

Well, those of you old enough to remember the .NET revolution will know how ludicrous that statement is. VB.NET changed everything, and I spent four weeks not sleeping as I learned not only a new language and framework, but a new way of thinking about how XML web services are implemented in the Microsoft world.

The book you are holding represents a similar revolution.

This time, the change is in the venerable Win32 stack. WinRT is a new core Windows library, 15 years overdue. It gives the developer unprecedented control over the hardware for which they are coding. It, however, comes at something of a cost.

To code for WinRT and the new Windows 8 desktop apps, you need to think about the user experience like never before. Microsoft has followed in the footsteps of Google and Apple, defining how apps interact with the user, and how a user interacts with apps.

Sure, you can still write the old Forms-style apps. In fact, you can still write a VB6 app and expect it to run on Windows 8. But if you want the app to appear in the Microsoft Store for instant access by a cool billion users, you have to play by the rules.

Fortunately for you, the rules are all in this book.

Who Should Read This Book

Windows 8 is a reimagining of Windows. Specifically, it doesn't have any Windows. How about that?

Clearly, the mobile revolution has hit home. Although there are still 10 Windows PCs sold for every iPad, there are still a *lot* of iPads out there, and given a choice, most people would rather use a tablet than a PC.

Microsoft has been writing software for tablets since Windows XP. The pen support in XP and Office 2003 was remarkable. Nonetheless, the hardware wasn't there yet.

Apple pushed the hardware barrier. Google followed. Now, Microsoft is ready to take center stage. They take ten years of experience and the best engineers in the industry and make things happen.

What is happening is you.

If you are ready to take advantage of the biggest developer opportunity in the history of computing, you are holding the right book. The apps you write after reading this will be available, through a simple search, to five hundred thousand users by the end of 2013.

That's three times more than there are Apple users out there.

In these pages, you get the gist of what Microsoft is trying to do with Windows 8, you learn how to make your app idea work in that system, and you get the tools you need to make it a reality. My goal as an author is to make sure your idea works in Windows 8.

About This Book

Windows 8 development can be done, now, for the first time, in HTML. No black box Windows forms or confusing XAML. Just plain old HTML, the same language we have been building web applications with since 1992.

That's what this book is about. Writing Windows 8 apps using HTML, CSS, and JavaScript — commonly called HTML5.

I focus on the Windows 8–specific stuff in this book. If you don't know HTML, please read the “Foolish Assumptions” section of this Introduction carefully. This isn't an HTML book.

You can write Windows 8 apps in C# or C++. C# apps are built using XAML, and, by design, are more business-oriented than C++ or HTML5. C++ apps are going to be the domain of game developers using DirectX. I do not cover either of these. If you are an enterprise or game developer, you will learn from this book, but it isn't going to give you everything you need.

Foolish Assumptions

In order to build software for Windows 8, I assume you have (at a minimum) Visual Studio 2012 Express. This software is the basis for the whole book and is used in all of the examples.

The good news is that it's free. Yup. Microsoft gives away the tools you need to write Windows Store apps. You can just go download it at msdn.microsoft.com/vstudio/express.

The other thing you need to use this book is an understanding of HTML, CSS, and JavaScript. I do not teach HTML5 in this book at all. If you need help with it, I recommend *HTML, XHTML, & CSS All-In-One For Dummies*, 2nd Edition, by Andy Harris (published by John Wiley and Sons, Inc.) for a guide to building HTML websites. From there, I'll take you the rest of the way to Windows 8.

Oh, one other thing. You'll need a willingness to learn and a little patience. You might be a web developer, or a mobile developer, or a .NET developer, or just someone with a good idea for an app. It's tough to write on a technical topic with so many potential audiences, so sometimes you'll have to bear with me.

How to Use This Book

This is my tenth *For Dummies* book, and the third I have written from scratch. This time, I think I have made sure that you can turn to any chapter in the book and start working. Try it.

The goal of the *For Dummies* series is to make you successful. Striving toward that goal, this book was designed to make sure that you can use the index or Table of Contents and just turn to the page that interests you. Then you can start working.

That said, I recommend reading from start to finish. There is a lot to learn about Windows 8. This book won't teach you everything, but it will give you a really good backlog of information. If you read it from front to back, you will have a tremendous background in Windows 8 development from the HTML5 side.

How This Book Is Organized

This book contains five Parts.

Part I: Discovering Windows 8

There are a lot of differences between coding for Windows 8 and . . . well . . . everything else. These four chapters give you everything you need to understand the changes.

Part II: Working with the Externals

The user is the center of the Windows 8 world, and, as such, the user interface is often the center of the Windows 8 app. In Part II, I go over all the various user interface features built into the APIs (application programming interfaces) you'll use.

Part III: Digging into the Internals

User interfaces are all well and good. At some point, though, your app has to *do* something. Here I cover contracts, networking, data and the very important program lifecycle.

Part IV: Getting Ready to Publish

After you have everything working, you need to fine-tune the edges and get ready to head to the store. In Part IV, I cover some things that you might not have thought of, like integrating with sensors, using Azure, advertising, and in-app purchasing. Of course, I talk about testing and actually publishing, too.

Part V: The Part of Tens

In the Part of Tens, if you are long on desire but short on ideas for apps, I'll get you started with ten app ideas. You'll find a fabulously useful list of where to get more information, too.

Icons Used in This Book

Throughout the pages of this book, I use the following icons to highlight important information:



This scary-sounding icon flags technical information that you can skip on your first pass through the book.



The Tip icon highlights a point that can save you a lot of time and effort.



Remember this information. It's important.



Try to retain any Warning information you come across, too. This one can sneak up on you when you least expect it and generate one of those extremely-hard-to-find bugs. Or, it may lead you down the garden path to La-La Land.

Conventions Used in This Book

Throughout this book, I use several conventions to help you get your bearings. Terms that aren't "real words," such as the names of program variables, appear in *this* font to minimize confusion. Text you should type is in **bold**. New terms are in *italics*. Program listings are offset from the text this way:

```
use System;
namespace MyNamespace
{
    public class MyClass
    {
    }
}
```

Each listing is followed by a clever, insightful explanation. Complete programs are included on the website for your viewing pleasure; small code segments are not.

When you see a command arrow, as in the instruction "Choose File⇨Open With⇨Notepad," you simply choose the File menu option. Then, from the menu that appears, choose Open With. Finally, from the resulting submenu, choose Notepad.

Where to Go from Here

Just write apps. This book gives you 80 percent of what you need to build even fairly complex apps, and the rest you can find in the additional resources I list in Chapter 18. The code for the examples in this book can be found at dummies.com/go/windows8appdevhtml5fd. I say just go and be creative! Get your apps in the Microsoft Store. And remember us little people when you get rich and famous.

Occasionally, we have updates to our technology books. If this book does have technical updates, they will be posted at dummies.com/go/windows8appdevhtml5fdupdates.

Part I

Discovering Windows 8

The 5th Wave

By Rich Tennant



"We're here to clean the code."

In this part . . .

“*W*indows 8 is a reimagining of Windows.”

Companies use the term *reimagining* all the time, but I haven't seen it more correctly used in a long time. Windows 8 is really new. And different. And coding for it is new and different. In this Part, I show you how to get started from scratch with Windows 8 development.

Chapter 1

Introducing Windows 8

In This Chapter

- ▶ Getting a grip on the new Windows
 - ▶ Starting fast out of the gate with Hello World
 - ▶ Deciding what development technology to use
 - ▶ Making sure that you put users first
-

For some of you, this chapter is an introduction, for others, a reminder. Windows 8 uses the same interface pattern used by Xbox Live and the Windows Phone. If you have used either, you know Windows 8.

Welcome to a new world of Windows.

A lot has changed since Windows 3.1 gained general popularity in 1991, both in terms of devices available to host an operating system and the bandwidth necessary to connect them. Sometimes Microsoft has blazed a trail, sometimes they have fallen behind, but Windows has always managed to more or less stay abreast of the world of computing.

Tablets are a story of both greatness and woe. Windows XP was tablet-ready, but the hardware world wasn't. We ended up with five-pound tablets that had a three-hour battery life and required a stylus. Apple and Google surged ahead as soon as the hardware was truly ready for the form factor.

Microsoft has an ace in the hole, though. Windows is present on 250,000,000 devices worldwide, and some of them are even legally licensed. Windows runs the majority of businesses on the planet. It is everywhere.

Windows 8 is not just a tablet operating system (OS). Microsoft has not chosen to make a separate OS for devices and phones like Apple has. They are upgrading the core OS to handle tablets natively. This means that the most popular OS on the planet, with the best developer tools on the planet, will soon work on tablets and have an app store like Google and Apple. That, in a word, is huge.

You see, rather than writing an app that works on one version of, say, six million Android phones out there, or that works on the iPhone 4S and nothing else, you can write something that works on nearly everyone's laptop. Then you can test it, market it, and sell it online with automatic installation, just like on those phones and tablets. It's a much, much bigger pond to fish in. But wait, there's more!

In order to do this, you probably need to learn Windows Presentation Foundation and C# and a bunch of other junk that you have never needed before and won't ever need again, right? *No*. You can build native Windows applications with HTML and JavaScript, just like you build web pages.

That is worth saying again. You can build native Windows applications for Windows 8 using HTML and JavaScript.

But I'm getting ahead of myself. In this chapter, I cover what Windows 8 is, how you code for it, and how it fits into the grand scheme of your overall software development practice.

Finding the Path

If you have followed the Microsoft developer experience for any number of years, you probably know that it follows a logical path. As Microsoft has embraced the open web more and more, they invest more and more in tools to make programs for normal people rather than line-of-business applications.

Access, FrontPage, and even Visual Basic 6 (VB6) can be described as Microsoft technologies for power users. They feature the core Microsoft Development technologies (OLE, ASP, and VBScript) but have template code and complete graphical user interfaces (GUIs) to make the product easier to use, if less functional.

What's been missing are tools for professional programmers — or good hobbyists — to make simpler consumer-grade programs that look great. There were no tools and no platform for this: People were on their own.

With Windows 8, Microsoft is changing that. You can get a preview of their path by looking at the Xbox and the Windows Phone. Apple and Google are spearheading a trend toward nice-looking, single-user apps, and Microsoft is right there.

Figuring out where Microsoft is headed

But how exactly is Microsoft working into the consumer market for apps? And where is the endpoint?

The plan is large and sophisticated, but by focusing on just the stuff that matters, you can get a clear view of it. In this chapter's introduction, I mentioned the existing Windows 8 applications in the Windows world — Xbox and Windows Phone. Figure 1-1 shows those two alongside a Windows 8 tablet, and the similarities are clear.



a



b



c

Figure 1-1:
Examples of
Windows 8
in the wild.

You can tell, just by looking at the applications, that Microsoft is trying to do a couple of things:

- ✓ **Optimize for touch:** It doesn't matter if you are using a finger on a Windows Phone, a Kinect on the Xbox, or a stylus in Windows 8 — the user experience is driven by touch.
- ✓ **Let users focus on one thing at a time:** Look at your PC. Every application has a menu and status bar. There is a menu and status bar on the OS itself! The idea is to increase productivity by easily switching between tasks. It's a multitasker's dream. Windows 8 isn't like that: It's all about one thing at a time.
- ✓ **Focus on consumption, not creation:** All three platforms are more for reading blog posts than writing them, watching movies rather than making them. If you want to create content, change to the desktop.

Does this mean that Windows desktop apps are dead as we know them? Not at all. The Windows 8 experience, effectively, is an addition to the regular Windows 7 experience. It really is just an enhanced Start bar that can run applications of its own.

Building a simple Hello World app

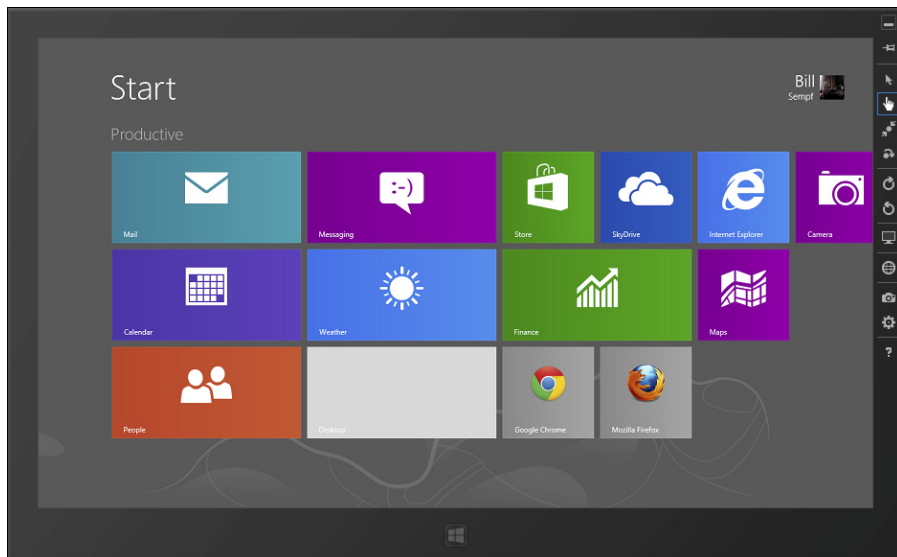
Before you get started building a Hello World app, make sure you have these essentials:

- ✓ A Windows 8 machine with some input mechanism, preferably a keyboard and mouse
- ✓ Visual Studio 2012
- ✓ Expression Blend

Visual Studio runs on the desktop, although you access it from the Windows 8 Start screen. Figure 1-2 shows my tablet's Windows 8 menu, with the Visual Studio icon on the center right. Give that a click, and you are off to the races.

To get an idea of how a Windows 8 application works, your best bet is to build a (very) simple one.

Figure 1-2:
My
Windows 8
menu
screen.



Basically, Windows 8 applications that use HTML5 as the user interface markup are constructed using an HTML file, a Cascading Style Sheets (CSS) file, and a JavaScript file that are compiled into a package. Although it's possible to just take an HTML application for the web and run it in Windows 8, if you want to make use of any Windows-specific features, you need to branch out a little.

1. **Open Visual Studio 11 by clicking the icon in the Windows 8 menu.**
2. **Click New Project in the upper left corner of the designer to start a new project.**

You'll see the New Project dialog box, as shown in Figure 1-3.

3. **In the tree view to the left, select Templates → JavaScript → Microsoft Store.**
4. **In the window to the right, select the Blank App template.**

This gives us the bare essentials for creation of a new application.

5. **In the Name field, enter Hello World.**

Why be original, right?

6. **Click OK.**

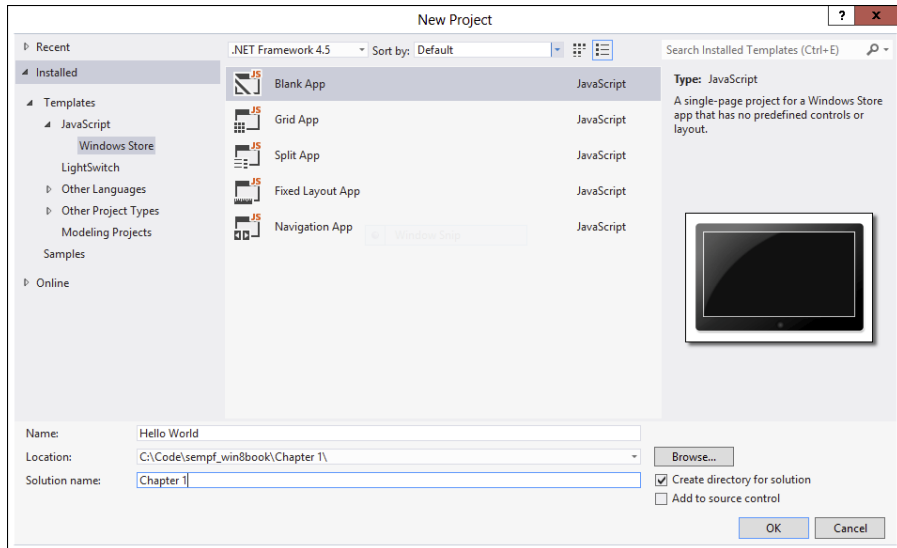


Figure 1-3:
Starting a
new project.

Visual Studio grunts and growls for a minute, and then gives you the basic files for a Windows 8 application. These consist of the following:

- ✓ **Default.html:** The starting form, page, window, view, and what have you for the Windows 8 application. This is where you lay out the form.
- ✓ **Default.css (in the CSS folder):** This file is referenced by default.html. It is unsurprisingly the starting location for style sheet information. This is where you set fonts, colors, and locations of items in the HTML file.
- ✓ **Default.js (in the JS folder):** The JavaScript file is also referenced by default.html. This is where the application loader looks for hints on how to get the application up and running. All application logic goes in .JS files.
- ✓ **Hello World_TemporaryKey.pfx:** This is the key you use to sign your app before it goes to the store. Eventually, you'll have an organizational key. I'll handle all of that in Part IV.
- ✓ **Package.appxmanifest:** The most interesting file in the batch, this is a mashup on the Project properties, Deployment config, and app.config of the .NET world.



You don't have to have all of these files. The only required files are default.html and package.appxmanifest. Just like with a regular HTML-based web page, the JavaScript and CSS can be introduced directly in the HTML. Also, you can use multiple JS files and reference them all, and cascade the CSS as usual in CSS3.

As you would expect, the code in the template runs by itself, although it just shows a blank screen. To get a little more spice in there, we need some HTML and JavaScript:

1. In the `default.html` file, add an `H1` tag with an ID of `headline` in the empty body tag. Also change that first style reference to the `light` style. The resulting HTML should look like this:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Hello_World</title>

  <!-- WinJS references -->
  <link href="//Microsoft.WinJS.1.0/css/ui-light.css"
        rel="stylesheet" />
  <script src="//Microsoft.WinJS.1.0/js/base.js">
    </script>
  <script src="//Microsoft.WinJS.1.0/js/ui.js">
    </script>

  <!-- Hello_World references -->
  <link href="/css/default.css" rel="stylesheet" />
  <script src="/js/default.js"></script>
</head>
<body>
  <h1 id="headline"></h1>
</body>
</html>
```

2. In the `default.js` file, set the `innerText` property of the new headline item to **This is my first Windows 8 app**. It will look something like this:

```
// For an introduction to the Blank template, see the
// following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232509
(function () {
  "use strict";

  var app = WinJS.Application;
  var activation = Windows.ApplicationModel.
    Activation;
  WinJS.strictProcessing();

  app.onactivated = function (args) {
    if (args.detail.kind === activation.
        ActivationKind.launch) {
```



```
    if (args.detail.previousExecutionState !==
        activation.ApplicationExecutionState.
            terminated) {
        var headline = document.
            getElementById("headline");
        headline.innerText = "This is my first Windows
            8 app.";
    } else {
        // TODO: This application has been reactivated
        // from suspension.
        // Restore application state here.
    }
    args.setPromise(WinJS.UI.processAll());
}
};

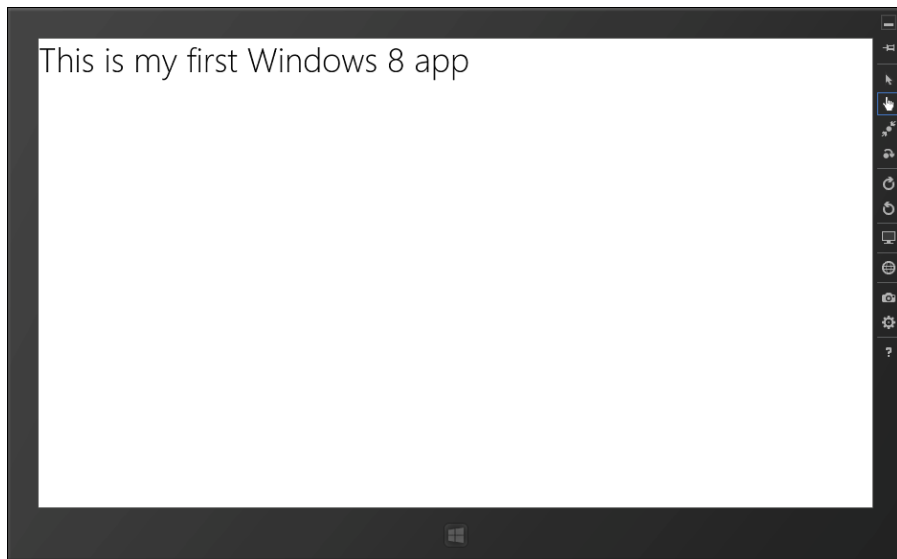
app.oncheckpoint = function (args) {
    // TODO: This application is about to be
    // suspended. Save any state
    // that needs to persist across suspensions here.
    // You might use the
    // WinJS.Application.sessionState object, which is
    // automatically
    // saved and restored across suspension. If you
    // need to complete an
    // asynchronous operation before your application
    // is suspended, call
    // args.setPromise().
};
app.start();
})();
```

This is enough to get you started — at least it will be obvious how everything comes together.

To start debugging a Windows 8 program, press F5. The execution system is built right into Visual Studio. When you press F5, you're taken to the Windows 8 app. It should look like Figure 1-4, although there isn't much to see. Windows 8 apps have no status bar, no menu bar, and no Close button: They are boring even by the pretty low standard set by Hello World apps globally.

This basic template for Windows 8 apps doesn't have any navigation. Flicking up from the bottom doesn't bring up anything of interest. Flicking from the right brings up the Charms bar just like any other Windows 8 application. If you need more information on Windows 8 design, I give a breakdown in Chapter 2.

Figure 1-4:
The
not-very-
exciting
Hello World
app.



If you perform the sweep gesture from the left, it returns you to the desktop and Visual Studio. From there, you can click the Stop button in Visual Studio to end the session. The Windows 8 application closes.

Debugging works as convention would dictate. Breakpoints are honored in JavaScript just like they are in the majority of other languages. When a breakpoint is reached in Visual Studio, the Windows 8 app pauses and Windows seamlessly brings you back to the debugger. I discuss the debugger more in Part II, when I start showing you how to squash bugs.

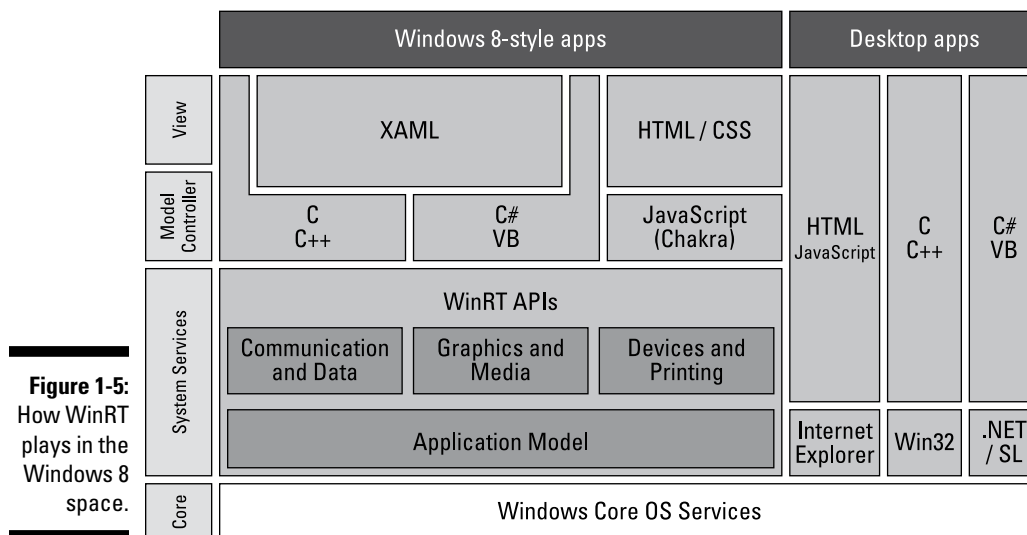
Comparing programs from yesterday

In some ways, coding for a Windows 8 application is no different than writing a website, even outside the Microsoft stack. HTML5, CSS, and JavaScript are common standards, and you could take the code from the Hello World application above and run it in any web browser dating back ten years or so.

Going back one version, the .NET Framework handled rendering the user interface details. XAML or Windows Forms would convert layout code to the Windows programs we are used to seeing, with status bars and menus and whatnot. Before that was a myriad of technologies, including the Ruby forms engine in VB6. Now there is WinRT, or Windows Runtime, which takes the layout code provided by the HTML and the logic provided by the JavaScript and converts that into a Windows 8–style app.

The user interface is in the lead

Figure 1-5 gives a pretty good overview of how WinRT plays into the structure of Windows 8 apps. Both XAML and HTML can be used to build layout code. A bunch of languages, including JavaScript, C#, VB.NET, and C++ can be used to write logic. WinRT provides the communication with Windows.



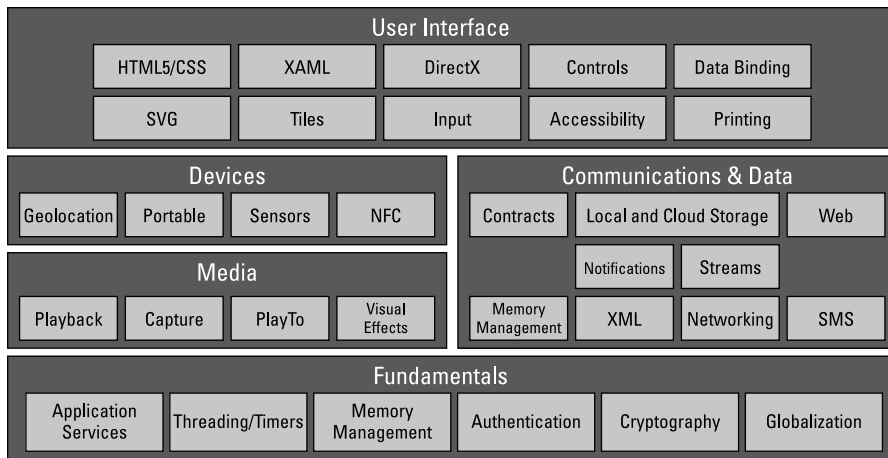
In order to create a smoother user experience and defend the developer from unnecessary technical detail, Microsoft has encapsulated DXGI (the DirectX Graphics Infrastructure) at the layout layer of the application, so that layout code is segmented from the rest of the application. Figure 1-6 shows how the Windows 8 application APIs (application programming interfaces) work together to bring this experience to the developer.

This is a good thing. If you have ever programmed in DirectX, you know that it practically requires a college degree in mathematics to just get started. On the other hand, the watered-down `System.Drawing` namespace in .NET provides only the barest of basics to get by graphically. When readers ask me if they can write games in .NET, I used to refer them to the Xbox because `System.Drawing` is so weak. Now there is another option.



You can learn a lot more about DXGI at <https://mdxgi.codeplex.com>. This is a managed version of the library (meaning that you can call it from .NET).

Figure 1-6:
The user
interface
architec-
ture.



This is not .NET

When the details of the Windows 8 user interface started to leak out in the summer of 2010, many people were worried that Microsoft was shifting away from .NET. This would have been a really big deal because it wasn't yet ten years old. Many people have pinned the future of their companies to the success of .NET. There was much wailing and gnashing of teeth.

Instead, Microsoft went and wrote a new API for the Windows 8 apps. Why? A lot of it has to do with the graphics. .NET is a *line-of-business* API. It is designed to write business applications. Although you can write nice user experiences with .NET, that isn't .NET's main goal. The main goal is to get data from here to there.

Windows 8, on the other hand, is entirely about user experience. Although you can create and move data with Windows 8, it is largely about consuming said data. It is not a line-of-business application. You can use it for that, but that isn't the reason it was created. Table 1-1 gives you a better idea of how I believe each should be used.

Table 1-1 Signs Your App Might Be a Windows 8 App

<i>Signs Your App Should Be Windows 8</i>	<i>Signs Your App Should Be .NET</i>
Brings together data from various services for user consumption.	Allows editable access to a common data source.
Refined graphic presentation.	Basic graphic presentation.
Designed for a tablet-like interface.	Designed for a mouse and keyboard.



This chart is *not* Microsoft canon: It's my opinion. I have worked on 14 .NET books, and, although this is my first Windows 8 book, the difference between the two is as plain as the nose on my face. That being said, there is a lot of room for argument here. Feel free to head over to the book's website and argue with me!

Treating Users Right

Windows 8 apps are different. As I mentioned in the previous section, Windows 8 apps are optimized for touch, usually mashups, and often single-use. They are not complex, menu-driven, or multifunctional. They are usually for consumption, not creation. It is a totally different user experience that we are after here.

Because of this, there is a totally different design paradigm. Microsoft has tried to lay on the user interface guidelines before with mixed success, but this time they are serious! The development environment is set up to only allow certain kinds of user-interface elements; if you step away from the path, you will find the going rather rough.

Setting some design principles

Microsoft is being very clear in laying out some user-interface and architectural design patterns. It remains to be seen if these will be upheld in the review process, but they are good guidelines and should be followed.

I go into more detail in Chapter 2, but I want to use a few core principles to describe Windows 8 apps, which make the rest of this chapter more logical. Windows 8 apps are

- ✓ **Socially connected:** Windows 8 apps make use of social and public-sharing networks to broadcast information, connect with friends, and make new groups.
- ✓ **Living in a sandbox:** An install of the Windows 8 app should never alter the core functionality of the user's machine.
- ✓ **Designed for touch:** These apps are designed to be run on a tablet, but they should work with a mouse. Much more on this in Chapter 6.
- ✓ **Screen-size sensitive:** Like a web page, Windows 8 apps should be sensitive to resolution, orientation, and docking.
- ✓ **Made for multitasking:** Users can “snap” Windows 8 apps into specified parts of the screen. If you want a nice-looking app, you have to handle the ability to snap and fill in your interface.

Using the Design Checklist

Microsoft has implemented the Design Checklist to help with the core principles of designing for Windows 8. It covers what you need to make sure you have planned appropriately for your app.

- ✓ Basic design groups: Core things you need to make sure you have covered
 - App tile: The icon that the user touches to launch your app
 - Splash screen: What the user sees while your app is loading
 - Application lifecycle: Be ready when the app gets suspended, or the battery runs out
 - App bar: All your app's commands go in the same place
- ✓ Engaging the user with app contracts (Like Search and Share. See Chapter 9.)
 - Settings: How the app communicates with Windows
 - Search: How it applies to your app
 - Share: Be social
 - Play to: Stream to other devices
- ✓ Various views
 - Full screen
 - Snapped
 - Filled
 - Portrait
 - Scaled
- ✓ Adding features: Take advantage of the neat stuff you can do
 - Semantic zoom
 - Notifications
 - Roaming
 - Content tiles
 - Gesture library
 - File picker
 - Animations
 - User tiles

Convention over configuration

You may have noticed by now that the Windows 8 style is a lot about convention — *convention* being defined as the approach Apple chose for their iPad. You don't actually have a lot of artistic license over how the user will interact with the meta-information related to your app; it is pretty well defined by the Design Checklist and Core Principles.

Take navigation, for example. In Windows Forms application, navigation is all over the place. There are menus and ribbons and buttons (oh my!). In Windows 8, however, the navigation of the application should be document-oriented and in the app bar at the bottom of the application.

The programming model supports this, in fact. It supports it so well that there really isn't any other way to implement the app bar. To see what I mean, open up the Hello World application we made in “Building a simple Hello World app.”

In the HTML, inside the `body` tag, add a `div` that represents the AppBar control. Notice the `data-win-control` declaration:

```
<body>
  <h1 id="headline"></h1>
  <div data-win-control="WinJS.UI.AppBar" id="appbar">
    <button data-win-control="WinJS.
      UI.AppBarCommand"
      data-win-options="{icon:'back', id:'',
        label:'example', onclick:null, section:'global',
        type:'button'}"></button>
  </div>
</body>
```

Now you can run the application and have a nice back button in the app bar. All of your navigational stuff goes here: Start Over, Change Views, Go Home. Not in a menu, not in the page, but in the app bar.

Embracing the Developer Experience

If Microsoft is going after the tablet market, wooing developers is a good start. Most everyone agrees that the Android and iOS development experience have a lot to admire. The Windows 8 experience, like the .NET experience, is already very good and is getting better.

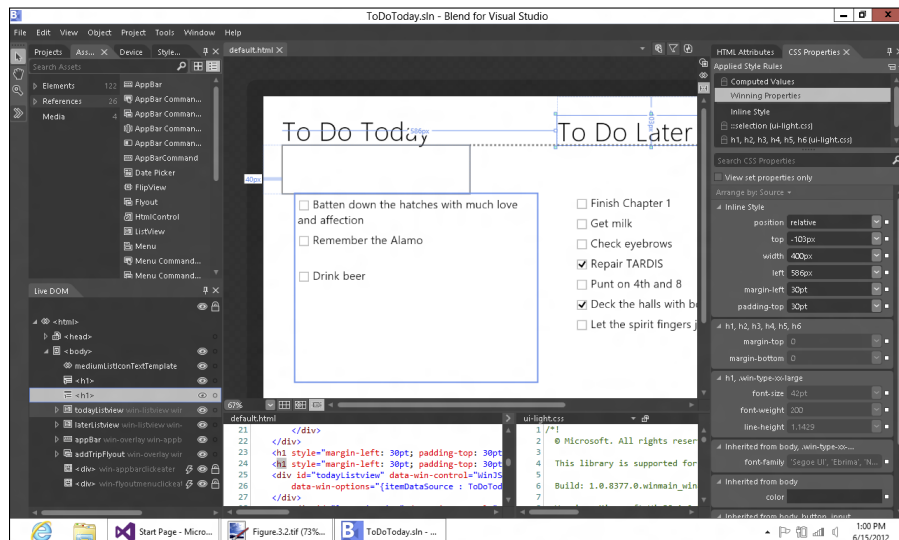
- ✓ First off, you can choose from a number of different programming languages in which to work. This book is about HTML and JavaScript, but XAML with C# or VB.NET are workable too. Additionally, you can use C++.
- ✓ The development environment compiles, packages, and deploys your app to the store automatically after it's configured.
- ✓ The controls included with the Windows 8 environment support both touch and mouse/keyboard. It's a slick arrangement.
- ✓ Deployment in several cultures is a snap with Windows 8. The localization configuration is baked into the deployment cycle; you just need to populate it with data.

Using the best developer tools on the planet

Aside from all that, for ten years, Microsoft has been working on the premier developer tool on Earth: Visual Studio. Not only do we get to use Visual Studio to develop, but we get a (relatively) new tool to design: Expression Blend.

Expression Blend (Figure 1-7) has been the XAML developer tool of choice for a while now. For Windows 8, it supports HTML/CSS as well. Blend gives designers a good environment to design the look and feel visually, without totally messing up the code underneath.

Figure 1-7:
Developing
with
Expression
Blend.



After you've developed the UI, you can easily hand the code off to the developer for the necessary JavaScript “back-end” coding that will make the app interact with Windows 8 and services and whatnot. The transition is painless, purposefully. Blend is designed to work this way. It is really a true one-two punch.

Folding in WinRT

ASP.NET developers have the .NET Framework backing them up. As Windows 8 developers, we have Windows Runtime. The .NET framework is a business-oriented API focused on getting data from one place to another. Windows Runtime, usually called WinRT, is more about using the features of the PC or tablet hardware to enhance the user experience.

- ✓ Nearly everything in WinRT is asynchronous, meaning the user interface won't stall waiting for results from a request. It was designed this way to smooth the user experience.
- ✓ You can use WinRT to access most everything the user has at his disposal: files, devices, network connections. The idea is to make a mashup of the features of the device and the abilities of the Internet.
- ✓ The Windows API is a big part of WinRT. Things that are hard to do in .NET (like graphics) are much easier in WinRT.

Chapter 2

Embracing Windows 8 Style

In This Chapter

- ▶ Falling in step with Microsoft's Windows 8 design style
- ▶ Applying the eight traits to your apps
- ▶ Helping your app get more user attention and love

Microsoft has come up with a list of eight traits of a Windows 8–style app. This chapter breaks that list into actual design principles that you can use. Implementation is what the rest of the book is about.

When I first saw the Microsoft list, I thought it was a little vague. After all, what *is* Windows 8 style? Isn't that what this whole thing is about? Oh, the humanity!

As it turns out, this list is actually pretty well thought out, as are the details behind each of the points. I suggest you take 30 minutes, read this chapter, absorb the details, and then go look at some of the apps I mention in the chapter. Then come back and read it again.

It's more important than you think to design well for your Windows 8 app. If you don't meet the guidelines, you won't get in the Store. If you don't get in the Store, you've wasted your time. You don't want to waste your time, so just take a deep breath and get ready to get Windows 8.

Without further ado, the eight traits a great Windows 8–style app should possess, as defined by Microsoft, are

- ✓ Windows 8 design style
- ✓ Fast and fluid
- ✓ Snap and scale beautifully
- ✓ Use the right contracts
- ✓ Invest in a great tile
- ✓ Feel connected and alive
- ✓ Roam to the cloud
- ✓ Embrace the Windows 8 design principles

Designing with Windows 8 Style

Isn't this whole chapter about designing with Windows 8 style? Well, yes and no. A handful of specific things that usually fall under the job description of designing are on this list, and the work is done for you.

Take typography, for instance. Usually we developers just take whatever default font Visual Studio is set for and use that for labels and select boxes. It's not always the best for the app we are building.

In Windows 8, however, the heavy lifting of selecting typography is taken from you. The best fonts, weights, and sizes are predefined for you. Follow the rules, and bang: You've got beautiful.

Content before chrome

The core of Windows 8 is that the content comes first. A user should see the content, not the app. In a user's mind, they don't run the Gallery; they look at pictures.

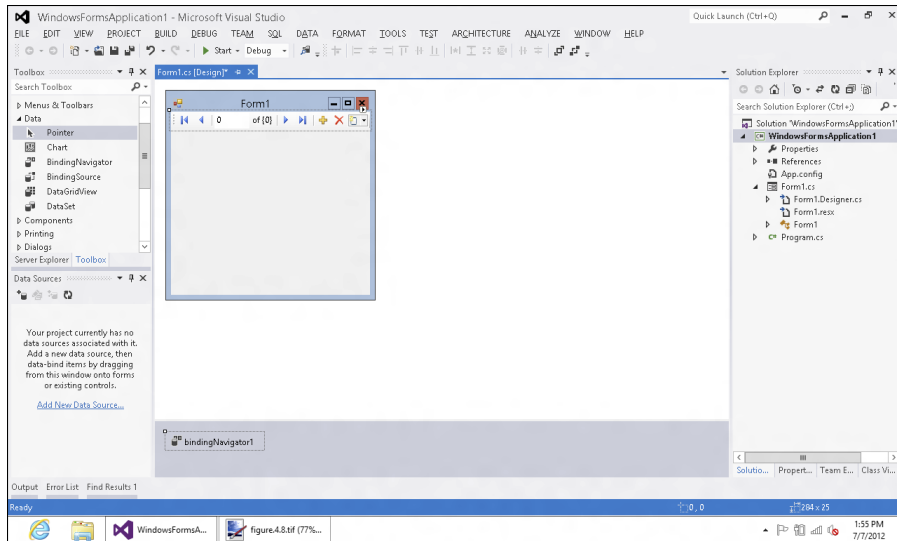
This is more than a philosophical difference. The content should come first not only in the user's mind but also in the layout of the app. Also, the user shouldn't ever feel like they are working with an app, nor have their eyes on more than one thing at a time.

You know what it reminds me of most of all? *Minority Report*. You know, that movie with Tom Cruise, based on the Philip K. Dick short story? In the movie, Cruise uses a computer with a monitor as thin and transparent as a windowpane, one without menus or any other obvious means of navigation. On this computer, Cruise can manipulate images, voices, data — everything — simply by gesturing and by speaking out loud. Now *that* is content first.

Content first

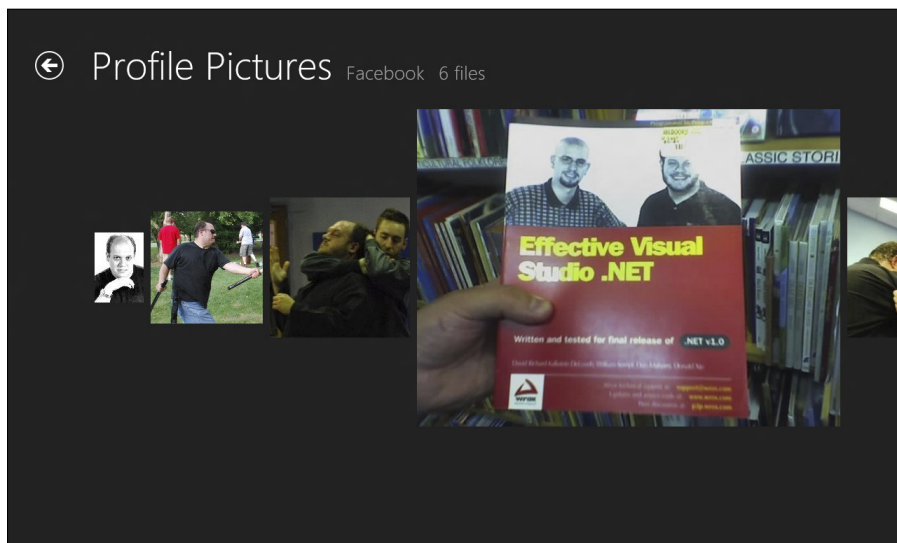
The best simple dichotomy that I can think of for “content first” is the old VCR control commands, like Play, Rewind, and Fast Forward, versus the new Photos app in Windows 8. When building a Windows Forms application, they were everywhere. You still, in fact, can drag a data entity onto a Windows Forms page in VS2012, as in Figure 2-1. You get VCR controls to go back or forward, first or last, add or delete.

Figure 2-1:
The VCR
control
commands
still live.



Open something like the Photos app in Windows 8. You just see a list of pictures, like Figure 2-2. Wanna see the detail? Click on it. Wanna go to the next picture? Swipe to the left. Or the right. Need to do something a little rarer? There's an app bar for that.

Figure 2-2:
The content-
driven
Photos app.



The idea here is that the user wants photos, they get photos. Not a bunch of stuff that uses photos. Just photos. The stuff that people love is what is on the screen, not the stuff that you use to manipulate the stuff that people love.

Reduce distractions

This focus on content has the added benefit of reducing the number of things that the user needs to look at. Figure 2-3 shows GIMP, a graphics tool favored by many digital artists.

The main screen has a lot of chrome on it. Chrome, window dressing, fluff: Whatever you want to call it, the GIMP main screen has a bunch. I find it terribly distracting to have everything sitting there, staring at me. I know the goal is just to have everything at your fingertips, but sheesh.

Contrast this with Autodesk SketchBook in Figure 2-4. Sketchbook is, in a word, beautiful. It has two pucks: one for color, one for width, and a “lagoon” of tools.

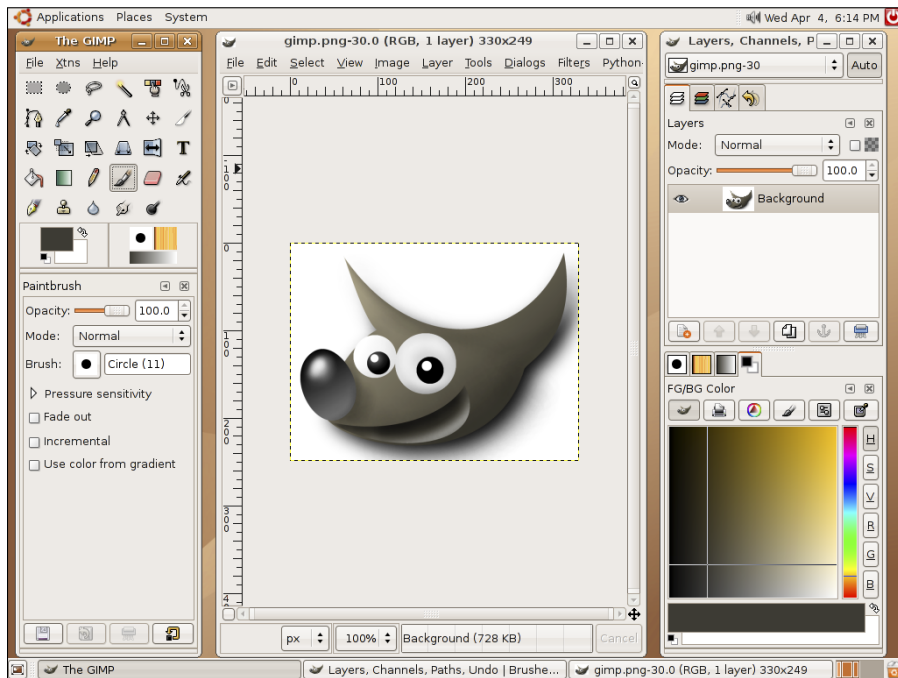


Figure 2-3:
GIMP's
distracting
main
screen.

Figure 2-4:
Autodesk
Sketch-
Book —
as non-
distracting
as it gets.



Even in an example like Figure 2-2, however, you might want to do something other than select a photo, go to the next photo, go to the last photo. What if you want to print it? Or use that photo as your profile picture? What if you want to add a photo or delete it? You need something else for more advanced tasks.

The app bar

The app bar holds application-specific functionality. When you swipe up from the bottom (or down from the top, strangely), a 90-pixel bar comes up from the bottom of the screen. This is where buttons for app-specific functionality go.

By convention, functionality that applies to the whole app goes on the left, and functionality that is specific to the page you are on goes on the right. The buttons should be an icon and text and do things that are particularly of interest to the user of the app.

I cover how to use the app bar in Chapter 6.

The Charms bar

The Charms bar is what holds Windows-level functionality. It has five buttons — the same in every app. They are

- ✓ Search
- ✓ Share
- ✓ Start
- ✓ Devices
- ✓ Settings

If your application is going to do something that should be handled by Windows, like printing, it will be found in the Charms bar. You don't add a print button to your app: You use the Devices button right there on the Charms bar.

Charms are integrated using Contracts. I cover how to choose contracts in *Choose the Right Contracts* below, and how to use them in Chapter 10.

Layout

Where you put the items on the screen is important to any app — Windows 8 or otherwise. There are a few placement suggestions in Windows 8, but most of the suggestions are rather meta in nature, if you know what I mean. Microsoft is more concerned about *how* you put things than where.

White space

Providing white space is probably the most important single layout consideration in Windows 8. Look back at Figure 2-2. See all the empty black around the pictures in the list? That's not just okay; it's recommended. Giving the content a frame of space emphasizes its importance.

Every developer can easily do two specific things to leverage white space right now, without even learning anything about user experience:

- ✓ Leave space around all content. Use margins, and make sure there is padding between significant elements.
- ✓ Remove lines and boxes from the organizational structure. Let the content organize itself. If the content looks disorganized without the lines and boxes, put less content on the screen.

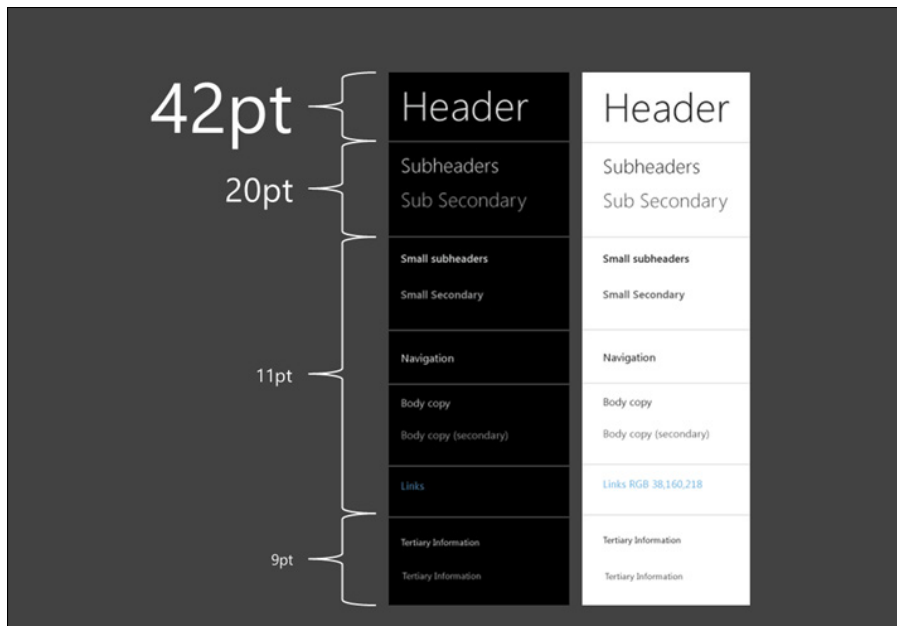
I cover how to improve your use of white space in Chapter 6.

Typography

If you let the content do the talking and don't have any boxes to tell the user where anything is, how exactly are you supposed to organize things? Typography. A big part of Windows 8 design is using header styles and text formatting to drive the user's interactions.

Fortunately, Microsoft has made it easy on the style-challenged, including your humble author. Not only have they provided a common font (Segoe), but there's also a style sheet that sets up the superstructure of the headers, navigation, and other styles like in Figure 2-5.

Figure 2-5:
The type
ramp.



The type ramp shows how a user's focus can be driven with a set of proportionately sized fonts that drive the user's attention appropriately.

The idea here is that users should be able to find their way through the content quickly by glancing at the screen and then hone in on that which is important to them.

For a good example of typography use, check out the News app on the Store. The headline in the lower left grabs the attention and the image provides breathing room. Off to the right, the smaller headlines of feature stories appear; users tap the story to be taken right to the content they care about.

Ergonomics

After the content is in place and the text is providing the structure, making sure the user can get around in the app becomes paramount. The content is the navigation, but it is also bigger than the stage. The user has to have a way to get to the rest of the content, even if it isn't on the screen they can see.

The content should flow easily when the user moves it, as if the screen were just an opening over a pool of water, and the content was floating on the water. To make it make as much sense as possible, it should only flow one direction — ideally, left to right.

Open the Store, for instance. You pan the content left to right, and the list view that the content is participating in supports this by making sure that if there is more content than fits in the window, it gets posted to the right.

I cover how to lay out data in Chapter 7.

Align on a grid

Windows 8 style includes a concept called the Windows 8 Silhouette. Shown in Figure 2-6 — and every app in the store, pretty much — it defines where the page title, group titles, and content should lay in the app in order to help the user make sense of the app.

There are two important things to know about the Windows 8 Silhouette. First, it is probably the most important part of layout that nobody talks about. It takes whitespace, typography, and ergonomics all into account, and really helps to give the app its Windows 8 feel.

Second, Microsoft has made it really easy for you to get started. The default templates, shown in detail in Chapter 4, all set up the Silhouette for you in order to help you get started. The default style sheet has a lot of elements as well. If you follow the guidelines and use the tools provided, making the app you are building more Windows 8–like is quite simple.

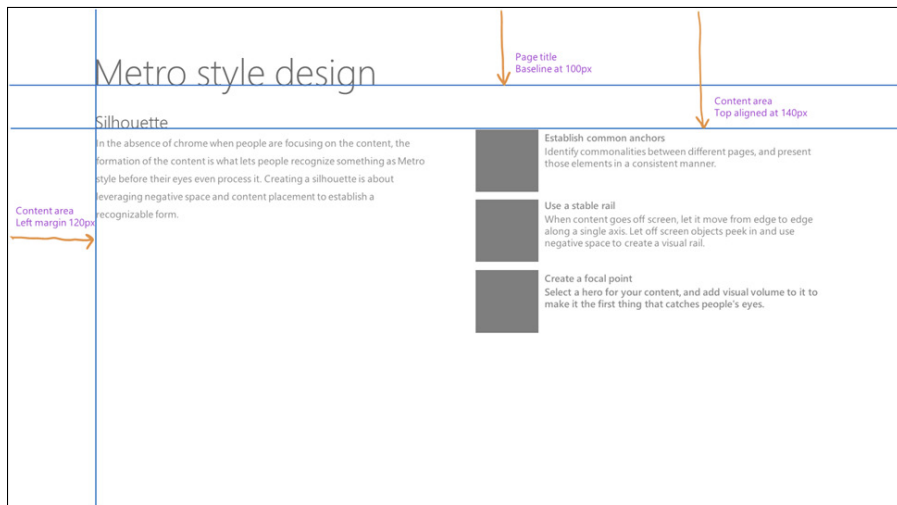


Figure 2-6:
An example
of the
Windows 8
Silhouette
from
Microsoft.

Interactions

How the content looks is just part of Windows 8 style. The way the user interacts with the app plays a big role too. The three places where you can expect to find interaction points are in the content, the app bar, and the Charms bar. The interaction should flow very naturally from the content, using the context of the content to guide the user rather than providing a bunch of menus and buttons.

The content provides the commands

Running a news app by clicking on the story headline is a pretty obvious use of using the content as a controller, but it isn't the only example. Almost all content can be used to give the user control over the command structure of the application.

Take a shooting game, for example. Putting a directional-pad-style controller on the screen is not the way you want to go. Just have the user touch on something to shoot it and, say, drag their ship around in order to move it.

Remember that you have a whole range of gestures to use too. For example, the user can swipe in one direction to throw, or drag their character to move it. You can use the pinch gesture to let the user focus in on something, and the rotate function changes the view of the camera.

The point here is to be creative with your commands. Take time to download a bunch of the games that have already been developed for the platform — or even for other platforms with a touch element — and gauge them for their Windows 8 commands. Is the content driving the commands? Should it be? Why or why not? Then take what you learned and put it in your own game.

A weather map is another good example. Design your app so that when a user taps on a space on the map, the radar refocuses to the weather station nearest that tap. If they drag the map around, just move it. If they pinch, zoom in. Following what others have done in any given genre is a good thing here. You don't have to reinvent the wheel.

Also important is the reduction of chrome that this philosophy brings. Don't provide a glyph if you don't need a glyph. Users don't need text on the screen that says "Touch here to move to this radar station." They will figure it out, really. Be a little Zen about the whole thing and don't worry.

Use the edge

In the "Content Before Chrome" section, earlier in this chapter, I went over the app bar and the Charms bar. Make sure your app supports these features, even if you don't use them fully. Try not to put content in the areas where the app bar and the Charms bar should appear.

What's even better is to use the app bar and Charms bar fully. Hiding any necessary chrome beyond the edge of the app really does put the content first and brings a new level of experience to the user.

One of the great examples of chrome that Microsoft would rather see go away is tabs. Internet Explorer is the best offhand example of an app that uses the app bar to get rid of tabs.

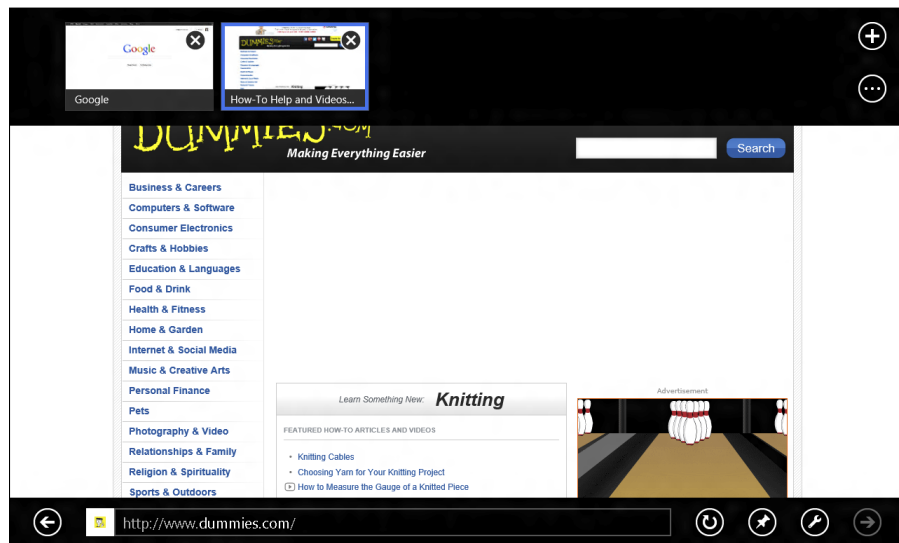
Use of the top app bar in Figure 2-7 completely gets rid of the tab bar that used to subtract 30 pixels of content from the IE window. There are two pieces of integrated chrome: a  glyph for adding a tab and an X glyph for deleting one. Otherwise, if you need to see a tab, touch it! That's putting the content to use.

Contextual commands

The app bar is the place for commands, but you can't just pile everything in there and expect users to find their way around. The app bar should take the context of the application into consideration when showing commands.

Figure 2-8 shows a bit of the growth of contextual command structures in the Windows world. First, Windows grayed out menu items when they weren't available. That at least prevented users from clicking on something that wouldn't work — for instance, you can't paste when there is nothing on your clipboard.

Figure 2-7:
The top
app bar in
Internet
Explorer.



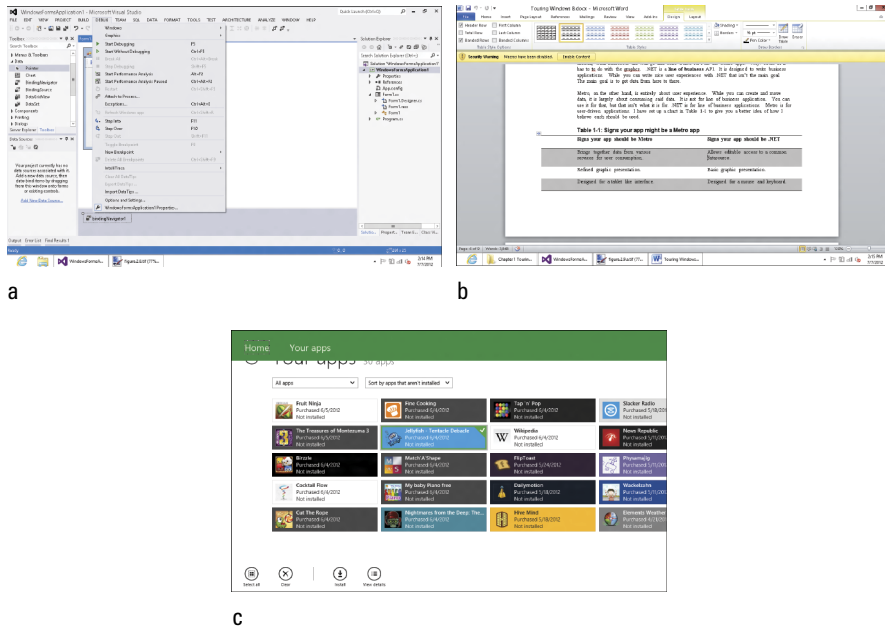


Figure 2-8:
The maturation
of contextual
commands.

With the invention of the ribbon, Microsoft pushed for even more contextual command structures. The Table ribbon isn't even available unless you are actually working with a table, and the same is true of the picture ribbon.

In Windows 8, Microsoft takes it to the next level — the glyph for Delete doesn't even show up on the app bar at all unless there is a deletable item selected on the page.

Navigation

Commands aren't the only piece of the puzzle. Some kinds of applications, like web browsers, need navigation. If you need to look at items in a collection, the application should have a structured way to go about that.

Fortunately, there's a pattern for that. Not unlike architectural patterns, *navigation patterns* give an app developer a structured way to put content together in such a way that it makes clear to the user how to get from one place to another.

Every screen should be about where you are in the content, not where you want to go. Where you are going should be straightforward, but the primary mission of the view should not be how to get you there.

The content provides the navigation, too

The content should provide navigation as best it can. Semantic navigation should make sense to the user: Clicking on the headline should take you to the article, and clicking on the picture ought to take you to a larger copy of the picture. Clicking a web address should bring up a browser — you get the idea.

The point is to make the application about the content, not about the functionality.

Avoid persistent navigation

Persistent navigation refers to the concept of keeping navigation tools active on the app even when they're no longer needed. In Figures 2-1 and 2-2 earlier in this chapter, I show an example of the VCR control buttons versus the current Pictures app. This is a great example. You don't want navigation hanging around on the page when you don't need it.

Another good example: the tabs in most contemporary web browsers. Tabs are a piece of navigation that you don't need around. The user can go get them when they need them.

The navigation-by-content idea focuses on being about where you are, not where you are going. Showing a Next button all the time is about focusing on where you are going. If the user wants to go ahead, they will. Instead of having a Next button permanently in place, provide a context-sensitive Next button on the app bar, go next when the user swipes left, or show the Next glyph when the user touches the screen. Don't focus on where you are going.

Use a navigation pattern

Two (actually pretty common) navigational patterns will help you meet the goals of Windows 8 Navigation. The flat pattern is designed for iteration through a list of items. The hierarchical pattern is good for drilling down through nested categories of items. Figure 2-9 gives you an overview of both.

The flat pattern is shown in the Picture viewer in Figure 2-2, or IE10 in Figure 2-6. It is a way to allow the user to iterate through the items in a list that aren't further categorized. Pictures are organized, you say? Yes, but when you are navigating through them, they have a set number and you look at them in a list.

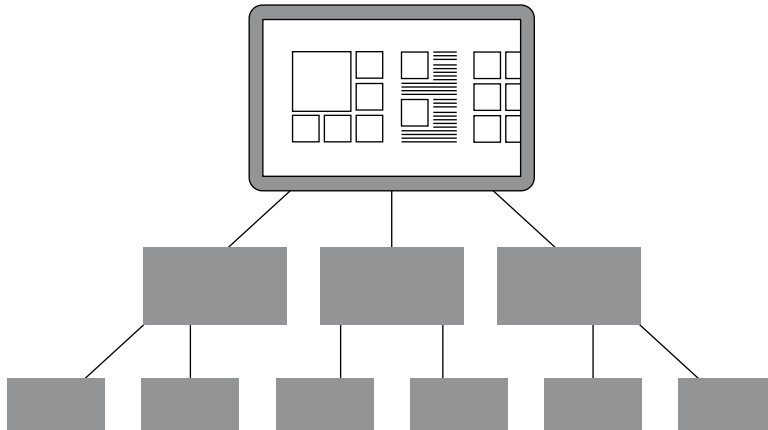
Hierarchical

Figure 2-9:
The two
navigational
patterns for
Windows 8.

Flat

The flat pattern is really good for apps with multiple items in a small, separately navigable set of sections. For instance

- ✓ Chat sessions
- ✓ Documents
- ✓ Tabs

The hierarchal pattern is better for large collections of content. The premise of a hierarchal pattern is a three-part navigation system heading from a hub page to a section page, and then to the detail. A few good example apps that use the hierarchal pattern include

- ✓ Start
- ✓ Store
- ✓ News
- ✓ Sports

Try Semantic Zoom

Navigating around a hierarchical pattern in Windows 8 can be rough, so Microsoft invented a new element and a whole programming model around it — Semantic Zoom. Semantic Zoom gives your users a way to navigate around groups of data as if they were data points themselves, in a very Windows 8 way.

To see what I mean, press the Start button and go to the Start screen. If you have a touchscreen, pinch the screen. See how the categories of items become items themselves? If not, check out Figure 2-10 — it's my Start screen in Semantic Zoom.

This makes group of items into manageable items themselves. To see what I mean, flick down any of the groups and open the app bar. The ability to name the groups is there, as a context command for the group selection.

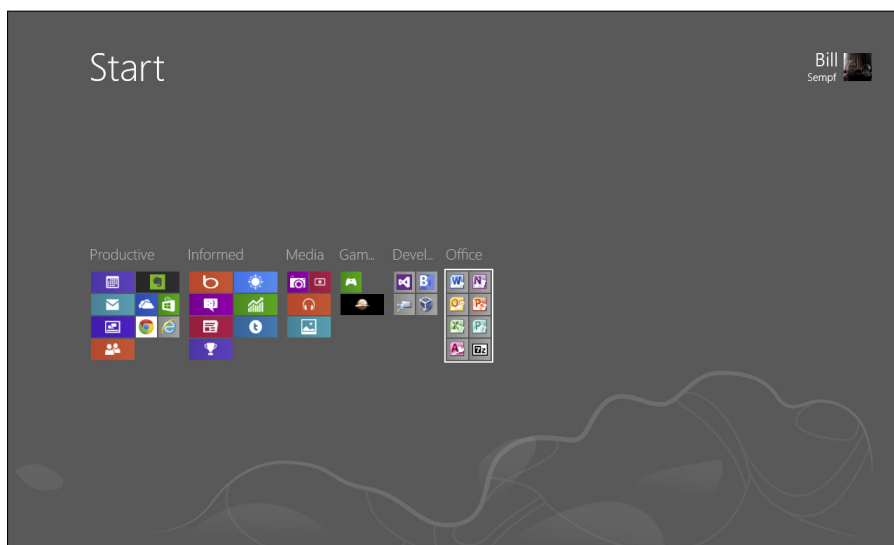


Figure 2-10:
Semantic
Zoom.

Keeping It Fast and Fluid

Animation is as much a part of the Windows 8 experience as the Windows 8 style. The animation in Windows 8 is about helping the user to understand what has happened and what is happening, as well as what will happen when the environment is interacted with in certain ways.

The animation library in Windows 8 gives the application developer an easy way to give users a consistent experience. Rather than being movement-focused, the animations are focused on certain features of Windows 8. You

don't just spin something around for the sake of spinning — you add context-specific movement to a particular command.

Use animation for a purpose

Motion brings life to Windows 8. The idea isn't to move things for the sake of moving them, but to provide a finished experience that appears very consistent and well-thought-out.

Because there is so much animation baked into the Windows 8 operating system, having motion in your application matches what the user sees elsewhere and builds confidence in the environment. Additionally, the use of animation in controls gives the user visual cues about what can be moved, how it can be moved, and how the app will respond when it moves there.

There are two ways to get animation into your app, and they are both covered in Chapter 6. The first is to use the built-in controls in the WinJS library. The second is to use the animations library in WinRT. Both provide the professional, consistent experience that the user wants to see from Windows 8 apps.

Design for touch first

The final piece of building a consistently fast and fluid environment for users is to design for touch first. A set of gestures is built into Windows 8 (Figure 2-11), and your apps should be designed around them.

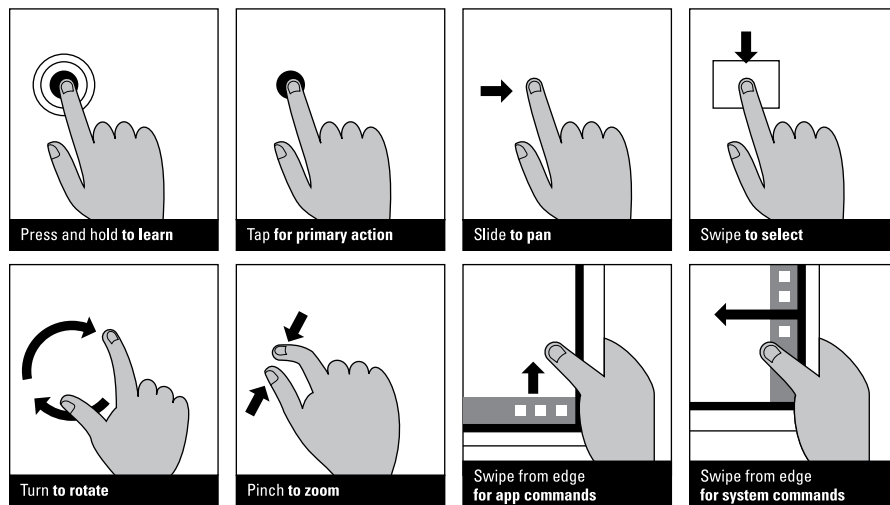


Figure 2-11:
The
Windows 8
Gesture
Library.

It's not enough to make sure the existing functionality in your app uses the right gesture. You need to think about what else your app might be able to do using gestures. Do you have check boxes on a variety of items in a list? Should you instead use swipe to select?

Using the built in gestures makes your app fit with the rest of the Windows 8 experience. The more comfortable the user is with your app, the better rating your apps gets and the more often the user uses your app.

Using the gesture library is covered in Chapter 6.

Snapping and Scaling Beautifully

I can't tell you much about how users will use your app, but I can tell you two things for sure:

- ✓ They use other apps, too.
- ✓ They use your app on multiple devices.

Some tools in WinRT can help you deal with scaling, but you need to think about a couple of design issues first.

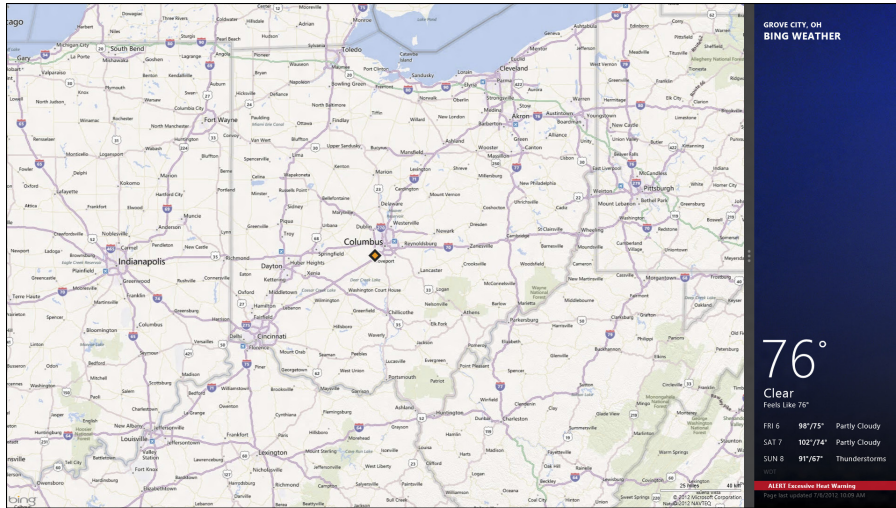
Design for multiple views

If their monitor is larger than 1366X768, users can snap your application to the left or right. In Figure 2-12, I have Weather snapped to the right while I look at Maps on the left.

Some apps look good snapped to 300 pixels wide. Yours might not — I'm sure mine don't. The best solution is to design for snap view and be ready for it.

Like the Weather app in Figure 2-12, a design for snap view isn't going to look like your normal app. You'll want to come up with a view of the app data that makes sense when it's being used with another app. Messenger, for instance, just shows the most current conversation. If you want to change conversations, you need to go to another view.

Figure 2-12:
Snap view.



Design for multiple screen sizing

The second thing I can tell you about your app is that it will be viewed on a number of displays. At the very least, you need to consider the baseline minimum of 1024X768, the common tablet size of 1366X768, and full HD, or 1900X1200.

It's like the old days of the Internet. You don't know exactly what size your users will have their browsers set at, so don't set dependent sizing. Use percentages in your style sheets. Don't expect the user to be able to reach way to the right — have them scroll there. You need to follow the same basic rules as a web browser.

I offer some tactics for dealing with screen sizing in Chapter 6.

Deal with pixel density

Have you tried to run Windows 7 on a full HD touchscreen? The menus are impossible to touch, even with a pen. They are just too small. The reason for this is that as resolution increases, pixel density increases. So the same 30-pixel high button in 1024X768 is tiny in 1920X1200.

To counteract this, Microsoft has three scales for images preset in WinRT. Text takes care of itself, thanks to fonts, but you'll need multiple versions of bitmap images if you don't want them to look too small at high resolutions.

The table below shows the three resolutions your app is most likely to be viewed at, and the percentages at which you should format your images for each. Save three copies of each of your images according to the scale percentages in the table. Name your bitmap images according to a naming scheme, and use the resource loader to make sure the right image is loaded in the right place. The resource loader will help make sure the right image is loaded at the right time. Use of the resource loader is covered in Chapter 6.

<i>Resolution</i>	<i>Scale Percentage</i>
1366X768	100%
1920X1080	140%
2560X1440	180%

Using the Right Contracts

Contracts are your application's way of communicating with the Charms bar, the operating system, and other apps installed on the user's device. They are a big, big deal. Understanding what to use when makes or breaks your app.

The goal is to improve the overall user experience by letting other apps fill out your user stories. Don't write common functions, such as printing, into your app — just hand it over to WinRT and let the Devices charm handle it. Rather than writing sharing into your app, let the sharing choices that the user selected do the work.

Implementing the contracts is covered in Chapter 10.

Share

Use the Share contract to give users the ability to share the content of an app, or consume content shared from other apps. Rather than guess at how the user wants to share content, just let WinRT get a list of the apps on a user's system that support sharing, and have them do the work.

Search

Thanks to the Search charm, users can search your app's content from anywhere in Windows. Doing this dramatically improves the usability of your

app, betters the app ecosystem in general, and will get you higher ratings in the market.

Search isn't the same thing as Find, however. Find (like CTRL+F in Internet Explorer) is for looking for some word or phrase on the current page of information. Search has a much broader scope in Windows 8, and refers to looking across the entire application for a list of items that contain the search term.

Exposing Search in your app is what allows people to find a task containing a search term when searching from the calendar. It makes the whole Windows 8 experience smaller and tighter.

Pickers

The Picker contract is the new common dialog of the Windows experience. No longer does a user have to search through folders to find files. They now search experiences to find content.

The file picker, for example, allows your app to be registered as a content provider so users can use the content provided by your app in other apps. Files and folders still exist, but the user experience is very different.

Settings

Whereas you used to have an Edit/Settings menu, now all settings for your app and the environment around it are under a common roof, in the Settings charm.

Your app controls the settings that are found in the Settings charm from the app, and in exchange, you let Windows provide the single hosting point for all the settings.

What kinds of things go in the Settings panel?

- ✓ Help
- ✓ About
- ✓ Log on/off
- ✓ Payment info
- ✓ Updates
- ✓ Preferences

Investing in a Great Tile

For more than 30 years, application developers have been making 32X32 pixel pictures to use as the icon for their apps. Trying to get the personality of an app across in less than the space of a postage stamp is something of a challenge.

In Windows 8, though, you get a 120X300 animated billboard with content from your app presented on it to advertise your wares. This is a *tile*.

Use the tile. It is the front door to your app, the storefront on the Start page.

Brand your tile

Tiles should be beautiful without being ostentatious. That can be a challenge for the design-impaired (like your humble author), but Microsoft has made the guidelines very straightforward.

You can look at Microsoft's advice on <http://msdn.microsoft.com/en-us/windows>. My advice? Stick to photos, glyphs, and text. You can't go wrong there. Find a licensed photo that is representative of your app (or take one yourself) and use it as the key image. Then, if you have text content to show, set up a template to support a representative glyph and the pertinent text.

Examples? Just look at the default apps that come with Windows 8. Launch every app so that the content refreshes. Attach your LinkedIn and Google account to the install so your People and Calendar and Messenger tiles come alive, and then work from there to decide how your tile will represent your app.

Drive traffic with secondary tiles

The tile isn't the only door into your app, either. Content from your app can be pinned (with the user's permission) using a secondary tile.

Writing a travel app? Let the user pin their upcoming trip to their start page. Easy access for them, and another way for users to remember your app. It's a win-win.

Chapter 9 covers tiles and pinning in detail.

Keeping Your App Connected and Alive

In the previous section, I suggested that tiles are animated billboards, and I wasn't kidding. Content sent to your tile while your app is active stays active even when your app isn't running, bringing a lot of user interest with it.

Your app can also set up subtle notifications that interact with the user when they aren't using the app — perfect for messaging apps or telling the user a long-running application is complete.

Of course, there are contracts and APIs for both live tiles and notifications; they are both covered in Chapter 9.

Bring your tile to life

Tiles can be populated with content that rotates in a subtle organized manner when the user is using her Start page. After an app is launched, one of the primary responsibilities is to update the tile so that when WinRT suspends the app, content stays fresh.

There are a number of approved templates for tile layout, which include long and short text in various sizes and Badges for simple image and numerical information. Before you decide how you want your tile to work, dig into Chapter 9 and find the big chart with all of the tile templates.

You can also opt to cycle through five updates, which can be anything that makes sense for your app:

- ✓ Messages
- ✓ Updates
- ✓ Stories
- ✓ Photos
- ✓ High scores
- ✓ Trips

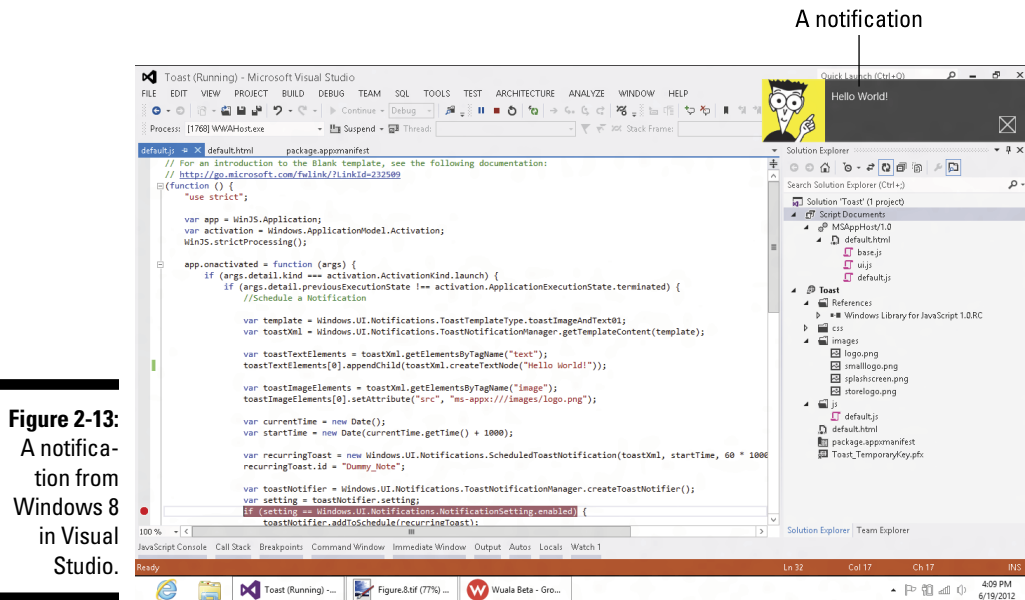
The idea is to make the app as much a part of the user's life as possible.

Use notifications

When the user is using another app, your app is suspended and doesn't have control of the screen, memory, network or any other resources of the system.

Your app can still feel alive, though, through use of notifications. Notifications are like the old dialog boxes you got through `MessageBox.show()` in .NET or `alert()` through a browser. They look a lot different though — see Figure 2-13.

Remember, the user can turn off notifications for your app (or the whole system) any time. Don't spam them.



Roaming to the Cloud

A big part of the connected experience is . . . well . . . being connected. Although your app should still function while it's not connected to the Internet (or at least gracefully fail), when it is connected, there are a lot of neat features of WinRT of which your app can take advantage.

Windows 8 apps can't use databases in the traditional sense. They are designed to use Web Services to save large data sets. I cover Web Services in Chapter 11, but there is a simple way to use WinRT to save smaller data sets.

Use Azure

Got more data than roaming settings can handle? I understand — I have that problem sometimes, too.

The integration between services built using the Azure cloud platform and Windows 8 apps is stunning. Windows Communication Foundation or the ASP.NET MVC web application programming interface can be used to build straightforward services. They can be simply and quickly deployed to Azure using Visual Studio, and then consumed via the

service references in the Windows 8 app. I cover using Azure Mobile Services to get services for your app in Chapter 15.

Using the big cloud services takes the management out of building apps. They aren't always perfect, but you don't have to worry about managing a server that is always Internet-connected in order to write something that has online storage. With the free small instances of Azure sites, you have a perfect opportunity to put larger online storage into your app.

Roaming settings are a part of Windows 8, and they use the SkyDrive to store smallish data sets for a given user and a given app. For instance, one of my Windows 8 apps stores the user's task list in the roaming settings as long as he is connected to the Internet.

The cool thing is that when the user does this, the settings are accessible from any machine that she is signed into. So if you set up your task list on your PC at home, and then sign into your Surface from the road, your task list is instantly accessible.

And what's the cost for this service? Nothing; it's free. It's part of the storage that every user gets as part of having a Microsoft account. Make use of it.

Chapter 3

Getting a Grip on Windows 8 Development

In This Chapter

- ▶ Comparing the ways to build apps in Windows 8
 - ▶ Touring the tools
 - ▶ Putting the users first in the Windows Store
-

The Windows 8 style is the basis for a lot of decision-making when it comes to applications for Windows 8, and Microsoft makes no bones about it. Unlike the battleship gray VB6/Windows Forms applications that, effectively, have no rules, Windows 8 apps seek to provide a certain user experience and have rules to enforce that. If you don't follow those rules, you don't get in the Store.

From Microsoft's perspective, Windows 8 is about the users *and* the developers. Focusing on the developers is nothing new, but focusing on the users is frankly a little new for Microsoft. They've always had the philosophy of "We'll take care of the developers, and they'll take care of the users." That didn't turn out so well, in hindsight.

The driver behind all of the new focus on the users was the development of the tablet, which Microsoft has been fiddling around with for years. Fact is, they did a pretty bad job. True, the hardware wasn't there yet, but Google and especially Apple swooped in and kicked Microsoft's behind. Taking their lumps, Microsoft has finally decided to make it about the users and the developers.

Focusing on the Users

Because focusing on the users is kind of new for Windows development, I'll start there.

Windows 8 enables a couple of core concepts very well, and you should learn them before you start building. In short, they are

- ✓ Immerse the user in your application.
- ✓ Provide no extraneous functionality.
- ✓ Design for touch first.
- ✓ Be prepared for multitasking.
- ✓ Remember that it is an interconnected world.

In the next few sections, I take each of these points one at a time and see how each impacts your design strategies.

Immerse the user

The biggest part of the Windows 8 application experience is taking advantage of the whole screen. This forces your hand as an application designer a little because there is no title bar, no exit button, no menus, and no status bar. You have no choice but to learn how to design applications a little differently.

Notice in Figure 3-1 that all of the chrome (that being the title bar and status bar and whatnot) in the new Control Panel is not visible. In the Windows world, you've been able to go full screen for a while now, but not like this. The chrome isn't hidden — it's just not there. At all. You have to find some other way.

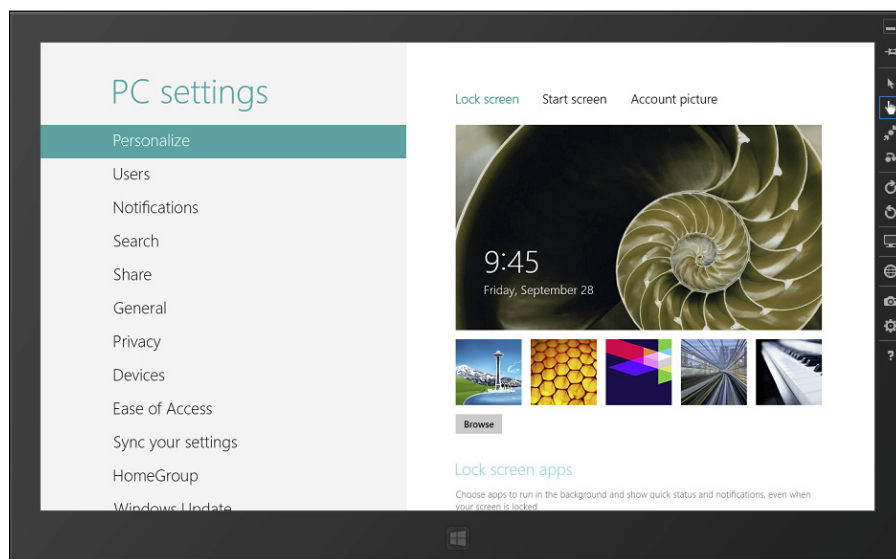


Figure 3-1:
Look, Ma.
No chrome!

Fortunately for you, you can find a ton of guidance and tools to help you to build apps that work in the Windows 8 style. The caveat is that you have to give up a little control and let the tools work for you rather than deciding that Microsoft is wrong about everything and rebuilding everything from scratch. At least give the style a chance.

Provide what the user needs

If there are no frills in the Windows 8 user interface, should the usability have no frills either?

In general, yes. Windows 8 applications are useful through and through. From the standpoint of requirements, no Windows 8 application should fill more than a few needs of a user — they are almost single-use applications.

Envision a chessboard. You can use it to play chess. That's what it does — engage the user fully in the chess-playing experience. Checkers? Well, okay, maybe. But that's it. No backgammon. No baseball. No Halo 3.

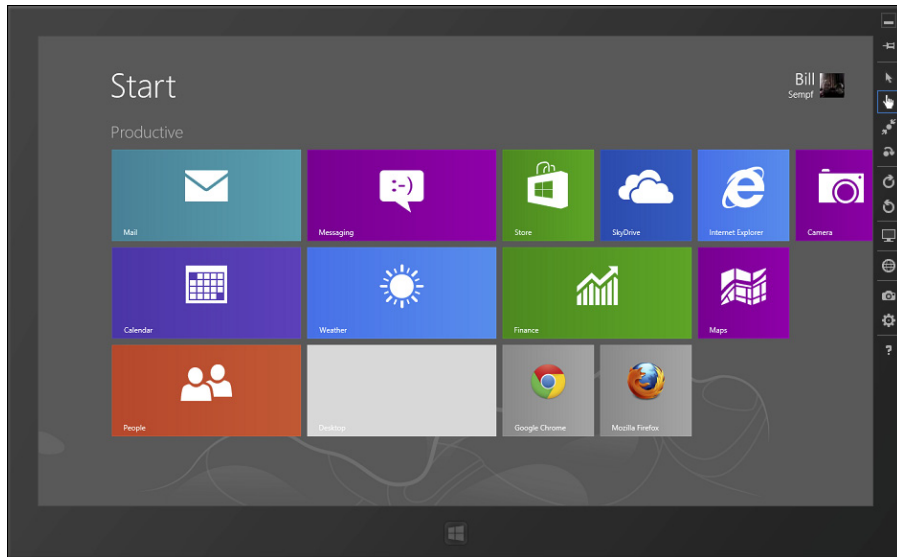
Windows 8 applications are a lot like that. They immediately immerse you in doing the task for which they've been created. A stocks application should launch directly into the overview chart. There is no menu. If you need information on a specific stock, tap on its chart line. A news application should show the news. The user should see pictures, headlines, story bits — things that directly relate to news.

Even the application icon, called a *tile*, can be part of that. Take the weather application in Figure 3-2, for example — the tile can host an image and text to show users the basic information they are after immediately upon starting the device.

Design for touch

Windows has been a mouse-and-keyboard thing since it started, and that isn't going to change anytime soon. Until we can whip things around Kinect-style, like Tom Cruise in *Minority Report*, keyboards and mice will be the main way to get data into computers.

Figure 3-2:
An active
application
icon, called
a tile.



But what about getting data *out* of computers? How is that different? Well, for one thing, the user doesn't have to type words and words into the machine — probably only a few here and there. Also, we can take advantage of newer technology — specifically miniaturization and inductive screens. *Miniaturization* gives us the ability to take the big PC and smoosh it down into a little package that the user can hold in her hand. *Inductive screens* give us the ability to move the mouse cursor with our finger rather than with a stylus, because everyone hates a stylus.

Windows XP was touch-ready according to Microsoft. It accepted electronic ink and had an onscreen (or soft) keyboard, and all that. Except it wasn't ready, even a little bit, at all. Ever tried to resize an app using a stylus? It's totally and completely impossible.

When you start to try and do things with your finger, it gets worse. Windows just wasn't designed for touch. All of the icons are tiny, scroll bars are just a few pixels wide, and on and on. It's a miserable touch experience.

Windows 8 fixes that. It is designed to be touch first, and the controls prove it. WinJS — the new JavaScript library provided by Microsoft — has a set of HTML controls pre-styled for touch. The touch-style input is handled for you. More importantly, though, the philosophy of application design is totally different. Touch is why the chrome is gone, why there are no menus, and why you interact with items rather than tell them how to behave.

Because of all of this, while you should be able to use a mouse and a keyboard to run a Windows 8 application, it needs to be designed with touch in mind first.

Enable multitasking

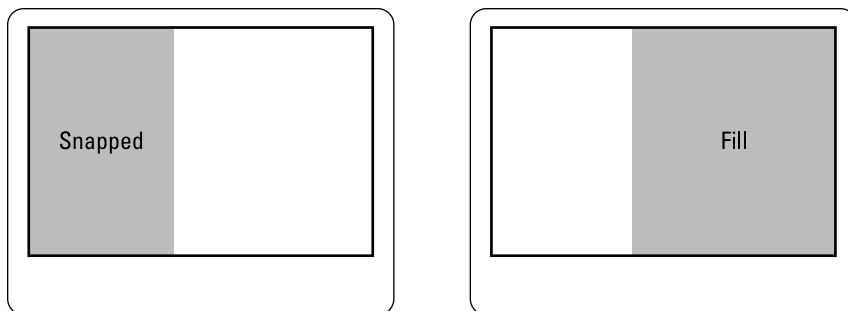
The people who use your application use other applications too. Take that for granted. Because all of the applications, by design, are single-use, more applications will be run at once.

This isn't a problem computer-wise because of the sandboxing that Windows 8 applications enjoy, but it can be a problem usability-wise. The user can't be expected to Alt+Tab all the time because, well, oftentimes, they won't have a keyboard!

A Windows 8 application can be in one of four different *view states*. Your Windows 8 application should be prepared to function in any of these states.

- ✓ **Full screen:** As you can imagine, *full screen* means the application occupies all of the screen real estate available except for a one-pixel border used to register edge swipes. This means full, as in no Start button, no clock, no system tray, nothing. Full screen means full.
- ✓ **Fill:** A two-thirds view for applications that have a snapped application on the screen. This is the majority stake for screen real estate sharing. As you can see in Figure 3-3, it leaves your application mostly square, so you need to account for the loss of real estate when you are setting up the layout. If you have a large detail with a list to the side, you might want to put the details on top, the list below, and allow scrolling.
- ✓ **Snapped:** A one-third view that is in contrast to the fill view. The idea is that a snapped and a fill view could work together as two different applications. With significantly reduced real estate, you need to plan your application to effectively show a navigation view in the meatier full screen view. For instance, multicolumn layouts should be compressed into a single column.
- ✓ **Suspended:** This isn't really a view. When your app is not in the foreground, Windows "suspends" it so it is no longer running. Even so, there are still ways that the suspended app can interact with the user, and that "view" of the app is something you need to take into consideration. Chapter 11 has a lot of detail about app suspension.

Figure 3-3:
Two of the
view states,
snapped
and fill.



I discuss implementation details for handling multiple views in Part II. Start thinking about it now, though, because this isn't just a factor of crushing up your content to fit into the smaller space. Microsoft recommends (and I agree) that you actually have a different layout for each view state. Start with the full screen view, and work down from there.

Keep in mind that this isn't just a for-fun thing. People use the view state feature to make work easier. For instance, if you are planning a trip, you can snap your itinerary to the right and the map to the left; or perhaps put the itinerary to the left and the airline reservation software to the right. All of these pieces need to work together and make a harmonious environment for getting the job done.

Assume connectivity

At some point, we need to assume network connectivity. That time is now. The Internet is nearly ubiquitous. When I was a little webmaster, back in the mid-'90s, a prescient CEO told me that someday the Internet would follow a pattern that would make *good* connectivity quite expensive and *mediocre* Internet connectivity free, for the masses. She was right. If you want a quality, high-speed connection you need to give your local cable company \$50 a month, or you can go to the coffee shop and share a narrow, low-speed connection with 20 other people for free.

Windows 8 UI apps are connected. They assume Internet connectivity, for better or worse. Anyone who has been developing for more than ten years is going to have a logical problem with that — I know that my 20 years of experience are *screaming* how wrong this is. But the fact is that most people are connected all of the time, and you shouldn't be afraid to assume it.

That being said, applications shouldn't fail if they are not connected to the Internet. Not working? That's fine. Show a message to the user that the app

isn't connected and perhaps offer politely to forward them to the Network Control Panel, but don't just let them outright fail.

But making an application that is expressly useless without network connectivity is okay in this day and age. It all needs to be balanced with the requirements for the application. For instance, Proton (the game I write in this book) is playable in solitaire mode, so you don't need to block the user if they don't have network connectivity. Now, they clearly can't do network play if there is no network, but they should be able to play solitaire.

Making an application that consumes news is another example. The application shouldn't fail if the network isn't available. Just show the last available update. Same with social networking applications. Just give the user last available information. That's only fair.

Taking Advantage of the Development Environment

From the looks of things, the users are taken care of by the Windows 8 UI. Now, the question is, how are the developers treated?

There is good news. The developer is still number one at Microsoft, and you can tell by the very high quality of tools. As developers, we have come to depend on Microsoft for providing the best tools in the industry, and that hasn't changed a bit.

Sticking with Visual Studio

Depending on where you come from, you might not know Visual Studio as the extremely powerful integrated development environment (or IDE) it has become over the last 14 years or so. Throughout the industry, even among those who dislike Microsoft products in general, Visual Studio is recognized as the gold standard for integrated development environments (IDEs).

Fortunately, Visual Studio is where Windows 8 development starts. It has been the center of Microsoft development for a while and it continues to be so. Through this book, much of the work is done in Visual Studio, including building the basic structure of the application, writing the JavaScript code to make it work, and integrating with outside libraries.

Picking a language

Within Visual Studio, however, there are choices to make. HTML5 is not the only platform for development. Microsoft has maintained ties with a number of other strong development communities in the Windows space.

Resurrecting .NET

When the first rumors of Windows 8 UI leaked, there was a lot of hand-wringing over the prospect that .NET might to be tossed out in favor of HTML5. Now, most savvy industry observers realized that this was very unlikely. .NET does very different things than HTML5, so it was pretty clear to most that both could enjoy a long and fruitful life in the .NET ecosystem. That is exactly what is happening.

So .NET, never on life support, has been resurrected. The logic for Windows 8 applications can be written using C#, VB.NET, or any other .NET language. You have a few new libraries and controls to get used to, but the core principles are there and aren't going away.

Additionally, there is a clear integration path between .NET libraries and HTML5. This means that your existing C# code need not go to waste. I have a Meetup.com .NET library that interacts with the API for Meetup, and I can use it without alteration in my HTML5 Windows 8 applications.

Coding a little C++

Another ecosystem within the Windows space that isn't as vocal as .NET is C++. The real hardcore applications (like Excel and Photoshop and the like) are written in C++. It is much closer to the operating system than .NET and gives very fine-grained control over the UI and device interactions.

It is possible — even recommended — to build Windows 8 application logic in C++. Games, especially, benefit from the large amount of existing code in the community and strong library support. C++ applications are likely faster than HTML5 or .NET, providing an outlet for very processor-heavy tasks. And, as with .NET, C++ libraries can be consumed by HTML5 Windows 8 applications, too.

Bringing on XAML

Whereas .NET and C++ provide a functional platform to build the logic of Windows 8 UI, at some point, we have to put the UI back in UI.



Here is an interesting nomenclature side note. When you say *HTML5*, you're referring to the layout power of HTML, the styling power of CSS, and the logic provided by JavaScript. There isn't a complimentary term for that kind of mashup in the Windows space. C# is a language, .NET is a library, and there are a number of different layout options such as ASP.NET and Windows Forms. Styling is sort of built-in. It's quite confusing, but look at it like this: .NET is another set of programming tools that you can use to code the back end of Windows 8 applications in a myriad of languages. In HTML5, you just use JavaScript.

But what about the UI for .NET? .NET and C++ both use Microsoft's own markup language for the UI, called XAML. XAML was the markup of choice for Silverlight and Windows Presentation Foundation — two of Microsoft's other UI design platforms. Windows 8 UI can be designed in XAML too.

I speak of Silverlight and XAML in past tense. It seems that the libraries for these two have been wrapped up in the .NET namespace that supports XAML. Details are sketchy at the time of this writing, but it looks like although those technologies will live on, their names might be changed, and not to protect the innocent.

Mentioning HTML5

In these pages, I'll be talking about HTML5, made up of HTML, CSS, and JavaScript. I more or less assume that you already have a passing familiarity with these technologies, so I don't need to drone on about them.



I do want to take a moment to talk about WinJS, though. WinJS is a JavaScript library, like jQuery, that has a collection of controls and functions that make it easier to write Windows 8 applications. Two of the main things that WinJS brings us are touch input and larger, prestyled controls that are tablet-appropriate. You'll see references to WinJS throughout the book because Windows 8 applications are pretty heavily dependent on the functionality it brings.

Touring Expression Blend

Expression Blend might be the logical successor to FrontPage, but you can barely see the resemblance these days. Blend is now a finely tuned design tool that even the most jaded user experience (UX) specialists agree is a joy to use.

To use Blend from Visual Studio, just right-click an HTML5 Windows 8 application and select "Open in Expression Blend" from the menu that appears. Blend becomes your designer at that point, and all of the changes you make in Blend are represented in the Visual Studio project.

Expression focuses on the User Interface (UI) elements of Windows 8. Visual Studio and Blend can both do lots of things, but some of them are obviously easier in one than the other. Blend is made to work with HTML and CSS. The Assets Panel and CSS properties give you the tools you need to set those parts of the applications up.

The Assets panel and artboard

The *Assets panel* is the palette where you store things you put on your canvas — the artboard. Elements of your Windows 8 application such as WinJS controls, HTML elements, images, and text reside on the Assets panel, where you can move them to artboard when you are ready to use them.

The *artboard* is your painting canvas. This represents the space where the application hangs out in when it's running. That doesn't mean that the application looks exactly the same as the artboard. The JavaScript can adjust how the finished application looks when it is running. The artboard gives you a place to start your elements and set important properties.

The CSS properties

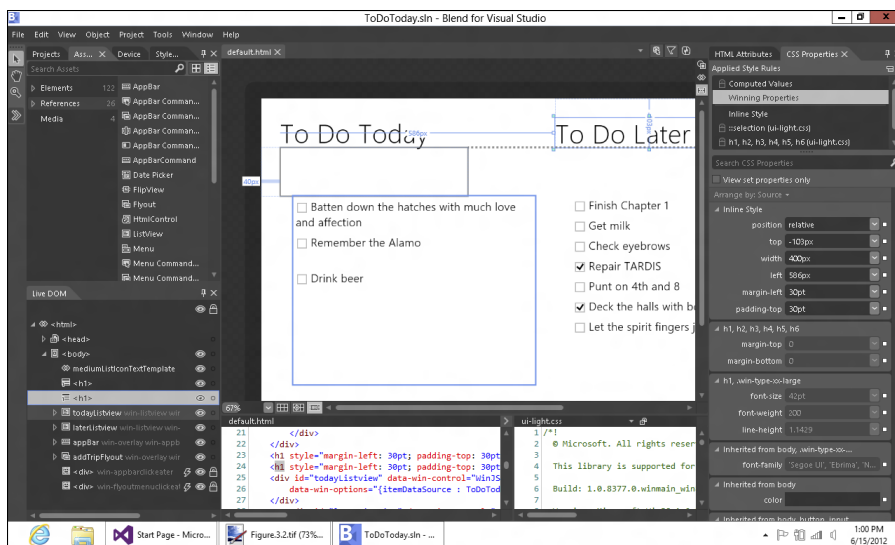
Speaking of properties, the properties of assets on the artboard are governed by a standard called Cascading Style Sheets (CSS). In Visual Studio, those properties are stored and managed in a code file, called a CSS file. They are denoted by .css and look like this:

```
body{  
    background-color: gray;}  
p {  
    color: blue; }  
h3{  
    color: white; }
```

In Blend, the styles look a lot more like properties, with a nice form to fill out the various elements. This is a lot easier for folks who don't spend every day looking at CSS because all of the important properties are in there, so you don't have to remember what they are called every time. Figure 3-4 shows how Blend lets you work with CSS.

In CSS, you can set rules globally for an entire element, for a specific style only (which might cross elements), or by a specific ID, which applies only to one instance of an asset. Confused? You aren't alone. This detail makes CSS a complex language to use, and that complexity isn't lost on the UI. It's fairly rough to build a tool panel that can support something as tough as CSS, and Microsoft did a great job with this one.

Figure 3-4:
Blend in
CSS.



In CSS, when you declare a style, the format is `name { };` with the specific properties of the tag you are styling in the curly braces. If you add a dot (.) before the name, you are specifying a particular instance of that tag. If you use a hash (#), you are creating a class name that you can use to style multiple kinds of tags.

In the panel, you select the element you want to style in the top section, be it a tag, class, or ID. Then you can set all of the properties in the area below, and Blend will fill out the area between the curly braces for you.

Handling JavaScript

Can you write JavaScript in Blend? Yeah, you can, but that work is better done in Visual Studio. Fortunately, the two are very well connected. If you open in Blend from the Solution Explorer, and then save and close in Blend to work on JavaScript, you'll find that all of your changes will be there, ready to see in debug mode or tweak in the code view.

There is something that we have really hoped for a lot in designers though: the active participation of the designer in the code. For instance, if you have code filling a dataset in the JavaScript and bind a container to the data using Blend, the data code will be executed *in the designer* by Blend, making it a lot easier to style the content.

The trick is to learn how best to use the two together. Sometimes, you'll need to edit the code. Other times, you'll want to use the designer. My usual pattern is to build the entire interface in Blend so that I know what data I will need, and then write the JavaScript to get the data with Visual Studio. Finally, I use JavaScript to hook them all together.

Making the Change from Windows Development

I figure this book has primarily two audiences. Most of the readers will be HTML5 web designers who want to give app development for Windows 8 a shot, or Windows developers who are used to building apps but need to learn the Windows 8 style.

If you are already a web designer who knows HTML5, you are probably going to be okay style-wise. If you are a Windows developer, you might need to get out of the battleship-gray mindset. Both parties will need to learn a few other things, too.

Learning the new SDKs

Two new significant libraries make up the majority of the Windows 8 style. I am not going to make a big deal about teaching the libraries; instead, you learn them organically as you go through this book.

WinRT is the Windows 8 library. It is not a replacement for .NET. If anything, it is a replacement for Win32. It is a tightly controlled, lightly structured library that is great for building tablet apps. It is part of the Windows 8 installation.

WinJS is the part of HTML5 that makes Windows 8 application development easy. HTML, CSS, and JavaScript help with touch-enabling the UI, making bigger, touchable controls. WinJS is included in each project.

Driving user confidence

Beyond style is a certain feel that Windows 8 apps have from the user's perspective. Specifically, users should want to use the apps. Ever open GIMP, the image processing software? Take a look at Figure 3-5. It's very powerful, but where do you start? You need a *For Dummies* book for the *For Dummies* book!

Windows 8 applications should be different. They are instantly usable. They are single purpose (or close to it). Nothing, or little, is hidden. The user should immediately feel confident using the software. Have a doubt about how easy your app is to use? Get a 6-year old to use it, or a 60-year old. If they pause, it isn't good enough.

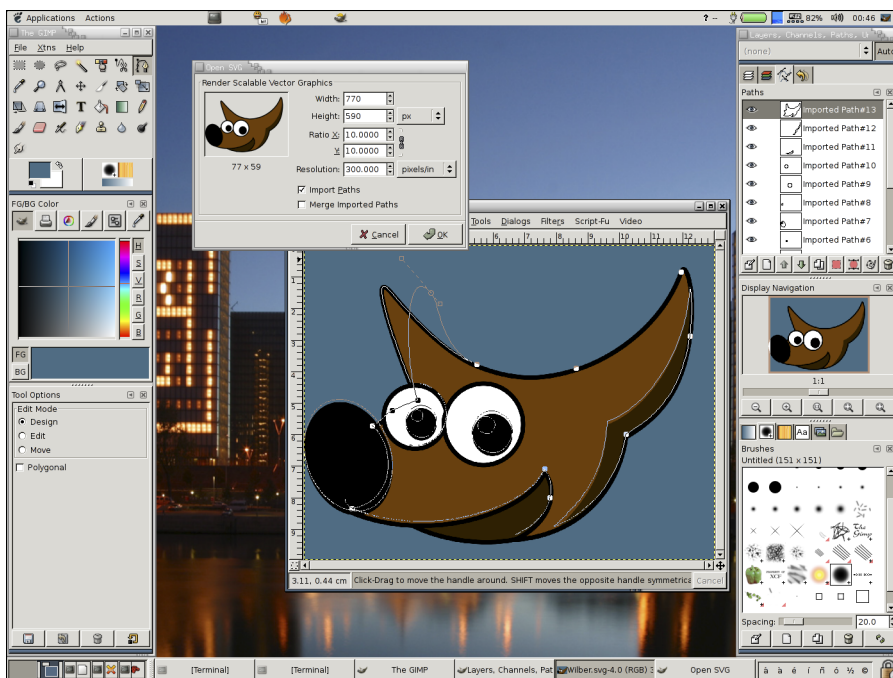


Figure 3-5:
The GIMP
interface.

Taking the next steps

To get started, you just need to get started. But I have a few suggestions of what *not* to do.

- ✓ **Don't worry about learning WinRT or WinJS out of the box.** Use examples (like from this book) and learn them organically. You can easily use `dev.windows.com/` to look up specifics as you need them.
- ✓ **Don't build Hello Universe.** Please don't make a list of requirements. Make one. Maybe two. I know the Windows-style Swiss Army Knife application is the norm, but that's not what Windows 8 is about. The app should do only one or two things, and it should do them well.
- ✓ **Don't do what everyone else is doing.** Don't build a map application or a tip calculator or a blog reader. Pick something you are interested in and design an application around it.

Now, *do* read the rest of this book and try some stuff. When you get done, you'll be ready to take on some serious Windows 8 applications.

Chapter 4

Setting Up a Windows 8 App

In This Chapter

- ▶ Starting with a new project
 - ▶ Using the templates for layout
 - ▶ Helping the user navigate
-

If you are an accomplished web programmer, a knowledgeable mobile developer, or an experienced desktop guru looking to program for Windows 8, I have bad news. A lot of your skills will require a little alteration. Developing in Windows 8 makes use of a lot of your existing skills, but it also has a few “gotchas.”

The goal of this chapter is to fill you in on the gotchas. There will be a quiz at the end of this chapter to see how you did. (Just kidding. I think.)

In this chapter, I walk you through an entire application — focusing on the structure of the Windows 8 template. This application won’t do anything, but that’s okay. I show you a lot of code that does neat stuff later on in this book.

Getting Started in Visual Studio

I gave you a glimpse of Visual Studio in Chapter 1, but that isn’t the only tool in our arsenal. A number of editions of Visual Studio are available for pay and for free. The various versions appeal to a number of different audiences.

Table 4-1 breaks down the different editions of Visual Studio.

Table 4-1 Visual Studio Editions	
<i>Name</i>	<i>Why You Would Use It</i>
Visual Studio 2012 Express Edition	It's free
Visual Studio 2012 Professional Edition	It's able to build all kinds of Microsoft applications, not just Windows 8
Visual Studio 2012 Premium Edition	Adds architecture, database, and test tools
Visual Studio 2012 Ultimate Edition	It has everything for the corporate programmer, including Team Foundation Server integration.

This book uses Express Edition. Express has everything you need to develop Windows Store apps except the developer key. Part IV covers all of the details of getting to the Store, including getting a developer key.

To get things rolling, launch Visual Studio (your choice of flavor for now), and click New Project right underneath the logo in the upper-left corner. This brings up the New Project dialog box.

Project types

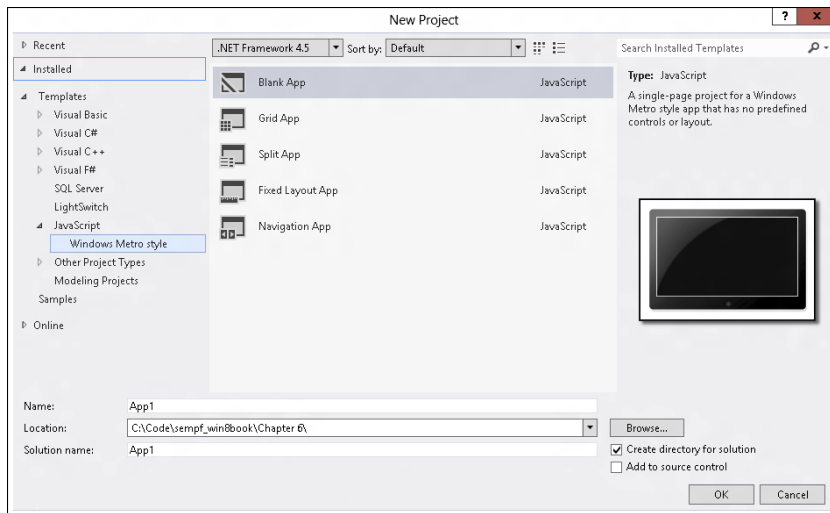
The New Project dialog box, shown in Figure 4-1, has four sections:

- ✓ **Recent:** Links to your recent projects.
- ✓ **Installed:** Contains samples and templates that either came installed or have been installed on this instance of Visual Studio.
- ✓ **Templates:** Handles the built-in project types. The Samples subheading has the sample code that has been installed locally.
- ✓ **Online:** Online access to templates and samples from the MSDN and the community.

The Online tab lets you download samples and templates from Microsoft. Once you get them, they will appear in the Installed section. If you need something you don't have, look for it first in this Online section so you don't have to waste time searching different online sources.

The Templates folder in the Installed section is where you are going to spend a lot of time. All of the languages you have installed will have a folder here (along with a few others). As of this writing, you are looking at JavaScript, C++, C#, and VB.NET.

Figure 4-1:
The New
Project dia-
log box.



Some things can only be done in certain languages. You can only write a WinRT Component DLL in C++. Unit testing doesn't fly in JavaScript. Navigation applications (essentially websites) are only done in JavaScript.

Making a new project

This book is about HTML5 apps. Still, it is important to know what the other Windows 8 templates can do. You never know when you will need to build a tool in a particular platform to meet a particular requirement.

Templates in .NET languages

The good news is that you can build things in .NET. The bad news is that you are still constrained with the Windows 8 rules for apps (which I cover over the course of this book). If you are coming from the .NET world, you will find that there are things that you can't do in Windows 8. For instance, Windows Store apps don't offer database access. More about that in Chapters 12 and 15.

There are several categories of Windows 8 templates in the New Project dialog (Figure 4-2) for the .NET languages. Three of the categories are layout-related, two are structural, and one is for testing.

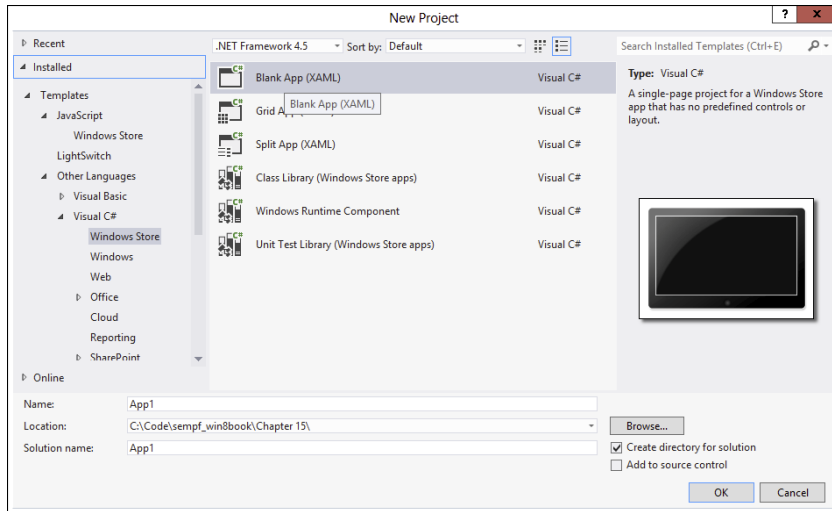


Figure 4-2:
.NET project
templates.

The Blank App, Split App (XAML), and Grid App (XAML) templates define layouts that are recommended for Windows 8 apps. I cover those in “Building Different Layouts,” later in this chapter. The Class Library and Windows Runtime Components project sets up a WinRT dynamic linking library (DLL), which is a portable collection of compiled code that is consumable by a variety of different applications. It is useful for code reuse. The unit test project is just what it sounds like — a way to make sure your program does what you tell it to do.

Working in C++

C++ is an interesting addition to the Windows 8 world. If you click on the tab for C++, you find a list that looks remarkably like the list for the .NET languages. The layout projects are the same, for instance, which makes sense because they use XAML for the markup language.

C++ also features the WinRT Component DLL. This is more or less the same as a Class Library, except it is specifically designed to extend the WinRT library that is the basis of the Windows 8 world. The .NET Class Library is more business-logic oriented.

I can’t imagine that most people building HTML-style applications need WinRT extensions. I did once work on a project that used some Windows internals features, and the WinRT Component DLL did come in very handy.

So why is C++ in the fold? The last project in the list — DirectX Applications — is the reason. C++ is the graphics language of Windows. If you’re building vector-drawn polygon-based games or applications, you want DirectX. C++ is the way to get to DirectX. Mapping applications, 3D games, and sophisticated user interfaces are all usually written in C++, using DirectX.



This book isn't about that. This book is about HTML, and it uses bitmaps. You can get by without DirectX.

JavaScript templates

Now the interesting stuff. HTML5 applications are all about layout. Aside from the remarkably useful Blank App, you have four layouts that are the driving focus behind most information-driven Windows 8 applications.

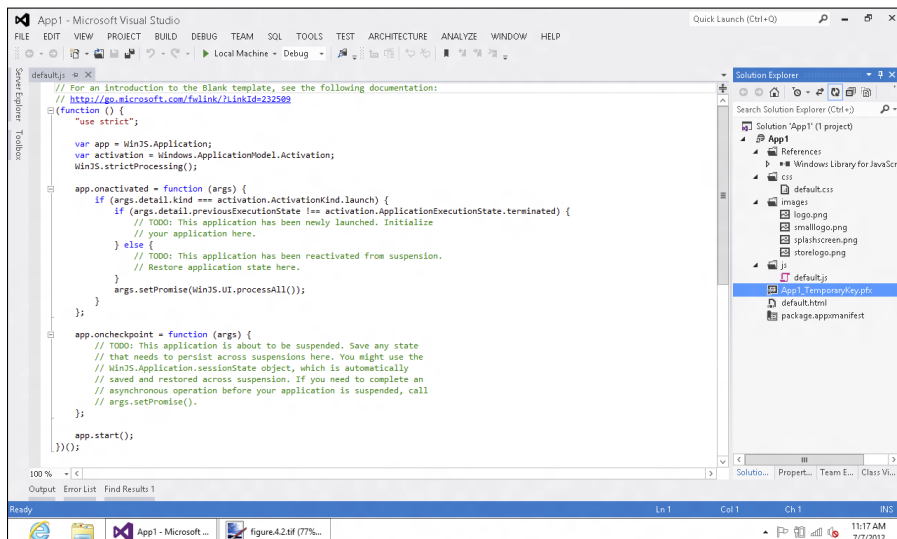
I cover the layout details in “Building Different Layouts,” later in this chapter, but for now, recognize that Fixed Layout, Grid, and Split are the three core navigational systems that Windows Store apps are supposed to use, according to the guidelines in Chapter 2.

Solution layout

In the misnamed JavaScript project (it really should be HTML5, but there you go), make a new Blank App project. Don't worry about the name — you'll toss it when you're done anyway.

Take a look at Figure 4-3. At the top are the usual project details. If you are unfamiliar with Visual Studio, you might not know about References, in the upper right of Figure 4-3. References is a virtual folder that holds the links to library files used by the project. If you need a special method or property, and it is available from a special DLL, the name and path to the DLL are found here.

Figure 4-3:
The core
JavaScript
project
layout.



Windows 8 applications in the HTML5 space are unsurprisingly like websites. The CSS folder keeps the style sheet files in one place. Unfortunately, that isn't always the case.



When you add a new file to a Windows 8 application, Visual Studio drops a CSS file, an HTML file, and a JavaScript file all in the same place. Generally speaking, this is a bad idea. You want an environment where developers know where files are stored. If you want the CSS files in the CSS folders, you have to move them and change references.

The Images folder holds bitmapped assets, like buttons and logos and what-not. The JS folder holds the JavaScript files and has the same problem as the CSS folder — files have to be moved as you create them.

The References folder is interesting. It holds the library that makes all the magic happen at the user-interface level. It is copied from project to project. That seems like an awful idea, but then you get to thinking about versioning. From version to version of the developer tools, a lot changes. The library won't change with it in this layout, and that's a good thing.

After WinJS, you see two files, the default HTML file and the application manifest (appxmanifest), shown in Figure 4-4. The default HTML file is where everything starts. The application manifest is the Project settings file — and it has a big impact on the app.

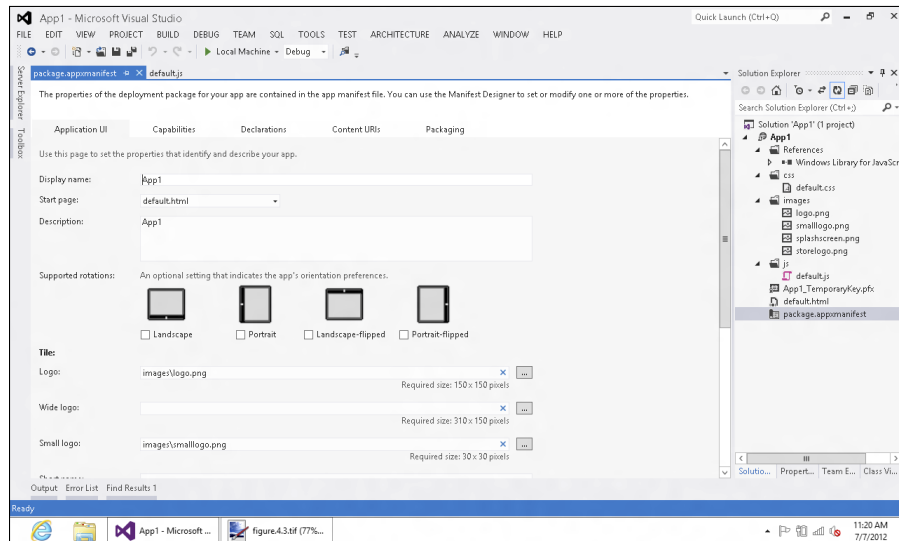


Figure 4-4:
The
application
manifest.

The application manifest contains all of the settings that are central to the application, like package information and the icons logo file. One of the most important details this manifest holds is the capabilities list. This is the list of things your application can do in the Windows 8 space. Each of the default templates include Internet Access in their declarations, but you have to check off anything else you are going to use, and the user will be told about them upon installation.

Building Different Layouts

HTML5 applications in the Windows 8 space are particularly good at displaying structured data — just like web pages. The three layouts provided in the templates I looked at in the “Making a new project” section earlier in this chapter are the go-to looks for Windows 8 applications, closely following the Windows 8 Style Guide.

Windows 8 isn't just some random name. It is a design language that Microsoft put together, first seen in Windows Phone 7 and Xbox user interfaces. Microsoft says this about it: “Windows 8 is our design language. We call it Windows 8 because it is modern and clean. It's fast and in motion. It's about content and typography. And it's entirely authentic.” The templates keep your application in the Windows 8 style.

Grid layout

Grid layout is the simplest master/detail layout. Items in a collection are shown in a grid, and clicking one of them changes the screen to a detail view. It's a classic and very useful way to show products, movies, stories, or anything else with an item/detail hierarchy.

From the New Project dialog, make a new Grid layout application. Visual Studio creates a set of default pages for you that actually make a pretty nice-looking Windows 8 app out of the box. The data is all fake, but close counts, right?

Press F5 on your keyboard and see what I mean. Depending on what hardware platform you are running, the Windows 8 interface comes up and you will have to Alt+Tab or sweep back to Visual Studio and stop debugging to get things back to Desktop mode.

The Grid layout is for an ontological structure that has items with pictures and a lot of detail data. Think about it. This is basically Netflix or Amazon or any other rich list/detail website.

Split layout

The Split layout is a little more complex programmatically but may be a little nicer for the user. Set up a new project and see what it can do.

1. Click **File** → **New Project**.
2. Select the **Grid Layout** template from the **JavaScript Windows Store Application** section.
3. Change the **Name of the project** to **Chapter4Split**.
Your final selections should look like Figure 4-5.
4. Click **OK**.
5. Press **F5** and check out the demo project. I've put a version in **Figure 4-6**, too.

The Split template is more usable for average data in the real world. It has a little less space for lists and details, but that's okay. Honestly, most sites don't usually have that much data to look at anyway. Also, the list-based navigation on the left of Figure 4-6 is a lot less dependent on the image data on the right side. If there isn't an image as part of the data, you can still make a nice-looking presentation.

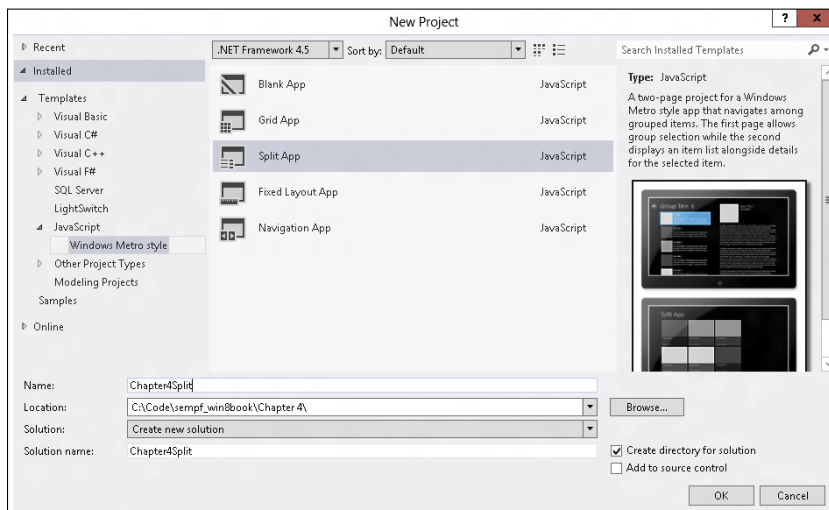


Figure 4-5:
The new
Split project.

Chapter4Split

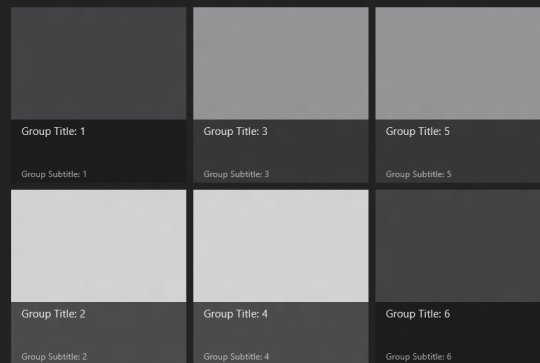


Figure 4-6:
The sample
Split
template.

In this section, I analyze the code from the template and see how all of this comes together.

Now things get tricky. The templates aren't always the simplest implementation. Take a look at `default.html`, and in the `body`, you find this:

```
<body data-homePage="/html/splitPage.html">
  <div id="contentHost"></div>
  <div id="appbar" data-win-control="WinJS.UI.AppBar" aria-label="Command Bar"
    data-win-options="{position:'bottom', transient:true, autoHide:0,
      lightDismiss:false}">
    <div class="win-left">
      <button id="home" class="win-command">
        <span class="win-commandicon win-large">&#xE10F;</span><span
          class="win-label">Home</span>
        </button>
      </div>
    </div>
  </body>
```

Checking out that `data-homePage` attribute? Yeah, so am I. That's . . . unusual. Microsoft is using the Windows 8 databinding to set the main page of the application.

Do I like this? No. Am I willing to work with it? For now, yes. This is hardly a best practice; nonetheless, it does work. Microsoft sets the home page URL in the default.js file.

```
WinJS.Application.onmainwindowactivated = function (e) {
    if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.
        launch) {
        homePage = document.body.getAttribute('data-homePage');

        document.body.addEventListener('keyup', function (e) {
            if (e.altKey) {
                if (e.keyCode === WinJS.Utilities.Key.leftArrow) {
                    WinJS.Navigation.back();
                }
                else if (e.keyCode === WinJS.Utilities.Key.rightArrow) {
                    WinJS.Navigation.forward();
                }
            }
        }, false);

        WinJS.UI.process(document.getElementById('appbar'))
            .then(function () {
                document.getElementById('home').addEventListener('click',
                    navigateHome, false);
            });

        WinJS.Navigation.navigate(homePage);
    }
}
```

The last runnable line of code is what matters here. It looks up the split page for the navigation, and that is where the important logic is.

The split layout is accomplished with the `section HTML5` tag. The `section` subdivides a `div`, and that's exactly what you're doing here. There is a `section` for the list and a `section` for the detail.

```
<section class="itemListSection">
    <div class="itemList"
        data-win-control="WinJS.UI.ListView"
        data-win-options="{dataSource: splitPage.items, oniteminvoked:
            splitPage.itemInvoked, layout: {type: WinJS.UI.ListLayout},
            selectionMode: 'none' }"></div>
</section>
<section class="articleSection" aria-live="assertive" aria-atomic="true" aria-
    label="Item detail">
    <header class="header">
```

```

<div class="image" data-win-bind="style.backgroundColor: backgroundColor;
    style.backgroundImage: backgroundImage; alt: title"></div>
<div class="text">
  <h1 class="title win-contentTitle" data-win-bind="textContent:
    title"></h1>
  <h2 class="subtitle win-itemText" data-win-bind="textContent:
    subtitle"></h2>
  <p class="description" data-win-bind="textContent: description"></p>
</div>
</header>
<article class="content" data-win-bind="innerHTML: content"></article>
</section>

```

The first section, `itemListSection`, has an `itemList` control in it that references the `WinJS.Binding.Template` above `itemTemplate`. This is a common pattern that you should emulate whenever possible.

```

<div class="itemTemplate" data-win-control="WinJS.Binding.Template">
  <div class="largeIconTextTemplate">
    <div class="largeIconTextTemplateImage" data-win-bind="style.
      backgroundColor: backgroundColor"></div>
    <div class="largeIconTextTemplateBackground">
      <div class="largeIconTextTemplateLargeText win-itemTextStrong" data-
        win-bind="textContent: title"></div>
      <div class="largeIconTextTemplateSmallText win-itemTextTertiary"
        data-win-bind="textContent: subtitle"></div>
      <div class="largeIconTextTemplateMediumText win-itemText" data-win-
        bind="textContent: description"></div>
    </div>
  </div>
</div>

```

This is the new databinding process for Windows 8. See the `data-win-control` tag in the `div`? That is the definition for a Windows 8 control. In the section code, you can find `data-win-bind`. That and the `data-win-options` are the property-setting mechanisms. This is the new way to get information into a UI in Windows Store apps.

I go over databinding in Chapter 7. For now, know that the layout is handled with the new `section` tag, and a separate binding template defines the layout of the repeating information.

Open `splitPage.js` to see how the navigation gets populated. The `fragment.load` event grabs namespace data that feeds the `navigate` function.

```
WinJS.Application.onmainwindowactivated = function (e) {
    if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.
        launch) {
        homePage = document.body.getAttribute('data-homePage');

        document.body.addEventListener('keyup', function (e) {
            if (e.altKey) {
                if (e.keyCode === WinJS.Utilities.Key.leftArrow) {
                    WinJS.Navigation.back();
                }
                else if (e.keyCode === WinJS.Utilities.Key.rightArrow) {
                    WinJS.Navigation.forward();
                }
            }
        }, false);

        WinJS.UI.process(document.getElementById('appbar'))
            .then(function () {
                document.getElementById('home').addEventListener('click',
                    navigateHome, false);
            });

        WinJS.Navigation.navigate(homePage);
    }
}
```

Fixed layout

Fixed layout is the simple option. It uses `ViewBox`, a WinJS control that helps with navigation in a hierarchical model.

```
<body>
  <div data-win-control="WinJS.UI.ViewBox">
    <div class="fixed-layout">
      <p>Content goes here</p>
    </div>
  </div>
</body>
```

Getting Around

A lot of Windows 8 apps work like websites: You have a home page and move among content pages. This is okay: In fact, it's good. If the content is dynamic, this method brings a lot of value — that's why the World Wide Web is so popular.

The normal HTTP request-response of the web isn't required in Windows 8, however, so you have a chance to tweak the model a bit. Microsoft has altered the HTTP model a bit with the Navigation library in WinJS.

Navigation

The Navigation library is designed to make fragment navigation easier in Windows 8 development. Fragment navigation is the use of small bits of HTML content injected into a main HTML page. JavaScript gives us the tools to move that HTML around in the application and give the user the appearance of changing pages.



If you have been developing websites for a while, you might remember the `iframe`. The `div` didn't always have the power it does now. We used to use the `iframe`, which actually spawned a new browser instance within the browser. Needless to say, back in the old days (the early 1990s), this used to cause all types of security and performance problems. These days, browsers handle `iframes` better, and you can use `divs`, which are better.

Take a look at a static example before you dig into code. In Figure 4-7, you see a main page with a content area. The content area — speaking code-wise — is a `div` tag. The idea is to load the content “fragments” on demand and show them to the user.

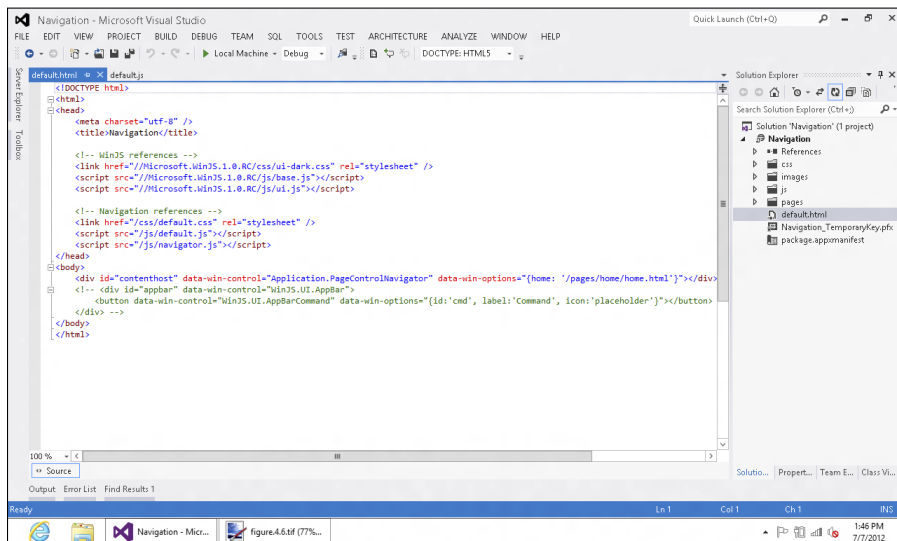


Figure 4-7:
The navigation system.

Something in the code needs to keep track of the state of the navigation. The app needs to remember which fragment it is on, and what happened before (in case the user presses the back button). Also, depending on the application, the app has to keep the *next* page in mind. This takes a little effort.

Loading fragments

Fortunately, WinJS gives us the tools you need to make navigation work with the Windows 8 constraints. The `WinJS.Navigation` namespace has the tools you need to get things done.

Better yet, Visual Studio has a navigation template. If you click File⇨New Project and dig into the Windows 8 JavaScript apps, you'll see the Navigation Application. It is described as a “minimal style application that uses Windows’ Windows 8–style frameworks and includes navigational support.”

A look at the source shows us that the Navigation template is a pretty contemporary divided-screen model. It has `div` tags that contain the content and another `div` that has the navigation tools.

```
<body data-homePage="/html/homePage.html">
  <div id="contentHost"></div>
  <div id="appbar" data-win-control="WinJS.UI.AppBar" aria-label="Command Bar"
    data-win-options="{position:'bottom', transient:true, autoHide:0,
      lightDismiss:false}">
    <div class="win-left">
      <button id="home" class="win-command">
        <span class="win-commandicon win-large">&#xE10F;</span>
        <span class="win-label">Home</span>
      </button>
    </div>
  </div>
</body>
```

The `div` labeled `contentHost` is used to store the content that is directed there. The app bar `div` is of more interest. It has a `data-win-control` property, which is a Windows 8–specific field that includes specific styles and functionality at render time.

This `data-win-control` is the `WinJS.UI.AppBar`, which is the little sliding bar that you can access in a Windows 8 app by sweeping from the bottom. There is a one-pixel trigger left in the bottom of all applications that makes the sweep work, and the `div` shown here sets up the content for the bar.

That app bar is supposed to give you the ability to load new fragments into `contentHost`. You can do this with direct buttons, page numbers, or VCR-style controls. It doesn't matter, as long as the content gets loaded.

Loading the fragments is set up in `default.js`. Remember that weird `data-homepage` attribute in the `body` tag of the default page? Well, that's used to set the initial page in `contentHost`:

```
WinJS.Application.onmainwindowactivated = function (e) {
    if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.
        launch) {
        launch() {
            homePage = document.body.getAttribute('data-homePage');

            document.body.addEventListener('keyup', function (e) {
                if (e.altKey) {
                    if (e.keyCode === WinJS.Utilities.Key.leftArrow) {
                        WinJS.Navigation.back();
                    }
                    else if (e.keyCode === WinJS.Utilities.Key.rightArrow) {
                        WinJS.Navigation.forward();
                    }
                }
            }, false);

            WinJS.UI.process(document.getElementById('appbar'))
                .then(function () {
                    document.getElementById('home').addEventListener('click',
                        navigateHome, false);
                });

            WinJS.Navigation.navigate(homePage);
        }
    }
}
```

After the app is activated, a few keyboard events are handled, and then the `WinJS.Navigation` namespace is used to navigate the page to the home page.

Adding fragments

One page is really quite boring. Why would you need navigation for one piece of content, right? To make things a little more interesting, do something wild, like add a page, screen form, or whatever you want to call it. From Microsoft's perspective, it is a *fragment*, and it's treated like a form or page in any other paradigm.

To add a page 2 to the template application, do the following:

1. In the Solution Explorer, right-click the HTML folder and click **Add→New Item**.
2. In the Add New Item dialog box that appears, select the **Page Control** item and name it **Page2.html**, just like Figure 4-8.

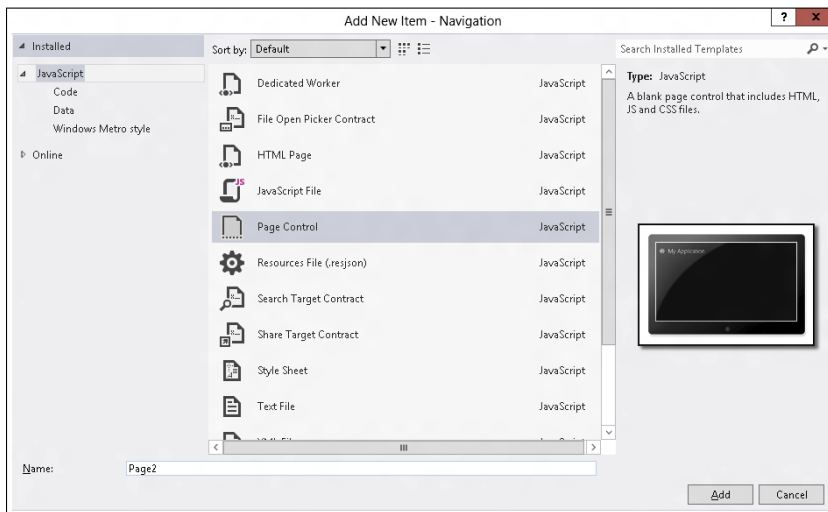


Figure 4-8:
Add a
new Page
Control.

3. Update the body content of **Page2.html** so that you can show some neat-o content when you get there.

```
<body>
  <div class="Page2 fragment">
    <header role="banner" aria-label="Header content">
      <button disabled class="win-backbutton" aria-label="Back"></button>
      <div class="titleArea">
        <h1 class="pageTitle win-title">Welcome to Page2</h1>
      </div>
    </header>
    <section role="main" aria-label="Main content">
      <p>This is my new page 2! Isn't it cool?</p>
    </section>
  </div>
</body>
```

4. In default.html, add a button that will take the user to page 2.

This goes in the app bar div.

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" aria-label="Command
  Bar" data-win-options="{position:'bottom', transient:true,
    autoHide:0, lightDismiss:false}">
  <div class="win-left">
    <button id="home" class="win-command">
      <span class="win-commandicon win-large">&#xE10F;</span>
      <span class="win-label">Home</span>
    </button>
    <button id="page2" class="win-command">
      <span class="win-commandicon win-large">&#xE10F;</span>
      <span class="win-label">Page2</span>
    </button>
  </div>
</div>
```

5. In default.js, you need an event handler for the new button. Add it to the process method for the app bar (it's near the end).

```
WinJS.UI.process(document.getElementById('appbar'))
  .then(function () {
    document.getElementById('home').addEventListener('click',
      navigateHome, false);
    document.getElementById('page2').addEventListener('click',
      navigatePage2, false);
  });
WinJS.Navigation.navigate(homePage);
```

6. Hey, you need a navigatePage2 method, don't you? Add that above the process code:

```
function navigatePage2() {
  WinJS.Navigation.navigate("html/page2.html");
  WinJS.UI.getControl(document.getElementById('appbar')).hide();
}
```

That should be everything you need. Press F5 to run and (if you are on a regular machine) press Windows+Z to bring up the new app bar. Tap that neat new Page 2 button.

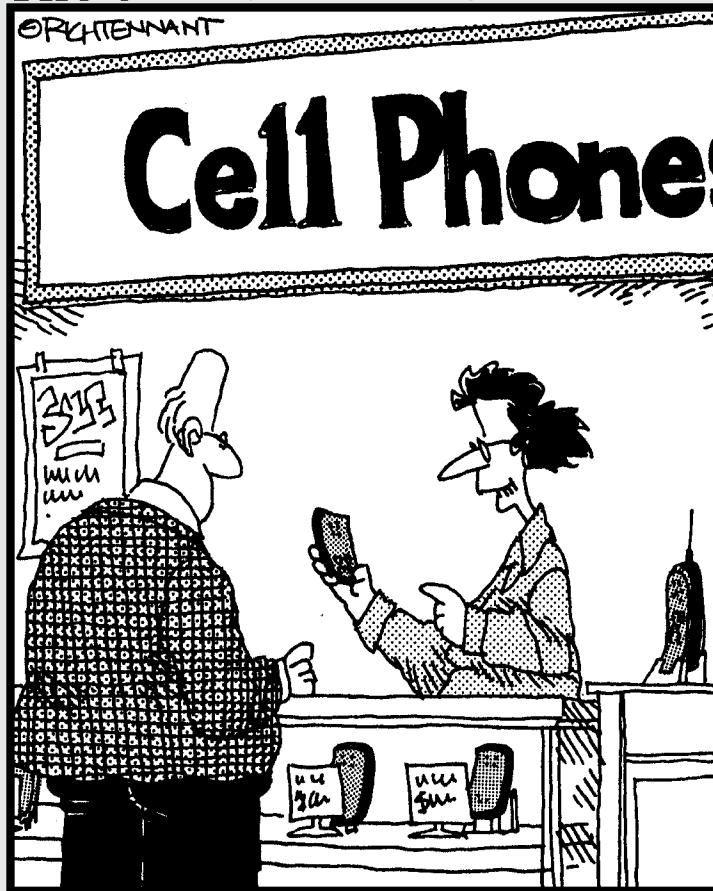
You might need to keep an array of pages in memory and provide a next/back button, or give random access via a menu. The options are endless.

Part II

Working with the Externals

The 5th Wave

By Rich Tennant



“This model comes with a particularly useful function – a simulated static button for breaking out of long-winded conversations.”

In this part . . .

If the user is the center of the Windows 8 world, the user interface is the center of the Windows 8 programming world. Making apps look the right way is easy with the tools that are part of the Visual Studio package, but there are still some things you will need to know. In Part II, I go over how touch has changed the world of HTML programming. I also cover layout, data presentation, and your app's tile.

Chapter 5

Using Everyday Controls

In This Chapter

- ▶ Getting navigation in place
 - ▶ Letting users select items
 - ▶ Creating ratings and sliders
-

HTML5 has a lot of wonderful controls. Some of them are even useful. Many of them, even some of the useful ones, have been Windows 8-enabled. Many of the HTML controls work well in the Windows interface, and the WinJS library helps them handle touch better. WinJS makes touch interaction better everywhere, making the check boxes bigger, the selectable items more accessible, and the margins wider. Also, where it makes sense, WinJS implements WinRT in place of browser functionality.

There are also controls that don't normally appear in the HTML stack. The Windows 8 interface is well modeled by the browser, but not perfectly. Navigation, for one, is different in the Windows 8 app as compared to the browser, so I start there. User-level communication is different, too, and I end the chapter there.

The core of what HTML is all about is the same as it has been for years. The way a user uses a form, text boxes, radio buttons, and whatnot, is all the same.

Using Basic HTML

After the user has started your app, he creates information or consumes information. If he creates information, you need to collect that information from him and do something with it.

This is nothing new to you, if you have been building applications for any length of time. You need to get the info from the user, and you have a set number of understood, common ways to do that. All of the tools that you are probably used to in the web world (like HTML controls and CSS styling) work just fine — maybe better — in the Windows 8 world.



In the sample code, I have a `PageControl` for each of these examples. To change which page launches when you press F5 for debugging, change the Start page in the package.appxmanifest. This is how I ran each of the sample pages in the code on the website. You can see that in Figure 5-1.

Div

The `div` tag isn't a control. It's a division in HTML, and you use it to divide this area from that area. However, in the Windows 8 app, just as in HTML, it is incredibly important and used for nearly everything that has to do with content.

You can see how Microsoft has considered the `div` in its design work when making a new page control. (Remember, a page control makes a fragment of content to which the user can navigate.) To make a new page control, follow these steps:

1. Right-click a new project (I called mine `EverydayControls`) and select **Add New** → **Folder**.
2. Name the new folder `Pages` and click **OK**.
3. Right-click the new `Pages` folder and select **Add New** → **Item**.
4. In the New Item dialog, shown in Figure 5-2, you can see that there is a variety of items other than the selected `Page Control`.

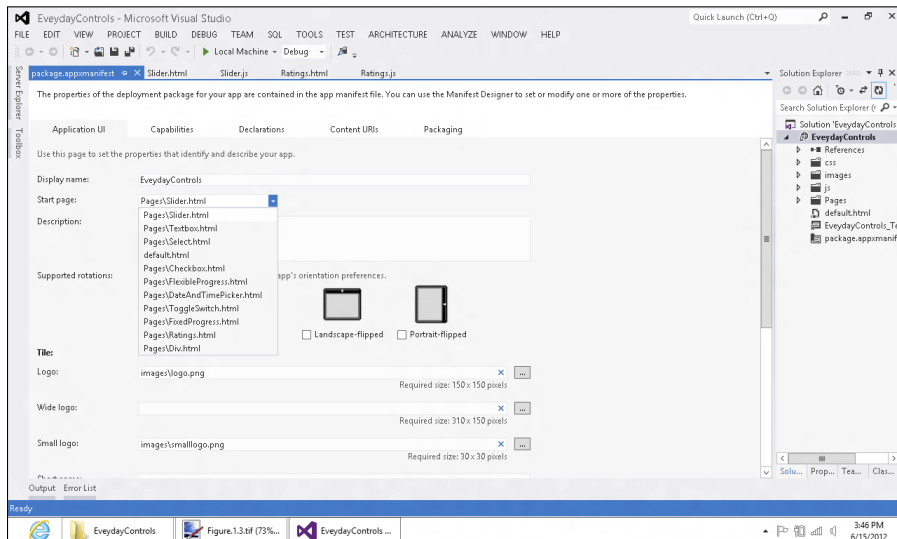
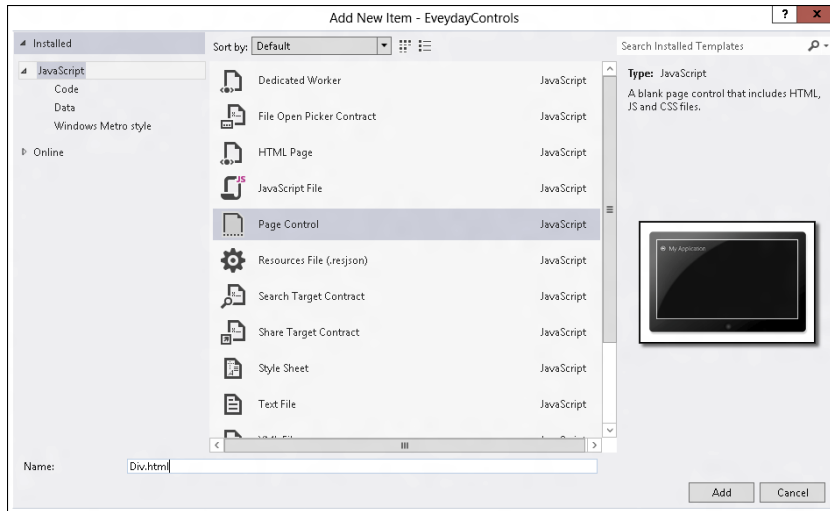


Figure 5-1:
Changing
the Start
page.

Figure 5-2:
Adding a
new page
control.



You want the Page Control item, just like I have selected. This adds a new HTML, CSS, and JS file to your project.

5. Type the name `div` in the Name field and click OK.



Naming the item `div` won't change the content of the template that it uses. That's just because the code in this control is all about the `div` markup.



If you look at the new `Div.html` file, you can see that the base content is all wrapped in a `div`. The content also makes use of the new HTML5 header and section tags, which is awesome. Using these tags makes for a better search experience because search engines know what kind of content they are looking at.

```
<div class="Div fragment">
  <header aria-label="Header content" role="banner">
    <button class="win-backbutton" aria-
      label="Back" disabled></button>
    <h1 class="titlearea win-type-ellipsis">
      <span class="pagetitle">Welcome to Div
    </span>
    </h1>
  </header>
  <section aria-label="Main content" role="main">
    <p>Content goes here.</p>
  </section>
</div>
```

Divs are used for content management. Headers and sections are basically just divs. When you add a section to your code that you want to add more HTML to, you use a `div` to do it.

In the `div` tag in the preceding code, you can see that it has a `class` tag of a `div` fragment. This is the reference to the style for the `div`. If you wanted to, you could change that style in the `div.css` file and change how that whole area of code looked on the screen, but I digress.

For now, go to `Div.js` and find the page level function. The top level function is to prevent all of our page code from “polluting” the whole application’s scope, and making it so that code on this page impacts other pages. Scoping is a totally different topic. For a complete education, check out *JavaScript and AJAX For Dummies* by Andy Harris (published by John Wiley & Sons, Inc.). Just trust me on this one.

First, I show you how to add an `id` function. This is like a macro in LISP (yes, I went there) to make it easier to get to the HTML elements on the page. I recommend that you put the `id` function somewhere accessible to the whole app. For now, I put it in the page control. (I might change my mind by the end of the chapter.)

Then I show you how to change the content of the `ready()` function. This is what it is called when the page is ready to handle element access. If you know JQuery, the `ready` function is like `$(document).ready()`. The final code looks like this:

```
// For an introduction to the HTML Fragment template, see
// the following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232511
(function () {
    "use strict";

    // This function is called whenever a user navigates
    // to this page. It
    // populates the page elements with the app's data.

    var id = function (elementID) {
        return document.getElementById(elementID);
    }

    function ready(element, options) {
        var numbers = [1,2,3,4,5];
        id('numberList').innerHTML = '<ul>';
        for (var x in numbers) {
            id('numberList').innerHTML += '<li>'+x+'</li>';
        }
    }
})
```

```
    }
    id('numberList').innerHTML += '<ul>';
  }

  function updateLayout(element, viewState) {
    // TODO: Respond to changes in viewState.
  }

  WinJS.UI.Pages.define("/Pages/Div.html", {
    ready: ready,
    updateLayout: updateLayout
  });
})();
```

Press F5 to run, and you'll see the 0..4 listing on the left side of the screen, just under the Welcome to Div header.

To make the style a little more entrancing, you can edit the Div.css file. Changing the margin-left parameter to a 40-pixel gap makes it look a little nicer. The resulting code looks like this:

```
.Div p {
  margin-left: 120px;
}
#numberList {
  margin-left: 40px;
}
```

Rerun the page and you can see that it now has the bullets and a little space. Much better.

This styling and code applies to everything in the book, and certainly everything in this chapter. Layout is in the .html file, functionality is in the .js file, and style is in the .css file.

Text box

Ah, the humble text box. It takes data in and gives it back on request. You just can't live without it in the web world, and the same is true with Windows 8 clients. If you are showing data but not editing it, the `div` is for you, but if you need the user to be able to make changes, think text box.

Right-click the Pages directory and add a new page control just as you did in the previous section on `div`. This time, name the page control **Textbox**. A `Textbox.js`, `Textbox.css`, and `Textbox.html` file is created for you. Open the `Textbox.html` file to get start working with the `textbox`.

The Toolbox is where Visual Studio hides all of the widgets, whoosits, and thingamabobs that you can use to build stuff if you are working in HTMLXAML controls. The toolbox is context-sensitive.

With the Textbox.html file open, put the cursor where the `<P>` content goes here`</P>` text is. Click the Toolbox tab to the left of the source code. You should see two tree views: HTML and General, as shown in Figure 5-3. The General tab has . . . well . . . nothing in it. The HTML tab holds a few goodies.

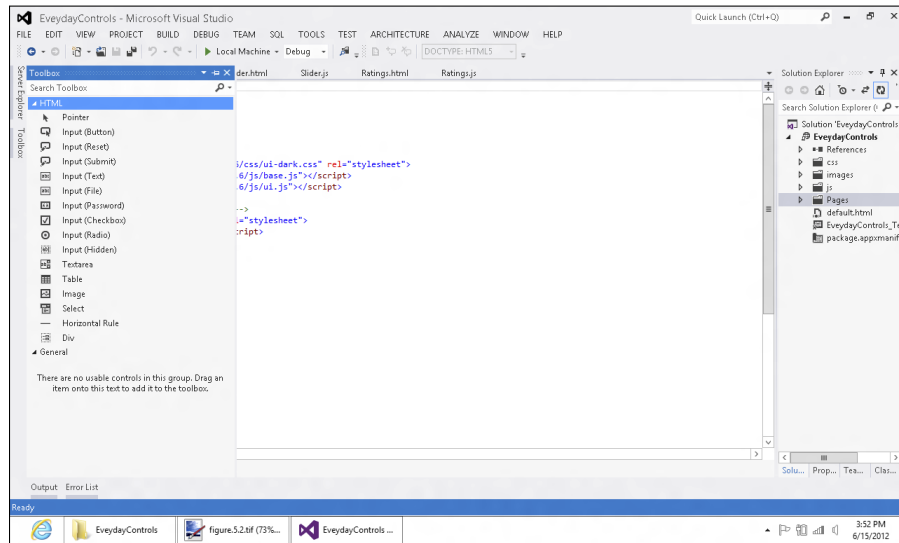


Figure 5-3:
The Toolbox.

And when I say the HTML tab has a few goodies, I mean just a few. Not much of the total collection of items that you can use in a Windows 8 app is there, but there is some benefit. First, it gets you started right in the development of your app. Second, you can add stuff to the Toolbox, like lesser-used HTML. For now, let it get you started. I'll show you how to add stuff later.

Getting the text box on the page

To get started, know that a text box takes a couple of forms. If you know HTML, you probably know that the text box is part of the `Input` family of controls, and that there are a few different kinds, specifically:

- ✓ Text
- ✓ File
- ✓ Password

Generally, the text box is known as a text input field. To bring it to life on the page, you can just drag the Input (Text) item from the Toolbox onto your HTML page. The resultant code looks like this:

```
<body>
  <div class="Textbox fragment">
    <header aria-label="Header content" role="banner">
      <button class="win-backbutton" aria-
        label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to
          Textbox</span>
      </h1>
    </header>
    <section aria-label="Main content" role="main">
      <p>Content goes here.</p>
      <input id="Text1" type="text" />
    </section>
  </div>
</body>
```

Using the text box

After the text box is on the page, we want to do two things with it: get data from it and put data in it. Either way, it works a lot like HTML on the web.

To set a value, you need to get a reference to the text box from the HTML in the JavaScript file, and then set its `value` property. Open `Textbox.js` and add this little bit of code to the `ready()` function:

```
//Setting a textbox
var firstTextbox = document.
  getElementById("Text1");
firstTextbox.value = "Let's type!";
```

That's all there is to it. Run the app by pressing F5, and you can see that the value is set. Pretty straightforward, yes?

Now, to get a value, you need to add a few extra bits.

- 1. In the HTML file, add a second text box.**

It's called `Textbox2` by default. That's fine for now.

- 2. Add a `div`, just like we did in the `div` section earlier in this chapter, below the text box. Give it an `id` of `'youTyped'`.**

- 3. Below that, drag an Input (Button).**

This gives you an event to attach to.

4. The final HTML (at least the body of the HTML) looks like this:

```
<body>
  <div class="Textbox fragment">
    <header aria-label="Header content"
      role="banner">
      <button class="win-backbutton" aria-
        label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to
        Textbox</span>
      </h1>
    </header>
    <section aria-label="Main content"
      role="main">
      <p>Content goes here.</p>
      <input id="Text1" type="text" />
      <input id="Text2" type="text" />
      <div id="youTyped"></div>
      <input id="Button1" type="button"
        value="button" />
    </section>
  </div>
</body>
```

Now, back in the JavaScript, you have to add an event handler. To do that in JavaScript, you use the `addEventListener` method of the control. The code looks like this when it is all done:

```
function ready(element, options) {
  //Setting a text box
  var firstTextbox = document.
    getElementById("Text1");
  firstTextbox.value = "Let's type!";
  //Getting from a text box
  var secondTextbox = document.
    getElementById("Text2");
  var youTyped = document.
    getElementById("youTyped");
  var theButton = document.
    getElementById("Button1");
  theButton.addEventListener('click', function () {
    youTyped.innerHTML = secondTextbox.value;
  });
}
```

Press F5, and then type something into the second text box (on the right). Click the button, and you see something like Figure 5-4.

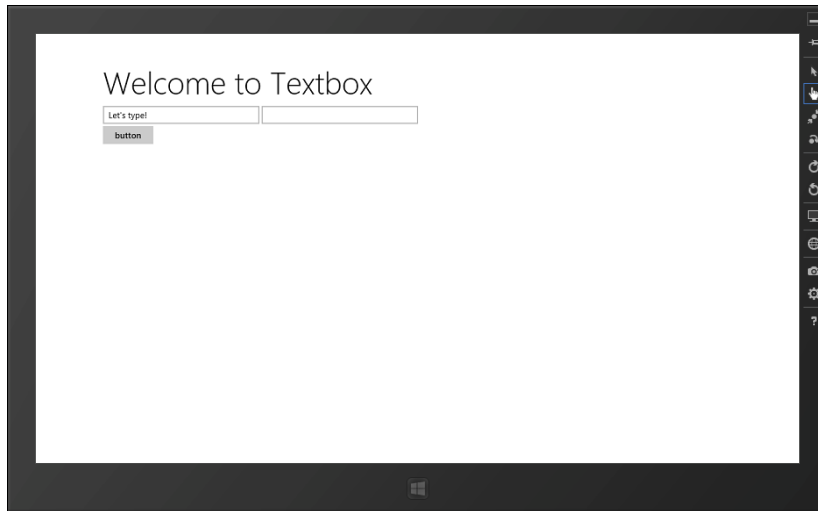


Figure 5-4:
Hello, event
handlers!



Setting the event handler actually changed a property of the text box control to the anonymous function that we added as a parameter of the function call. These event handlers are how the whole functional structure of Windows 8 is handled and are important to learn. You have a lot of that code in the ready functions of your pages when all is said and done.

Select

If you are a Windows developer, you may recognize the `Select` element as a `ListBox`. It is the rendering of a selection list that drops down when you click on it. It shows you a list of options and collapses back to one line after you have selected an item.

The `Select` element is awesome for a number of things: picking from a discrete list of items on a form, navigating to a list of places, choosing an option in settings, or any situation when you want to give the user an (essentially) fixed list of options.

Populating the Select page control

The `Select` tag in HTML is a container tag, and the items in your list are `Options`. The tag and the list of options can be directly manipulated in JavaScript, as you will see, making it an awesome element to handle arrays of data.

1. Add a `Select` page control to the `EverydayControls` project, just as you did in the previous step lists in this chapter.

2. In the HTML file, delete the `<p>Content goes here.</p>` code and add the following:

```
<select id="Controls" size="1">
  <option value="1">Div</option>
  <option value="2">Textbox</option>
  <option value="3" selected>Select</option>
  <option value="4">Radio Button</option>
  <option value="5">Toggle Switch</option>
</select>
```

3. Press F5 to run the Windows 8 app.

You can see that the tag is already in Windows 8 style for you — it's a freebie from the WinJS library. The tag is large enough for touch and fits the style guidelines.

4. Stop the app by going back to Visual Studio and clicking Stop.
5. Open `Select.js`.
6. Add this code to the `ready` function:

```
var selectControl = document.
  getElementById("Controls");
var newOption = document.createElement("OPTION");
newOption.text = "Date Picker";
newOption.value = "6";
selectControl.add(newOption);
```

Here, you're getting a reference to the existing `Select` element with the `getElementById` method, creating a new `Option` element with `createElement`, and then adding the `Option` to the `Select` page control.

7. Press F5 to see the new option.

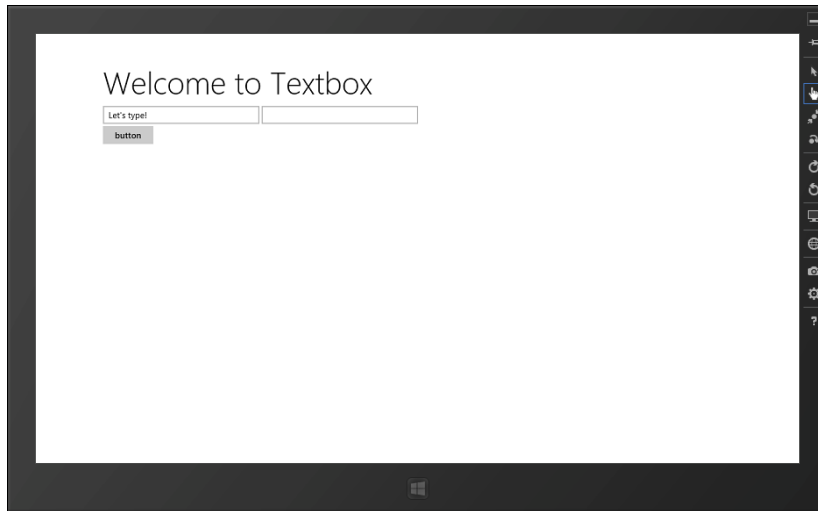
It should look like Figure 5-5.

Looking through the Select items

You can sort through the items in the `Select` list with a `for` loop — the data that comes back from a `Select` is an iterable collection. Below the `selectControl.add` call in the `ready` function, add this loop:

```
for (var item in selectControl.all) {
  if (selectControl[item].innerText) {
    itemsDiv.innerHTML +=
      selectControl[item].innerText + "<br />";
  }
}
```

Figure 5-5:
Selecting a
Select page
control.



Then in the HTML, add the referenced `div`:

```
<div id="Items"></div>
```

What we are doing here is using the `all` collection, which has a few prototypical methods that won't have an `innerText` property (that's why you have a condition in there to check for entities that don't have `innerText`). Run the code again and you will see the items in the `Select`. You also could have populated an array, parsed it with `JSON.parse`, and then sent it back to a service. I show you how to do that in Chapter 10.

Check box

The check box control is an excellent individual control for doing things like providing settings and options. It is a simple on/off switch and is in the WinJS library, so it has Windows 8 styling.

Using a single check box is pretty boring. The HTML looks just like one would expect from a web page:

```
<input type="checkbox" id="newCheckbox"/>  
  <label style="font-size: 18pt">This is a  
checkbox</label>
```

You can set the checked property of the check box in the JavaScript (it's just a Boolean) and read the property similarly:

```
If (document.getElementById("newCheckbox").checked()) {  
    //do work  
}
```

It gets much more interesting when you have a list of check boxes. I discuss databinding a little in Part I, but things get real when you have an active list of data that the user can manipulate.

For instance, in *ToDoToday*, my task management application, I have started the user with a basic list of tasks:

```
dataArray = [  
    {  
        task: "Finish Chapter 1",  
        complete: false  
    },  
    {  
        task: "Get milk",  
        complete: false  
    },  
    {  
        task: "Check eyebrows",  
        complete: false  
    },  
    {  
        task: "Repair TARDIS",  
        complete: true  
    },  
    {  
        task: "Punt on 4th and 8",  
        complete: false  
    },  
    {  
        task: "Deck the halls with boughs of  
holly",  
        complete: true  
    },  
    {  
        task: "Let the spirit fingers jog my  
mind",  
        complete: false  
    }  
]
```

To bind those to a set of check boxes, I need to set up a `ListView`. I discussed `ListView` briefly in Chapter 2, and I use it throughout the book. It is the standard mechanism for making lists of things in the Windows 8 interface.

There are two elements to setting up a `ListView` — the container for the list and the list template. The container sets which template is used for each item and what the core data source is. The template sets what element from each row in the data source goes to which control.

For instance, here is the HTML for the check box list that maps to the array I set up previously:

```
<div id="mediumListIconTextTemplate" data-win-
control="WinJS.Binding.Template">
  <div style="width: 480px">
    <input type="checkbox" data-
win-bind="checked: complete" /><label
data-win-bind="innerText: task" style="font-
size: 18pt"></label>
  </div>
</div>
<div id="todayList" data-win-control="WinJS.
UI.ListView"
  data-win-options="{itemDataSource :
dataList, itemTemplate: select('#mediumListI
conTextTemplate')}" style="margin-left: 5%;
margin-right: 5%; width: 514.4px; left: 0px;
top: 16px;">
</div>
```

The template is on top, and the `ListView` container is beneath. The `ListView` container has two `data-win-options`: the name of the data to be bound, and the name of the template. The template has two controls: the input and label I used previously in the simple example. Notice that they both have `Data-Win-Bind` options that match the names of the elements in the array.



You can also set the binding in the JavaScript code. If you get a reference to the `ListView`, all of the `data-win-options` are properties of the class that is exposed. Should you need to change the binding of a list at runtime, this is a great way to do so.

```
var todayList = document.getElementById('todayList');
todayList.itemDataSource = dataList;
```

ToggleSwitch

The `ToggleSwitch` control is not dissimilar to the `Checkbox` control, but it's a Windows 8-specific control that was created just for the touch friendly interface. That's why I include it here.

Like the other WinJS-specific controls, the `ToggleSwitch` is a `div` with a `Data-Win-Control` property. By itself on a page, it looks like this:

```
<div id="lightSwitch" data-win-control="WinJS.
    UI.ToggleSwitch" data-win-options="{title:
    'Lights'}">
```

Unlike with the `Checkbox` control, you don't need to include a label — the title can be rendered by including a `data-win-option`. The same is true with the state, as you can see in Figure 5-6. It tells the user whether it is on or off. (You can change what the text of the toggle says with `labelOn` and `labelOff`.)

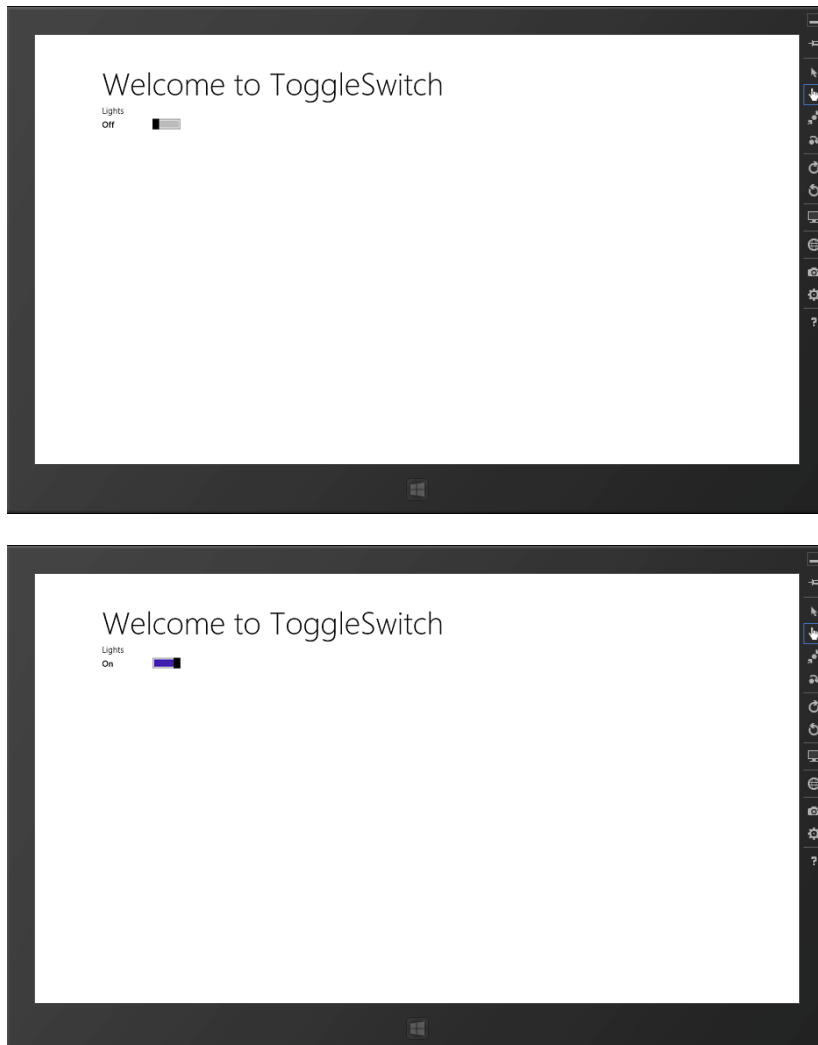


Figure 5-6:
On and off.

Notice that unlike the check box, a toggle switch supports a specific gesture, the *swipe*. I talk about the eight core gestures in Chapter 6, but this is the first control that supports something that the user might not expect.

When I first saw a toggle switch, I tried to tap it, and just got the little bit of motion that Microsoft built into the swipe response, presumably to give users a clue that they are supposed to swipe. Honestly, I am still getting used to swiping the toggle switch.

Consider your users when you pick the controls you use. Does it make more sense to use a check box or a toggle switch? For me, I see the check box as more of a list item, and the toggle switch as more of a single-use kind of thing.

DatePicker and TimePicker

The `DatePicker` and the `TimePicker` are two more WinJS controls in the UI library. They are touch-first, but work well with a mouse and keyboard. Like the other controls, they can be bound to a data item, or can respond to events.

The default `DatePicker` is easy to use, but the good news is that customizing it is easy too. The default `DatePicker` looks like you would probably expect:

```
<div id="setTheDate" data-win-control="WinJS.
    UI.DatePicker"></div>
```

The cool part is the option list, though. It offers a number of options, all settable in the `data-win-options`, that affect how the control looks to the user. They are

- ✓ `Calendar`: Sets which calendar you want to use
- ✓ `Current`: Sets the selected date
- ✓ `datePattern/monthPattern/yearPattern`: A very cool option to set up the pattern for localized apps, like weeks starting with Sunday or Monday
- ✓ `disabled`: Obvious, yes?
- ✓ `Element`: Access to the DOM element
- ✓ `maxYear/minYear`: Controls how far up and back the user can browse dates

Flexible viewing options

The Pattern options are the best — similar to passing a value to `ToString` when displaying a date in C#. Here are just a few of the options:

```
✓ {day.integer} | day.integer(n)}  
✓ {dayofweek.full} | {dayofweek.abbreviated} | {dayof-  
  week.abbreviated(n)}  
✓ {dayofweek.solo.full} | {dayofweek.solo.abbreviated} |  
  {dayofweek.solo.abbreviated(n)}  
✓ {month.full} | {month.abbreviated} | {month.  
  abbreviated(n)}  
✓ {month.solo.full} | {month.solo.abbreviated} | {month.  
  solo.abbreviated(n)}  
✓ {month.integer} | {month.integer(n)}  
✓ {year.full} | {year.full(n)} | {year.abbreviated} |  
  {year.abbreviated(n)}  
✓ {era.abbreviated} | {era.abbreviated(n)}
```

So for example, to show the day of week along with the date, you would pass this string into the options:

```
function ready(element, options) {  
    var datePicker = document.getElementById("setTheDate")  
    datePicker.datePattern = "{day.integer(2)}  
        ({dayofweek.abbreviated})";  
}
```

Doing this nets you something akin to Figure 5-7.

Registering events

As with almost every control in WinJS, you can register events against a named instance. Registering for an event is telling Windows to run a specific function when something happens. There are specific details that Windows watches, such as clicking on a button, that have properties that you can set to function names. When the event happens, the function runs.

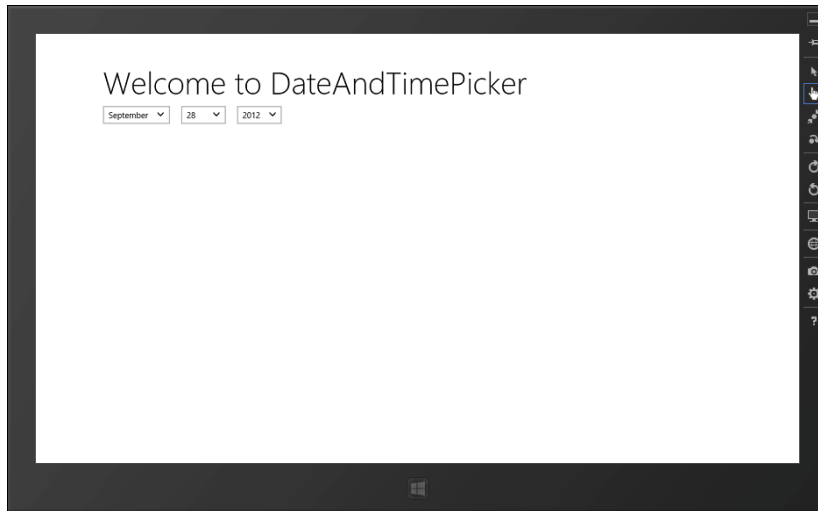


Figure 5-7:
A customized
date picker.

Also like many of the WinJS events, there is only one event to hook to: `onChange`. Registering for an event just requires the use of the property for `onChange`:

```
function ready(element, options) {
    var datePicker = document.
        getElementById("setTheDate")
    datePicker.datePattern = "{day.integer(2)}
        ({dayofweek.abbreviated})";
    datePicker.onChange = theDateChanged;

    function theDateChanged(event) {
        // do work
    }
}
```

Informing the User

Controls that inform the user as to the status of the app are probably used even more often than the interaction controls. Generally, you want to tell a user when they're going to have to wait. The "fast and fluid" premise of Windows 8 demands that if you are going to do something that might take some time, you should tell the user what's going on. That's only fair, right?

And sometimes, you just have information to pass along, and these WinJS controls are pretty good for that too.

Determinate progress status

When you know exactly how much of something should happen, you can use a progress bar that shows the completion percentage. This is for when you are using XHR (the WinJS async `XmlHttpRequest` function) or another communication class with a progress callback and can get the metrics on when something should be complete.



When you don't have access to progress stats, try an indeterminate progress bar or ring, which I discuss in the section titled, "Indeterminate progress status," later in this chapter.

The progress bar

The `ProgressBar` is a HTML5 control that has a current value and max value set. (Not setting those values is how you get an indeterminate progress bar.) To show that the progress is at about sixty percent, you would put this in the HTML:

```
<progress id="fixedProgress" value="60" max="100"></progress>
```

Setting the value isn't hard — it's just a property in the class. (You can of course change it later in the code.)

```
var bar = document.getElementById("fixedProgress ");  
bar.value = 70;
```

Getting access to progress statistics

Things get interesting when we actually start using the progress bar in the way it was designed. The idea is to get progress statistics from potentially long-running operations for which the user must wait, and show them visually somehow.

For instance, the `XmlHttpRequest` has an event listener for progress. Progress event listeners are actually defined in the W3C standard, so they all work about the same way.

In Windows 8, the `ProgressEvent` is fired when the Windows 8 UI gets around to checking the progress, so don't depend on it for anything. That being said, you can certainly update the progress bar.

```
myLongRunningFunction.onprogress = function(prog) {  
    if(prog.lengthComputable) {  
        progressBar.max = prog.total  
        progressBar.value = prog.loaded  
    }  
}
```

Those total and loaded properties are where it is all at. You are effectively setting a percentage for the progress bar, and as such, you give the bar the total amount and the amount done so far, and let it do the math.



Generally speaking, you don't even want to use a progress bar. Something that long-running just shouldn't be in your Windows 8 app, unless you are uploading a picture or downloading a movie.

Also, don't block the UI thread with a progress bar. If you show it, allow the user to continue using your application while progress is being updated.

Indeterminate progress status

In this new world of “fast and fluid,” the phrase “indeterminate progress” should strike you with a cold spike of fear. In general, your application should not leave users waiting. It needs to show the goods or at least the shelves where the goods stay.

And therein lies the rub. The application needs to load slow-loading data at some point. Eventually, you need to load a big image, or get the weather report from a slow service, or wait for the message from a remote server overseas.



That's when you use an indeterminate progress status icon. Not when you are loading the main page of your app, or any page of your app. When you are loading some slow-loading data into a specific location, show the progress icon there. Only there. Don't use it for a slow loading app — your app will be rejected.

Are we clear on that? Good.

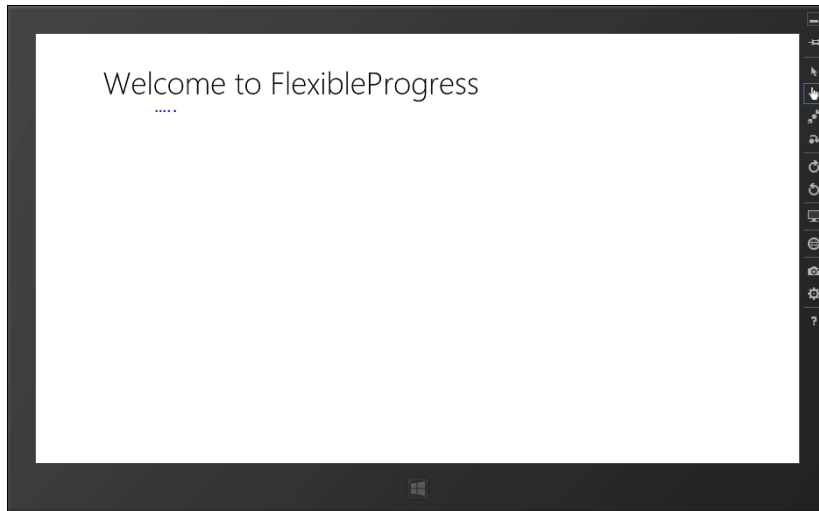
Using the progress bar

If you use the progress control I used in the “Determinate progress status” section without any parameters, you get the standard Windows 8 loading status bar, with the five dots like I show in Figure 5-8. The code looks like this:

```
<progress id="flexibleProgress"></progress>
```

To give the user feedback that the process is continuing, just show the control. To show the user the app is ready, hide the control. It's that straightforward.

Figure 5-8:
The
Windows 8
loading
status bar.



Blocking the user

On the very rare occurrence when you might have to stop the user interface (like when they have to wait for a file to load), you use a different progress control, called the ring. The *ring* is just a progress control with a different CSS style that is built into WinJS. As such, you can implement it like this:

```
<progress class="win-ring"></progress>
```

Text that is in the `innerHTML` of the control shows up in the center of the ring. You can give the user an idea of how much is complete (total minus loaded, divided by total), or just give them kind words of moral support.

How to use it is significantly less important than *when* to use it. As I may have mentioned before (ahem), the Windows 8 UI should be fast and fluid, and you shouldn't block the user. And yet, here I am with a nice control that makes it easy.

At times, blocking a user is unavoidable. You need to do all the math to prep the images for a game, or queue up some music, for example, and that's just going to keep the user waiting for a bit.



The idea isn't to make an app that never makes the user wait. The idea is to never make the user *feel* like she is waiting. Design your app to do as much as it can asynchronously, and load information while the user is doing other things, so as not to have to use the progress bars.

Ratings

Rating controls are a lot more fun than progress bars. They are designed to show the user rating of an object — either the user's rating or the average rating from a community.

Microsoft has gone all in with ratings. You can see the rating control at work everywhere, from the Contoso Cookbook sample app to the Windows Store app. One of the quickest ways to build community is to allow people to express their opinion of something, and the ratings control is a good way to do it.

Ratings are everywhere

Looking through my Windows 8 box, I see ratings anywhere there are users sharing access to information.

- ✓ Apps in the store
- ✓ Videos on the Build app
- ✓ Forum posts
- ✓ Music in the Listening app
- ✓ News stories
- ✓ Add-ins for Visual Studio
- ✓ Movies in Netflix

Use of ratings in Windows Store apps all goes back to Windows 8 being connected. It's not just a connection to the Internet we are talking about here; it is a connection between people that really gets apps like these an audience. Giving people an opportunity to share their opinion on a movie, product, or comment makes them feel more connected to the app, the item in question, and the community at large.

Using the Rating control

Unlike the progress controls, which are HTML5 standards and have their own tags, Ratings are Windows 8-specific and are implemented as `Data-Win-Control`. As such, you can set the properties of the control at design time in the HTML, or at runtime with JavaScript.

Including a rating control on the page works much like any other of the WinJS controls:

```
<div id="rating" data-win-control="WinJS.UI.Rating"></div>
```


After you reference the control in the JavaScript, you have access to three significant properties on the control:

- ✓ `averageRating`: What the community rated the item
- ✓ `userRating`: What the user rated the item
- ✓ `maxRating`: The maximum possible rating for the item

The idea is that you should be using some data store for your item list in a Windows 8 app that contains these three basic pieces of ratings data. The data store should tell you the max possible review. When you collect the list of items and bind the data on your page to them, you set the community's review. Then you can use local settings or the user's own data from the source to set the user review.

Take, for instance, Amazon. You can give a maximum of five stars for a review. For my book *C# 2010 All-in-One For Dummies* (published by John Wiley & Sons, Inc.), the average review is 3.8 (not bad if I say so myself). The JavaScript would look like this:

```
// For an introduction to the HTML Fragment template, see
// the following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232511
(function () {
    "use strict";

    // This function is called whenever a user navigates
    // to this page. It
    // populates the page elements with the app's data.
    function ready(element, options) {
        //Get a reference to the HTML control
        var rate = document.getElementById("rating").
            winControl;
        //Amazon has a max of 5 stars
        rate.maxRating = 5;
        //The current community rating is 3.8
        rate.averageRating = 3.8;
    }

    function updateLayout(element, viewState) {
        // TODO: Respond to changes in viewState.
    }

    WinJS.UI.Pages.define("/Pages/Ratings.html", {
        ready: ready,
        updateLayout: updateLayout
    });
})();
```

You can see the end result of the settings in Figure 5-9. The community review is preset by the code. The control is still active, however. Go ahead and click on the control and set your own settings. The `onChange` event can be handled in the JavaScript just like any other control in the library, and the user rating can be saved back to the data source. You can see an example of how that works in the next section (the slider is a different control, but it works the same).

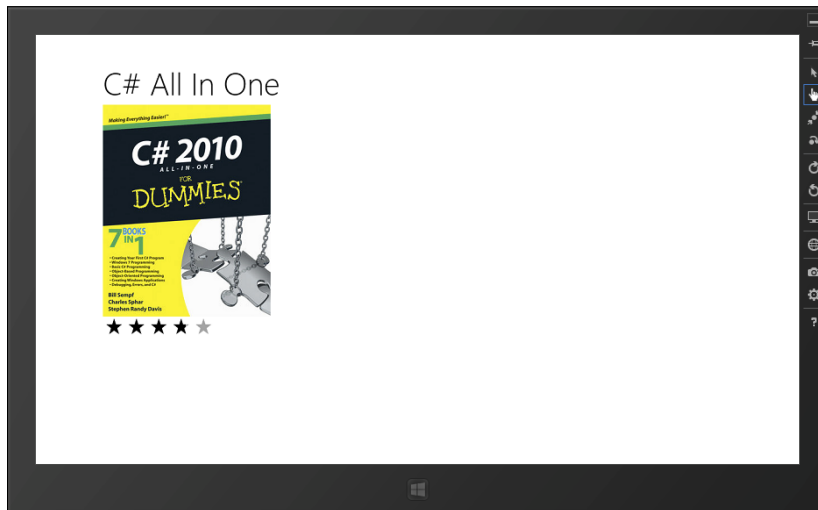


Figure 5-9:
The rating
control.

Sliders

Like the Rating control, the Slider shows the user a preset value in a discrete range. Also like the Rating control, the Slider allows the user to change that value, and the change event can be captured and the value saved or used.

Unlike the Rating control, the Slider allows fine-tuned control of the number in question. Also unlike the Rating control, the Slider is a general-use control. While Ratings are just for ratings, the Slider can be used anytime you want to inform the user about a number in a range. Also unlike the Rating control, the Slider is a HTML Input type, called Range.

Sliders are awesome for such things as settings, sizes, or colors — anything that has a smooth range rather than a discrete set of values. Another example might be a Percentage Complete slider. The user might drag the slider back and forth to describe how much they have read of a book, for example.

The core properties of the range HTML control are all implemented well in Windows 8. They include

- ✓ max: The highest number you can set the bar to
- ✓ min: The lowest number you can set the bar to (may be negative)
- ✓ step: The increment that the bar moves
- ✓ value: The current set value

All of these can be initially set in the HTML and changed in the JavaScript. For instance, the example of the percentage read of a book might look like this, with a min of 0, a max of 100, and a step of 1.

```
<section aria-label="Main content" role="main">
  <br />
  I am <div id="percentComplete">0</div> percent done
    with this book.<br />
  <input id="slider" type="range" min="0" max="100"
    step="1" value="0"/>
</section>
```

This gives us the look in Figure 5-10, which admittedly isn't styled very nicely.

To change that text, you just need to handle the `onChange` event. This looks more or less the same as all of the other controls, although it might feel a little more active. Change the `ready` function as shown here and add the `sliderChanged` function:

```
function ready(element, options) {
    document.getElementById("slider").
        addEventListener('change', sliderChanged,
            false);
}
function sliderChanged() {
    document.getElementById("percentComplete").
        innerHTML = document.getElementById("slider").
            value;
}
```

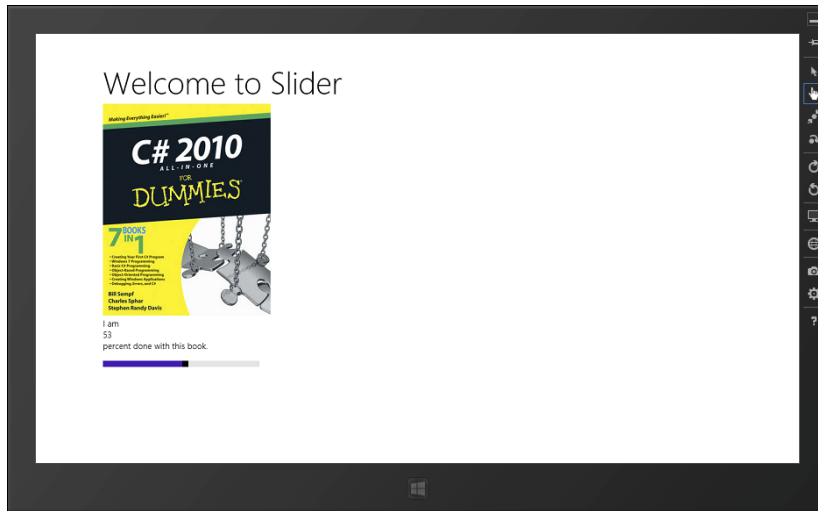


Figure 5-10:
The Slider
control.

Now run the program with F5 (remember to change the Start page in the package.appxmanifest) and drag the slider around. Every time the UI senses a change, it fires off your new method, and you can see the number change as in Figure 5-10. Pretty slick, huh?

Playing Media Files with HTML5

Two of the new tags in HTML5 handle much of your media playback needs in Windows 8, as long as you stick with WinJS. The `<audio>` and `<video>` tags work in Windows 8 just as they do in the browser, and, in fact, are hardware-enhanced thanks to WinJS and WinRT. It's a good combination — better, in fact, than what you can find in the XAML and C# world.

Listening to audio

The audio tag is one of the nicest, simplest things to do in the world of HTML5. You don't need fancy JavaScript or CSS. Just a little dab will do ya. Try these steps to add the audio tag to your app.

1. Add the audio tag to your page with a `src` element that points to your file.

```
<body>
  <div class="audio fragment">
    <header aria-label="Header content"
      role="banner">
      <button class="win-backbutton" aria-
        label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to
        audio</span>
      </h1>
    </header>
    <section aria-label="Main content"
      role="main">
      <audio src="http://archive.org/download/
        VaughnDeLeath/BlueSkies.mp3" controls></audio>
    </section>
  </div>
</body>
```

2. If it's an Internet file (like in the example), make sure the Internet Client is checked in the `package.appxmanifest`.

If you are setting up something that should give the user control, the control parameter is the way to go — they get the full play/pause thing. If you want to play in the background, use the `autoplay` parameter. It happens invisibly.

```
<body>
  <div class="audio fragment">
    <header aria-label="Header content"
      role="banner">
      <button class="win-backbutton" aria-
        label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to
        audio</span>
      </h1>
    </header>
    <section aria-label="Main content"
      role="main">
      <audio src="http://archive.org/download/
        VaughnDeLeath/BlueSkies.mp3" autoplay></audio>
    </section>
  </div>
</body>
```

Watching video

Video isn't any more difficult than audio in the Windows 8 space. You essentially need to add the video tag, and then declare the Internet client capability if you are streaming from something other than local sources.

1. Add a video tag to your markup:

```
<body>
  <div class="video fragment">
    <header aria-label="Header content"
      role="banner">
      <button class="win-backbutton" aria-
        label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to
        video</span>
      </h1>
    </header>
    <section aria-label="Main content"
      role="main">
      <video src="http://archive.org/download/
        Cake/Cake.wmv" controls/>
    </section>
  </div>
</body>
```

2. Make adjustments to the video properties.

One great example would be height and width. Leaving it up to Windows 8, like the browser, is a mistake. Do what you can to get the video dimensions from the source before you encode the tag.

3. Declare the Internet Client capability in the package.appxmanifest file.

Set the capability by double-clicking the file, and then selecting the Capabilities tab.



The Internet Client capability is selected by default in Visual Studio, but your mileage may vary.

That's about all there is to it. The VCR controls are added with the controls tag, and the hardware acceleration is done automatically. Doesn't get much easier than that.

Chapter 6

Laying Out the App

In This Chapter

- ▶ Implementing Windows 8 style guidelines
 - ▶ Getting around in your app
 - ▶ Animating content
 - ▶ Handling resize events
-

In Chapter 2, I go over the built-in templates for Windows 8 development. They are pretty handy, especially as a reference, but they won't do everything that you need to do. The Grid and Split templates provide a nice starting point for list/detail styles of apps. The Fixed template is great for single page apps. The Navigation template is good for hub-style apps that were covered in Chapter 2.

Eventually you'll need to meet specific requirements of your application, and the built-in templates won't do the job. In fact, they probably won't even come close. You'll need to write some layout yourself.

This being HTML5, making a Windows Store app is kind of like setting up a web page. You have a site map of sorts, and you navigate from page to page.

This being Windows 8, of course, you have some special considerations to think about. There are no menus, and your app shouldn't have a landing page because the app should be more or less focused on a single-use case.

The site layout is one thing, but the page layout is quite another thing. As I discussed in Chapter 3, Windows 8 has a number of layout requirements that can make or break you in the Store testing. In Chapter 3, you read about it. In this chapter, you implement it.



The page layout drives the user experience. Your app needs to offer a clear path to getting things done, and leave no question in the user's mind how to get to and fro. The app also must be nice to look at. Cramped user interfaces are verboten and will get you tossed from the Store.

The page layout also drives user acceptance. People are more likely to use your app, put in app content, and rate you higher if they like to use your app. It's that simple. The Windows 8 guidelines are not designed to shackle you; they are designed to make your app fit in with the seamless experience that is Windows 8. Use them, don't fight them.

Building a Windows 8 Layout

People like Windows 8 because it focuses on the stuff they want to do, look at, and play with, and not the mechanism of running said content. Some content requires a lot of buttons and switches, but not all of it does. Most types of content don't need a menu bar to run.

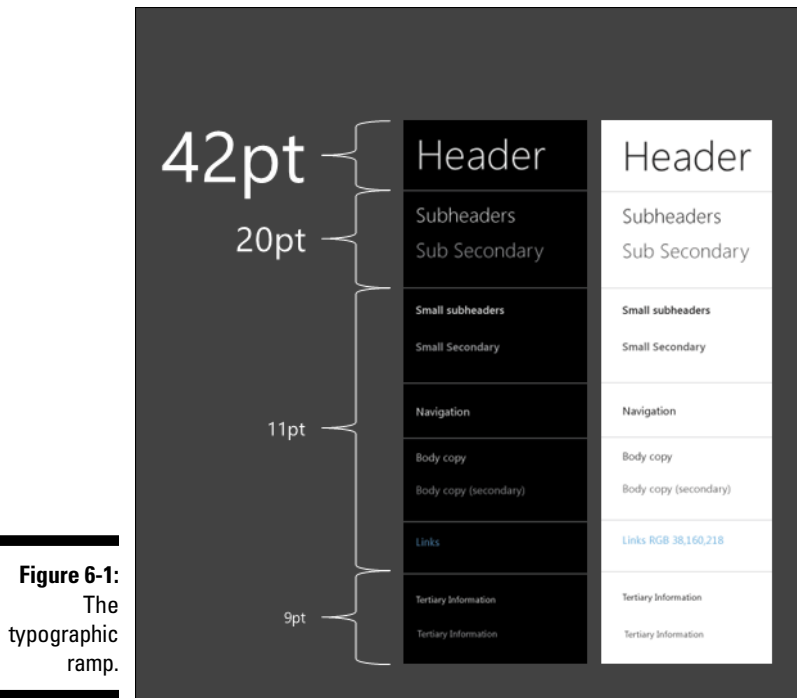
In Chapter 2, I talk about chrome and solving the problems of designing in a world where there is no status bar or menu bar. Here I show you how to do all of it.

Using typography

Windows 8 drives the user's understanding of hierarchy with font sizes. The "typographic ramp" shown in Figure 6-1 defines what is important and what is inconsequential.

This is not terribly different from a website, where heading sizes H1, H2, H3, and whatnot have default definitions in the browser. The Windows 8 environment has default setups for size too, but you get to manage them through the default style sheet.

This isn't to say that you can't make up your own header layout, but you might want to consider doing more or less the same thing that is in the default styles if you want to avoid trouble getting into the Store.



Researching Microsoft's style sheet

The suggested setup isn't much different from what most of us do anyway. Compared to the Normal style — that is, standard text — subnormal text is about 80 percent of the size.

The headers are pretty standard too. The subheader is about 200 percent larger than the normal text. The header is about 200 percent larger than that, or 400 percent larger than standard text.

You can break down the style sheet in the WinJS library to see what I mean. The fonts and sizes are laid out just like Figure 6-1.

```
h1, .win-type-xx-large {
    font-size: 42pt;
    font-weight: 200;
    line-height: 1.1429; /* 64px when font-size is 42pt */
}
h2, .win-type-x-large {
    font-size: 20pt;
    font-weight: 200;
    line-height: 1.2; /* 32px when font-size is 20pt */
}
h3, .win-type-large {
    font-size: 11pt;
    font-weight: 600;
    line-height: 1.3636; /* 20px when font-size is 11pt */
}
h4, .win-type-medium, code, pre, samp {
    font-size: 11pt;
    font-weight: 400;
    line-height: 1.3636; /* 20px when font-size is 11pt */
}
body, h5, .win-type-small, legend {
    font-size: 11pt;
    font-weight: 300;
    line-height: 1.3636; /* 20px when font-size is 11pt */
}
.win-type-x-small {
    font-size: 11pt;
    font-weight: 300;
    line-height: 1.3636; /* 20px when font-size is 11pt */
}
h6, caption, figcaption, small, .win-type-xx-small {
    font-size: 9pt;
    font-weight: 400;
    line-height: 1.6667; /* 20px when font-size is 9pt */
}
```

A combination of size, weight, and line height are used to implement the ramp shown in Figure 6-1. One thing I found interesting was the difference between H3 and H4. The fonts are the same size as the body text, but they are weight 600 and 400 as compared to the body text's 300.

Implementing the styles

I recommend you use Microsoft's style sheet for most apps. For some apps, you may want something else, but that's the 20 percent case. The standard styles should meet most people's needs.

Because of the way Microsoft's team structured the styles, you can just code the way you would normally code HTML and get the Windows 8 look for free. Headers are set up just right compared to the ramp, so use them! Headers

are kind of passé in the world of HTML4 programming, but in HTML5, they semantically make a lot of sense.

Headers aren't the only style bit that you might not be used to using. The semantic nouns like *code*, *legend*, and *caption* have also fallen out of favor, but now is the time to bring them back. Not only are they particularly well suited for Windows 8 style development stylistically, but they also help with the description of the content of the page. Captions, well, are captions, and should be treated as such.



The Semantic Web conversation is a big one and I don't have space to get into it here. Essentially, it provides a framework for information to be presented on a page in such a way that both the styles and the data definition are controlled by the markup. To learn more, browse http://semanticweb.org/wiki/Main_Page.

The default styles also include the `win-type` styles that are used in WinJS controls like the `SemanticView` and `ListView`. These styles are automatically used by the controls, so don't worry about them. The WinJS library picks them up when necessary.

The implementation details for the default styles are unsurprising. The header tags are just used as defaults, such as `<H1>This is a large header</H1>`. The other semantic nouns are used the same way, like `<code>//This is sample code</code>`.



Semantic nouns might have other styles associated with them. For instance, the preceding `<code>` example also includes a monotype font and some margin settings.

Making space

Typography isn't the only visually differentiating factor in the Windows 8 design language. White space plays a huge role too. In Chapter 2, I give you a few dos and don'ts as well as some nice examples. In this chapter, I get into implementation details.

White space in Windows 8 is just about the same as white space in a web app, only you need more of it. When you build the structure of your screen, use margins and padding to frame content with white space.

For this example, I started a new project in Visual Studio. You can do the same or open the sample project from the book's sample code. In the default HTML file, replace the body text with a few headers and `div`s.

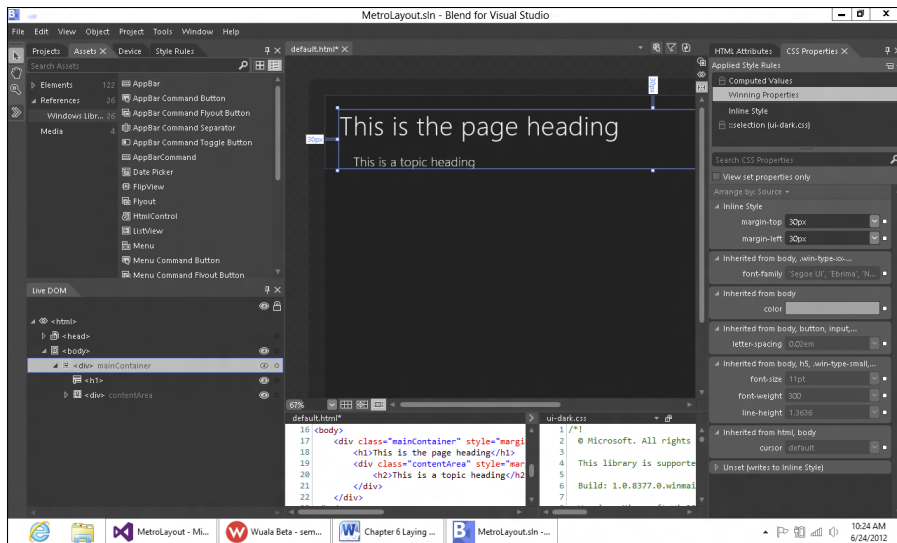
```
<div class="mainContainer">
  <h1>This is the page heading</h1>
  <div class="contentArea">
    <h2>This is a topic heading</h2>
  </div>
</div>
```

Next, right-click the project and open it in Blend.

1. Click on the **mainContainer** div in the HTML view.
2. Find the margin settings in the CSS Properties panel.
3. Change **margin-top** and **margin-left** to 30.
4. Click on the **contentArea** div and make the same changes.

The resulting screen in Blend should look something like Figure 6-2. Notice that the white space is shown in the layout screen with cool rulers.

Figure 6-2:
Setting
white space
in Blend.



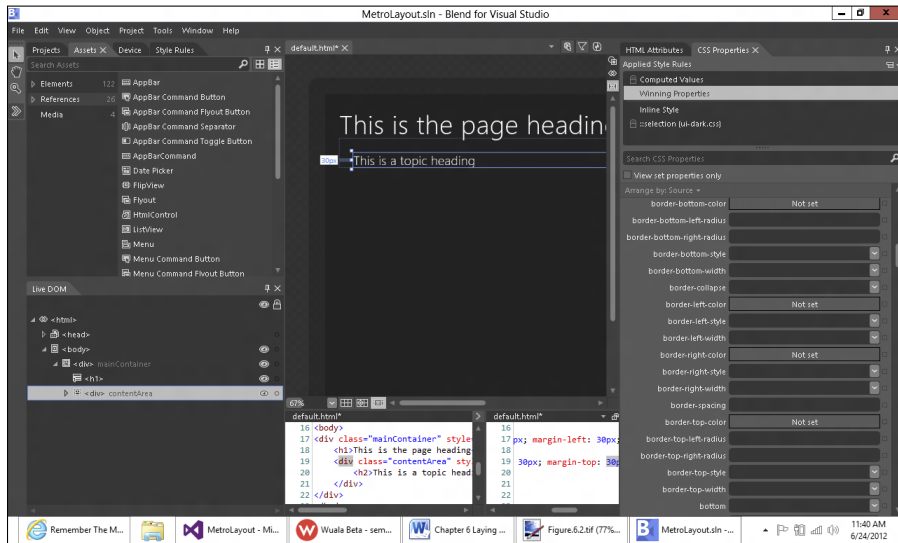
The resultant styles appear inline for the divs, which is just fine.

```
<div class="mainContainer" style="margin-top: 30px;
  margin-left: 30px;">
  <h1>This is the page heading</h1>
  <div class="contentArea" style="margin-left: 30px;
    margin-top: 30px;">
    <h2>This is a topic heading</h2>
  </div>
</div>
```

In Chapter 2, I talk about how spacing, rather than windows and lines, is how we define space in Windows 8. A lot of what needs to be done to make this happen really needs to *not* be done.

For instance, don't surround `div`s with borders or populate them with colors. Instead, use the white space and indentation to show the layout. All of the border styles are shown in Figure 6-3. Don't use them unless you have a really good reason.

Figure 6-3:
The border
styles in
Blend.



Keeping ergonomics in mind

After you've given the user a grip on the visual hierarchy, you need to make it as simple as possible for him to move around the application.

As I mentioned in Chapter 2, everything should flow one direction. In order to best handle this, you have to get a grip on which direction your repeater controls are going to move.

The `ListView` is the primary control that needs help in this area. Fortunately, Microsoft has provided the hooks that you need to effectively handle scrolling. Three classes cover the layers you need to style:

- ✓ **win-surface:** The entire content area, viewable or not
- ✓ **win-viewport:** The part of the content that you can see
- ✓ **win-listview:** The whole control

You can see a breakdown of these three entities in Figure 6-4.

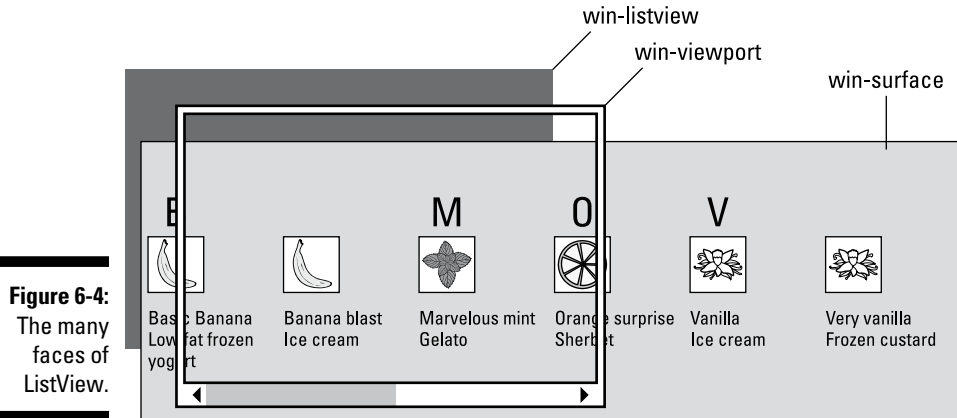


Figure 6-4:
The many
faces of
ListView.

The win-surface defines the scrolling direction of the content. If the win-surface is bigger than the win-viewport to the right, the control scrolls horizontally. If the win-surface is bigger than the win-viewport to the bottom, the control scrolls vertically.

If the win-surface is bigger than the win-viewport in two directions, you are doing it wrong.

In Windows 8, everything should scroll the same way. It doesn't matter if you are scrolling vertically or horizontally, as long as you only do one or the other.

That being said, Microsoft *really* wants us to scroll horizontally. That goes against the grain of many web developers, including your humble author. Nonetheless, after using many Windows 8 apps, I have to admit that on a tablet especially, it makes a lot of sense.

You don't have to manually style the layout of the win-surface either. The templates for list and grid view that are provided in WinJS do a lot of the heavy lifting for you. For instance, check out the CSS for the `textListMediaQueueTemplate`. The win-viewport is pre-specified to handle the scrollbar, and your ListView automatically pushes the win-surface to the bottom.

```
/* overall list dimensions */
#textListMediaQueue
{
    margin-left: 110px;
    width: 618px; /*+18 px to account for scrollbar*/
    height: 632px;
```

```

}

/*-----*/
/* textList-mediaQueue - used for Media app queue item*/
/*-----*/
/*-----*/
/* individual item dimensions */
.textListMediaQueue
{
    display: -ms-flexbox;
    width: 100%;
    height: 40px;
    overflow: hidden;
    padding: 8px 5px;
}

.textListMediaQueue .textListMediaQueueDetail1
{
    -ms-flex: 1 auto;
}

/* Text line 1: Track */
.textListMediaQueue .textListMediaQueueDetail1
    .textListMediaQueueTextTrack
{
    overflow: hidden;
}

/* Text line 2 col 1: Artist */
.textListMediaQueue .textListMediaQueueDetail1
    .textListMediaQueueTextArtist
{
    width: 280px;
}

/* Text line 2 col 2: Album */
.textListMediaQueue .textListMediaQueueTextAlbum
{
    line-height: 60px;
    width: 180px;
}

/* Text line 2 col 3: Length */
.textListMediaQueue .textListMediaQueueNumberLength
{
    line-height: 60px;
    margin: 0px 10px;
    width: 40px;
}

```




This is another example of getting to know the built-in tools and using them when you can. Microsoft put those templates in there to make it easier for you to build products. They have the added benefit of already being in Windows 8 style. Use them.

Putting things in a grid

Do you know what Microsoft likes even more in Windows 8 apps than a nicely scrolling list? A nicely scrolling grid. You should always put things in a grid when you can, preferably with a nice image. Now, that doesn't always work, but when it does, you should consider it.

There are a number of Grid templates, as shown in Table 6-1, and they start with the template that is in the New Projects dialog box.

Table 6-1 Grid Templates		
<i>Template</i>	<i>Type</i>	<i>What It Does</i>
iconTextApplicationsTemplate	Icon and text (applications) template	Defines an item that represents an item. The item contains an icon and descriptive text.
imageGalleryTemplate	Image (gallery) template	Defines an item in an image gallery.
imageGalleryBasketTemplate	Image (gallery basket) template	Defines an item in an image gallery basket.
imageOverlayAlbumTemplate	Image overlay (album) template	Defines an item that shows album info: an image of the album, info about the album and the artist.
imageOverlayGalleryFolder Template	Image overlay (gallery folder) template	Defines an item for a folder in an image gallery.
imageOverlayLanding Template	Image overlay (landing) template	Defines an item that contains an image, a section with large text, and a section with small text.

Template	Type	What It Does
imageAndTextCollection Template	Image and text (collection) tem- plate	Defines an item that contains an image, a large heading, a sub- heading, and body text.
imageAndTextListFile Template	Image and text (list file) template	Defines an item that represents a file: an icon, the name, a date, and its size.
textAlbumTrackTemplate	Text (album track) template	Defines an item that contains the track number and title of an album track.
textTileLandingTemplate	Text tile (landing) template	Defines an item that represents a tile. The item contains a sec- tion with larger, bold text, and a section with smaller text.
textTileListFolderTemplate	Text tile (list folder) template	Defines an item that represents a tile. The item contains a heading.

I'll leave looking at all of these templates as an exercise for you, the reader. Describing them all here would add another hundred pages to the book.

To use one of the preset templates for grids, just use a hashtag before the name of the class you need in the `data-win-control` definition:

```
<div id="basicListView"
  data-win-control="WinJS.UI.ListView"
  data-win-options="{ itemDataSource : DataExample.
    itemList.dataSource,
    itemTemplate: select('#mediumListIconTextTempl
ate'),
    layout: {type: WinJS.UI.ListLayout}}">
</div>
```

Interacting with Your App

The user is supposed to feel that the app is connected and alive. One of the best ways to accomplish that is to make the navigation a consistent part of the data.

The World Wide Web went a long way in the right direction with the Anchor tag, but still we developers tend toward the “Click here to view this story” model of navigation.

Back in the day before search-engine-optimization experts existed, I discovered something interesting. If you had content that said *Click here for more about widgets*, and you made the *Click here* part the hyperlink, you did poorly in the search ratings. If you made the word *widgets* the hyperlink, you did much better.

This is because the Semantic Web depends on meta information for the pages it indexes. “Click here” tells me nothing. “Widgets” tells me everything. The search engine only has a few things to glean metadata from — the headings in the target and the link to the target. That’s about it. Make them both as good as you can.

Integrating navigation

In that vein, there is no navigation, per se, at all in Windows 8. The content drives everything. From games to newsreaders, the content should be the navigation.

Actually doing this is simpler than it sounds. Instead of a Read More link, you just make the content the link. Use the Navigation library to get from page to page.

The best example of that is in the default Grid template in Figure 6-5. Clicking on the item takes you to the content page.

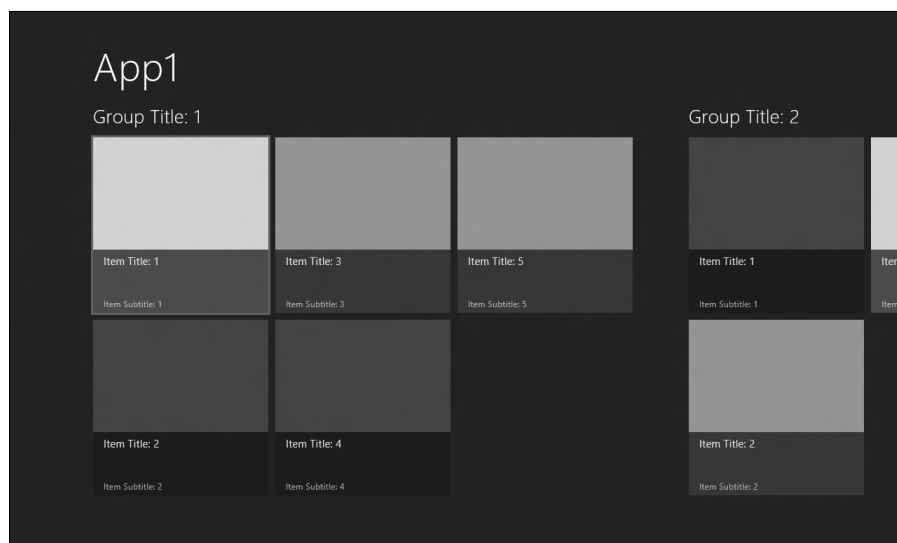


Figure 6-5:
The default
Grid
template.

The Navigation library in WinJS gives you a method to pass the page name and the relevant model to the page. It is handled in the `itemInvoked` event of the page:

```
itemInvoked: function (args) {
    var item = this.items.getAt(args.detail.
        itemIndex);
    nav.navigate("/pages/itemDetail/itemDetail.
        html", { item: Data.getItemReference(item) });
},
```

That's about all there is to it. When you have content, link the content item template to the detail. If you find yourself about to make a button, stop. Don't do it.

Some situations call for onscreen navigation, however: for example, a Back button. The idea of a Back button is so firmly ingrained in our browsing experience that many users find it hard to live without. If you look at the detail page in the Grid template, you can see a Back button, which is acceptable in the Windows 8 world.

Also, if you have a content creation app, you might need navigation to features like Add and Edit. Those buttons shouldn't be on the pane however; they should be in the app bar.

Using the app bar

The app bar has made an occasional appearance in the first chapters of this tome, but it deserves a more significant introduction. The app bar is important. For instance, if the controls for your app don't fit in the app bar, the app is too complex for Windows 8.

That's a significant statement, and this is me speaking, not Microsoft. Nonetheless, I implore you to consider the truth here. Take a look at the app bar section of Chapter 2 and consider whether your app is the right size.

The app bar is a feature of WinJS, which is channeling the WinRT libraries. It is a totally new feature of Windows, much like the System Tray way back in Windows 95.

At this point, I probably don't have to reiterate that the app bar is what comes up from the bottom when you swipe from the bottom. Nor should I have to say that buttons go there: page-specific ones to the right, global ones to the left.

The app bar is a WinJS control. As such, it is implemented as a `div` with a `data-win-control` property.

```
<div data-win-control="WinJS.UI.AppBar">
  <button data-win-control="WinJS.
    UI.AppBarCommand"
    data-win-options="{icon:'back', id:'',
      label:'example', onclick:null, section:'global',
      type:'button'}"></button>
</div>
```



By far, the easiest way to add an app bar to a page is to use Blend. That's what I did in the example. Use the Assets panel, under References. It's right on the top. Buttons and stuff can be added there too.

Speaking of buttons, the `<button>` tag is put to good use here. Because it isn't a form, `<input>` would be less useful and inaccurate. It lacks some functionality in HTML5, however, so we label it as an app bar command and set up some options.

AppBar itself is just a container. The `AppBarCommand` class is where it is at. That's what defines all of the cool buttons that I talk about in Chapter 2. A handful of properties make the buttons the most effective way to handle complex functionality in a Windows 8 app, as shown in Table 6-2.

Table 6-2 Properties of the AppBar Buttons

<i>Property</i>	<i>What It Does</i>
Disabled	Enables or disables the individual button.
Flyout	Awesome property that auto invokes a flyout when the button is pressed. Try it!
Hidden	Can you see it? No! It's hidden!
Icon	Gets or sets the icon — see Figure 6-6.
Label	The text with the button.
Onclick	The onclick method. You can also assign this with a listener at design time.
Selection	Selection or Global. Selection signifies buttons for that page, global for the whole app.
Tooltip	What the user sees when they mouse over the button.

I include bunches of examples of using the app bar and the `AppBarCommand` in almost every chapter of this book, but I want to make sure you have some neat icons you can try out. The JavaScript app bar has a bunch of built-in glyphs that you can use, and they are shown in Figure 6-6. Just set the icon property, and they'll work!

Figure 6-6:
A fine
selection of
glyphs.

previous	⏮	up	⬆	twopage	📄	zoomin	🔍	contact	👤
next	⏭	forward	➡	leavechat	💬	bookmarks	🔖	rename	🏷
play	▶	right	➡	mailforward	✉	document	📄	pin	📌
pause	⏸	back	⬅	clock	🕒	protecteddocument	🔒	musicinfo	🎵
edit	✍	left	⬅	send	✉	page	📄	go	➡
save	💾	favorite	★	crop	✂	bullets	☰	keyboard	🖱
clear	✖	camera	📷	rotatecamera	🔄	comment	💬	dockleft	📱
delete	🗑	settings	⚙	people	👤	mail2	✉	dockright	📱
remove	➖	video	📺	closepane	📄	contactinfo	👤	dockbottom	📱
add	➕	sync	🔄	openpane	📄	hangup	📞	remote	🖱
cancel	✖	download	⬇	world	🌐	viewall	🔍	refresh	🔄
accept	✓	mail	✉	flag	🚩	mappin	📍	rotate	🔄
more	⋮	find	🔍	previewlink	📄	phone	📞	shuffle	🔀
redo	↺	help	?	globe	🌐	videochat	💬	list	📋
undo	↻	upload	⬆	trim	✂	switch	🔗	shop	🛒
home	🏠	emoji	😊	attachcamera	📷	contact	👤	selectall	📋
							orientation	📱	

You can, of course, make your own icons, but why would you? The list of glyphs to use for icons is pretty comprehensive, and Windows 8–styled, free, and easy! Just use them!

Making commands contextual

Besides the app bar, you need to give your users more choices for invocation and navigation other than just clicking on buttons. Navigation should be contextual in order to complement the fast and fluid nature of Windows 8 design.

Touch is a totally different integration language and shouldn't mimic the mouse. As such, you should let the user use the content on the page and the gestures in the interaction library to help the content come alive.

What are the gestures in the GestureLibrary, you ask? Awesome question!

- ✓ **Tap:** One finger touches the screen briefly.
- ✓ **Press and hold:** One finger touches the screen and stays in place.
- ✓ **Slide:** One or more fingers touch the screen and move in the same direction.
- ✓ **Swipe:** One or more fingers touch the screen and move a short distance in the same direction.
- ✓ **Turn:** Two or more fingers touch the screen and move in a clockwise or counter-clockwise arc.

- ✓ **Pinch:** Two or more fingers touch the screen and move closer together.
- ✓ **Stretch:** Two or more fingers touch the screen and move farther apart.

To access all of these events, we use the `GestureLibrary`. The `GestureLibrary` is a poorly documented (at least for now) bit of code that is designed to integrate Windows 8 applications with the touch-enabled screens that will be on tomorrow's devices.



If you are coding on a laptop, the mouse-click gestures aren't enough to test `GestureLibrary`. A tap isn't the same as a click. If you want to give different results for those discrete events, you can use the `GestureLibrary` to discriminate. If not, the Windows 8 interpreter treats a tap as a click.

To use the `GestureLibrary`, you need to initialize the individual gestures in the `gestureSettings` property. Those initializations map pretty well, but not perfectly, to the gestures defined in the design language that I laid out in the preceding bulleted list. They look like this:

- ✓ **tap:** Touching an item
- ✓ **doubleTap:** Touching it twice
- ✓ **hold:** Touching until a configurable threshold is reached
- ✓ **drag:** Touch and move (this book is going to lose its PG rating if this keeps up)
- ✓ **crossSlide:** Dragging across something
- ✓ **manipulationRotate:** Slide in a circle
- ✓ **manipulationTranslateX:** Slide left and right
- ✓ **manipulationTranslateY:** Slide up and down
- ✓ **manipulationScale:** Drag fingers apart
- ✓ **manipulationRotateInertia:** The speed at which fingers are rotated
- ✓ **manipulationScaleInertia:** The speed at which fingers are moved together or apart
- ✓ **manipulationTranslateInertia:** The rate of speed at the end of a manipulation

Clearly many of these are for game development, such as when you might want to toss Twinkies at an armadillo at varying rates of speed. That's a little outside of the scope of this book, but I can imagine many applications that might want to use gestures for more mundane reasons. For instance, you might want to allow the user to change the opacity of an image by tapping on it.

In fact, let's do just that. In a new blank project, add an image to the default HTML file, like this:

```
<body>
  <div id="horizon">
    <div id="content">
      
    </div>
  </div>
</body>
```



Horizon and content directives help with centering. The CSS file looks like this:

```
#horizon
{
  background-color: transparent;
  text-align: center;
  position: absolute;
  top: 50%;
  left: 0px;
  width: 100%;
  height: 1px;
  overflow: visible;
  visibility: visible;
  display: block
}

#content
{
  background-color: transparent;
  margin-left: -125px;
  position: absolute;
  left: 50%;
  visibility: visible
}
```

In the `onactivated` event, get a reference to the image. Make sure you declare it in the function scope!

```
mainImage = document.getElementById("mainImage");
//After that, declare and initialize the GestureRecognizer.
recognizer = new Windows.UI.Input.GestureRecognizer();
recognizer.gestureSettings = Windows.UI.Input.
    GestureSettings.tap;
//Finally, add an event listener for the tapped event.
recognizer.addEventListener('tapped', that.tappedHandler);
//Elsewhere in your code base, add the tappedHandler.
that.tappedHandler = function (evt) {
    evt.target.changeOpacity();
};
```


And again, elsewhere in your code base, write the `changeOpacity` handler.

```
function changeOpacity() {  
  if (mainImage.getAttribute("style") ===  
      "filter:progid:DXImageTransform.Microsoft.  
        Alpha(opacity=50)") {  
    mainImage.setAttribute(null, "style",  
        "filter:progid:DXImageTransform.Microsoft.  
          Alpha(opacity=100)");  
  } else {  
    mainImage.setAttribute(null, "style",  
        "filter:progid:DXImageTransform.Microsoft.  
          Alpha(opacity=50)");  
  }  
}
```

Run the program, and you will get the image in the center of the screen. Tap on it (on a touch screen) and it turns to 50 percent opacity. Tap again and you return it to 100 percent.

This is about the simplest application of `GestureRecognizer` that I could come up with. It is very complex, and I imagine that partners will come out with better libraries. If you are planning on building a touch-heavy game, you might want to consider moving to DirectX and using C++, because the JavaScript support is rather weak.

Making the App Fast and Fluid

After the layout is right, and follows all of the rules laid out in Chapter 2, you need to make sure it's fast and fluid.

Snapping is something we have to deal with. The application needs to handle snap and scale views gracefully. It should have some view that makes sense in all three modes.

Speaking of changing modes, getting from one page or content point to another should be snappy too. Page- and content-level animations should be animated, with transitions that are available to all Windows 8 apps.

Animating Windows

The component animation that is inherent to HTML5 and even JQuery is not only supported in Windows 8, but also encouraged. The navigation model is new, however, and there are bits in WinRT that make it easy to animate.

Everything for page- and content-level animation (sometimes called Transitions) is covered by WinJS, and you can find it in `WinJS.UI.Animation`. Here is a sample of what you can get there:

- ✓ **Page transition:** Animates the contents of a page into or out of view.
- ✓ **Content transition:** Animates one piece or set of content into or out of view.
- ✓ **Fade in/out:** Shows transient elements or controls.
- ✓ **Crossfade:** Refreshes a content area.
- ✓ **Expand/collapse:** Shows additional inline information.
- ✓ **Reposition:** Moves an element into a new position.
- ✓ **Show/hide pop-up:** Displays contextual UI on top of the view.
- ✓ **Show/hide edge UI:** Slides edge-based UI into or out of view.
- ✓ **Show/hide panel:** Slides large edge-based panels into or out of view.
- ✓ **Add/delete from list:** Adds or deletes an item from a list.
- ✓ **Add/delete from search list:** Adds or deletes an item from a list when filtering search results.
- ✓ **Badge update:** Updates a numerical badge.
- ✓ **Start/end a drag or drag-between:** Gives visual feedback during a drag-and-drop operation.

That's a lot of animating. Don't feel like you have to animate everything to get into the Store. I would, however, seriously consider animating page and content transitions in order to show that you have given a nod to the realities of the Windows 8 design language.

Page-level transitions

Compared to dealing with the `GestureLibrary`, page transitions are a snap. In `WinJS.UI.Animation`, you find an `enterPage` function and an `exitPage` function. They both do about the same thing, but they just need to be run at different times in the page lifecycle.

Both methods take an element object and an offset object and return a promise. You can't act on an element while it's in transition, so you want to use the promise to delay any page functionality (like removing an element) until the animation is over.

```
WinJS.UI.Animation.enterPage(content, startingPosition).done();
```

The `content` is the element or elements that you are animating. This seems counterintuitive, but I believe it's set up this way so that you can choose to animate only certain parts of the page and still have content-level control later. The `enterPage` function is designed to be used with the navigation libraries I talk about in Chapter 3.

The `startingPosition` is exactly what it looks like. It's where the content collection should start before the animation occurs. You can pick a place on the page and pass in a JavaScript array of the location to make this happen. I recommend that you use `undefined` for the default start location unless you have a very specific need to start elsewhere. I haven't had to yet.

The `done`, of course, represents the promise that is being returned by the function. When the animation is done, the rest of the code will be run. The delay is to prevent the entire UI from locking up while an animation is being performed.

To set up an animated page sequence, try the following steps:

1. **Create a blank JavaScript Windows 8 project.**
2. **Make a new directory called Pages for the pages we are going to transition between.**
3. **Add three pages. Right-click the Pages directory and select Add New Item. Select Page Control.**

Do it three times. I named mine `firstpage`, `nextpage`, and `thepageafter`.

4. **In `default.html`, add a content `div` replacing the paragraph tag, like this:**

```
<body>
  <div id="content"></div>
</body>
```

In the `default.js` file, add a navigated event handler.

```
WinJS.Navigation.addEventListener("navigated",
    function (eventObject) {
        var url = eventObject.detail.location;
        var content = document.
            getElementById("content");
        WinJS.Utilities.empty(content);
        eventObject.detail.setPromise(WinJS.UI.Pages.
            render(url, content, eventObject.detail.
                state).then(function () {
                    WinJS.Application.sessionState.lastUrl =
                        url;
                }));
    });
```

5. Navigate to the first content page in the onactivated event.

This gets things started in the app.

```
app.onactivated = function (args) {
  if (args.detail.kind === activation.
    ActivationKind.launch) {
    args.setPromise(WinJS.UI.processAll()).
    then(function(){
      return WinJS.Navigation.navigate("/
        pages/firstpage.html");
    }));
  }
};
```

The firstpage.html, nextpage.html, and thepageafter.html give the header and main sections an ID.

6. While you're at it, add a button for navigation:

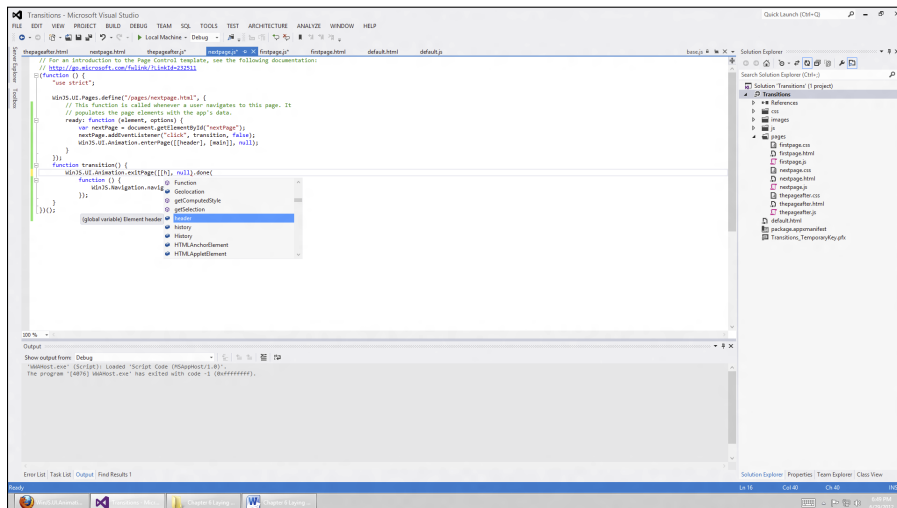
```
<body>
  <div class="firstpage fragment">
    <header aria-label="Header content"
      role="banner" id="header">
      <button class="win-backbutton" aria-
        label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to
        firstpage</span>
      </h1>
    </header>
    <section aria-label="Main content" role="main"
      id="main">
      <p>Content goes here.</p>
      <button id="nextPage">Next Page</button>
    </section>
  </div>
</body>
```

7. Add a transition function to firstpage.js, nextpage.js, and thepageafter.js that navigates to the chosen page and uses the animation:

```
function transition() {
  WinJS.UI.Animation.exitPage([[header],[main]],
    null).done(
    function () {
      WinJS.Navigation.navigate("/pages/
        nextpage.html");
    });
}
```

Here's something awesome: The content entity (`[[header], [main]]` in this example) has IntelliSense, which is a Visual Studio feature that auto-completes code you are typing in. You can see it in Figure 6-7 next to the cursor where I'm starting to type **header**. This makes it a lot easier to get the right IDs in a complex page.

Figure 6-7:
IntelliSense
in the
exitPage
function.



The navigation endpoint is different for each page. It should navigate to the page you want to go next.

8. Finally, in the `firstpage.js` (and the others), change the `ready` event to fire the new transition function when the `click` event occurs.

```
WinJS.UI.Pages.define("/pages/firstpage.html", {
    // This function is called whenever a user
    // navigates to this page. It
    // populates the page elements with the app's
    // data.
    ready: function (element, options) {
        var nextPage = document.
        getElementById("nextPage");
        nextPage.addEventListener("click",
            transition, false);
        WinJS.UI.Animation.
        enterPage([[header], [main]], null);
    }
});
```

I wish I had a pretty picture of this to show you, but my publisher *still* hasn't implemented those animated pages in printed books. If you can't get it to work, download the sample code and give that a try. See this book's Introduction for more on how to download the sample code. There is also a fantastic sample (if a little overcomplicated) on the <http://msdn.microsoft.com/en-us/windows/samples> site.

Animating content

Good news! Animating content is almost exactly the same as animating entire pages. The Animation library has `enterContent` and `exitContent` functions that work a lot like `enterPage` and `exitPage`.

```
WinJS.UI.Animation.enterContent(content,  
    startingPosition).done()
```

The content object is an HTML element (the ID of said object) that you want to animate. The `startingPosition` is best set to null unless you have a specific starting position. The `promise` returned is just to make sure that you don't do anything with the entity until the animation is finished.

I'll spare you the step-by-step instructions — it looks a lot like the previous example of animating pages. Just put the `enterContent` in the `load` function, and the `exitContent` in the `navigation` event.

Snapping and scaling

When your application is running in a normal mode, it fills the whole screen. What isn't of immediate importance to a developer in the early stages of an app is handling the other two states that an app may find itself in: snapped and filled.

Planning for views

In Chapter 2, I cover the importance of snap and scale view. The user expects that two apps should be able to be used in conjunction — task list and calendar, e-mail and contacts, recipes and shopping lists, or whatever. Your app should be prepared to handle all of these form factors if it makes any sense at all.

Implementing the views in Chapter 2 is a matter of deciding what your app should look like in various frames, building separate views for each look, and then using CSS media queries to tell Windows 8 when to show what.

I leave doing the design work and laying out the pages as an exercise for the reader. This book has covered how to design in Windows 8 and how to make new views. For your current purposes, you need to have two principle functions — media queries for `resize` and the `resize` event in WinRT.

Using media queries

Media queries are labels placed in the CSS file that the browser can use to make decisions about what styles to use. If you need a primer on media queries, I recommend *HTML5 For Dummies*, by Andy Harris (published by John Wiley & Sons, Inc.). It's a pretty big topic, and we have more than enough Windows 8 stuff to cover here. They are neat, and both Microsoft and I recommend their use.

Microsoft has provided a number of browser messages for the current layout of the app. (Remember, the Windows 8 sandbox is just IE10.) They come in the `-ms-view-state` property and include

- ✓ **fullscreen-landscape:** Your app covers the whole screen and the width is bigger than the height.
- ✓ **fullscreen-landscape:** Your app is covering the whole screen and the height is bigger than the width.
- ✓ **filled:** The app is taking up two-thirds of the screen.
- ✓ **snapped:** The app is taking one-third of the screen.

Using these media types allows you to use CSS to style the views in a custom layout for each view. It looks something like this in the CSS file:

```
@media (-ms-view-state: fullscreen-landscape)
{
    /*layout for landscape view */
}
@media (-ms-view-state: fullscreen-landscape)
{
    /*layout for portrait view */
}
@media (-ms-view-state: filled)
{
    /*layout for filled (2/3 of the screen) */
}
@media (-ms-view-state: snapped)
{
    /*layout for snapped (1/2 of the screen) */
}
```

Using the resize event

The other option for layout is to actually change the view when a resize occurs. This requires that you handle the `resize` event, which is doable because the app must be active to resize it.

You can actually add an `EventListener` to the window object and change details of the view — or even the page itself — if the view state changes. The code looks like this:

```
window.addEventListener("resize", function (e) {
    var currentViewState = Windows.UI.ViewManagement.
        ApplicationView.value;
    var snapped = Windows.UI.ViewManagement.
        ApplicationViewState.snapped;

    if (currentViewState === snapped) {
        that.listView.layout = new WinJS.UI.ListLayout();
    }
    else if (lastViewState === snapped && currentViewState
        !== snapped) {
        that.listView.layout = new WinJS.UI.GridLayout();
    }

    lastViewState = currentViewState;
});
```

This is not the recommended method according to Microsoft, but for me, it makes the most sense. I am not a CSS wizard, so handling the changes in code really seems to fit. It gives a lot of control and allows you to actually give different functionality in different views. It's up to you and your app style.

When to not provide view-state change

I can imagine apps that don't require a view-state-change function. Games, for instance. Just make sure the user can pause and leave it at that. Content production tools also might not benefit from a smaller version.

You have to consider your audience and make the right decision. No app so far has been rejected for not having a snapped or filled view, but it is on the certification list, so take it under advisement.

Chapter 7

Presenting Data

In This Chapter

- ▶ Getting data on the screen
 - ▶ Using JavaScript to populate content
 - ▶ Making use of the WinJS binding tools
 - ▶ Grouping and sorting
-

At its core, Windows 8 is about giving the user access to their data in a beautiful way. Microsoft has a bunch of fancy guidelines and whatnot that I talk about in Chapters 2 and 6, but if you don't give the user value, that beauty goes for naught.

HTML5 is less about data than it should be. There are really no good structures for data presentation in the HTML toolset other than the Table, and that is fraught with layout issues. HTML is great for getting stuff on the page, but it just doesn't have a databinding superstructure.

WinJS has fixed that. The `WinJS.Binding` namespace provides a way to mark up HTML `divs` with data attributes in either JavaScript or HTML. You can configure single one-time bindings or group bindings for lists. It offers sophisticated sorting and grouping. It even offers special controls that take advantage of touch and the tablet form factor.

Showing Single Bits of Data

WinJS offers you two different ways to get data to the user. Both of them involve some field in HTML that can hold text or an image, and some value in JavaScript that can be used to populate the field. You can use the DOM just as you would in a browser application, or you can use the cool new `WinJS.Binding` namespace.

Setting single values using the DOM

In classic web browser fashion, you can get a reference to any entity in the HTML document object model (DOM) and change a property. Sometimes, simpler is better, right?

If you know JavaScript, this will be no surprise to you. In the HTML, just set up a `div` (or a `H1` or an image or whatever you want) and give it an ID. I called my `div` `PlaceToPutText`.

```
<div id="PlaceToPutText"></div>
```

In the JavaScript file, in the ready event handler, I am going to do two things. I'm going to get a reference to that `div`, and I'm going to set the `innerText` property.

```
ready: function (element, options) {  
    var placeToPutText = document.getElementById("PlaceToPutText");  
    placeToPutText.innerText = "Testing 1 2 3";  
},
```

This is not databinding. This is just modifying the DOM with JavaScript, just like I've been doing since 1995. It's an effective way to get data on the screen and you'll use it all the time, but that's not what you're about here. You want to bind some data!

jQuery and Windows 8

Fortunately, jQuery works in Windows Store apps, just as it does in IE10. All you need to do is download the jQuery file, drop it in your .js folder, and then reference it in the HTML.

After that, you can just do stuff the same way you would in a web app. Need to change the value of `PlaceToPutText`? Just use the dollar symbol.

```
$("PlaceToPutText").innerText  
= "Testing 4 5 6"
```

That's all there is to it.

For more on jQuery, go to www.jquery.com, or check out Lynn Beighley's excellent *jQuery For Dummies* (published by John Wiley & Sons, Inc.).

Using the declarative binding in WinJS

Most Microsoft products have had databinding as part of their process. Back in Active Server Pages, there was poor ActiveX black-box binding that no one used. ASP.NET web forms had very good databinding that depended on the page's view state. Of course, most Windows programming depended on binding because there was no direct object access to the forms themselves.

Windows 8 programming is no different. XAML has its own binding methodology, which is carried over from Windows Presentation Foundation. For the HTML5 side, the JavaScript team had to come up with something that made sense, and I think they did. The `WinJS.Binding` namespace has it right.

Binding to a field

At its most straightforward, databinding ties a value in JavaScript to a field in HTML. You use databinding when you want the values in the HTML to change at the same time the values in the JavaScript change, without having to constantly reset everything using the DOM.

Binding starts simple and gets really complex. Most significantly, databinding requires data, not just a variable.

Note that at the base level, any content-level DOM element can be bound, and all it needs is a `data-win-bind` parameter. Try this out to see what I mean:

1. **Add a pagecontrol to your project. I called mine `basicBind`.**
2. **Use the same HTML you used in the section, “Setting single values using the DOM.”**

Just add a `data-win-bind` parameter that specifies the `innerText` property of the element that should be bound and to which JavaScript variable.

```
Testing <span id="PlaceToPutText" data-  
win-bind="innerText: testvalue"></span>
```

3. **Create a new JavaScript array to store some data.**

```
var testing = { testvalue: 7 }
```

4. **Get a reference to the span and then pass the array and the span to `processAll`.**

```
var placeToPutText = document.getElementById("PlaceTo  
PutText");  
WinJS.Binding.processAll(placeToPutText, testing);
```

This isn't much better than what you did before, though. You still have to call `processAll` every time the data changes because it's a one-time binding. By itself, a JavaScript object doesn't have the capability to update the user interface. You need a binding source.

Creating a binding source is easy. Just pass the JavaScript object into `WinJS.Binding.as`, and then pass that into `ProcessAll`. Now when the values change, the UI changes, too. Here is what my `basicBind.js` file looks like when I'm done:

```
(function () {  
    "use strict";  
    var testing = { testvalue: 7 }  
    var testingSource = WinJS.Binding.as(testing);  
    WinJS.UI.Pages.define("/pages/basicBind.html", {  
        ready: function (element, options) {  
            var placeToPutText = document.getElementById("PlaceToPutText");  
            WinJS.Binding.processAll(placeToPutText, testingSource);  
        }  
    });  
})();
```

Any function that changes the data in the testing array now causes the data to update. You just need to remember to change the binding source instead of the underlying object. The binding source handles change in both the UI and the underlying object.

You can test this by adding a button that increments the `testvalue` parameter of the testing object. This simulates the effect of the value being changed by a user event. Actually, it *is* the effect of a value being changed by a user event.

Just add a button to the HTML, like this:

```
<button id="increment"></button>
```

Then tack on an event listener that incrementally modifies the binding source. Here is my new `.js` file:

```
(function () {  
    "use strict";  
    var testing = { testvalue: 7 }  
    var testingSource = WinJS.Binding.as(testing);  
    WinJS.UI.Pages.define("/pages/basicBind.html", {  
        ready: function (element, options) {
```

```

        var incrementButton = document.
        getElementById("increment");
        increment.addEventListener("click",
        incrementValue);
        var placeToPutText = document.getElementById("
        PlaceToPutText");
        WinJS.Binding.processAll(placeToPutText,
        testingSource);
    }
});
function incrementValue() {
    testingSource.testvalue++;
}
})();

```

Changing the data

In WinJS, databinding is one-way. This means that while the data is bound to show on the UI, you need to do something else if the user wants to change the data.

The simplest way to do this is to have an event listener on the element with the data that can change. In the preceding binding example, I used a span, but what if it were a text box?

```

Testing <input type="text" id="PlaceToPutText" data-win-
        bind="value: testvalue"></input><br />
<button id="increment"></button>

```

Now you can attach an event listener to the text box and use that to update the binding context. Here is the final JavaScript file:

```

(function () {
    "use strict";
    var testing = { testvalue: 7 };
    var testingSource = WinJS.Binding.as(testing);
    WinJS.UI.Pages.define("/pages/basicBind.html", {
        ready: function (element, options) {
            var incrementButton = document.
            getElementById("increment");
            increment.addEventListener("click",
            incrementValue);
            var placeToPutText = document.getElementById("
            PlaceToPutText");
            placeToPutText.addEventListener("change",
            updateData);
            WinJS.Binding.processAll(placeToPutText,
            testingSource);
        }
    });
}());

```

```
    }  
  });  
  function incrementValue() {  
    testingSource.testvalue++;  
  }  
  function updateData() {  
    var placeToPutText = document.getElementById  
      ("PlaceToPutText");  
    testingSource.testvalue = placeToPutText.value;  
  }  
  })();
```

If you run the app, change the value, and press the button, you can see that the increment adds to the value that you set. I show that in Figure 7-1. And if you change over to the debugger, you can check the underlying data and see that it is staying in sync.

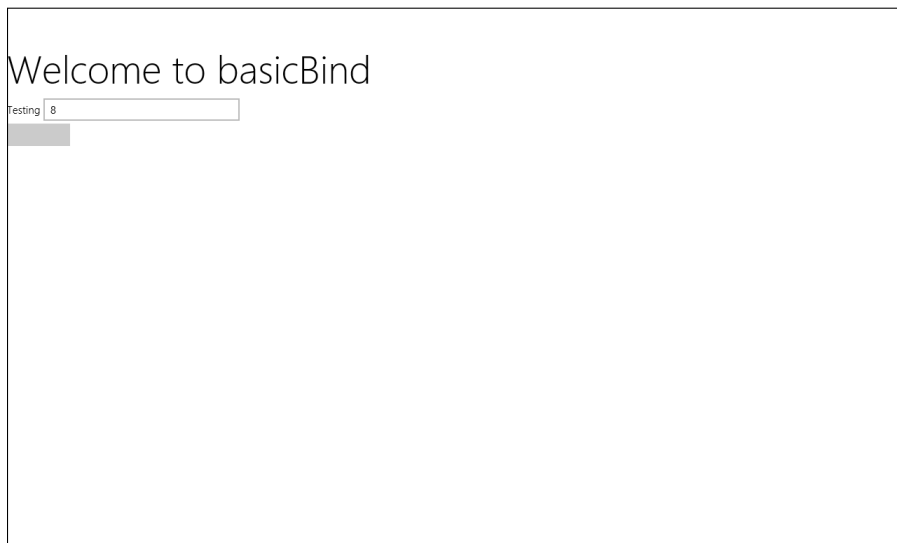


Figure 7-1:
Changing
the underly-
ing data
with a bind-
ing source.

Listing Data

Single fields are important, but not everything. You, at some point, need to bind to a collection of objects. For instance, list/detail applications need to show the user the list before they show the user the detail. The details are all single-field binding, which uses the techniques described in the previous section. The list needs something a little more complex.

In the basic example in Chapter 4, I did some really simple databinding to a `ListView`. That was, more or less, the essentials of binding to a collection of objects. Here I go into, er, detail (excuse the pun). The goal here is to give you the tools that you need to apply this to any situation you run across.

Dealing with collections of objects

In JavaScript, anything can be an object. You can declare any variable to be an object and just randomly assign variables to it:

```
var person = new Object();
person.name = "Bill";
person.age = 41;
person.isWriting = true;
```

The problem with this is you can't easily describe more than one entity in the collection. It is possible with an array of objects, but it quickly gets cumbersome.

Usually, in Windows 8, I use JavaScript Object Notation format, or JSON, to build out objects. This starts as text, and then JavaScript just treats it like arrays in the code.

```
var person = [
{
    name: "Bill",
    age: 41,
    isWriting: true
}]
```

Now, if I need more than one person, I can just declare them in text when I set up the object.

```
var people = [
{
    name: "Bill",
    age: 41,
    isWriting: true
},
{
    name: "Gabrielle",
    age: 37,
    isWriting: false
}];
```

Here you have a collection, and it can be presented in text or as an object. To represent it as text, use `JSON.stringify()`. To represent it as an object, use `JSON.parse()`.


```
var peopleArray = JSON.parse(people);  
var peopleBackToString = JSON.stringify(peopleArray);
```

That is the kind of entity to use to make a user interface for a list.

Using the ListView

`WinJS.UI.ListView` is the construct that is going to let us put our collection on the screen for people to use. Along with `WinJS.UI.Template`, you can describe how one object looks in the collection, and how it should be handled in a group.

Templates

To describe these two elements, you can set up two `div`s on the target HTML page and let Windows 8 stitch them together. For a simple example, look at how it works in `POINTtodo`, where you need two columns of tasks. Figure 7-2 shows a picture of `POINTtodo`'s main page.

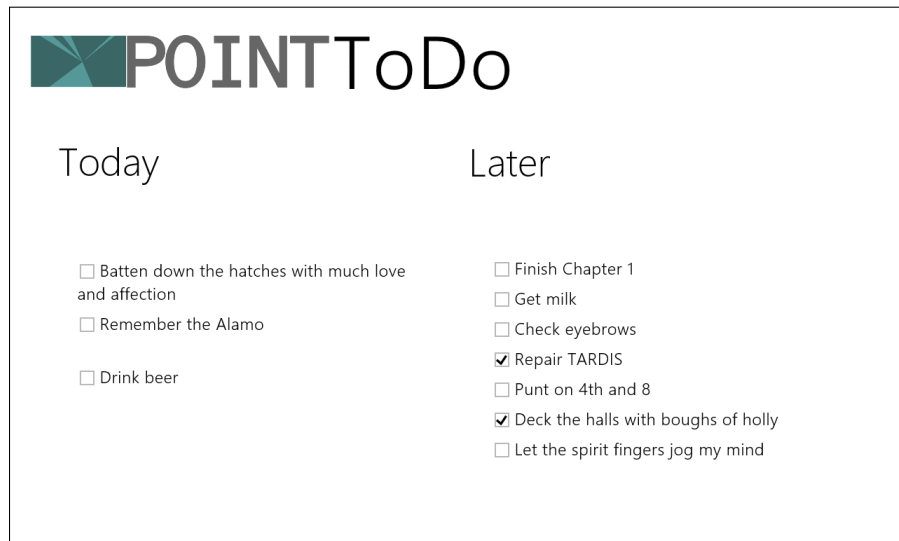


Figure 7-2:
The output
of the tem-
plates on
POINTtodo.

In each of the columns, you need a check box and a label. You can use the same template to describe each of these. Note that the `data-win-control` parameter is used in the top-level `div` to tell Windows 8 to make sure it doesn't randomly show up on the page.

```
<div id="listTemplate" data-win-control="WinJS.
    Binding.Template">
    <div style="width: 480px">
        <input type="checkbox" data-win-bind="checked:
            complete;" /><label data-win-bind="innerText:
                task" style="font-size: 18pt"></label>
    </div>
</div>
```

So that's a line item in the binding. You've described the components on the screen, and now you need to tell Windows 8 how to repeat them.

You do this with another set of divs that are the actual ListView. They have their own styling and declare two important data-win-options: `itemDataSource` and `itemTemplate`.

```
<div id="todayListview" data-win-control="WinJS.
    UI.ListView"
    data-win-options="{itemDataSource :todayList.
        dataSource, itemTemplate: listTemplate}"
    style="margin-left: 5%; margin-right: 5%;
        width: 40%; left: 0px; top: 0px;">
</div>
<div id="laterListview" data-win-control="WinJS.
    UI.ListView"
    data-win-options="{itemDataSource :laterList.
        dataSource, itemTemplate: listTemplate}"
    style="margin-left: 5%; margin-right: 5%;
        width: 40%; left: 592px; top: -403px;">
</div>
```

Why two ListViews? Two lists. Take another look at Figure 7-2. If you are going to present two different data sets, you need two ListViews. If you are going to present the data two different ways, you need two templates.

Binding the data

In the HTML, you defined the data source and the template in the `data-win-options`. The template is fairly self-explanatory: It points to the `WinJS.UI.Template` div. The data source is less so. Where does that come from?

Not surprisingly, you define that in the JavaScript. The process for setting up a data source is the same every time:

- 1. Get your data. Could be a service call, or it could be in a file. You might even make it yourself!**

Either way, it needs to be in an array.

```
var todayArray = [
{
    task: "Batten down the hatches with much love and
    affection",
    complete: false,
},
{
    task: "Remember the Alamo",
    complete: false,
},
{
    task: "Drink beer",
    complete: false,
}
];
```

2. Declare a `WinJS.Binding.List` with the data set as the parameter.

```
var todayList = new WinJS.Binding.List(todayArray);
```

Wait, that's it? Two steps? What's the world coming to?

Fact is, that's pretty much it. There are more details out there, but the binding that makes `POINTtodo` run starts and stops there. Make a template; get some data; bind.

Fleshing out the details

In the HTML, you set some `data-win-options`. Those are properties of the `WinJS.UI.ListView` object — the same things that are defined as part of the template.

In the HTML or the JavaScript, you can grab the `ListView` object and set some of those properties to change the behavior of the list. This is how to handle the details of presenting groups of information. Table 7-1 describes some of the angles you have control over.

Table 7-1 Options for the `ListView`

<i>ListView Property</i>	<i>Details</i>
<code>pagesToLoad</code>	Tells Windows 8 how many pages to load in the background.
<code>automaticallyLoad Pages</code>	Sets when the user scrolls into the end of the list, if the next set of pages is automatically loaded.
<code>itemDataSource</code>	Sets the data source.

<i>ListView Property</i>	<i>Details</i>
itemTemplate	Sets the template. This is what you did in the previous example.
currentItem	Sets what item is selected when the user views the page.
indexOfFirstVisible	It is possible to set what the first item is in the list that the user interacts with. Slick functionality.
indexOfLastVisible	You can post the last visible item, too.
layout	You can give a <code>ListLayout</code> or <code>GridLayout</code> object to the <code>ListView</code> to control the look of the list.
loadingBehavior	Determines how many items are available to the DOM.
scrollTop	Sets how far the scroll bar is from the content.
selection	Of course you can set the currently selected item.
selectionMode	A surprisingly important property. You can determine how the user can select items in the <code>ListView</code> . You can set to none (nothing can be selected), single (one item can be selected) or multi (several or even all items can be selected).
swipeBehavior	Amazing enough, this sets up how the list will behave when a user swipes his finger along it.
tapBehavior	Same deal — handles the tap.
zoomableView	Gives the UI a tip that the control accepts the pinch gesture for zoom. I talk about this more in the “Working with Groups Using Semantic Zoom” section later in this chapter.

So how does the `ListView` change things? Well, for `POINTtodo`, I wanted check boxes because it is a task list, and task lists should have check boxes. But say that I didn’t? Say I wanted to have the `ListView` handle the selection and take care of the user interaction. I could do that with a `div` and the `selectionMode` property.

```
<div id="todayListview" data-win-control="WinJS.
  UI.ListView"
  data-win-options="{itemDataSource :todayList.
    dataSource, itemTemplate: listTemplate,
    selectionMode: multi}" style="margin-left: 5%;
    margin-right: 5%; width: 40%; left: 0px; top:
    0px;">
</div>
```

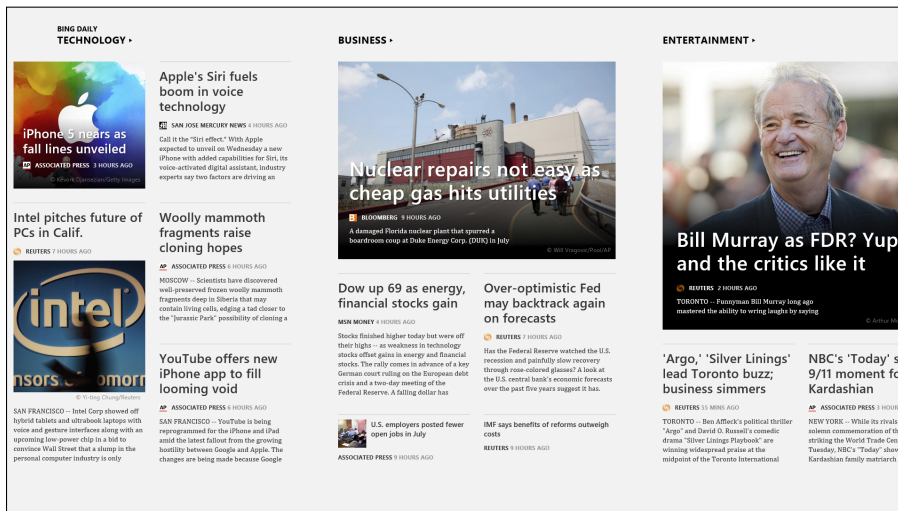
That gives the user the experience of selecting the task in the `ListView`, rather than clicking the check box. It’s just a matter of user experience design, and it’s up to you. For my instance, I needed a check box to give that task list perspective. For other applications, you might be better off to have a total item selection.

Show us the grid

The `layout` property of the `ListView` accepts an object that can be a `WinJS.UI.ListView` or `WinJS.UI.GridView`. This is how you describe the various layouts that I talk about in Chapter 2.

Clearly the gridview is designed more for images, although the news application in Windows 8 (in Figure 7-3) gets away with using text, so there are times for it.

Figure 7-3:
Discourse
makes text
in a grid
look good!



FlipView

The `WinJS.UI.FlipView` is a full-screen version of the `ListView`. I recommend you bind it to a collection. When bound to a data source, the items will show full screen, with the ability to flip between them.

Binding the `FlipView` looks exactly like binding a `ListView`. The only difference is the `HTML` tag itself, which obviously has the `data-win-control` parameter of `WinJS.UI.FlipView`.

```
<div id="theFlipView"
      data-win-control="WinJS.UI.FlipView"
></div>
```

Why they decided to make `FlipView` a separate control and `GridView` a layout type, I'll never know, but that's how it is. Seems like it should be one or the other.

Anyway, the FlipView, like the ListView, has a nice collection of properties, listed in Table 7-2, that alter the performance and formatting of the control. You set them the same exact same way, with `data-win-options`.

Table 7-2 FlipView Properties	
Property	Description
itemDataSource	Just as with ListView, this property sets the collection of objects.
itemTemplate	The layout that will be used for each item.
currentPage	Sets which page the FlipView is on. Great for navigation.
orientation	Set up the portrait or landscape formatting here.

Grouping, Sorting, and Selecting

Getting data into groups is an important and neat feature of the ListView. A lot of things go with grouping, though. Sorting, selecting, and showing the user what they want to see are all important.

Group control

So here are two things that you need to know about grouping in WinJS:

- ✓ You can only do grouping with a layout using the `GridView` layout.
- ✓ Setting up grouping uses two data sources — one with the data, and one with the group. That being said, they can both be from the same root data.

The data you've used for binding up till now is the data sources in POINTtodo, one of my Store apps. This isn't the best example for grouping, but because it's familiar, give it a try anyway:

1. Make a dataset that has a groupable property.

A *groupable property* is something that is repeated in the set that makes an interesting grouping. The classic example is using the first letter of the object name and grouping by alphabet. For the task list example, you can do something different like put the Today tasks and the Later tasks in the same set, and then give them a `WhenToDo` property that makes them sortable:

```
var laterArray = [
{
  task: "Finish Chapter 1",
  whenToDo: "today",
  complete: false
},
{
  task: "Get milk",
  whenToDo: "today",
  complete: false
},
{
  task: "Check eyebrows",
  whenToDo: "today",
  complete: false
},
{
  task: "Repair TARDIS",
  whenToDo: "today",
  complete: true
},
{
  task: "Batten down the hatches with much love
and affection",
  whenToDo: "later",
  complete: false
},
{
  task: "Remember the Alamo",
  whenToDo: "later",
  complete: false
},
{
  task: "Drink beer",
  whenToDo: "later",
  complete: false
}
];
```

2. After you have a binding list, you discover that there is a property of the list for grouping — the `createGrouped`. It requires three things:

- The field that will be grouped upon, or the key.
- The data that makes up the grouping set — in our case, just Today and Later.
- A function for the sorting of items. That's not the most interesting thing in this example, which has two members in the sorting group.

```
function sortGroup(leftKey, rightKey) {
    return leftKey - rightKey;
}

function getGroupKey(dataItem) {
    return dataItem.whenToDo;
}

function getGroupData(dataItem) {
    return {
        title: dataItem.whenToDo
    };
}

var groupedItemsList = itemsList.
    createGrouped(getGroupKey, getGroupData,
        sortGroup);
```

3. Set up a ListView that uses the Grid layout and has both an item DataSource and a groupDataSource.

```
<div id="groupedListView"
    data-win-control="WinJS.UI.ListView"
    data-win-options="{itemDataSource:groupedItems
        List.dataSource,
        groupDataSource:groupedItemsList.groups.
        dataSource,
        layout: {type: WinJS.UI.GridLayout}}">
</div>
```

That's really all there is to it. You just need to give WinJS another data source for the grouping, and it takes care of things for you. The same list item template works for you as worked in the ungrouped view.

Single and multiselect

In POINTtodo, I use a check box and label to have the user select the task items. It was done that way because it is more task list-like.

Usually, though, you can use the `SelectionMode` of the `ListView` to let the user select items in the view. The `SelectionMode` allows for three kinds of selection, as shown in Table 7-3.

Table 7-3 Selection Modes		
<i>Member</i>	<i>Value</i>	<i>Description</i>
none	none	Items are not selectable. This is what I use in POINTtodo because it has check boxes.
single	single	Allows the user to select one item at a time, like a radio button.
multi	multi	Allows the user to select multiple items by clicking more than one item.

I could have used `SelectionMode` for POINTtodo. Figure 7-4 has a picture of how it might look.

Built-in animations

A large collection of animations can be used in Windows Store apps — some of them are discussed in Chapter 6. The `ListView` is especially set up for animation because the users are used to seeing sliding content when list collections are used.

The property of the `ListView` that sets animations is the `ListViewAnimationType`. There are two animation types:

- ✓ **entrance:** Shown when the `ListView` is first displayed
- ✓ **contentTransition:** Used for changing content

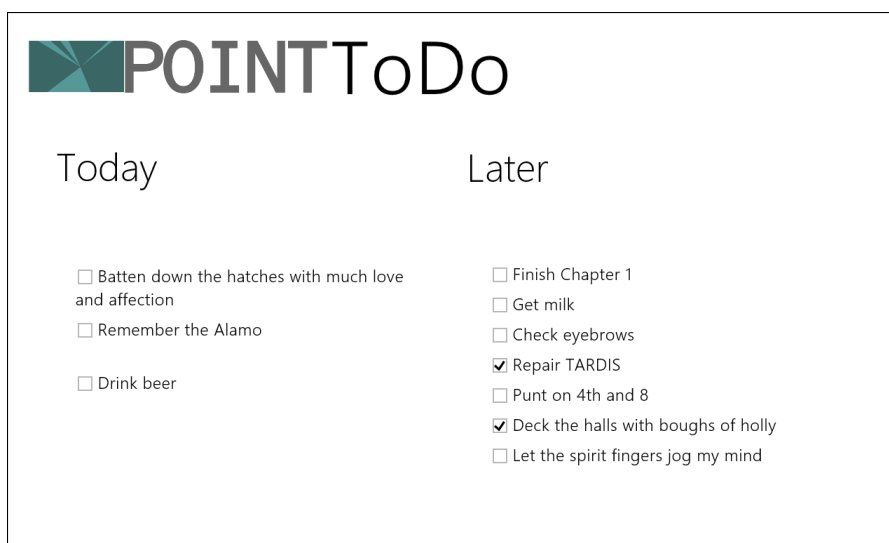


Figure 7-4:
Using multi-
select to do
POINTtodo.

You can set the property in the `data-win-properties` in the HTML or in the JavaScript. The JavaScript is just an array setting, like this:

```
var WinJS.UI.ListViewAnimationType = {  
    entrance : "entrance",  
    contentTransition : "contentTransition"  
}
```

Windows 8 handles the details for you.

Working with Groups Using Semantic Zoom

A control is built into Windows 8 to help the user (and the developer) with management of groups. *Semantic Zoom* is a touch-aware user interface feature that allows the user to “back up” from the data and see the overlying group structure as a data set of its own.

The best example of this that I have seen in any app so far is right on the Start screen. Press the Windows key to go to the Start screen and then use the pinch gesture on the screen. You’ll see something like Figure 7-5, where the groups of times become tiles of their own.



You can zoom in and out with `Ctrl+-` and `Ctrl++` in the mouse and keyboard world too.

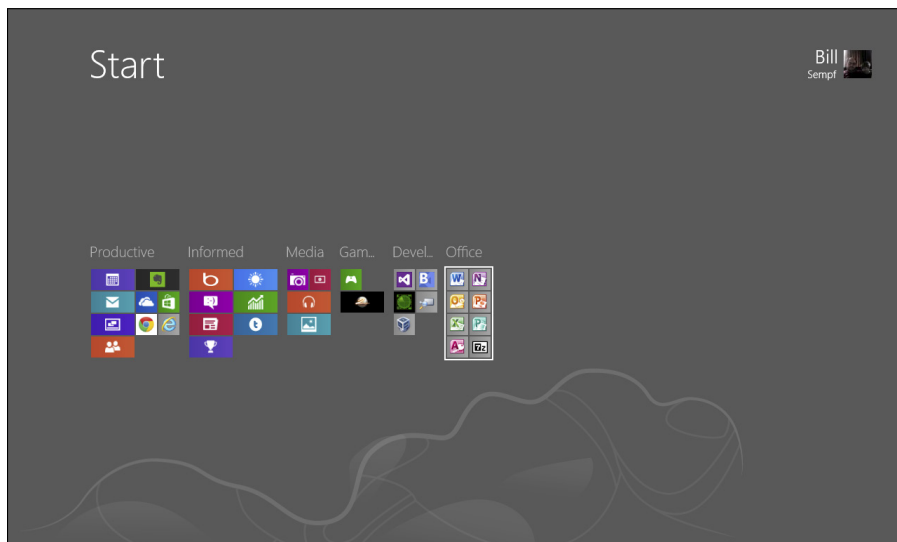


Figure 7-5:
The Start
screen after
zooming out.

Figure 7-5 is my Start screen; yours, of course, will differ. No matter, this is where you can do two vital things related to group management — move them and name them. Here, I can flick over a group to select it, and then touch Name Group to give the group a name. You can also just touch and hold and change the order of groups.

So that's what you're looking at here — almost a meta-management feature of Windows. Fortunately, it's built right into WinRT so you can get to it easily from your apps.

Using Semantic Zoom

Semantic Zoom is used when you can give the user an overview of data that makes sense. The POINTtodo example that I used previously doesn't really apply that much here because there are only two groups and the group view isn't that interesting.

For a large number of apps, using semantic zoom makes sense. For instance:

- ✓ Anything that can be organized by alphabet, like a contact list. The zoom-in view would consist of the individual contacts, and the zoom-out view would consist of the letters of the alphabet, and perhaps a number representing the quantity of contacts in each group.
- ✓ A stock market app with assets grouped by market segment. The zoom-in view shows the stock details, and the zoom-out view shows the segment, perhaps green for net gains and red for net losses.
- ✓ A product catalog, grouped by category. The zoom-in view shows product details and the zoom-out view shows tiles with each category listed. You could even size the tiles based on how many products are in each category.
- ✓ A network management app with devices organized by type. The zoom-in view shows a list of device details, and the zoom-out view shows a collection of device types. You could also group by physical location, company department, or even subnet, but only one grouping is supported at a time. In Figure 7-6, I have laid out the grouping by device type.
- ✓ Library books grouped by the Dewey Decimal System. The zoom-in view shows a list of books, and the zoom-out view shows the hundreds level of the book's number. This example shows how even with a set group, you might have to make some decisions. You could group by the hundreds, or the tens, or even the ones column of the Dewey Decimal Number.

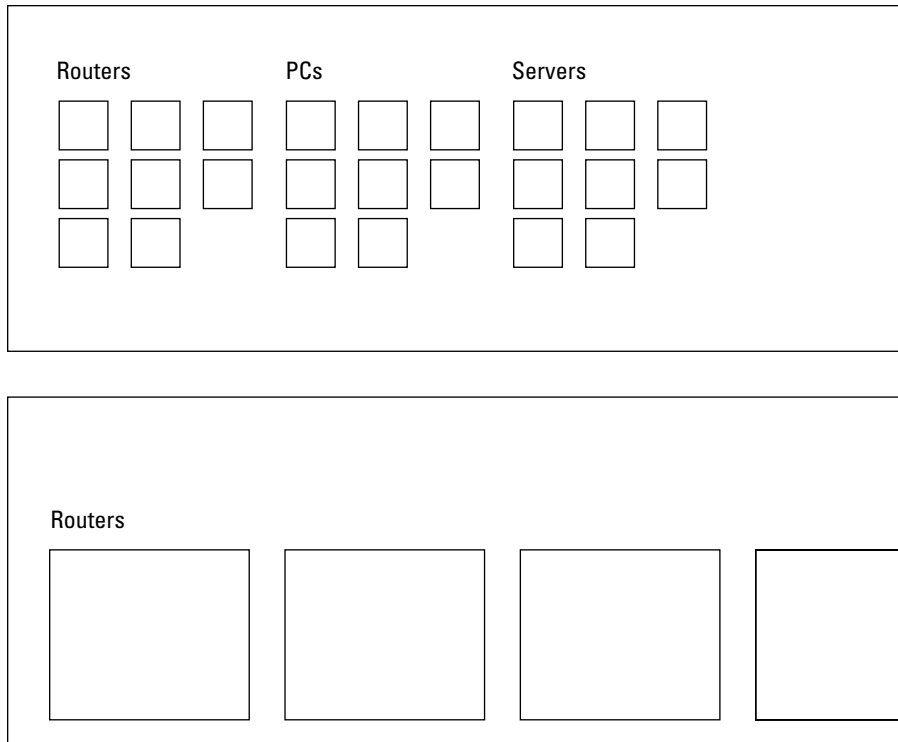


Figure 7-6:
Pan, Zoom,
and Detail
views of
a network
device list.

Also in Figure 7-6, notice the Detail view. Semantic Zoom is effectively adding a third layer to the traditional list/details paradigm of dealing with datasets. This is done to prevent the page view that is traditionally associated with lists of data.

Consider Amazon. Instead of “Here are the first 20 books, here are the next 20 books” from a search result, in the Windows 8–style app, you would search and then be brought to a list that could be zoomed out to the group view of the book title alphabetically. Then you only have 26 things to sort through and can look at the groups one at a time.

When designing for Semantic Zoom, you should keep a few guidelines in mind:

- ✓ **Remember size and space:** Make the zoom area big enough to use the pinch gesture effectively, and make sure zoomed-out view items are big enough to touch.
- ✓ **Don’t change the overall flow of the app when you change from zoom to pan view:** For instance, if your list scrolls left to right, the Semantic Zoom view should also.

- ✓ **Make sure the grouping makes sense to the user:** Use names for the groups and look for a good grouping theme.
- ✓ **As with other Windows 8 designs, don't use borders for the Semantic Zoom control or its items:** It just doesn't look good.

Considering the technical details

Setting up Semantic Zoom in a project requires two things — building grouped data just like you did in “Grouping, Sorting, and Selecting,” earlier in this chapter, and setting up the `WinJS.UI.ListView` with two formats for the panned and zoomed view.

1. Get your dataset and make a `WinJS.Binding.List` from it.

```
var data = [
  { 'name': 'Castillo de Rumba', 'address': '334
    Main Street', 'category': 'Italian' },
  { 'name': "Bob's Diner", 'address': '12 Town
    Street', 'category': 'Diner' },
  { 'name': 'Italian Feast', 'address': '503
    Rich Street', 'category': 'Italian' },
  { 'name': 'Hamburger Palace', 'address': '44
    State Street', 'category': 'American' },
  { 'name': 'Hot Dog Palace', 'address': '45
    State Street', 'category': 'American' },
  { 'name': 'Taj Palace', 'address': '1337
    Hacker Street', 'category': 'Indian' },
  { 'name': 'Spaghetti Warehouse', 'address':
    '10 Mound Street', 'category': 'Italian' },
  { 'name': 'Dal Restaurant', 'address': '994
    Lambert Street', 'category': 'Indian' },
  { 'name': 'Sandwiches by Joe', 'address': '440
    Main Street', 'category': 'Diner' },
  { 'name': 'Bar on Fifth', 'address': '12 Fifth
    Street', 'category': 'Diner' },
  { 'name': 'The Sub Shop', 'address': '229 E
    Broad Street', 'category': 'American' },
  { 'name': 'Foor Deng', 'address': '289 Main
    Street', 'category': 'Indian' },
  { 'name': 'Soups and More', 'address': '26
    Columbus Street', 'category': 'American' },
  { 'name': 'Beer Shack', 'address': '12 Norton
    Street', 'category': 'Diner' },
  { 'name': 'House of Food', 'address': '998
    Circle Street', 'category': 'American' },
]
var dataList = new WinJS.Binding.List(data);
```

2. Set up the grouping functions and make a `groupedItemsList` from the data.

```
function compareGroups(leftKey, rightKey) {
    return leftKey.charCodeAt(0) - rightKey.
        charCodeAt(0);
}

function getGroupKey(dataItem) {
    return dataItem.category;
}

function getGroupData(dataItem) {
    return {
        category: dataItem.category
    };
}

var groupedItemsList = dataList.
    createGrouped(getGroupKey, getGroupData,
        compareGroups);
```

3. Set up the two `ListView`s in the HTML to support zoomed-out and zoomed-in views.

```
<div id="zoomedInListView"
    data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemDataSource:
        groupedItemsList.dataSource,
        itemTemplate: select('#mediumListIcon
        TextTemplate'), groupHeaderTemplate:
        select('#headerTemplate'), groupDataSource:
        data.groupedItemsList.groups.dataSource,
        selectionMode: 'none', tapBehavior: 'none',
        swipeBehavior: 'none' }"
></div>

<div id="zoomedOutListView"
    data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemDataSource:
        groupedItemsList.groups.dataSource,
        itemTemplate: select('#semanticZoomTemplate'),
        selectionMode: 'none', tapBehavior: 'invoke',
        swipeBehavior: 'none' }"
></div>
```

4. Three templates are used between the two `ListView`s. Note the `groupHeaderTemplate` in the zoomed-in view. That's the third one. Set up those three templates:

```
<div id="headerTemplate" data-win-  
control="WinJS.Binding.Template" style="display:  
none">  
<div>  
  <h1 data-win-bind="innerText: category"></h1>  
</div>  
</div>  
  
<div id="mediumListIconTextTemplate" data-win-  
control="WinJS.Binding.Template" style="display:  
none">  
<div class="mediumListIconTextItem">  
  <div class="mediumListIconTextItem-Detail">  
    <h4 data-win-bind="innerText: name"></h4>  
    <h6 data-win-bind="innerText: address"></h6>  
  </div>  
</div>  
</div>  
  
<div id="semanticZoomTemplate" data-win-  
control="WinJS.Binding.Template" style="display:  
none">  
  <div>  
    <h1 class="semanticZoomItem-Text" data-win-  
bind="innerText: name"></h1>  
  </div>  
</div>
```

And there you have it. The pinch gesture is baked-in behavior, so you don't need to handle the event. Windows 8 handles the switch for you.

Chapter 8

Building Tiles and Using Notifications

In This Chapter

- ▶ Adding live tiles for your app
 - ▶ Notifying the user to changes in your app
 - ▶ Using Toast to get information to your users
-

Although new to Windows programming, tiles are pretty familiar to anyone who has built Windows Phone apps or used an Xbox 360 recently. They are the new icons of the Windows world, and much more too.

Tiles represent the personality of your app. They are a button to click and launch, yes, but they are also a window into the application when it isn't running. The Windows 8 start screen should have the feeling of being alive, and the tiles play a big role in that.

Your apps can send text, images, and even animations to the tile when it's running. When it's suspended, it loses this ability, but that's okay. If you remember to update things when you go into suspension, it seems to the user that the tile is representing the current state of the app.

Beyond updating from the client app, Windows also allows Live Services to update your tile from the Internet. It's a way to push the usual rules of Windows 8, and have your app be alive when it isn't running.

Outside the scope of the tile itself, other features also tell the user what's going on. The Notification system lets your app inform the user when clearly defined events are fired, no matter if your app is running or not.

You are probably sensing a theme here. Despite the whole thing about Windows 8 apps getting suspended and not running anymore, there are a few ways to get to the user when your app is suspended. Clearly more

communication with the user is a good idea. The more you can communicate with your user, the more they use your app, and the more good reviews and downloadable content sales you get.

Communication good, suspension bad, right?

To see what I mean, just press the Windows key on your keyboard. If you have used the Weather, Music, or Photos app, your Start screen should be alive with activity. You can see my Start screen in Figure 8-1.

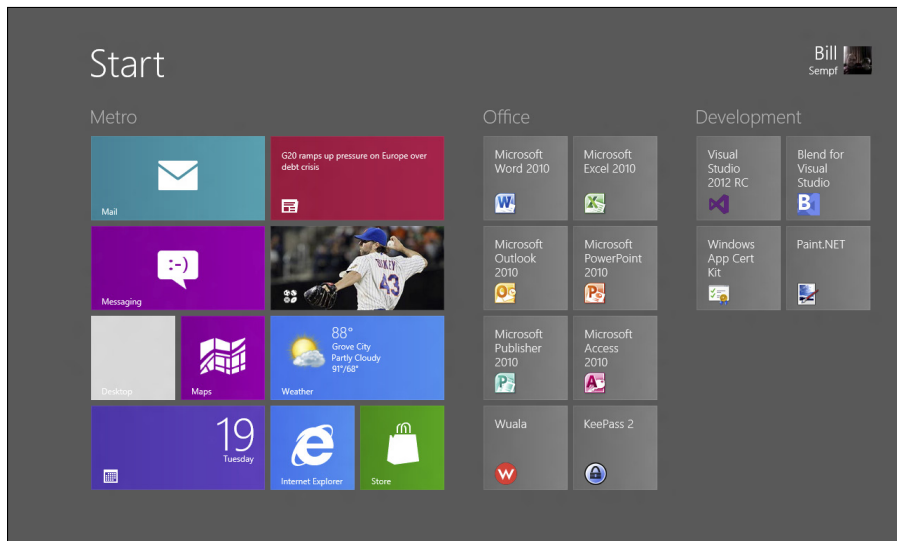


Figure 8-1:
My Start
screen.

Using Basic Tiles

When you build the random app in Visual Studio and run it in the debugger, an app tile is made for you out of the default logo images in the project template. All of this is defined in the application manifest, which Microsoft has kindly given us a user interface to modify.

The core ideas of the tile, though, are only maintained through code. The tile layouts are XML, and they are handled through the `TileUpdateManager`, a class in the `Notifications` object of WinRT. This class provides what we need to keep the tile in scope for Windows 8, expose the capabilities of your apps billboard, and fulfill obligations to Windows.

Core to the Windows 8 design language

One of the core principles of the Windows 8 design language is consolidating groups of common tasks to speed up usage. The tile is an implementation of this principle. It allows the user to see into the application from the desktop, rather than just having an icon to click to run the application.

Tiles are more than just an updated icon, however. Throughout the next few sections, I discuss the programming of live updates, animation, and images. The tile is a little application of its own.

An alive and responsive UI should do even more than display static panels of information as tiles, right? I certainly think so. It should have interesting, moving, and informative slides passing subtly through the user's awareness as she uses the Start screen.

Toward that end, tiles have certain capabilities that extend the power of WinRT a little beyond the sandbox. They can get updated when the application isn't running, and bend a few of the graphics rules. More about this in Building Live Tiles and throughout the chapter.

For now, understand that the tile might be the most important part of your app that isn't directly part of your app. It is core to the design language and central to the Windows 8 principles. It also makes a great billboard for your app.

Tile capabilities

It's pretty clear based on what they can do that tiles have certain capabilities that sort of fly in the face of the general idea of WinRT. There is this Windows sandbox that your Store apps have to play in, but the tiles can play out in the grass. Here are some of the things tiles can do:

- ✓ They can update when your app isn't running. Tiles can get updates from the Windows Notification service even when your app isn't running. For example, if the user uploads a picture from a website, the tile for your photo app can be updated to show it.
- ✓ Tiles draw inspiration from a variety of sources. The layout for a tile isn't HTML or XAML, or just old-fashioned XML (POX, or plain ol' XML), as we used to call it back in the day. The content, badge, and name are handled with different content sources and are updated independently.
- ✓ You can almost show a movie. The image, animation, and cycling updates of a tile allow you as an app developer to do some pretty gaudy stuff. Although the apps are supposed to be prim and proper, the tiles can be pretty noisy.

Why all these capabilities for something that is basically an application icon? It's the window to your app, that's why. You can, and should, use the tile to show off your app, get people to use it more, and generally impress the users.

Tile expectations

That said, people have certain expectations of tiles, and those expectations are largely implemented as constraints on the development model. Tiles can only be a certain size, and can only update so often. The tile sandbox is bigger, but it still exists.



The first and most important expectation is that the app can handle two sizes of tile: square and wide. The square tile is usually used as just an icon, but it can be updated with active content too, as well as branding and badges. Wide is the format usually associated with active content because it is more than twice as large as the square tile.

The user decides if there is a square or wide tile. To see what I mean, right-click (or swipe up) a wide tile in the Start screen and notice the app bar that shows up. It has a smaller button. Because the user has control, you have to be prepared to show something in the square tile as well as whatever you are showing in the wide format.

- ✓ Tiles aren't expected to be active. In fact, the user's Start page can become too noisy if every tile is animated. The tile can still have relevant, up-to-date information and be stylistically consistent with the app.
- ✓ Updating a tile is a matter of the use of the app. For something that has content updated by the app itself — say perhaps the high score of a game — the app should do the updating, just as it is preparing for suspension. If the app consumes content from the outside, the tile can update irrespective of the app, but only once every 30 minutes at most.
- ✓ Tile content should be relevant. Show the user stuff that will make him happy. If you show the user ads or promote other apps, the Store will probably reject your app.

In general, the tile is supposed to make the user's Start screen nicer. Look at your Start screen. You have it just the way you want it? Good — give the user the same opportunity. Remember, we are in a very democratic sales environment here, and a bad tile will get you uninstalled.

Building Live Tiles

Setting up a new tile for your application is largely a configuration game. The layouts and structure are pretty much sorted for you. All you need to do is set the settings that need to be set, if you know what I mean.

In order to do that at a basic level, you need to provide some images. The basic template is more or less preconfigured to use a provided icon. At a more sophisticated level, you need to hook into some events that are supported by the operating system to get content to the tile.

First though, you need to know how the layout works. There are three regions on the tile — the layout, the badge, and the name. Figure 8-2 shows where those regions are on one of the wide (left) and square (right) tile templates.

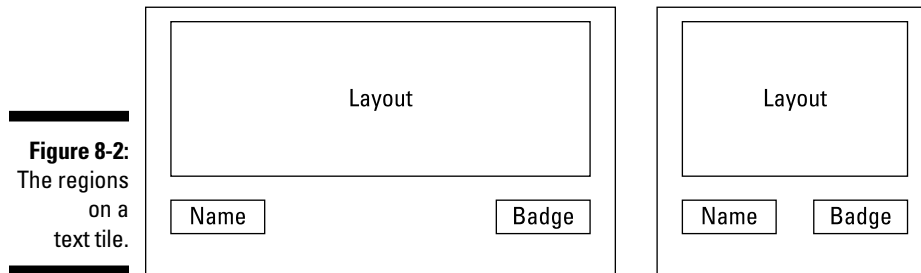


Figure 8-2:
The regions
on a
text tile.

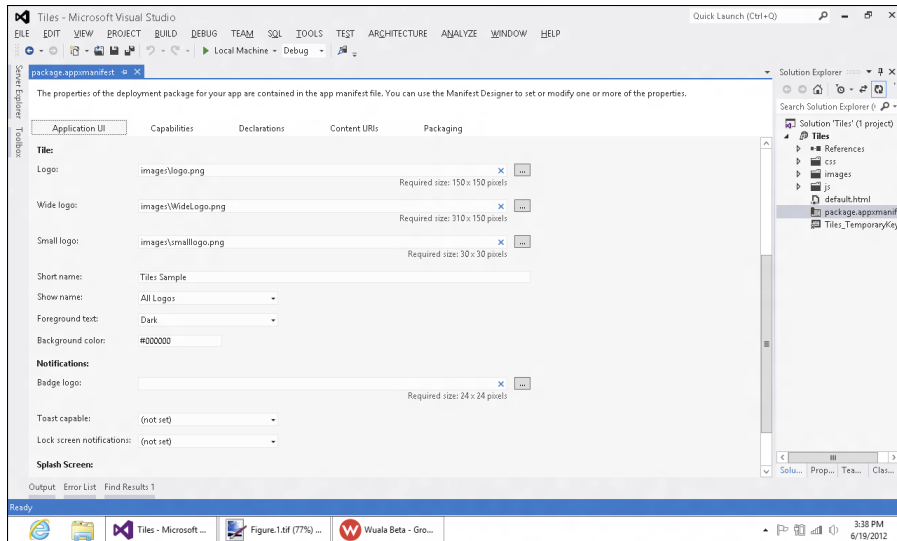
The name and badge are simply configured in the `package.appxmanifest` file. The layout is determined by an XML structure, and by default uses an image that is also configured in the `appxmanifest` file. As such, you configure yourself a basic tile, with no coding at all.

Configuring a basic tile

When creating a new project, the tile comes for free: All apps have a tile. If you have been playing with the SDK samples, you are familiar with the white-on-blue tile that all of the samples come with. If you have toyed with the default templates in Visual Studio, you have seen the white-star-on-black tile.

Those tiles are configured for you in the template; they aren't built-in. It's pretty simple to change them. Open up any project (or use the Tiles sample from the book code) and double-click on the `package.appxmanifest` file, or take a look at Figure 8-3.

Figure 8-3:
Tiles in the
package.
appxmani-
fest file.



Scroll down a little to find the tile section of the file. Notice the logo and small logo items have files associated with them. The logo file is what is used to make the tile image in the new project deployment.

There isn't a wide logo defined, and unsurprisingly there isn't a wide tile option for a default app in the Start screen. You can see what I mean by right-clicking the project and selecting Deploy. When deployment is done, go to the Start screen, right-click the new tile, and check the app bar. Is there a wide option? Nope.

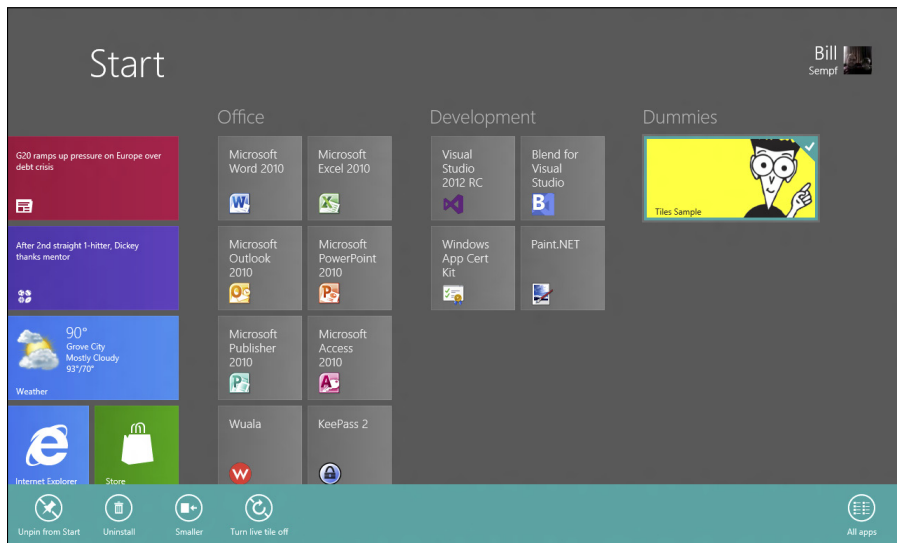
You can change that. Create an image that is 310 pixels wide and 150 tall and put it in the Images folder of the project. I used Windows Paint to make mine. You can use whatever tool you wish as long as it produces a .gif, .jpeg or .png file. Even Visual Studio has an image editor now.

When the file is in the Images folder, click the Browse button next to Wide Logo in the `package.appxmanifest` editor and find the file you loaded. I called mine `WideLogo.png`. When the file is shown in the editor, right-click the project and select Deploy again.

Now press the Windows key and check out the tile. It will default to the wide tile, and the user now has an option to change to the smaller tile. The existence of the Wide Logo in the app manifest defines what tile size your app starts with.

Along those lines, if you fill in the Short Name field in the manifest editor, that appears in the Name region of the Tile, as shown in Figure 8-4. This value is editable in the code, so you can use it to do more than just pass the name on to the user.

Figure 8-4:
The wide
tile with
a Name
region.



Publishing with the templates

Tile content other than the default layout is handled by a set of XML templates that are hidden away in the deepest reaches of WinRT. Forty-six of these templates handle all sorts of different setups, icon types, and content.



The template types (see Figure 8-1) are found in the `Windows.UI.Notifications.TileTemplateType` enumerator. Search <http://msdn.microsoft.com/en-us/windows> for `TileTemplateType` if you want a full list with samples.

The template types include some edge cases and a few layouts that aren't that nice-looking. The content types also include Peek, which is another advanced feature of the tile. Generally speaking, you'll use a few template types all the time.

Table 8-1 Tile Template Layouts		
<i>Type</i>	<i>Description</i>	<i>Image</i>
tileSquareImage	An image that fills the whole square tile.	
tileSquareBlock	One large string over one small string. Like a calendar.	
tileSquareText01	One header and three smaller lines of text. There are four versions of this with various formatting.	
tileWideImage	One wide (310px) image only that fills the whole wide tile area.	
tileWideImageAndText01	An image with a space at the bottom for text.	
tileWideBlockAndText01	Like the TileSquareBlock only with lines for text at the left.	
tileWideSmallImageAndText01	A small icon with space for text to the right. There are four versions of this.	
tileWideText	Probably the most common — 11 versions of text only in the wide tile, no image.	

The pattern is this: You decide what your users will be most interested in seeing on the tile, and then you pick the template that best mimics what you want. Each template has a set of slightly different properties that you fill with content at runtime. Bam! You have a live tile.

Properties for the template are defined in the XML. For instance, the XML for the `TileWideImageAndText01` looks like this:

```
<tile>
  <visual>
    <binding template="TileWideImageAndText01">
      <image id="1" src="image1.png"/>
      <text id="1">Text Field 1</text>
    </binding>
  </visual>
</tile>
<tip>
```

If you wish, you can look at the XML that you retrieve using the `getTemplateContent` method. In the step list in the following section, just set a breakpoint on the second line of code, and then look at the content of `tileTemplate`.

But I'm getting ahead of myself.

Rendering content in the app's personality

Using the wide Dummies tile image, put some text on it and give the users something to look at.

- 1. Get the XML template from the `TileUpdateManager`.**

This is like the code at the end of the previous section. You need to set the values in the template with your own content.

- 2. Grab the `Text` element from the XML using `getElementsByTagName`, and set the value.**

- 3. Set up the `Image` element by grabbing the element from the XML — just like the text. Set the value to a local image.**

- 4. Make a new `TileNotification`.**

This has the new data that you need to send to the tile.

- 5. Create a new `TileUpdater` with the Notification.**

This actually sets up the asynchronous call to populate the tile.

- 6. Put all of this code somewhere where it will get called. You can put it on the `oncheckpoint` event if, for instance, you want to update just before suspension.**

For this example, I just put my code in the `onactivated` event.

Here is the code, including the `onactivated` event handler setting:

```
app.onactivated = function (eventObject) {
    if (eventObject.detail.kind === Windows.ApplicationModel.Activation.
        ActivationKind.launch) {
        if (eventObject.detail.previousExecutionState !== Windows.
            ApplicationModel.Activation.ApplicationExecutionState.terminated)
        {
            var tileTemplate = Windows.UI.Notifications.TileUpdateManager.
                getTemplateContent(Windows.UI.Notifications.TileTemplateType.
                    tileWideImageAndText01);
            var tileTextAttributes = tileTemplate.getElementsByTagName("text");
```



```

tileTextAttributes[0].appendChild(tileTemplate.
    createTextNode("Welcome to the Dummies Guide to Tile making"));
var tileImageAttributes = tileTemplate.getElementsByTagName("image");
tileImageAttributes[0].setAttribute("src", "ms-appx:///images/
    WideLogo.png");
var tileNotification = new Windows.UI.Notifications.
    TileNotification(tileTemplate);
Windows.UI.Notifications.TileUpdateManager.
    createTileUpdaterForApplication().update(tileNotification);
}
WinJS.UI.processAll()
}
};

```

Where does that leave us? Deployment isn't enough now — this is about running the app. After it's run (even in the debugger), you can see the new tile in the Start screen, hopefully much like Figure 8-5.

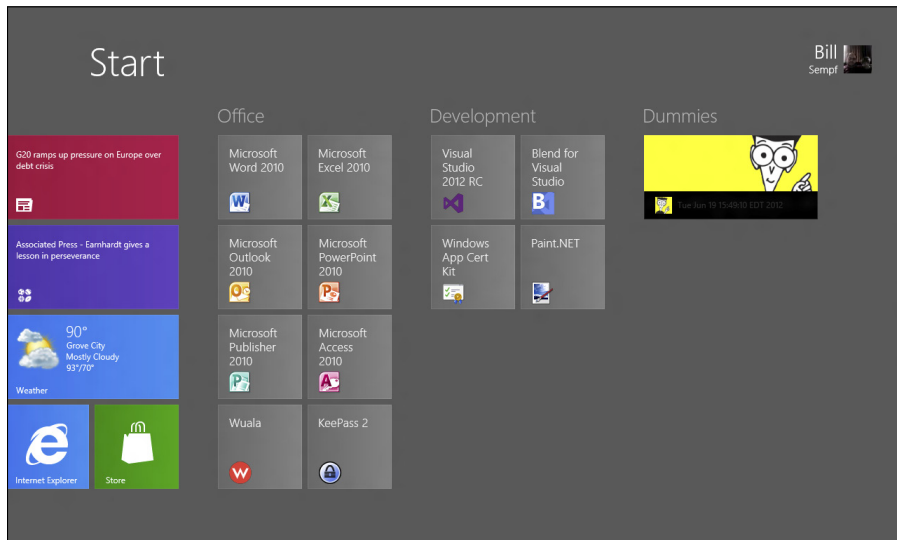


Figure 8-5:
The new
tile, with
template.

Getting Notified

After you have the template set up, getting notifications on the tile is not that tough. You want to decide what events in your system should update the tile, and then attach tile updates to them.

So think about your application. In the `ToDoToday` app, I update the tile whenever the user updates the task list, making sure I show the latest tasks in the Notification area. If you're writing a game, you probably would want to update the tile every time the user clears a level.

In general, treat the notification as exactly what it is — the opportunity to notify the user about the last thing that happened in the app.

- ✓ Update the notifications when the user does something significant
- ✓ Update the notifications when there is significant information to pass on to the user.
- ✓ Update the notifications when the app is about to be suspended.

Making notifications part of your tile

The good news is that you have already made a notification part of your tile, in the “Rendering content in the app’s personality” section. Setting the `Text` element value in the tile XML is essentially a notification — the only difference is in use and strategy. The tactics are the same.

To best implement it, construct a function that is designed to be called from any of the important events:

```
function updateTile(textNotification) {
    var tileTemplate = Windows.UI.Notifications.TileUpdateManager.
        getTemplateContent(Windows.UI.Notifications.TileTemplateType.
            tileWideImageAndText01);
    var tileTextAttributes = tileTemplate.getElementsByTagName("text");
    tileTextAttributes[0].appendChild(tileTemplate.createTextNode
        (textNotification));
    var tileImageAttributes = tileTemplate.getElementsByTagName("image");
    tileImageAttributes[0].setAttribute("src", "ms-appx:///images/WideLogo.
        png");
    var tileNotification = new Windows.UI.Notifications.
        TileNotification(tileTemplate);
    Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication().
        update(tileNotification);
}
```

This we can call and maybe pass in the date just to see how it works. For instance, add a promise-activated function after the `processAll` function in the `default.js` file of the sample project:

```
WinJS.UI.processAll().done(function (e) {
    var today = new Date();
    updateTile(today);
});
```

Run this, and the tile should be updated with the date in the text region. Now, say you want to update the tile live as the user is using the app. To replicate this, you can add a button and an event handler for said button right there on the default page.

```
<body>
    <button type="button" id="actionButton"
        style="position: relative; left: 300px; top:
        300px;">Click Me</button>
</body>
```

Set up a variable for the button in `default.js` (I called it `theButton`), and then add a little code to call the `updateTile` function:

```
theButton = document.getElementById("actionButton")
theButton.addEventListener("click", function (e) {
    var today = new Date();
    updateTile(today);
});
```

Run the app again, and then press the Windows key and confirm that the date and time in the tile is correct. Then go back to the sample app and click the button.

Press the Windows key again and the tile will have changed. This could be the score of the big game, the news, or an appointment — the choice is yours.

Getting a message on the queue

What if you have more than one thing you want to show the user? You could use one of the templates with a lot of small text, but there is a better choice. Microsoft allows you to stack up to five messages on the queue and cycle through them one at a time.

In order to do this, you *must* enable the queue. This is a one-line property setting that you can put in the initialization code of your app. I used the `onactivated` event for this example:

```
Windows.UI.Notifications.TileUpdateManager.
    createTileUpdaterForApplication().
    enableNotificationQueue(true);
```

After you do that, I would recommend clearing the queue. You don't want what was in there before rolling through with your new content.

```
Windows.UI.Notifications.TileUpdateManager.  
    createTileUpdaterForApplication().clear();
```

So now the `onAppActivated` looks like this:

```
app.onactivated = function (eventObject) {  
    if (eventObject.detail.kind === Windows.ApplicationModel.Activation.  
        ActivationKind.launch) {  
        if (eventObject.detail.previousExecutionState !== Windows.  
            ApplicationModel.Activation.ApplicationExecutionState.terminated)  
        {  
            Windows.UI.Notifications.TileUpdateManager.  
                createTileUpdaterForApplication().enableNotificationQueue(true);  
            Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication().  
                clear();  
            theButton = document.getElementById("actionButton")  
            theButton.addEventListener("click", function (e) {  
                var today = new Date();  
                updateTile(today);  
            });  
        }  
        WinJS.UI.processAll().done(function (e) {  
            var today = new Date();  
            updateTile(today);  
        });  
    }  
};
```

Press F5 to start the app, and then click the Windows button to take a look at the tile. It should have the date on it. Go get a cup of coffee, and come back. I'll wait.

Now press the Click Me button, and then take a look at the tile. The notification should be scrolling between the two times — when you started the app, and when you clicked the button.

Go back to the app and click the button again. Now look at the tile; three dates are scrolling by. You can have up to five of these, and the Notification manager drops the oldest automatically.

Creating Secondary Tiles

Users can create a secondary tile that directs them to specific content within your app. Think of it as a bookmark in a web browser that allows the user to go to a specific page rather than navigating there.

Getting the idea of a secondary tile

Secondary tiles are a lot like app tiles. They have the same size, image constraints, templates, and user interaction. When clicked, they launch the app, and when the app is uninstalled, they go away.

There are a few differences though. Users have a lot more control over the creation of secondary tiles. Secondary tiles can be created and destroyed at the user's whim. Secondary tiles are only created at runtime and prompt the user for confirmation.

Unfortunately, unlike app tiles, secondary tiles are a little complex to create. First, apps can't create secondary tiles; only users can. Second, the app needs to give the user a way to create the secondary tile with a button or whatnot. Third, the app must handle the event that is caused by the pinning of the secondary tile.

I look at these one at a time because it's more complicated than it probably needs to be.

Pinning a secondary tile

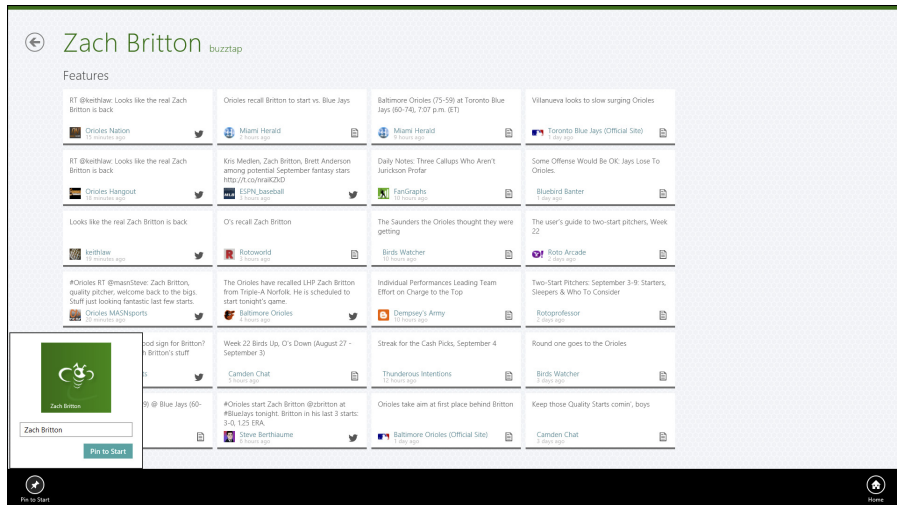
A user has to be the one who decides to create a secondary tile. This is a good thing, actually. It prevents self-serving apps from covering the user's start screen with little "reminders" that are just disguised advertisements.

You can accept any type of event to pin a secondary tile, but usually there is a button added to the app bar that lets the user pin the active page. For instance (as shown in Figure 8-6), in Buzztap (a sports news app), you might want to pin a specific player to your Start screen.

To pin a start tile, use the `StartScreen` class in the WinRT UI API. The function call looks something like this:

```
Var secondaryTile = Windows.UI.StartScreen.  
    SecondaryTile("recipe001",  
        "Baklava",  
        "Bill's Best Baklava Bake",  
        "recipeID=440",  
        Windows.UI.StartScreen.TileOptions.showName,  
        new Windows.Foundation.Uri("ms-appx:///images/logo.  
            png")  
    )
```

Figure 8-6:
Pinning a
player in
Buzztap.



Look at those arguments one at a time:

- ✓ The `TileID` is a unique, developer-created identifier for the tile. If you reuse it, the old one is replaced.
- ✓ The short name is shown directly on the tile.
- ✓ The display name is what shows in the tile's tooltip.
- ✓ The arguments are given back to the application when the tile is clicked. The arguments would be something meaningful to the application, like an item ID.
- ✓ The `TileOptions` enumerator tells Windows 8 whether the short name should be displayed.
- ✓ The image that should be used for the background is the final argument.

After the tile object is created, it needs to be installed on the Start screen. That requires the calling of `requestCreateAsync`, which asks the user (with a flyout) if it's okay. It then installs the tile.

```
secondaryTile.requestCreateAsync().then(function
    (isCreated) {
        if (isCreated) {
            // It worked, go on with your life
        } else {
            // Something went wrong
        }
    });
```

Windows 8 is now in charge of placing the tile, and the user can move it, resize it, or delete it at will.

Handling launch from secondary tiles

Your app can be activated from a secondary tile just like it is from a regular tile. Fortunately, the same activated event is used. Also fortunately, you can tell what tile was used to perform the activation.

In the activated function (the one you use for the event handler for the `Activated` event your app is listening for), you need to check the arguments if your app uses secondary tiles.



If you don't implement secondary tile pinning in your app, the user can't do it. Remember, although the user has to do the work, the app has to provide the functionality.

```
function activated(eventObject) {
    if (eventObject.detail.kind === Windows.
        ApplicationModel.Activation.ActivationKind.
        launch) {
        if (eventObject.detail.arguments) {
            //There are arguments.
            //Get them and use them to navigate.
        } else {
            //No arguments means it is launched from the
            tile.
        }
    }
}
```

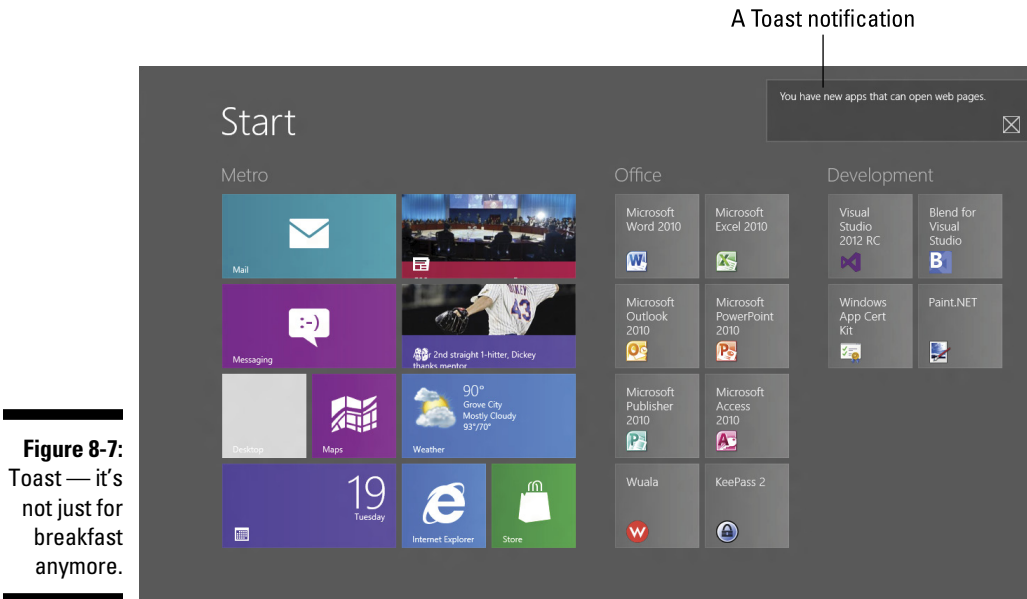
Those arguments are going to be the same as what you set in the tile constructor you built in the “Pinning a secondary tile” section. You can get them out, parse them up, and use them for navigation.

Getting the User's Attention with Toast

Updating a tile is an excellent way to get a user's attention on the Start screen. When the user is in another app, however, that is a totally different story.

In the Windows 7 world, you used to be able to create SysTray applications that could notify users about important events. In the web space, you can use an alert to communicate even across the barrier between tabs. In Windows 8, we have Toast.

The Toast in Figure 8-7 (and everywhere else) is a little slab of real estate that appears in the upper-right corner of the Windows 8 screen. It can be timed or active from the app, or even cloud-driven. It can have an image, text, or both. It is there to tell the user something important and bring them back to your app.



Deciding when to use Toast

Just putting Toast in front of the user and saying “Hey, isn’t our app cool? Come use it more!” is pretty bad form and will get you kicked out of the Store. There are a few determining factors as to when and where Toast notifications are best used.

Good uses of Toast include

- ✓ Informing the user when something they are waiting for is done, like a background worker or upload.
- ✓ Telling the user that they have a message.
- ✓ Reminding the user of something he asked to be reminded about.

When the user gets an appropriate Toast notification, clicking on the notification should take them to exactly the right space in the app to deal with it.

You have to realize, though, that the user might not see every Toast notification because they are transient: they only last for a set amount of time, and it is user configurable.

The get-you-kicked-out-of-the-Store variety of Toast usages include

- ✓ High-volume messages, like every time the temperature changes.
- ✓ Routine things, like PC maintenance events.
- ✓ Really important things the user *must* see. Toast notifications are transient, so they're not a good choice for crucial messages.
- ✓ Directing the user to your app for no reason. "Beat the last high score of 349!" isn't a good Toast notification.



While you're trying not to annoy the user, remember to leave the Toast app name out of the notification. You generally shouldn't include app icons in your Toast notifications either, but if the image adds something, that's okay. Finally, make sure you checked to see if notification is disabled before you send a Toast notification.

Calling on Toast from your app

In order to send Toast notifications, you must be able to receive them. It's one of *those* kinds of arrangements. Figure 8-8 shows the Notifications area of the manifest, where you can set the Toast Capable setting.

One use of Toast is a timed notification. When the notification is on the queue, it will run even if your app is suspended because the queue is handled by the OS.



You can also send notifications with the cloud. (Yes, I said cloud. You can take a drink now.) Take a look at "Notifying your users when the app isn't running" to see what I mean. Figure 8-9 shows a notification over a non-Store app.

Scheduling a notification happens at the user's behest, of course. Something would be set up in advance — some value set and some button pushed. For this example, however, I put the schedule in the `onactivated` event handler. Just move the code around to where you need it!

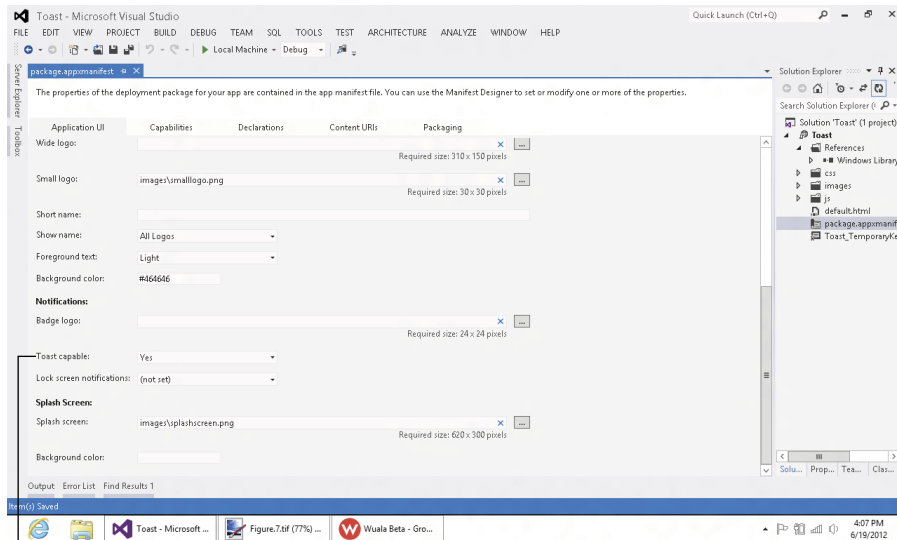


Figure 8-8:
Setting
yourself up
for Toast.

A Toast Capable setting

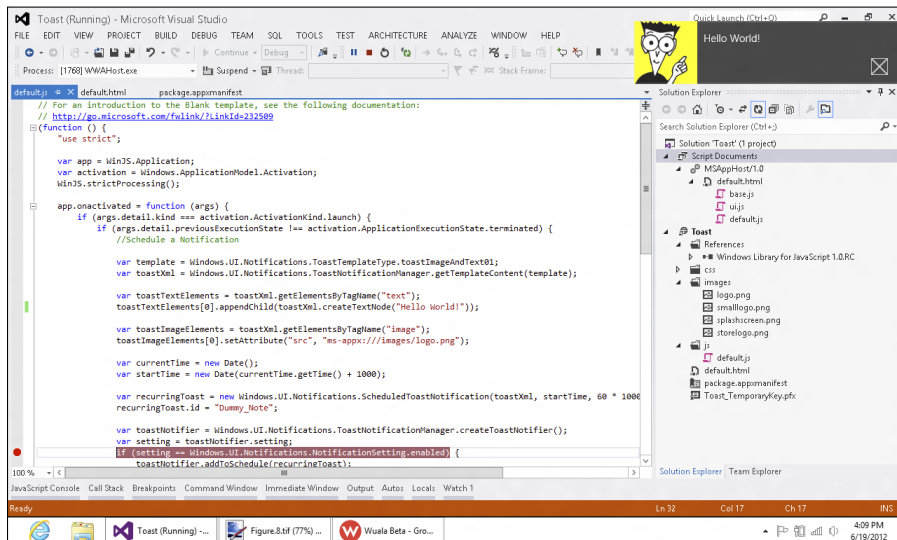


Figure 8-9:
Toast over
Visual
Studio.

1. Just like the tile, you need to set the template for your Toast.

The templates are very similar to tiles, but for a full listing, search <http://msdn.microsoft.com/en-US/windows> for ToastTemplateType.

```
var template = Windows.UI.Notifications.ToastTemplateType.  
    toastImageAndText01;  
var toastXml = Windows.UI.Notifications.ToastNotificationManager.  
    getTemplateContent(template);
```

2. First set the text element of the notification.

```
var toastTextElements = toastXml.getElementsByTagName("text");  
toastTextElements[0].appendChild(toastXml.createTextNode("There is a new  
message from a Dummy."));
```

3. Then, set the image value. This of course varies based on the template you have chosen.

```
var toastImageElements = toastXml.getElementsByTagName("image");  
toastImageElements[0].setAttribute("src", "ms-appx:///images/logo.png");
```

4. In order to set the schedule for the notification, you need a time value.

```
var currentTime = new Date();  
var startTime = new Date(currentTime.getTime() + 1000);
```

5. Now set the schedule using `ScheduledToastNotification`.

```
var recurringToast = new Windows.UI.Notifications.  
    ScheduledToastNotification(toastXml, startTime, 60 * 1000, 5);  
recurringToast.id = "Occasional_Dummy_Note";
```

6. Finally, add the notification to the queue to let Windows handle the rest.

```
var toastNotifier = Windows.UI.Notifications.ToastNotificationManager.  
    createToastNotifier();  
toastNotifier.addToSchedule(recurringToast);
```

Together, the onactivated event handler looks like this:

```
app.onactivated = function (args) {  
    if (args.detail.kind === activation.ActivationKind.launch) {  
        if (args.detail.previousExecutionState !== activation.  
            ApplicationExecutionState.terminated) {  
            //Schedule a Notification  
            var template = Windows.UI.Notifications.ToastTemplateType.  
                toastImageAndText01;  
            var toastXml = Windows.UI.Notifications.  
                ToastNotificationManager.getTemplateContent(template);  
            var toastTextElements = toastXml.getElementsByTagName("text");  
            toastTextElements[0].appendChild(toastXml.createTextNode("There  
is a new message from a Dummy."));  
            var toastImageElements = toastXml.getElementsByTagName("image");  
            toastImageElements[0].setAttribute("src", "ms-appx:///images/  
logo.png");  
            var currentTime = new Date();  
            var startTime = new Date(currentTime.getTime() + 1000);  
            var recurringToast = new Windows.UI.Notifications.Scheduled  
                ToastNotification(toastXml, startTime, 60 * 1000, 5);
```

```

    recurringToast.id = " Dummy_Note";
    var toastNotifier = Windows.UI.Notifications.
    ToastNotificationManager.createToastNotifier();
    toastNotifier.addToSchedule(recurringToast);
}
args.setPromise(WinJS.UI.processAll());
}
};

```

Press F5 to run the app; press the Windows key to return to the Start Screen, and wait a few seconds. The notification appears with the default sound. Clicking the notification resumes the app from suspension.



For a complete look at everything Toast has to offer, take a look at Toast's SDK sample at <http://msdn.microsoft.com/en-US/windows>. It offers a complete look at images, sound, handling activation from a Toast notification, and even long-duration Toast messages.

Giving the user control

The user can decide not to receive notifications, as shown in Figure 8-10, and you have to respect that. Although it seems like Microsoft could choose just to disable *all* Toast notifications when the user opts out, I imagine they have reserved the right for their own apps to use Toast even when the user has opted out, and that's probably okay. System apps should play by their own rules to provide a stable working environment.

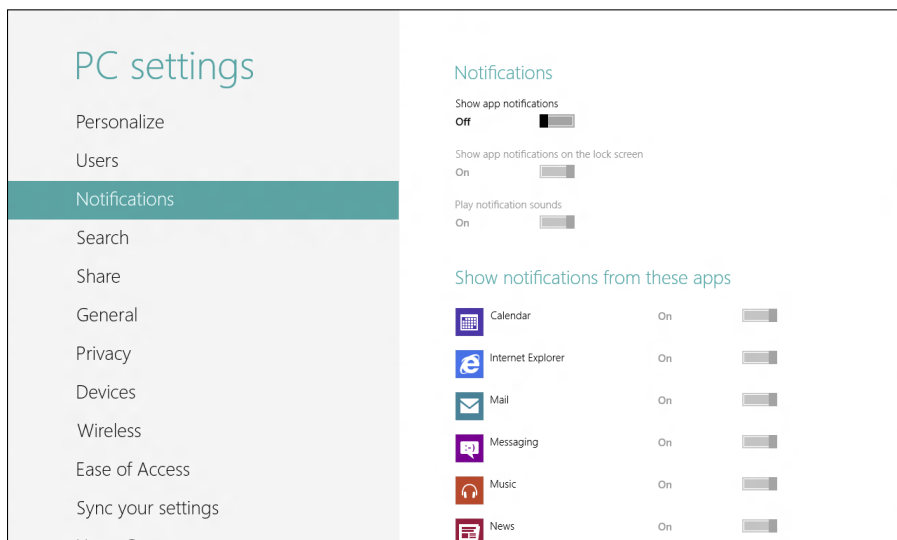


Figure 8-10:
Turning
notifications
off.

To turn off notifications, go to the Settings charm and click on Change PC Settings at the very bottom of the flyout. Figure 8-10 shows the third tab down (depending on your machine, the items may be slightly different), which is Notifications.

The user has the ability to show or hide all notifications, or just on a per-app basis. Your app needs to honor these requests.

The `ToastNotifier` has a setting property specifically designed to handle this. After you have called `createToastNotifier`, you can check the settings to see if they are enabled. It looks like this:

```
var toastNotifier = Windows.UI.Notifications.ToastNotificationManager.  
    createToastNotifier();  
var setting = toastNotifier.setting;  
if (setting == Windows.UI.Notifications.NotificationSetting.enabled) {  
    toastNotifier.addToSchedule(recurringToast);  
}
```

You might have to do a little refactoring of the code to make this more efficient. In the code in “Calling on Toast from your app,” earlier in this chapter, the `createToastNotifier` call is at the end — you might want to do it at the beginning and short-circuit the whole function if it isn’t enabled. As usual, what constitutes good patterns and practices depends on your specific circumstances.

Notifying Users When Your App Isn’t Running

You have four ways to get a notification to a live tile or Toast:

- ✓ Do it while your app is running.
- ✓ Schedule it for later, like the previous tile example.
- ✓ Set it up to periodically run.
- ✓ Use Windows Push Notification Service, or WNS.



WNS is designed for getting data to users when your app isn’t active. Real-time information, such as messages, invitations, updates, and news briefs, are good candidates for push. The service is offered by Microsoft for free as part of the Windows 8 UI development tools (you can use it in Windows Phone, too), so take advantage of it.

Designing for Windows Push Notification Service

WNS is designed to hook a service that you already have built in to your app, even when it isn't running. It does this by tracking registration of individual users that use your app, and then watching the service on their behalf.

That being said, this isn't a simple undertaking. It assumes a few things:

- ✓ You have an app that needs information from a custom service.
- ✓ That service is already written.
- ✓ The written service is hosted on the Net.

If you have those three things, you are golden.

This little push system has four parts, as shown in Figure 8-11. Two of them are Microsoft's responsibility, and two of them are yours.

The two top pieces — the Windows 8 app and the service — are your problem. Microsoft takes care of the Notification Client Platform and the WNS services.

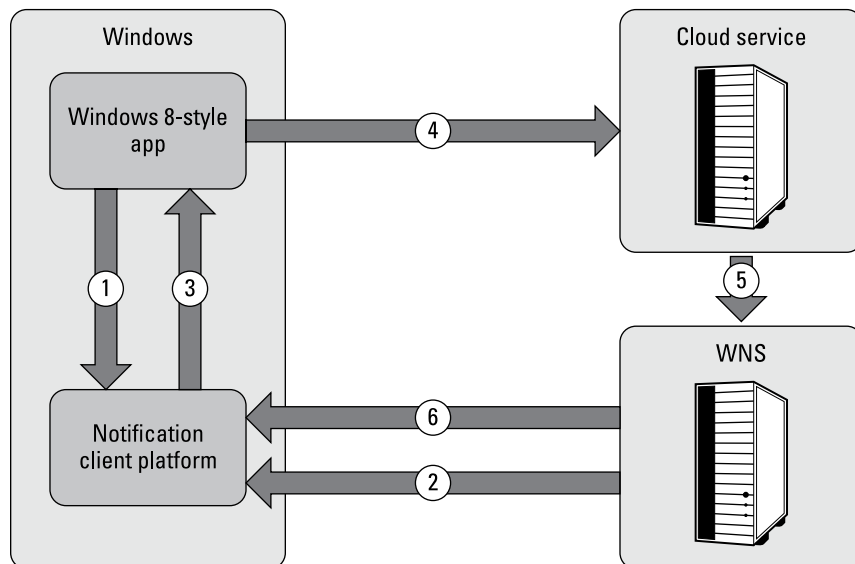


Figure 8-11:
WINS
architecture.



When Microsoft says *cloud service*, they mean a web service hosted on the Internet. I have no idea whatsoever why that is suddenly a cloud service, but it is.

After your app is set up to use the service, the process in Figure 8-11 begins to unfold:

1. **The Windows 8 app sends a request to the local notification client platform for a URL for the notification service.**
2. **The client platform gets a URL from WNS.**
3. **That URL goes to your app for use.**
4. **You send it to your service so it knows where to send stuff.**
5. **When your service has something new to push to the tile, it sends it to the URL that the app has sent.**
6. **WNS sends it back to the platform, which updates the tile or Toast.**

This means that your service must keep a list of all registered installed instances of apps to update; this cycle happens for every app. The design of that service is a little beyond the scope of this book, but there are a lot of good WCF books out there. Host the service on AppHarbor, and you are good to go.

The Windows 8 app doesn't need any special design decisions really — the service just has to be something that has information that is periodically updated from the Internet, and it needs a place to put that information.

Registering your app

WNS needs to know that your app is going to send stuff. It uses a whitelist for authorization, so if your app isn't on the list, or it doesn't know the super-squirrel secret password, it won't be let in.

In order to register your app, you need to make sure a few pieces are in place in the App Dashboard. The Dashboard (which I talk about more in Part IV) is where you configure your app for the Store, and it is also how you set up WNS.

1. **Make sure your app is registered.**

If you planning on going to the Store this should be done, but during development you might put it off. Don't. You need the Dashboard to get WNS credentials.

2. **Set the identity information for the app listing in the Dashboard.**

This is inconveniently placed in the Advanced Settings section.

1. On the Dashboard, click on Advanced Features.
2. Click Push Notifications and Live Connect services.
3. Click Identifying your app.
4. Set the values found in the `package.appxmanifest`.

3. Get your WNS credentials.

This is the most important part — the secret you'll need to give your service. Click **Authenticating Your Service** and grab the Package Security Identifier (SID) and Client Secret from the page shown in Figure 8-12. Do not lose or share these! It's a massive security consideration. A malicious user could use these to hijack every installed instance of your app.

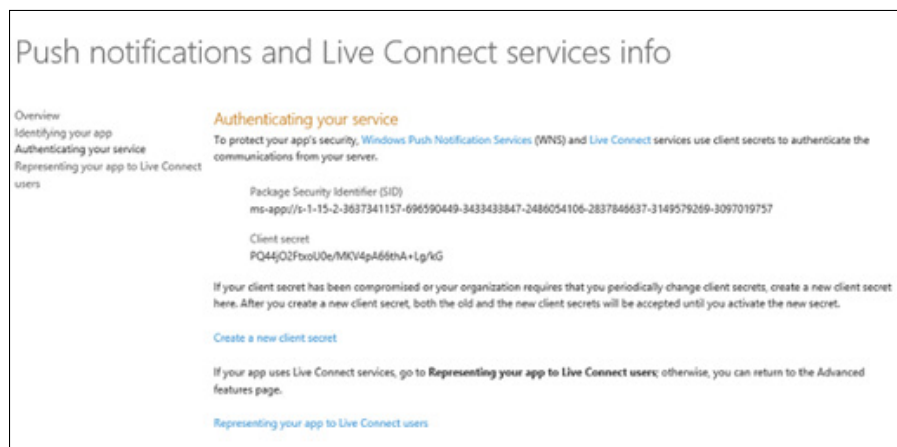
4. Teach your service to pass the credentials to WNS.

This happens in Step 5 of Figure 8-11.

Pushing notifications

Getting the notifications to the tile is now more or less the responsibility of the service. Because you can't write a WCF project in JavaScript, this is the province of another book. I could write a whole book on the topic of WNS and correctly designing, configuring, coding, and deploying a Windows 8 app with push notifications.

Figure 8-12:
Your package security information.





Use dev.windows.com/ for this. You can find a good article series there that offers a bunch of sample code and diagrams — much more than I can fit in this book. Just search for *notifications* to find it. Also, at the same URL, there is a sample for this that includes an example service and everything. Use the resources that Microsoft gives you.

I will say this — you have to consider two important bits in the Windows 8 app. First, you need to request the channel (the path to the WNS services) from your Windows 8 app. Second, you need to get that URL that you are given to your service and store it there.

To get a channel URL, you need to use the `PushNotification` class in WinRT. It has the worst-named method in WinRT: `createPushNotificationChannelForApplicationAsync`. The code looks like this:

```
var channel;
var pushNotifications = Windows.Networking.PushNotifications;
var channelOperation = pushNotifications.PushNotificationChannelManager.
    createPushNotificationChannelForApplicationAsync();
return channelOperation.then(function (newChannel) {
    channel = newChannel;
    // Success. The channel URI is found in newChannel.uri.
},
function (error) {
    // Could not create a channel. Retrieve the error through error.number.
});
```

Next, you need to get that `channel.uri` to your service. Fortunately, that's just an `xhr` call. The code for that is very straightforward.

```
var serverUrl = "http://YourServer.com";
var xhr = new WinJS.xhr({
    type: "POST",
    url: serverUrl,
    headers: {"Content-Type": "application/x-www-form-urlencoded"},
    data: "channelUri=" + channel.uri
}).then(function (req) {
    // Channel URI successfully sent to server. Retrieve the response from
    req.response.
},
function (req) {
    // Could not send channel URI to server. Retrieve the error through req.
    textStatus.
});
```

That's pretty much all you need to do from the Windows 8 app. Everything else is handled by the service.

Painting it Azure

Because your service has to be hosted on an Internet server anyway, it is worth mentioning that Windows Azure, which is part of Microsoft's cloud platform, supports up to 10 web applications (including WCF projects) for free.

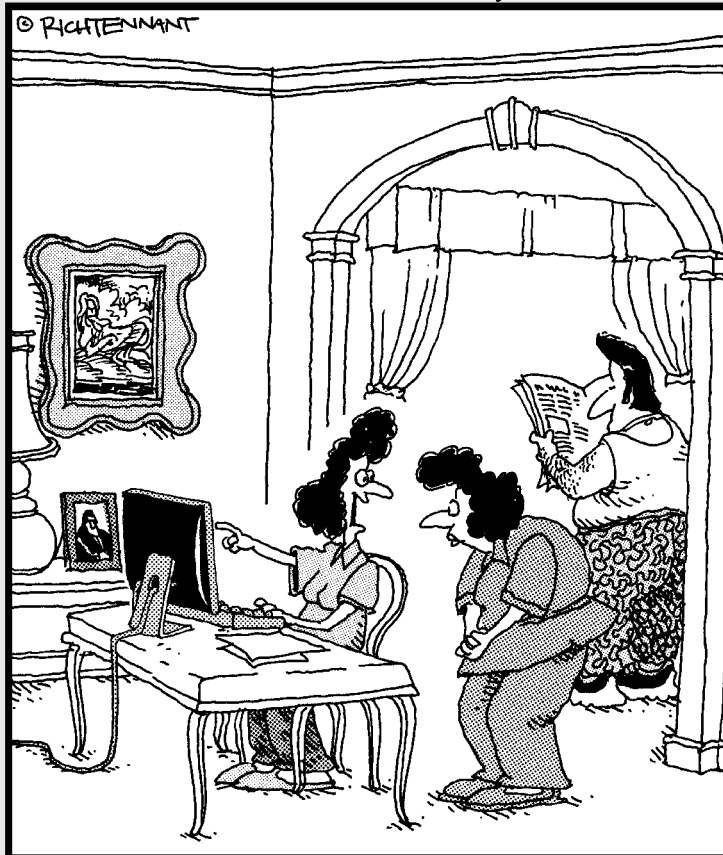
This gives developers an interesting opportunity to make a cloud-based service layer for their apps that can be registered with WNS. It doesn't have to be Azure, but the tech is really good and the price is right! I cover the new Mobile Services feature of Azure in depth in Chapter 15.

Part III

Digging into the Internals

The 5th Wave

By Rich Tennant



"Oh, Anthony loves working with HTML5. He customized all our web pages with a sound file so they all close out with a 'Bada Bing!'"

When it comes to really getting things done, you need data, networking, program flow . . . in short, your app needs some internals. This part is all about WinRT's new asynchronous platform for getting things done in a fast and fluid way.

Chapter 9

Programming to Contract

In This Chapter

- ▶ Allowing sharing with other apps
 - ▶ Providing search functions
 - ▶ Making use of devices
-

Once upon a time, if you needed to do something external to an application, like print to a printer or communicate with another program, you would copy your code from a magazine.

Okay, that was a *long* time ago. But it wasn't long ago that you would copy the code from the Internet. Then, after a while, libraries appeared that you could copy into your program.

Eventually, services appeared that you could use. If you wanted to provide the information, you could create your own service. Finding those services, or getting people to find yours, was rather difficult at times, though. Still is.

Windows 8 is trying something totally new. Instead of depending on Internet-enabled code — either copied from or provided within — the commonly used interface code is provided as part of the operating system's programming interface. It's called a contract, and it's taking Windows programming to new places.

Coding to a Whole New Tune

Windows 8 applications use charms to interface with the user, and contracts to communicate with the outside world. *Charms* are the navigational elements that appear on the right side of the screen when you swipe from the right or move your mouse to the lower-right corner of the screen. *Contracts* are agreements with other apps.

When Microsoft said that Windows 8 is Windows reimagined, few people thought they meant that developing for the platform would require a total change in attitude about how applications behave, but it does. Use of the charms and contracts interact with some of the most common aspects of contemporary application development.

Each feature has two sides — implementing the feature within your application and exposing the feature throughout the Windows 8 experience. I start with the Search contract and see where things go from there.

The Search Contract

You can do Search a few ways. You can write a Search view, and a procedure, and give users the ability to search within their data in your app. Or you could implement the Search contract, and not only give users the ability to search within the app, but also give other apps the ability to search within your app.

Not sure what this means? Don't feel like you are alone — search by contract is a pretty new concept. In Chapter 2, you got a tour of the Charms bar and the five built-in charms to be found there. Remember the Search charm? This is your chance as a developer to change how that feature works for users.

The Search charm is context-aware. If you search in the Start screen, like Figure 9-1, you search apps. If you search in Internet Explorer, you search Bing. But it gets even more interesting than that.

When I search for “word” in the apps screen, I get results off to the left, but that isn't what is interesting. What is interesting is the categories I get right below. This is the Apps application, so as expected, you get categories of apps: Apps, Settings, Files.

What isn't expected is the list of specific apps below: Store, Finance, Internet Explorer. They are there because there is data in those apps that match the search. Even though you're in the Apps search, you get results from Finance. That's pretty cool.

The Search contract is how your app participates in this neat process. If you implement the search and handle the events associate with search requests, your app shows up in the results of searches, even within other apps.

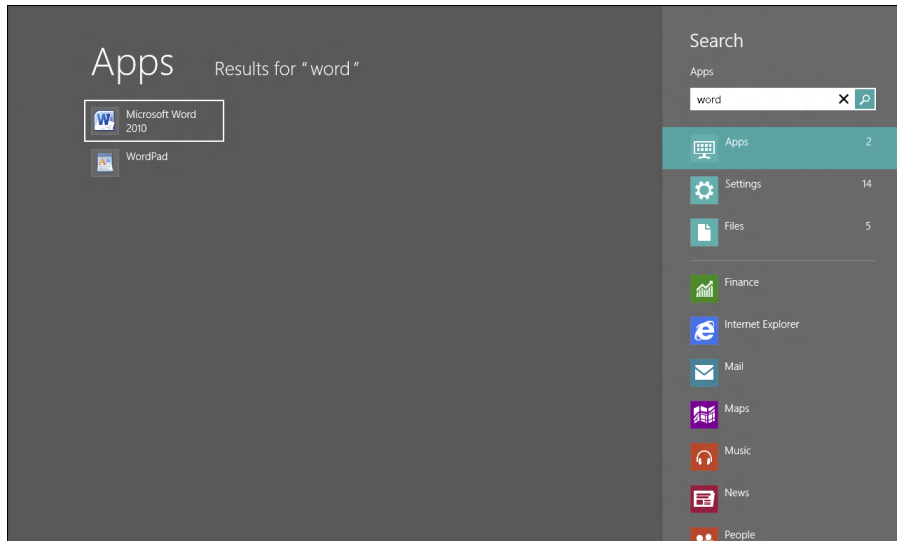


Figure 9-1:
The Search
flyout.

If you want people to use your apps, this is the way to do it. First, implement some searching within your app, and then handle other apps asking you for information.

Searching within your app

Fortunately, setting up search inside your application is the first step to having other apps able to search inside your app. The first step is participation in the Search contract.

You can start with any app and add a search contract. I started with a blank Windows 8 app and added a data.js file like that used in other samples (both downloadable from this book's website — see Introduction for more details). From any application, you can follow these steps to have that app play in the Search playground.

- 1. Add a new Search contract to the project by right-clicking the project file and selecting Add New Item.**
- 2. Select the Search Contract and click Add.**

3. Check out the package.appxmanifest file to make sure the search extension was added. Looks for the code

```
<Extensions>
  <Extension Category="windows.search" />
</Extensions>
```

4. Reference the search code in the default.html file. The references should look like this now:

```
<!-- Search references -->
<link href="/css/default.css" rel="stylesheet">
<script src="/js/default.js"></script>
<script src="/js/data.js"></script>
<script src="/js/search.js"></script>
```

That sets up the basic Search contract. Next you need to set up the Search filters and make sure you have a results view. The Search filters are a way for the programmer (that's you!) to control what you let the user search for within the app, as compared to what people who search from other apps see.

Whoa, there, I need to quit getting ahead of myself and do one thing at a time.

So . . . searching within the app. All you have left to do is create some view that users see when they complete a search. This, in a perfect world, is the gridview in your app, just with a new datasource.

For this application, just whip up a quick bound list of tasks. Right-click the project and click Add, and then New Item. In the dialog box in Figure 9-2, select the PageControl from the Windows Store Apps collection in the tree view.

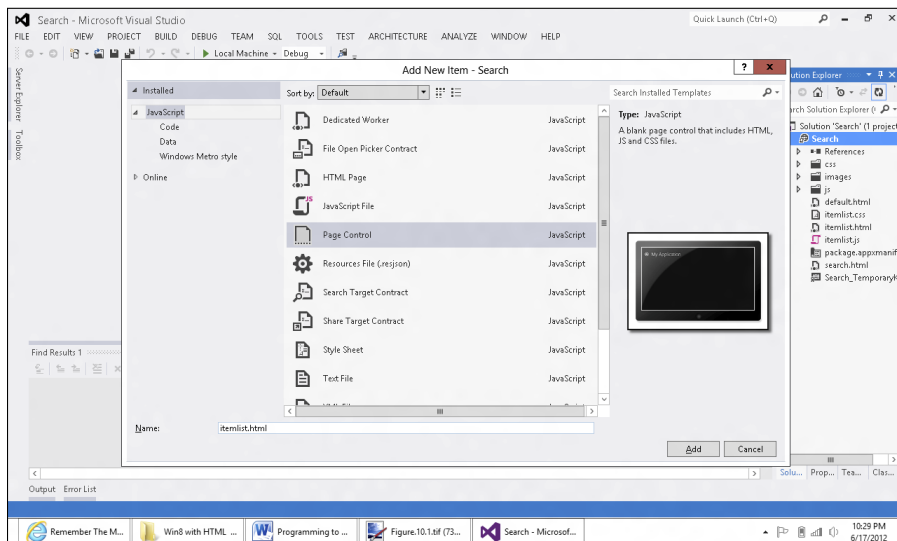


Figure 9-2:
New page
control.

Then follow these steps to bind to the search items:

1. Add a ListView to the HTML inside the Main Content section.

This is how the whole list is formatted. It is used once to hold the whole list.

```
<div id="basicListView" data-win-control="WinJS.
    UI.ListView"
    data-win-options="{itemDataSource : SearchSample.
        itemList.dataSource, itemTemplate: select('#
        mediumListIconTextTemplate')}" style="margin-
        left: 5%; margin-right: 5%;">
</div>
```

2. Above the list template, add the item template.

This is how each item is formatted. One of the template items is shown for every item in the list.

```
<div id="mediumListIconTextTemplate" data-win-
    control="WinJS.Binding.Template">
    <div style="width: 480px" data-win-bind="innerText: task"
        style="font-size: 18pt">
    </div>
</div>
```

3. Around line 28 of search.js, direct the user to the itemlist.html page when the search is invoked.

```
itemInvoked: function (eventObject) {
    eventObject.detail.itemPromise.then(function
        (item) {
        // TODO: Navigate to the item that was invoked.
        nav.navigate("/itemlist.html", {item: item.data});
    });
},
```

4. Remove the sample filters around Line 24 of search.js.

Now when you run the app (press F5), open the Charm bar. Click the search charm and type **Tardis**. You will be sent to the `ItemList` and see that one item presented for you.

You also need to consider how users of other apps see your results when coming from their app. Fortunately, you get to see the same view if you wish, but there are a few bits that must be included in the app code and configuration.

Returning results to other apps

So when a user searches from another app, and your app isn't on the screen, you can still return results. Because your app has agreed to participate in the Search contract via the manifest, you get certain things for free, and that is one of them.

Your app still has to do a few things (meaning you have to code for them). When a search activates your app, you have to handle the `Activated` event. Shocking, I know.

In the `default.js` file is a generic function for the `app.onactivated` event. There is a check in there for `ActivationKind.launch`, and you're going to add one for `ActivationKind.search`.

```
if (eventObject.detail.kind === Windows.ApplicationModel.  
    Activation.ActivationKind.search) {  
    nav.navigate("/search.html", { query :  
        eventObject.queryText });  
}
```

Hey, guess what? That's pretty much all there is to it. Well, okay, you should do a few things to make sure the query is valid, but just sending the data to the search is enough to get started. If you want to follow the standards for the Search contract, here are four things you should do:

- 1. Save the state of the app.**
- 2. Check the view state of the app.**
If it is snapped or filled, you might have a different view.
- 3. Test for an empty `queryText`.**
- 4. See if the `queryText` is the same as the last one you processed.**

After you've done those things, you can go ahead and navigate. The additional standards are important, but not vital to the function of the app. There are two standards at play here — making it work and following the Windows 8 principles. To have a truly responsive app that inspires confidence, you should check the state and the incoming data. I am in favor of that.

But first, just make it work.

The Share Contract

The social interaction available using Internet-connected technology has forever changed the way applications are written. It is now expected that a

recipe can be sent via e-mail, a high score can be posted to Twitter, and a picture can be posted to Facebook from most any kind of app.

The good news is that most all of the social networks have provided easy-to-use APIs to facilitate this sharing. It is in their best interests, after all, to have people provide content to their networks.

The bad news is that today you need to implement this repeatedly in each application that you write. For every game that shares with Twitter, you need to write that method that PUTs to the Post endpoint over and over again. When the API changes, it's up to you to fix up the app.

Loving sharing

Windows 8 apps won't suffer having to constantly change sharing code, ever, thanks to the Sharing charm. When the user wants to share that new achievement or post their latest thoughts from your app, they can just touch the Share charm and share content from the active app to any other app that declares itself as a sharing contract.

For instance, take a look at the News app from the Windows Store. Open the app and click on a story. If you swipe from the right, or press Windows+C, and then click the Windows charm, you get a list of all apps that support sharing. On a default install of Windows 8, this is just the Mail app. If you have more apps that support sharing, you'll see them, like Figure 9-3.

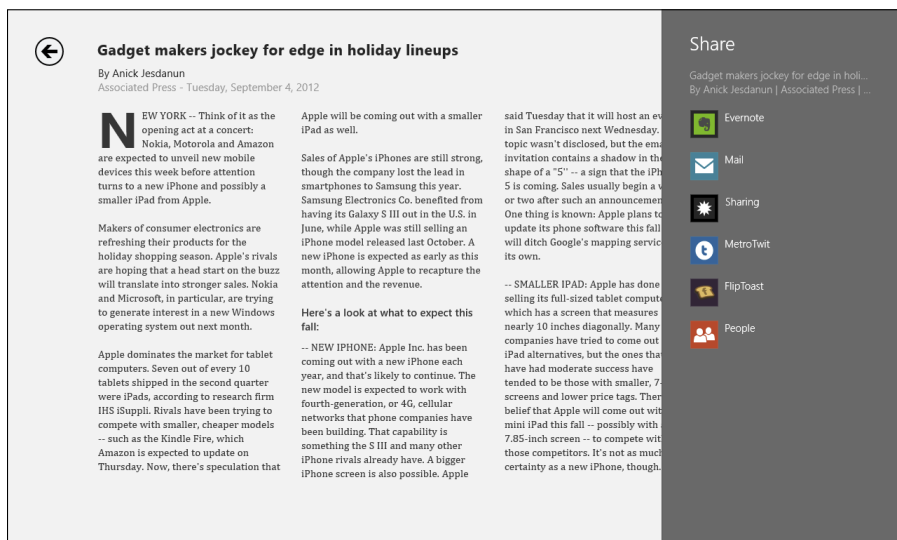


Figure 9-3:
The Share
charm
in use.

Touching the Mail share icon brings up a mashup between the Mail app and News that should look like Figure 9-4. Because the Mail app declared that it could share text and the News app agreed to provide it, the mail app can pre-populate whatever it does (in this case, a mail message, obviously) with text it will get from the News app.

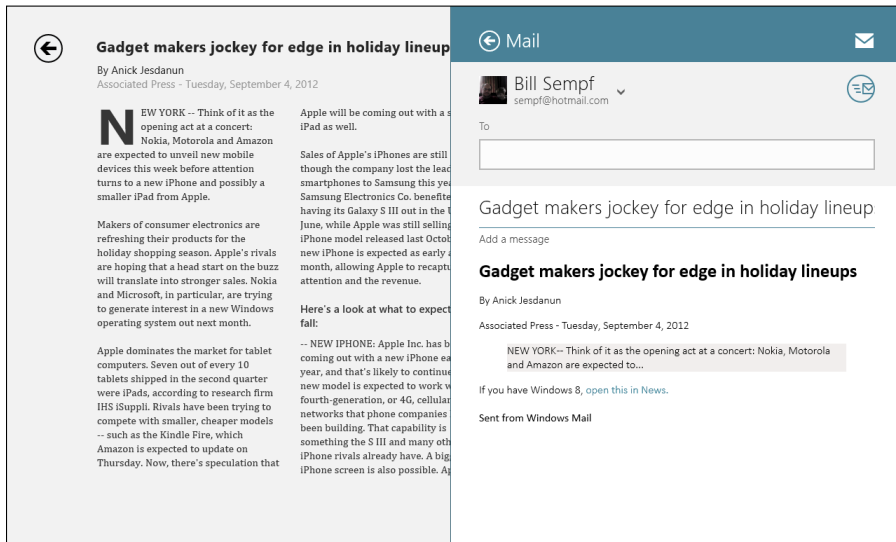


Figure 9-4:
Mail me a
story!

In order to make this work, you need two things:

- ✓ An agreement for the content producing app (in this case, Contoso Cookbook) to provide certain content types
- ✓ An app that declares itself as a sharing target (in this case, Mail)

In the next couple of sections, I take a look at both of those.

Allowing sharing from your app

Providing source content from the Windows 8 client is crazy easy. There are basically five things that happen:

1. Your app has to register with the `DataTransferManager`.
2. When the system detects that the Share charm has been tapped, it sends a `datatransfer` event to the running app — hopefully yours!
3. Your app fills a `datatransfer` object.
4. The system filters a list of target apps and QuickLinks.
5. Your app fulfills any asynchronous calls and returns the power to the user.

Coding for this is a small-scale exercise, especially if you have simple content, like text. Check out these steps:

1. Get a reference to the `DataTransferManager`.

```
var dataTransferManager = Windows.ApplicationModel.  
    DataTransfer.DataTransferManager.  
    getForCurrentView();
```

2. Add an event listener for the `daterequested` event.

```
dataTransferManager.addEventListener("daterequested",  
    function (e) {}
```

3. In the anonymous function, set the title and the text.

```
dataTransferManager.addEventListener("daterequested",  
    function (e) {  
        e.request.data.properties.title = "Windows 8  
        Programming for Dummies.";   
        e.request.data.setText("I'm reading the chapter  
        on contracts, especially sharing. This is my  
        share example!");  
    });
```

4. Put all of this right before the `processAll` statement. The `onActivated` handler should look something like this:

```
app.onactivated = function (eventObject) {  
    if (eventObject.detail.kind === Windows.  
        ApplicationModel.Activation.ActivationKind.  
        launch) {  
        if (eventObject.detail.previousExecutionState  
            !== Windows.ApplicationModel.Activation.  
            ApplicationExecutionState.terminated) {  
            // TODO: This application has been newly  
            launched. Initialize  
            // your application here.  
        } else {  
            // TODO: This application has been  
            reactivated from suspension.  
            // Restore application state here.
```

```
}  
var dataTransferManager = Windows.  
ApplicationModel.DataTransfer.  
DataTransferManager.getForCurrentView();  
dataTransferManager.addEventListener("data  
requested", function (e) {  
    e.request.data.properties.title = "Windows  
8 Programming for Dummies.";  
    e.request.data.setText("I'm reading the  
chapter on contracts, especially sharing. This  
is my share example!");  
});  
WinJS.UI.processAll();  
}  
};
```

After that little bit of code, you can just press F5 and run the app. There won't be much to look at, I admit, but press Windows+C and click the Share charm and you'll get Figure 9-5. The values are preset for the user, just as they should be.

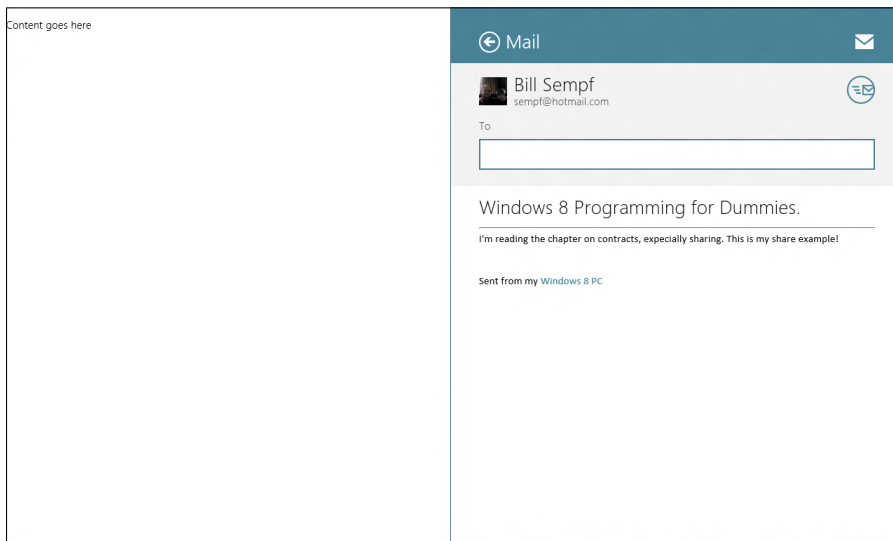


Figure 9-5:
Sharing.

Text isn't the only thing you can share. There are properties to handle a number of common formats, and you have the ability to improvise too. The built-in formats include

- ✓ Text
- ✓ Link
- ✓ Image
- ✓ File
- ✓ HTML
- ✓ Delayed rendering (that is, "I ain't sure what it is, so I'll tell you later")
- ✓ Custom data
- ✓ Error

Sharing from your app is not what brings the users, but being a sharing target does. You want people to use your app to share to the hot new social network — that's where the eyes are.

Becoming a share target

In order to be a share target, you need to do just two things:

- 1. The application has to be registered as a share target in its package. `appxmanifest`.**
- 2. You need to handle the `Share ActivationKind` in the `Activated` event.**

This is a lot like Search, and that's by design. The ideas behind consuming these Charm contracts should be very similar, and the use of these charms should be standardized too.

Setting up the `appmanifest`

To register the application as a search target, the `appmanifest` has to be changed. You can, in fact, edit it by hand (it's just XML), but Microsoft was nice enough to make an easy-to-use UI to do this, as you can see in Figure 9-6.

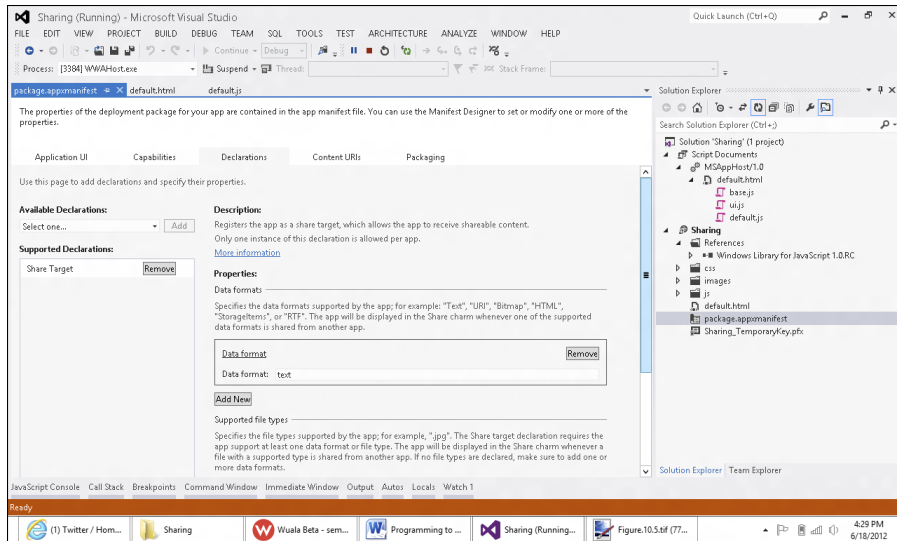


Figure 9-6:
The appx-manifest editor.

1. Double-click the `package.appxmanifest` file to open the editor.
2. Click the **Declarations** tab.
3. In available Declarations, select **Share Target** (it's the last one).
4. Under **Data Formats**, click **Add New**.
5. Here, type the data formats you accept. **Text** is a good choice.
6. Under **Supported File Types**, you can enter which files you support, or check **All Types** (which is what I did).
7. Click **Save**. You're done!

So that's the appxmanifest. If you open the code, you can see the change you made:

```
<Extensions>
  <Extension Category="windows.shareTarget">
    <ShareTarget>
      <SupportedFileTypes>
        <SupportsAnyFileType />
      </SupportedFileTypes>
      <DataFormat>text</DataFormat>
    </ShareTarget>
  </Extension>
</Extensions>
```

Next, it's time to handle the case when your app is activated by a share.

Handling activation

When someone shares to your app, you can bet dollars to donuts that your app won't be active. There is a chance that it could be snapped or filled, but I am betting it'll at least be suspended. As such, a share action activates your app.

To handle this, you just need a new `if` block in the `app.activated` event handler. There is already one there for `launch`, and maybe one for `search` if you're playing along at home. Now you need one for `share`.

```
if (eventObject.detail.kind === Windows.ApplicationModel.Activation.  
    ActivationKind.shareTarget) {  
    //do magic here  
}
```

That's all there is to it. You can do whatever you want (within reason) in that setup and handle the content that was sent. That content is available from the `shareOperation` class. You can check the type and then deal with the content. For instance, this code takes the text sent and sets the content of a `div` in the UI.

```
if (shareOperation.data.contains(  
    Windows.ApplicationModel.DataTransfer.StandardDataFormats.text)) {  
    shareOperation.data.getTextAsync().then(function (text) {  
        if (text !== null) {  
            document.getElementById("output").innerText = text;  
        }  
    });  
}
```

If the data were an image, you would effectively do the same thing and get access to the bits that way. You can look at a lot of different examples in the `ShareTarget` sample in the SDK.

Settings

The application settings, like Search and Share, shouldn't be a principal part of the user interface anymore — it should be handled by the Charm bar. The Settings charm is used for nearly all Windows settings now, and it should also handle the settings for your app.

Like the other charms, the Settings charm is context-sensitive. Windows makes sure that the apps get a chance to handle the event that occurs when the charm is clicked, and from that you can make sure that the application responds appropriately.

Clicking the Settings charm launches a Settings flyout, shown in Figure 9-7. Fortunately, it's just a HTML form and you can script it up without a problem as long as everything is named based on the values you stated in the contract. You can have multiple panes and implement all of the controls you covered in Part II and give the user a complete control experience over your app.

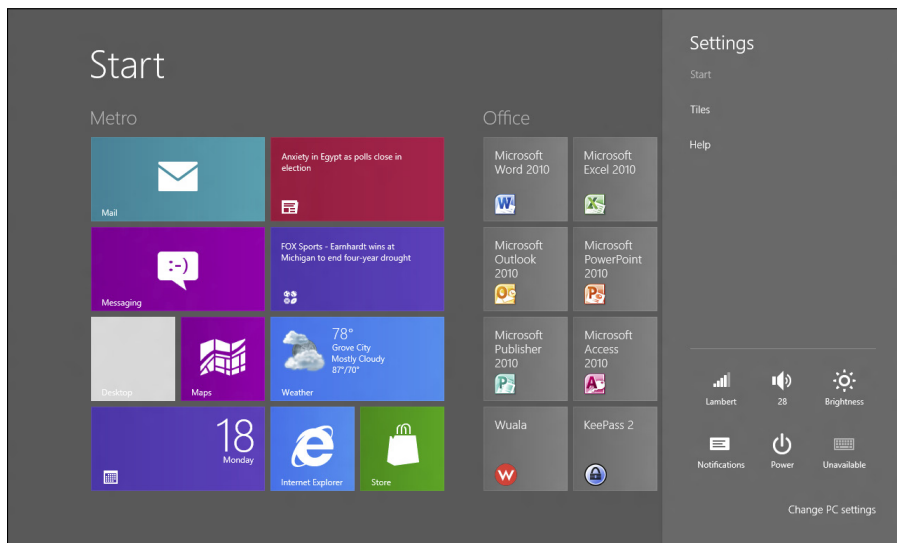


Figure 9-7:
The Settings
flyout.

Take a look at implementing a simple Settings contract:

1. **Open a new blank JavaScript Windows 8 project.**
2. **Right-click the project and add a new HTML page.**
3. **Put the following code in the HTML page.** It uses the WinJS control `SettingsFlyout` to make sure that the binding is compatible with expectations of the Settings charm. In the `win-content` area, you can set up whatever settings you wish.

```
<!doctype HTML>
<html>
  <head>
    <title>Settings flyout</title>
  </head>
  <body>
    <div id="settingsFlyout" data-win-
      control="WinJS.UI.SettingsFlyout"
      aria-label="App defaults settings flyout"
      data-win-options="{width: 'wide'}">
      <div class="win-header" background-
        color:#dbf9ff">
        <button type="button" onclick="WinJS.
          UI.SettingsFlyout.show()" class="win-
            backbutton"></button>
        <div class="win-label">Settings</div>
        
        </div>
        <div class="win-content">
          This is where the settings go
        </div>
      </div>
    </body>
  </html>
```

4. The only remaining part is to wire up the event handler.

After the DOM has loaded, you just need to code up something in the `onsettings` event in the `WinJS.Application` class. This code goes right before the `app.start` in the `default.js` file in the basic template.

```
WinJS.Application.onsettings = function (e) {
  e.detail.applicationcommands = {
    "settingsFlyout": { href: "Settings.html",
      title: "Settings"},
  };
  WinJS.UI.SettingsFlyout.populateSettings(e);
}
```

WinJS makes this really easy on us. You don't have to do a lot of work to get the flyouts wired up. There is WinRT code that is being invoked here by the WinJS objects, however.

You can see the end result of our labor in Figure 9-8.

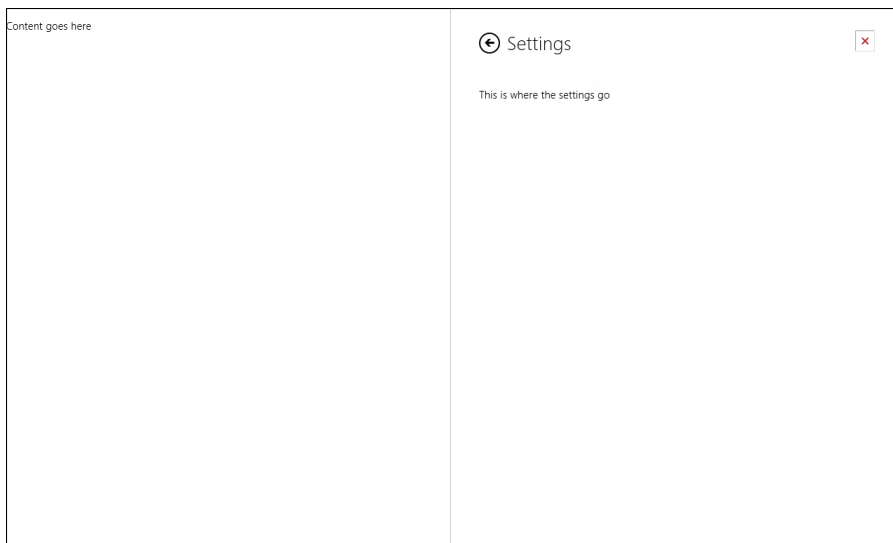


Figure 9-8:
Our new
Settings
flyout.

Playing to Devices

Working with media — audio, pictures, and video — is a core part of Windows 8 applications and the consumption of connected apps in general. In the Windows 8 world, getting media from one place to another is central to the idea of connected applications. Video should be sent to a TV, pictures to a phone or digital frame, and audio to home stereo.

Getting the gist of Play To

The Play To contract is all about that getting media from one place to another. It's run from the Devices charm and is designed to get media from an application (say a movie manager) to a DLNA-compliant (Digital Living Network Alliance) device. There are more of those devices than you think. It's UPnP (Universal Plug and Play), a very common protocol, with perhaps 10,000 different products supporting the protocol.



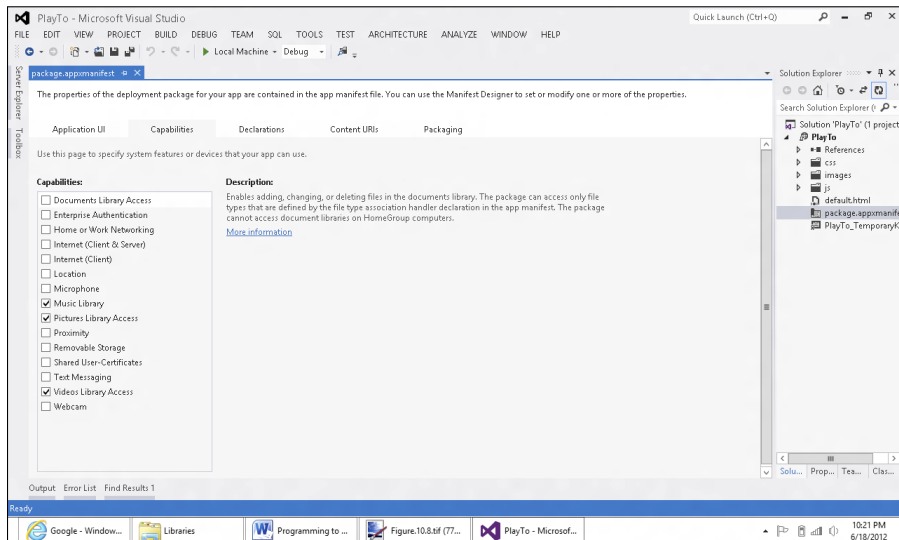
Why bother setting up Play To? Two reasons. I've already established that the Charm bar is going to be where users go to look for functionality like search and device access. Also, the Play To contract is immune from suspension. If the user picks a video, plays it to a DLNA television, and then puts your app in the background, that video keeps playing. In a normal scenario, the connection would be severed.

Coding a Play To example

Implementing Play To isn't much more complex than any of the other contracts I've discussed. Here's a look at how best to deal with it:

1. Create a new JavaScript Blank template in Visual Studio. Name it PlayTo (or download the sample).
2. Double-click the package.appxmanifest and click the Capabilities tab.
3. Check the Music Library, Pictures Library Access, and Videos Library Access items in the Capabilities list, as shown in Figure 9-9.

Figure 9-9:
Adding the
Capabilities
list.



4. Get a reference to the PlayToManager and wire up the source requested event.

The following code should do it for you:

```
var ptm = Windows.Media.PlayTo.PlayToManager.
    getForCurrentView();
ptm.addEventListener("sourcerequested",
    sourceRequestHandler, false);
function sourceRequestHandler(e) {
    try {
        var sr = e.sourceRequest;
        var controller;
        try {
```

```
        controller = mediaElement.msPlayToSource;
    } catch (ex) {
        id("messageDiv").innerHTML +=
            "msPlaytoSource failed: " + ex.message +
            "<br/>";
    }
    sr.setSource(controller);
} catch (ex) {
    id("messageDiv").innerHTML +=
        "Exception encountered: " + ex.message +
        "<br/>";
}
}
```

5. You'll need a PlayTo target, assuming you don't have a fancy monitor attached to your workstation (If you did, you could use that!).

I would recommend starting Media Player. You can do this by pressing the Windows button and then clicking on Media Player. You might have to set up the app. (The default settings are not horrible.)

6. Press F5 to run the app.

7. Click the Devices charm and check out the Media Player goodness.

That's Play To. Set up a program to send media to a device and change the atmosphere.

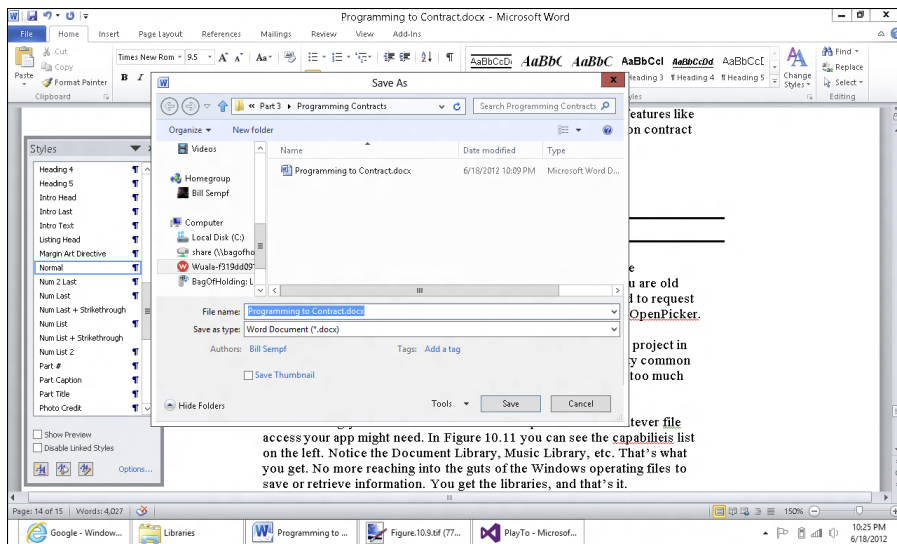
File Picking from App to App

One of the hardest philosophical changes to make in Windows development is that of the isolation of applications. In Win32 or .NET development, an app can connect to a database, touch any file (almost) in the file system, and communicate with Windows services. In Windows 8, that is rarely the case.

Communication with parts of Windows, as you have seen with features like Search and Share, is done with contracts. By far, the most common contract that you'll see is the File Picker.

The File Picker is the new `CommonDialog`. You'll remember the `CommonDialog` (in Figure 9-10) from .NET or even VB6 if you are old enough. The File Picker works about the same way, but you need to request permission to see the files, and then you need to implement `FileOpenPicker`.

Figure 9-10:
The Common
Dialog.



Getting this set in action is pretty straightforward. Create a new project in Visual Studio (or use an existing project). File picking is a pretty common function, so you can stick it into an existing application without too much difficulty.

The first thing you need to do is declare the capabilities for whatever file access your app might need. In Figure 9-9, you can see the capabilities list on the left. Notice the Documents Library Access, Music Library, and so on. That's what you get. No more reaching into the guts of the Windows operating files to save or retrieve information. You get the libraries, and that's it.

Check off whatever capabilities you might need. For this example, the Documents Library Access will do just fine. Just know that's what you get — other files outside the libraries won't be accessible.

After the functionality is available, you just have to invoke the picker in your code. This can be from the App bar (recommended) or from content, if you can do that. Just remember that adding an Open File button to the screen is verboten in Windows 8.

Here is a sample of the code that might be used to make a picker like the one in Figure 9-10:


```
var fileOpenPicker = new Windows.Storage.Pickers.  
    FileOpenPicker();  
fileOpenPicker.suggestedStartLocation = Windows.Storage.  
    Pickers.PickerLocationId.documentLibrary;  
openPicker.fileTypeFilter.replaceAll([".docx", ".doc"]);
```

Implemented thusly, the picker works about the same way as the .NET version, except you are restricted in terms of what folders on the host system you can see.

Chapter 10

Talking to the Internet

In This Chapter

- ▶ Creating different layouts
 - ▶ Having a chat with the Internet
 - ▶ Handling data
-

Windows Store applications make good mashups. A *mashup* is a new way of viewing or using existing applications. Getting geocache locations from www.geocaching.com and showing them with Bing maps is a mashup. Allowing lookup of .NET classes from MSDN, and then getting code samples from CodePlex is a mashup.

A mashup doesn't even have to involve more than one source of data. A blog viewer in a mashup of sorts — it is mixing the blog content, traditionally viewed on the web — with the Windows 8 design surface. This might more accurately be called a viewer, but this is one thing for sure — it consumes an external feed of data.

Consuming external feeds of data ranks among the number one strengths of WinRT, and the Windows 8 way of doing things. There are a few different ways to do it, and an infinite number of ways to present the data. In this chapter, I show you how to get to burning some feeds, but I start with a couple of layouts.

Building Different Layouts

Most data is organized hierarchically. You have a category, with perhaps some categories below it, and then a set of detail items within the categories. A few ways exist to present these to users; this is the provenance of User Experience (UX) experts.

Fortunately for us, a few UX folks work at Microsoft, and templates are available to get us started on the path to an excellent user experience. Here I cover two, which handle the majority of cases of hierarchical data that you'll encounter.

The Grid Application template

The Grid Application template presents a user experience usually called the Tile layout — a setup that is similar to a Memory game. A series of cards are set up in a XY coordinate grid that can be sorted and organized by the cards' categories.

For some reason, Microsoft calls this the Grid Application template, and it works very well. In the Windows Store style, it gives the user a chance to use a finger to select items or change the sorting and filtering without too much trouble. You can see it in Figure 10-1 in default form.

Make an application using the Grid Application template and see how it works. In Visual Studio, start a new project and select Grid Application. I named mine something sensible, like GridApplication.

Figuring out the files

There are a lot of files in the solution, huh? When you are looking at one of these templates, just focus on the HTML files, just like a website. That's where the magic is. The user interface drives the whole deal, so start there.

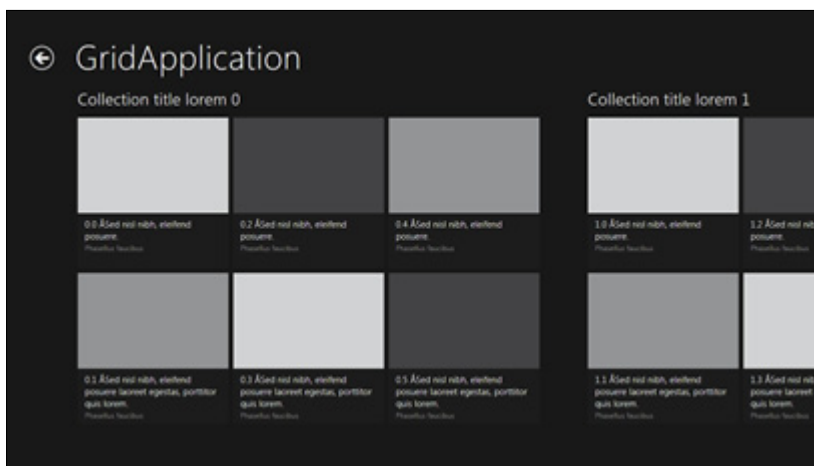


Figure 10-1:
The Grid
Application
template.

This template has three user interface files:

- ✓ **default.html:** In the root of the directory, this is the holding page for the fragments in the solution. (For more on fragments, see Chapter 4.) This page has a pointer to the first fragment loaded in the body tag; just look for it.

```
<body data-homePage="/html/landingPage.html">
```

- ✓ **landingPage.html:** In the HTML directory, this is the pointed-to page. This is the first page of content that is loaded in the `contentHost` `div` in `default.html`. It owns the collection of collections — the list of Collection Title ABC, Collection Title XYZ, and so on, is what the landing page shows for us.
- ✓ **collectionPage.html:** This is the repeated bit under each collection title. It is what manages the actual grid shown in Figure 10-1 — the images and text. If you look at the `itemTemplate`, you can see the basic layout — image, title, subtitle, and description.

```
<div class="itemTemplate" data-win-control="WinJS.Binding.Template">
  <div class="largeIconTextTemplate">
    <div class="largeIconTextTemplateImage"
      data-win-bind="style.backgroundColor:
        backgroundColor"></div>
    <div class="largeIconTextTemplateBackground">
      <div class="largeIconTextTemplateLargeText win-
        itemTextStrong"
        data-win-bind="textContent:
          title"></div>
      <div class="largeIconTextTemplateSmallText
        win-itemTextTertiary" data-win-bind="textContent:
          subtitle"></div>
      <div class="largeIconTextTemplateMediumText win-itemText"
        data-win-bind="textContent:
          description"></div>
    </div>
  </div>
</div>
```

- ✓ If you click on an item, you go to the `detailPage.html` layout. This is a classic detail page, hosted inside `default.html`.

So you have `default.html`, containing `landingPage.html`, which in turns contains `collectionPage.html`, all through the magic of JavaScript. This kind of nesting is at the core of layout for Windows Store-style apps, so consider it closely.

Accessing the data

The naming standard for the JavaScript files in the templates is excellent — files are named the same and are easy to find. The `collectionPage` has the

content, and it calls the data methods, so in this template, the `collectionPage` has the data. Open up that page and scroll down to see the sample data.

At the end of that page, you can see how `GetGroups` and `GetItems` are just populating the arrays with data manually. This is all handled at the end of the JavaScript code block. Remember, it's just a big function, so it is handled in order:

```
var pageData = {};  
pageData.groups = getGroups();  
pageData.items = getItems();  
  
WinJS.Namespace.define('collectionPage', {  
    fragmentLoad: fragmentLoad,  
    itemInvoked: itemInvoked  
});
```

So how to get new data in here? Replace the code in the `getGroups` method. I give you some suggestions for that in “Communicating with the Internet,” later in this chapter, but first look at the other major layout in the templates.

What the Grid Application template is good for

Not every layout is good for every type of data. Clearly the Grid layout is best for items with important pictures: product catalogs, sports teams, and hotel listings.

For something that might not have images, a list-ier format might be nice. Fortunately, you can turn to the List layout.

List layout

List layout is exactly what it sounds like: an item list that is designed for a descriptive listing that leads to a comprehensive detail view. There are two main differences: it is on one page, and the list format makes it better for items with no pictures.

Figure 10-2 shows the startup page, denoted in the `body` tag of the default. `html` file. It is actually the `ListView` page for a selected category. If you click the back arrow, you go back to the category page.

Figuring out the files

Like the Grid Application template, this template uses the generally accepted fragment model I explained in Chapter 4. As such, it has a major page that contains the navigation and whatnot, and then separate pages for the content. The major page is `default.html` and the fragments are in the `html` folder.



Figure 10-2:
The List
layout.

- ✓ `splitPage.html` is the page where the application starts. It shows one category's items and the details to the right.
- ✓ `categoryPage.html` is accessed with the Back button from `splitPage`. This is a grid view of the categories that were all just a list in the Grid Application template.



Honestly, I don't think the default List layout is the best implementation of a UI. The first ten or so times I ran this sample, I couldn't figure out where the categories were. The Back button is obviously there, but it isn't obvious (to me anyway) what it does.

That's neither here nor there, however. The key is in the layout field of the data `win-options` parameter. For the grid, use layout `{type: WinJS.UI.GridLayout}`. For the list, you use layout `{type: WinJS.UI.ListLayout}`.

Accessing the data

The data access point is the same for the List template using the `ListLayout` as the template using the `GridLayout`. You'll find the data in the `splitPage.js` file.

If you were getting raw data from a service, like I discuss in “Communicating with the Internet” section later in this chapter, you need to replace the code inside the `getGroups` and `getItems` methods. Microsoft used arrays in the official samples in `dev.microsoft.com`, but I use JSON. Not that much of a difference.

What the List layout is good for

Figure 10-2 shows image slots, but the code could easily be edited to handle just a text list, like for a blog, for instance. You have a title, maybe a slug for the list, and then the whole article for the detail. There is no image (usually) to show in a GridView.

Take a look at the template used by ListView in splitPage.html:

```
<div class="itemTemplate" data-win-control="WinJS.Binding.Template">
  <div class="largeIconTextTemplate">
    <div class="largeIconTextTemplateImage" data-win-bind="style.
      backgroundColor: backgroundColor"></div>
    <div class="largeIconTextTemplateBackground">
      <div class="largeIconTextTemplateLargeText win-itemTextStrong"
        data-win-bind="textContent: title"></div>
      <div class="largeIconTextTemplateSmallText win-itemTextTertiary"
        data-win-bind="textContent: subtitle"></div>
      <div class="largeIconTextTemplateMediumText win-itemText" data-
        win-bind="textContent: description"></div>
    </div>
  </div>
</div>
```

Taking out the `largeIconTextTemplateImage` div removes that image and makes it appropriate it for something that doesn't support an image. You'll see more of this when you modify the template for a blog in the following section.

Communicating with the Internet

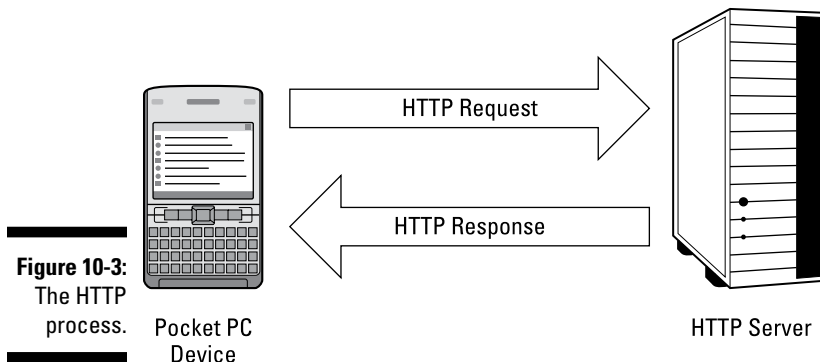
Houston, we have layout. Now we need data.

The Windows Store style is about the look of applications, indeed, but there is an implicit dependence on data — information that is drawn from network sources. That focus on the look of apps has a few assumed caveats. They are

- ✓ **Sources:** The information has to come from somewhere. Maps, posts, people, images of cats — whatever it is has to have a source.
- ✓ **Connection:** To get to the source, you need a connection to the Internet. The data is out there. You just need to get to it.
- ✓ **Format:** After the information gets to you, you need to be able to handle the format the data comes in. Data formats are the least of your worries, though. From RSS to ATOM, JSON to XML, whatever it is, tools exist to handle it. That's something you can play with as you go along. For now, get out and get some data.

The original *XMLHttpRequest*

In order to get information from a web server to a scripting source, JavaScript (along with a bunch of other languages) implemented a HTTP Request method. This is essentially a GET wrapped in the language of choice's semantics, taking advantage of the relative simplicity of the HTTP protocol you see in Figure 10-3.



If you understand what a web browser does, you understand how the `XMLHttpRequest` works. Essentially, it is just like typing a URL into the address bar of a browser, except the results are returned to the script.

The script bit is key though — this is how jQuery and AJAX in general made their splash. The results of the request — HTML, XML, or JSON — are returned to the script to parse and insert anywhere in the DOM the script chooses. That's how this whole *web 2.0* thing got rolling.

Nothing changes when it comes to the Windows 8-style applications. The data is out there, and you need to get it. `XMLHttpRequest` is the path — at least in the JavaScript language. It gets the data in some format (you have parsers, no matter) that you can then insert into the application as needed.

The only problem is timing. The Window Store style guide says that any transaction that takes longer than 50 milliseconds should be asynchronous. Clearly, an Internet call takes longer than 50 milliseconds — have you tried to load YouTube lately? Anyway, something has to be done.

It's possible that you could use the callback mechanism in JavaScript to do all of this. Or you could just use the XHR library that the Windows 8 tools team at Microsoft was kind enough to build.

XHR: Microsoft's little secret

XHR is the first of many toys that Microsoft created in the WinJS library. WinJS, which I cover in Chapter 2, is a set of JavaScript methods and CSS styles that help us do things in the Windows Store style. Remember the whole “Things more than 50 milliseconds need to be async” rule? Well, XHR is for that.

WinJS.xhr works a lot like `XmlHttpRequest` because it isn't just based on `XmlHttpRequest`, it actually calls `XmlHttpRequest`. What it brings to the table is the async operation, which I talk about more in just a second. For now, check out the parameters for the XHR method:

- ✓ **Method:** The method can specify the call type. This is as you would expect — the usual HTTP types. In a REST-driven environment, you get to use the neat types that you don't use very often:
 - GET
 - HEAD
 - POST
 - DELETE
 - PUT
 - TRACE
 - CONNECT
- ✓ **URL:** The URL is the only required characteristic. Remember, it has parameters built into the query string.
- ✓ **User and password:** These are for authentication. Please use those only if you are using SSL (Secure Sockets Layer).
- ✓ **Headers:** Headers are an optional field that are passed directly to the `XMLHttpRequest.setRequestHeader()` method.
- ✓ **Data:** Data is a parameter sent directly to the `XMLHttpRequest.send()` method. You can look at that documentation (really good docs on Mozilla's site, (sadly) better than Microsoft's — <https://developer.mozilla.org>). It's used for posting data that isn't in the URL, like binary data and such.

Modern versions of `XmlHttpRequest` have an `onreadystatechange` event handler that accepts a callback method as a property. This allows the thread to continue, and the handler will be called when the object returns a value.

WinJS.xhr basically takes those four elements — `instance`, `open`, `onreadystatechange`, and `send` — and rolls them into a method that returns the promise object. You can see it in the source code for XHR.

```
function (c, e, p) {
    req = new XMLHttpRequest();
    req.onreadystatechange = function () {
        if (req.readyState === 4) {
            if (req.status >= 200 && req.status < 300) {
                c(req);
            } else {
                e(req);
            }
            req.onreadystatechange = function () { };
        } else {
            p(req);
        }
    };

    req.open(
        options.type || "GET",
        options.url,
        // Promise based XHR does not support sync.
        //
        true,
        options.user,
        options.password
    );
    req.responseType = options.responseType || "";

    Object.keys(options.headers || {}).forEach(function
        (k) {
            req.setRequestHeader(k, options.headers[k]);
        });

    req.send(options.data);
}
```

Check out the overall message of that code — you have the same four elements, right in the method:

- ✓ The `XmlHttpRequest` was instantiated.
- ✓ An `onreadystatechange` method was set.
- ✓ The open parameters were set.
- ✓ Someone pressed Send.

A promise kept

The `XHR` method returns a promise. A *promise* is the new model for asynchronous behavior in JavaScript as it relates to Windows Store-style apps. It is an object that can be expected to return a value sometime in the future. Any resultant functionality can be run as a fluent style chain from the promise.



The promise is actually a JavaScript standard proposal. The specification is called *Promises/A* and can be read at www.commonjs.org.

Look at the `XMLHttpRequest` as it differs from `XHR`. After you spin up a new instance of `XMLHttpRequest`, you call `open` to configure a call to an external resource, and then `send` to pass it to the network stack.

```
var client = new XMLHttpRequest();
client.open("GET", "sample.html");
client.send();
```

At this point, the thread that the request is running in is locked. Everything stops while the network goes and retrieves the file referenced in the `open` method. In the preceding section on `XHR`, I showed how Microsoft used the `onreadystatechange` event handler to set up an asynchronous pattern that returns a promise object.

So, now you have a promise. The promise has an important function, called `then`. Unsurprisingly, the function that is passed to `then` is called when the promise returns its results. It is as if you are saying “I promise to return this, and then you can loan it to someone else.”

Promise parameters

A promise accepts three functional parameters and provides three return values. The three functional parameters are the return handler, an error handler, and a progress report. The three parameters that you can use in those functions are the returned result, any errors, and a measure of the progress.

A complete implementation of a promise, then, looks like this:

```
WinJS.xhr("http://www.sempf.net")
    .then(
        function(result) { /* handle the returned object */ },
        function(error) { /* handle any errors */ },
        function(progress) { /* report progress */ }
    );
```

Notice that you can build the functions right in the `then` method signature. Functions are data in JavaScript. You could have put function names in there as well and reused logic.

Composing a then method

So any function that returns a promise has a `then` method that fired the contents on completion, error, and progress. If the object returned by promise is another promise, chain another `then` on the end of the line.

For instance, if you needed to send a request to purchase an item and get that item's details, you might do this:

```
WinJS.xhr("http://www.sempf.net/order/1234/")
    .then(function(result) { WinJS.xhr("http://www.sempf.net/item/ + result +
        "/" )})
    .then(function(result) {return doSomething(result)});
```

This has the effect of getting the values from order 1234 (imagine that is a service that gets an order by number), and after it returns, it passes the result on to another service that then looks up the results in an item search. After that returns, some final method is called.

Notice that I skipped the error handling and progress reporting. I did so for simplicity, and to point out that they are optional.

Handling Data

After you have a user interface model and a data access model, it's time to write a program that uses external data. For this example, I use Flickr because I'm relatively sure it will still be around by the time you read this.

Flickr has a public data feed that works well for this application because it has some metadata about the image, like the title and whatnot, and then the image itself. This lends itself well to the Grid Application template because every item has an image and there is no category list.

Setting the stage

Get set up with a Grid Application template:

1. In Visual Studio, click on **File** ⇨ **New Project**.
2. Select the **Grid App** template and name the project **FlickrViewr** (or anything else you want to).
3. Click **OK**. Visual Studio makes the project for you.
4. You're not going to use the **groupedItemsPage**. In **default.html**, change the **contenthost div** to **groupDetailPage.html** in the **data-win-options** field.

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

5. Delete the **groupedItems.html**, **groupedItems.css**, and **groupedItems.js** files. You might have to click **Show All** in the **Solution Explorer** to see all of these files, depending on how your Visual Studio install is set up.

You don't need them, so why mess up the solution?

6. In the `groupDetailPage.html`, find the H1 with a class of `titlearea` and replace it with some static text, like this:

```
<h1 class="titlearea win-type-ellipsis">
    Flickr Public Feed
</h1>
```

7. Find the code that used to update the headers and comment it out. It's in `groupDetailPage.js`:

```
element.querySelector("header[role=banner]
    .pagetitle").textContent = group.title;
```

Okay, that should do it. Now you just need to get the data in there.

Calling the feed

At this point, all of the sample data is in `data.js` in the `js` folder. You're going to replace the calls to get the data from the arrays in that file with calls to get the data from the Flickr API.

The Flickr feed

First, look at the Flickr data feed just to see what is there. The URL that gets things going is

```
http://api.flickr.com/services/feeds/photos_public.gne
```

In order to get something other than ATOM format, you need to use the format parameter. So put this in your browser:

And you will get some JSON, which looks like this:

```
jsonFlickrFeed({
  "title": "Uploads from everyone",
  "link": "http://www.flickr.com/photos/",
  "description": "",
  "modified": "2012-03-07T04:50:49Z",
  "generator": "http://www.flickr.com/",
  "items": [
    {
      "title": "IMG_6401",
      "link": "http://www.flickr.com/photos/jumpstartweb/6814740920/",
      "media": {
        "m": "http://farm8.staticflickr.com/7052/6814740920_4521f82eb3_m.jpg"
      }
    }
  ]
})
```

```

    "date_taken": "2011-12-10T12:25:50-08:00",
    "description": " <p><a href=\"http://www.flickr.com/people/
                    jumpstartweb/\">JumpStartWeb</a> posted a photo:</p> <p><a
                    href=\"http://www.flickr.com/photos/jumpstartweb/6814740920/\"
                    title=\"IMG_6401\"><img src=\"http://farm8.staticflickr.com/70
                    52/6814740920_4521f82eb3_m.jpg\" width=\"240\" height=\"160\"
                    alt=\"IMG_6401\" /></a></p> ",
    "published": "2012-03-07T04:50:49Z",
    "author": "nobody@flickr.com (JumpStartWeb)",
    "author_id": "63672446@N06",
    "tags": ""
  }
]
})

```

That JSON is some data you can use. You just need to get the Windows 8 app using it instead of the information in `data.js`. Starting around line 14 of the template code of `groupDetailPage.js`, the data in `data.js` gets integrated into the datasource of the page.

```

group = (options && options.group) ? options.group : data.
groups.getAt(0);
items = data.getItemsFromGroup(group);
var pageList = items.createGrouped(
function (item) { return group.key; },
function (item) { return group; }
);
var groupDataSource = pageList.groups.dataSource;

```

You need to structure the `groupDataSource` as a container for that Flickr data. First use the XHR method to go get it, and then reorganize the data to fit the plan.

Using *xhr*

In `data.js`, I add some code to call the Flickr feed near the end. There is even a comment down there in the template:

```

// TODO: Replace the data with your real data.
// You can add data from asynchronous sources whenever
// it becomes available.

```

Nice! Anyway, just before the namespace definition code, I add a call to the Flickr feed. It looks like this:

```

WinJS.xhr({ url: "http://api.flickr.com/services/
               feeds/photos_public.gne?format=json&nojsoncall
               back=1" }).then(processPhotos, processError);

```

So you pass in the URL to Flickr. The `nosoncallback` parameter prevents them from sending us JSONP. The promise that is returned gets two methods (I ignored the `progress` method for this example): `processPhotos` and `processError`.

`ProcessPhotos` is where I'll populate the data structure that gets bound in the HTML. The function called gets one parameter — the result of the query — which is understood. You won't see it in the reference, but it's there. I called it `result`, and that gives me the opportunity to do a little reorganizing.

```
function processPhotos(result)
{
    var photoData = JSON.parse(result.responseText);
    //bind here
    data.items.forEach(function (item) {
        list.push(item);
    });
}
```

I use `JSON.parse` to change the text string to an actual JavaScript array, and then flip through the items and fill up the `BindingList` that was defined elsewhere in the template.

The other method was `processError`, and that's important too. I don't have a big error handling system here, so I just write to the console. You could write to a text file or the event viewer.

```
function processError(error) {
    console.log(error.message);
}
```



Put a breakpoint on the code inside the `processPhotos` and `processError` method. You'll learn more than you think. You can take a look at the content of those objects when the project is paused and see more that you can get into on your own.

Now you have the data in the `photodata` variable, and that's what you want. Put your mouse over the variable in the debugger to check out all that is in there. Remember that the data in the feed is always changing.

After the array is populated, it's just a matter of changing the default template layout in the HTML to support the new data, and then you're good to go.

Formatting the results

Getting data on the screen will seem familiar to you, if you have done any databinding in any Microsoft-developed UI code in the past. From ASP Classic to XAML, the same basic principles apply.

This time, though, Microsoft went with pretty strict HTML5 specifications, which is nice for a change. To databind, declare a Repeater template and an Item template. The Repeater template formats the list itself, and the Item template formats the individual items in the list.

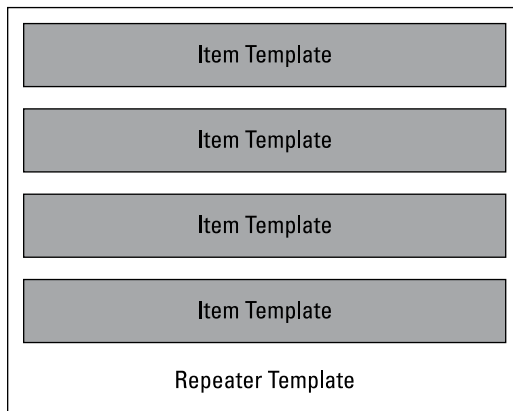
The Repeater example in Figure 10-4 would read something like this:

“The Repeater template is a box, white inside with a black outline. It should be as wide as the screen allows, with any text at the bottom— and the text is black. Stack any items vertically.”

The item example would read:

“The Item template is a box, a little smaller than its container. The border is black, and the body is light gray. Any text should be centered and black.”

Figure 10-4:
Databinding
with
templates.



In the HTML file that has the list, you define each of these separately, in order to be able to replace the Item template look within the Repeater template if needed.

Repeater template

Take the template text, for example. The Repeater template is near the bottom of the page: a section tag with the aria-label “Main Content”. Aria-label is a global property in HTML5. It is used to give an item a recognizable name that is not displayed on the screen.

```
<section aria-label="Main content" role="main">
  <div class="grouplist" aria-label="List of this group's items" data-
    win-control="WinJS.UI.ListView" data-win-options="{ selectionMode:
      'none' }"></div>
</section>
```


The `div` inside the section tag has a few attributes that aren't in HTML5 — they are specific to WinJS. Those attributes are `data-win-control` and `data-win-options`.

`Data-win-control` defines what kind of control the `div` should appear as. WinJS comes with a number of controls, all of which are specifically styled for the Windows 8 UI. They aren't all repeaters, either. Here is a partial list of the controls that come with WinJS:

- ✓ **AppBar:** Displays commands.
- ✓ **DatePicker:** Enables the user to select a date.
- ✓ **FlipView:** Displays a collection of items, one item at a time.
- ✓ **Flyout:** Displays a message that requires user interaction. (Unlike a dialog box, a `flyout` doesn't create a separate window.)
- ✓ **ListView:** Displays a collection of items in a Grid or List layout.
- ✓ **HtmlControl:** Displays an HTML page.
- ✓ **Menu:** A menu flyout for displaying commands.
- ✓ **Page control:** A set of HTML, JavaScript, and CSS that describes a page. You can navigate to a Page control or use it like a custom control.
- ✓ **Rating:** Lets the user rate an item.
- ✓ **SemanticZoom:** Lets the user zoom between two different views supplied by two child controls. One child control supplies the zoomed-out view and the other provides the zoomed-in view.
- ✓ **SettingsFlyout:** Provides users with fast, in-context access to app settings.
- ✓ **TimePicker:** Enables the user to select a time.
- ✓ **ToggleSwitch:** Turns an item on or off.
- ✓ **Tooltip:** Displays a tooltip that can support rich content (such as images and formatted text) to show more info about an object.
- ✓ **ViewBox:** Scales a single child element to fill the available space without resizing it. The control reacts to changes in the size of the container, and to changes in size of the child element. For example, this control responds if a media query results in a change in aspect ratio.

The other attribute that is WinJS-specific is the `data-win-options` attribute. This is how you tell the control specifics about how you want it to handle its data. The properties of the control object are the same ones you could set in the JavaScript.



Deciding whether to set properties in the HTML or in the JavaScript is a judgment call. In general, if it is something I think I might have to change at runtime with my JavaScript code, I set it in the JavaScript. If it's something that is set permanently at design time, I set it in the HTML.

Item template

Above the text for the repeater is the Item template, which contains the HTML for the stuff you want repeated. Depending on how the repeater is set up, you can organize items in different ways, and use the HTML to show them just the way you want them:

```
<!-- These templates are used to display each item in the ListView declared
      below. -->
<div class="headerTemplate" data-win-control="WinJS.Binding.Template">
  <h2 class="group-subtitle" data-win-bind="textContent: subtitle"></h2>
  
  <h4 class="group-description" data-win-bind="innerHTML: description"></h4>
</div>
<div class="itemtemplate" data-win-control="WinJS.Binding.Template">
  
  <div class="item-info">
    <h4 class="item-title" data-win-bind="textContent: title"></h4>
    <h6 class="item-subtitle win-type-ellipsis" data-win-bind="textContent:
      subtitle"></h6>
    <h4 class="item-description" data-win-bind="textContent: description"></
      h4>
  </div>
</div>
```

There are actually two item templates here, one `headertemplate` and one `itemtemplate`, both of `data-win-control` type `WinJS.Binding.Template`. The `data-win-bind` attribute is the important one here. It's what sets fields in the binding datasource to other attributes of the HTML.

In the `headerTemplate`, for example, the `textContent` attribute of the H2 header is what is bound to the `subtitle` field in the datasource. In the next template, the `src` and `alt` fields of the `img` tag are bound. Any HTML attribute can be bound this way.

Styling

All of the stuff that defines how this works is in the CSS file. These can be found, surprisingly enough, in the CSS folder in the solution.



I'm not going to cover a lot of CSS in this book; there just isn't enough room, and I've gotta cover a lot of Windows 8 stuff. Most of the items that are in the WinJS library are expertly styled and you don't need to change them. Anything else can be styled in a WYSIWYG (What You See Is What You Get) fashion using Expression Blend.

Just to give a quick overview, however, the `headerTemplate` div has a `H2` tag in it that has a class of `group-subtitle`. This class can be found in the `groupDetail.css` file and looks like this:

```
.groupdetailpage .grouplist .win-groupheader .group-subtitle {  
    -ms-grid-row: 1;  
    margin: 0;  
    margin-bottom: 11px;  
    max-height: 48pt;  
    overflow: hidden;  
    padding-bottom: 4px;  
}
```

If you need to make changes, you can make them here. However, I really do recommend that you use Expression Blend to get the classes the way you like them, and then use the classes just like styles in Word.

If you want to learn more about CSS, I heartily recommend *HTML*, *XHTML*, & *CSS For Dummies*, by Ed Tittel and Jeff Noble (published by John Wiley & Sons, Inc.).

Chapter 11

Managing the Process Lifecycle

In This Chapter

- ▶ Determining when Windows Store apps run and don't run
 - ▶ Creating seamless application experiences
 - ▶ Making your app faster and more fluid
-

process lifecycle management (or PLM) is a large, complex, technical, and sometimes controversial topic that isn't the usual fodder for a *For Dummies* book. It's a bunch of stuff that you have to know to improve the user experience for your app, but it doesn't have a tremendous amount of extremely visible outcomes.

It's just something you have to do.

In short, Windows Store apps run like phone apps, not like Windows 7 or web programs. In the Windows world, apps run until the user stops them. In the web world, apps run when the user requests something and then immediately forget the user ever existed.

Windows Store apps are somewhere in between. They have status as long as they are visible, and then they might lose their status with the operating system, or they might not. They're like the Schrödinger's cat of the programming world. (You know, the act of checking on it determines its existence.)

Nonetheless, you have to always be prepared for the operating system to forget about your app. The user has the right to look at something else, and when that happens, your app has about five seconds to live. Unless you want the user to have to go through startup every time, you need to be able to handle the changes in the app's status.

As you would expect, Microsoft has provided the tools to do just that. They are complex to use, unless you already know more than you should have to know about the internals of PLM. For that reason, I am going to go a lot more into the "why" of the process lifecycle than I usually would.

Handling App Suspension

The Windows 8 experience is supposed to be fast and fluid, and to that point, users should be able to change apps at any time. That only makes sense.

The problem with that in practice is that users end up with practically every app on their device running. This doesn't bode well for battery life or keeping the device usable for any length of time.

Not to put too fine a point on it, but Microsoft fixed that problem by taking the lifecycle of the app out of the user's and the developer's hands. Now Windows decides when an app is allowed to run. Here is the list of times when an app can run:

- ✓ When the app is on the screen in full, snapped, or filled view
- ✓ When the app is playing music (sort of)
- ✓ When the app is moving a file

That's it. Other than during those times, your app probably has no resources from the operating system.

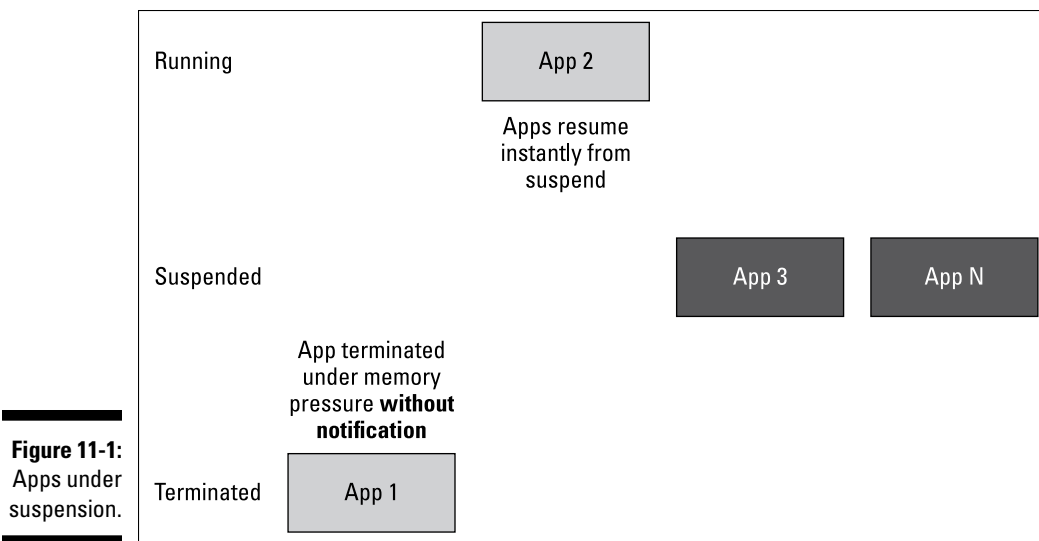
There is good news, however. Certain events tell us when the app has lost focus, when it is suspended, and when the user is resuming use of the app. What's more, they are very quick and work well. You just need to know how to use them right.

Switching tasks

The in-depth nature of handling the lifecycle stems from the user switching apps. In a touch environment, the user switches apps by swiping from the left to the right. Try it now, unless you are reading this book on a tablet, in which case you should take my word for it.

Now, when the app is on the screen, in either full, fill, or snap view, it's considered running. This is the normal state of your app, when it's doing things. This is the state that you should be generally programming for.

As soon as you swipe and bring up another app on the screen, your app effectively enters the suspended state. In Figure 11-1, App 2 was just brought into focus, whereas App 3 was just placed in suspension.



All in all, suspension isn't so bad a place to be. More or less, the apps in suspension have little or no access to the operating system resources because they are all focused on the running app. This means that inactive apps can't do work, but they also don't drain the battery or slow down the running app.

You can watch how Windows 8 handles suspension by looking at the Task Manager. To get to the Task Manager,

1. **Go to desktop view.**
2. **Right-click the Status bar.**
3. **Select Task Manager.**
4. **Navigate away from your app. In five seconds or so, it should show as suspended, and the Task Manager should look like Figure 11-2.**

The less-than-great part about suspension is that you have five seconds (give or take) to clean house before the app could potentially be terminated. *Termination* means that Windows has decided to close the app — as in goodbye, no resuming, you are starting over.

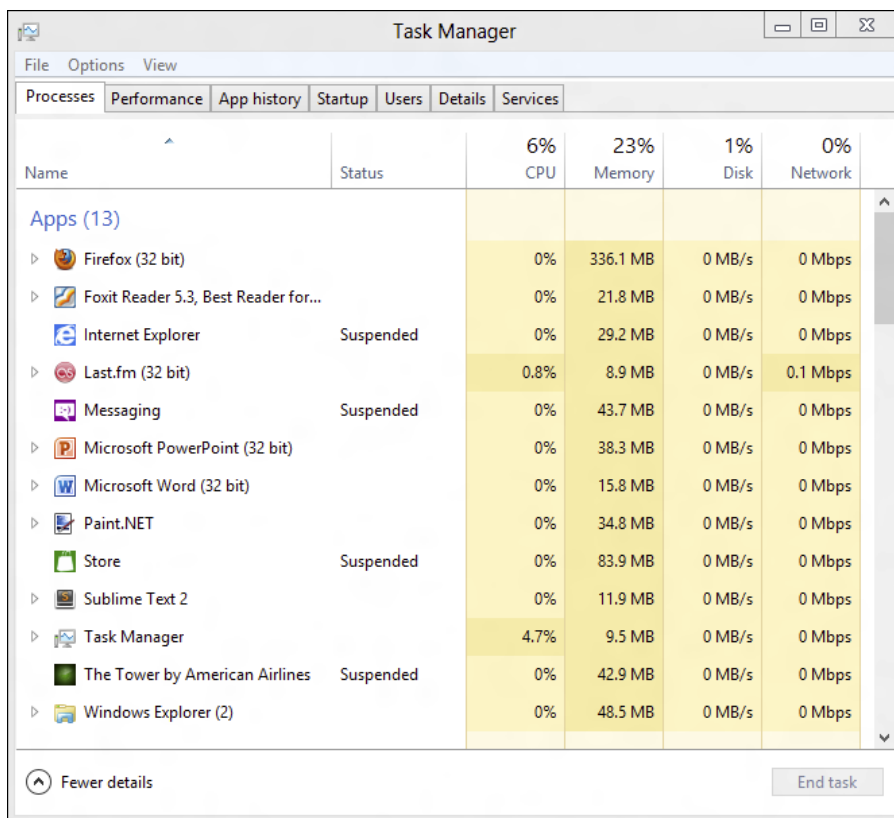


Figure 11-2:
The
Windows 8
Task
Manager.

Apps don't get notified that they are terminating. There is a suspension event, but then you had better be ready. Even when an app terminates for another reason besides suspension, you get the five-second suspension warning. Other things that can cause termination are

- ✓ The system is feeling memory pressure
- ✓ The user logs off
- ✓ The user force-closes the app
- ✓ There is a system shutdown
- ✓ There is a crash
- ✓ The operating system feels like terminating

Every time the user switches away from your app, you need to handle two things: suspending information in preparation for a resume, and cleaning up in case you are terminated. And you need to do it in five seconds. Then you need to get the app ready to come back from either state, at any time.

Dealing with suspension and termination

With the `onappactivated` and `oncheckpoint` events, Microsoft has made it really simple for us in the JavaScript world to deal with suspension and termination, as well as resume (which I talk about in the section, “Registering a resume event,” later in this chapter).

To see what I mean, open Visual Studio and create a new blank JavaScript project. Near the bottom of the screen, the comment in the stubbed-in `oncheckpoint` event says it better than I ever could:

TODO: This application is about to be suspended. Save any state that needs to persist across suspensions here. You might use the WinJS.Application.sessionState object, which is automatically saved and restored across suspension. If you need to complete an asynchronous operation before your application is suspended, call args.setPromise().

Well, that’s about it. Chapter over, right?

No, seriously: It’s true that you need to put code in there that handles suspension, and that you might want to use `sessionState` and `setPromise`. I’ll go over how to use those two tools, and what all needs to be in this event handler.

What needs to be in the oncheckpoint handler

When the `oncheckpoint` event is fired, your app has five seconds before it loses any network, display, or processor access. It means that while your app is still alive, it is in suspended animation thus the term *suspended*.

The `oncheckpoint` handler is your chance to

- ✓ Make sure the app resumes as though it had never been suspended
- ✓ Prepare to start over if the app has been terminated
- ✓ Be nice and release anything you were using exclusively

So, the kinds of things you want to save (programmatically speaking) are

- ✓ Any edited data, like a document or message
- ✓ The current page or place in the app
- ✓ The place where the user paused his music, game, or video
- ✓ Items in a cart
- ✓ Selected items, like people or files
- ✓ Changed settings or preferences
- ✓ Information that may have been recently retrieved from a service

Where do you save all of this stuff? While the app is suspended, you can just keep it in RAM. When it gets terminated, the RAM storage goes away and you will need something more persistent. You might have a service backing up your app; if you don't, you should use the `sessionState`.

Using sessionState

`sessionState` is a WinJS class that populates when the application is launched after a termination. If the device was restarted, or the app was stopped by Windows due to memory pressure, the `sessionState` can be used.

To store something in the `sessionState`, you can create a new object in JavaScript, populate it with your settings, and then drop it in `sessionState`. Here's what my `oncheckpoint` handler looks like:

```
app.oncheckpoint = function (args) {  
    var currentState = new Object();  
    currentState.page = 'videoPlayer.htm';  
    currentState.video = 'MySummerVacation.wmv';  
    currentState.secondsWatched = 385;  
    app.sessionState.stateObject = currentState;  
};
```



There are a lot of ways to store persistent data in Windows Store apps, but a SQL Server database isn't one of them. To learn more about data storage in Windows 8, check out Chapter 12.

Working with setPromise

`sessionState` is great for the persistent data, but what if you have operations that mess up your app if they're stopped, like an XHR call to your back-end service?

In order to make sure that the `oncheckpoint` won't be called until a promise returns, you can use `setPromise`. This tells Windows 8 that the suspend operation can't be completed until a promise is fulfilled or the timeout expires.

```
app.oncheckpoint = function (args) {
    var currentState = new Object();
    currentState.page = 'videoPlayer.htm';
    currentState.video = 'MySummerVacation.wmv';
    currentState.secondsWatched = 385;
    app.sessionState.stateObject = currentState;
    args.setPromise(
        WinJS.xhr(
            {url: myBackendService, data:
              myDataObject}
        )
    );
};
```

Registering a resume event

After you have everything safely stored away, you can rest assured that the user will have a great experience even if the app is terminated, right? Well, not exactly. Next you have to load all of the app's data and settings back-up.

Is it fun? No. Is it necessary? Yup.

I don't care for the recommended practice here, though. Take a look at the template code for the `app.onactivated` event:

```
app.onactivated = function (args) {
    if (args.detail.kind === activation.
        ActivationKind.launch) {
        if (args.detail.previousExecutionState
            !== activation.ApplicationExecutionState.
            terminated) {
            // TODO: This application has been newly
            // launched. Initialize
            // your application here.
        } else {
            // TODO: This application has been
            // reactivated from suspension.
            // Restore application state here.
        }
        args.setPromise(WinJS.UI.processAll());
    }
};
```

If the app was launched, you see whether the previous state was *not* terminated. If it wasn't terminated, you are supposed to totally initialize the app. Then you should only resume state if the app was terminated.

I think that's a little silly. In my apps, I check to see whether the app was suspended, and then refresh any stale data I might have. Otherwise, I refresh from `sessionState` if it's populated. That way, it doesn't matter how the app was stopped. I always get my state data if it's available.

```
app.onactivated = function (args) {
  if (args.detail.kind === activation.
      ActivationKind.launch) {
    if (args.detail.previousExecutionState ===
        activation.ApplicationExecutionState.suspended)
    {
      // refresh any data from services or
      machine status or whatnot
    } else {
      if (args.setPromise) {
        var currentState = new Object();
        currentState = args.setPromise.
            stateObject;
        //repopulate from state
      } else {
        //First time run
      };
    }
    args.setPromise(WinJS.UI.processAll());
  }
};
```

Making Your App Look Alive

Your app activates and vanishes on the whim of Windows now. This fact, by necessity, changes the way you plan for user interaction. Fortunately, there are some guidelines to help you along.

The most important thing is to make sure the user feels like the application is still running when they switch to it. Most of that is saving appropriate data at suspension and bringing the right things back at resume. Some other tricks help, too — like running certain tasks when the app isn't running, and updating the tiles.

In short, the goal is to make Windows 8 look more like . . . Windows 7. Make sure when the user presses Alt+Tab, everything works as expected. The app leaves the user where they expect. The data doesn't vanish. It's all good.

Dealing with application launch

In the section, “Registering a resume event,” I discuss resuming the application from suspension, but you actually have multiple options to consider. Two properties that come back from the arguments of the `onactivated` event handler will help you out: `kind` and `previousExecutionState`.

The `kind` argument tells you what launched your app. Windows 8 offers you more than one way to get launched, and your app might have to respond differently based on how it was launched. There are, in fact, 12 values in the `ActivationKind` enumerator, and you can handle each separately if you wish (although you probably don't have to):

- ✓ **launch:** Sort of the default case. This is when the user launches an app normally.
- ✓ **search:** Activated through a search from another app.
- ✓ **shareTarget:** Activated through a share from another app.
- ✓ **file:** Another app launched a file that this app is registered to handle.
- ✓ **protocol:** Another app launched a URI (like a web address) that your app is registered to handle.
- ✓ **fileOpenPicker:** The user picks a file for which this app is registered.
- ✓ **fileSavePicker:** The user saved a file to your application.
- ✓ **cachedFileUpdater:** Used when your app provides content management for a file updated by the user.
- ✓ **contactPicker:** Activated by the user picking a contact.
- ✓ **device:** Used if your app is registered for autoplay.
- ✓ **printTaskSettings:** Used if your app is registered as a print target.
- ✓ **cameraSettings:** Used if your app is registered to capture pictures from a camera, and the camera is activated.

Looking at the template code in the section, “Registering a resume event,” you see that Microsoft suggests checking the `kind` right away, and I concur. Don't forget to put a catchall in there in case the app gets loaded by something you weren't expecting!

The `previousExecutionState` argument tells us what the last thing the application was doing before it was activated. This is the second thing to check after activation, just after the `ActivationKind`.

The app could have been doing any of five different things before activation, and each of them should elicit a different response from your app.

- ✓ **NotRunning:** Activation after first install or a reboot. Start the app up from the beginning.
- ✓ **Running:** A search contract or something like it activated the app when it was already running. Just handle the activation event.
- ✓ **Suspended:** A contract has activated the app when it was suspended. Again, just handle the activation event. Everything is still stored in RAM.
- ✓ **Terminated:** Activation after Windows has terminated the app. This is when you use the saved session data.
- ✓ **ClosedByUser:** User purposefully closed the app. Just restart like `NotRunning`.

The most important of these is `Terminated` because that is the state that you have to recover from gracefully every time. Remember, the user doesn't know that it might have been closed!

Doing some things after suspension

By default, an app is allowed to do a few things after it's suspended. They are tasks that are grandfathered in by Windows. They include playing audio and moving files.

Playing audio

Playing a podcast and then switching to Visual Studio to get some work done is one of those things that programmers just do. Playing background audio is possible in even the least-forgiving mobile platforms.

Windows 8 is no different. As long as you declare the ability and make a few adjustments to the code base, Windows lets your app stay alive and play audio even when it should be suspended.



Is the app actually suspended when those five seconds expire? My tests say yes. The audio thread continues, but other functions of the operating system are not available. You can't call services, for instance. Does Internet radio work? I'll leave testing that as an exercise for the reader. The documentation states "If you have an app that performs other tasks in addition to streaming

audio or video, then when the app loses focus and is no longer the active window, your app should stop doing non-media related work.” But in my honest opinion, that isn’t very clear.

To get background audio working, take the following steps:

1. **Update the package manifest.** In Figure 11-3, you can see the Audio check box that needs to be selected in the package.appxmanifest. Just select the Background Tasks item from the Available Declarations drop-down list, and then select the Audio check box.

Either the Entry Point or Start Page field has to be populated.

You can do the same thing for video. WinRT doesn’t have a video item, but if you set the Audio type, video works too. It doesn’t recover well when you switch back to the app (at least in Release Preview), but it does work.

2. **Use the HTML5 audio tag in the HTML file to make the audio happen.** Add a `msAudioCategory` tag to the audio tag and set it to `BackgroundCapableMedia`.

```
<audio msAudioCategory="BackgroundCapableMedia">
  <source src="/audio/sample.mp3"/>
</audio>
```

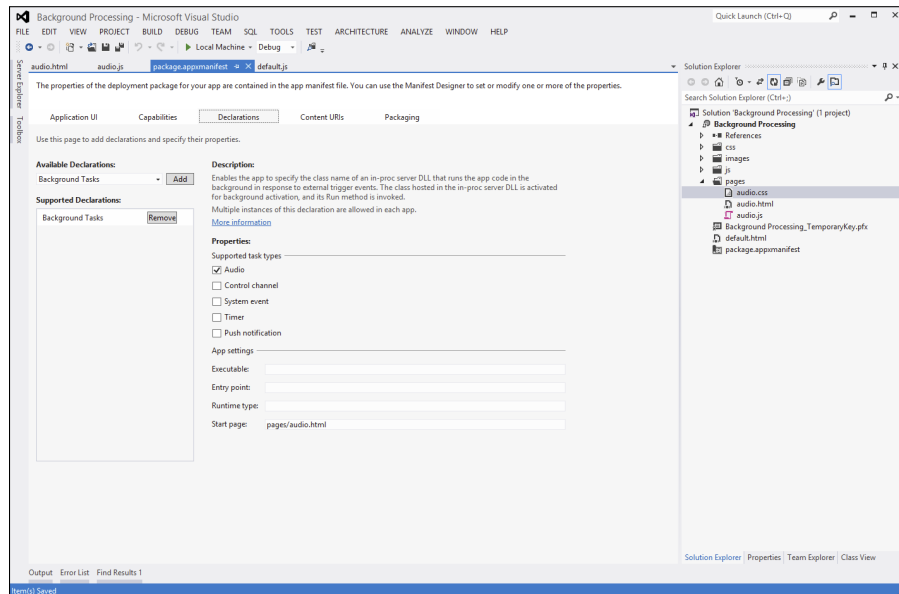


Figure 11-3:
Declaring
audio in the
manifest.

3. In the JavaScript file, register the media transport controls. Fortunately, this is easier than it sounds. Create an instance of `Windows.Media.MediaControl` in the ready event first.

```
ready: function (element, options) {
    var media = Windows.Media.MediaControl;
    media.addEventListener('playpausetogglepressed', playswitch, false);
},
After that you just have to add that playswitch
function
function playswitch() {
    if(mediaControls.isPlaying === true) {
        document.getElementById('audiotag').
        pause();
    } else {
        document.getElementById('audiotag').
        play();
    }
}
```

That's all there is to it. Windows 8 and WinJS handle everything else.

Moving files

Something else that might need to continue when you Alt+Tab away from an app is a file download. It's handled in a similar fashion to audio. When you use the `BackgroundTransfer` collection of classes, you effectively give the responsibility of transferring your file over to Windows. Even if your app is suspended, the file transfer continues.

To wire together a background transfer, you need two principal activities: a new file and a `BackgroundDownloader`. You can make a new file with the `createFileAsync` method in the `Windows.Storage` namespace, and then set up the background downloader in the callback. It looks like this:

```
try {
    Windows.Storage.KnownFolders.musicLibrary.
    createFileAsync(
        'sample.mp3',
    Windows.Storage.CreationCollisionOption.
    generateUniqueName
    ).done(function (file) {
        var uri =
    Windows.Foundation.Uri('http://ia600609.us.archive.
    org/16/items/tpalmieri2012-06-30.akg481-sbd.flac16/
    tpalmieri2012-06-30t15.mp3');
        var downloader = new
```

```
Windows.Networking.BackgroundTransfer.Background
Downloader();

        // Create a new download operation.
        download =
download.createDownload(uri, file);

        // Start the download and persist the promise
to be able to cancel the download.
        promise = download.startAsync();
    });
} catch (err) {
    console.log(err);
}
```

I just grabbed a file from the Internet Archive for the sample, Of course, any file will do.

The trick here isn't that the download continues if your app is terminated, but instead

- ✓ The download is continued after the app is suspended.
- ✓ If the app is terminated, the operating system saves where it stopped and remembers where it was.

If you have background downloads in your app, you need to check the `getCurrentDownloadsAsync` collection when your app resumes. To do this, add check code to the `resume` event you set up in the section, "Registering a resume event":

```
var downloader = Windows.Networking.
BackgroundTransfer.BackgroundDownloader
downloader.getCurrentDownloadsAsync().done(
function (downloads) {
    for (var i = 0; i < downloads.size;
i++) {
        //Process the downloads as
appropriate for your app
    }
});
```

Running background tasks

Besides running prescribed tasks in the background, you can also make your own. It isn't particularly straightforward, but it also isn't something that you need to do every day:

1. Register your application as a background task provider with the application manifest.

Figure 11-4 shows where that is in the Declarations — it's a lot like registering for audio.

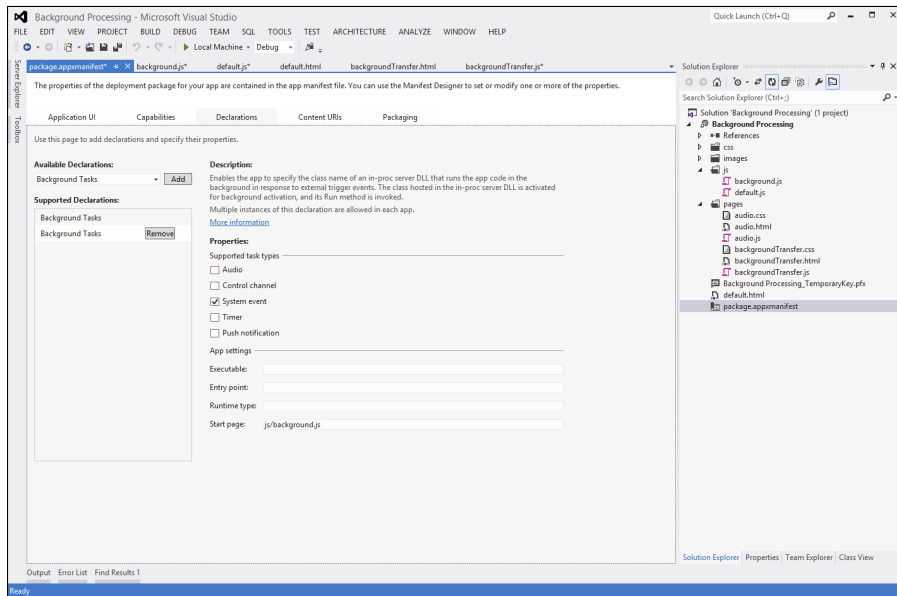


Figure 11-4:
Declaring a
background
task.

2. Build the background task. Just make a new JavaScript file in your project and wrap your JavaScript code in the background task handler.

Microsoft provides some template code for this that works well. Here is an example for a task that computes primes:

```
(function () {
    "use strict";
    var backgroundTaskInstance = Windows.UI.WebUI.
        WebUIBackgroundTaskInstance.current, cancel =
        false;

    function onCancel(sender, cancelReason) {
        cancel = true;
    }

    backgroundTaskInstance.addEventListener("canceled",
        onCancel);

    function doWork() {
```

```

        var key = null, settings =
Windows.Storage.ApplicationData.current.
localSettings, primeHolder = "";
        //Here is where our code goes that does the
actual work
for (number = 1; number <= 1000; number++) {
    primeFlag = true;
    maxTest = number / 2;
    if ((number != 2) && ((number % 2) == 0)) {
        primeFlag = false;
        Test = 3;
        while ((Test <= maxTest) && (primeFlag
== true)) {
            if ((number % Test) == 0) {
                primeFlag = false;
            }
            Test = Test + 2;
        }
        if (primeFlag == true) {
            primeHolder = primeListHolder + number + " ";
        }
    }
    //Now we just do a little housekeeping
    key =
backgroundTaskInstance.task.taskId.toString();
    settings.values[key] = "Succeeded";
    //Always call close when done
    close();
}

if (!cancel) {
    doWork();
} else {
    // Let the app know we are cancelled.
    key =
backgroundTaskInstance.task.taskId.toString();
    settings.values[key] = "Canceled";
    //Always call close when done
    close();
}

})();

```

First prep for a Cancel event listener, and then set up the `doWork` function that actually has the working code in it. Background tasks always have to call `close` when done.

3. Finally, register your background task by adding some code in the `default.js` page. The `Windows.ApplicationModel.Background.BackgroundTaskBuilder` has a property called `taskEntryPoint` that expects an object name, but if you pass it a JavaScript file name, it works, fortunately.

```
var builder = new Windows.ApplicationModel.Background.  
    BackgroundTaskBuilder();  
  
var trigger = new Windows.ApplicationModel.Background.  
    SystemTrigger(Windows.ApplicationModel.  
        Background.SystemTriggerType.timeZoneChange,  
        false);  
builder.name = "Busily calculating primes.";  
builder.taskEntryPoint = "js\\background.js";  
builder.setTrigger(trigger);  
builder.addCondition(new Windows.ApplicationModel.  
    Background.SystemCondition(Windows.  
        ApplicationModel.Background.  
            SystemConditionType.userPresent));  
var task = builder.register();
```

At the base level, that's all there is to it. Well-behaved background tasks gracefully handle the `completed` event, but there are a remarkable number of options there. I recommend that you search dev.windows.com for the words *Create and register a background task* to find the Quickstart guide, and check out Step 3 of the step list for all of the things you can do with `completed`.

Using live tiles

Your app tile is something else you can use to make your app look alive even when it isn't running. You can register a cloud service (on Azure or elsewhere) to update the tile's information when Windows has a free second, rather than just when the app runs.

Chapter 8 has a complete breakdown on making your tile more alive. It's an important part of the whole Windows 8 experience.

Chapter 12

Keeping Local Storage

In This Chapter

- ▶ Storing settings
 - ▶ Making use of SQLite
 - ▶ Storing things in the file system
-

One of the most interesting things about the Windows 8 platform, as far as systems architecture goes, is that you can't do what Microsoft products have been best at for years — connect to a SQL Server.

You heard that right. Because the socket-level connection capability has been seriously constrained in Windows Store apps, you can't connect to an external database. That means that the standard forms-over-data approach to programming that was fostered in a large way by Visual Basic back in the '80s is . . . well . . . dead.

What's replacing it? All data should have a service over it. Most of the bad programs out there are bad because the application developer had direct access to a database. It's just too easy to make bad choices when that is possible.

And now, it isn't possible anymore, at least for Windows Store apps. If you want to build an app that uses real data — like an enterprise reporting app, or something like that — you have to get the data from some service layer.

That said, three significant sources of data are permissible and shouldn't be ignored:

- ✓ Local databases, like simple text files and deployable relational databases
- ✓ Settings
- ✓ The file system

These solutions don't lend themselves to large datasets, so please don't try. They do, however, lend themselves to storing information locally quite well. Remote data sources still need a service layer, but some things are better local.

Putting Settings Where You Can Get Them

In the average application, a surprising amount of data that relates directly to the application is used for storing user settings. Customization, session information, choices the user made, and other user-focused data is often tucked away in a table in the relational data store, where it probably shouldn't be anyway. (It really should be in some local settings format.)

In the Windows Store app space, there are better choices than the old .ini file for storing settings. There is a whole infrastructure for it in fact, and even a facility to share settings between machines.

Organizing local settings

The `Windows.Storage` namespace has a few tools to help manage settings in Windows 8. Chiefly among these is the `localSettings` class, which is a simple holding object that has certain stateful characteristics that are taken care of for you by Windows.

Using `localSettings` poorly, though, is pretty easy. At its most basic, it is just data in, data out.

```
var settings = Windows.Storage.ApplicationData.  
    current.localSettings;  
//write a setting  
settings.values["theAnswer"] = 42;  
//read a setting  
var thatSetting = settings.values["theAnswer"];
```

To remove a setting, just use the `remove` function of the `localSettings.values` collection, like this:

```
settings.values.remove("TheAnswer");
```

This kind of thing will work. In a *very* simple app, you might just need a few settings, and just storing them as one-offs and getting them back out when you need them might be enough.

In my opinion, however, a structured approach works much better. You might not need it right now in the development of your app, but you will be glad you chose it later. As an app grows, the structure should support the larger files, namespaces, and assets.

This pattern is loosely based on Microsoft's pattern for sample apps and Adam Kinney's stylings as well:

1. **Create a new file called `settings.js` in your `js` folder.**
2. **Get a reference to `Windows.Storage.ApplicationData.current.localSettings`.**
3. **Add a variable for your setting.**
4. **Write a getter and a setter for the setting that uses the `localSettings` feature.**
5. **Add the getting and settings to your namespace structure.**



Don't forget to declare the namespace in the `default.js` file:

```
WinJS.Namespace.define("MyApp");
```

The finished product looks something like this:

```
(function () {
    'use strict';

    var settings = Windows.Storage.ApplicationData.
        current.localSettings;
    var theAnswer;

    var getTheAnswer = function () {
        if (!theAnswer) {
            theAnswer = settings.values['theAnswer'];
        }
        return theAnswer;
    }

    var setTheAnswer = function (newValue) {
        if (theAnswer !== newValue) {
            theAnswer = newValue;
            settings.values['theAnswer'] = theAnswer;
        }
    }

    WinJS.Namespace.defineWithParent(MyApp, 'Settings', {
        getTheAnswer: getTheAnswer,
        setTheAnswer: setTheAnswer
    });
});
```

This has the awesome benefit of organizing your settings into a structured object. To get a setting, you just call the function in another file:

```
var theSetting = MyApp.Settings.getTheAnswer();
```

It takes all of the complexity out of using the settings system for you and your other developers. It's not that hard to code for, but it's easy to accidentally mess up a stored setting if you are directly touching the `localSettings` object every time.

Shh! Roaming settings, too

This is supposed to be about local storage, but I'm going to cheat and talk about some roaming settings. Don't tell my editor.

You see, you can use the `ApplicationData` class to do basically the same thing as you did with `localSettings`, except use `roamingSettings`, and the settings follow the user from machine to machine.

You read that right: You will attach the setting as a variable to the user's Windows Live login, and it moves with them to any machine that they are logged into and that has the same app installed on it.

It looks just the same. Look at the first simple example that I did:

```
var settings = Windows.Storage.ApplicationData.  
    current.localSettings;  
//write a setting  
settings.values["theAnswer"] = 42;  
//read a setting  
var thatSetting = settings.values["theAnswer"];
```

This data is stored securely, but I still advise you to not store any truly private information in there. Users expect to know where their data is going.

Keeping more stuff with ApplicationDataContainer

Although you can store things in individual values within `localSettings`, it might be better to organize them into buckets, especially if you have a lot of them. The `localSettings` class allows for the creation of `ApplicationDataContainers` for just such instances.

Why would you want to do this? For instance, the user might have to store a bunch of temporary settings for one session of application use. With the `ApplicationDataContainer`, it's easy to keep them all in one place, and then delete the whole thing when you are done.

Implementing something like this looks a lot like the `Settings` above, except I need to do something to initialize the container. Then I can use it pretty much like the `localSettings` example:

1. Add another JavaScript file to the project.

I called mine `tempSettings.js`. Of course, you don't have to just use this for only temporary settings; I'm just following the example I used previously.

2. Create an `init` function that does two things:

1. Sets up the container using `createContainer` and the `ApplicationDataCreateComposition` value so that the function always replaces the collection.
2. Initialize all of the values.

3. Add a function to delete the container.

4. Add all of the setters and getters, just like the `settings.js` file.

5. Remember to add the file to your namespace, and add all of the public Getters and Setters as well as the `init` and `deleteContainer` function.

The final product looks like this:

```
(function () {
    'use strict';

    var settings = Windows.Storage.ApplicationData.
        current.localSettings;
    var isUp, isDown, isBackward;
    var directionsContainer;

    function init() {
        //Initialize the container
        directionsContainer = localSettings.
            createContainer("directions", Windows.Storage.
                ApplicationDataCreateDisposition.Always);
        directionsContainer.values['isUp'] = false;
        directionsContainer.values['isDown'] = false;
        directionsContainer.values['isBackward'] = false;
    }
    function deleteContainer(){
```



```
        settings.deleteContainer("directions");
    }
    var getIsUp = function () {
        if (!isUp) {
            if (settings.containers.hasKey("directions"))
            {
                isUp = directionsContainer.values['isUp'];
            }
        }
        return isUp;
    }
    var setIsUp = function (newValue) {
        if (isUp != newValue) {
            isUp = newValue;
            if (settings.containers.hasKey("directions"))
            {
                directionsContainer.values['isUp'] = isUp;
            }
        }
    }
    var getIsDown = function () {
        if (!isDown) {
            if (settings.containers.hasKey("directions"))
            {
                isDown = directionsContainer.
                values['isDown'];
            }
        }
        return isUp;
    }
    var setIsDown = function (newValue) {
        if (isDown != newValue) {
            isDown = newValue;
            if (settings.containers.hasKey("directions"))
            {
                directionsContainer.values['isDown'] =
                isDown;
            }
        }
    }
    var getIsBackward = function () {
        if (!isBackward) {
            if (settings.containers.hasKey("directions"))
            {
                isBackward = directionsContainer.
                values['isBackward'];
            }
        }
        return isUp;
    }
}
```

```
var setIsBackward = function (newValue) {
    if (isBackward !== newValue) {
        isBackward = newValue;
        if (settings.containers.hasKey("directions"))
        {
            directionsContainer.values['isBackward'] =
            isBackward;
        }
    }
}

WinJS.Namespace.defineWithParent(MyApp,
    'TempSettings', {
        init: init,
        deleteContainer: deleteContainer,
        getIsUp: getIsUp,
        setIsUp: setIsUp,
        getIsDown: getIsDown,
        setIsDown: setIsDown,
        getIsBackward: getIsBackward,
        setIsBackward: setIsBackward
    });
});
```

Here I've tried to catch all of the possibilities inside the `tempSettings` class. This makes it so the developer doesn't have to check for the existence of the container before using the setting.

Speaking of using the settings, you'll do just that in the section "Giving the User Access with the Settings charm," later in this chapter.

Keeping settings in files

It, of course, is totally possible and permissible to keep a JavaScript array of data with settings in the file system, and just read and write that. Everything you need to know about using the file system in WinRT is in the section later in this chapter titled "Filing Things Away for Later."

Giving the user access with the Settings charm

Having a neat settings class without having the user be able to edit the values would be kinda useless. Fortunately, WinJS allows for that with the `WinJS`.

UI.SettingsFlyout. This class implements the contracts in the Settings charm for you:

1. Add a pages folder to your project.

You probably already have this if it's a larger project.

2. Add a settings page control. Right-click the new page and select New ➞ Add New Item, and then select PageControl from the items dialog box that appears.

I called mine **Settings**.

3. Put three toggles on the home page for the three settings. To do this, you use the SettingsFlyout WinJS control. It looks like this:

```
<body>
  <div id="settings" data-win-control="WinJS.
    UI.SettingsFlyout" aria-label="Settings
    flyout" data-win-options="{width: 'wide'}">
    <div class="win-header">
      <button type="button" onclick="WinJS.
        UI.SettingsFlyout.show()" class="win-
        backbutton">
      </button>
      <div class="win-label">Settings</div>
    </div>
    <div class="win-content">
      <div id="settingsDiv">
        <div id="isUpToggle"
          data-win-control="WinJS.UI.ToggleSwitch" data-
          win-options="{title: 'Is Up', labelOn: 'Yup',
            labelOff: 'Nope'}"></div>
        <div id="isDownToggle"
          data-win-control="WinJS.UI.ToggleSwitch" data-
          win-options="{title: 'Is Down', labelOn: 'Yup',
            labelOff: 'Nope'}"></div>
        <div id="isBackwardToggle" data-
          win-control="WinJS.UI.ToggleSwitch"
          data-win-options="{title: 'Is Backward',
            labelOn: 'Yup', labelOff: 'Nope'}"></div>
      </div>
    </div>
  </div>
</body>
```

4. Change the ready event handler in the JS file to populate the toggles and set the event handlers.

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

    var isUpToggle = document.
    getElementById('isUpToggle');
    isUpToggle.checked = MyApp.TempSettings.
    getIsUp();
    isUpToggle.addEventListener('onClick',
    function () {
        MyApp.TempSettings.setIsUp(isUpToggle.
        checked);
    });
    var isDownToggle = document.
    getElementById('isDownToggle');
    isDownToggle.checked = MyApp.TempSettings.
    getIsDown();
    isDownToggle.addEventListener('onClick',
    function () {
        MyApp.TempSettings.
        setIsDown(isDownToggle.checked);
    });
    var isBackwardToggle = document.getElementById(
    'isBackwardToggle');
    isBackwardToggle.checked = MyApp.
    TempSettings.getIsToggle();
    isBackwardToggle.
    addEventListener('onClick', function () {
        MyApp.TempSettings.
        setIsBackward(isBackwardToggle.checked);
    });
},
```

5. In `default.js`, add the `onsettings` event handler. This wires the page to the charm and sets up the defaults.

```
WinJS.Application.onsettings = function (e) {
    e.detail.applicationcommands = {
        "settings": { title: "Settings", href:
        "/pages/settings.html" },
    };
    WinJS.UI.SettingsFlyout.
    populateSettings(e);
};
```

When you have that working, run the app and press Windows+C to bring up the Charms bar, and then click Settings. The Settings bar appears, where you can click the Settings link, which you can name anything you want in the `default.js` file. Tap the link, and you'll see Figure 12-1.

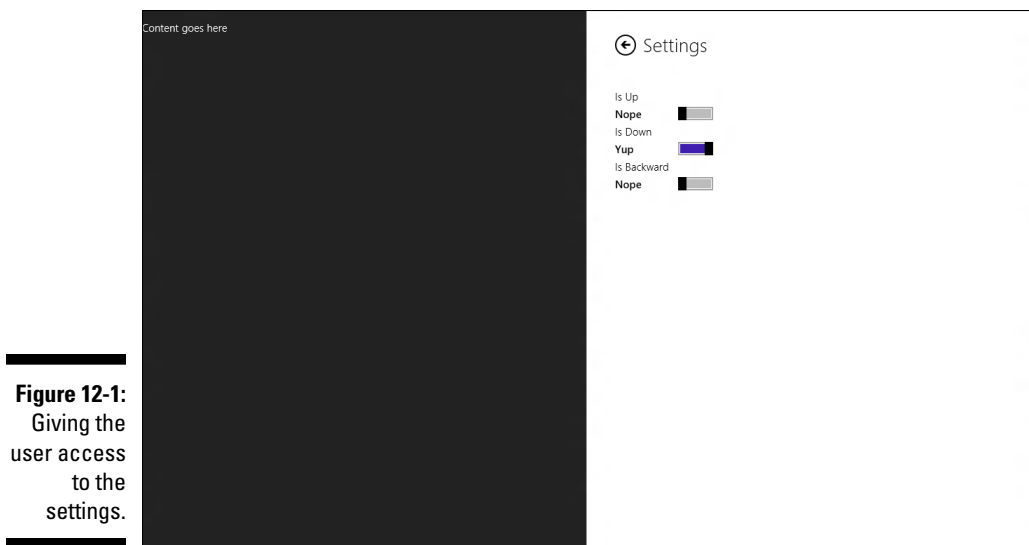


Figure 12-1:
Giving the
user access
to the
settings.

You can add as many flyouts (the Windows 8 version of a dialog box) as you like here — just wire them up with the `applicationcommands` object in WinJS. Remember, using the Charm bar isn't optional in the Windows Store. The testing process checks!

Filing Things Away for Later

Sometimes you need to store more things besides settings. Sometimes the user is creating content, and you need to save that content off on their local machine, or even on the network.

That's when the File System classes come in. WinRT isn't really a file system-driven API; it's more driven by Port 80 and 443 communication and calling web services. Nonetheless, there is still file access, and you have the ability to store user-created data as a file.

Using the file system

Take, for instance, POINTtodo, a task list that should be available at the Windows Store by the time you read this. It needs to keep a list of active tasks somewhere. I could use the Settings or build a remote service and store it on Azure. For something like a task list, however, keeping the data in a file would be the most sensible thing.

To manage data internally using a file, you need a few things:



- ✓ **Some data:** A JavaScript array would be a good choice. It can be serialized using the JSON namespace and stored as text.
- ✓ **A way to save that serialized array to a file:** This code goes into the `oncheckpoint` event handler.
- ✓ **A way to get the file and populate the array:** This goes in the `onactivated` event handler.

Now, there are all kind of lifecycle considerations here, but I cover all of that in Chapter 11. For now, keep the timeline simple and just focus on using the file.

Setting up some data

You can start the user with an empty array, of course, in any system, and have them populate from there. Doing this, you would just start the system up with an empty set.

```
var todayArray = [];
```

It makes sense in some applications, but I believe that an app like POINTtodo should have some starting data. To that end, I created a default starting set of data with some funny sample tasks in it.

```
var todayArray = [
{
  task: "Batten down the hatches with much love and
    affection",
  complete: false,
},
{
  task: "Remember the Alamo",
  complete: false,
},
{
  task: "Drink beer",
  complete: false,
}
];
```

This is good enough to start the user off, and of course he can mark these as complete and add their own data when they use the app.

The JSON format is good for this kind of application because of the JSON namespace, which has simple `parse()` and `stringify()` functions hanging off of it that serialize and deserialize the data for us without a lot of mucking around. XML works too, but I like JSON better.

Writing to a file

Now you have some content. In order to write it to a file, you need to take a few steps:



1. **Declare the Document Library capability in the package.appxmanifest file. Do this by opening the file in Visual Studio and clicking the Capabilities tab, and then selecting the Document File Access check box.**

Notice that the Capabilities tab now has an error notification in the form of a red circle with an X in it. If you mouse over it, you can see that it's instructing us to associate a file type to the app.

2. **Change to the Declarations tab, select the File Type Associations, and click Add.**

You'll get another error notification in the body of the form, and the same mouseover trick works.

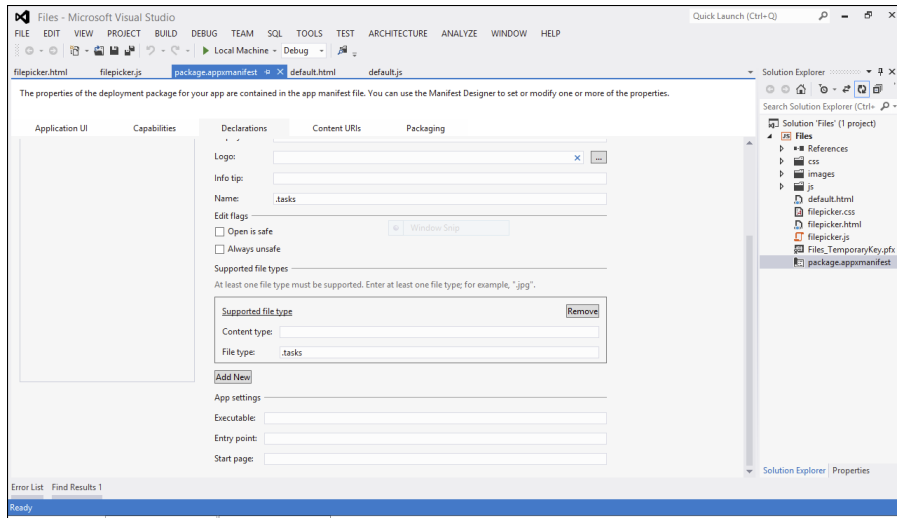


3. **Fill out the declaration form as shown in Figure 12-2. Just add the extension you will be using into the Name and File Type fields — that's enough for what you're doing here.**

A list of extensions is reserved, so you can't use them. They include

- | | |
|---------------------|-------|
| • Accountpicture-ms | • lnk |
| • Appx | • Msi |
| • application | • Msp |
| • Appref-ms | • ocx |
| • Bat | • pif |
| • Cer | • Ps1 |
| • Chm | • Reg |
| • Cmd | • Scf |
| • Com | • Scr |
| • Cpl | • Shb |
| • crt | • Shs |
| • dll | • Sys |
| • drv | • ttf |
| • Exe | • url |
| • fon | • Vbe |
| • gadget | • Vbs |
| • Hlp | • Ws |
| • Hta | • Wsc |
| • Inf | • Wsf |
| • Ins | • Wsh |
| • jse | |

Figure 12-2:
Setting up
the file
declaration.



4. In `default.js` header, declare a variable to hold the file.

```
var tasksFile;
```

5. In the `oncheckpoint` event, save the value of the array to the file. To do this, you need to either make sure the file is there or create it. (I just create it new each time.) Now populate it with `writeTextAsync`:

```
app.oncheckpoint = function (args) {
    Windows.Storage.KnownFolders.documentsLibrary.
        createFileAsync("POINTtodo.tasks",
            Windows.Storage.CreationCollisionOption.
                replaceExisting).done(function (file) {
                tasksFile = file;
            });
    Windows.Storage.FileIO.
        writeTextAsync(tasksFile, JSON.
            stringify(todayArray)).done();
};
```

Reading the file

Now that the file is on disk (or will be when your app is suspended) you need to get it back by reading it when the app is activated. In the `onappactivate` event, read the file from disk and populate the array with `readTextAsync`.


```
app.onactivated = function (args) {
    if (args.detail.kind === activation.
        ActivationKind.launch) {
        if (args.detail.previousExecutionState
            !== activation.ApplicationExecutionState.
            terminated) {
            Windows.Storage.FileIO.
            readTextAsync(sampleFile).then(function
            (contents) {
                todayArray = JSON.parse(contents)
            });
        }
        args.setPromise(WinJS.UI.processAll());
    }
};
```

The `Windows.Storage` APIs make this pretty simple. All you have to remember is that file operations are all asynchronous and you have to handle the return with `then` or `done` when you use them.

The File Picker contract

Sometimes you want to give the user a chance to save and load files of their own creation. In WinRT, you do this with the `Windows.Storage.Pickers` class. It works a lot like the `CommonDialog` of .NET fame, but it's all fancy for the new look.

To let the user pick files to open, create a new page control in your project. (I called mine `filepicker`.) Try these steps:

1. Add a button to the `filepicker.html` file that opens the picker.

```
<body>
  <div class="filepicker fragment">
    <header aria-label="Header content" role="banner">
      <button class="win-backbutton" aria-
        label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to
        filepicker</span>
      </h1>
    </header>
    <section aria-label="Main content" role="main">
      <button class="action" id="pickfile">Select a
        file</button>
    </section>
  </div>
</body>
```

2. In the ready event handler, add a click event handler for the button:

```
ready: function (element, options) {  
    document.getElementById("pickfile").  
        addEventListener("click", selectTaskFile,  
            false);  
},
```

3. Write a function that uses the FileOpenPicker to select files.

```
function selectTaskFile() {  
    var openPicker = new Windows.Storage.Pickers.  
        FileOpenPicker();  
    openPicker.viewMode = Windows.Storage.Pickers.  
        PickerViewMode.thumbnail;  
    openPicker.suggestedStartLocation =  
        Windows.Storage.Pickers.PickerLocationId.  
        documentsLibrary;  
    openPicker.fileTypeFilter.replaceAll([".  
        tasks"]);  
    openPicker.pickSingleFileAsync().then(function  
        (file) {  
        //Do something  
        });  
}
```

Change the default start page to filepicker.html in the package.appxmanifest file, and then press F5 to run the app. When you click the button, you see the new File Picker as shown in Figure 12-3.

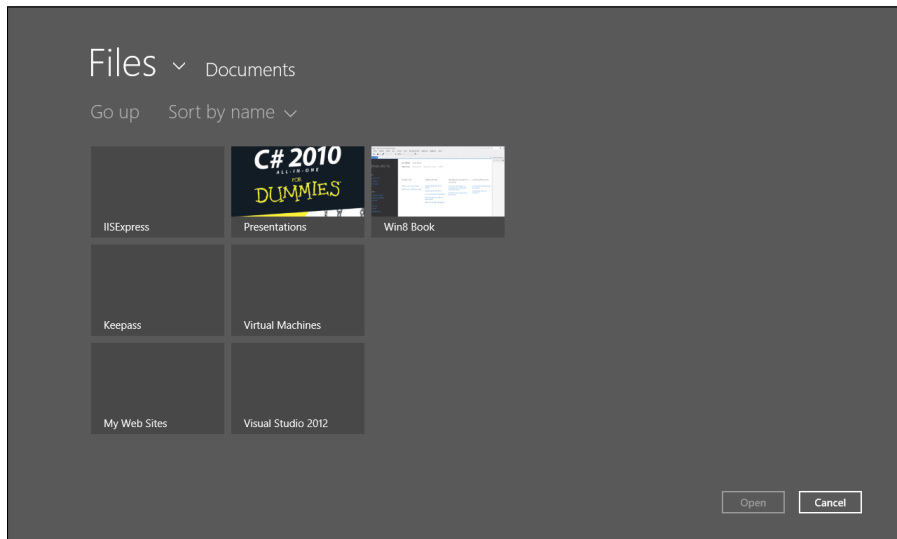


Figure 12-3:
Selecting a
file with the
File Picker.

You can do a lot more with the File Picker for both Open and Save, but I refer you to <http://msdn.microsoft.com/en-US/windows> for that information. Just search for **Windows.Storage**.

Taking Advantage of Deployable Databases

At times, neither settings nor files suit the bill. Sometimes, you need a local, relational database. Unfortunately, there isn't a built-in tool for that in Windows 8. Using local data isn't its thing.

There are ways around this, however. Textual databases stored as JSON or XML can be saved as files, and you might be able to include an open source database or two in your deployment. (Not sure, you know — just sayin'.)

Providing some relational power

It may be that you need to deploy a larger, more structured relational database with your application, like you used to do with Microsoft Access and the JET provider. There is no built-in way to do that with Windows 8 — that just isn't the goal. Microsoft wants us to use services, but that isn't always optimal.

The other option is the open-source SQLite database. Well, there are other options, but SQLite is the best one. It offers the best set of features and has good community support.

Getting SQLite

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine, according to the documentation. It is basically a Windows DLL (dynamic linking library), written in C++, that you can include directly in your application. It takes care of the implementation details of the database file for you.

To get SQLite, head over to their download page at www.sqlite.org/download.html and get the tool for your version of Windows, 32- or 64-bit. Then register and include it with your application.

SQLite isn't your normal database, however. There is no management, no GUI. It doesn't run as a service. Everything is done in code. Everything from creation of a new database file to deleting rows is done with the code in your application.

Wrapping SQLite for WinRT

SQLite is a COM DLL; it isn't designed for the Windows Store style WinMD format. For that reason, getting the binary and including it won't work. Jan Nedoma and a few other data community members have put together a WinRT component that wraps the current SQLite component.

As of this writing, however, it hasn't been updated for the final release version of Windows 8. For that reason, you have to get the source code from Bitbucket and compile it yourself. Don't worry, it isn't a big deal:

1. **Go to** <https://bitbucket.org/MnemonicWME/sqlitemetro/wiki/Home> **and download the source code for SQLiteMetro.**
2. **Decompress the project and open the solution file in Visual Studio.**
3. **Right-click the project and select Properties from the menu that appears.**
4. **Select Configuration and change the platform to x64, as shown in Figure 12-4.**

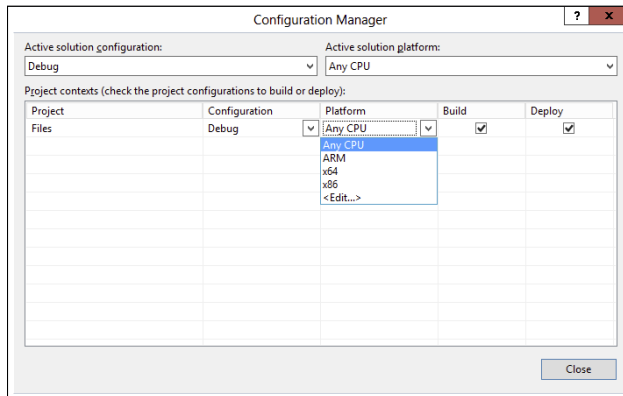


Figure 12-4:
Setting the
platform in
SQLiteMetro.

5. **In Project Properties, change the Project Output to WinMD.**
6. **Add the project to your solution and reference it from there.**

You're good to go from there. At this point, I'll refer you to the codeplex site at <http://sqlwinrt.codeplex.com> for samples and getting started. It should get you started in building local databases.



Here's some late-breaking information: Check the SQLite repository at sqlite.org for a new Windows 8 version. Microsoft apparently is working with them, which might be why SQLiteMetro isn't as actively developed as it could be.

Using the HTML5 database options

Window Store apps support the Indexed Database API, which is a W3C standard for web browsers that largely replaced Web SQL. If you enjoy reading standards documentation, you can check it out at www.w3.org/TR/IndexedDB/, but don't hurt yourself rushing over there to do it.

The IndexedDb is very raw, in my opinion, and you should consider very closely why you need it rather than one of the other options in this chapter. It is a very low-level feature, with at-the-metal level access to database structures. If you are building a database management system, take a look at it. Otherwise, use one of the other options.

Part IV

Getting Ready to Publish

The 5th Wave

By Rich Tennant



"Of course your current cell phone takes pictures, functions as a walkie-talkie, and browses the internet. But does it shoot silly string?"

In this part . . .

So you have a user interface, and your app does stuff. Great! Now it's time to think about a few things that you might not have thought of. How about taking some of the device's sensors for a spin? Have you thought about advertising in your app, or providing some in-app purchasing? What about storing some of your app's information in Microsoft Azure? After you've considered all of this, maybe you're ready to test and publish to the store.

Chapter 13

Integrating with Hardware

In This Chapter

- ▶ Using the sensors on devices
 - ▶ Collecting contextual information
 - ▶ Getting feedback to the user
-

WinRT does one thing that is really hard to do from the .NET framework: It gives us access to devices. The webcam, microphone, sensors, near-field communication, and the touch screen itself is really hard to access from .NET, and it's very easy in WinRT.

Tight integration with the hardware is important for making the app feel alive to the user. When the user is holding a device and she turns around, have the application react to that if it's appropriate. When the user touches something on the screen, provide some haptic feedback (that little bump that your phone does when you touch a button). Give the user the ability to communicate with your app by voice.

Closely related to the sensors on a device is the media that can be shown on it. The camera and microphone gives a content creator a path to generating content that makes her device hers, and the HTML5 features I discussed in Chapter 5 close the loop, and give her a way to view that content.

WinRT takes all of the cryptic communication structures out of programming for the device's sensors. The camera, compass, GPS, accelerometer, proximity, and other sensors all have an object-oriented API that is available from C#, C++, or JavaScript. Making the device feel alive is a priority of Microsoft's — make sure you make it a priority in your app, too.

Using the Camera and Microphone

These days, almost everything that runs Windows comes with a microphone and a camera. Monitors for desktop PCs even have them now. Creating video and audio media is just something that is expected from Windows, and yet it is fairly difficult compared to other operating systems.

Part of that is because the APIs needed to get to the camera are fairly obscure. The awesome video source dialog box at WM_CAP_DLG_VIDEOSOURCE, for some reason, is just not something that everyone knows about or uses.

WinRT changes that with the Media APIs. Now devices like the webcam are much easier to access via C# or JavaScript — no longer do you have to use C++ or depend on a third-party to get to video sources.

On top of that, WinJS makes good use of the HTML5 media tools to bring the video to the user interface with a minimum of fuss. Taking a photo, recording a video, or creating an audio feed is pretty much handled for you thanks to the WinJS HTML5 renderer.

Windows.Media API

The `Windows.Media` API is a collection of namespaces that wrap up the previously confounding `vwf.dll` library into a nice object-oriented package. The functionality of the library is divided pretty evenly between capturing media and managing it afterwards.

Media capture

The most useful class in the namespace is probably `Capture`. All of the Windows 8 UI for selecting a device can be found here, such as the `CameraCaptureUI`. This class is the super-simplified version of the media API, with one-liner photo capture capabilities:

```
var cameraCapture = new Windows.Media.Capture.  
    CameraCaptureUI();  
cameraCapture.captureFileAsync(  
    Windows.Media.Capture.CameraCaptureUIMode.photo).then(  
    process(item), error(err));
```

That's all there is to it, but you don't get much in the way of options.

There are more detailed, finer-grained features of the API too. Capturing media generally depends on selecting a device, and the `Windows.Media`.

Devices namespace should help you out with that. For instance, the `MediaDevice` class has a `getVideoChapterSelector` class that gets the device ID for the selected video recording device on the host machine. This can be used with the `MediaCapture` class for much more detailed control of the webcam.

The section, “Accessing the webcam from your app” later in this chapter offers good initial case studies of using the capture APIs.

Encoding and management

Getting the media from the device is one thing, but making it useful for your app is something else altogether. There is a fine selection of tools in the `Windows.Media` namespace, which assist with the encoding, transcoding, and conversion of video and audio files in a number of different ways.

Encoding in WinRT is based on the Microsoft Media Foundation introduced in Windows 7. It does three main jobs for you in the encoding department:

- ✓ Taking a video or audio stream and turning it into a formatted file — for instance, encoding a video as an MP4.
- ✓ Encoding two streams into a single file — for example, taking an audio and video stream and making a single stream out of it.
- ✓ Taking an encoded, multiplexed stream and saving it as a file.

The principle class for all of this media magic is the `Windows.Media.Transcoding.MediaTranscoder`. The `MediaTranscoder` has a `prepareFileTranscodeAsync` function that accepts a profile and asynchronously creates an output file. The profile contains the information about the encoding that is to take place.

For instance, this is how the `MediaTranscoder` might create an MP4 video:

```
var transcoder = new Windows.Media.Transcoding.  
    MediaTranscoder();  
var profile = Windows.Media.MediaProperties.  
    MediaEncodingProfile.createMp4(  
        Windows.Media.MediaProperties.VideoEncodingQuality.  
        hd1080p);  
return transcoder.prepareFileTranscodeAsync(sourceFile,  
    destinationFile, profile);
```

Signing the PlayTo contract

`Windows.Media` is where you will find the PlayTo contract classes, too. PlayTo is the contract that you have to fulfill to get your media somewhere

other than your app, or to hold up your app as an example of something that can play certain types of media.

You can read about using the PlayTo contract in Chapter 9.

Accessing the webcam from your app

These are the main things that users want to do with their webcam: take a picture, record audio, or capture a video. They each have their own ins and outs, but are largely very similar.



Videoconferencing is a popular use of webcams, too. Passing network streams of video is outside the scope of WinJS, however (at least as of this writing). You need to work in C++ to make that happen.

Snap a photo

Taking a photo is pretty simple with the `CameraCaptureUI` class. You can use this for simple in-app image captures for things like profile pictures and whatnot.

1. **Add the webcam capability to the `package.appxmanifest` file by clicking on the Capabilities tab and selecting the Webcam check box.**

The user sees as message asking if it's okay to use the webcam, as shown in Figure 13-1.

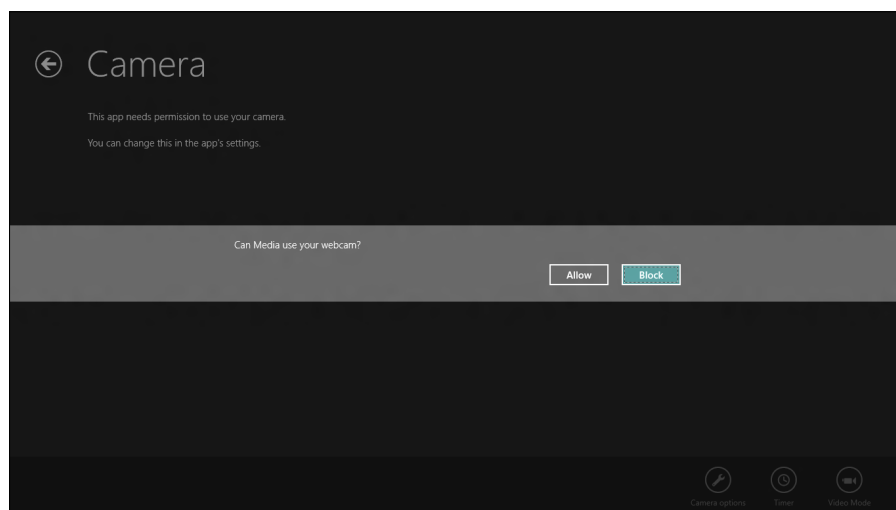


Figure 13-1:
Getting the
user's
permission.

2. Add a button to the HTML that takes the picture:

```
<body>
  <input type="button" id="takeThatPic" value="Click!
    Take a pic!" /><br />
</body>
```

3. Write a function that uses the Camera CaptureUI. It eventually looks like this:

```
function takeaPic() {
  var captureUI = new Windows.Media.Capture.
    CameraCaptureUI();
  captureUI.captureFileAsync(Windows.Media.
    Capture.CameraCaptureUIMode.photo).
    then(function (capturedItem) {
      if (capturedItem) {
        //Save the file
      }
      else {
        //the user must have changed his mind
      }
    });
}
```

4. Tie the new function to the button using addEventHandler.

```
var picButton = document.getElementById('takeThatPic');
picButton.addEventListener('click', takeaPic);
```

That's about all there is to it. The picture brings up the Capture user interface, with the preview and everything. That leaves you with not much to do except to save the file.

Record a video

I'm sure you aren't surprised to find out that the CameraCaptureUI can record video instead of audio if you want it to. All you have to do is change the mode in the takeaPic function:

```
function takeaPic() {
  var captureUI = new Windows.Media.Capture.
    CameraCaptureUI();
  captureUI.captureFileAsync(Windows.Media.Capture.
    CameraCaptureUIMode.video).then(function
    (capturedItem) {
    if (capturedItem) {
      //Save the file
    }
    else {
      //the user must have changed his mind
    }
  });
}
```

You can also capture video using the `MediaCapture` class. This has a few benefits, notably the profile you can set up. It gives you a lot more options in the way your video is recorded.

That profile is made up of `MediaProperties`, which is a class that is designed around defining specific properties of the video or audio being stored. Some of the property values that you can set include

- ✓ `AudioEncodingProperties`
- ✓ `ImageEncodingProperties`
- ✓ `videoEncodingProperties`
- ✓ `ContainerEncodingProperties`
- ✓ `MediaRatio`
- ✓ `MediaPropertySet`
- ✓ `MediaEncodingProfile`

These settings allow you to get much greater control over the video, but you pay a price in greater code complexity. You'll need to wire up your own asynchronous methods and provide your own UI for the video capture.

The general process is

1. Enable the webcam capability.
2. Create and initialize the `MediaCapture` object.
3. Create and initialize the encoding profile.
4. Make a recording UI.
5. Handle the start and stop recording methods.

To see all of this in action, check out the recording sample in the SDK samples. It has a good, if slightly overly complex, breakdown of how it all works.

Capturing audio

Audio recording has about the same structure as video recording using `MediaCapture`. The `MediaProperties` API has the audio encoding bits that you need to save audio in a variety of formats. The built-in formats include

- ✓ AAC audio (M4A)
- ✓ MP3 audio
- ✓ Windows Media Audio (WMA)

There will be more formats, and, of course, there is an API to build your own `MediaProperties` from existing codecs.

Collecting Data from Sensors

Laptops don't have a lot of sensors. They are pretty much about the keyboard and the screen. Boring, but true. Some of them have a camera, which adds a little bit to the experience.

Contemporary devices, like the Surface and tablet computers, have large numbers of sensors baked in. WinRT gives you access to all of them with a minimum of fuss. The sensor namespaces in WinRT include

- ✓ **Geolocation:** GPS and the like
- ✓ **Input:** Touch screen and keyboard
- ✓ **Sensors:** All the good stuff, like a compass, accelerometer, gyrometer, inclinometer, light sensor, and orientation sensor
- ✓ **Portable:** MP3 players and phones you might hook to the device
- ✓ **Printers:** You know, for dead trees
- ✓ **SMS:** Talks to the cell network



No amount of technical information will help you use the sensor array well in your app. All I can ask you to do is to use good apps. The good apps use the sensors well. The mapping application changes from white background to black background when it gets dark. Stuff like that. You can't teach that stuff.

Getting the user's location

There was a day, not long ago (wait, it was yesterday!) when you needed to search through three or four different location APIs to see what the most accurate location of the user is. In WinRT, that at least is taken from you. All of the items that might have location are checked by the API.

Determining if the device is moved

Once upon a time, there was a wonderful sensor called the accelerometer. It could tell if the device was shifted from one position to another. Sadly,

Windows programmers could not access such a device because it was so incredibly tough to program for.

Well, no longer is that true.

The `Accelerometer` class has made things a lot easier. Now calling for the default sensor in this class makes the best of an often bad set of options.

```
accelerometer = Windows.Devices.Sensors.Accelerometer.  
    getDefault();
```

WinRT does its best to find the accelerometer for you. This could be under a number of guises — even the hard-drive lock protection. As long as it has a driver, it will be found.



You might want to code for a shake sometime. It's actually a pretty cool feature. I like my Twitter client, which updates my feed when I shake my phone. It's pretty straightforward to use the accelerometer in your project now.

You don't need to declare the accelerometer as a capability. It's a freebie.

1. Write a function that will be called when the accelerometer registers a shake. It can do whatever you want.
2. Get an instance of the default accelerometer.
3. Set the `onreadingchanged` event to the function you added.

The code for my example looks like this (working from a default `PageControl`):

```
(function () {  
    "use strict";  
    var checkingShake, accelerometer;  
    function itshook()  
    {  
        var field = document.getElementById("isitshaken");  
        field.innerHTML="It shook!";  
    }  
    WinJS.UI.Pages.define("/pages/shaken.html", {  
        // This function is called whenever a user  
        // navigates to this page. It  
        // populates the page elements with the app's  
        // data.  
        ready: function (element, options) {  
            accelerometer = Windows.Devices.Sensors.  
                Accelerometer.getDefault();  
            accelerometer.onreadingchanged = itshook;  
        },  
    });  
})
```

```
updateLayout: function (element, viewState,
    lastViewState) {
    /// <param name="element" domElement="true" />
    /// <param name="viewState" value="Windows.
    UI.ViewManagement.ApplicationViewState" />
    /// <param name="lastViewState"
    value="Windows.UI.ViewManagement.
    ApplicationViewState" />
    // TODO: Respond to changes in viewState.
},
unload: function () {
    // TODO: Respond to navigations away from this
    page.
}
});
})();
```



If your device does not have an accelerometer, this code doesn't do anything. No amount of shaking your monitor helps (trust me, I've tried). What's more, I don't think the simulator does it either. If you have a device without an accelerometer, it won't work. I tried it with my wife's ThinkPad, which has a hard-drive shake monitor, and it worked there, for what that's worth.

But that `onreadingchanged` event isn't the only toy. `GetCurrentReading` gives you a three-dimensional idea of where the device is in space. Although it's all relative, it does tell you if the user is moving the device left or right, up or down.

The result from `getCurrentReading` gives you those X, Y, and Z parameters in the return values. To use it, try something like this:

```
function getThePosition() {
    var position = accelerometer.getCurrentReading();
    if (position) {
        var x = reading.accelerationX.toFixed(2);
        var y = reading.accelerationY.toFixed(2);
        var z = reading.accelerationZ.toFixed(2);
    }
}
```

Using this isn't straightforward because you need to keep checking the position when the user is in a position to change it. It requires a little algorithmically trickery. Just remember that anything that speaks directly to the hardware of the device like this is streaming cast under WinRT, so you might be in a good position to check the values more often than you think.

Being aware of lighting

You might not be thinking about ambient lighting when you are writing your app, but you should be. You can use the lighting in the room where the user is to determine what styles to show in your app.

The light sensor is less obscure than the others I've covered in this chapter. It's either built into your machine or it isn't, so as long as you do a little defensive coding, it's a good tool to use.

Using the light sensor is a lot like the other sensors. You'll see this pattern repeated throughout the sensor array:

1. Declare function variables.
2. Write a function for the `on<sensor>changed` event handler.
3. Handle the event.
4. In the handler, get the current reading and do something with it.

This process a pretty common pattern for Windows 8, I'm starting to think. If you are familiar with mobile development in other platforms, you've probably seen this before. Here's how it looks in JavaScript:

```
(function () {  
    "use strict";  
    var light;  
    function onLightChanged(eventArgs) {  
        light = lightSensor.getCurrentReading();  
    }  
    WinJS.UI.Pages.define("/pages/light.html", {  
        // This function is called whenever a user  
        // navigates to this page. It  
        // populates the page elements with the app's  
        // data.  
        ready: function (element, options) {  
            lightSensor.onreadingchanged = onLightChanged;  
        },  
        updateLayout: function (element, viewState,  
            lastViewState) {  
            // TODO: Respond to changes in viewState.  
        },  
        unload: function () {  
            // TODO: Respond to navigations away from this  
            // page.  
        }  
    });  
})();
```

This all goes back to giving users a good experience — as though the device is an extension of them. Knowing what sensors your users are likely to have, and taking advantage of that, is an important part of application development in the Windows 8 space.

Handling the fact that the sensor is not there is important too. Don't forget to check for `null` values. Don't depend on user input. Remember, there is a chance that the machine is misconfigured, or that the user has a switch turned off. Assume the worst.

Touching the Surface

Windows 8 is a “touch first” operating system. Machines that come out with Windows 8 preinstalled largely have touch screens enabled, and the tablets of course have a touch screen.

As such, you would expect WinRT to have largely touch-ready APIs and tools, and you would be right. All of the user controls in WinJS are touch-ready, and there is a very solid touch API. All of this is covered in Chapter 6 in depth.

Comprehensive mouse and pointer features

Not every device has a touch screen. Desktops and some laptops just have a mouse and a keyboard, and that's fine. Everything that is touch-enabled is mouse-enabled too.

There are some design considerations to use with the mouse and the keyboard with Windows Store apps, however. The scroll wheel and whatnot may mimic certain functions of the touch screen, for instance.

I cover the mouse and keyboard and using them with WinJS UI controls in Chapter 5.

Writing with the pen

The last of the input devices you are likely to encounter (at least until the Kinect gets installed on everything — that's a different API) is the pen. An awesome interface for the pen is built into the WinRT API, but it doesn't get used often enough. In this section, I take you on a brief tour of the pen — at least enough to get some ink on the screen.

Making a place to draw

To start off, you need a place to draw. In the WinJS world, that means a Canvas element. The Canvas element is an element devised by the W3C (the group that creates the web standards) to allow for faster graphic creation on the web. Scalable vector graphics, or SVG, have been the standard for dynamic graphics on the Internet for many years, but there was a need to move to an updatable bitmap model for more advanced color and motion animation.

As it turns out, it's perfect for Windows 8, too.

Adding a canvas element is easy:

```
<body>
  <canvas id="pieceOfPaper"></canvas>
</body>
```

Using it is less so. Fortunately, a few WinRT bits make writing on the Canvas a lot easier.

Listening for pen input

To get the pen input, you need to use parts of the `Windows.UI.Input.PointerPoint` class. (For the record, I did not name that class.) Either way, that's where you're going.

- 1. The first thing that you need to do is to get a reference to the InkManager. Just put that in the declarations section of your JavaScript page.**

```
var inkManager = new Windows.UI.Input.Inking.
    InkManager();
```

- 2. Get an instance of the canvas and set up some event handlers.**

You'll write them in the next few steps. You can put all of this in the `onactivated` event handler.

```
app.onactivated = function (args) {
    if (args.detail.kind === activation.
        ActivationKind.launch) {
        if (args.detail.previousExecutionState
            !== activation.ApplicationExecutionState.
                terminated) {
            inkCanvas = document.
                getElementById("Canvas");
            inkContext = inkCanvas.
                getContext("2d");
```

```

        inkCanvas.addEventListener("MSPointer
Down", onPointerDown, false);
        inkCanvas.addEventListener("MSPointer
Move", onPointerMove, false);
        inkCanvas.
addEventListener("MSPointerUp", onPointerUp,
false);
    } else {
        // TODO: This application has been
reactivated from suspension.
        // Restore application state here.
    }
    args.setPromise(WinJS.UI.processAll());
}
};

```

3. You need three functions for those three events in Step 2.

One handles the down event, one handles move, and one handles up.

```

function onPointerDown(evt) {
    pointerDeviceType = getPointerDeviceType(evt.
pointerId);
    if ((pointerDeviceType === "Pen") ||
((pointerDeviceType === "Mouse") && (evt.
button === 1))) {
        if (pointerId === -1) {
            var current = evt.currentPoint;
            inkContext.beginPath();
            inkContext.lineWidth = strokeWidth;
            inkContext.strokeStyle = strokeColor;
            inkContext.moveTo(current.position.x,
current.position.y);
            inkManager.processPointerDown(current);
            pointerId = evt.pointerId;
        }
    }
}

function onPointerMove(evt) {
    pointerDeviceType = getPointerDeviceType(evt.
pointerId);
    if ((pointerDeviceType === "Pen") ||
((pointerDeviceType === "Mouse") && (evt.
button === 1))) {
        if (evt.pointerId === pointerId) {
            var current = evt.currentPoint;
            inkContext.lineTo(current.
rawPosition.x, current.rawPosition.y);
            inkContext.stroke();
        }
    }
}

```

```

        inkManager.
        processPointerUpdate(current);
    }
}
function onPointerUp(evt) {
    pointerDeviceType = getPointerDeviceType(evt.
    pointerId);
    if ((pointerDeviceType === "Pen") ||
    ((pointerDeviceType === "Mouse" && (evt.
    button === 0))) {
        if (evt.pointerId === pointerId) {
            inkManager.processPointerUp(evt.
            currentPoint);
            inkContext.closePath();
            renderAllStrokes();
            pointerId = -1;
        }
    }
}
}

```

4. Last, you need that **getPointerDeviceType** function.

You could duplicate the code in every event, but why would you?

```

function getPointerDeviceType(pId) {
    var pointerDeviceType;
    var pointerPoint = Windows.UI.Input.
    PointerPoint.getCurrentPoint(pId);
    switch (pointerPoint.pointerDevice.
    pointerDeviceType) {
        case Windows.Devices.Input.
        PointerDeviceType.touch:
            pointerDeviceType = "Touch";
            break;

        case Windows.Devices.Input.
        PointerDeviceType.pen:
            pointerDeviceType = "Pen";
            break;

        case Windows.Devices.Input.
        PointerDeviceType.mouse:
            pointerDeviceType = "Mouse";
            break;
        default:
            pointerDeviceType = "Undefined";
    }
    deviceMessage.innerText = pointerDeviceType;
    return pointerDeviceType;
}

```

Collecting the ink

Now that you have the event handlers in place, you can just run the app and give it a draw. Remember, if you don't have a pen on your device, you can use the Simulator to make like a pen.

This is a barebones example of pen input. Two awesome samples in the Windows SDK Samples are Input: Ink Sample and Input: Simplified Ink Sample. What I've shown you is simpler than both of those, so if you need something more, you might want to check out the Microsoft code as well.

Chapter 14

Preparing for the Store

In This Chapter

- ▶ Making sure your app works as advertised
 - ▶ Getting approval from Microsoft
 - ▶ Keeping the app up to date
-

As with many of the mobile app stores, Microsoft has an approval process for apps that go in the Windows Store. If you want your app to be up for sale to the general public, you have to go through that process.

Although Microsoft has made the Store certification process very transparent, it is still quite complex. There are a lot of moving parts, and keeping an eye on everything is important.

Before you ever have to go to the Store, you need to make sure your app meets certification. There is a guideline for this, and I go over it in this chapter. The readier you are, the faster you start making money.

But before you worry about certification, you need to make sure your app works. I haven't talked a lot about unit testing this far, but I do here.

After all of that, you need to get the app in the Store and keep it up to date. Aside from fixing bugs, you also need to keep the app fresh so that you have happy users. This is really simple using Visual Studio, but I go over it in this chapter.

Testing Your App

No unit testing or code analysis template for JavaScript apps in Visual Studio exists. Testing is a “bring your own” experience.

That being said, you still have to test your apps. Just clicking through the app before you send it off to the Store is not enough, but it does have to be part of the testing process.

Because there isn't any unit-testing project type in the JavaScript projects, you need to come up with auxiliary options.

Old-fashioned testing

One of the tricks I've come up with is testing JavaScript with JavaScript. It isn't the optimal situation because you don't have a testing framework, but it's better than nothing:

1. **Right-click the project and add a folder. Name it Tests.**
2. **Right-click the new folder and add a new Page Control.**
The default name is fine.
3. **Open the HTML file it created and drag all of the .js files onto the code surface next to the other .js files.**
4. **Add a function that does something in the main project.**
This is the test method.
5. **Add a button that calls the test method.**

That might be as good as it gets for many projects. Without a unit test project type, you're pretty much back to old-fashioned "Click the button to see if it works" testing.

Getting a unit testing framework

Some of the new research in Windows 8 revolves around using a unit testing framework with your JavaScript, just as if you are writing a jQuery or Node.JS project. Some of those frameworks work well with Windows 8.

For instance, take QUnit as an example. Originally created to be the unit testing environment for jQuery projects, QUnit is now a standalone project. The source can be included in a Windows 8 project, and your tests can get a lot more interesting.

To run tests using QUnit, download the QUnit file and the CSS file from GitHub (at <https://github.com/jquery/qunit>), and then put this into your default HTML in your tests file:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Qunit</title>

  <!-- WinJS references -->
  <link href="//Microsoft.WinJS.1.0.RC/css/ui-dark.css"
        rel="stylesheet" />
  <script src="//Microsoft.WinJS.1.0.RC/js/base.js">
    </script>
  <script src="//Microsoft.WinJS.1.0.RC/js/ui.js">
    </script>

  <!-- Qunit references -->
  <link href="/css/default.css" rel="stylesheet" />
  <script src="/js/default.js"></script>
  <script type="text/javascript" src="js/qunit.js">
    </script>
  <link rel="stylesheet" type="text/css" href="css/
    qunit.css" />
</head>
<body>
  <div id="qunit"></div>
</body>
</html>
<tip>

```

Make sure the script source matches.

This gives you a reference point to start your tests. We just need to insert a test into the Windows 8 code. A test looks like this, from the sample website:

```

app.onactivated = function (args) {
  if (args.detail.kind === activation.
    ActivationKind.launch) {
    if (args.detail.previousExecutionState
      !== activation.ApplicationExecutionState.
        terminated) {
      test("hello test", function () {
        ok(1 == "1", "Passed!");
      });
    } else {
      // TODO: This application has been
      reactivated from suspension.
      // Restore application state here.
    }
    args.setPromise(WinJS.UI.processAll());
  }
};

```



QUnit gives you a nice output on the HTML page, just like Figure 14-1.

I had to comment out line 1021 of the QUnit.js file because it failed a security requirement of Windows 8. This might not be the case in the final version.

Clearly you need to write tests that work with your code, but QUnit is a good start. You can find more information at qunitjs.com. For more about unit testing, I recommend James Bender's excellent *Professional Test Driven Development with C#*, published by Wrox. It's in C#, but the concepts are as sound as they come.

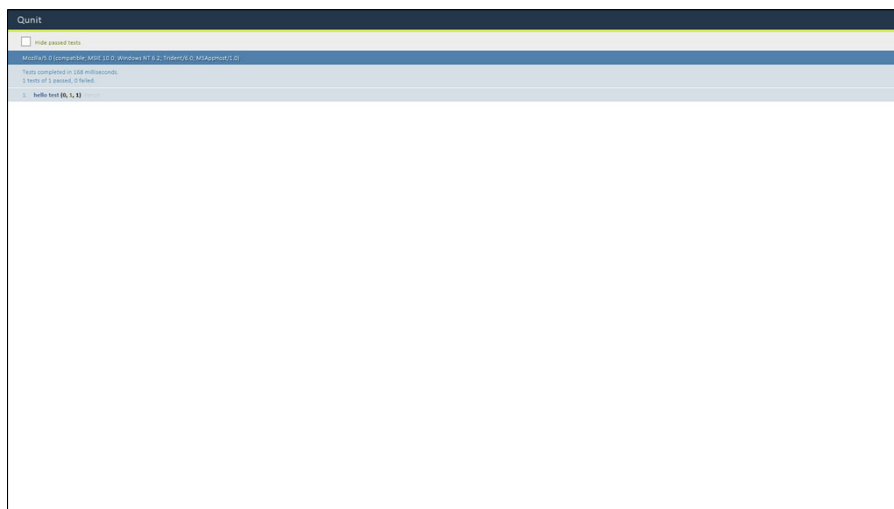


Figure 14-1:
Using a unit
test frame-
work like
QUnit.

Verifying runtime behavior

There are some things that unit tests can't fix. The first and most significant is the app's interaction with the user. Windows 8 is all about the user experience, and that is just hard to unit test.

First, you need to manually check all of the possible activations of the app. Just clicking the tile and launching isn't good enough. There are lots of ways to launch an app.

- ✓ If your app is a search target, run a search from another app and launch your app from there.
- ✓ If your app is a share target, check to make sure every share type that you support works as expected. This means you will have to go get a lot

of apps that make the kind of output that your app shares and test them. Don't forget to test all features both from suspension and from a cold start.

- ✓ Click the tile, but that's kind of a given. Just testing from F5 in Visual Studio ain't good enough.
- ✓ Check to make sure secondary tiles work as expected. Lay down secondary tiles from every page in the app that supports them, and make sure the app launches as expected.
- ✓ If you have registered for a protocol, like handling `irc:` or something, activate your app from a link.
- ✓ Do you have Toast notifications? Launch your app from them.
- ✓ Finally, if you have any file registrations, or are using the File Picker contract, make sure clicking a registered file type launches your app in the correct state.

Now that your app is running, the next check is to test all of the interactions with Windows 8. All of the contracts, charm commands, animations, and anything that uses something in WinRT basically needs to be checked.

- ✓ Check all of the contracts. Anything that you have declared in the package.appxmanifest should be checked against as many different apps as you can get your hands on.
- ✓ Look at your tile. Are you updating it with WNS or within the app? Make sure it works as expected. Reboot the machine, and make sure it is there.
- ✓ Check all of the possible views. Use the emulator if you have to and check every resolution. Test snapped and filled view in all resolutions. Then rotate it all four ways.
- ✓ Make certain that all of the expected touch features work. Check the app bar on every page. Check the Settings flyout. Make sure you don't override the Charms bar on every page. Test every control on every page with touch, pen, and mouse.
- ✓ Make sure the sound works. Try it on every device you can. If you are using feedback sounds, make sure the app still makes sense if the device is muted.
- ✓ Turn off the networking and make sure the app still works, or at least fails gracefully. Assure yourself that the roaming profiles, if you are using them, have a fallback position and still work without a network.
- ✓ Check every animation. Look at them in snap and fill view. Make sure they work when the machine is busy doing other things. Make sure that long-running animations work if the user rotates the screen while the animation is playing.

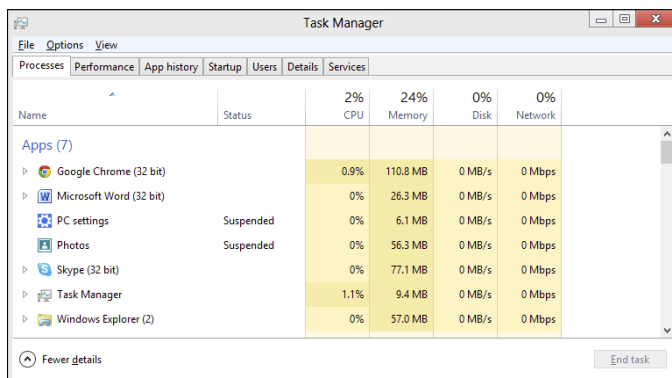
- ✓ Is the app responsive? Remember, the fast and fluid rule of Windows 8 is taken very seriously by the testers. The app should respond instantly to any user interaction.
- ✓ Enter junk into the app and make sure it fails gracefully. Let your cat walk on it. Touch where you aren't supposed to. Upload corrupt files. Make sure your error-handling works.
- ✓ Assure that your app meets the baseline for accessibility. Test it with the magnifier and the screen reader. Test speech input.
- ✓ Change the base language of your machine and make sure the app still works. The Store is global. Change your time zone. Change your date format. Use a virtual machine if you have to.
- ✓ Finally, if you want to allow people that use the ARM architecture (the lightweight mobile chip that runs Windows RT devices) to use your app (and I would recommend that), test it on an ARM device. The emulator doesn't seem to support that yet, but maybe it will by the time you read this.

Verifying lifecycle

If your app's process lifecycle management (PLM) doesn't meet the standards, it will get you tossed from the Store. The Certification Kit (which I cover in the next section) tests some of this, but it will be manually tested, too.

The first thing you should do is to check suspended status. Launch your app from the tile, switch to another app, and then switch to the desktop. Run the Task Manager, and click **View** ⇨ **Status Values** ⇨ **Show Suspended Status**. It will look something like Figure 14-2. Watch your app suspend. Make sure the memory footprint reduces.

Figure 14-2:
The Task
Manager
with
suspended
apps.



Task Manager					
File Options View					
Processes Performance App history Startup Users Details Services					
Name	Status	CPU	Memory	Disk	Network
Apps (7)					
Google Chrome (32 bit)		0.9%	110.8 MB	0 MB/s	0 Mbps
Microsoft Word (32 bit)		0%	26.3 MB	0 MB/s	0 Mbps
PC settings	Suspended	0%	6.1 MB	0 MB/s	0 Mbps
Photos	Suspended	0%	56.3 MB	0 MB/s	0 Mbps
Skype (32 bit)		0%	77.1 MB	0 MB/s	0 Mbps
Task Manager		1.1%	9.4 MB	0 MB/s	0 Mbps
Windows Explorer (2)		0%	57.0 MB	0 MB/s	0 Mbps
Fewer details		End task			

Then switch back to your app by swiping from the left (you might have to swipe a few times). Make sure the app resumes gracefully. You can keep the Task Manager on in the foreground and watch the suspension go away, and make sure the resources go back.

Test termination, too. In the Task Manager, right-click on the app and select End Task. Make sure it terminates. Then go back to the Start screen and start the app. Make sure it resumes gracefully. It doesn't have to restart the user on the same page they ended on, but it shouldn't completely forget everything that has ever happened.

Finally, make sure the app does everything it is supposed to do when it isn't running. Terminate it from the Task Manager and then check the tile, Toast, and any background tasks you have coded for. Do they still work? Do they behave as expected?

There is more about the PLM in Chapter 11. Test it well. Microsoft will.

The Windows App Certification Kit

The WACK, as the Windows App Certification Kit is called, is a tool provided by Microsoft to run the kinds of tests that your app will go through in the certification process. When you think your app is ready for the Store, be sure and WACK it first!

The WACK is basically an automated test suite that runs your app a few times on what amounts to a baseline computer (virtually, of course) and looks at a few things. Specifically, it tests whether

- ✓ All interactions with Windows are recorded in the package.appxmanifest file
- ✓ Your app is localized for all languages you say you support
- ✓ Your app recovers if it crashes
- ✓ You are still in debug mode
- ✓ The package is encoded correctly (in other words, that it's not corrupt)
- ✓ The performance eats the processor
- ✓ You use any disallowed APIs
- ✓ All of the security specifications are followed (learn more by running WACK)

Running the Windows App Certification Kit is pretty straightforward, but there are a few tricks:

1. **Run WACK by pressing the Windows button and typing Windows App — it's probably the only app named that on your machine.**
2. **On the startup screen, click Validate Metro Style app.**

It might be called something a little different by the time you read this, but you don't want to click Desktop Application.

3. **After the app has surveyed the WinRT apps on your system, you'll get a list of them like Figure 14-3.**

Check the ones you want to check — I am testing ToDoToday, which is the original name for POINTtodo.

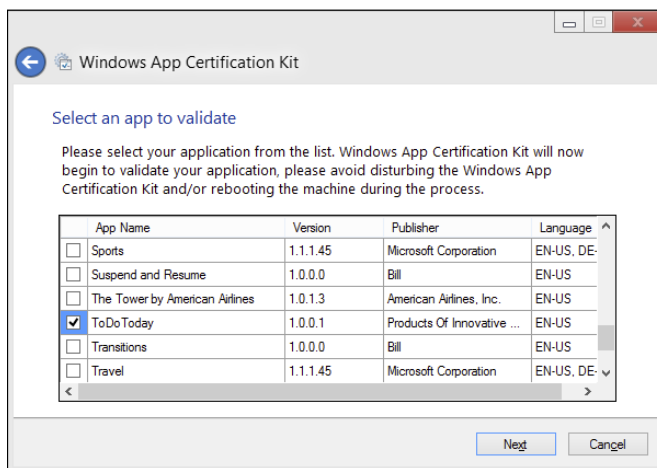


Figure 14-3:
Selecting
your app for
testing.

4. **Click Next, and go get a cup of coffee.**

WACK runs your app a number of ways. You'll see the screen flash a lot — don't interact with your app. Sometimes it will take a long time to run the test. Just be patient. It will come back.

You will also see the console a number of times. Wow, this really does take a while. Eventually, the tests finish and prompt you to save an XML file.

5. **Save the XML file someplace you can find it.**

This is the file you upload later to show that you have passed the basic tests. That doesn't mean you will pass app certification, however.

6. **Next you get your result — Pass, Fail, or Warning. This screen has a link to the results. Click it.**

My app failed because of a known bug in the current version of WACK. It shows nearly all apps as being in debug mode, even when they aren't. That will likely be fixed by the time you get the tool.

Manually Checking Windows Store Style

In the early stages of Windows 8 app building, to get early placement in the Store, you had to meet personally with a Microsoft engineer. They used a manual checklist to walk through the user experience points of an app to make sure that all of the bases were covered.

Given the possibility that this list might go away, I've paraphrased it here. I found it useful in designing the POINTtodo app, and in designing this book, for that matter.

✓ Scenario

- Has a clear mission and scope; implementation is far enough to represent the final scope

✓ UX Polish

- Follows Windows Store style principles
- Conveys brand and purpose
- Follows a consistent layout and alignment
- Leverages typography and whitespace, and follows a font hierarchy
- Embraces Windows 8 personality
- Designed for a great first impression
- Animations are purposeful, create continuity, and convey confidence to the user
- Designed for touch

✓ User Experience: Controls, Navigation, Animation

- Content is primary navigation mechanism
- Makes appropriate use of the app bar
- Makes appropriate use of commanding surfaces and respects the edges
- Displays unobtrusive, meaningful, and actionable error-handling
- Leverages Semantic Zoom appropriately

- ✓ User Experience: Views & Form Factors
 - Supports landscape and portrait views
 - Adapts to different form factors and aspect ratios
 - Handles snapped, filled, and full view states
 - Pans and scrolls in single axis to create sense of stability
 - Is functional and feels natural on a system that has only a mouse or keyboard
 - Is functional and feels natural on a system that uses an onscreen keyboard
- ✓ User Experience: Settings
 - Implements Settings charm and does not have settings and preferences anywhere else in the app
 - Roams user settings and user state
 - Has a great sign-in/sign-out experience
- ✓ User Experience: Contracts
 - Implements search contract
 - Implements share target contract
 - Implements share source contract
 - Has beautifully branded contracts UI
 - Leverages file pickers
- ✓ User Experience: PLM
 - Handles PLM states appropriately
 - Makes appropriate use of background tasks
- ✓ User Experience: Networking
 - Behaves well while offline and while on intermittent network connectivity
- ✓ User Experience: Tiles and Notifications
 - Has a beautiful, branded tile
 - Has a live tile or a secondary tile
 - Uses Toast notifications for time-sensitive, relevant information
 - Properly handles activation from contracts, secondary tiles, and notifications

- ✓ Store
 - Expected to pass the Windows 8 App certification requirements.
 - Passes the Windows App Certification Kit (WACK) tests
- ✓ Performance
 - UI is always responsive and provides visual feedback on interactions
- ✓ Performance: Animation
 - Sustains 60 frames per second
- ✓ Performance: Lifecycle
 - Optimize your suspend and resume handlers
 - Suspends and resumes in allotted times
- ✓ Performance: Cache and Local Data
 - Rely on local, packaged content as much as possible
- ✓ Performance: Startup
 - First page must load in less than five seconds.
 - Take advantage of bytecode caching
- ✓ Performance: Memory
 - Manages its memory working set to desired targets
- ✓ Performance: Network
 - Manages network resources appropriately
- ✓ Performance: Background Tasks
 - Respects allotted timeslots assigned by background tasks
- ✓ Other
 - Is accessible
 - Is localized

All of these are things that more or less have to be tested by looking at the app, and that's exactly how it is done.

Pushing to the Store

In order to push your app into the Store, you start with a developer account. After registering for this, you can upload your package and submit it for testing. After the testing process and anything that goes with it, your app is made available in the Store.

Registering for a developer account

If you don't already have one, you will need to register for a developer account. You can register for an account — or log into your existing account — right from Visual Studio. In the Project→Store menu is an Open Developer Account item. Click that to be taken to your web browser.

You start at the Developer Portal, where your Live account information is shown to you to confirm that is the information you want used. If you are using a browser other than IE10, you might need to log in first. Click Continue.

On the next screen, you need to add a little information to your Live account (I needed to update my phone number). At the bottom is an important step — you choose your publisher name. This is what will be shown when the user installs your app, so it is important for the overall branding scheme.

Next you will see the developer agreement. Read it. I am not going to put it all here because it is long, but you need to actually look through it. For instance, the fee information is there:

“You will pay Microsoft a Store Fee for each of your apps that Microsoft makes available through the Windows Store. Microsoft may deduct and retain the store fees you owe Microsoft from any amounts it receives from customers for those apps. You will also pay any Store fees that Microsoft does not deduct, within five (5) days after you receive a written request. The Store Fee is the percentage of Net Receipts that is retained by Microsoft as a fee for making your app available through the Windows Store. That percentage is 30%, unless and until your app takes in total Net Receipts of USD\$25,000, after which time the percentage is 20% for that app.”

That information doesn't show up very many other places.

After this, you need to part with some hard-earned cash. There is a fee for setting up the account, to prevent people from doing it willy-nilly. As of this writing, the fee was \$49 USD, although early adopters got a code to get it for free. I still had to give them a credit card number, though.

You're done. If you wish, you can set up the account to send payment directly to your checking or savings account. You can also go straight to the dashboard and get started registering your app.

Submitting an app

The dashboard is shown in Figure 14-4. From here, you can submit new apps, see how people are using your apps, and check out how much money you are making. Also, under the Profile heading, you can set up your payout account as promised.

More important is getting ready to submit your app to the Store. You have a multistep process to go through before you even upload the app. The single most important thing you can do, however, is reserve your app name.

Your app name

App names are unique. If you code up a name in your app and your Tile and whatnot, and then find out it isn't available when you go to submit, you're going to be in bad shape.

You have a year to submit an app after you have reserved a name. Use it. Register your name right away, as soon as you think of the idea, and then take your time coding. The name is important.

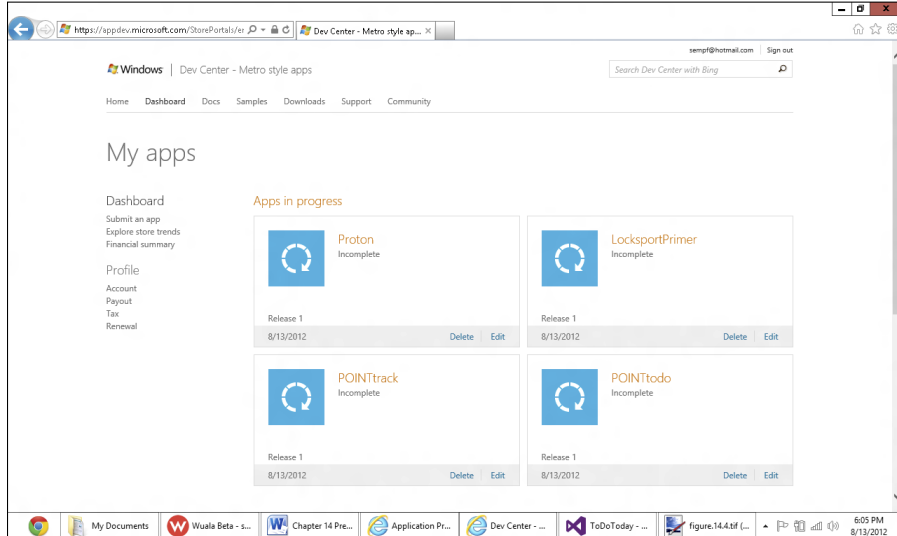


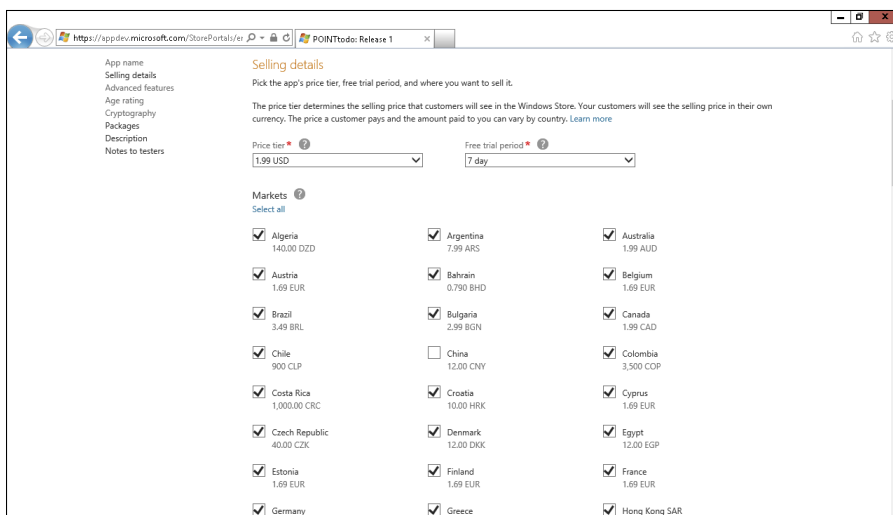
Figure 14-4:
Your app
dashboard.

Selling Details

In the Selling Details section of the dashboard, you get to pick a price. Of course, one option is free, but what fun is that? (I cover making money with your app in Chapter 16). Other than free, you can set the app price to as low as \$1.49, as high as \$999.99, or at any 50-cent increment in between. You also get to set a free-trial length, which I discuss in Chapter 16.

That's not awesome. What's awesome is Figure 14-5, where you can set the markets in which your app is available. I set the price for POINTtodo at \$1.99 USD and Microsoft helpfully points out that in Belgium I'm charging 1.69 euros, in Denmark it is 12 kroner, and in Columbia, it will be 3,500 of whatever Columbians use for money.

Figure 14-5:
The app store calculates cost in other markets.



Just set your checkmarks in the markets in which you want your app to be made available. Note this: You had better be ready to support the languages of other countries you choose to be marketed in! In POINTtodo, for instance, I have to make sure I support right-to-left reading order if I pick something like Hong Kong (where it will be 16 Hong Kong dollars). I love technology.

At the bottom of the Selling Details page, you have a few more items to deal with:

- ✓ Release date
- ✓ App category and subcategory. This is important because it represents where you set yourself up on the Store.

- ✓ Minimum DirectX Support
- ✓ Minimum RAM
- ✓ Accessibility requirements — are they met?

Advanced Features

In the Advanced Features section, you set up push notifications, Live Connect services, and in-app offers. I cover in-app offers in Chapter 16, so here I focus on the first two. And as it turns out, there isn't much you need to do on the site, either — most of this work has to be done in Visual Studio.

The most significant thing that you need in order to make use of WNS and Live Services is the Package Security Identifier and the Client Secret. These are used by the Windows 8 app to authenticate with the services.

To get the Package Security Identifier and the Client Secret, just click the Push Notifications and Live Connect services info link, and then click Identifying Your Service.

Age Rating

As part of the app profile, you get to select an age rating. Now, this age rating is for appropriateness, not necessarily functionality. For instance, POINTto do can be rated as Suitable for Young Children even though no reasonable three-year old needs a task list. Realistically, unless you are targeting children, use the 12+ setting.

Here are the options, right off the site:

- ✓ **3+, Suitable for Young Children:** These applications are considered appropriate for young children. There may be minimal comic violence in nonrealistic, cartoon form. Characters should not resemble or be associated with real-life characters. No content should be frightening, or contain nudity or references to sexual or criminal activity. Apps with this age rating also cannot enable features that could access content or functionality unsuitable for young children. This includes, but is not limited to, access to online services, collection of personal information, or activating hardware such as microphones or webcams.
- ✓ **7+, Suitable for Ages 7 and Older:** Apps with this age rating have the same criteria as the 3+ applications, except these apps can include content that might frighten a younger audience and can contain partial nudity, as long as the nudity does not refer to sexual activity.
- ✓ **12+, Suitable for Ages 12 and Older:** Choose this rating if you are not sure which age rating to select for your app. Apps with this age rating can contain increased nudity of a nonsexual nature, slightly graphic

violence towards nonrealistic characters, or nongraphic violence towards realistic human or animal characters. This age rating might also include profanity, but not of a sexual nature. Also, apps with this age rating may include access to online services, and enable features such as microphones or webcams. Can also include more profanity than is allowed in a 7+ app, but the profanity cannot be sexual in nature.

- ✓ **16+, Suitable for Ages 16 and Older:** Apps with this age rating can depict realistic violence with minimal blood, and they can depict sexual activity. They can also contain drug or tobacco use and criminal activities, and more profanity than would be allowed in a 12+ app, within the limits laid out in Section 5 of the certification requirements.



Notice that you can't have adult content (17+) in a Windows Store app. Sorry.

As part of the age certification, certain countries require rating certificates. You can upload those on this page.

Cryptography

I haven't covered the topics of security and risk in this book very much. They are complex, in-depth topics that have a lot of variables and matter different amounts to different markets, people, and apps.

So if you have found that you need to encrypt something other than password encryption, copy protection, authentication, digital-rights management, or using digital signatures, you need an ECCN for your app. The ECCN is an Export Control Classification Number. It classifies items for export to countries with different standards for content than the United States. For more information, go to the Bureau of Industry and Security website at www.bis.doc.gov/.

Packages

Finally, you get to upload your package. If you haven't finished coding or testing, this would be a good time to stop and take stock of the situation.

If you have finished, you are ready to go. You can drag the package file to the file input type field that Microsoft chose to use, or you can just use Visual Studio to do it. Because you have to use Visual Studio to create the package, and it will upload it for you as well, I'm just going to focus on that.

1. In the Project➤Store menu, select **Create App Package**.
2. Select **Yes** in the dialog box that asks about creating a package.
3. **Sign in if requested.**
4. **Select the app for which you are building a package.**

5. Set the version and build configuration.**6. Click Create.**

If you get an error about your temporary key, you may have not yet created a certificate for your project. Check out <http://go.microsoft.com/fwlink/?LinkID=241478> about certificates, if you need to work on that — it was a struggle for me and I'm very familiar with public key cryptography and code signing. I hope Microsoft makes the process easier by the time you get to it.

Anyway, when you are done uploading — from the site or Visual Studio — you'll get notified by the portal. You're almost done.

Description

Now you get to write a little marketing copy. If you go to the Windows Store and click on any random app, you can see what goes in the Description field. This text is really the one and only thing that will make people look twice at your app. Don't skimp here.

In fact, if you aren't of the writing bent, get a writer to write for you. (If you get this far into the book, then shoot me an e-mail. I might write your description. You never know.) But either way, make it good. You may think your app is good and it can stand on its own, but I *promise you* that no person will ever download it if the description isn't good.

Here is POINTtodo's description:

The problem with a task list is that it isn't ever done. You never get that sense of completion, that feeling of having gotten your work done for the day.

Task managers, on the other hand, are so complex. Outlook and the like have due dates and importance ratings and whatnot. You have to put in a task just to manage your tasks every day.

Enter POINTtodo: the task list for the rest of us.

With POINTtodo, you keep two lists. One of them is that endless Honey-do list that you won't ever want to think about, but have to. The other is what needs to get done today. Just today — don't worry about that other list.

Spend three minutes every morning moving a few tasks from the Later list to the Today list, and then tack on a few other urgent matters, and your day is planned. That easy.

It's not Shakespeare, but it is easy to read, is light on technical details, and gets all of the important details across. If you started well and got a solid design to your app, writing a compelling description should be easy. If not, well then, much luck to you!

Notes to testers

Finally, you get 500 characters to tell the testers what to look for. Don't get flowery — 500 characters is *not* a lot.

Most important, give the Windows Store testers a test account, if you have user account management, and tell them how to get into locked features, if there are any. Also, make certain any external service or asset to the app is available when you submit it.



A real human reads this and runs your app, and then makes a decision about whether you get into the Store. Don't blow this part off. Stop and think about it. I'll wait.

Surviving testing

Once all of this is done, your app goes to testing, and you wait. When it comes back — usually around a week later — it will be either passed and waiting to get into the Store or failed and waiting for you to fix it. There are a select number of failures you might get, and solutions.

- ✓ **Digitally signed file test:** Tests executable files and device drivers to verify they have a valid digital signature.
- ✓ **File-encoding test:** Tests the contents of app packages to make sure they use the correct file encoding.
- ✓ **Install-uninstall test:** Installs and uninstalls the app and checks for residual files and registry entries.
- ✓ **Launch and suspend performance test:** Tests how quickly an app launches when the user starts it and how fast it suspends when the user switches to another app.
- ✓ **Windows Store-style app-manifest test:** Tests the contents of app manifest to make sure its contents are correct.
- ✓ **Multiuser session test:** Tests how the app behaves when run in multiple sessions at the same time.
- ✓ **Restart manager message test:** Tests how the app responds to system shutdown and restart messages.
- ✓ **Safe mode test:** Tests if the driver or service is configured to start in safe mode.
- ✓ **Test for apps that crash and stop responding:** Monitors the app during certification testing to record when it crashes or stops responding.
- ✓ **Test for changes to Windows security features:** Verifies that the app uses the Windows security features and strong ACLs.

- ✓ **Test for correct folder use:** Verifies that the app writes its program and data files to the correct folders.
- ✓ **Test for debug apps:** Tests the app to make sure it is not a debug build.
- ✓ **Test for use of APIs for Windows Store-style apps:** Tests the app for the use of any noncompliant APIs.
- ✓ **Test for use of Windows compatibility fixes:** Verifies the app doesn't use any Windows compatibility fixes.
- ✓ **Test of OS version check logic:** Tests how the app checks for the version of Windows on which it's running.
- ✓ **Test of resources defined in the app manifest:** Tests the resources defined in the app manifest to make sure they are present and valid.
- ✓ **User Account Control test:** Verifies that the app doesn't need unnecessarily elevated permissions to run.
- ✓ **Windows platform support test:** Tests the app to make sure the .exe is built for the platform architecture onto which it will be installed.

To fix any of these failures, just remedy the situation in the description. Honestly, the e-mail that comes back from Microsoft (at least, the one that came back in my case) has specific instructions on fixing the app and getting it back into testing.

Managing Your App's Existence

After your app is in the Windows Store, you will want to keep tabs on the way it is used, how much it is being used, and any problems or errors. Microsoft's management experience is second to none, so you should find this very straightforward.

But so you know before you get in there, the dashboard gives you a very good set of stats on some of the following topics:

- ✓ Downloads by age group and market
- ✓ Ratings and reviews
- ✓ Conversion (views to downloads)
- ✓ Listing views
- ✓ Usage per day (isn't that awesome?)
- ✓ In-app purchases by age and market

There is also an extensive quality metric, which drills down to some serious statistical usage data that you never used to get with Windows apps:

- ✓ JavaScript exception rate
- ✓ Crash rate
- ✓ App unresponsive rate

These values are very important for improving the experience over the app's lifetime. Even though you might put out bug-free code, the WinRT library will change, and there will be environments that you didn't test on. It's good to be prepared, but even better to be informed. The Windows Store does a great job of informing you about the experiences users are having with your app.

Chapter 15

Going to the Cloud

In This Chapter

- ▶ Supporting your app with services
 - ▶ Using Azure to bring more value to your users
 - ▶ Making up for the lack of database support in WinRT
-

Throughout this book, I have mentioned that one of the strangest changes for .NET developers is the lack of database support in WinRT. It's not that Microsoft doesn't like databases (they make one, after all), but the network access required by a database connection is costly and insecure.

In Chapters 10 and 12, I cover a lot of options to work around the database issue. Local data is a possibility, as is using SQLite and the remote settings. The best option, however, is to wrap your database in a structured service layer and use WinJS.xhr to get what you need.

Unfortunately, for many of us, firing up a full-blown Windows Communication Services project with a SQL Server backend is just not within the realm of possibility. You could use Ruby on Rails and AppHarbor and MySQL, but those might not be in your technology set. It can be a difficult problem.

Fortunately, Windows Azure, the cloud platform from Microsoft, has come to the rescue with an inexpensive, preconfigured, easy-to-use-and-manage service platform. The Windows Azure Mobile Services product is the database for the Windows Store app, and I show you how to use it in this chapter.

A word of caution: Azure, as of this writing, is not even in beta yet. Things — especially the appearance of the interface — may change. Don't let this fool you. Please use online documentation in addition to this chapter to set up your initial service. If you discover differences from the product over the course of working with the product, please let me know at the contact form on the book's website. See this book's Introduction for more on the website.

Touring Azure

Windows Azure is a big, sophisticated Internet-hosting company run by Microsoft. It gives developers access to servers that are available on the Internet 24 hours a day, 7 days a week. You'll never see the server, you probably won't know where it is physically, and you don't have the ability to actually log in and change anything in the operating system itself.

And that's a good thing.

I spent the first 15 years of my professional career primarily working in the academic and entrepreneurial Internet space. I built websites, managed e-mail, and gave lots and lots of advice. I am, in fact, one of very few Certified Internet Business Strategists.

During that time, I expended more effort in building, maintaining, and protecting servers than I did coding. Keeping a server online — especially in today's dizzying security landscape — is incredibly demanding. Even if you have the resources and Internet connectivity, it drains the time you need to manage your apps. Who needs that worry?

Enter Azure. Azure gives developers a one-stop show for all things backend. Figure 15-1 shows the Azure dashboard, where you can access the complete range of Azure services, including database, virtual machines, worker processes, and more.

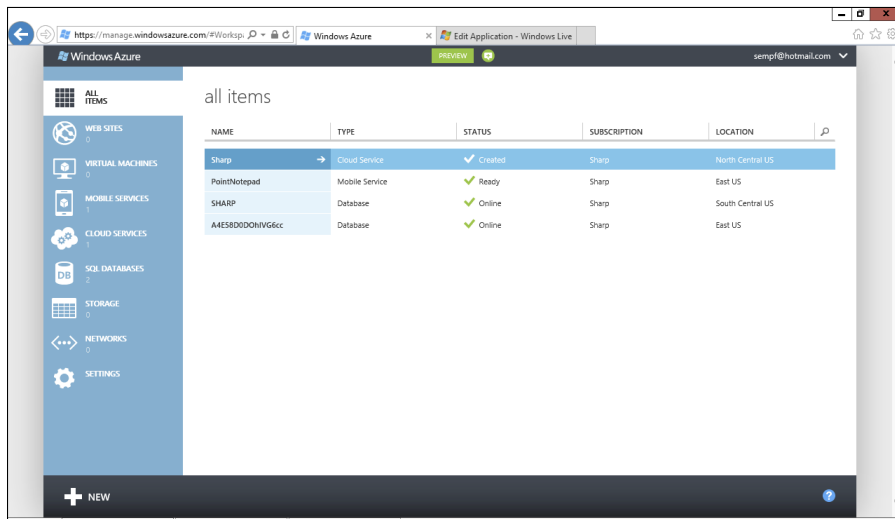


Figure 15-1:
The all-new
Windows
Azure
portal.

Azure is many things, but for Windows Store developers, it is a place to build your databases, access your authentication, and deploy your push notifications. Windows Azure Mobile Services, which should be out right about when this book comes out, will make it even easier.

Announcing Windows Azure Mobile Services

Windows Azure Mobile Services is a cloud-backed service model designed specifically for mobile application development. Although the APIs are all Window Store-based at the time of this writing, by the time you *read* this, it will support iOS, Android, and Windows Phone development as well. Mobile Services is going to be the one-stop shop for storing and accessing data in the cloud from mobile devices.

Mobile Services uses collections of preestablished patterns to solve really specific problems in mobile app development, all of which I have discussed in this book:

- ✓ Data storage
- ✓ Authentication
- ✓ Push notifications and more

This product is just entering beta as I write this, so more features are coming. To give you a taste of how this all works, I'm going to show you how to set up a backend database using Mobile Services.

Getting what you need to get started

To get started with Mobile Services, you need a couple of things.

- ✓ **An Azure account:** If you have an MSDN license, the Azure account comes with it. Just log in to manage.windowsazure.com with your MSDN credentials and go through the setup. If you don't have an MSDN license, many free trials are available. Just go to www.windowsazure.com and see what is there.
- ✓ **The Mobile Services SDK:** The latest SDK can be found at <http://sempf.me/MobileServicesSdk>. Get it, download it, and install it. You're ready to go.

Creating a new mobile service

You create a new mobile service at the Azure Management Portal. You can find the portal at manage.windowsazure.com:

1. Click on the Mobile Services tab to the left.

It should tell you that you have no Mobile Services currently and ask if you would you like to add one.

2. Click on the link to add a service and fill out the form.

I show mine all filled out in Figure 15-2.

- All the URLs have to be unique, so put something that makes sense in that field. Remember, it isn't user-visible.
- Select Create a New SQL Database in the Database field. I show you how to make the new database in the next section.
- Pick a region close to you.

3. In the Specify Database settings page, keep the default name for simplicity and set up some decent credentials with a quality password.

Make sure the Mobile Services and Database are both deployed to the same region in the Region Select list.

4. Click the check mark to finish.

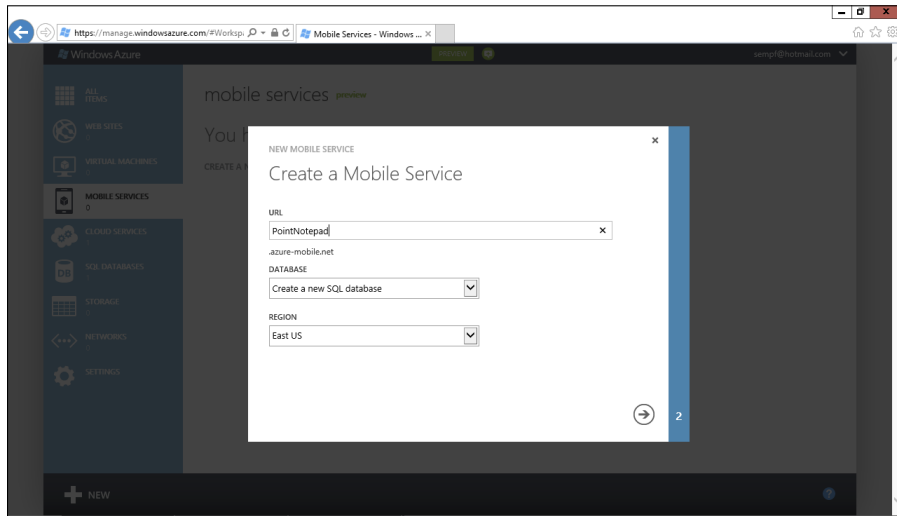


Figure 15-2:
Creating a
new Mobile
Service.

The system grinds for a little bit, but that's okay. At least it's on Microsoft's server and not yours, right?

After it's done, click the new line in the Mobile Services tab, and you are taken to the service landing page for your new service, shown in Figure 15-3. It doesn't do anything yet, but that's okay. It's here!

In one of the coolest features I've ever seen in any software product, you can click the Create a New Windows 8 Application link and download a brand-spanking-new template, prefilled with everything you need to connect your app to the service. It's neat. I talk about connecting an existing mobile app to the service in "Connecting your Windows Store app," later in this chapter.

Constructing a database

From the landing page shown in Figure 15-3, you can get to the database you configured. Do that now. Click the Data menu item, and Azure helpfully points out that you have no tables. Click Add a Table to continue.

The Create New Table dialog box lets you set a new table name and some permissions. Make a table called Notes, and keep the permissions set as they are by default.

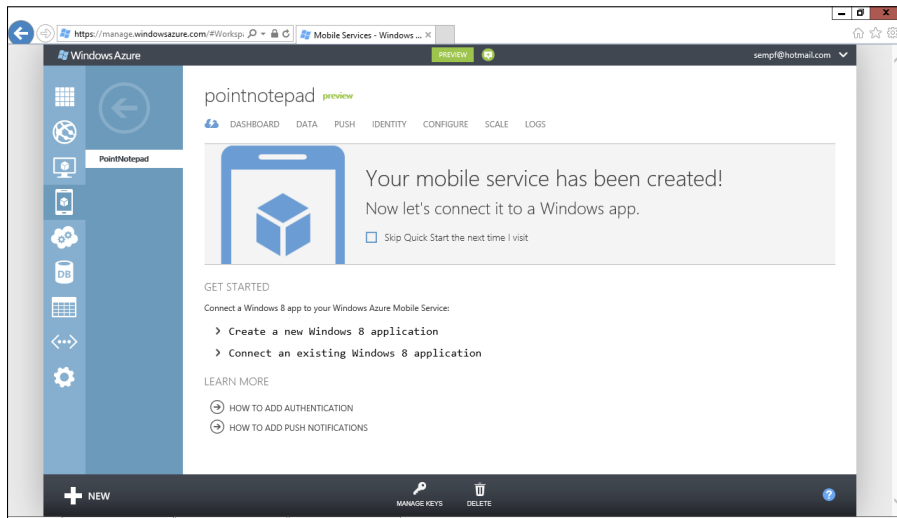


Figure 15-3:
Landing on
the service
landing
page.

When Azure is done creating your Notes table, you can click it and get to the Table landing page. Here you can browse the data, write a script against the data, manage the fields in the table, or change the table's permissions.



Columns or fields are created automatically in SQL Azure. It's based on what JavaScript Object Notation (JSON) you pass in from your mobile service. Don't go looking for an Add Column button — there isn't one.

Believe it or not, that's it. You're ready to run.

Connecting your Windows Store app

Your next step is telling Visual Studio what you've been up to. Make a quick app that stores notes in the cloud.

- ## 1. Open Visual Studio and create a new blank JavaScript Windows 8 app.

I called mine **CloudNotes**.

- 2. Right-click the References item and select Add Reference. Then check the Mobile Services SDK, as I did in Figure 15-4.**

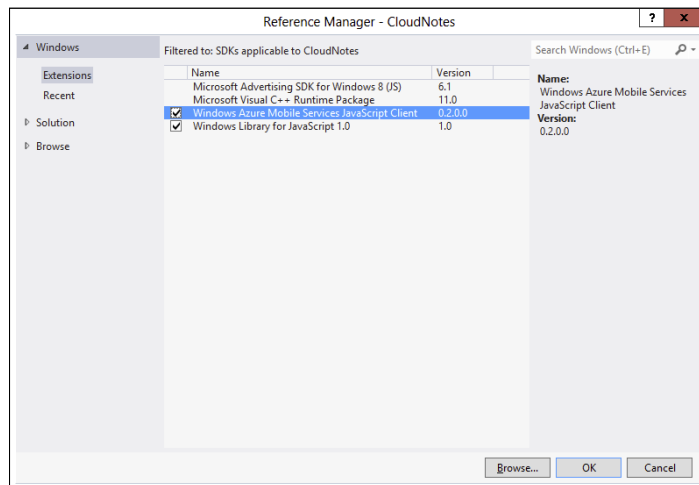


Figure 15-4:
Adding
the Mobile
Services
SDK
reference.

- 3. You may have to manually reference the service script in your HTML file.**

Here's what my default.html looks like.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CloudNotes</title>
  <!-- WinJS references -->
  <link href="//Microsoft.WinJS.1.0/css/ui-dark.css"
        rel="stylesheet" />
  <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
  <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>
  <script src="//MobileServicesJavaScriptClient/
    MobileServices.js"></script>
  <!-- CloudNotes references -->
  <link href="/css/default.css" rel="stylesheet" />
  <script src="/js/default.js"></script>
</head>
<body>
  <p>Content goes here</p>
</body>
</html>
```

4. In the Azure Management Portal, click the link in the middle of the page that says **Connect an Existing Windows Store App**.
5. Get the code from Step 2 and paste it into the default.js page right under the app and activation declarations.
6. Get the little block of code in Step 3 and paste it into the app.on activated event handler.

This is just to see how it works. It won't do anything on the screen, but that's okay for now.

7. Change the table name in `getTable` to `Notes` because that's what our Table is called.

My finished default.js file looks like this:

```
// For an introduction to the Blank template, see the
// following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232509
(function () {
  "use strict";

  WinJS.Binding.optimizeBindingReferences = true;

  var app = WinJS.Application;
  var activation = Windows.ApplicationModel.Activation;
```

```
var client = new Microsoft.WindowsAzure.  
    MobileServices.MobileServiceClient(  
        "https://pointnotepad.azure-mobile.net/",  
        "ThisIsMyPrivateKeyPleaseMakeSureYouDontLoseYours"  
    );  
app.onactivated = function (args) {  
    var item = { text: "Awesome item" };  
    client.getTable("Notes").insert(item);  
};  
app.start();  
})();
```



The third parameter of the Mobile Services call is the private key for your service. It is a password. Do not lose it, e-mail it, paste it into a forum post, or leave it on PasteBin. If you do, someone can steal your data. Bad news, that. What's more, make sure you use YOUR key. It is in the Azure portal.

All the code is going to do is put a little data in the clouds table so you can go look at it. Well, what are you waiting for? Check out the next section and give it a try!

Running your app with a service

This is the tough part.

Press F5.

Not much is going to happen on the screen, but head back to the Azure Management Portal, and click on the Data tab. Then click on the Notes table. You should see something like Figure 15-5. The data was added, with a new column and everything. Simply amazing.

So what's this mean for you? With absolutely zero effort, you've added a comprehensive database backend to your app that requires absolutely zero backend database design. Sit and think about that for a minute.

Done? Good.

That means that if you have an app in the Store that is storing stuff locally, you can change it to store stuff in Azure with maybe a total of 10 to 30 lines of code, depending on the complexity of the app.

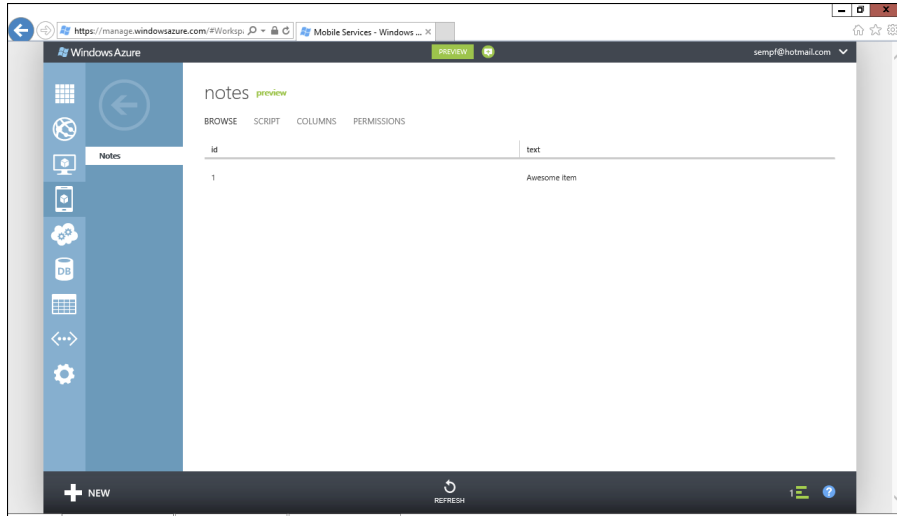


Figure 15-5:
Viewing our
new data.

Taking it to the next level

So you got some data in the Mobile Services. You need to get some data out of there, too.

I started by adding a few more records to the Notes table. I did it the cheap way, by rerunning the demo app with different values in the Text field. You can work on the Scripting feature of Mobile services if you wish. It is pretty cool.

Then I tossed a ListView in the default.html page. I don't set the itemData Source here; I'll set it in the code a little later.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CloudNotes</title>
  <!-- WinJS references -->
  <link href="//Microsoft.WinJS.1.0/css/ui-dark.css"
        rel="stylesheet" />
  <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
  <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>
  <script src="/MobileServicesJavaScriptClient/
        MobileServices.js"></script>

  <!-- CloudNotes references -->
```

```

<link href="/css/default.css" rel="stylesheet" />
<script src="/js/default.js"></script>
  <script type="text/javascript">WinJS.
    UI.processAll();</script>
  <script src="sampleData.js" type="text/
    javascript"></script>
</head>
<body>
  <div class="sampleListViewItemTemplate" data-
    win-control="WinJS.Binding.Template">
    <div>

    <div data-win-bind="innerText:text">      </div>
                                          </div>      </div>
    <div data-win-control="WinJS.
      UI.ListView" id="SampleDataList" data-
      win-options="{itemTemplate:select('.
        sampleListViewItemTemplate')}"></div>
</body>
</html>

```

In the backend JavaScript, I am going to use `getTable.read()`, which is an async function. This gets the JSON back from my table and allows me to make a binding list.

```

(function () {
  "use strict";

  WinJS.Binding.optimizeBindingReferences = true;

  var app = WinJS.Application;
  var activation = Windows.ApplicationModel.Activation;
  var client = new Microsoft.WindowsAzure.
    MobileServices.MobileServiceClient(
      "https://pointnotepad.azure-mobile.net/",
      "UQumkjaBjjGlgcmvDrrcLKKZtzcudr53");
  var bindingData;
  app.onactivated = function (args) {
    client.getTable("Notes").read().done(function
      (result) {
        bindingData = new WinJS.Binding.List(result);
        document.getElementById("SampleDataList").
          itemDataSource = bindingData;
      });
  };
  app.start();
})();

```

That's all it takes to get data out and make it binding quality.

By the time you read this, Mobile Services will likely be able to do more than it does now. In the meantime, I should chat briefly about push notifications and authentication.

Pushing Data to the User

You can leverage Windows push notifications with Mobile Services by giving it two values to tie the app and the login service together — your Windows push notifications client secret and your package SID. With these, Mobile Services can broker push notifications for your app.

To get the client secret from Windows Live Services, go to `https://manage.dev.live.com`, and either select your application (it might be there if you have been through other chapters) or add it. Then you can click on API settings to get or create the client ID and secret. You can see what it looks like in Figure 15-6.

Then you can head to `manage.windowsazure.com` link things up. Take the client secret and the package SID from the Live Connect Developer Center and enter them into the fields in the Windows Application Credentials screen.

From there, you can use the `Windows.Networking.PushNotifications` class from WinRT to get things rolling. Check out the sample on the website or review the latest documentation for Azure Mobile Services to get the details.

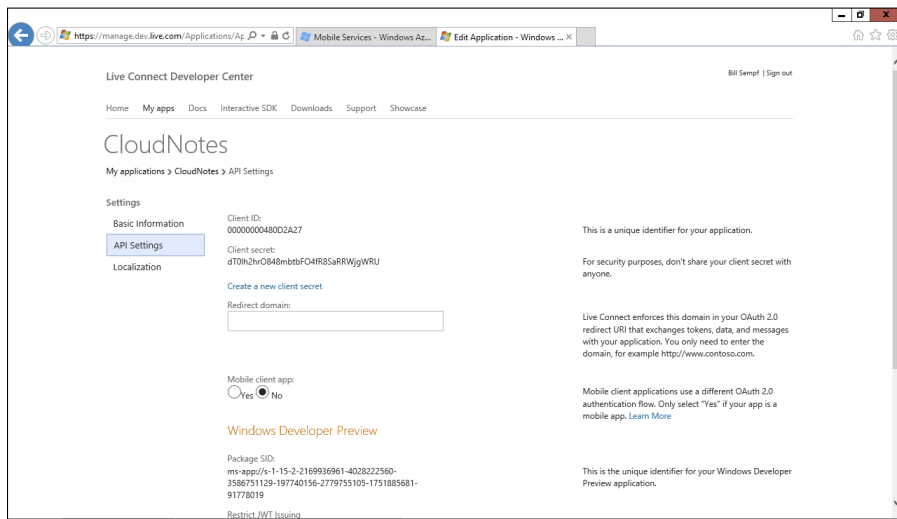


Figure 15-6:
Getting
the client
secret from
Live.com.

Getting Authenticated

You can also use Mobile Services to integrate with the authentication features in Windows Live. You know the ID that you use to log into MSDN, the Xbox, and everything else? You can have users log into your app with it as well.

As of this writing, the Microsoft.Live namespace isn't in the SDK, but check the code on the book's website to see some glorious sample code. (See this book's Introduction for more on the website.) I'd give you a little C# and XAML, but then title of the book would be wrong.

Chapter 16

Making Money with Your App

In This Chapter

- ▶ Working with the Windows Store
 - ▶ Setting up in-app purchasing
 - ▶ Showing your users ads
-

I realize that not everyone reading this is looking to make crazy money with their app. Some of you are writing free apps, just for fun, or experience, or a résumé item, or because you are passionate about something. Some of you are writing apps for enterprise deployment and are (I hope) being paid a hefty salary to write these apps.

The rest of us, however, are looking to make a few bucks in what is sure to be the biggest developer opportunity in the history of computing. To be sure, charging a few bucks for your app is the easiest way to go about it. I cover in Chapter 14 how to set a price for your app and set up the deposits from the Store.

There are other ways to make money, though. The first is just like so many websites do: show ads. Although it's not strictly part of the WinRT API, Microsoft has provided its own advertising network, and you can implement other networks. Even the vaunted Google AdWords runs in Windows 8 apps.

One other less well-known way to make money is through in-app purchasing. This is a mechanism in which your users can choose to unlock specific functionality by spending real money. For instance, say you have a greeting card app. You might provide extra card templates to users for money, but give the basic app itself away.

Some of this interacts with the Store, and some of it doesn't. Keeping the income organized and sorting out what you have to watch for is a skill in itself. I start with advertising income.

Adding Ads

Advertising is the bane of some people's existence, but it also pays for much of the content on the World Wide Web. With the declines in viewership on television and readership in magazines and newspapers, advertisers are vying for eyes.

You don't have to put ads on your app. There are other ways to make money. That being said, it is possible to put ads in your app without alienating your users. It's just a matter of design, like almost everything else in the Windows 8 world.

Designing for advertising

Much of the reason that people dislike ads is that they're often in poor taste. It is ostentatious, or badly placed, as shown in Figure 16-1, or something the viewer is totally disinterested in.

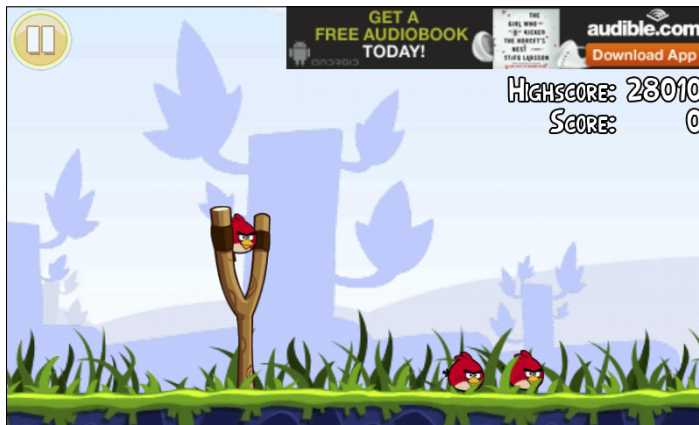


Figure 16-1:
Hard to play
when you
can't see.

You can prevent that kind of user experience by simply doing a little design work. Make sure your advertising isn't covering parts of the app. (That happened in Angry Birds when they were ad-supported. I was so disappointed.) Make sure there won't be flashing, moving, or wiggling ads. Try to get something that will target your market. You don't want your fishing app to be polluted with ads for Barbies.

Advertisers want your audience

If you have written an app that pings servers and reports on results (like the new Ping app that just showed up in the store), you have an audience that advertisers want. That's also true if you have an app for runners, astronomers, or movie buffs. Turns out, if you know who your audience is, companies can target ads to them. Truthfully, everyone is happier that way.

This is why Google AdWords is so popular. If you are searching for SQL Server, for example, you see ads for SQL Server tools, hosting, and books. If you are searching for movies and have made your location available, you might even see local listings in the ads.

There are privacy concerns, though. If advertisers store this information about users in order to “follow” them around the Internet (including via connected apps), people start to get testy. I know I do. The two advertising APIs that I cover here do a good job of *not* doing that: MSN and Google.

Figuring out how you're doing

All kinds of metrics go into figuring out if an ad campaign is working. Some of the terms you will see in the code and descriptions in this chapter include

- ✓ **Ad format:** Type of ad (for example, image, video, rich media) that an advertiser would buy.
- ✓ **Ad size:** Size of the ad in pixels (for example, 300X250).
- ✓ **Impression:** The number of views of an ad unit.
- ✓ **UUs:** Unique users who see the ad impressions in a given time period.
- ✓ **Ad inventory:** Volume or number of advertising impressions available on your app.
- ✓ **Yield:** Revenue generated from the ad inventory; often measured as revenue per impression.

These concepts directly relate to the metrics used by advertisers to determine how the ad campaign is doing. Some of these metrics include

- ✓ **CPM:** Cost per thousand impressions; the advertiser pays when a user sees an ad.
- ✓ **CPC:** Cost per click; the advertiser pays when a user clicks on the ad.
- ✓ **CPA:** Cost per action; the advertiser pays when a user completes a specific action.

- ✓ **CTR:** Click-through rate; it's the number of clicks on the ad divided by the number of delivered impressions of that ad.
- ✓ **CVR:** Ad conversion, typically represented as a percentage calculated by the number of ad conversions, divided by the number of clicks, multiplied by 100.

All of these come together to answer the question “Am I making money with this?” I couldn’t write a better answer than the one found in the Microsoft Advertising Ads in Apps 101 whitepaper, so I am just going to reproduce it here:

“Your ad revenue or yield is driven by how much advertisers pay per ad impression, and how many impressions are bought by advertisers. Knowledge of your audience demographic and usage behavior helps advertisers provide relevant ads and drives up the value of your app’s ad impressions. Advertisers pay a certain cost per impression to reach this audience. This cost is known as CPM or *cost per thousand*. Advertisers have the opportunity to buy your advertisements at this rate. The goal of the developer is to carefully monitor advertising campaigns to drive the best yield. Adjusting ad type and size may help drive up the ad yield. Providing accurate audience demographic information and using engaging ad formats may also help.”

The Windows 8 Ads SDK

Enough theory. Time to get an ad rolling.

When Microsoft announced Windows 8, they also quietly introduced Microsoft Advertising, which is effectively a copy of Google AdWords. You can’t blame them — AdWords is unquestionably the best and most popular ad network out there. Lots of people use it to make money every day on their Android apps and websites.

I am not going to put it past Microsoft to be a success as the second one out of the gate. It has made a career of watching others innovate and perfecting the model for its own use. I’m going to give Microsoft Advertising a try.

Getting started

Here are the three steps to getting started.

1. **Register with `pubcenter.microsoft.com`. You can use your Live ID to log in, but you need to create a pubCenter account after you authenticate — don’t let that throw you. That’s how it’s supposed to work.**

1. Put in your personal information.
2. Add an Ad Unit. This is where you set up the ad you want for your app. You can have more than one of these. In Figure 16-2, I added a right-side tower to POINTtodo. This setup gives me the registration code that I need to add the advertisement into my code.

3 Register your first application and create an ad unit (optional)
 To enable ads, you must register an application and create an ad unit that you will use in your application code. You can skip this step now and register the application and create the ad unit later in pubCenter.

Application information ?

Application name:

Device type:

Ad unit information ?

Ad unit name:

Ad unit size:



Note: The ad unit size that you have selected is shown in a sample location of a device. You define the location of your ad unit in your application code.

Figure 16-2:
Making a
new ad unit.

3. Pick your categories for targeted advertising. Instead of using search terms, this tool allows you to pick the category of product you want advertised.
 4. Note your ApplicationId and your App Unit ID at the bottom of the page.
- 2. Download and install the Microsoft Ad SDK. You can download the SDK at <http://advertising.microsoft.com/windows/advertising/developer>.**
 - 3. Add the Ad SDK to your project.**

Working with the SDK is covered in the next section.

Building in ads

To build ads into your project, open your project in Visual Studio. I am starting with a blank project.

- 1. Make sure the Internet Client capability is declared. Double-click the package.appxmanifest and make sure the Internet (Client) check box is selected.**
- 2. Right-click the References folder and select Add Reference.**
- 3. In the Add References dialog box, select Microsoft Advertising SDK for Windows 8. Click OK.**

4. In the head of your default.html file, make a reference to the SDK script.

```
<!-- Microsoft Advertising required references -->  
<script src="/MSAdvertisingJS/ads/ad.js" ></script>
```

5. In the body tag of the page you want the ad to appear on, add a div tag with the AdControl tag.

```
<div id="myAd" style="position: absolute; top: 53px;  
    left: 0px; width: 160px; height: 600px;  
    z-index: 1"  
    data-win-control="MicrosoftNSJS.Advertising.  
        AdControl"  
    data-win-options="{applicationId: 'test_client',  
        adUnitId: 'Image_160x600'}">  
</div>
```

6. Run the app. You should see a test ad like Figure 16-3.

(Is anyone else ready for Gears of War 3? I sure am!)



Figure 16-3:
Adding a
test ad to
a blank
project.

For the test environment, that's it. You can place and style the `div` as you see necessary for your app. Note that there is a variety of test layouts — the same layouts as defined in the `pubCenter` site. These include

- ✓ `Image_160x600`
- ✓ `Image_250x250`
- ✓ `Image_300x250`
- ✓ `Image_300x600`
- ✓ `Image_320x480`
- ✓ `Image_320x50`
- ✓ `Image_728x90`
- ✓ `ImageText_160x120`
- ✓ `ImageText_160x160`
- ✓ `ImageText_250x250`
- ✓ `ImageText_320x50`
- ✓ `Text_300x250`
- ✓ `Text_320x250`
- ✓ `Text_320x50`

Don't forget to update the height and width of the `div`.

The next thing, when you are done testing the layout, is to go live.

Going live

I looked all through the documentation, expecting the going live process to be ginormous. Usually things like that are, right? You usually have to change your code all around and whatnot.

Couldn't be further from that, in this case.

Remember back in the section, "Getting started," when I suggested that you should take note of your Application ID and Ad Unit ID? This is where you need them. Just replace the test values in the `data-win-options`:

```
<div id="myAd" style="position: absolute; top: 53px; left:
    0px; width: 160px; height: 600px; z-index: 1"
    data-win-control="MicrosoftNSJS.Advertising.
    AdControl"
    data-win-options="{applicationId: '8620f1f2-
    d420-49fe-b8bb-5d3aa386b4d1', adUnitId:
    '10045398'}">
</div>
```

That's it. Run the app. I'd show you a pretty picture, but the ads aren't live yet. I got an error, which reminds me to handle any potential errors in the backend JavaScript. Error codes include the following:

- ✓ **Unknown (0):** An exception has been thrown but the state or cause cannot be determined.
- ✓ **NoAdAvailable (1):** No ad is available.
- ✓ **NetworkConnectionFailure (2):** A connection to the network could not be established.
- ✓ **ClientConfiguration (3):** The ad control has not been configured properly.
- ✓ **ServerSideError (4):** The ad server network has reported that a server error has occurred.
- ✓ **InvalidServerResponse (5):** The server response contained invalid data.
- ✓ **Other (6):** The error cannot be classified as one of the known errors.

I recommend just handling all of these by hiding the `div` with the ad control in it.

Using other ad networks

If you're like me, you've been using Google AdSense for your websites for several years now. Now, my blog doesn't get a ton of hits, and I don't like to monetize the sites for these books, but I get \$20 a month or so, enough to cover hosting and images that I license and whatnot.

Google AdSense works just fine in Windows 8. Because it's external content, you need to add exceptions in the `package.appxmanifest`, but that's a small thing. I do that in Step 3.



The rules for content URIs is changing all the time. You might need to make small adjustments based on the new rules by the time this book gets to you. To add AdSense content, try these steps:

1. **Log into to your AdSense account and get your code for the ads. If you don't have AdSense set up, do so at www.google.com/adsense/v3/app.**
2. **Paste the code into the `body` tag of the HTML.**

```
<script type="text/javascript"><!--
    google_ad_client = "ca-pub-415291810545798";
    google_ad_slot = "54345768975";
    google_ad_width = 160;
    google_ad_height = 90;
    //-->
</script>
<script type="text/javascript"
src="http://pagead2.googlesyndication.com/pagead/show_ads.
js">
</script>
```

3. In the package.appxmanifest, click the Content URI tab and add the URL for the AdSense distribution point.

Mine looks like Figure 16-4.

4. Add an iframe to the page where you want the ad to appear, and reference the code you just added to the project.

Bang, you're done. Because you're working in HTML5, you get to just use the tools as they are designed. If you do have a problem with the content URI, check the JavaScript console. In Figure 16-5, I saw the URI in the JavaScript console that was causing my problem (see those APPHOST9601 errors?) and added it to the manifest from there.

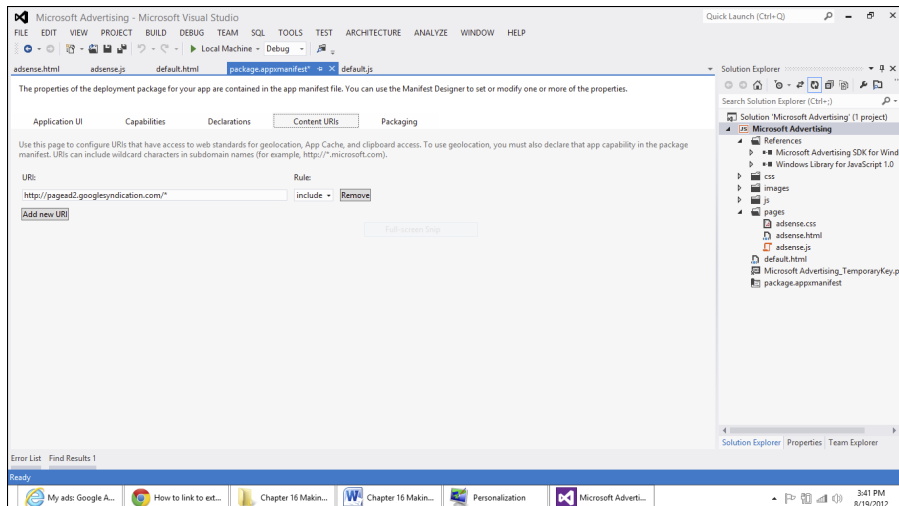


Figure 16-4:
Adding the
content URI
for Google
AdSense.

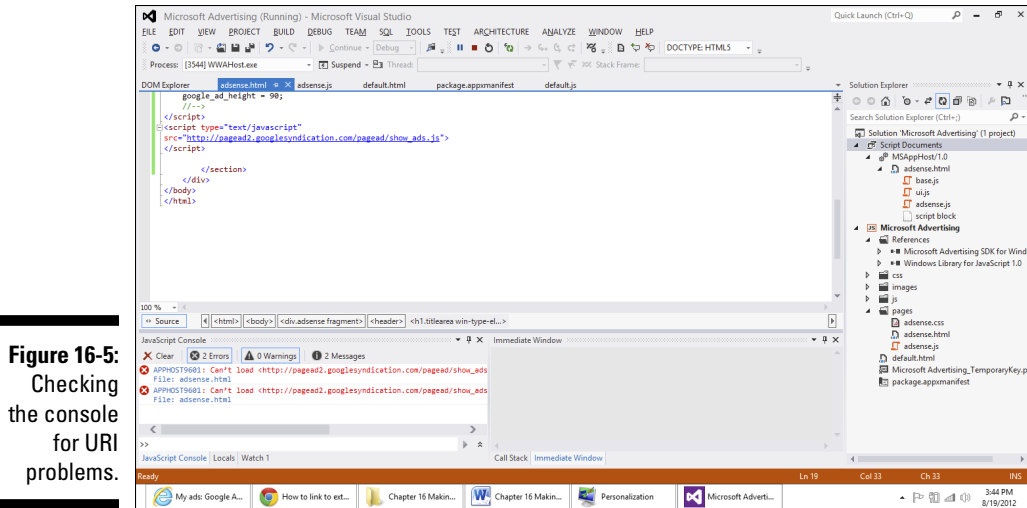


Figure 16-5:
Checking
the console
for URI
problems.

Handling In-App Purchasing

Selling ads isn't the only way to make money. Getting the user hooked with an always-free app but charging for More of the Same™ is another sound strategy, and generally alienates fewer people. Even users who don't like ads in apps understand that the programmer needs to get paid for his work, and paying for the next expansion is a way to live and let live while rewarding consumers who are willing to shell out a little coin for features.

Take Pinball FX2, for example. You get the whole game for free, and can play all you want. It's a great game too. If you've never tried it, do so. The game has no ads and doesn't time out with a free trial or anything.

If you like the game play, you might want to try another table. They are three dollars U.S. each. Too rich for your blood? Not a problem. Play the Mars table all you want. If you want another table top, just head to Figure 16-6 and get one. It's their pleasure to serve you.

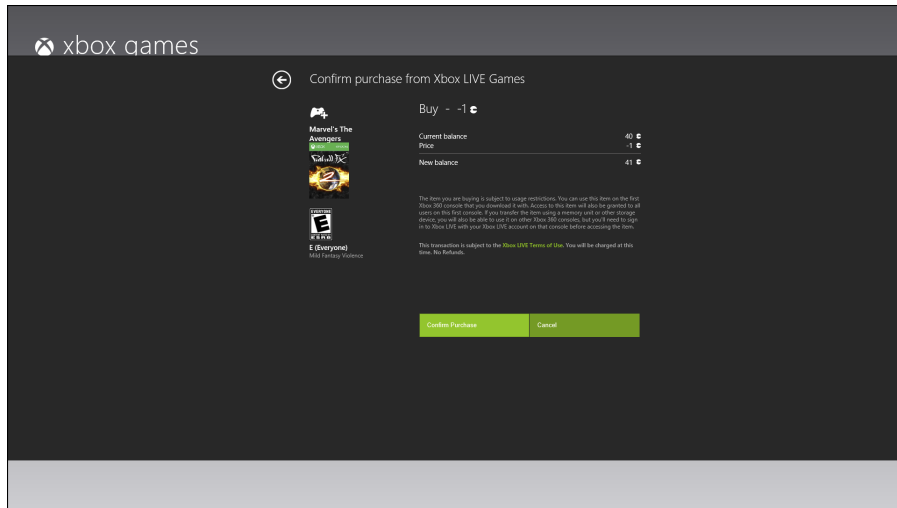


Figure 16-6:
Getting a
new table in
Pinball FX2.

The Windows Store handles the download, transaction, and everything. You just need to give the user somewhere to go to order it. In Pinball, unpurchased content shows up as gray; purchased content is in color.

Planning for app expansion

In your app, you need to decide what content will be for pay and what will be free. This is not a decision you should make lightly. If you over-promise and under-deliver, people trash you in your app's ratings. If you give too much away, no one buys your expansion content.

Here are some examples of lines you might want to consider drawing:

- ✓ If you have a game, consider giving away the first 20 percent of the levels and charging for the rest. Ten levels? Give away two and sell the rest for a dollar a pop.
- ✓ For a content management app, considering charging for advanced editing capabilities. Users can edit text and share around all they want for free, but they pay for image-editing capability.
- ✓ Online interaction is a good feature to charge for. For instance, a poker app lets you play against the computer all you want, but if you want to play another human online, you have to pay for the privilege.
- ✓ Have a tax app? The 1040 forms can be free, but other forms come in packs that you have to buy: one for business, one for investments, and so on.

- ✓ Having the capability for paying for more of something is always a good choice. If you have a record-keeping app, offering 10 records for free and 100 for \$2 is an option.
- ✓ Media tools have given in-app purchasing. You just pay for the media, and the tool is free.
- ✓ If you are providing a media mashup app — say, a sports app — you might want to consider buying into a data source that is better but has a subscription, and then charging users for the enhanced data pack.
- ✓ Fantasy football apps give access to statistical analysis tools for a fee, but the games are free to play.
- ✓ If you have a shopping app, you can offer notifications as a paid feature. Users tell the app what they're looking for, and get a notification when the price changes.

The options are pretty much endless. In fact, you could make the argument that one should never charge or accept advertising — that in-app purchasing is the only way to go because it's fair to everyone. I can't find much wrong with that argument.

Getting your app ready for purchasing content

The way you configure your app internally is up to you. In my sample app, I have an app bar with the MOTS button (for More of the Same) that gives users access to the purchasable feature. If the user clicks it, it offers to let them buy it. If they've bought it, it takes them to the feature.

1. Add an app bar with the MOTS button that leads to the new feature:

```
<body>
  <p>Content goes here</p>
  <div data-win-control="WinJS.UI.AppBar">
    <button data-win-control="WinJS.
      UI.AppBarCommand" data-win-
        options="{icon:'back', id:'', label:'example',
          onclick:null, section:'global',
          type:'button'}"></button>
    <button id='motsButton' data-
      win-control="WinJS.UI.AppBarCommand"
      data-win-options="{label:'MOTS',
        type:'button'}"></button>
  </div>
</body>
```

2. Get the license information for the app in the activation code.

You'll need it to determine if they have bought the feature as you activate.

```
currentApp = Windows.ApplicationModel.Store.  
    CurrentAppSimulator;  
licenseInformation = currentApp.licenseInformation;  
<tip>
```

Of course, if you were in production, you would use the `CurrentApp` object.

```
currentApp = Windows.ApplicationModel.Store.  
    CurrentAppSimulator;
```

3. Check to see if the MOTS feature is purchased. If it is, send them to the promised land when they click the button. If not, send them to buy it.

```
        if (licenseInformation.  
productLicenses.lookup("motsFeature").  
isActive) {  
            document.getElementById('MOTS').  
onclick = AccessTheNewFeatureMethod();  
        }  
        else {  
            document.getElementById('MOTS').  
onclick = SendThemToTheStore();  
        }
```

4. Finally, provide the `SendTheToTheStore` function. This uses the Purchase features of Windows 8 to allow the user to purchase the keyed feature. Here's what the code base looks like now:

```
(function () {  
    "use strict";  
  
    WinJS.Binding.optimizeBindingReferences = true;  
  
    var app = WinJS.Application;  
    var activation = Windows.ApplicationModel.  
        Activation;  
  
    app.onactivated = function (args) {  
        if (args.detail.kind === activation.  
ActivationKind.launch) {  
            if (args.detail.previousExecutionState  
!= activation.ApplicationExecutionState.  
terminated) {  
                currentApp = Windows.ApplicationModel.  
Store.CurrentAppSimulator;  
                licenseInformation = currentApp.  
licenseInformation;  
            }  
        }  
    }  
}
```

```
        if (licenseInformation.  
productLicenses.lookup("motsFeature").  
isActive) {  
            document.getElementById('MOTS').  
onclick = AccessTheNewFeatureMethod();  
        }  
        else {  
            document.getElementById('MOTS').  
onclick = SendThemToTheStore();  
        }  
    }  
    args.setPromise(WinJS.UI.processAll());  
}  
};  
  
function SendThemToTheStore() {  
    currentApp.requestProductPurchaseAsync  
("motsFeature").then(  
        function () {  
            // they bought it, turn it on  
            document.getElementById('MOTS').  
onclick = AccessTheNewFeatureMethod();  
        },  
        function () {  
            // They didn't buy it, feel free to  
            play the sad trombone sound  
        })  
    );  
}  
app.start();  
})();
```

Adding the purchase UI

Don't forget that the UI for the feature has to be in your app. The license is in the Store, but the feature has to already be there.

In this case, whatever the MOTS button does, the code has to be in the project. If it is a link to another page in the app, the page has to already be in the project.

To protect against accidental escalation of privilege exploits (users getting access to content they haven't paid for), I would recommend checking the licensing before the page loads.

It doesn't have to be a big deal, but you never know what potential exploits will be discovered.

Setting up the Store for your app

Back in Chapter 14, I mentioned the Advanced Features section of the app setup process. One of the advanced bits was the in-app options, and this is where they come in.

In order to sell a feature, you need to add it as a product, attached to your app. This gives it a little separate license that you can check in the code of the main app using the `licenseInformation` object. Windows 8 handles the conversation with the Store under the covers.

To set up the Store for an in-app purchase, go to <https://appdev.microsoft.com/StorePortals> in your browser, and log in with the credentials that you use for your Microsoft account. Select the Edit item in the app you want to change, and then click Advanced Features.

At the bottom of the page, you can add in-app offers. Select an App ID, which should be descriptive of the exact thing you are offering, rather than a generic entity label. For example,

- ✓ AtlantaTrackLevel rather than Level4
- ✓ BusinessTaxFormPackage rather than NewForms
- ✓ DisableAdsPackage rather than Premium
- ✓ 20ExtraRecordsOfStorage rather than BonusRecords

This is for two reasons. One, it keeps you from getting confused; two, this is reviewed by a human as a condition for being allowed in the Windows Store, and you need to give them a clue about what things in your app do. I added a Disable Ads feature to POINTtodo, as you can see in Figure 16-7.

You also need to select the price tier — they are the same tiers as app pricing. You also get the option to select a timeframe and give people the ability to get some feature for a limited amount of time. For instance, I could say “Buy three months of ad-free use for only \$1.49!” That’s a pretty nice offer.

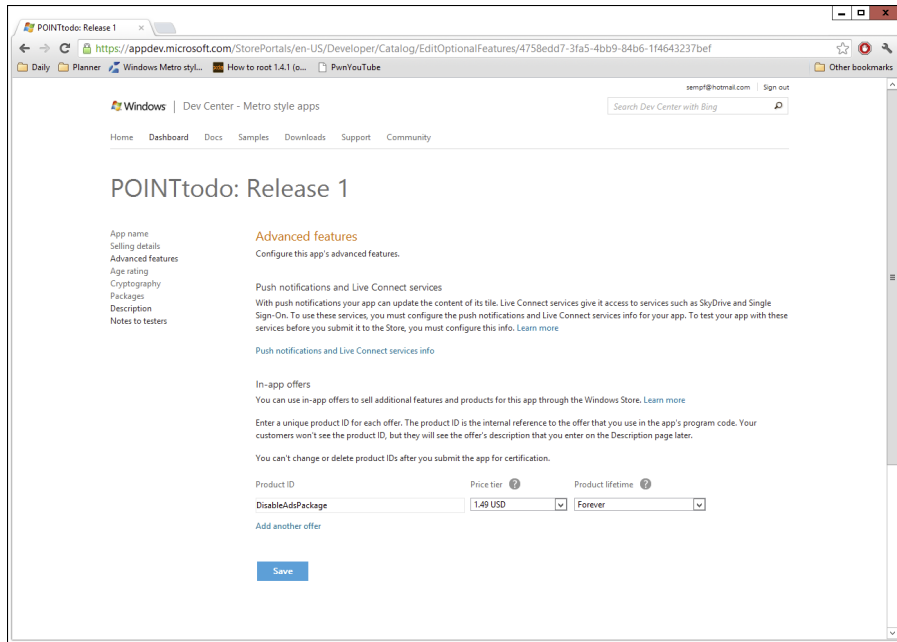


Figure 16-7:
Adding an
in-app offer.

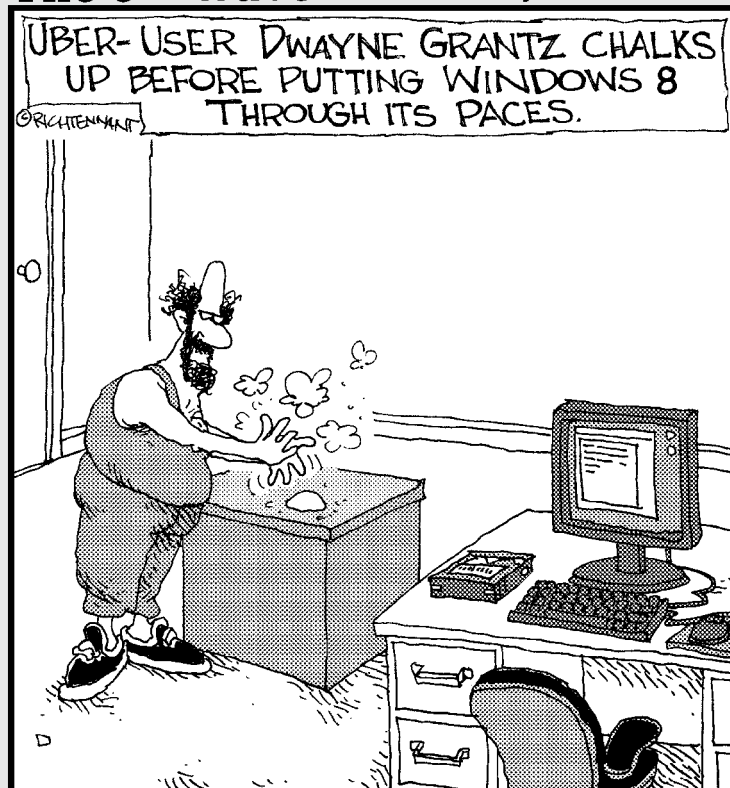
After setting up your Store account is done, you use `ProductId` in your code to see whether the offer is licensed. Then you can get your offer in the hands of customers!

Part V

The Part of Tens

The 5th Wave

By Rich Tennant



W ***In this part . . .***

hen your brain is all full of programming goodness, but lacking good app ideas, you can always turn to the Part of Tens! I pass on my Ten Greatest App Ideas Ever, and then give you a fine list of the resources I used to write my apps and this book.

Chapter 17

Ten App Ideas

In This Chapter

- ▶ Getting out of the gate fast
 - ▶ Grabbing users with neat content
 - ▶ Making use of services
-

If you've read this book and worked through the examples, you're probably ready to get started. But where to start? Never fear: I am here with ten app ideas that are sure to get the developer juices flowing.

Code away!

Publishing Social Content

Social networking has a special place in the heart of a lot of people, and for a good reason. It is possible to keep in touch with people like never before using the Internet.

I've heard more than a few jokes about social networking sites coming and going over time (MySpace, anyone?), but the market has some stability right now. If you are a new consumer of content, however, wouldn't it be awesome to post something to more than one site at a time, in order to reach a lot of different people?

I think it would. Here's your challenge:

Make an app that mashes up posting to multiple social networking sites.

Use the APIs provided by the social networking sites and build one form for content creation. Then when users click Post, the content goes to Facebook, their blogs, Twitter, LinkedIn, or whatever.

Integrating Planning Tools

Basic mail, contact, and calendar app suites are built into Windows 8, but they're not well integrated. You can't easily say why you are meeting for given meetings, there is no task list, and you have to change apps constantly to get something done.

A new app that gives the user one place to go to do planning could be awesome. Here is the challenge:

Use the Windows Live contacts and calendars API to build a comprehensive, unified planner app.

Calendar, tasks, e-mail, contacts all together — think Outlook without the chrome and the complexity. Just build a simple planner.

Finding Your Way Around

There is a Maps app that uses Bing Maps as part of the Windows 8 default installation. It's actually pretty cool — it uses the GPS or the wireless signal to get your position and show you what is going on around you.

You can search for locations and get directions, and even show traffic. As a mapping app, it's pretty slick. It falls down in one particular area, though. You can't plan a trip.

You know what I mean. All you want to do is run your errands, stop for lunch somewhere in there, and end up back at home. Can't something plan that route for you, recommend lunch locations on the route, and give you the directions? That's your challenge:

Let users plan their trips with the Bing mapping API.

An app that solves a particular problem for the user always does well in the store. Just follow the design guidelines, and it will do very well.

Playing the Day Away

This book isn't about game development. Frankly, most of the games for the Windows 8 platform will be built by C++ programmers using existing physics engines, and is way, way beyond the scope of this book.

That's just *most* of the games, though. Arguably the most popular game in the Windows 8 library in the release preview was Cut the Rope, which is an HTML5 game. I work with one of the guys who converted it to WinJS, and learned that some kinds of games are admirably suited for HTML5 implementations. So here is the challenge:

Write a game!

No, I don't have any ideas. For my game (Proton), I actually found a game idea from a brilliant game developer (Andy Looney) and asked him if I could make a game out of it. He said yes, and the rest was history.

Reading What There Is to Read

A blog reader is a pretty simplistic idea for Windows 8, and is in fact the initial "Hello, World" that Chris Sells originally wrote for the Windows 8 developer preview back in 2011.

Something that is specifically designed to read a certain kind of syndicated content, based on the kind of content that it provides, is a pretty neat idea. Maybe a movie review blog reader with a built-in video player, or a writer's ideas blog with a built-in notebook — something like that. The challenge?

Build a unique app that focuses on consuming RSS or Atom content.

Organizing Your Stuff

I don't know about you, but I miss the focus on the file system. I don't feel like I am going to be able to find anything when this operating system becomes my primary system. So I challenge you to

Write a way to file away digital assets on SkyDrive.

Actually, it doesn't even have to be SkyDrive. Even a Dropbox or Wuala client would be nice. But something that lets me store away nontraditional assets like log files, e-books, and notes in a storage system that makes sense.

I would buy an app like that, without question.

Painting with Photos

Instagram made burgeoning photographers the world over take perfectly good photographs and turn them into grainy works of “art” to share with their friends and family. I vote that you take it a step further:

Build an app that uses filters to make photos instantly look like paintings.

A number of neat bits of code out there can help you mangle digital photos. On SourceForge, I easily found four projects that take a photo and make it look like paint on canvas.

Mash that up with the great device management tools in Chapter 13 and you are good to go.

Managing Your Business

Windows 8 isn’t just about putting apps on the store and making crazy money. Sometimes you have to make the boss happy, too.

Corporate managers have a lot of information to manage. I figure that some of you reading this book (if not all of you) have worked at one time as a bug business developer, and you know that reporting is a big deal.

Lists of numbers aren’t as useful as one would think. What really helps people is living, memorable pictures of the data. Windows Store style is uniquely positioned to provide exactly that. As such, your challenge is

Create a dashboard of corporate information.

This app is admittedly totally different from everything else here. It assumes access to data, and the time and availability to work on this kind of project. But for the 10 percent of you out there with that kind of environment, think about it!

Keeping Healthy

At the release preview, the store had a whole Medical category with nothing in it.

Want to change that? A few of you, at least, should try this challenge:

Use Roaming Storage to keep diet and exercise logs that sync between tablet, PC, and phone.

Build an app that lets the user enter health data and track it. I heard an awesome talk by one Tottenkoph about Bio Mining (search the Internet for it with Bing), and the awesome things you can learn about yourself, your diet, and your health by carefully tracking information.

Building an app to do this would actually be fairly trivial. It's just data in, data out. Give it a try!

Going Shopping

The world of shopping has changed. Amazon, eBay, and the other online commerce giants have changed the way we buy everything from model airplanes to diapers. My toothpaste is delivered monthly, for free, for 50 percent of what it costs at the grocery store.

Whether you're shopping for TVs or mac and cheese, online comparison is important. Because of that, I challenge you to

Use shopping APIs to compare prices.

If you want to get fancy, use the sensor APIs to get the bar code from the scanner and look things up that way. It would be a popular app!

Chapter 18

Ten Places to Find Samples and Info

In This Chapter

- ▶ Discovering more than I can fit in this book
 - ▶ Going farther with templates
 - ▶ Finding sample code
-

This book can only go so far. It has only so many pages, and new information is coming out all the time. Also, this is a beginner book — it is only designed to give you an overview of the topics. Eventually, to further your knowledge and skills, you'll need to go find advanced topics that are specific to your needs.

Fortunately, a lot of information is out there already. Windows 8 is new, but Microsoft and its partners have done an awesome job getting the good word out there.

That being said, this list probably has an expiration date. Some of the links here probably won't still be around when the book gets into your hands. I keep a canonical list of resources on the books' companion website. For more information on the book's website, please see the Introduction.

dev.windows.com

When I was at a Train the Trainer course for Windows 8, one of the developer evangelists said something very compelling:

“Searching for solutions to coding problems on Google or Bing is inefficient. The information is so often out of date, for the wrong technology, or just plain wrong. With `dev.windows.com/`, we are using a solid combination of searching and filtering to make the developer search experience something of a pleasure.”

The fact is that they have succeeded. Through almost a year of full time Windows 8 development, I've have had much more luck searching `dev.windows.com/` than using Google. What's more, you can easily filter results to just Desktop apps, just JavaScript, or even to just library, forums, or blogs. It is very powerful.

That being said, I use the ontology as much as the search. It is very easy to navigate the API for WinRT, or browse documentation on architecture principles or advances features using the on-screen navigation.

design.windows.com

The `design.windows.com` site is actually a subset of `dev.windows.com/`, but it's worth pointing out. Here's where you'll find all of the resources to make sure that your app meets design style guidelines:

- ✓ Planning resources
- ✓ Design guidance
- ✓ Category guidance
- ✓ Case studies
- ✓ Assets

This last one is quite awesome — prebuilt Photoshop templates with the most common Windows 8 patterns, for easy customization. This just shows the level of support that Microsoft is giving designers these days. Also, it gives design developers — people with a foot in both camps — a really awesome place to start.

social.msdn.microsoft.com/forums

For many years, the MSDN Forums at `social.msdn.microsoft.com/forums` have been the place to go and get support for a Microsoft product that isn't supported through the consumer channels. If you haven't yet done so, go there and set up a forum account using your Live account. It's a very good resource.

For most queries, you should look at the Building Windows Store Apps with HTML5/JavaScript forum. I would recommend filtering your searches to that forum; otherwise, you'll get a lot of false hits on Internet Explorer-related posts.

If you want to ask questions, this is the place. Right now, it's staffed with a bunch of Microsoft folks, such as the incomparable J.P. Sanders. After the General Availability release, a lot more community members, including me, will start replying.

www.stackoverflow.com

For those days when Microsoft doesn't have the answer you want (like when dealing with Facebook Integration, jQuery, or the intricacies of advanced JavaScript), try www.stackoverflow.com.

Stack overflow is a little bit forum, a little bit link aggregation, a little bit blog, a little bit wiki, a little bit country, and a little bit rock and roll (okay, those last two, not so much). It is a user-maintained, -edited and -moderated question-and-answer site just for programmers.

It has its rules. Don't post without searching first to see if the topic has already been addressed elsewhere — they work very hard to prevent duplication and take it very seriously. Do your due diligence to make sure what you are looking for isn't on the site before you post.

But do post. More importantly, register and participate. Programming is a team sport, and nothing exemplifies that more than www.stackoverflow.com.

jQuery.com

You don't have to use jQuery in your Windows 8 app. I don't even cover it in this book. That being said, it can make your life a lot easier when dealing with user-interface details. It is Microsoft-friendly, in general — it's even included in the default ASP.NET install. It can significantly reduce the overall complexity of your code.

You might find the plug-ins for jQuery of particular interest. As I write this, the plug-ins site is under development, so I'm looking at an archive of the old site. I hope the new plug-in site will be done by the time you get around to reading this.

The plug-ins are usually found at plugins.jquery.com, and are just scripts that make use of jQuery as a platform. Need a countdown timer? Done. Fancy form setup? Got it. Cool slide-out calendar control? Have a dozen. jQuery plug-ins can really save you time and effort.

www.buildwindows.com

Build is the new name for the Professional Developers Conference that Microsoft has been holding since the 1990s. It is where they announced Windows 8, and where it will be launched in October 2012.

Every talk at Build is recorded, and the audio and slides are on the Build website (www.buildwindows.com). This is an absolute wealth of information and is must-watching, especially for advanced topics. You can learn about topics like process lifecycle or live Tiles from the people who wrote the code.

jsonformatter.curiousconcept.com

This is a weird entry, but I use it almost every day, so I thought I would toss it in.

JSON, the data language of Windows 8, is usually transferred over the wire without line breaks or tabs to save space. This makes it really hard to debug.

You can copy JSON from the debug visualizer in Visual Studio and paste it into the jsonformatter.curiousconcept.com, and then get nicely formatted JSON back. It even validates it to make sure your braces all line up. It's a pretty slick, and free, tool.

www.w3schools.com/js

It seems like every time I look up a JavaScript syntax question, I find the result at the www.w3schools.com/js website. This is an excellent tutorial site with tons of little simple examples that clarify odd points about the JavaScript language.

There is also an HTML subsite, which is a little less helpful because Microsoft has tweaked the HTML so much to work with WinRT, but the Cascading Style Sheet content is awesome and works very well with Windows 8. In general, the site should earn a place in your bookmarks.

windowsteamblog.com

If you are looking for a little something for your RSS reader, I have to recommend the Windows team blog. This feed, especially the developer blog at windowsteamblog.com/windows/b/developers, is the essential resource for detailed information on how Microsoft went about building Windows 8, WinRT, and WinJS.

This site also contains blogs for user experience, business, security, and a whole host of other things. It's a real gold mine of in-depth strategic information. Add it to your favorite feed reader today!

www.microsoftstore.com

This is where we're all headed, isn't it?

The Windows Store (www.microsoftstore.com) is where the apps that we have created in this book will go for sale. It is also a good starting point for getting a Microsoft Surface or any of the other cool gadgets that are being made to support Windows 8. It is accessible for Windows 8 Desktop apps from your Start screen by clicking Store, but there's more to it than that; make sure to check it out.

Index

• *Symbols and Numerics* •

(hash), 59
3+ setting, 293
7+ setting, 293
12+ setting, 293–294
16+ setting, 294

• *A* •

AAC audio (M4A), 268
Accelerometer class, 270
Access (Microsoft), 10
accessing
 data, 211–214
 events, 126
 Expression Blend from Visual Studio, 57
 Visual Studio, 12
 webcam from app, 266–269
Activated event, 174, 199
activation handling, 201
ActivationKind enumerator,
 199, 235–236
ActivationKind.launch, 194
ActivationKind.search, 194
Ad format, 313
Ad inventory, 313
Ad size, 313
addEventHandler method, 90
adding
 video tag to markup, 109
 ads, 312–320
 AdSense content, 318–319
 buttons for navigation, 131
 fragments, 77–79
 image to default HTML file, 127
 purchase UI, 324–325
 transition functions, 131

 adjusting
 data, 141–142
 ready event, 132
 video properties, 109
 ads
 about, 312
 adding, 312–320
 building, 315–317
 designing for advertising, 312–314
 AdSense (Google), 318–319
 Advanced Features section, 293
 AdWords (Google), 313–314
 age rating, 293–294
 aligning on grids, 32
 animations
 built-in, 152–153
 content, 133
 designing, 38–40
 tiles, 45
 Windows, 128–133
 app bar, 29, 33–34, 123–125
 App Dashboard, registering apps from, 182
 app names, 291
 app.activated event, 201
 AppBar control, 22, 124, 224
 AppBarCommand class, 124
 ApplicationData class, 246
 ApplicationDataContainer, 246–249
 appmanifest, setting up, 199–201
 app.onactivated event, 194, 233–234
 apps
 accessing webcam from, 266–269
 allowing sharing from, 196–199
 business management, 332
 content management, 321
 Cut the Rope, 331
 fantasy football, 322
 health, 332–333
 Hello World, 12–17

apps (*continued*)

- ideas for, 329–333
- Instagram, 332
- interacting with, 121–129
- launching, 235–236
- Mail, 195–196
- managing, 297–298
- Maps, 40–41, 330
- media mashup, 322
- medical, 332–333
- Messenger, 40
- network management, 154
- News, 31, 195–196
- notifying users when not running, 180–185
- organization, 331
- photo, 332
- reading, 331
- record-keeping, 322
- registering, 182–183
- returning results to other, 194
- running with services, 306–307
- sample, 335–339
- searching within, 191–193
- shopping, 322, 333
- stock market, 154
- submitting, 291–296
- suspension of, 228–234
- tax, 321
- testing, 279–285
- ToDoToday, 169
- Weather, 40–41
- Windows Forms, 17, 22
- artboard (Expression Blend), 58
- ASP.NET MVC web application
 - programming interface, 47
- Assets panel (Expression Blend), 58
- attributes
 - data-homePage, 71
 - data-win-control, 22, 73, 76, 103, 121, 123–124, 144–145, 148–149, 224–225
 - data-win-options, 73, 95–97, 145–146, 224
 - textContent, 225

audio

- background, 237–238
- capturing, 268–269
- listening to, 107–108
- playing, 236–238
- <audio> tag, 107–108
- AudioEncodingProperties
 - property, 268
- authentication, 310
- Autodesk SketchBook, 28–29
- automaticallyLoadPages property, 146
- averageRating property, 104
- Azure cloud platform, 47, 185, 300–301

• B •

- Back button, 123
- background audio, 237–238
- background tasks, running, 239–242
- BackgroundDownloader, 238
- BackgroundTransfer collection, 238
- Beighley, Lynn (author)
 - jQuery For Dummies* (Beighley), 138
- Bender, James (author)
 - Professional Test Driven Development with C#, 282*
- binding
 - data, 145–146
 - to a field, 139–141
- BindingList, 222
- Bing Maps, 330
- Blank App template, 66
- blocking users, 102
- body tag, 22, 77, 212
- branding tiles, 44
- breakpoints, 17
- Build, 337–338
- building
 - ads, 315–317
 - Hello World app, 12–17
 - layouts, 69–74, 112–121, 209–214
 - live tiles, 163–168
 - mobile service, 302–303
 - projects, 65–67
 - secondary tiles, 171–174
 - white space, 115–117

built-in animations, 152–153
 business management apps, 332
 <button> tag, 124
 buttons, adding for navigation, 131

• C •

C#, 2, 10
 C++, 2, 56, 66–67
C# 2010 All-in-One For Dummies (Sempf), 104
 cachedFileUpdater value, 235
 Calendar option, 97
 calling on Toast from your app, 176–179
 camera, 264–269
 CameraCaptureUI class, 264–268
 cameraSettings value, 235
 Canvas element, 274
 capabilities, tile, 161–162
 captions, 115
 Capture class, 264–265
 capturing audio, 268–269
 Cascading Style Sheets (CSS), 58–59
 categoryPage.html, 213
 certificates, 295
 changeOpacity handler, 128
 changing
 data, 141–142
 ready event, 132
 video properties, 109
 charms, 189
 Charms bar, 29–30, 33–34
 Checkbox control, 93–95
 choosing languages, 56–57
 class tag, 86
 classes
 Accelerometer, 270
 AppBarCommand, 124
 ApplicationData, 246
 CameraCaptureUI, 264–265, 266, 267–268
 Capture, 264–265
 getVideoChapterSelector, 265
 group-subtitle, 266
 MediaCapture, 265, 268–269
 MediaDevice, 265
 MediaProperties, 268–269
 PushNotification, 183
 sessionState, 232
 setPromise, 232–233
 shareOperation, 201
 StartScreen, 172–173
 tempSettings, 249
 TileUpdateManager, 160
 Windows.Media.Transcoding.
 MediaTranscoder, 265
 Windows.Networking.
 PushNotifications, 309
 Windows.Storage.Pickers, 256
 Windows.UI.Input.PointerPoint, 274
 WinJS.Application, 203
 click-through rate (CTR), 314
 ClientConfiguration error code, 318
 ClosedByUser response, 236
 cloud
 about, 46–47, 299
 Azure, 47, 185, 300–301
 getting authenticated, 310
 pushing data to user, 309
 Windows Azure Mobile Services, 301–309
 code
 for Hello World app, 15–16
 Play To contract example, 205–206
 for this book, 6
 <code>, 115
 collectionPage, 211–212
 collectionPage.html template, 211
 collections, of objects, 143–144
 commands
 content providing, 33
 contextual, 34–35, 125–128
 commonDialog, 206, 256
 communicating, with Internet, 214–219
 comparing programs, 17–20
 configuring tiles, 163–166
 CONNECT, 216
 connecting Windows Store app, 304–306
 connectivity, network, 54–55

- constructing databases, 303–304
- contact lists, 154
- contactPicker value, 235
- ContainerEncodingProperties
 - property, 268
- content
 - animating, 133
 - designing, 26–30
 - providing commands, 33
 - providing navigation, 36
 - rendering, 167–168
- content management app, 321
- contentHost, 76, 77, 211
- contentTransition, 152
- contextual commands, 34–35, 125–128
- contracts
 - about, 189–190
 - charms, 189–190
 - defined, 189
 - File Picker contract, 206–208, 256–258
 - Play To contract, 204–206, 265–266
 - Search contract, 42–43, 190–194
 - Settings contract, 43, 201–204, 249–252
 - Share contract, 42, 194–201
- controls
 - about, 83
 - HTML, 83–99
 - informing the user, 99–107
 - playing media files with HTML5, 107–109
- conventions, 5, 22
- cost per action (CPA), 313
- cost per click (CPC), 313
- cost per thousand impressions (CPM), 313
- CPA (cost per action), 313
- CPC (cost per click), 313
- CPM (cost per thousand impressions), 313
- Create New Table dialog box, 303–304
- createContainer, 247
- createFileAsync method, 238
- createGrouped, 150–151
- createPushNotificationChannel-
 - ForApplicationAsync method, 183
- createToastNotifier, 180
- creating
 - ads, 315–317
 - Hello World app, 12–17

- layouts, 69–74, 112–121, 209–214
- live tiles, 163–168
- mobile service, 302–303
- projects, 65–67
- secondary tiles, 171–174
- white space, 115–117
- crossSlide, 126
- cryptography, 294
- CSS properties, 58–59
- CTR (click-through rate), 314
- Current option, 97
- currentItem property, 147
- currentPage property, 149
- Cut the Rope app, 331
- CVR (ad conversion), 314



- data. *See also* presenting data
 - accessing, 211–212, 213–214
 - binding, 145–146
 - changing, 141–142
 - handling, 219–226
 - listing, 142–149
 - pushing to users, 309
 - setting up, 253
- data collection, from sensors, 269–273
- Data parameter, 216
- databases
 - constructing, 303–304
 - deployable, 258–260
 - HTML5 options, 260
- data-homePage attribute, 71
- datarequested event, 197
- dataSource, 145, 151, 192
- data-win-bind tag, 73, 139, 225
- data-win-control attribute, 22, 73, 76, 103, 121, 123–124, 144–145, 148–149, 224, 225
- data-win-options attribute, 73, 95–97, 145, 146, 224
- data-win-properties property, 153
- datePattern option, 97
- DatePicker control, 97–99, 224
- debugging Windows 8 programs, 16
- declarative binding, 139–142

- Default.css file, 14
 - Default.htm file, 14
 - default.html template, 211
 - Default.js file, 14
 - DELETE, 216
 - deleteContainer, 247
 - deployable databases, 258–260
 - Description field, 295
 - design checklist, 21
 - design language, 161
 - design principles, setting, 20
 - designing
 - for advertising, 312–314
 - animation, 38–40
 - cloud, 46–47
 - content, 26–30
 - contracts, 42–43
 - interactions, 33–35
 - layout, 30–32
 - for multiple screen sizing, 41
 - for multiple views, 40–41
 - navigation, 35–38
 - notifications, 46
 - pixel density, 41–42
 - with style, 26–38
 - tiles, 44–45
 - for touch, 51–53
 - for Windows Push Notification Service, 181–182
 - design.windows.com, 336
 - developer accounts, registering for, 290
 - Developer Portal, 290
 - developers
 - about, 22–23
 - tools, 23–24
 - WinRT, 24
 - development. *See also specific topics*
 - about, 49
 - designing for touch, 51–53
 - driving user confidence, 60–61
 - enabling multitasking, 53–54
 - environment, 55–59
 - Expression Blend, 57–59
 - languages, 56–57
 - learning new SDKs, 60
 - network connectivity, 54–55
 - providing what users need, 51
 - tips, 61
 - user focus, 49–55
 - Visual Studio, 55
 - device value, 235
 - dev.windows.com, 335–336
 - Digital Living Network Alliance (DLNA-compliant), 204
 - Digitally signed file test, 296
 - DirectX Applications, 66
 - DirectX Graphics Infrastructure (DXGI), 18–19
 - disabled option, 97
 - disabling notifications, 180
 - displaying grids, 148
 - distractions, reducing, 28–30
 - div tag, 22, 75, 84–87, 138
 - DLNA-compliant (Digital Living Network Alliance), 204
 - `$(document).ready()`, 86
 - DOM, setting single values using, 138
 - doubletap, 126
 - down event, 275
 - drag, 126
 - DXGI (DirectX Graphics Infrastructure), 18–19
- **E** ●
- ECCN (Export Control Classification Number), 294
 - edge, of screen, 33–34
 - editions, Visual Studio, 64
 - Element option, 97
 - elements
 - Canvas, 274
 - instance, 216
 - onready, 216
 - Select, 91–93
 - statechange, 216
 - enabling
 - multitasking, 53–54
 - queue, 170
 - encoding, 265
 - enterContent function, 133
 - enterPage function, 129, 130

- entrance, 152
- ergonomics, 31–32, 117–120
- EventListener, 135
- events
 - accessing, 126
 - Activated, 174, 199
 - app.activated, 201
 - app.onactivated, 194, 233–234
 - datarequested, 197
 - down, 275
 - fragment.load, 74
 - itemInvoked, 123
 - move, 275
 - navigation, 133
 - onActivated, 127, 131, 167–168, 170, 176, 178–179, 197, 235, 274
 - onappactivated, 171, 231–233, 255–256
 - onChange, 99, 105–106
 - oncheckpoint, 231–233
 - onreadingchanged, 271
 - ready, 132
 - registering, 98–99
 - resize, 135
 - resume, registering, 233–234
 - sourcerequested, 205–206
 - up, 275
- exitContent function, 133
- exitPage function, 129
- expectations, tile, 162
- Export Control Classification Number (ECCN), 294
- Expression Blend
 - about, 23–24, 57–58
 - Assets panel, 58
 - CSS properties, 58–59
 - handling JavaScript, 59
- extensions, 254

• F •

- fantasy football app, 322
- field, binding to, 139–141
- file extensions, 254
- File Picker contract, 206–208, 256–258
- file system, 252–256

- file target, 235
- File-encoding test, 296
- FileOpenPicker value, 206, 235
- files
 - keeping settings in, 249
 - moving, 238–239
 - reading, 255–256
 - writing to, 253–255
- fileSavePicker value, 235
- filing
 - File Picker contract, 256–258
 - file system, 252–256
- fill view, 53
- filled, 134
- Find feature, 43
- finding directions, 330
- fixed layout, 74
- Fixed template, 111
- flat navigation pattern, 36–37
- flick gesture, 16
- Flickr feed, 220–221
- FlipView, 148–149, 224
- Flyout, 224
- for loop, 92–93
- format, of data, 214
- formatting results, 222–226
- fragment.load event, 74
- fragments
 - adding, 77–79
 - loading, 76–77
- FrontPage (Microsoft), 10
- full screen view, 53
- fullscreen-landscape, 134
- fullscreen-portrait, 134
- functions
 - enterContent, 133
 - enterPage, 129–130
 - exitContent, 133
 - exitPage, 129
 - getPointerDeviceType, 276
 - load, 133
 - navigate, 74
 - parse(), 253
 - prepareFilesTranscodeAsynch, 265
 - processAll, 139–140, 169, 197

ready(), 86, 106
 sliderChanged, 106
 stringify(), 253
 takeapic, 267–268
 transition, 131
 XmlHttpResponse, 100

• G •

games
 charging for, 321
 developing, 330–331
 Geolocation namespace, 269
 GestureLibrary, 39, 126
 GestureRecognizer, 128
 gestures, designing for, 33
 gestureSettings property, 126
 GET, 215–216
 getCurrentDownloadsAsync, 239
 getCurrentReading, 271
 getGroups method, 212–213
 getItems method, 212–213
 getPointerType function, 276
 getTemplateContent method, 167
 getVideoChapterSelector class, 265
 GIMP, 28
 Github, 280–282
 glyphs, 33
 ‘going live,’ 317–318
 Google AdSense, 318–319
 Google AdWords, 313–314
 Grid Application template, 66, 210–212
 Grid layout, 69
 Grid template, 111
 grids
 aligning on, 32
 displaying, 148
 putting things on, 120–121
 GridView layout, 149, 192
 group control, 149–151
 groupable property, 149
 groupDataSource, 151, 221
 groupedItemsList, 157
 group-subtitle class, 226

• H •

handling
 activation, 201
 data, 219–226
 JavaScript, 59
 hardware integration
 about, 263
 accessing webcam from app, 266–269
 camera, 264–269
 data collection from sensors, 269–273
 microphone, 264–269
 touch features, 273–277
 Windows.Media API, 264–266
 Harris, Andy (author), 3, 86, 134, 226
 HTML, XHTML, & CSS All-In-One For Dummies, 3
 HTML, XHTML, & CSS For Dummies, 226
 HTML5 For Dummies, 134
 JavaScript and AJAX For Dummies, 86
 hash (#), 59
 HEAD, 216
 header tag, 85, 216
 headers, 115
 headerTemplate, 225
 health apps, 332–333
 Hello World app, 12–17
 Hello World_TemporaryKey.pfx file, 14
 hierarchical navigation pattern, 36–37
 hold, 126
HTML, XHTML, & CSS All-In-One For Dummies (Harris), 3
HTML, XHTML, & CSS For Dummies (Harris), 226
 HTML controls
 about, 83–84
 Checkbox control, 93–95
 DatePicker, 97–99, 224
 div tag, 22, 75, 84–87, 138
 Select element, 91–93
 textbox, 87–91
 TimePicker, 97–99, 224
 ToggleSwitch control, 95–97, 224

HTML5

about, 57

database options, 260

playing media files with, 107–109

HTML5 For Dummies (Harris), 134

HtmlControl, 224

1

icons, explained, 4–5

iconTextApplicationsTemplate, 120

IDEs (integrated development

environments), 55

imageAndTextCollectionTemplate, 121

imageAndTextListFileTemplate, 121

ImageEncodingProperties property, 268

imageGalleryBasketTemplate, 120

imageGalleryTemplate, 120

imageOverlayAlbumTemplate, 120

imageOverlayGalleryFolder-
Template, 120

imageOverlayLandingTemplate, 120

images, adding to default HTML file, 127

implementing styles, 114–115

Impression, 313

in-app purchasing

about, 320–321

planning for expansion, 321–322

preparing for purchasing content,
322–324

indeterminate progress status, 101–102

Indexed Database API, 260

IndexedDB, 260

indexOfFirstVisible property, 147

indexOfLastVisible property, 147

inductive screens, 52

innerHTML, 102

innerText property, 138

Input control, 88–89

Input namespace, 269

<input> tag, 124

Instagram app, 332

Installed section (New Project dialog
box), 64

Install-uninstall test, 296

instance element, 216

integrated development environments
(IDEs), 55integrating. *See also* hardware integration

navigation, 122–123

planning tools, 330

interactions

with apps, 121–129

designing, 33–35

Internet, communicating with, 214–219

Internet resources

Azure, 301

Build, 337–338

certificates, 295

code for book, 6

codeplex, 259

DXGI, 18

File Picker contract, 258

Github, 280–282

in-app purchase setup, 325

jQuery, 337

jsonformatter.curiousconcept, 338

Microsoft Ad SDK, 315

Microsoft design guidelines, 44

microsoftstore, 339

Mobile Services SDK, 301

promises, 218

JUnit, 280–282

Semantic Web, 115

SQLite, 258

SQLite repository, 259

stackoverflow, 337

standards, 260

Toast, 176, 179

updates for this book, 6

Visual Studio 2012 Express, 3

w3schools, 338

Windows Live Services, 309

windowsteamblog, 338

Windows.UI.Notifications.

TileTemplateType enumerator, 165

InvalidServerResponse error code, 318

Item template, 225

itemDataSource property, 146, 149

itemInvoked event, 123

ItemList, 73, 193
itemListSection section, 73
itemTemplate property, 73, 145, 147, 149,
211, 225

• J •

JavaScript
 handling, 59
 templates, 67
JavaScript and AJAX For Dummies (Harris), 86
JavaScript Object Notation format (JSON),
143, 304
jQuery, 138, 337
jQuery For Dummies (Beighley), 138
JSON (JavaScript Object Notation) format,
143, 304
jsonformatter.curiousconcept, 338
JSON.parse(), 93, 143
JSON.stringify(), 143

• K •

kind property, 235
Kinney, Adam (developer), 245

• L •

landingPage.html template, 211
languages
 choosing, 56–57
 design, 161
Launch and suspend performance test, 296
launch value, 235
launching apps, 235–236
layouts
 about, 67–69, 111–112, 209–210
 animation, 128–135
 building, 69–74, 112–121, 209–214
 design, 30–32
 ergonomics, 31–32, 117–120
 fixed, 74
 grids, 69, 120–121, 148
 Grid Application template, 66, 210–212
 interacting with app, 121–128

 List layout, 212–214
 Split, 70–74
 typography, 30–31, 112–115
 white space, 30, 115–117
layout property, 147, 148
library books, 154
licenseInformation object, 325
lifecycle, verifying, 284–285
light sensor, 272–273
lighting, 272–273
line-of-business API, 19
List layout, 212–214
Listbox, 91
listening to audio, 107–108
listing data, 142–149
ListView, 94–95, 117, 144–148, 224
ListViewAnimationType, 152
Live Services, 159
live tiles, 242
load function, 133
loading fragments, 76–77
loadingBehavior property, 147
local settings, organizing, 244–246
local storage
 about, 243–244
 ApplicationDataContainers, 246–249
 deployable databases, 258–260
 File Picker contract, 256–258
 file system, 252–256
 filing, 252–258
 HTML5 database options, 260
 organizing local settings, 244–246
 relational database, 258–259
 roaming settings, 246
 settings, 244–252
 Settings charm for user access, 249–252
localSettings, 244–247
location, of user, 269
Looney, Andy (developer), 331

• M •

M4A (AAC audio), 268
Mail app, 195–196
managing apps, 297–298

- manipulationRotate, 126
- manipulationRotateInertia, 126
- manipulationScale, 126
- manipulationScaleInertia, 126
- manipulationTranslateInertia, 126
- manipulationTranslateX, 126
- manipulationTranslateY, 126
- Maps app, 40–41, 330
- mashups
 - about, 209
 - building layouts, 209–214
 - communicating with Internet, 214–219
 - handling data, 219–226
- max property, 106
- maxRating property, 104
- maxYear option, 97
- media capture, 264–265
- media files, playing with HTML5, 107–109
- media mashup app, 322
- media queries, 134
- media tools, 322
- MediaCapture class, 265, 268–269
- MediaDevice class, 265
- MediaEncodingProfile property, 268
- MediaProperties class, 268–269
- MediaPropertySet property, 268
- MediaRatio property, 268
- MediaTranscoder, 265
- medical apps, 332–333
- Menu, 224
- MessageBox.show(), 46
- Messenger app, 40
- Method parameter, 216
- methods
 - addEventListener, 90
 - createFileAsync, 238
 - createPushNotificationChannel-
ForApplicationAsync, 183
 - getGroups, 212–213
 - getItems, 212–213
 - getTemplateContent, 167
 - onreadystatechange, 216–218
 - open, 216–218
 - Request, 215
 - then, 218–219
 - XHR, 216–217, 221–222
 - XMLHttpRequest.send(), 216
 - XMLHttpRequest.setRequestHeader-
Header(), 216
- metrics, 313–314
- microphone, 264–269
- Microsoft
 - path of, 10–12
 - style sheet, 113–114
- Microsoft Access, 10
- Microsoft FrontPage, 10
- Microsoft Media Foundation, 265
- microsoftstore (website), 339
- min property, 106
- miniaturization, 52
- minYear option, 97
- mobile service, creating, 302–303
- Mobile Services SDK, 301
- monetization
 - about, 311
 - ad networks, 318–320
 - ads, 312–320
 - designing for advertising, 312–314
 - in-app purchasing, 320–326
 - Windows 8 Ads SDK, 314–318
- monthPattern option, 97
- MOTS button, 324
- mouse features, 273
- move event, 275
- moving files, 238–239
- MP3 audio, 268
- ms-view-state property, 134
- multitasking, enabling, 53–54
- Multiuser session test, 296



- names, app, 291
- navigate function, 74
- navigating
 - about, 74–76
 - adding buttons for, 131
 - adding fragments, 77–79
 - designing, 35–38
 - integrating, 122–123
 - loading fragments, 76–77

persistent, 36
 in Windows Forms app, 22
 navigation event, 133
 Navigation library, 75, 122–123
 navigation patterns, 35–36, 36–37
 .NET, 17, 19, 56, 65–66
 network connectivity, 54–55
 network management app, 154
 NetworkConnectionFailure error code, 318
 New Project dialog box (Visual Studio),
 64–65
 News app, 31, 195–196
 NoAdAvailable error code, 318
 nosoncallback parameter, 222
 Notes to testers field, 296
 notifications. *See also* tiles
 about, 159–160, 168–169
 combining with tiles, 169–170
 designing, 46
 disabling, 180
 messages on queue, 170–171
 pushing, 183–184
 when app is not running, 180–185
 Notifications object, 160
 NotRunning response, 236

• 0 •

objects, collections of, 143–144
 onActivated event, 127, 131, 167–168,
 170, 176, 178–179, 197, 235, 274
 onappactivated event, 171, 231–233,
 255–256
 onChange event, 99, 105, 106
 oncheckpoint event, 231–233
 online interactions, charging for, 321
 Online section (New Project dialog box), 64
 onreadingchanged event, 271
 onready element, 216
 onreadystatechange method,
 216, 217, 218
 open method, 216, 217, 218
 optimizing, for touch, 12
 options, viewing, 98
 organization apps, 331

organizing local settings, 244–246
 orientation property, 149

• p •

Package Security Identifier (SID), 183
 package.appxmanifest, 14, 107, 109,
 163–164
 packages, 294–295
 PageControl, 84, 139, 192, 224
 Page-level transitions, 129–133
 pages, getting text boxes onto, 88–89
 pagesToLoad property, 146
 parameters, promise, 218
 parse() function, 253
 Password parameter, 216
 Peek feature, 165
 pens, writing with, 273–277
 persistent navigation, 36
 photo apps, 332
 photos, snapping, 266–267
 Pickers contract, 43
 Pinball FX2, 320–321
 pinch gesture, 39, 126
 pinning secondary tiles, 172–174
 pixel density, 41–42
 PlaceToPutText, 138
 planning
 for app expansion, 321–322
 tools, 330
 for views, 133–134
 Play To contract
 about, 204
 coding example, 205–206
 signing, 265–266
 playing
 audio, 236–238
 media files with HTML5, 107–109
 PLM (process lifecycle management)
 about, 227
 features after suspension, 236–239
 handling app suspension, 228–234
 launching apps, 235–236
 live tiles, 242
 registering resume events, 233–234

PLM (*continued*)

- running background tasks, 239–242
- switching tasks, 228–231
- termination, 231–233
- plug-ins, 337
- pointer features, 273
- POINTtodo app, 151, 287, 292, 293, 295
- populating Select page control, 91–92
- Portable namespac, 269
- POST, 216
- prepareFilesTranscodeAsynch
 - function, 265
- presenting data
 - about, 137
 - built-in animations, 152–153
 - collections of objects, 143–144
 - declarative binding in WinJS, 139–142
 - FlipView, 148–149, 224
 - group control, 149–151
 - listing data, 142–149
 - Listview, 94–95, 117, 144–148, 224
 - selecting, 151–152
 - Semantic Zoom, 38, 153–158
 - setting single values using DOM, 138
- press and hold gesture, 39, 125
- previousExecutionState property, 235, 236
- Printers namespace, 269
- printTaskSettings value, 235
- process lifecycle management (PLM)
 - about, 227
 - features after suspension, 236–239
 - handling app suspension, 228–234
 - launching apps, 235–236
 - live tiles, 242
 - registering resume events, 233–234
 - running background tasks, 239–242
 - switching tasks, 228–231
 - termination, 231–233
- processAll function, 139–140, 169, 197
- processError, 222
- processPhotos, 222
- product catalog, 154
- ProductID, 326
- Professional Test Driven Development with C# (Bender)*, 282

programs

- comparing, 17–20
- debugging, 16
- progress status
 - determining, 100–101
 - indeterminate, 101–102
- ProgressBar control, 100–102
- ProgressEvent, 100
- projects
 - creating, 65–67
 - types, 64–65
- promise, 217–218
- properties
 - of AppBar buttons, 124
 - AudioEncodingProperties, 268
 - automaticallyLoadPages, 146
 - averageRating, 104
 - ContainerEncodingProperties, 268
 - currentItem, 147
 - currentPage, 149
 - data-win-properties, 153
 - gestureSettings, 126
 - groupable, 149
 - ImageEncodingProperties, 268
 - indexOfFirstVisible, 147
 - indexOfLastVisible, 147
 - innerText, 138
 - itemDataSource, 146, 149
 - itemTemplate, 73, 145, 147, 149, 211, 225
 - kind, 235
 - layout, 147, 148
 - loadingBehavior, 147
 - max, 106
 - maxRating, 104
 - MediaEncodingProfile, 268
 - MediaPropertySet, 268
 - MediaRatio, 268
 - min, 106
 - ms-view-state, 134
 - orientation, 149
 - pagesToLoad, 146
 - previousExecutionState, 235–236
 - scrollPosition, 147
 - selection, 147
 - selectionMode, 147, 151–152

- step, 106
- swipeBehavior, 147
- tapBehavior, 147
- userRating, 104
- value, 89, 106
- video, 109
- videoEncodingProperties, 268
- WhenToDo, 149–150
- zoomableView, 147
- protocol value, 235
- publishing
 - social content, 329
 - with templates, 165–167
- purchase UI, adding, 324–325
- pushing
 - data to users, 309
 - notifications, 183–184
 - to Store, 289–297
- PushNotification class, 183
- PUT, 216

• Q •

- queue, enabling, 170
- JUnit, 280–282

• R •

- Rating, 224
- Rating controls, 103–105
- reading apps, 331
- reading files, 255–256
- readTextAsync, 255–256
- ready event, changing, 132
- ready() function, 86, 106
- Recent section (New Project dialog box), 64
- recording video, 267–268
- record-keeping app, 322
- reducing distractions, 28–30
- registering
 - apps, 182–183
 - for developer accounts, 290
 - events, 98–99
 - resume events, 233–234

- relational databases, 258–259
- Remember icon, 5
- rendering content, 167–168
- Repeater template, 223–225
- Request method, 215
- requestCreateAsync, 173
- resize event, 135
- resolutions, designing for, 42
- Restart manager message test, 296
- results, formatting, 222–226
- resume events, registering, 233–234
- returning results to other apps, 194
- ring, 102
- roamingSettings, 246
- Ruby forms, 17
- running
 - apps with services, 306–307
 - background tasks, 239–242
- Running response, 236
- runtime behavior, verifying, 282–284

• S •

- Safe mode test, 296
- sample apps, 335–339
- scaling, 133–135
- ScheduledToastNotification, 178
- scheduling notifications, 176
- screen, edge of, 33–34
- scrollTop property, 147
- SDKs, 60
- Search contract
 - about, 42–43, 190–191
 - returning results to other apps, 194
 - searching within apps, 191–193
- search value, 235
- searching within apps, 191–193
- secondary tiles, 44, 171–174
- section tag, 72, 85, 223
- Secure Sockets Layer (SSL), 216
- Segoe font, 30–31
- Select element, 91–93
- selection property, 147
- selectionMode property, 147, 151–152
- Selling Details section, 292–293

- Sells, Chris (developer), 331
- Semantic Web, 115, 122
- Semantic Zoom
 - about, 38, 153–154
 - technical details, 156–158
 - using, 154–156
- SemanticZoom, 224
- Sempf, Bill (author), 104
 - C# 2010 All-in-One For Dummies*, 104
- sensitivity, 20
- sensors, data collection from, 269–273
- Sensors namespace, 269
- ServerError error code, 318
- sessionState class, 232
- setPromise class, 232–233
- setting(s)
 - age rating, 293–294
 - design principles, 20
 - keeping in files, 249
 - local, 244–246
 - roaming, 246
 - single values using DOM, 138
- Settings contract
 - about, 43, 201–204
 - giving user access with, 249–252
- SettingsFlyout, 202–203, 224
- setup
 - about, 63
 - animated page sequences, 130
 - appmanifest, 199–201
 - building layouts, 69–74
 - navigating, 74–79
 - Store for your app, 325–326
 - Visual Studio, getting started in, 63–69
- 7+ setting, 293
- Share contract
 - about, 42, 194–195
 - allowing sharing from app, 196–199
 - share targets, 199–201
 - using, 195–196
- share targets, 199–201
- shareOperation class, 201
- shareTarget value, 235
- shopping apps, 322, 333
- SID (Package Security Identifier), 183
- signing Play To contract, 265–266
- Silhouette (Windows 8), 32
- Silverlight, 57
- 16+ setting, 294
- slide gesture, 39, 125
- Slider controls, 105–107
- sliderChanged function, 106
- SMS namespace, 269
- snapped, 134
- snapped view, 53
- snapping, 20, 133–135
- snapping photos, 266–267
- social content, publishing, 329
- social.msdn.microsoft.com/
forums, 336
- Solution Explorer, 59
- sourcerequested event, 205–206
- sources of data, 214
- Split App (XAML) template, 66
- Split layout, 70–74
- Split template, 111
- splitPage.html, 213
- SQLite, 258–259
- SSL (Secure Sockets Layer), 216
- stackoverflow (website), 337
- startingPosition, 130, 133
- StartScreen class, 172–173
- statechange element, 216
- step property, 106
- stock market app, 154
- Store
 - about, 279
 - managing apps, 297–298
 - manually checking style, 287–289
 - pushing to, 289–297
 - setting up for your app, 325–326
 - testing apps, 279–285
 - Windows App Certification Kit, 285–287
- stretch gesture, 126
- stringify() function, 253
- style
 - about, 25
 - animation, 38–40

- cloud, 46–47
- content, 26–30
- contracts, 42–43
- designing for multiple screen sizing, 41
- designing for multiple views, 40–41
- designing with, 26–38
- implementing, 114–115
- interactions, 33–35
- layout, 30–32
- manually checking, 287–289
- navigation, 35–38
- notifications, 46
- pixel density, 41–42
- tiles, 44–45
- style sheet (Microsoft), 113–114
- styling, 225–226
- submitting apps, 291–296
- Suspended response, 236
- suspended view, 53
- suspension
 - of apps, 228–234
 - dealing with, 231–233
 - features allowed after, 236–239
- sweep gesture, 17
- swipe gesture, 39, 97, 125
- swipeBehavior property, 147
- switching tasks, 228–231
- System.Drawing namespace, 18

• T •

Table ribbon, 35

tags

- <audio>, 107–108
- body, 22, 77, 212
- <button>, 124
- class, 86
- data-win-bind, 73, 139, 225
- div, 22, 75, 84–87, 138
- header, 85, 216
- <input>, 124
- section, 72, 85, 223
- <video>, 107–108
- takeapic function, 267–268

- tap gesture, 39, 125, 126
- tapBehavior property, 147
- Task Manager, 229
- tasks, switching, 228–231
- tax app, 321
- Technical Stuff icon, 5
- templates
 - Fixed, 111
 - Grid, 111
 - Grid Application, 66, 210–212
 - JavaScript, 67
 - landingPage.html, 211
 - ListView, 144–145
 - in .NET languages, 65–66
 - publishing with, 165–167
 - Repeater, 223–225
 - Split App (XAML), 66
- Templates section (New Project dialog box), 64
- tempSettings class, 249
- Terminated reponse, 236
- termination
 - about, 229–231
 - dealing with, 231–233
 - testing, 285
- Test for apps that crash and stop
 - recording, 296
- Test for changes to Windows security
 - features, 296
- Test for correct folder use, 297
- Test for debug apps, 297
- Test for use of APIs for Windows Store-style apps, 297
- Test for use of Windows compatibility
 - fixes, 297
- Test of OS version check logic, 297
- Test of resources defined in the app
 - manifest, 297
- testing
 - apps, 279–285
 - termination, 285
 - tips, 296–297
- text, 178
- textAlbumTrackTemplate, 121

- textbox, 87–91
- textContent attribute, 225
- textListMediaQueueTemplate, 118
- textTileLandingTemplate, 121
- textTileListFolderTemplate, 121
- then method, 218–219
- 3+ setting, 293
- TileID, 173
- TileOptions enumerator, 173
- tiles. *See also* notifications
 - about, 44, 51, 159–160
 - animating, 45
 - branding, 44
 - capabilities of, 161–162
 - configuring, 163–165
 - design language, 161
 - expectations, 162
 - live, building, 163–168, 242
 - publishing with templates, 165–167
 - rendering content, 167–168
 - secondary, 44, 171–174
 - Toast, 174–180
 - using, 160–162
- tileTemplate, 167
- TileUpdateManager class, 160
- TileWideImageAndText 01, 166
- TimePicker control, 97–99, 224
- Tip icon, 5
- Toast
 - about, 174–175
 - calling on from your app, 176–179
 - when to use, 175–176
- ToastNotifier, 180
- ToDoToday app, 169
- ToggleSwitch control, 95–97, 224
- tools
 - developer, 23–24
 - media, 322
 - planning, 330
- Tooltip, 224
- ToString, 98
- touch
 - designing for, 39–40, 51–53
 - optimizing for, 12
 - touch features, 273–277
- TRACE, 216
- transition functions, 131
- transitions, 129–133. *See also* animations
- turn gesture, 39, 125
- 12+ setting, 293–294
- typography, 30–31, 112–115



- unit testing framework, 280–282
- Universal Plug and Play (UPnP), 204
- Unknown error code, 318
- up event, 275
- updates, for this book, 6
- UPnP (Universal Plug and Play), 204
- URL parameter, 216
- User Account Control test, 297
- user confidence, driving, 60–61
- user focus
 - about, 49–50
 - designing for touch, 51–53
 - enabling multitasking, 53–54
 - immersing user, 50–51
 - network connectivity, 54–55
 - providing what user needs, 51
- user interface, WinRT, 18
- User parameter, 216
- userRating property, 104
- users
 - about, 20
 - blocking, 102
 - convention, 22
 - design checklist, 21
 - design principles, 20
 - getting location, 269
 - giving access with Settings charm, 249–252
 - giving control to, 179–180
 - informing, 99–107
 - pushing data to, 309
- UUs, 313

• V •

- value property, 89, 106
- values
 - cachedFileUpdater, 235
 - cameraSettings, 235
 - contactPicker, 235
 - device, 235
 - FileOpenPicker, 206, 235
 - fileSavePicker, 235
 - launch, 235
 - printTaskSettings, 235
 - protocol, 235
 - search, 235
 - setting using DOM, 138
 - shareTarget, 235
- VB6 (Visual Basic 6), 10
- verifying
 - lifecycle, 284–285
 - runtime behavior, 282–284
- video recording, 267–268
- <video> tag, 107–108
- videoconferencing, 266
- videoEncodingProperties
 - property, 268
- view states, 53–54
- ViewBox, 74, 224
- viewing options, 98
- views
 - designing for multiple, 40–41
 - planning for, 133–134
- view-state change, 135
- Visual Basic 6 (VB6), 10
- Visual Studio
 - about, 55, 63–64
 - accessing, 12
 - accessing Expression Blend from, 57
 - editions, 64
 - layout, 67–69
 - making new projects, 65–67
 - project types, 64–65
- Visual Studio 2012 Express, 3

• W •

- w3schools (website), 338
- WACK (Windows App Certification Kit), 285–287
- Warning! icon, 5
- Weather app, 40–41
- Web Services, 46–47
- webcam, accessing from app, 266–269
- websites
 - Azure, 301
 - Build, 337–338
 - certificates, 295
 - code for book, 6
 - codeplex, 259
 - DXGI, 18
 - File Picker contract, 258
 - Github, 280–282
 - in-app purchase setup, 325
 - jQuery, 337
 - jsonformatter.curiousconcept, 338
 - Microsoft Ad SDK, 315
 - Microsoft design guidelines, 44
 - microsoftstore, 339
 - Mobile Services SDK, 301
 - promises, 218
 - QUnit, 280–282
 - Semantic Web, 115
 - SQLite, 258
 - SQLite repository, 259
 - stackoverflow, 337
 - standards, 260
 - Toast, 176, 179
 - updates for this book, 6
 - Visual Studio 2012 Express, 3
 - w3schools, 338
 - Windows Live Services, 309
 - windowsteamblog, 338
 - Windows.UI.Notifications.
 - TileTemplateType enumerator, 165
- WhenToDo property, 149–150
- white space, 30, 115–117

win-content, 202–203

Windows

animations, 128–133

jQuery and, 138

Windows 8. *See also specific topics*

about, 9–10

building Hello World app, 12–17

comparing programs, 17–20

convention, 22

debugging programs, 16

design checklist, 21

design principles, 20

developer experience, 22–24

developer tools, 23–24

Microsoft's path, 10–12

users, 20–22

WinRT, 24

Windows 8 Ads SDK, 314–318

Windows 8 Silhouette, 32

Windows App Certification Kit (WACK),
285–287

Windows Azure, 47, 185, 300–301

Windows Azure Mobile Services

about, 301

advanced features, 307–309

connecting Windows Store app, 304–306

constructing databases, 303–304

creating new mobile service, 302–303

getting started, 301

running apps with services, 306–307

Windows Communication Foundation, 47

Windows Forms app, 17, 22

Windows Live Services, 246, 309

Windows Media Audio (WMA), 268

Windows platform support test, 297

Windows Presentation Foundation, 10

Windows Push Notification Service (WNS),
180–182

Windows Runtime (WinRT)

about, 1, 17

animations library, 39

for developers, 24

user interface, 18

wrapping SQLite for, 259

Windows Store app, connecting, 304–306

Windows Store-style app-manifest test, 296

Windows.Media API, 264–266

Windows.Media.Devices namespace,
264–265

Windows.Media.Transcoding.

MediaTranscoder class, 265

Windows.Networking.

PushNotifications class, 309

Windows.Storage namespace,

238, 244, 256

Windows.Storage.Pickers class, 256

windowsteamblog (website), 338

Windows.UI.Input.PointerPoint
class, 274

Windows.UI.Notifications.

TileTemplateType enumerator, 165

WinJS, 39, 57, 113–114, 139–142

WinJS.Application class, 203

WinJS.Binding namespace, 137

WinJS.Binding.as, 140

WinJS.Binding.List, 146, 156

WinJS.Binding.Template, 73, 225

WinJS.Navigation namespace, 76, 77

WinJS.UI.Animation, 129

WinJS.UI.AppBar, 76

WinJS.UI.FlipView, 148–149

WinJS.UI.GridView, 148

WinJS.UI.ListView, 144, 146, 148,
156–158

WinJS.UI.SettingsFlyout, 249–250

WinJS.UI.Template, 144

win-listview style, 117

win-options parameter, 213

WinRT (Windows Runtime)

about, 1, 17

animations library, 39

for developers, 24

user interface, 18

wrapping SQLite for, 259

WinRT Component DLL, 66

win-surface style, 117–118

win-type style, 115

win-viewport style, 117–118

WMA (Windows Media Audio), 268

WNS (Windows Push Notification Service),
180–182

wrapping SQLite for WinRT, 259

writing
to files, 253–255
with pens, 273–277

• X •

XAML, 17, 56–57, 139
XHR method, 216–217, 221–222
XMLHttpRequest, 100, 215–218
XMLHttpRequest.send() method, 216
XMLHttpRequest.setRequestHeader()
method, 216
XmlHttpRequest function, 100

• Y •

yearPattern option, 97
Yield, 313

• Z •

zoomableView property, 147
zooming. *See* Semantic Zoom

Apple & Mac

iPad 2 For Dummies,
3rd Edition
978-1-118-17679-5

iPhone 4S For Dummies,
5th Edition
978-1-118-03671-6

iPod touch For Dummies,
3rd Edition
978-1-118-12960-9

Mac OS X Lion
For Dummies
978-1-118-02205-4

Blogging & Social Media

CityVille For Dummies
978-1-118-08337-6

Facebook For Dummies,
4th Edition
978-1-118-09562-1

Mom Blogging
For Dummies
978-1-118-03843-7

Twitter For Dummies,
2nd Edition
978-0-470-76879-2

WordPress For Dummies,
4th Edition
978-1-118-07342-1

Business

Cash Flow For Dummies
978-1-118-01850-7

Investing For Dummies,
6th Edition
978-0-470-90545-6

Job Searching with Social
Media For Dummies
978-0-470-93072-4

QuickBooks 2012
For Dummies
978-1-118-09120-3

Resumes For Dummies,
6th Edition
978-0-470-87361-8

Starting an Etsy Business
For Dummies
978-0-470-93067-0

Cooking & Entertaining

Cooking Basics
For Dummies, 4th Edition
978-0-470-91388-8

Wine For Dummies,
4th Edition
978-0-470-04579-4

Diet & Nutrition

Kettlebells For Dummies
978-0-470-59929-7

Nutrition For Dummies,
5th Edition
978-0-470-93231-5

Restaurant Calorie Counter
For Dummies,
2nd Edition
978-0-470-64405-8

Digital Photography

Digital SLR Cameras &
Photography For Dummies,
4th Edition
978-1-118-14489-3

Digital SLR Settings
& Shortcuts
For Dummies
978-0-470-91763-3

Photoshop Elements 10
For Dummies
978-1-118-10742-3

Gardening

Gardening Basics
For Dummies
978-0-470-03749-2

Vegetable Gardening
For Dummies,
2nd Edition
978-0-470-49870-5

Green/Sustainable

Raising Chickens
For Dummies
978-0-470-46544-8

Green Cleaning
For Dummies
978-0-470-39106-8

Health

Diabetes For Dummies,
3rd Edition
978-0-470-27086-8

Food Allergies
For Dummies
978-0-470-09584-3

Living Gluten-Free
For Dummies,
2nd Edition
978-0-470-58589-4

Hobbies

Beekeeping
For Dummies,
2nd Edition
978-0-470-43065-1

Chess For Dummies,
3rd Edition
978-1-118-01695-4

Drawing For Dummies,
2nd Edition
978-0-470-61842-4

eBay For Dummies,
7th Edition
978-1-118-09806-6

Knitting For Dummies,
2nd Edition
978-0-470-28747-7

Language & Foreign Language

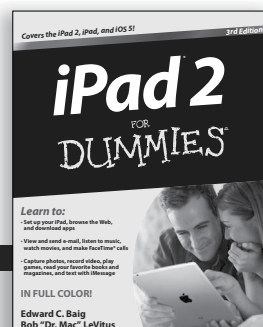
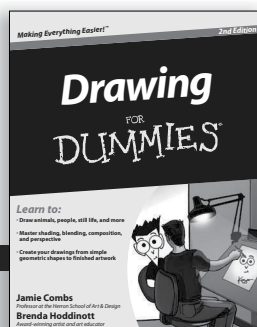
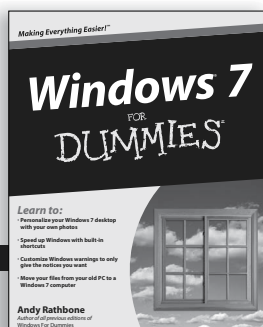
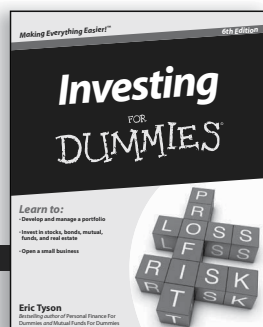
English Grammar
For Dummies,
2nd Edition
978-0-470-54664-2

French For Dummies,
2nd Edition
978-1-118-00464-7

German For Dummies,
2nd Edition
978-0-470-90101-4

Spanish Essentials
For Dummies
978-0-470-63751-7

Spanish For Dummies,
2nd Edition
978-0-470-87855-2



Available wherever books are sold. For more information or to order direct: U.S. customers visit www.dummies.com or call 1-877-762-2974.

U.K. customers visit www.wileyeurope.com or call (0) 1243 843291. Canadian customers visit www.wiley.ca or call 1-800-567-4797.

Connect with us online at www.facebook.com/fordummies or [@fordummies](https://twitter.com/fordummies)

www.it-ebooks.info

Math & Science

Algebra I For Dummies,
2nd Edition
978-0-470-55964-2

Biology For Dummies,
2nd Edition
978-0-470-59875-7

Chemistry For Dummies,
2nd Edition
978-1-1180-0730-3

Geometry For Dummies,
2nd Edition
978-0-470-08946-0

Pre-Algebra Essentials
For Dummies
978-0-470-61838-7

Microsoft Office

Excel 2010 For Dummies
978-0-470-48953-6

Office 2010 All-in-One
For Dummies
978-0-470-49748-7

Office 2011 for Mac
For Dummies
978-0-470-87869-9

Word 2010
For Dummies
978-0-470-48772-3

Music

Guitar For Dummies,
2nd Edition
978-0-7645-9904-0

Clarinet For Dummies
978-0-470-58477-4

iPod & iTunes
For Dummies,
9th Edition
978-1-118-13060-5

Pets

Cats For Dummies,
2nd Edition
978-0-7645-5275-5

Dogs All-in One
For Dummies
978-0470-52978-2

Saltwater Aquariums
For Dummies
978-0-470-06805-2

Religion & Inspiration

The Bible For Dummies
978-0-7645-5296-0

Catholicism For Dummies,
2nd Edition
978-1-118-07778-8

Spirituality For Dummies,
2nd Edition
978-0-470-19142-2

Self-Help & Relationships

Happiness For Dummies
978-0-470-28171-0

Overcoming Anxiety
For Dummies,
2nd Edition
978-0-470-57441-6

Seniors

Crosswords For Seniors
For Dummies
978-0-470-49157-7

iPad 2 For Seniors
For Dummies, 3rd Edition
978-1-118-17678-8

Laptops & Tablets
For Seniors For Dummies,
2nd Edition
978-1-118-09596-6

Smartphones & Tablets

BlackBerry For Dummies,
5th Edition
978-1-118-10035-6

Droid X2 For Dummies
978-1-118-14864-8

HTC ThunderBolt
For Dummies
978-1-118-07601-9

MOTOROLA XOOM
For Dummies
978-1-118-08835-7

Sports

Basketball For Dummies,
3rd Edition
978-1-118-07374-2

Football For Dummies,
2nd Edition
978-1-118-01261-1

Golf For Dummies,
4th Edition
978-0-470-88279-5

Test Prep

ACT For Dummies,
5th Edition
978-1-118-01259-8

ASVAB For Dummies,
3rd Edition
978-0-470-63760-9

The GRE Test For
Dummies, 7th Edition
978-0-470-00919-2

Police Officer Exam
For Dummies
978-0-470-88724-0

Series 7 Exam
For Dummies
978-0-470-09932-2

Web Development

HTML, CSS, & XHTML
For Dummies, 7th Edition
978-0-470-91659-9

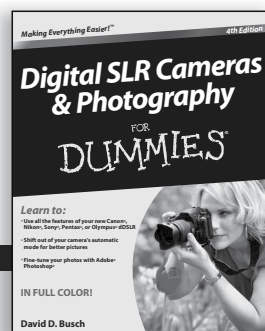
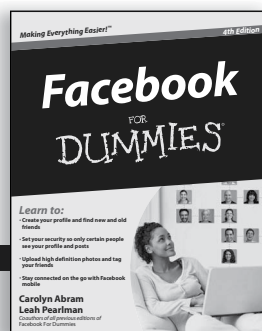
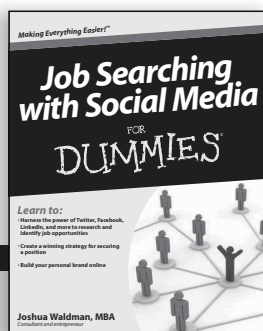
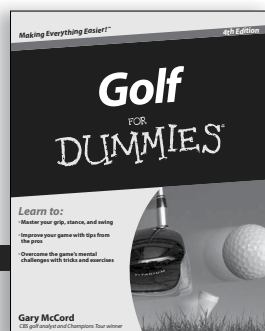
Drupal For Dummies,
2nd Edition
978-1-118-08348-2

Windows 7

Windows 7
For Dummies
978-0-470-49743-2

Windows 7
For Dummies,
Book + DVD Bundle
978-0-470-52398-8

Windows 7 All-in-One
For Dummies
978-0-470-48763-1



Available wherever books are sold. For more information or to order direct: U.S. customers visit www.dummies.com or call 1-877-762-2974. U.K. customers visit www.wileyeurope.com or call (0) 1243 843291. Canadian customers visit www.wiley.ca or call 1-800-567-4797.

Connect with us online at www.facebook.com/fordummies or @fordummies

www.it-ebooks.info

DUMMIES.COM[®]

Wherever you are
in life, Dummies
makes it easier.

From fashion to Facebook[®],
wine to Windows[®],
and everything in between,
Dummies makes it easier.



Visit us at Dummies.com and connect with us online at
www.facebook.com/fordummies or @fordummies
www.it-ebooks.info

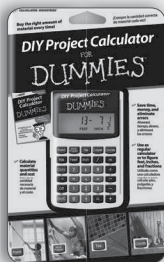
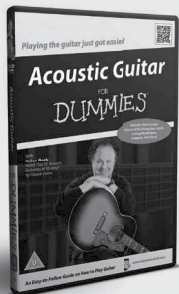


Dummies products make life easier!

- DIY
- Consumer Electronics
- Crafts
- Software
- Cookware
- Hobbies
- Videos
- Music
- Games
- and More!

For more information, go to **Dummies.com®** and search the store by category.

Connect with us online at www.facebook.com/fordummies or @fordummies



Mobile Apps FOR DUMMIES®

There's a Dummies App for This and That

With more than 200 million books in print and over 1,600 unique titles, Dummies is a global leader in how-to information. Now you can get the same great Dummies information in an App. With topics such as Wine, Spanish, Digital Photography, Certification, and more, you'll have instant access to the topics you need to know in a format you can trust.

To get information on all our Dummies apps, visit the following:

www.Dummies.com/go/mobile from your computer.

www.Dummies.com/go/iphone/apps from your phone.

