The Insider's Guide to Developing Applications in JavaScript using the Palm Mojo™ Framework

Palm® webOS™

O'REILLY®

Mitch Allen

Palm® webOS™

This is the official guide to building native JavaScript applications for Palm's new mobile operating system, Palm webOS. Written by Palm's software chief technology officer along with the Palm webOS development team, *Palm webOS* provides a complete tutorial on the design principles, architecture, UI, tools, and services necessary to develop webOS applications—including the Mojo JavaScript framework and Palm's SDK.

Palm webOS is designed to support a fast and superb user experience using established web standards, so if you're familiar with HTML, CSS, and JavaScript, you're ready to build applications for any webOS-based device, including the Palm Prē. You'll gain expertise, chapter by chapter, as you build a working mobile application through the course of this book. You'll also learn how to extend existing web apps to work with the new generation of mobile phones.

- Get a thorough overview of the webOS platform and architecture
- Understand the critical concepts for application design—what separates webOS from other web and mobile platforms
- Learn the details of the webOS SDK and development tools for building and testing mobile applications
- Examine best practices, important considerations, and guiding principles for developing with webOS and the Mojo framework

"Mitch Allen has been the driving force for webOS, and he realizes that developers want more than an assortment of simple 'Hello World' examples. Get this book, get the SDK, and start writing webOS applications."

—Greg Stevenson
Sierra Blanco Systems
preDevCamp Global Organizer

Mitch Allen, software CTO at Palm, Inc., is a member of the webOS engineering team and led the development team during the initial design stage.

INTRODUCTORY INTERMEDIATE ADVANCED

Some knowledge of HTML, CSS, and JavaScript is recommended.

O'REILLY®
oreilly.com

US $44.99 CAN $56.99
ISBN: 978-0-596-15525-4

Palm® webOS™

Palm® webOS™

Mitch Allen

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Palm® webOS™
by Mitch Allen

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Steven Weiss
Developmental Editor: Jeff Riley
Production Editor: Sumita Mukherji
Copyeditor: Amy Thomson
Proofreader: Teresa Barensfeld

Production Services: Molly Sharp
Indexer: Seth Maislin
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
August 2009: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-15525-4

[M]

1248713379

"That it all matters." He draws off his cigar and lets it dribble out all over his face. "You know every game starts with one pitch, and that pitch determines everything else in the game. The first pitch sets up the second, second the third. If Reyes gets in a 2-0 hole he throws the pitch that Young hits for a single. With a runner on first, he'll throw differently to German. And German will swing differently with a runner on first."

"And then the next time that Reyes sees Young, he'll choose his pitches based on the previous at-bat, and the scouting report, and every at-bat Young has ever had against Reyes. It all counts."

"But as much as everything counts," Jack says. "Is that you can have the right pitch, and have it knocked out. You can have the right swing and still screw up. That's the thing that makes this game great, is that everything counts so much, that the factors involved in that one pitch are almost infinite. So anything can happen."

—From the short story "Delay," Parker Zane Allen

Table of Contents

Foreword

Many of us remember that special sense of accomplishment, even excitement, as we got our very first program to run or web page to display. It was probably something very simple like a classic "Hello, World!" program, or a simple (often gaudy) web page using different styles and sizes of text. Steadily we learned and experimented more. In a short time, we soon had something actually useful. It was probably something like a tip calculator or a personal web page, or even a tip calculator on a web page. It was exciting because we realized the huge potential for doing much more with our new-found knowledge.

That first moment for me was more than 30 years ago. Like many others at that time, it was the start of a hobby that soon became a career. Programming and web development can be one of the most exciting and one of the most frustrating careers one can have. It is immensely rewarding to create something that benefits hundreds, thousands, even millions of people. At the same time, the pace at which things change can really wear one down. I've now seen, used, and discarded so many cool technologies it is rare for any of them to get me really excited. Prior to Palm's announcement of webOS, I can only think of two times a new technology generated a similar visceral excitement as I had when I first learned to program. (Forth and Delphi, if you are curious. Search for those terms and my name to see what made them special.) webOS has rekindled those feelings all over again.

I, like many other smartphone users and developers, was very curious to see what Palm would show on January 8th, 2009, at the Consumer Electronics Show (CES) in Las Vegas. Truthfully, I was not expecting much. I just wanted to give Palm one more chance before switching from my Treō to another smartphone. However, that day the Palm Prē became the star of CES. Nothing else introduced that week came close to generating the buzz of the Prē. But while the Prē was nice, there were other smartphones that looked cool, had a great UI, and even did multitasking. Even after the great demos, for me the Prē was just a "take it or leave it" proposition. What got my interest was webOS, the underlying technology that made those great demos possible. I had to learn more, and the more I learned, the more excited I got.

For me, the best way to force myself to learn is to teach. About 15 years ago, when I last got this excited, I started a user group for Delphi months before I actually had a

copy of it (I was tight on funds at the time). Continuing the tradition, I started a Meetup for webOS and the Prē on February 18th. I volunteered to be an organizer for preDevCamp, a world-wide day for developers to share and learn about webOS. I also volunteered to talk about webOS development at a regional CodeCamp. At the same time, I started developing for webOS. All this was happening long before the webOS software development kit (SDK) was even announced. Granted, there was little public information, but everything I gleaned confirmed my initial feeling that webOS was something to master.

How could I develop or talk about webOS when there wasn't even an SDK? That is the beauty of webOS. It is a very unique blend of existing technologies with some special Mojo provided by Palm. It allows one to develop native-style applications like you would find running on a traditional computer using web-based technologies such as HTML, JavaScript, and CSS. My group started working on the JavaScript parts of our applications that seemed be the most portable and incorporated new knowledge as it became available. When you think about it, the Internet is so pervasive that most developers today already have considerable experience with these web-based technologies. Native webOS applications are launched via *index.html*. Sound familiar? Most developers seem to come up to speed on webOS quickly. It helps that webOS does a lot of the tricky stuff for you automatically.

Not long after I started my journey on the road to webOS, O'Reilly announced that a Rough Cuts version of this webOS book was available. Rough Cuts is a great program that allows you to read chapters of a book as it is being written. I immediately got "copies" of the book for myself and my developers. As chapters became available, I would print them out and study them. This period also saw the appearance of websites, forums, and IRC channels that were dedicated to webOS. preDevCamp also had organizers in about 75 cities with close to 1,000 developers signed up to attend. I was by no means alone in my desire for the webOS SDK and a Palm Prē to test my applications. When Palm finally announced they were taking applications for the SDK beta, I remember Palm posting that they got about a gazillion applications in just a couple of days.

Of course, I also applied and was fortunate enough to get accepted into the program sooner than many. It was like getting a new bike for Christmas, only you couldn't ride it in public or tell anyone else about it. Fortunately, this book, the webOS SDK, and Prē phones are all now readily available—you don't have to wait. Since the SDK runs on Linux, Mac OS X, and Windows, this means you can probably even use your existing development environment for webOS development as well. There are also several plug-ins to automate webOS development in popular IDEs.

So, what did I find so special about webOS? It is the almost elegant way in which it solves a lot of issues surrounding the current direction of application development in general—and mobile application development in particular—with its unique integration of native and web-based computing. Chapter 1, *Overview of webOS*, has a lot more specifics. As I stated earlier, it smoothly leverages the latest in open technologies and

standards like Linux, the WebKit engine, HTML 5, Javascript, and CSS3, to bring mobile device development to all programmers. It has the potential to have the same impact for mobile applications as Visual Basic and Delphi did for win32 applications. The rapid appearance of homebrew applications shortly after the Palm Prē was released demonstrates the relative ease of development. The richness of some the applications in Palm's Application Catalog demonstrates that webOS is fully capable of supporting sophisticated software.

Another thing I like about webOS is that it tends to encapsulate best practices, such as using the Model-View-Controller pattern as a natural part of webOS development. By carefully exposing services and APIs, developers have ready access to powerful features and yet still allow each application to play nicely with others. Although I expect more low-level access in the future, the design of webOS is such that low level features can easily be wrapped and exposed through webOS's Mojo framework.

Developing applications is pointless if nobody wants to use them. The Mojo framework provides a smart and polished user interface with lots of useful widgets. The card metaphor for switching between applications and the notification system is currently without peer in the realm of smartphones. There is another advantage that webOS provides users: the ability for applications to dynamically interact with each other and with network services in a clean and consistent fashion. For example, contact information is available to other applications, not just the contact application. Using Palm's Synergy, Contact information can be automatically updated from a variety of sources over the network just like web mashups are able to do. However, as a native application, the latest information is still available, even if the network is not. Regardless of the circumstances, webOS lets applications "just work" as the user expects.

Since webOS is a new platform, it has lots of room to grow. Palm emphases that the Prē is/was just the first of many devices on which webOS can run. This means more devices, more services, and more APIs are planned for the future. Each iteration will spawn a need for new applications to exploit new features. webOS has the potential for keeping developers very busy for many, many years.

I could go on, but Mitch already gives a fine introduction to webOS in the first chapter of his book. Mitch is uniquely qualified to be the author of the first book on the topic. He has been doing software development for a long time, especially on mobile platforms. As software CTO at Palm, he has been the driving force for webOS. In writing this book, he realizes that developers want more than an assortment of simple "Hello World" examples—they want to be able to develop real working applications.

Mitch gradually introduces the reader to webOS while building a fully functional RSS news reader. Each step of the development process is fully explained in tutorial fashion. The reader also learns best practices for webOS development along the way. This book does not try to pad itself with reference information readily available in the SDK. I also like the fact that Mitch points out current limitations in webOS so developers can work around them to provide a positive user experience.

Lastly, I would like to share how committed Palm is to developers. There have been rough spots. The majority of my posts in the developer forums have been and will be regarding issues I have with Palm and webOS. Even so, I'm actually amazed at how open, helpful, and accessible Palm has been. Palm's webOS team frequents the forums and answers questions directly. They totally get that their success is intimately tied to an active, prolific community of webOS developers.

That said, get this book, get the SDK, and start writing webOS applications. I hope you enjoy it as much as my team and I do.

—Greg Stevenson
Sierra Blanco Systems
preDevCamp Global Organizer

Preface

It would be difficult to miss the revolution in computing that is happening around us. While the Internet has been a viable commercial environment for almost two decades and mobile phones commonplace for years beyond that, the last two years have seen incredible developments as these two movements have converged and begun to accelerate together forming the next generation of computing. Since the introduction of the Apple iPhone, our expectations of what we should be able to do with a phone has grown by magnitudes. There has been a rush to provide applications and services, operating systems, and hardware in an attempt to fulfill these expectations.

The world of application development is in transition with web-based applications and services becoming the dominant development model:

- Increasingly powerful web applications are now providing solutions previously addressed only with embedded or desktop applications.
- Web developers have assumed the leadership in software application innovation.
- Mobile users have strong preferences toward web brands and aren't willing to accept equivalent solutions—only the authentic experience provided by the preferred brands will do.
- Web services are providing easy-to-use building blocks and tools to allow developers to leverage those web services through mashups and specialized applications.
- Web applications can be built faster and easier than embedded applications; they are easier to deploy, update, and maintain, resulting in a lower development cost.

Where once the client operating systems provided the complete platform that application developers leveraged to deliver their solutions, the Web itself is emerging as the platform, and client operating systems are becoming a means to access the web platform. Those who can deliver a superior user interface (UI) on highly optimized hardware while leveraging web services and applications stand to gain.

Mobile Web Challenges

The challenge for client OS providers is far greater than simply delivering a fast, fully featured web browser on a phone. The classic web browser navigation model works poorly on a phone (in fact, some would argue that it's poor even on a desktop computer).

Mobile users are, well, mobile. They are usually in motion, walking, driving, or occupied with something other than their phones. Launching a browser each time you want something on the Web—wading through multiple pages to get to the right spot—is tedious, distracting, and slow.

Web pages have their own UI models, with navigation and controls separate from and frequently inferior to those of the device they are displayed on. Often, the only option is to walk links. Menus, selectors, text editors, and other UI tools that enable rapid user interaction in native applications on the same device can't be used within the web browser. Launching web pages from bookmarks or moving between web pages usually involves a completely separate UI model from that used to launch native applications and generally requires invoking the browser before anything else, adding at least one extra step to most actions.

In addition, web users are forced to initiate all interactions. They must make a request and wait for it to be fulfilled. It is clearly more effective for applications to monitor external events and prompt the user only when something of interest occurs. Ajax and web applications have made a big improvement by handling user input on the client and providing some level of dynamic user interface, but even these applications can't employ commonly used techniques such as background execution, user alerts, and notifications.

The truth is that despite the hype, a phone with just a fast web browser is still not a truly smart phone.

To fully realize the mobile Web, a new application model is needed, one that retains the strengths of web development, but with the type of access and power that has been available to native, mobile applications for years.

Palm webOS

Palm addresses these challenges with its next generation operating system, Palm webOS. Palm webOS is based upon an innovative design that integrates a window-based modern operating system with a web technology runtime that allows you to build applications using common web languages and tools, without the restriction of working within a web browser. The application model is based on an integrated web runtime and the Mojo framework, a JavaScript framework with powerful UI services, local storage, and methods to access application, cloud, and system services.

Applications are built using JavaScript, HTML, and CSS, and while similar to web applications, webOS applications are actually native applications. This application model allows you to use the same languages and tools to build powerful mobile applications that you use to build web content.

While Palm webOS is the first to provide this integrated model in a broadly available computing platform, it's not likely to be the last. There is growing interest in supporting standard APIs within web platforms, such as those in the proposed HTML 5 standard. It seems likely that in time there will be broad support for this development paradigm across all types of hardware and systems.

The Mobile Web Is the Web

We are still in the early stages of application development on mobile devices. Until very recently all mobile applications were designed to work alongside the PC. Some mobile applications, like Palm's classic PDA applications, were specifically created with the PC in mind, and today's most popular media solutions continue to rely on the PC for content delivery and storage. Other applications are essentially desktop applications ported to a phone, like many of the wireless email solutions. We are just beginning to see applications that are completely designed and optimized for the wireless mobile user.

Phones are far more personal than PCs; they are almost always with the user, even if they're not being engaged by the user. With phones, an event-driven model is more appropriate, and mobile applications can best leverage web and device services in useful mashups. Applications that notify users of upcoming calendar events or incoming emails are common, but webOS applications can notify users of traffic on the route to their next appointment, or monitors social network feeds. A movie guide allows users to find movies within the immediate vicinity, purchase tickets, get directions, and set a reminder for the movie time.

Applications designed for the mobile Web are different than applications built before now, and they require a different type of platform. This book explores how Palm webOS is providing that type of platform and shows you how to build those next generation applications and with them, the new Web—the mobile Web.

About This Book

The book was conceived after the architecture and core design of Palm webOS and the Mojo framework had been completed, but while the team was fully engaged with implementation of the application runtime, the Mojo framework, and while many of the core applications were still in prototype form. As a result, the book has been written at the same time as the software, which makes it fresh but raw information.

The project changed dramatically soon after it began. Originally, I saw my role more as that of an editor. I expected to pull together the engineering and developer documentation and write a heavily annotated reference book that would provide a guided tutorial to webOS and Mojo. After the first chapter, though, it became clear that I would have to write a specific application that would use a significant portion of the API and document my experience. I scaled back the outline from a reference book to more of an application-centered guidebook focused on an RSS reader application called News.

This book is not a comprehensive reference, but more of a guided tutorial. It covers all the basics for creating and building an application and for using UI widgets, storage, and services. It includes specific chapters on building background applications, a huge topic of its own, and on specialty topics of building localized applications and on styling. You will want to augment this book with SDK documentation or other reference material as it becomes available.

You don't need to be an expert, but you will need some basic knowledge of JavaScript, HTML, and CSS to follow the examples presented here. This book is intended to provide an introduction to webOS and building webOS applications, but should not be used as a guide to writing JavaScript code. In fact, I have to warn you that I wrote my first JavaScript code as part of writing this book and it's very likely that you will see several examples of not-so-good JavaScript in here.

So please read this book to learn how to write great webOS applications, but look for your JavaScript guidance in other sources such as Douglas Crockford's outstanding *JavaScript: The Good Parts* (O'Reilly) or the comprehensive *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly).

Conventions Used in This Book

The following typographical conventions are used in this book:

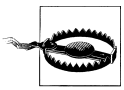
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, properties, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon signifies a warning or caution.

Using Code Examples

The code in this book was written by an employee of Palm, Inc. and is Palm's intellectual property. If you are interested in using this code, it is important for you to review Palm's software development kit (SDK) license, which can be found at *http://developer.palm.com/termsofservice.html*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9780596155254

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://oreilly.com

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com/*.

Acknowledgments

Many people contributed to the subject of this book, and in several cases contributed directly to the material that you are about to read. Projects like this are built upon a foundation created by dozens if not hundreds of people; it is impossible to properly acknowledge everyone's contribution.

To start with: the entire webOS product team, particularly the Apps SW & Services, System Software, and Product Engineering teams, have accomplished an incredible feat with the release of the Palm Prē and webOS. This is the finest and most committed group of people that I have ever worked with. The past year was one in which it seemed there was a new development everyday that would leave me amazed, and over time it was clear that the team was on the verge of something truly special.

Despite the incredible workload that everyone was under, everyone on the team gave generously of his or her time. I am particularly grateful to Rob Tsuk, a principal webOS architect and developer of critical parts of the Mojo framework, for all of his help throughout this book project. Rob spent hours at some very inconvenient times, patiently answering questions and educating me on many aspects of web technology and details of Palm webOS and the Mojo framework. And if that weren't enough, Rob reviewed every chapter in the book and did his best to keep me on track throughout.

Special thanks to Justin Tulloss, whose candid comments had a huge impact on the structure and shape of the book and for keeping me honest with his very critical eye. Without Daniel Shiplacoff there would not have been a Chapter 7 (nor would we have a considerable part of the webOS UI); Daniel's tutoring on styling and CSS was invaluable. Thanks also to Steven Feaster, who along with Justin and Rob, read every page of the book and diligently corrected my all too frequent coding errors and technical mistakes. And thanks to Craig Upson, for providing the original News prototype and some valuable SDK and tools feedback very early on.

Many, many other people at Palm reviewed different parts of the book and made direct contributions in big and small ways. It's hard to acknowledge everyone, but among those who provided direct assistance are Greg Simon, Matias Duarte, Michael Abbott,

Jesse Donaldson, Renchi Raju, Jon Rubinstein, Mike Bell, Paul Cousineau, Gray Norton, Andy Grignon, Geoff Schuller, Rich Dellinger, Wesley Yun, Joe Paley, Rik Sagar, Mindy Pereira, Charlie Won, Kiran Prasad, Mike Rizkalla, Jeremy Lyon, Neeta Srivastava, Peter Conrad, Ed Wei, Doug Luftman, Mark Kahn, Susan Juhola, Melissa Cha, Aaron Hyde and Edwin Hoogerbeets.

We had several early and passionate developers, but none more so than the team at Pivotal Labs. Pivotal's CTO, Ian McFarland, is the book's technical reviewer and provided countless insightful and critical suggestions. I feel very fortunate to have had his guidance and support. Also, a shout out to Christian Sepulveda, Davis Frank, Rajan Agaskar, and the rest of the Pivots whose relentless encouragement and high expectations have made the Mojo SDK a far better product than it would have been otherwise.

The O'Reilly team has made me feel as if none of my demands or any of the numerous schedule and scope changes caused them any trouble at all. I know that behind the scenes they scrambled and adapted like mad to maintain that illusion for me and still meet their obligations. Thanks in particular to Molly Sharp and Steve Weiss for consistently going above and beyond, and to Sumita Mukherji, Jeff Riley, Amy Thomson, and Rachel Monaghan for their support.

If there's a failing here, it's all mine. The material that I had to work with, and the quality of the team and the support that they gave, is more than anyone could expect.

CHAPTER 1
Overview of webOS

Palm webOS is Palm's next generation operating system. Designed around an incredibly fast and beautiful user experience and optimized for the multitasking user, webOS integrates the power of a window-based operating system with the simplicity of a browser. Applications are built using standard web technologies and languages, but have access to device-based services and data.

Palm webOS is designed to run on a variety of hardware with different screen sizes, resolutions, and orientations, with or without keyboards, and works best with a touch panel, though it doesn't require one. Because the user interface (UI) and application model are built on a web browser runtime, the range of suitable hardware platforms is quite wide, requiring only a CPU, some memory, a wireless data connection, a display, and a means for interacting with the UI and entering text.

You can think of webOS applications as native applications, but built from the same standard HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript that you'd use to develop web applications. Palm has extended the standard web development environment through a JavaScript framework that provides a toolkit of UI widgets and access to selected device hardware and services.

The user experience is optimized for launching and managing multiple applications at once. Palm webOS is designed around multitasking and makes it utterly simple to run background applications, to switch between applications in a single step, and to easily handle interruptions and events without losing context.

You will build webOS applications with common web development tools following typical design and implementation practices for Ajax applications, but your webOS applications will be installed and run directly on the device, just as you are used to doing with native applications.

Application Model

As shown in Figure 1-1, the original Palm OS has a typical native application model, as do many of the popular mobile operating systems. Under this model the application's data, logic, and UI are integrated within an executable installed on the native operating system, with direct access to the operating system's services and data.

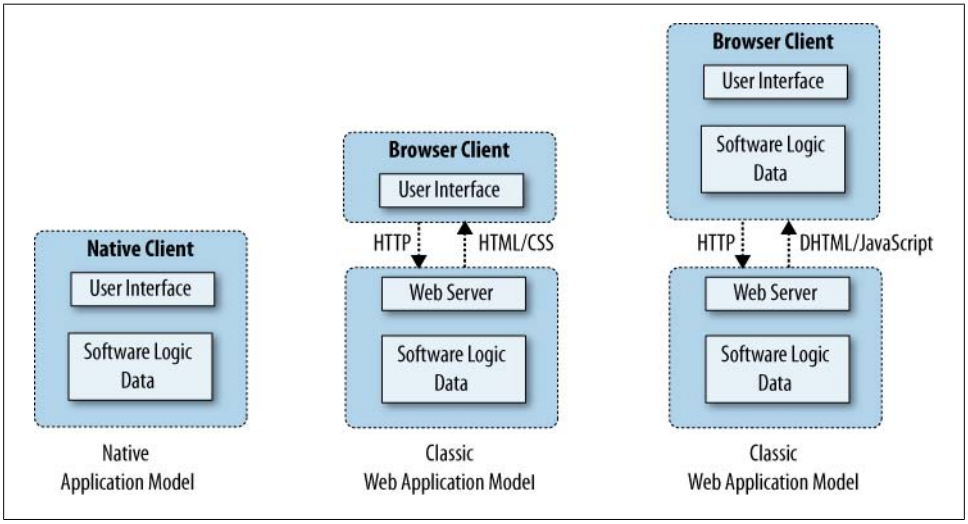


Figure 1-1. Native and web application models

Classic web applications are basic HTML-based applications that submit an HTTP (Hypertext Transfer Protocol) request to a web server after every user action and wait for a response before displaying an updated HTML page. More common in recent years are Ajax applications, which handle many user interactions directly and make web server requests asynchronously. Ajax applications are able to deliver a richer and more responsive user experience. Some of the best examples of this richer experience are map applications, which allow users to pan and zoom while asynchronously retrieving needed tiles from the web server.

Web applications have some significant advantages over embedded applications. They are easier to deploy and update using the same search and access techniques as web pages use. Developing web applications is far easier, too: the simplicity of the languages and tools, particularly for building *connected* applications, allows developers and designers to be more productive. Connected applications, or applications that leverage dynamic data or web services, are becoming the predominant form for modern applications.

The webOS application model combines the ease of development and maintenance of a web application with the deep integration available to native applications, significantly advancing the mobile user experience while keeping application development simple.

Application Framework and OS

Through Palm's application framework, applications can embed UI widgets with sophisticated editing, navigation, and display features, enabling more sophisticated application UI. The framework also includes event handling, notification services, and a multitasking model. Applications can run in the background, managing data, events, and services behind the scenes while engaging the user when needed.

You can create and manage your own persistent data using HTML 5 storage functions and you can access data from some of the webOS core applications, such as Contacts and Calendar. You also have access to some basic system services, most of which are device-resident, such as Location services and Accelerometer data, along with some web services, such as Publish and Subscribe.

Architecturally, Palm webOS is an embedded Linux operating system that hosts a custom UI System Manager anchored by the open source WebKit core. The System Manager provides a full range of system UI features including navigation, application launching and lifecycle management, event management and notifications, system status, local and web searches, and renders application HTML/CSS/JavaScript code.

You don't need to build a webOS application to make your web content accessible to webOS devices. Palm webOS maintains a separate instance of WebKit, which supports a browser application to handle standard web pages and browser-based web applications. While it's expected that more and more web content and services will be delivered as webOS applications, there are millions of legacy websites and other information sources that will continue to be presented in ways best viewed with a classic web browser. Palm webOS supports traditional web content very competitively.

Beyond the operating system, webOS includes a number of core applications: Contacts, Calendar, Tasks, Memos, Phone, Web, Email, and Messaging. Other applications are included in the initial release, such as a Camera, Photos, Music, Videos, and Maps, but the full application suite for a given webOS device will vary depending on the model and carrier configuration.

User Interface

Palm webOS is designed for mobile, battery-operated devices with limited but variable screen sizes, and a touch-driven UI. User interaction is centered on one application at a time, though applications, once launched, continue to run until closed even when

moved out of the foreground view. There is a rich notification system enabling applications to subtly inform or directly engage the user at the application's discretion.

Navigation

Navigation is based on a few simple *gestures* with optional extensions that create a rich vocabulary of commands to drive powerful navigation and editing features. To start with, though, all you need to know is:

tap (act on the indicated object)
: Commonly in a view that contains clusters or lists of items, tapping reveals information contained in an item. This can be thought of as an *open* function, which changes the nature or context of the view to be about the selected item exclusively. Alternately, a tap will change an object's state such as setting a check box or selecting an object.

back (the inverse of open)
: This feature looks like the opposite of a tap: the item compresses down to its summary in the containing context where it belongs. Typically, it reverses a view transition, as going from a child view to a parent view.

scroll (flick and quick drags)
: Used to quickly navigate lists and other views.

Beyond this, you can learn to pan, zoom, drag and drop, switch applications, switch views, search, filter lists, and launch applications. But to begin with, only these three gestures are needed to use a webOS device.

Launcher

When a user turns on a webOS device, the screen displays the selected wallpaper image with the *status bar* across the top of the screen and, hovering near the bottom, the *Quick Launch bar*. The Quick Launch bar is used to start up favorite applications or to bring up the *Launcher* for access to all applications on the device. From this view, users can initiate a search simply by typing the search string; searches can be performed on contacts, installed applications, or to start a web search. Figure 1-2 shows both the Quick Launch bar and the Launcher.

The launched application takes over the available screen becoming the *foreground* application; the application's view replaces the wallpaper image and the Quick Launch bar is dismissed. The status bar remains and is always visible except in full screen mode, which is available to applications such as the video player or others that request it. The application launch sequence is fluid and smooth, as you will see with all webOS transitions.

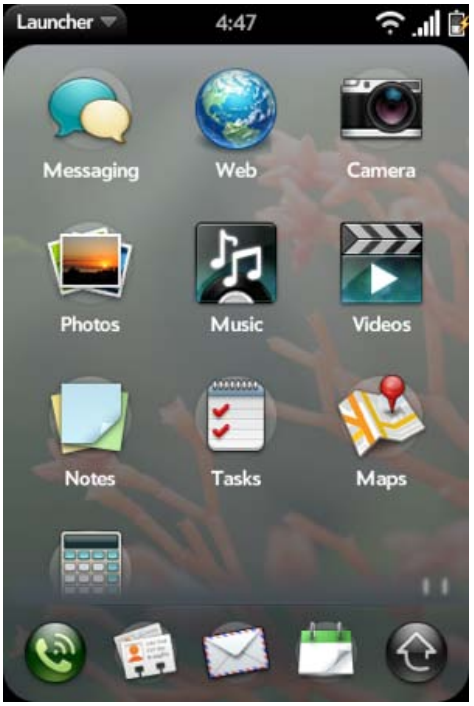


Figure 1-2. Quick Launch bar and Launcher

Card View

Figure 1-3 shows an application's main view, in this case the Email application's folder view. The main view includes UI elements that make up the basic email application. The inbox view displays specific folders, which users can select to open a new card with a detail view of the messages contained within the selected folder. At the bottom, floating icons represent menu items. A tap to a menu icon will typically reveal another view associated with that menu action, a submenu, or a dialog box.

Running one application at a time, or performing one *activity* at a time, can be terribly restrictive and inefficient. Palm webOS makes it easy to work on more than one thing at a time. Swiping up, from the bottom of the display, brings up a new view, the *Card view*, an example of which is shown in Figure 1-4. From the Card view, users can switch to another activity simply by scrolling to and tapping the card representing that activity. Users can also launch another application from the Quick Launch bar.

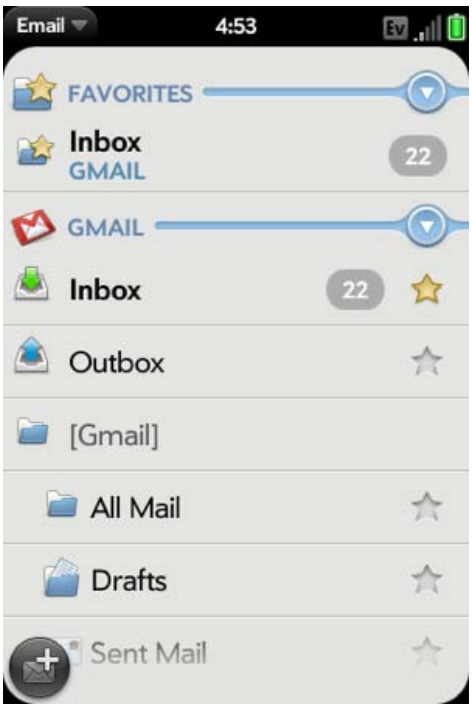


Figure 1-3. Email application

The Card view was inspired by the way people handle a deck of cards. You can fan the cards out to see which card is where, and within the deck of cards, you can select or remove any single card with a simple gesture, or move it to a new location by slipping it between adjacent cards. Users can manipulate the webOS Card view in similar ways by scrolling through the cards, selecting and flicking cards off the top to remove them, or selecting and dragging a card to a new location.

The term *activity* needs further explanation. In most applications, users will, by design, work on one activity at a time. However, with some applications, it is more natural to work on several activities in parallel. A common email activity is writing a new email message, but in the middle of writing that message, the user may want to return to the inbox to look up some information in another message or perhaps read an urgent message that has just arrived.

Figure 1-4. Card view with email and other applications

With a webOS device, the draft email message has its own card, separate from the email inbox card. In fact, users can have as many draft email messages as they need, each in their own cards; each is considered a separate activity and is independently accessible. Switching between email messages is as simple as switching between applications, and the data is safe, as it is saved automatically by the Email application. Figure 1-5 shows the Card view with the email application's inbox card and a draft email compose card.

Notifications and the Dashboard

What happens to the foreground application when the user switches to a new application? The previous application is not closed, but continues to run as a *background* application. Background applications can get events, read and write data, access services, repaint themselves, and are generally not restricted other than to run at a lower priority than the foreground application.

Figure 1-5. Card view with email application and new email message draft

To allow background applications to communicate with the user, Palm provides a notification system with two types of notifications:

Pop-up
: Nonmodal dialog boxes that have a fixed height and include at least one button to dismiss the dialog box. Pop-up notifications interrupt the user and are appropriate for incoming phone calls, calendar alarms, navigation notifications, and other time-sensitive or urgent messages. Users are forced to take action with pop-ups or explicitly clear them, but since they are not modal, users are not required to respond immediately.

Banner
: A nonmodal icon and single unstyled string of text. Banner notifications are displayed along the bottom of the screen within the *Notification bar*, which sits just below the application window in what is called *negative space* since it is outside of the card's window. After being displayed, banner notifications can leave a *summary icon* in the Notification bar as a reminder to the user. Figure 1-6 shows an example of a banner notification and the summary icons are shown in Figure 1-7, indicating that the music player is active and that there is an upcoming calendar event and new messages.

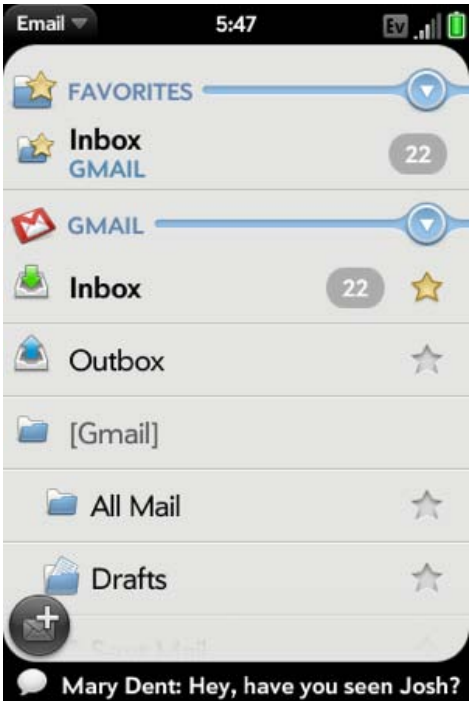


Figure 1-6. Banner notification

At any time, the user can tap the Notification bar, which brings up the *Dashboard view*, shown in Figure 1-8. Notifications that are not cleared should display their current statuses within a dashboard *summary*.

A dashboard summary is more than just a history of a notification—it is a dynamic view that allows any background application to display ambient information or status. For example, the Calendar application always displays the next event on the calendar even before the event notification has been issued. In Figure 1-8, the Music application shows the current song along with playback controls that you can manipulate to pause the music or change the selection.

Dashboard applications are those that can be completely served through the dashboard, as their entire purpose is to monitor and present information. For example, a weather application could display the current weather for a targeted location in a dashboard without having a Card view at all.

The Notification bar and Dashboard view manage messages and events, keeping users abreast of changes in information without interrupting their current activities. It may help to think of the Dashboard view as an event-driven model, while the Card view

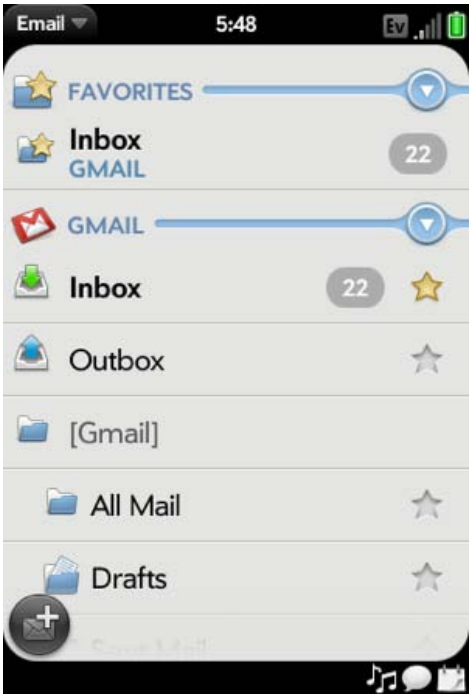


Figure 1-7. Summary icons

provides the user with task-oriented navigation tools. The combination enables the user to quickly track and access the information he needs when he needs it.

User Interface Principles

There are some foundational principles or values that support the overall webOS user experience. You can exploit these principles to more deeply integrate your application into the overall user experience. You can rely on the framework to provide most of what is required at an implementation level, but you should keep in mind these key principles when designing your application:

- *Physical metaphors* are reinforced through direct interaction with application objects and features, instant response to actions, and smooth display and object transitions with physics-based scrolling and other movement. For example, users delete objects by flicking them off the screen and they can edit in place without auxiliary dialog boxes or scenes.
- Maintain a *sense of place* with repeatable actions, reversible actions, stable object placement, and visual transitions that take the user from one place to the next.

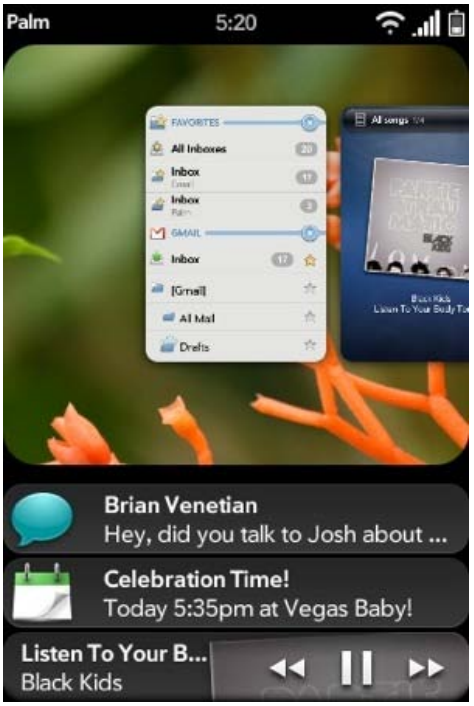


Figure 1-8. Dashboard

- Always display *up-to-date data*, which requires both pushing and pulling the latest data to and from the device so that the user is never looking at stale data when more recent data is available. This also means managing on-device caches so that when the device is out of coverage or otherwise offline, the user has access to the last data received.
- Palm webOS is *fast and simple* to use. All features should be designed for instant response, easy for novices to learn and efficient for experienced users.
- *Minimize the steps* for all common functions. Put frequently executed commands on the screen and less frequently executed commands in the menus. Avoid preferences and settings where possible. If you must use them, keep them minimal.
- *Don't block the user*. Don't use a modal control when the same function can be carried out nonmodally.
- *Be consistent*. Help users learn new tasks and features by leveraging what they have already learned.

Palm applications have always been built around a direct interaction model, where the user touches the screen to select, navigate, and edit. Palm webOS applications have a significantly expanded vocabulary for interaction, but they start at the same place. Your

application design should be centered on direct interaction, with clear and distinguishable targets. The platform will provide physical metaphors through display and navigation, but applications need to extend the metaphor with instantaneous response to user actions, as well as smoothly transitioning display changes and object transitions.

You can find a lot more on the UI guidelines and design information in the Palm Mojo SDK. We'll touch on the principles and reference standard style guidelines in the next few chapters, but will not be covering this topic in depth.

Mojo Application Framework

A webOS application is based on standard HTML, CSS, and JavaScript, but the application model is not like the web application model. Applications are run within the UI System Manager. The UI System Manager is an application runtime built on WebKit, an open source web browser engine, to render the display, assist with events, and handle JavaScript.

The webOS APIs (application programming interfaces) are delivered as a JavaScript framework, called the Mojo framework. Mojo includes common application-level functions, a suite of powerful UI widgets, access to local storage and various application, and cloud and system services. To build full-featured webOS applications, many developers will also leverage HTML 5 features such as video/audio tagging and database functions. Although not formally part of the framework, the *Prototype* JavaScript framework is bundled with Mojo to assist with event and DOM (Document Object Model) handling among many other great features.

The framework provides a specific structure for applications that is based on the Model-View-Controller (MVC) architectural pattern. This allows for better separation of business logic, data, and presentation. Following the conventions reduces complexity; each component of an application has a defined format and location that the framework knows how to handle by default.

You will get a more extensive overview of Mojo in Chapter 2, and you'll get details on widgets, services, and styles starting in Chapter 3. For now, you should know that the framework includes:

- *Application structure*, such as controllers, views, models, events, storage, notifications, and logging
- *UI widgets*, including simple single-function widgets, complex multifunction widgets, and integrated media viewers
- *Services*, including access to application data and cross-app launching, location services, cloud services, and accelerometer data

Anatomy of a webOS Application

Outside of the built-in applications, webOS applications are deployed over the Web. They are located in Palm's *App Catalog*, an application distribution service that is built into all webOS devices and is available to all registered developers. The basic lifecycle stages are illustrated in Figure 1-9.

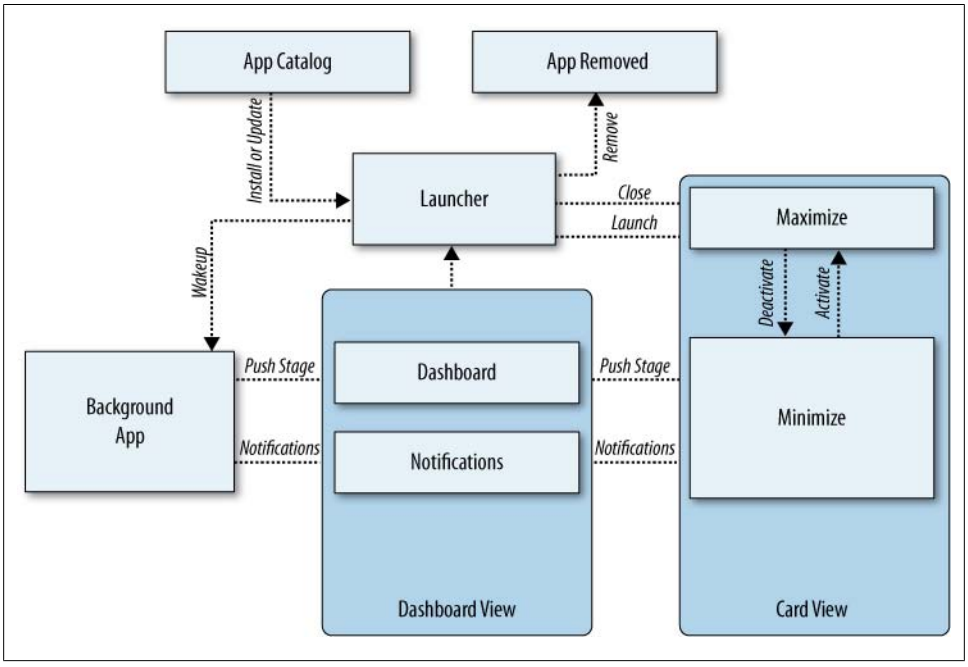


Figure 1-9. Application lifecycle stages

When you download an application to the device, it will be installed, provided it has been validly signed. After installation the application will appear in the Launcher.

Users can launch other applications from the Launcher into the foreground and switch them between the foreground and background. Each of these state changes (launch, deactivate, activate, close) is indicated by one or more events. Applications are able to post notifications and optionally maintain a dashboard while in the background.

If it is a dashboard application, a card is not required; the application uses a dashboard and notifications to communicate with the user. Dashboard applications typically include a simple card-based preferences scene to initiate the application and configure its settings. Every application requires at least one visible window at all times (either a card or dashboard).

Applications are updated periodically by the system. If running, the application is closed, then the new version is installed and then launched. There isn't an update event, so the application needs to reconcile changes after installation, including data migration or other compatibility needs.

The user can remove an application and its data from the device. When the user attempts to delete an application, the system will stop the application if needed and remove its components from the device. This includes removing it from the launcher and any local application data, plus any data added to the Palm application databases such as Contacts or Calendar data.

Stages and scenes

Palm's user experience architecture provides for a greater degree of application scope than is normally considered in a web application. To support this and specific functions of the framework, Palm has introduced a structure for webOS applications built around *stages* and *scenes*.

A *stage* is similar to a conventional HTML window or browser tab. Applications can have one or more stages, but typically the primary stage will correspond to the application's card. Other stages might include a dashboard, a pop-up notification, or secondary cards for handling specific activities within the application. Refer to email as an example of a multistage application, where the main card holds the account lists and inbox, and displays the email contents, but new email messages are composed in a separate card to allow for switching between compose and other email activities. Each card is a separate stage but still part of a single application.

Scenes are mutually exclusive views of the application within a stage. Most applications will provide a number of different kinds of scenes within the same stage, but some very simple applications (such as the Calculator) will have just a single scene. An application must have at least one scene, supported by a controller, a JavaScript object referred to as a *scene assistant*, and a *scene view*, which is a segment of HTML representing the layout of the scene.

Most applications will have multiple scenes. You will need to specifically activate (or *push*) the current scene into the view and *pop* a scene when it's no longer needed. Typically, a new scene is pushed after a user action, such as a tap on a UI widget and an old scene is popped when the user gestures back.

As the terms imply, scenes are managed like a stack with new scenes pushed onto and off of the stack with the last scene on the stack visible in the view. Mojo manages the *scene stack*, but you will need to direct the action through provided functions and respond to UI events that trigger scene transitions. Mojo has a number of stage controller functions specifically designed to assist you, and are described in Chapter 2, *Application Basics*, and Chapter 3, *Widgets*.

Application lifecycle

Palm webOS applications are required to use directory and file structure conventions to enable the framework to run the applications without complex configuration files. At the top level the application must have an *appinfo.json* object, providing the framework with the essential information needed to install and load the application. In addition, all applications will have an *index.html* file, an *icon.png* for the application's Launcher icon, and an *app* folder, which provides a directory structure for assistants and views.

By convention, all of an application's images, other JavaScript, and application-specific CSS should be contained in folders named *images*, *javascripts*, and *stylesheets*, respectively. This is not required, but makes it simpler to understand the application's structure.

Launching a webOS application starts with loading the *index.html* file and any referenced stylesheets and JavaScript files, as would be done with any web application or web page. However, the framework intervenes after the loading operations and invokes the application, stage, and scene assistants to perform the application's setup functions and to activate the first scene. From this point, the application is driven either by user actions or dynamic data.

Significantly, this organizational model makes it possible for you to build an application that will manage multiple activities that will be in different states (active, monitoring, and background) at the same time.

Applications can range from the simple to the complex:

- Single-scene applications, such as a Calculator, which the user can launch, interact with, and then set aside or close.
- Dashboard applications, such as traffic alert application that only prompts with notifications when there is a traffic event and whose settings are controlled by its dashboard.
- Connected applications like a social networking application, which provides a card for interaction or viewing and a dashboard to provide status.
- Complex multistage applications like email, which can have an inbox card, one or more compose cards, and a dashboard showing email status. When all the cards are closed, the email application will run in the background to continue to sync email messages and post notifications as new messages arrive.

Events

Palm webOS supports the standard DOM Level 2 event model. For DOM events, you can use conventional techniques to set up listeners for any of the supported events and assign event handlers in your JavaScript code.

There are a number of custom events for UI widgets. These are covered in more detail in Chapter 3. For these events, you will need to use custom event functions provided within the framework. Mojo events work within the DOM event model, but include support for listening to and generating custom Mojo event types and are stricter with parameters.

The webOS Services work a bit differently, with registered callbacks instead of DOM-style events, and are covered starting in Chapter 8. The event-driven model isn't conventional to web development, but has been part of modern OS application design and derives from that.

Storage

Mojo supports the HTML 5 database APIs directly and provides high-level functions to support simple create, read, update, or delete (CRUD) operations on local databases. Through these Cookie and Depot functions, you can use local storage for application preferences or cache data for faster access on application launch or for use when the device is disconnected.

UI Widgets

Supporting webOS's UI are UI *widgets* and a set of standard styles for use with the widgets and within your scenes. Mojo defines default styles for scenes and for each of the widgets. You get the styles simply by declaring and using the widgets, and you can also override the styles either collectively or individually with custom CSS.

The *List* is the most important widget in the framework. The webOS user experience was designed around a fast and powerful list widget, binding lists to dynamic data sources with instantaneous filtering and embedding objects within lists including images, icons and other widgets.

There are some basic widgets, including *buttons*, *selectors*, and *indicators*. The *Text Field* widget includes text entry and editing functions, including selection, cut/copy/paste, and text filtering. A Text Field widget can be used singly, in groups, or in conjunction with a list widget.

Menu widgets can be used within specified areas on the screen; at the top and bottom are the *View* and *Command* menus, which are completely under your control. The *App* menu is handled by the system, but you can provide functions to service the Help and Preferences items or add custom items to the menu. Some view and command menu types are shown in Figure 1-10.

Figure 1-10. View and command menu types

Pickers and *viewers* are more complex widgets. Pickers are for browsing and filtering files or for selecting numbers, dates, or times. If you want users to play or view content within your application, such as audio, video, or web content, then you need to include the appropriate viewer.

Using widgets

You must declare widgets within your HTML as an empty div with an `x-mojo-element` attribute. For example, the following declares a Toggle Button widget:

```
<div x-mojo-element="ToggleButton" id="my-toggle"></div>
```

The `x-mojo-element` attribute specifies the widget class used to fill out the div when the HTML is added to the page. The `id` attribute must be unique and is required to reference the widget from your JavaScript.

Typically, you would declare the widget within a scene's view file, then direct Mojo to instantiate the widget during the corresponding scene assistant setup method using the scene controller's `setupWidget()` method:

```
//  Setup toggle widget and an observer for when it is changed.
//  this.toggle        attributes for the toggle widget, specifying the 'value'
//                     property to be set to the toggle's boolean value
//  this.togglemodel   model for toggle; includes 'value' property, and sets
//                    'disabled' to false meaning the toggle is selectable
//
//  togglePressed      Event handler for any changes to 'value' property

this.controller.setupWidget('my-toggle',
  this.toggle = { modelProperty : 'value' },
  this.toggleModel = { value : true, disabled : false });

this.controller.listen('my-toggle', Mojo.Event.propertyChange,
  this.togglePressed.bindAsEventListener(this));
```

This code directs the scene controller to set up `my-toggle`, which passes a set of attributes called `this.toggle` and a data model called `this.toggleModel` to use when instantiating the widget and to register the `togglePressed` function for the widget's `propertyChange` event. The widget will be instantiated whenever this scene is pushed onto the scene stack.

To override the default style for this widget, select `#my-toggle` in your CSS and apply the desired styling (or use `.sliding-toggle-container` to override the styling for all toggle buttons in your application). For example, the following will override the default positioning of the toggle button to the right of its label so that it appears to the left of the label:

```
#my-toggle    {  float:left;
              }
```

There's a lot more to come, so you shouldn't expect to be able to use this to start working with any of these widgets yet. Chapters 3, 4, and 5 describe each of the widgets and styles in complete detail.

Services

Even if you limited yourself to just using the webOS System UI, application model, and UI widgets, you would have some unique opportunities for building web applications, particularly with notifications and the dashboard. But you'd be missing the access and integration that comes with a native OS platform. The *services* functions complete the webOS platform, fulfilling its mission to bridge the web and native app worlds.

Through the services APIs, you can access hardware features on webOS devices (such as location services, the phone, and the camera) and you can leverage the core application data and services that have always been a key part of a Palm OS device. Almost

all of the core applications can be launched from within your application, and there are CRUD functions for the calendar and contacts databases.

A service is an on-device server for any resource, data, or configuration that is exposed through the framework for use within an application. The service can be performed by the native OS (in the case of device services), an application, or by a server in the cloud. The model is very powerful, as evidenced by the initial set of offered services.

The services differ from the rest of the framework because they are called through a single function, `Mojo.Service.Request()`. The request passes a JSON (JavaScript Object Notation) object specific to the called service, and specifies callbacks for success and failure of the service request.

Starting with Chapter 8, you'll find a full description of the general model and handling techniques, as well as enumeration of all the services and the details for using each one.

Palm webOS Architecture

Palm webOS is based on the Linux 2.6 kernel, with a combination of open source and Palm components providing user space services, referred to as the *Core OS*.

You won't have any direct interaction with the Core OS, nor will the end users. Instead your access is through Mojo and the various services. Users interact with applications and the UI System Manager, which is responsible for the System UI. Collectively, this is known as the *application environment*. Figure 1-11 shows a simplified view of the webOS architecture.

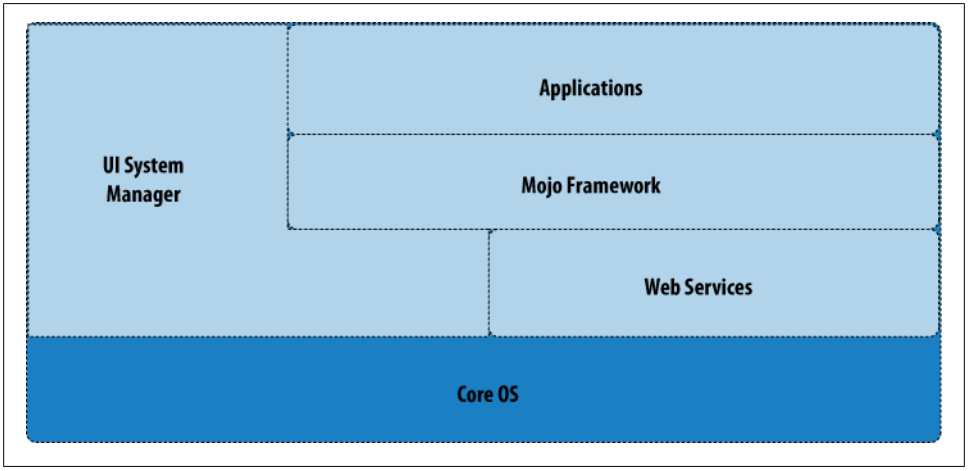


Figure 1-11. Simplified webOS architecture

This overview is included as background to give you an idea of how webOS works—this information is not needed to build applications, so you can skip it if you aren't interested.

Application Environment

The application runtime environment is managed by the UI System Manager, which also presents the System UI that is manipulated by the user. The framework provides access to the UI widgets and the Palm webOS services. Supporting this environment is the Core OS environment, an embedded Linux OS with some custom subsystems handling telephony, touch and keyboard input, power management, storage, and audio routing. All these Core OS capabilities are managed by the application environment and exposed to the end user as System UI and to the developer through Mojo APIs.

Taking a deeper look at the webOS architecture, Figure 1-12 shows the major components within the application environment and the Core OS.

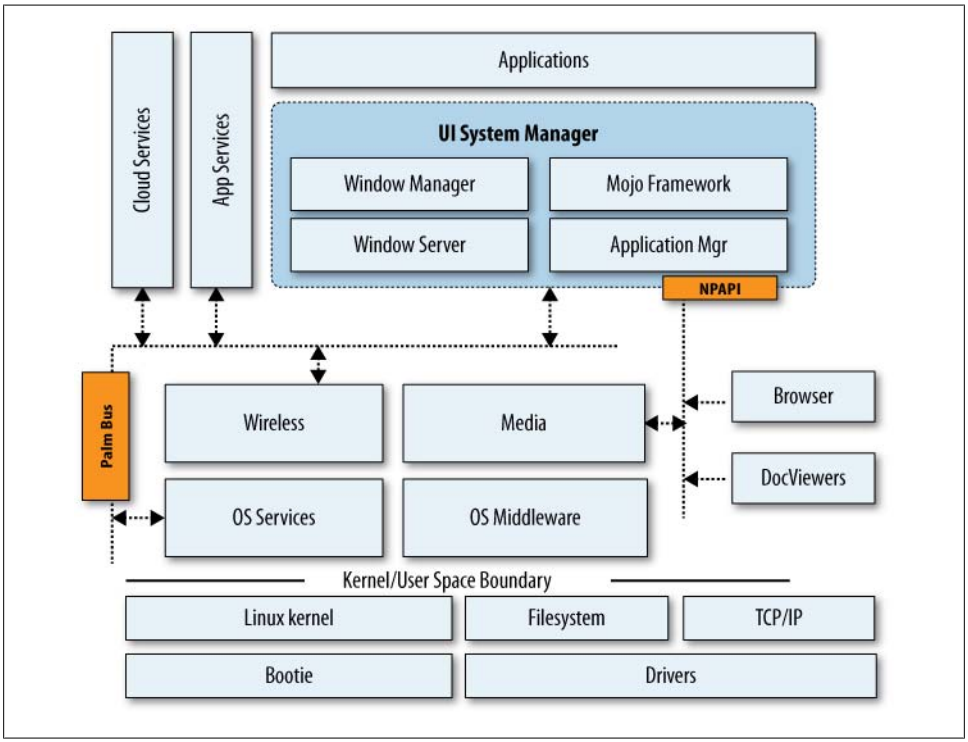


Figure 1-12. webOS system architecture

The application environment refers to the system user experience and the feature set that is exposed to the application developer, as represented by the Mojo framework and the Palm services. The Core OS covers everything else: from the Linux kernel and drivers, up through the OS services, middleware, wireless, and media subsystems. Let's take a brief look at how this all works together.

The UI System Manager is responsible for almost everything in the system that is visible to the user. The application runtime is provided by the application manager, built on

top of an instance of WebKit, which loads the individual applications and hosts the built-in framework and some special system applications, the status bar, and the Launcher. The Application Manager runs in a single process, schedules and manages each of the running applications, handles all rendering through interfaces to the graphics subsystem, and handles on-device storage through interfaces to the database engine.

Applications rely on the framework for their UI feature sets and for access to services. The UI features are built into the framework and handled by the Application Manager directly, but the service requests are routed over the Palm bus to the appropriate service handler.

Core OS

The Core OS is based on a version of the Linux 2.6 kernel with the standard driver architecture managed by udev with a proprietary boot loader. It supports an ext3 filesystem for the internal (private) file partitions and FAT32 for the media file partition, which can be externally mounted via USB (Universal Serial Bus) for transferring media files to and from the device.

The Wireless Comms system at the highest level provides connection management that automatically attaches to WAN (wide area network) and WiFi networks when available, and switches connections dynamically, prioritizing WiFi connections when both are available. EVDO or UMTS telephony and WAN data are supported depending upon the particular device model. Palm webOS also supports most standard Bluetooth profiles and provides simple pairing services. The Bluetooth subsystem is tightly integrated with audio routing to dynamically handle audio paths based upon user preferences and peripheral availability.

The media server is based on *gstreamer* and includes support for numerous audio and video codecs and all mainstream image formats, and supports image capture through the built-in camera. Video and audio capture is not supported in the initial webOS products, but is inherently supported by the architecture. Video and audio playback supports both file- and stream-based playback.

Software Developer Kit

Of course, the best way to get started writing webOS applications is to continue reading this book, but you should also go to Palm's developer site, *http://developer.palm.com*, and download the Mojo Software Developer Kit (SDK). The SDK includes the development tools, sample code, and the Mojo Framework, along with access to tutorial and reference documentation. Palm also hosts a webOS discussion forum for registered developers, where they can share ideas and ask questions in an environment that is monitored by Palm staff.

Development Tools

Palm makes the Mojo SDK and tools available for Linux, Windows (XP/Vista), and Mac OS X. The tools allow you to create a new webOS application project using sample code and framework defaults, search reference documentation, debug your application in the emulator or an attached Palm device, and publish an application. Chapter 2 includes more details about the tools in the SDK and third-party tools, but you'll find a brief summary in Table 1-1.

Table 1-1. Palm developer tools

Tools	Major features
SDK installer	Installs all webOS tools & SDK
Emulator	Desktop-hosted device emulator
Command-line tools	Create new project
	Install and launch in desktop emulator or device
	Package and sign application

The tools can be installed and accessed as command-line tools on every platform. They include a plug-in to Eclipse as well as Aptana Studio, a popular JavaScript/HTML/CSS editor for Eclipse.

Mojo Framework and Sample Code

The Mojo SDK includes the Mojo framework and sample code to help you design and implement your application. Unlike most JavaScript frameworks, you won't need to include the Mojo framework with your application code, since Palm includes the framework in every webOS device. The framework code included in the SDK is for reference purposes to help you debug your applications.

The sample code is also for reference. There are samples for most of the significant framework functions, including application lifecycle functions, UI widgets, and each of the services. Simple applications are included to get you started. You can review and leverage these applications as you choose.

webOSdev

Your main entry point to Palm's developer program is *http://developer.palm.com/*, which is where Palm hosts *webOSdev*, the developer web community. This site provides everything that you might need to build webOS applications, including access to the SDK, all development tools, and documentation and training materials, as well as developer forums and a blog specifically for the developer audience.

webOSdev is also the source for your application signing services and access to the Application Catalog. This is an application store that is published and promoted with every webOS device through a built-in Application Catalog application. Applications need to be signed for installation on a webOS device, and at webOSdev, you can get all the information you need to use the signing tools and to upload your application to the catalog, once they are made available.

You can find more information on the Palm developer program in Appendix A of this book and online at *http://developer.palm.com*.

Summary

In this introductory chapter, you were introduced to webOS, Palm's next generation operating system. You should now have a basic understanding of the webOS architecture and application model along with the basic services available in the SDK.

You'll find that it's pretty easy to get started writing webOS applications. After all, you're simply building web applications using conventional web languages and tools. You can port a very simple Ajax application by creating an *appinfo.json* file for your application at the same level as your application's *index.html* file. With as little as that, your application can be published and made available for download to any webOS device.

From there you can invest more deeply by building in the Mojo UI widgets to take advantage of the fluid physics engine, gesture navigation, beautiful visual features, text editing, and the powerful notification system. You can move beyond simple foreground applications that rely on active user interaction, and adapt your application to run in the background or even as a dashboard application. You can also create an application that can open new windows for each new activity, allowing users to multitask within a single application. There's a whole new generation of applications possible on the webOS platform, just waiting to be built.

CHAPTER 2
Application Basics

Palm webOS provides a great environment for building applications. The use of standard web development languages and tools, combined with access to native services and local data gives you a powerful and productive platform. Even Java and C/C++ developers will find that building applications using dynamic languages on webOS is fun and exciting. And despite what you might have heard, you can build real applications, not just web gadgets and spinners.

A browser-based web application is really just a set of complex web pages. They are downloaded from a web server and present their UIs as HTML, often with JavaScript as a client-side language to validate input, animate page elements, and make background Ajax calls back to the web server for additional interactivity.

If you are a developer writing these web applications, the Palm webOS development environment will feel familiar. JavaScript library code generates the HTML UI, interacts with page elements, and issues Ajax calls to web servers. You can style the UI with CSS, either to make your application look and feel consistent with Palm's style guidelines or to make your own unique look.

The programming model is a little different. Since the HTML is not generated on a server (say, using Java, PHP, or Ruby), there is no request/response lifecycle. Instead, all of your application code is in JavaScript—even interactions with key webOS systems (UI widgets, location services, and other applications) are made with JavaScript.

If you are a developer writing desktop or other native mobile phone applications in Java or C++, the Palm webOS development environment will feel familiar as well. There is a robust API for creating UI elements, accessing local storage, and making system calls. There is an application framework that makes it simple to do common tasks.

What is different for you is that the programming language is JavaScript and the UI is generated using HTML and styled using CSS. If you're new to JavaScript, HTML, or CSS, you may want to familiarize yourself with their fundamentals before tackling the next few chapters. Even so, the material presented here is fairly basic, and you don't need to be a web development expert to build applications for webOS.

In this chapter, you'll learn how to build a basic webOS application, starting with the installation of the SDK. You'll create a new application project, customize the critical application components, and develop the first parts of the News application, which will be used throughout the book as our sample application. We will also go into detail on how to use the framework and apply the different APIs, widgets, and styles.

Getting Started

You'll find everything you need to get started in developing Palm webOS applications at webOSdev, the Palm developer site (*http://developer.palm.com*). You'll need to sign up as a Palm developer and download the SDK. There are options for Mac OS X, Windows XP/Vista, and Linux, so download the SDK package that matches your development platform and run the installer.

The installation will put a copy of the SDK, including the Palm Mojo framework and Palm development tools, into one of the project directories listed in Table 2-1.

Table 2-1. SDK installation directories

Platform	Location
Mac OS X	*/opt/PalmSDK/Current/*
Windows XP/Vista	*C:\Program Files\Palm\SDK*
Linux	*/opt/PalmSDK/Current*

The installer will give you the option of installing different tool bundles. The tools package includes a collection of command-line tools which can be run on all platforms. In addition, the tools have been integrated into some popular IDEs and web development editors. Check the developer portal for an up-to-date list of bundles and supported editors.

The application samples in this book were all developed on a Mac with TextMate and the command-line tools. The command-line option for the tools will be shown in the examples and is the same on every platform. If you are using Eclipse/Aptana or another tool bundle, there should generally be direct menu options for the commands used in the book. In some cases, several commands may be combined into one menu option.

Creating Your Application

Palm webOS applications have a standard directory structure with naming conventions and some required files. To help you get started quickly, the SDK includes *palm-generate*, a command-line tool that takes an application or scene name as arguments and creates the required directories and files. You can run this from the command line (the *$* is the command-line prompt and should not be entered as part of the command):

```
$ palm-generate AppName
```

The command-line tools all work off of the current directory. You should change the directory to a projects directory (or wherever your workspace is located) before running the tools.

The tool creates a functional application within a conventional webOS directory structure. Every webOS application should have a directory structure similar to the following, and some parts of the structure are required:

```
AppName
   -> app
       -> assistants
           -> first-assistant.js
           -> second-assistant.js
           -> ...
       -> views
           -> first
               -> first-scene.html
           -> second
               -> second-scene.html
           -> ...
   -> appinfo.json
   -> icon.png
   -> images
       -> image_1.png
       -> image_2.jpg
       -> ...
   -> index.html
   -> sources.json
   -> stylesheets
       -> AppName.css
       -> ...
```

You are free to choose any project directory name, but `AppName` should correspond to the value of the `id` property in *appinfo.json*, discussed later in this section. An application's logic and presentation files reside in the *app* directory, which provides a directory structure loosely based on the MVC pattern.

There are scene assistants in the *assistants* directory of your application. As discussed earlier, a scene is a particular interactive screen in an application. Scene assistants are delegates implemented in your application, and are used by the framework's controllers to customize an application's behavior.

All layout files are located in the *views* directory of your application. A scene assistant has one main HTML view file, which provides the structure and content of its presentation page. It also includes optional HTML template view files that may be used to display dynamic data, such as JavaScript object properties for UI controls. These files are fragments of the UI that are combined together by the framework to produce the final presentation of the scene.

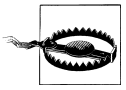
An application's images are located in the *images* directory and the CSS files are placed in the *stylesheets* directory. As with web applications, webOS applications use HTML to structure the layout of a page and CSS to style the presentation. CSS files are used to style your custom HTML, and you can also use CSS to override Mojo's default styles.

The `appinfo.json` object gives the system the information it needs to load and launch your application. Within `appinfo.json` there are both required and optional properties, which are described in Table 2-2.

The `palm-generate` tool creates an *appinfo.json* file with a common set of properties set to default values. The most important property is the `id`, which must be unique for each application; it's used in service calls to identify the caller and serves as a unique application identifier.

Table 2-2. appinfo.json properties and values

Property	Values	Required	Description
`title`	any	Yes	Name of application as it appears in Launcher and in application window
`type`	web	Yes	Conventional application
`main`	any	Yes	Application entry point
`id`	any	Yes	Must be unique for each application
`version`	x.y.z	Yes	Application version number
`vendor`	any	Yes	A string representing the maker of the application; it is used in launcher and deviceinfo dialogs
`vendorurl`	any	No	A string representing a URL that turns the vendor portion in deviceinfo dialogs to hyperlinks
`noWindow`	true/false	No	Background application; defaults to false
`icon`	file path	No	Application's launcher icon; defaults to *icon.png*
`miniicon`	file path	No	Notification icon; defaults to *miniicon.png*



A warning about syntax:

- Don't include any comments in *appinfo.json* files (/* or //).
- Must use double quotes around properties—no single quotes!
- Strict JSON parser; this file must follow all the rules for correct JSON.

The Application Manager is responsible for putting the application in the Launcher using the *icon.png* as the icon for your application. Application icons should be 64 × 64 pixels, encoded as a PNG with 24 bit/pixel RGB and 8 bits alpha. The icon's image should be about 56 × 56 pixels within the PNG bounds.

Refer to the webOS style guidelines found in the SDK documentation for more information about icon design.

Following web standards, *index.html* is the first page loaded when a webOS application is launched. There are no restrictions on what you put into the *index.html*, but you do need to load the Mojo framework here. Include the following in the header section:

```
<script src="/usr/palm/frameworks/mojo/mojo.js"
    type="text/javascript" x-mojo-version="1"></script>
```

This code loads the framework indicated by the `x-mojo-version` attribute; in this case version `1`. You should always specify the latest version of the framework that you know is compatible with your application. If needed, Palm will include old versions of the framework in webOS releases, so you don't need to worry about your application breaking when Palm updates the framework.

You can load your JavaScript using the sources tag in *index.html*, but this will load all the JavaScript at application launch. To improve launch performance, it is recommended that you use *sources.json* to provide lazy loading of the JavaScript. The `palm-generate` tool will create a template, and you can add the application specific files as they are created.

The generated file includes only your stage assistant, but a typical *sources.json* includes some scene assistants and perhaps an app assistant:

```
[
  {
    "source": "app/assistants/app-assistant.js"
  },
  {
    "source": "app/assistants/stage-assistant.js"
  },
  {
    "source": "app/assistants/first-assistant.js",
    "scenes": "first"
  },
  {
    "source": "app/models/data.js",
    "scenes": ["first", "second"]
  },
  {
    "source": "app/assistants/second-assistant.js",
    "scenes": "second"
  }
]
```

The `app-assistant` file path comes first, followed by the `stage-assistant` and the `scenes` file paths after that. The `scene` file paths can be in any order, but must include both the `source` and `scenes` properties. Note the example where both the first and second scenes with a dependency on the same file. HTML files are not included.

Applications can add other directories to the structure above. For example, you might put common JavaScript libraries under a *javascripts* or *library* directory, or put test libraries under *tests*. The required elements are:

- The *app* directory and everything within it
- *appinfo.json*

The rest of the structure and naming is recommended but not required.

Testing and Debugging

Most web applications can simply be loaded into a browser to run and debug them, and webOS apps can also be tested and debugged that way. However, you'll run into difficulty if your application is using Mojo widgets or webOS services. And it's difficult to fully test your application without seeing it working within the webOS System UI and other applications.

For this reason, you'll want to use the webOS Emulator with integrated JavaScript debugger and DOM inspector. Unlike the other development tools, the emulator is a full native application on every platform and will be found in the *Applications* directory on MacOS X and Linux or in the *Programs* directory on Windows XP/Vista. First, you can launch the emulator directly; it will bring up a window that looks like a Palm Prē. Or, you can use command-line tools; use `palm-package` to package your application and use `palm-install` to run it on the emulator.

You'll also use the debugger for testing your applications after connecting any Palm webOS device to your system using a USB cable. From the command line you can run `palm-package` and `palm-install` to run your application on the device as well.

News

After the core webOS applications, one of the first applications built for webOS was the News application. In August 2008, the webOS platform was far enough along that the webOS Engineering team wanted to have someone completely unfamiliar with webOS write a webOS application. An experienced JavaScript developer built a primitive version of the News application and gave us feedback on the tools and documentation. After a month, we ended the experiment, put the prototype into Subversion and went on with the project.

Over a recent holiday, I started poking around at the application thinking that it would be fun to get it updated to the latest framework and see what could be done with it.

Within a few days, I had rewritten the application and had something useful; a week or so later it was ready to post on the internal website. I was amazed at how much fun I was having with it and felt guilty for continuing to work on it; it was addictive.

An RSS (Really Simple Syndication) reader is a useful application and although simple to write, it uses a lot of the features of the framework, so it seemed like a good choice for a sample application in this book. We're going to build the application up bit by bit throughout the book to explore how to write a webOS application and how to use the different APIs, widgets, and styles. We won't use every API or every widget or every style, but there'll be enough from each part of the framework that you can see how to apply the examples to things that are not covered.

The News application manages a list of newsfeeds, periodically updating the feeds with new stories. It has the following major features:

- Users can monitor the newsfeeds for unread stories, scroll through the stories within a feed, and display individual stories.
- Feeds and stories are cached on the device for offline viewing.
- The original and full story can be viewed in the browser.
- Keyword searches are carried out over the entire feed database.
- Stories are shared through email or messaging.
- Feeds are updated in the background with optional notifications and a dashboard summary.
- The application was localized for the Spanish language.

News Wireframes

It's useful to block out the design for an application before you begin any coding or detail design.A wireframe shows the layout of an application view and the UI flow from one view to the next. A set of the News wireframes is shown in Figure 2-1.

This shows the main scene of the News application with a featured story in the top third of the screen. This is a text *scroller* widget encapsulated with a base div style called *palm-group-title*. Below the feature story area is the feed *list* widget, which serves as an index of the selected feeds. At the top of the scene is a search field built with a *filter-list* widget, which is hidden until some text is keyed, at which point the entered text will be used in a keyword search of the entire feed database.

None of the style or widget names will make much sense right now, so don't worry about that, but note that the wireframe calls out these base styles and the widgets that will be used.

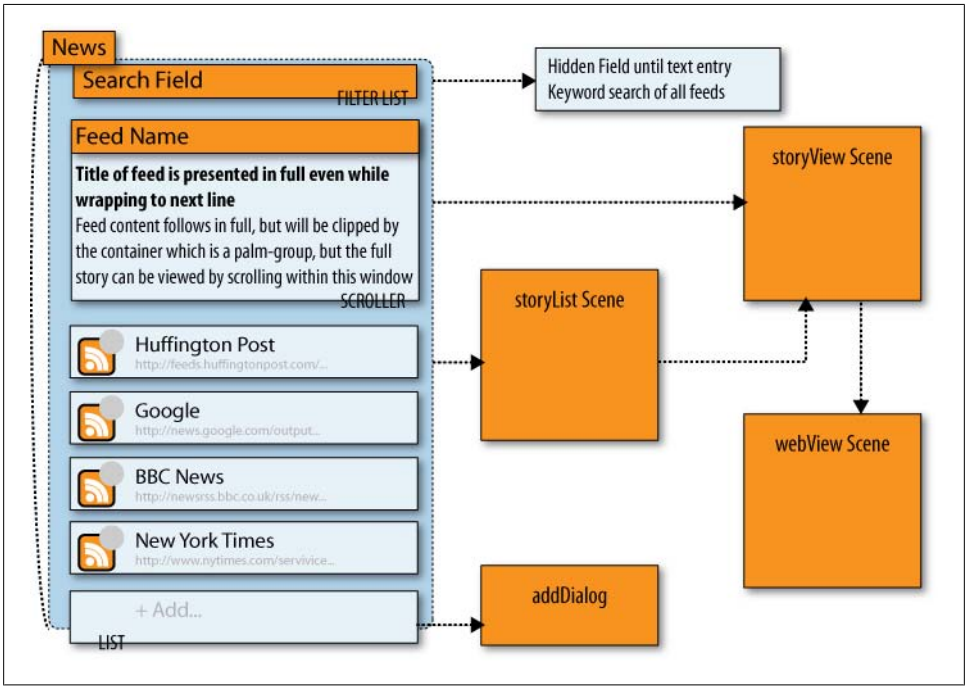


Figure 2-1. Part of the News wireframes

The News application supports other scenes, which users can bring into view by tapping different elements. Tapping a feed list entry will bring up a new scene called `storyList`, which is a list of the stories in the feed. A scene of a specific story is viewed by tapping the story in the list. You can also get to the `storyView` scene by tapping the featured story. Tapping a story from *storyView* will go to the original story URL in yet another scene, the `webView` scene.

A complete set of wireframes would include a diagram for each scene and for each dialog box, such as the dialog box in this scene that adds new feeds to the list.

Chapter 1 includes an overview on style guidelines, and it's at this wireframe design stage that the guidelines are best applied. The News application uses physical metaphors of showing the feeds and stories in lists that can be tapped (to view), scrolled, deleted, or rearranged. Users add new feeds by tapping on the end of the list. No menus are needed; all the actions are directly applied to the elements on the screen.

The SDK includes a comprehensive set of style guidelines that you may find helpful. It covers broad user-experience guidelines for designing webOS applications, and includes technical details that are essential for visual and interaction designers. The style guidelines will help you design for the platform and not just a single device.

Designing for Palm webOS

Palm webOS is initially available on the Palm Prē, but it is a platform that will be used on other devices with different screen sizes and resolutions. Design your application so that it works well on different devices by following these key guidelines:

- Total usable screen real estate will be at least 320 pixels wide in primary use mode, but will often be larger. An application should gracefully handle different window sizes, usually by having at least one section of the screen that can expand or contract.
- The minimum hit target is 48 pixels.
- The minimum font size is 16 pixels for lowercase text and 14 pixels for all caps.
- Use Mojo.Environment.DeviceInfo to retrieve device specific property values.

You should refer to the style guidelines in the SDK for the complete and most up-to-date information.

Creating the News Application

We'll start by repeating the steps covered in the previous section to create a new application project. Using `palm-generate`, create a new application named **`News`**. You'll see an initial application structure (shown in Figure 2-2):

```
$ palm-generate News
```

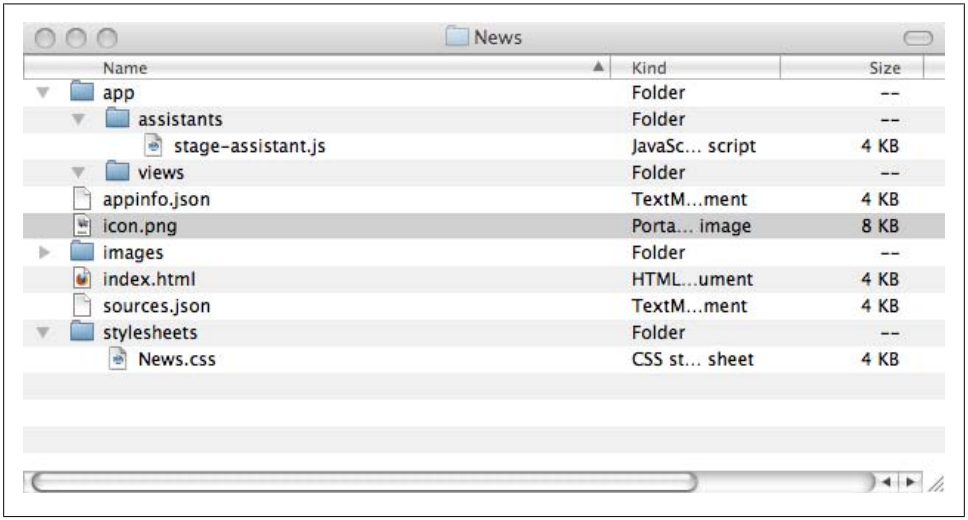


Figure 2-2. Creating a new Palm webOS application

Running this app simply displays some text, which isn't very interesting. We'll add the first scene and some actual application content, but first there's some basic housekeeping to do.

Start by cleaning up the *index.html* file. Remove all of the HTML code between the `<body>` and `</body>` tags, and update the application title. In the following sample, the application title has been left as `News`, but it has been formatted into title case:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <title>News</title>
  <script src="/usr/lib/mojo/framework/mojo.js" type="text/javascript"
    x-mojo-version="1"></script>

  <!-- application stylesheet come in after the one loaded by the framework -->
  <link href="stylesheets/News.css" media="screen" rel="stylesheet"
     type="text/css" />
</head>
<body>
</body>
</html>
```

Customizing the Launcher Icon and Application ID

While not strictly necessary, each application should have a unique launcher icon. To implement this, you will create a custom graphic and replace the generic *icon.png* in the News directory. If you don't want to draw a custom graphic you can repurpose other graphics as long as they have close to a 1:1 aspect ratio (square, in other words) by using resampling tools like the Preview application on the Macintosh. See the sidebar "Designing for Palm webOS" on page 33 for more information on creating application icons.

Next, we'll open the *appinfo.json* file and update the `id` property. You should at least replace the `yourdomain` string with your actual domain name (in this case we'll use `palm`) or some other unique string and add the vendor name. We'll also modify the title to put the application name in title case:

```
{
    "title": "News",
    "type": "web",
    "main": "index.html",
    "id": "com.palm.app.news",
    "version": "1.0.0",
    "vendor": "Palm",
    "icon": "icon.png"
}
```

The application doesn't yet do anything more than before, but now it is uniquely the News application. It's time to add the first newsfeed features.

Adding the First Scene

The first scene will just display a story from one newsfeed, which will be hardcoded to start with. This scene will use several of Mojo's scene styles, including `palm-page-header`, `palm-page-header-wrapper`, and `title`.

The scene view

As you learned earlier in the chapter, a scene is defined by an HTML view file and controlled by a JavaScript assistant. Again, use `palm-generate`, this time to create a scene:

```
$ palm-generate -t new_scene -p "name=storyView" News
```

This command creates a directory with the scene name, and within that directory, an empty HTML file with the scene name followed by `-scene.html`. In this case, the filename will be *views/storyView/storyView-scene.html*. It also creates a JavaScript assistant, *assistants/storyView-assistant.js*, with the same base scene name. The JavaScript assistant will be discussed in the next section. The file structure is shown in Figure 2-3.

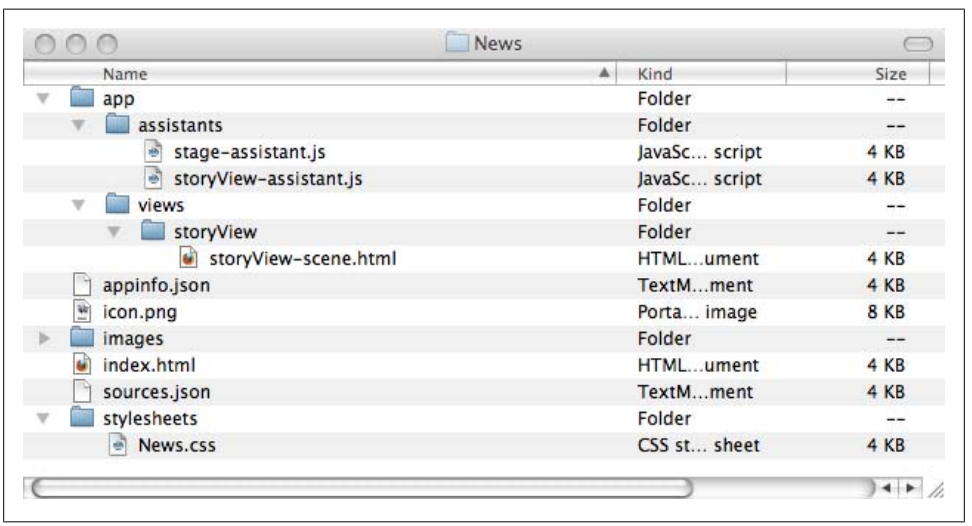


Figure 2-3. A new scene view file

Custom Application Structures

If you find the conventional structure too constraining, you can customize it for your application. Place your scene's *.html* file where you want it under the *app* directory. For

example, you can group several scenes into a separate directory called *main* within the *views* directory, such as this scene, named *front*:

app/views/main/front/front-scene.html

In this case, we've added another directory between the *views* directory and the *scene* directory. When pushing example-scene-one you will do it like so:

```
Mojo.Controller.stageController.pushScene({name:"front",
    sceneTemplate:"main/front/front-scene"});
```

The call to push the scene tells the framework explicitly where to find the scene's template. This is useful for code organization purposes; particularly with large applications or in cases when you might want to reuse code in different applications. It's also a technique that you can use to work around the naming convention between the scene's view file name and assistant. You can find some examples of this in the SDK sample code.

Within the scene's view file, you need to specify the complete HTML required by the scene. The title and text will be inserted dynamically, so there are just div tags for the title with an id `storyViewTitle`, and the story item with the id `storyViewSummary`, with some template strings between the div tags for the dynamic data:

```
<div id='storyViewScene'>
    <div class="palm-page-header multi-line">
        <div class="palm-page-header-wrapper">
            <div id="storyViewTitle" class="title left">
            </div>
        </div>
    </div>
    <div class="palm-text-wrapper">
        <div id="storyViewSummary" class="palm-body-text">
        </div>
    </div>
</div>
```

You'll notice that the `storyViewTitle` div is wrapped by two div tags, which each have class names corresponding to Mojo scene styles. Scene styles are covered later in this chapter, but for this example you should note that the scene styles are defined through empty divs with class names corresponding to the scene style selected. Where multiple elements can be included in the style, there will be an inner `wrapper` style for each element. Occasionally, the styles will have modifiers; in the example above, the base style is modified with `multi-line`, signifying some behavior to accommodate titles that are longer than a single line.

In Mojo, CSS class names belong to the designers and are used for styling, whereas the element IDs belong to the developer. This rule allowed the Mojo design and development teams to work somewhat independently without conflicting. You always invoke a Mojo style by assigning the appropriate style class name to the div. For scene styles, the element will usually be an empty div, sometimes with some required nested div(s).

The scene assistant

Mojo requires that the scene assistant's name match the view name, so you should notice that `palm-generate` created a scene assistant called *storyView-assistant.js* in the *News/app/assistants* directory. The assistant includes a function definition along with the four standard methods:

- `setup`
- `activate`
- `deactivate`
- `cleanup`

The function naming is important; the assistant's name should be the same as the filename, with the removal of any delimiters and beginning with a capital letter, as in this example:

```
/*  StoryViewAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Passed a story element, displays that element in a full scene view.

    Major components:
    - StoryView; display story in main scene

*/

function StoryViewAssistant() {

}

// setup
StoryViewAssistant.prototype.setup = function() {

    // Update story title in header and summary
    var storyViewTitleElement = this.controller.get("storyViewTitle");
    var storyViewSummaryElement = this.controller.get("storyViewSummary");
    storyViewTitleElement.innerHTML =
        "Green Inc.: Cities Target Lending to Speed Energy Projects";
    storyViewSummaryElement.innerHTML =
        "A number of municipalities across the country are getting creative and
         experimenting with incremental, neighborhood- or district-based lending
         programs that help homeowners pay the up-front capital costs for efficiency
         or renewable energy projects.";
};

// activate - called each time the scene is revealed
StoryViewAssistant.prototype.activate = function(event) {

};

// deactivate - called each time the scene is replaced by another scene
StoryViewAssistant.prototype.deactivate = function(event) {
```

```
};

// cleanup - called once, after deactivate when scene is popped
StoryViewAssistant.prototype.cleanup = function(event) {

};
```

The `setup` method is invoked, and the story's title and text are put into the div tags and into the scene before the scene is activated. We use the Mojo controller method, `get()`, to retrieve the `storyViewTitle` and `storyViewSummary` DOM elements from the scene, and then insert the title and text items into the respective element's innerHTML.

The `get()` method is functionally equivalent to the conventional `getElement()` or Prototype's `$()` functions, but is required when developing multistage and background applications (see Chapter 10). It's good practice to use it in even simple applications like this.

The code sample shows empty activate, deactivate, and cleanup methods, which are created by `palm-generate` when generating a new scene. Empty methods are okay, but can be omitted. If any of the standard methods are omitted, the framework will just skip over the call to that method.

Pushing the scene

To push the scene, modify *stage-assistant.js* to push the `storyView` scene:

```
function StageAssistant () {
}

StageAssistant.prototype.setup = function() {
    this.controller.pushScene("storyView");
};
```

Notice that only the scene name is used here. You can see why the naming convention is important; the framework uses the scene name to access the assistant, view directory, and view file. Finally, add a reference into *sources.json* to load *storyView-assistant.js*:

```
[
  {
    "source": "app/assistants/stage-assistant.js"
  },
  {
    "source": "app/assistants/storyView-assistant.js",
    "scenes": "storyView"
  }
]
```

Now run the application by packaging and installing the application. You'll need to first launch the emulator or attach a device to your desktop over USB. In either case, first run `palm-package` and then `palm-install`:

```
$ palm-package News
Creating package in /Users/Mitch/Documents/workspace/com.palm.app.news_1.0.0_all.ipk
$ palm-install --install com.palm.app.news_1.0.0_all.ipk
```

Figure 2-4 shows the first scene.

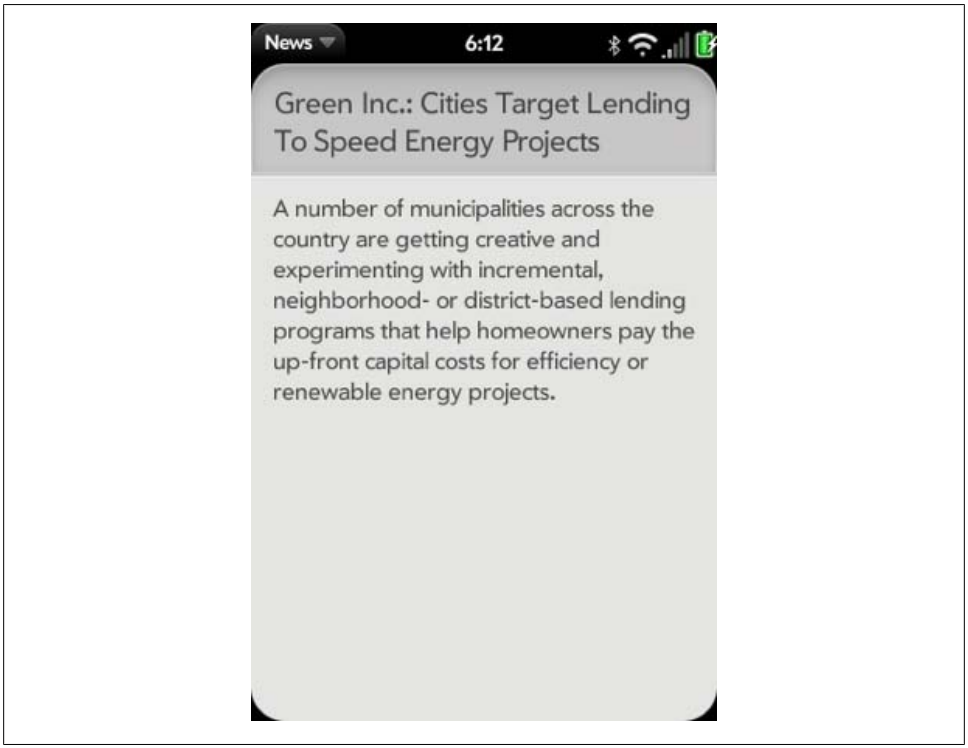


Figure 2-4. storyView scene

Styling the scene

The first scene was styled using standard framework styles. The story title is styled with `palm-page-header`, `palm-page-header-wrapper`, `title`, and `left`, while the story text uses `palm-text-wrapper` and `palm-body-text`. In many cases, the framework styles will be all you need, but you can use the application's stylesheet to override those styles or add your own application-specific styles.

For example, we could override the `palm-body-text` style if we wanted to change the text body style to enlarge text so that it's easier to read. Add a rule to the *News.css* file, and you will see a new version of the scene, as shown in Figure 2-5:

```
/*    News CSS
    App overrides of palm scene and widget styles.
*/
```

```
.palm-body-text  {
    font-size: 16pt;
}
```

The style override is applied to the base style class `palm-body-text`, but this change will affect every use of that style within your application. If you are just trying to change this one instance of this style, you should add another class to the element you wish to style and apply the rule to that class instead. Here we create a rule with a new class name, `news-large`:

```
/*    News CSS
    App overrides of palm scene and widget styles.
*/

.news-large  {
    font-size: 16pt;
}
```

Then add this class name to the text element in `view/storyView/storyView-scene.html` and you'll again see the results shown in Figure 2-5:

```
<div class="palm-text-wrapper">
    <div id="storyViewSummary"  class="palm-body-text news-large">
    </div>
</div>
```

With the base styles, it's the class names that must be kept intact in order to retain the base styling, but you can freely create custom styles to augment the base styles.

It may be necessary to override the base styles to get the right appearance for your application, so don't be afraid to dive into the CSS. Palm's development tools help you see the styling rules applied to any given element to help you make the desired adjustments.

Base Styles

Applying styles is an important topic, so we'll return to it again later. Chapter 7 is devoted entirely to advanced styles, covering some general styling tips and techniques. A lot of time can be spent getting the right pixels in the right places.

Mojo includes a sophisticated suite of base styles that are used heavily by the framework and the core webOS applications. You should leverage them within your applications where the base styles will make your application feel like a native webOS application, and users will learn how to use your application more quickly, based on their experience with other applications. That said, you have ultimate flexibility to change the appearance of your application as needed, and you can override individual properties within the styles to tailor them to your needs.

This section will give a brief overview of the webOS base styles and provide some basic techniques that you can use. It's beyond the scope of this book to describe the base

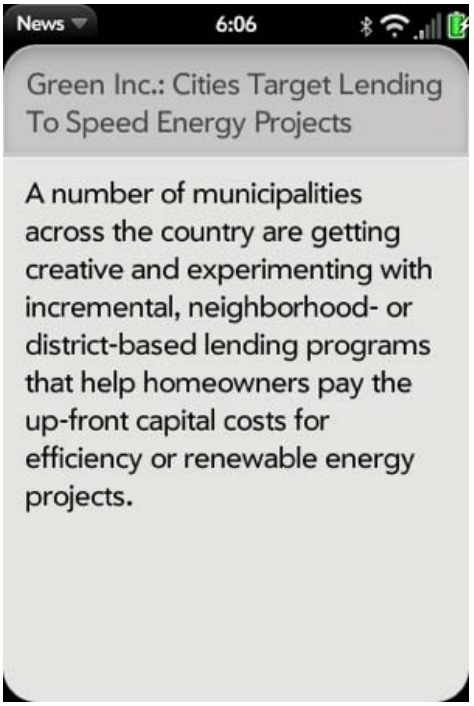


Figure 2-5. Modified storyView scene

styles in detail. It's a big topic and, given the central nature of the UI design to Palm webOS, it's also changing frequently. If you're interested in an in-depth description of webOS styling, you should check out the "Human Interface Guidelines," and the "Styles and CSS Reference" in the Palm SDK. In addition, Appendix C includes a reference to the base styles with more detail on the options for applying those styles.

Elements

Mojo defines base styling for key textual elements such as body, paragraph, input, button, and others. If you want your application to have the look and feel of a webOS application, you shouldn't override these styles. These style definitions can be found in the Mojo framework in *stylesheets/global-base.css*.

Scene styles

The most basic style elements are those used to present a scene template. These include page headers, groups, labels, spacers, and dividers, along with background images. Some of the styles that you will find are shown in Table 2-3; most of these styles are found in the Mojo framework in *stylesheets/global-lists.css* and are described in more detail in Appendix C.

Table 2-3. Scene styles

Style class name	Description
`palm-page-header`	Scene style with header that adjusts to text dimension; can be single line or multi-line
`palm-header`	Scene style with pill at top; single line only
`palm-header-spacer`	Keeps other scene elements from folding under the header
`palm-group`	Container for lists, text fields, and/or widgets; internal to scene
`palm-group-title`	Title for palm-group
`palm-group unlabeled`	Internal to scene; no title
`palm-divider labeled`	Labeled divider
`palm-divider collapsible`	Collapsible divider

You can use each of these styles as-is or adjust the style properties in your CSS. These are just a few of the scene styles available through the framework; the SDK includes a much more extensive discussion of the styles available to you.

Widget styles

Each widget is designed with a base style. In the next three chapters, we'll cover widgets in detail, including the base styles and properties. Generally speaking, however, you will be able to override the widget style in your application CSS, and you can refer to the style definitions in the framework's CSS files for guidance. Be careful though; the widgets are tuned around the provided styles and you can easily break the widget's behavior by overriding the widget's style. Table 2-4 cross-references the widget styles with the framework's CSS files.

Table 2-4. Widget style definitions and corresponding CSS files

Widget styles	CSS filenames
Buttons	*global-buttons.css, global-buttons-dark.css*
Dialogs	*global.css, global-dark.css*
Drawers	*global-lists.css, global-lists-dark.css*
Indicators	*global.css, global-dark.css*
Lists	*global-lists.css, global-lists-dark.css*
Menus	*global-menus.css, global-menus-dark.css*
Notifications	*global-notifications.css*
TextFields	*global-textfields.css, global-textfields-dark.css*
Pickers	*global-menus.css, global-menus-dark.css*

Application Launch Lifecycle

It's helpful to understand what's happening when an application is launched and the first scene is pushed. When News is launched, the Mojo framework will first look for an *AppAssistant*, an optional controller class. The AppAssistant is used to handle launch arguments from the system and to set up stages if the application requires more than the main Card view or has multiple stages, but it isn't a required element. If the system finds the AppAssistant, it will be called. Otherwise this step will be skipped.

In this minimal form, News doesn't have an AppAssistant; many simple single stage applications don't. But as you'll see in Chapter 10, the AppAssistant is important for background applications and to applications with multiple stages.

In Chapter 10 we'll add the AppAssistant to News and explore its use in detail.

The *StageAssistant* contains code that applies to all scenes in a stage and is used to push the first scene onto the stage, which will create the initial view for the application. The terms *push* and *pop* are used to refer to scenes being made visible in a window (push) or removed from the window (pop), reinforcing the concept that scene navigation is like managing a stack. The user opens new scenes with a tap and then uses the *back* gesture to return to the previous scene (as if tracking along a stack of scenes).

The framework always creates a stage controller and if the application includes a defined Stage Assistant, its constructor function is used to create a Stage Assistant and its setup method is called.

If there isn't a Stage Assistant, or if there is one but it doesn't push an initial scene, the framework will look for a scene named *main* to push as the first scene. If you provide a Stage Assistant and push the first scene directly, you may call the scene whatever you choose; otherwise, the first scene assistant must be named main, and *main-assistant.js* must be included in your assistant's directory.

Most of what happens within a webOS application occurs within a scene. Once the initial scene is pushed, the framework will load the scene's view file, then invoke first the `setup` and then the `activate` methods of the Scene Assistant. The `setup` method sets up widgets and event listeners, and performs other setup functions that persist across the life of the scene. `activate` is always called before the scene is put into the view, either because of a push or because a later scene was popped. `setup` is only called when the scene is pushed.

When a scene is popped or covered up by the push of a new scene, the framework invokes the `deactivate` method for the old scene before activating the new scene. The naming conventions allow the framework to manage both the view and assistant methods of the scene without explicit configuration files. It may seem a little cumbersome

at first, but it's a simple and self-documenting technique. `cleanup` is called only when the scene is popped.

All Palm applications will have at least one stage, with each stage supporting one or more scenes. Applications can have multiple stages and the means to transition between these stages. When creating applications that can support multiple activities, such as an email application that supports both viewing a message list and composing multiple email messages, you should structure the application stages around activities.

Adding a Second Scene

At this point, the application displays a single story within a single scene, which is very limited. We can display more stories by adding some additional scenes, but first we'll create the core data model to store the newsfeeds and stories. Under *News/app/models*, create *feeds.js*, an object that will include the global array of newsfeed objects, each of which in turn includes an array of stories. Later we'll add methods to the object to perform common functions, such as updating and backing up the data. The following code sample includes a breakdown of these data objects and expands the definition of our sample newsfeed with four sample stories:

```
/*  Feeds - the primary data model for the News app. Feeds includes the primary
      data structure for the newsfeeds, which are structured as a list of lists:

    Feeds.list entry is:
      list[x].title           String    Title entered by user
      list[x].url             String    Feed source URL in unescaped form
      list[x].type            String    Feed type: either rdf, rss or atom
      list[x].numUnRead       Integer   How many stories are still unread
      list[x].newStoryCount   Integer   For each update, how many new stories
      list[x].stories         Array     Each entry is a complete story

    list.stories entry is:
      stories[y].title        String    Story title or headline
      stories[y].text         String    Story text
      stories[y].summary      String    Story text, stripped of markup
      stories[y].unreadStyle  String    Null when Read
      stories[y].url          String    Story url

    Methods:
    initialize(test) - create default and test feed lists
    getDefaultList() - returns the default feed list as an array
*/

var Feeds = Class.create ({
    //  Default Feeds.list
    defaultList: [
        {
        title:"New York Times",
        url:"http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml",
        type:"rss",
        numUnRead:4,
```

```
    stories:[
        {
            title: "Obama Warns of Prospect for Trillion-Dollar Deficits",
            text: "Barack Obama delivered a stark assessment of the
              economy, saying that his administration would be forced
              to impose tighter discipline on government.",
            unReadStyle: "unReadStyle",
            url: "http://www.nytimes.com/2009/01/07/world/asia/
              07india.html?_r=1&partner=rss&emc=rss"
        },
        {
            title: "Hundreds of Coal Ash Dumps Lack Regulation",
            text: "Most of the coal byproduct dumps across the United
              States are unregulated, although they contain chemicals
              considered as threats to human health.",
            unReadStyle: "unReadStyle",
            url: "http://www.nytimes.com/2009/01/06/world/asia/
              06iqbal.html?partner=rss&emc=rss"
        },
        {
            title: "Gazprom Dispute Entangles Europe",
            text: "Russia's gas price dispute with Ukraine escalated,
              disrupting deliveries to the European Union in the midst
              of a bitter cold spell.",
            unReadStyle: "unReadStyle",
            url: "http://www.nytimes.com/2009/01/07/world/europe/
              07gazprom.html?partner=rss&emc=rss"
        },
        {
            title: "Green Inc.: Cities Target Lending to Speed Energy
              Projects",
            text: "A number of municipalities across the country are
              getting creative and experimenting with incremental,
              neighborhood- or district-based lending programs that help
              homeowners pay the up-front capital costs for efficiency or
              renewable energy projects.",
            unReadStyle: "unReadStyle",
            url: "http://greeninc.blogs.nytimes.com/2009/01/06/cities-use-
              creative-targeted-lending-to-speed-energy-projects/
              ?partner=rss&emc=rss"
        }
    ]
}],

// initialize - Assign default data to the feedlist
initialize: function()  {
    this.list = this.getDefaultList();
},

// getDefaultList - returns the default feed list as an array
getDefaultList: function() {
    var returnList = [];
    for (var i=0; i<this.defaultList.length; i++)   {
        returnList[i] = this.defaultList[i];
    }
```

```
        return returnList;
    }
});
```

The Feeds object is defined using Prototype's `Class.create()` function, a convenient way to build an object with class-like behavior. If you're not familiar with this notation, refer to *http://www.prototypejs.org/api* for more information about Prototype functions.

`Feeds` includes a single feed entry for the New York Times RSS feed and four stories. Each feed entry includes a title, a URL, the feed type, and some other properties that we'll be using in the upcoming chapters. The stories array contains the individual stories, each of which has a title, the text body, a read/unread flag, and a stripped version of the text.

The stage assistant is modified to create an instance of the Feeds object and then push the `storyView` scene as before, but this time with some arguments identifying the source feed (`this.feeds.list[0]`) and the index of the first story:

```
/*  StageAssistant - NEWS
    Responsible for app startup.
    Major components:
    - setup; app startup and initial load of feed data
        from the Depot and setting alarms for periodic feed updates

    Data structures:
    - feeds; feed object used for main feedlist in feedList-assistant
    - globals; set of persistant data used throughout app

*/
//  ---------------------------------------------------------------
//  GLOBALS
//  ---------------------------------------------------------------

//  News namespace
News = {};

//  Constants
News.unreadStory = "unReadStyle";


function StageAssistant () {
}

StageAssistant.prototype.setup = function() {
    this.feeds = new Feeds();           // initialize the feeds model
    // start with the first feed and the first story in the feed
    this.controller.pushScene("storyView", this.feeds.list[0], 0);
};
```

In the StageAssistant setup method, arguments have been added to the `pushScene` method call. Any number of arguments can be provided after the first required argument, which is the scene name.

Globals

Global variables can be problematic when writing client-side JavaScript, and it's generally a good idea to avoid them.

Each webOS application is its own document, so no application globals are visible to any other application. Mojo uses only three globals itself, the Mojo prefix, `$L`, and `$LL`, which are used for localization encapsulation.

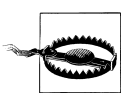
You can use global variables more freely in webOS, though you may still want to avoid them for other reasons (such as code portability and maintenance). If you do use them, creating a global namespace for your application is a good practice.

The `StoryViewAssistant` is updated to handle the input arguments and the new data structures. In the function invocation, the arguments are assigned to properties of the scene assistant, making them available to each scene method. This is a common technique when handling input arguments to a scene so that the arguments can be used throughout the scene assistant's scope:

```
/*  StoryViewAssistant - NEWS

    Passed a story element, displays that element in a full scene view and
    offers options for next story (right command menu button) and previous
    story (left command menu button) Major components:
    - StoryView; display story in main scene
    - Next/Previous; command menu options to go to next or previous story

    Arguments:
    - storyFeed; Selected feed from which the stories are being viewed
    - storyIndex; Index of selected story to be put into the view
*/

function StoryViewAssistant(storyFeed, storyIndex) {
    //  Save the passed arguments for use in the scene.
    this.storyFeed = storyFeed;
    this.storyIndex = storyIndex;
}

StoryViewAssistant.prototype.setup = function() {

    if (this.storyIndex === 0)    {
        this.controller.get("previousStory").hide();
    }

    if (this.storyIndex == this.storyFeed.stories.length-1)    {
        this.controller.get("nextStory").hide();
    }

    this.nextStoryHandler = this.nextStory.bindAsEventListener(this);
    this.previousStoryHandler = this.previousStory.bindAsEventListener(this);
    this.controller.listen("nextStory", Mojo.Event.tap, this.nextStoryHandler);
    this.controller.listen("previousStory", Mojo.Event.tap,
```

```
        this.previousStoryHandler);

    var storyViewTitleElement = this.controller.get("storyViewTitle");
    var storyViewSummaryElement = this.controller.get("storyViewSummary");
    storyViewTitleElement.innerHTML = this.storyFeed.stories[this.storyIndex].title;
    storyViewSummaryElement.innerHTML = this.storyFeed.stories[this.storyIndex].text;

};

//    activate - display selected story
StoryViewAssistant.prototype.activate = function(event) {

   if (this.storyFeed.stories[this.storyIndex].unreadStyle == News.unreadStory) {
        this.storyFeed.numUnRead--;
        this.storyFeed.stories[this.storyIndex].unreadStyle = "";
    }
};

StoryViewAssistant.prototype.deactivate = function(event) {

};

StoryViewAssistant.prototype.cleanup = function(event) {
    this.controller.stopListening("nextStory", Mojo.Event.tap,
      this.nextStoryHandler);
    this.controller.stopListening("previousStory", Mojo.Event.tap,
      this.previousStoryHandler);
};

StoryViewAssistant.prototype.previousStory = function(event) {
    this.controller.stageController.pushScene("storyView", this.storyFeed,
      this.storyIndex-1);
};

StoryViewAssistant.prototype.nextStory = function(event) {
    this.controller.stageController.pushScene("storyView", this.storyFeed,
      this.storyIndex+1);
};
```

In the `setup` method, listeners are set up for `previousStory` and `nextStory`. These are button elements that, when tapped, will cause a new scene to be pushed with the appropriate story. Notice that the listeners are removed in the cleanup method.



Remove all event listeners in the scene's cleanup method. Failing to do so is a common cause of memory leaks in webOS applications.

The `nextStory` and `previousStory` methods push a new scene with the new story. The scene that is pushed is this same `storyView` scene, showing some of the flexibility of the scene model.

You may have noticed that we created properties of the scene assistant for data used within the assistant scope. It's a useful way to manage data that is limited to a particular instance of the assistant. To access the assistant object and its properties in your event listeners, you need to bind the assistant's controller instance to the event listeners. The code examples throughout the book use Prototype's `bindAsEventListener`, but you aren't required to use that method.

The button elements are added to the end of *storyView-scene.html*:

```
<div id="previousStory" class="palm-button">Previous</div>
<div id="nextStory" class="palm-button">Next</div>
```

Although Mojo has button widgets, the framework does support all standard HTML elements and has included a `palm-button` class for the button element that provides a style for HTML buttons that is consistent with the widget styles.

Run this version of the application and push the buttons to go from one story to the next and back again. You should see views similar to those shown in Figure 2-6.

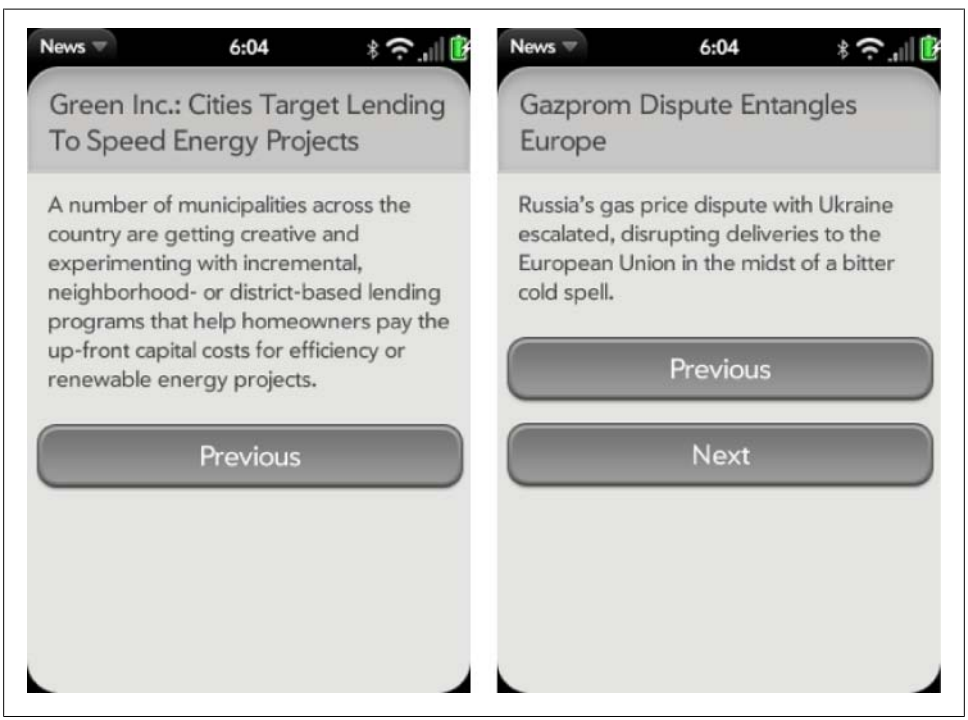


Figure 2-6. Additional scenes

But there's a problem with this solution. Since scenes are stacked when pushed, each button press (whether next or previous) adds another scene to the stack—to no real advantage. Try tapping next and previous a few times, then start using the back gesture. You just unwind all the stories that you pushed on the stack.

`pushScene` is just is one of the `StageController` methods provided to help you manage the scene stack efficiently. In our example, it would be better to use the `swapScene` method. As its name implies, `swapScene` swaps the new scene for the old and doesn't increase the stack depth. It's an easy change because `swapScene` uses the same syntax as `pushScene`:

```
StoryViewAssistant.prototype.previousStory = function(event) {
    this.controller.stageController.swapScene("storyView", this.storyFeed,
      this.storyIndex-1);
};

StoryViewAssistant.prototype.nextStory = function(event) {
    this.controller.stageController.swapScene("storyView", this.storyFeed,
        this.storyIndex+1);
};
```

Now when you tap through the stories, you are just swapping one story for another on the stack. So when you swipe back you'll find that you are already at the top of the stack. It's hard to see how important this is given that this is currently such a simple application, but it will become more obvious as we go forward.

Both examples using `pushScene()` and `swapScene()` use the default transition animation when changing scenes, which is zooming style. This is the recommended style for moving up or down the scene stack, but not for a lateral transition. You can override the default animations and in this case, we specify a *cross-fade* transition. The `swapScene()` method calls below use a `sceneArguments` object, with `name` and `transition` properties; you'll use the `sceneArguments` object anytime you need to override the default arguments of a scene method:

```
StoryViewAssistant.prototype.previousStory = function(event) {
    this.controller.stageController.swapScene(
        {
            transition: Mojo.Transition.crossFade,
            name: "storyView"
        },
        this.storyFeed, this.storyIndex-1);
};

StoryViewAssistant.prototype.nextStory = function(event) {
    this.controller.stageController.swapScene(
        {
            transition: Mojo.Transition.crossFade,
            name: "storyView"
        },
        this.storyFeed, this.storyIndex+1);
};
```

This is as much as we're going to do with News in this chapter, so if you're eager for more hands-on information, you can skip to the next chapter at this point. The remaining section of this chapter covers more advanced topics that will help you understand the underlying framework design, but they aren't strictly needed to write webOS applications. You can always come back to this topic later if you are interested in learning more about it.

Controllers

So far we've used two controller classes (`StageController` and `SceneController`) and referred to a third (`AppController`). All three classes are part of `Mojo.Controller` namespace. The assistants that we've created are associated with their respective controller classes and rely heavily on the methods in those classes.

`AppController` and the use of stages within an application will be covered in depth in Chapter 10, when we cover notifications and background applications.

It is worth noting that an application has just one application controller object and may optionally have a single application assistant to create and manage stages.

Controllers and Assistants

An application can have multiple stage controller objects, and each stage controller can have a stage assistant. A stage assistant is not an instance of `StageController`, but is actually a delegate of the controller. The assistant has a controller property set to a reference to the associated controller, which is used to directly call the controller's methods. The assistant defines its own methods as well.

Each stage controller has a stack of scene controllers. When a scene is pushed, a new scene controller is created and pushed onto the stack. As with the stage, each scene controller has a scene assistant delegate that, after initialization, will have its controller property set to a reference of the scene controller it belongs to.

To illustrate this, let's go back to our News application. Although it's quite simple now with only one stage and two scenes, it will, over time, grow to having multiple stages to handle the dashboard along with the card stage that we are currently working in, and there will be at least five scenes. The controller/assistant hierarchy is shown in Figure 2-7.

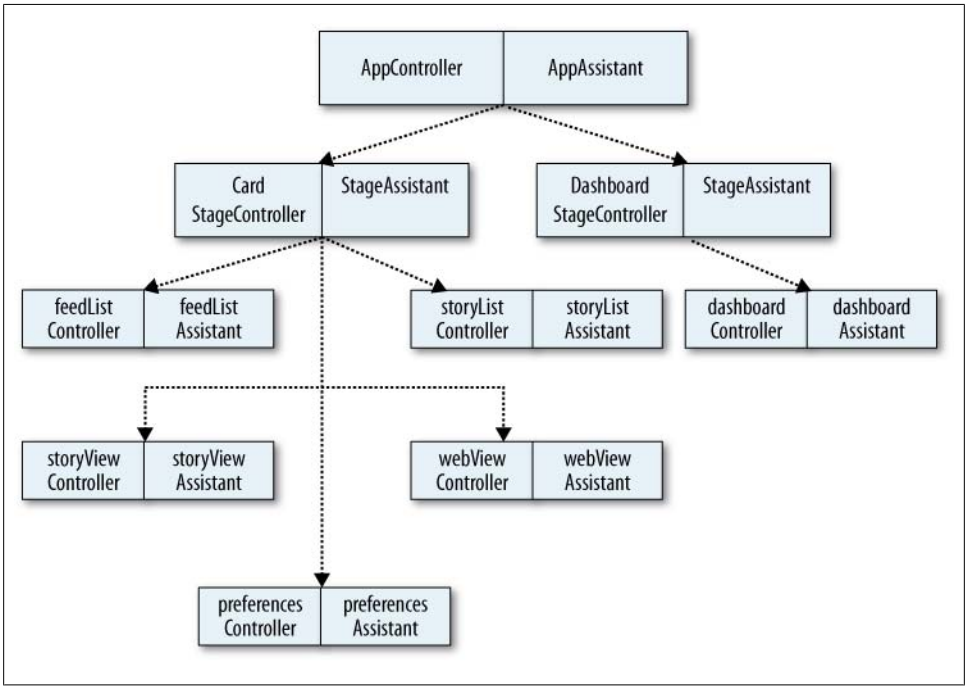


Figure 2-7. The News application controller/assistant hierarchy

Scene Stack

We've already looked at `pushScene` and `popScene`, but there are other methods that you will use to manage the scene stack. Refer to the SDK documentation for a complete and up-to-date list of available methods. Some of the more commonly used methods include:

`pushScene (sceneArguments)`
: Pushes a new scene, passing in the optional `sceneArguments`.

`popScene(returnValue)`
: Removes a scene from the scene stack, passing the `returnValue` to the newly revealed scene's `activate` method.

`popScenesTo (targetScene, returnValue)`
: Removes scenes from the scene stack until the `targetScene` is reached or there are no scenes remaining on the stack, passing the `returnValue` to the new scene's `activate` method.

`swapScene (sceneArguments)`
: Pops the current scene and simultaneously pushes a new scene without activating or deactivating any underlying scenes, passing in the optional `sceneArguments`.

`topScene()`
: Returns the topmost scene from this stage.

`getScenes()`
: Returns an array of scene controllers currently on the stack.

`activeScene()`
: Returns the currently active scene from this stage, if any.

`sceneArguments` and `returnValue` can be any number of arguments of any type. They are simply passed through to the target scene.

Summary

We built the first version of News, the sample application we'll be building throughout the book. So far, News can display multiple stories from a hardcoded test feed, using one scene recursively. We started with SDK installation and used the webOS toolset to create a new project, and stepped through all the application basics to build the News card and a couple of scenes. From these basics you should be able to build simple web applications and style them.

With the webOS SDK and a handful of Mojo APIs, you can build a conventional web application that can be downloaded, installed, and run on any webOS device. In the next chapter, you'll learn how to add widgets to your application and leverage the rich UI built into Palm webOS.

CHAPTER 3
Widgets

The heart of the Mojo framework is the UI feature set, delivered in a collection of dynamic widgets. Mojo widgets are configured with scene controller methods and custom CSS styles, and are managed through Mojo events, briefly introduced to you in Chapter 1.

Widgets let you build static and dynamic lists or employ various button controls, selectors, and text fields. You can choose from several kinds of menus and dialog boxes, and employ sophisticated pickers and viewers, each of which specialize in handling different types of data. There's a common model for declaring, instantiating, and managing your widgets, which makes it easy to learn and simple to code for your applications.

A widget is declared within your HTML as an empty div with an `x-mojo-element` attribute declaring the type of widget to display. Typically, you declare the widget within a scene's view file, then configure and set up the widget in the corresponding scene assistant's setup method. You listen for events associated with the widget to take actions dictated by the user through the widget or to update data associated with the widget. The framework applies default styles to the widget; you can override those styles in your CSS, but in many cases the default styles will work perfectly.

In this chapter, we will start with a design overview of Mojo widgets, then walk through some basic widgets: buttons and selectors, lists, and text fields. We'll use a number of these widgets in the News application; at least one from each category to show you how to apply them in your applications.

All About Widgets

Widgets are dynamic UI controls that can be integrated within any application. They can be tailored to the application, yet provide reusable, stylistically consistent UI functions. The term "widgets" is widely used within web development, but Mojo widgets are different than other widgets. Mojo widgets have a defined behavior and have many options; they generate complex HTML and are easily styled with CSS.

It helps to understand the HTML that the widget generates. This is especially true for widgets like List and Dialog, for which the application specifies HTML templates that largely define the widget's appearance.

Declaring Widgets

Widgets are declared in HTML as empty div tags:

```
<div id="my-toggle" x-mojo-element="ToggleButton"></div>
```

The `x-mojo-element` attribute specifies the widget type used to fill out the div when the HTML is added to the page. This can happen in either of the following circumstances:

- When a scene is pushed and the scene's view HTML includes widgets.
- When a widget is specified in an HTML template used by another widget.
- When the application inserts HTML which includes widgets, and makes a call to explicitly instantiate them.

In the second case, for example, if your scene includes a List widget whose list items include other widgets, a new set of the list item's widgets are instantiated each time a new item is added to the list.

Setting Up a Widget

Before a widget is inserted into the scene, it must be set up. You should do this in the scene assistant's `setup` method by calling the scene controller method, `setupWidget()`.You need to provide three arguments to this call (shown in Table 3-1).

Table 3-1. setupWidget arguments

Argument	Description
Widget ID	The ID or `name` of the div element in which the widget was declared
Attributes	Object containing the widget's static properties, normally options or attributes of the widget
Model	Object containing the widget's dynamic properties, usually data associated with the widget, but occasionally including dynamic attributes

For example, a Toggle Button would be set up this way:

```
var toggleAttr = {trueValue: "on", trueLabel: "On",
                  falseValue: "off", falseLabel: "Off"};
this.toggleModel = { value: "on", disabled: false };
this.controller.setupWidget("my-toggle", toggleAttr, this.toggleModel);
```

The `my-toggle` argument specifies the widget being set up. The argument can be either the `id` or `name` of the widget's div element. It's usually fine to use `id`, but it won't be unique if your widget is instantiated more than once. This can happen if the widget is declared in a template for a list item, or if your scene might be pushed multiple times (without being popped). In these cases, use the `name` attribute instead.

The second argument specifies the attributes for the widget. These are properties that affect the behavior and display of the widget, but are not tied to the actual data being displayed or edited. Widget attributes cannot be changed after the widget is instantiated. A toggle button is a simple binary selector; its attributes include `trueValue`, `trueLabel`, `falseValue`, and `falseLabel`, among others. The values allow you to toggle between on/off, left/right, up/down, in/out, and so on, while the labels can either track those values or offer different terms for the user.

The last argument specifies the widget's data model object. This is the actual user data displayed by the widget. The contents of the model object will often change, each time requiring the widget to be updated. In our example, the model includes the toggle's value and a disabled property, set to `false`.

The split between attributes and model objects was designed to allow you to use widgets within list entries. The attributes represent the setup shared between list items, and the model provides the per-item data. This will make more sense when you get to the section "Lists" on page 64.

Updating a Widget's Data Model

When a widget model is changed outside of the widget, the widget will not automatically update and reflect those changes. The application (usually the scene assistant) is required to call the `modelChanged()` method on the widget's scene controller, passing the model object that changed. The scene controller will then notify all widgets using that model, so they can properly display the current model data. For example, suppose you disabled the toggle button in our example:

```
this.toggleModel.disabled = true;
this.controller.modelChanged(this.toggleModel, this);
```

The first argument to the `modelChanged()` method is the model object that has changed. Model change notification uses the identity of the model object to determine which widgets are using that model object and then notify them to update.

The second argument identifies which object has changed the model. This ensures that objects are not notified of their own changes to the model.

Scene assistants (and widget assistants, where applicable) will usually simply pass the keyword `this`. The argument is optional if called from something other than a widget controller.

The `modelChanged()` method notifies widgets of changes to a model object; if you need to directly change the model, you should use `setWidgetModel()` instead. While `setupWidget()` applies to all widgets with the given HTML name attribute, `setWidgetModel()` only ever applies to a single widget instance. So you must pass the widget's ID or the actual widget DOM element.

Calling `modelChanged()` with an entirely new model object will not update the model. Instead, there will be no change, since the specified model will not be used by any existing widget; no notifications will be generated or received. A common bug occurs when assigning a model to another object then calling `modelChanged()` using that new object. This will not work; you need to use the original model object for `modelChanged()` or use `setWidgetModel()` with the new object.

The following will change the model in our toggle button example:

```
// Use a new model object in place of the old one:
this.newToggleModel = {value: "off"};

// Set the widget to use the new model:
this.controller.setWidgetModel("my-toggle", this.newToggleModel);
```

Widget Event Handling

Each widget is supported by events. Where possible, the widget will use common events, such as `Mojo.Event.tap` or `Mojo.Event.propertyChange`. Where that's not possible, widget-specific events are defined, such as `Mojo.Event.listDelete` or `Mojo.Event.listReorder` for List widgets, or `Mojo.Event.scrollStarting` for the Scroller widget.

You should set up event listeners in the scene assistant's setup method when you set up the widget, by adding the listeners to the div element that declares the widget. For example, the toggle button sends a `Mojo.Event.propertyChange` when the widget is toggled, meaning that the toggle button's model changes value.

Using the example toggle button that we've been building on, you would set up a listener with code like this:

```
This.controller.listen("my-toggle", Mojo.Event.propertyChange
  this.handleSelectorChange.bindAsEventListener(this));
```

For more details on this, look at the section "Events" on page 15, which covers the entire event model for more information, or consult the Mojo.Event API reference in the Palm SDK.

Using Widgets

Now we're going to start using and discussing the individual widgets in depth. Table 3-2 summarizes the widgets included in Mojo version 1, though keep in mind that new widgets will be added to the platform periodically. You should check the Palm Developer site for the latest information.

Table 3-2. Mojo widgets

Collection	Widgets
Buttons & Selectors	Button, Check Box, Radio Button, Toggle Button, List Selector, Slider
Lists	List, Filter List
Dialogs & Containers	Custom Dialog, Alert Dialog, Error Dialog, Drawer, Scroller
Text Fields	Text Field, Filter Field, Password Field, Rich Text Edit
Menus	App Menu, Command Menu, View Menu, SubMenu
Pickers	Date Picker, File Picker, Integer Picker, Time Picker
Viewers	Image View, Web View, Audio & Video Objects
Indicators	Progress Bar, Progress Pill, Progress Slider, Spinner

We'll add various widgets to the News application, and through those examples you'll learn how to use the Mojo widgets in your application.

Widgets are declared in the HTML scene. You set them up and instantiate them from within the JavaScript assistant, and you can style them through CSS. You've seen that all widgets have an attribute object that contains static properties applied at the time the widget is instantiated, and a model object, which holds the dynamic values associated with the widget.

In the remainder of this chapter and the two that follow, you'll see how to use each of the widgets in concrete examples. They each have their own unique capabilities and there are some tips for using them that you might find helpful.

Buttons and Selectors

Buttons and selectors are the simplest Mojo widgets. Earlier in the chapter, you saw how to use a toggle button; all the other buttons and selectors work in a very similar way. In this section, we'll work directly with the Button widget, adding one to the News application, and we'll touch on the other widgets in this group: Toggle Button, Check Box, Radio Button, List Selector, and Slider.

Buttons

You can use simple HTML divs *styled* as button widgets; these will work quite well in many cases. Button *widgets*, can behave dynamically, for example, displaying a spinner to show activity. Figure 3-1 shows an example of the Button widget.

Buttons are the most basic UI element, bounding an action to a region. When a user pushes a button, the button can change state and then gracefully return to the previous state, much like a doorbell. You can create unstyled buttons, or you can style them as objects, and you can label them in some way with text or images. You can disable Mojo buttons, and you can configure them to show activity indicators.

Figure 3-1. A Button widget example

Use an HTML button for initiating actions, but use a Button widget when you are combining an action initiation with an indicator, or for any asynchronous actions. As we did in Chapter 2 for switching scenes, declare HTML buttons in your view file using conventional HTML notation.

Assigning the button div's class to `palm-button` means the button will display like a Mojo Button widget. The framework applies the same style to HTML buttons of class `palm-button` as it does to Mojo Button widgets.

If desired, you can override the style for either type of button in your CSS. The most typical style modification is to adjust the button width, which, by default, is centered across the width of the card's window. There are additional styles when a button is used as the primary or secondary choice, or to indicate dismissal, affirmative, or negative actions. Include any of these styles in your declaration, and the framework will apply default styling.

Adding a button to News

At the end of Chapter 2, we added a scene to the News application using HTML buttons and the back gesture. While it's not really visible in the UI, we're going to replace the HTML buttons with Button widgets so you can see how to add a simple widget. If you think you are already clear on how this works, you can skip ahead to the section on Lists.

The first change is simple: replace the button tags in *storyView-scene.html* with widget declarations:

```
<div id="storyViewScene">
    <div class="palm-page-header multi-line">
        <div class="palm-page-header-wrapper">
            <div id="storyViewTitle" class="title left">
            </div>
        </div>
    </div>
    <div class="palm-text-wrapper">
        <div id="storyViewSummary" class="palm-body-text">
        </div>
    </div>

    <div x-mojo-element="Button" id="previousStory"></div>
    <div x-mojo-element="Button" id="nextStory"></div>

</div>
```

The next change is to the *storyView-assistant.js*; add the widget setup in front of the listeners for the button taps. We're not changing the button `id` in either case, so we don't have to change the listener setup functions at all:

```
StoryViewAssistant.prototype.setup = function() {
    this.nextModel = {disabled: false};
    this.previousModel = {disabled: false};

    if (this.storyIndex === 0)    {
        this.previousModel.disabled = true;
    }

    if (this.storyIndex == this.storyFeed.stories.length-1)    {
        this.nextModel.disabled = true;
    }

    this.controller.setupWidget("previousStory", {label: "Previous"},
    this.previousModel);
    this.controller.setupWidget("nextStory", {label: "Next"}, this.nextModel);

    this.nextStoryHandler = this.nextStory.bindAsEventListener(this);
    this.previousStoryHandler = this.previousStory.bindAsEventListener(this);
    this.controller.listen("nextStory", Mojo.Event.tap,
      this.nextStoryHandler);
    this.controller.listen("previousStory", Mojo.Event.tap,
      this.previousStoryHandler);

    // Update story title in header and summary
    var storyViewTitleElement = this.controller.get("storyViewTitle");
    var storyViewSummaryElement = this.controller.get("storyViewSummary");
    storyViewTitleElement.innerHTML = this.storyFeed.stories[this.storyIndex].title;
    storyViewSummaryElement.innerHTML = this.storyFeed.stories[this.storyIndex].text;
};
```

The rest of the code stays the same. When you run this version of the application, it behaves the same as the previous version even though it uses Button widgets instead of the HTML button.

Selectors

The simple selectors will be used in other parts of the News application, shown in later examples. For now, let's look briefly at each of the selectors and how you can use them in your application.

Check Box

A Check Box widget (Figure 3-2) controls and indicates a binary state value in one element.

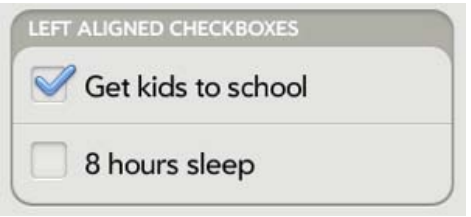


Figure 3-2. A Check Box widget example

Tapping a check box on the screen will toggle its state, presenting or removing a checkmark, depending on the previous state. The framework handles the display changes and will manage the widget's data model for you, toggling between two states that you defined at setup time.

Toggle Button

The Toggle Button is another widget for displaying and controlling a binary state value. As with a check box, a toggle button (Figure 3-3) will switch between two states each time it is tapped.

Figure 3-3. A Toggle Button widget example

Radio Button

If you need a single widget to select from among multiple choices while also showing selection status, then a Radio Button (Figure 3-4) is a good choice. Mojo provides a classic radio button, which presents each button as a labeled selection option in a horizontal array, where only one option can be selected at a time.

The number of options is variable, constrained only by the width of the display and the minimum button size that can be pleasingly presented or selected. You can expect to handle between two and five states, given the typical screen size for a webOS device, but the framework won't limit you.

Figure 3-4. A Radio Button widget example

List Selector

Even though you might expect to find the List Selector as one of the List widgets, it behaves and is managed as a selector. It enables the selection of one of many options, presented in a pop-up list in which there is no practical limit to the number of options presented. It is similar to the Submenu widget's behavior. Figure 3-5 shows an example of the List Selector widget.

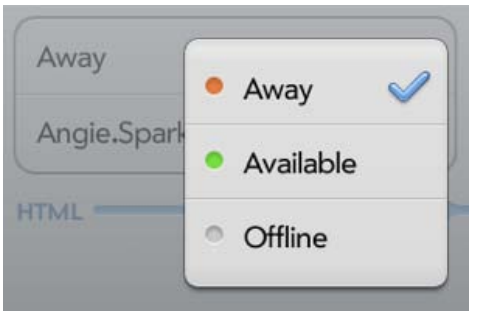


Figure 3-5. A List Selector widget example

The selection options are defined in a required choices array, which defines each selection's displayed label and a corresponding value. If the choices are static, meaning they never change over the life of the scene, you define the array as a property in the widget's attributes. If the choices are subject to change, attach them as a model property instead.

List Selectors in Forms

To group List Selectors as you might do in a form, use a div with the `palm-group unlabeled` class followed by a div with the `palm-list` class, then individual selector divs containing List Selector widgets with using the various `palm-row` classes. For example:

```
<div class="palm-group unlabeled">
  <div class="palm-list">
    <div class="palm-row first">
      <div id="trainSelector" x-mojo-element="ListSelector"></div>
    </div>
    <div class="palm-row">
      <div id="departureSelector" x-mojo-element="ListSelector"></div>
    </div>
    <div class="palm-row">
      <div id="destinationSelector" x-mojo-element="ListSelector"></div>
    </div>
    <div class="palm-row last">
      <div id="timeSelector" x-mojo-element="ListSelector"></div>
    </div>
  </div>
 </div>
```

A related tip with forms: you can combine the various models into single object, with different properties for each widget, by specifying the `modelProperty` to a property name in a shared object. Having one object simplifies the processing of the forms.

The List Selector is like the List widget in its styling. To have the webOS look and feel, you'll need to wrap your widget declaration with styling divs like those used with the List widget, and you may need to style those List classes with your own CSS. See the below section "Lists" for more information.

Slider

The last widget in this group of selectors is the Slider, which presents a range of selection options in the form of a horizontal slider with a control knob that users can drag to the desired location. You must specify minimum (leftmost) and maximum (rightmost) values for the slider. Figure 3-6 shows an example of the Slider widget.

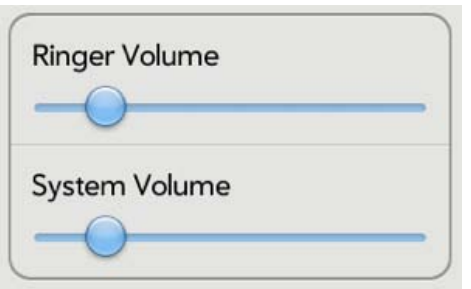


Figure 3-6. A Slider widget example

Lists

The design for Mojo began with the List. To validate the webOS architecture and the concept of Mojo, the principle webOS architects were challenged to design a list widget that would pull dynamic data from the Contacts database as the user flicked through the list, without any perceptible delay or loss of data (on a low-end CPU, no less). Needless to say, this was not a trivial challenge. However, the challenge was met and the rest of the framework took shape around the resulting design.

The webOS user experience makes extensive use of lists in many applications. Given the form factor and the navigation model, most applications will incorporate a List widget in one way or another. To get the most out of Mojo you need to fully understand the List widget.

The other list widget, the Filter List, is derived from the List widget. It has many of the same features as the List widget, but is designed around a more specialized use case. Our sample application will make use of Filter List in Chapter 5.

List Widgets

Lists are rendered by inserting objects into the DOM using provided HTML templates for both the list container and the individual list rows. Lists can be variable height and include single and multi-line text, images, or other widgets. Some Lists are static, meaning the list items are provided to the widget directly as an array. Other Lists are dynamic, meaning the application provides items as needed for display. Lists can be manipulated in place, with the framework handling deletion, reordering, and add item functions for the application.

There are examples of the list in the core applications, including the Email inbox, the Message chat view, the Contacts list, the Music library, and more. You can see that lists are flexible, yet fast and very efficient.

Back to the News: Adding a Story List

We're going to use a List widget in a few places in News. First, we're going to convert the sample newsfeed to a list, then hook it up to an Ajax call to get the live newsfeed into the application. That should give us a basic news reader for one feed, but to handle multiple feeds we'll add another List widget as a list of feeds. The application will start to take its basic shape in this section.

We'll create a list to hold the sample list that we've been working with. Using the `palm-generate` tool, create a new scene for a list view, called *storyList*:

```
$ palm-generate -t new_scene -p "name=storyList"
```

In the view file, *views/storyList/storyList-scene.html*, declares a List widget under a `palm-header` that includes some text to which we'll later assign the list's title. The `storyListWgt` div is the List widget declaration:

```
<div id="feedTitle" class="palm-header center">
    Feed Title
</div>
<div class="palm-header-spacer"></div>
<div id="storyListScene" class="storyListScene">
    <div x-mojo-element="List" id="storyListWgt" ></div>
</div>
```

Next, create the list templates. These are two HTML files that you put into the *views/storyList* directory; the container template *storyListTemplate.html* and the row template *storyRowTemplate.html*.

All List widgets are all built using HTML templates to lay out and format the list container and the individual rows. You normally include these templates as separate HTML files in your scene's view folder (where your scene view file is located), but you can also specify each template's pathname, which allows you to share templates between scenes or organize them in other ways. Pathnames are specified with relative notation `scene-dir/template-file`, where `scene-dir` is the directory for the current scene's view file. Within the template, you will reference properties from the list.

In Mojo, pathnames are relative to the *'app'* directory, not the location of *index.html*.

The `listTemplate` is optional; it defines the path to an HTML template for the list's container, which, if missing, will simply put the list items into the scene enclosed in div with the `palm-list` classname. If present, the `listTemplate` can have only one top level element.

The `itemTemplate` is required; it is set to the path of an HTML template for the list items. Use the notation `#{property}` to identify specific `items` properties for insertion into the template.

The *storyListTemplate* includes a single line using the `palm-list` class to format the list and a template entry for `#{-listElements}`:

```
<div class="palm-list">#{-listElements}</div>
```

By default, Mojo will escape any HTML that is inserted into a template to limit the risk of JavaScript insertion into views. You can add a leading hyphen to any property reference to prevent HTML from being escaped on that property. This is required in list container templates for the list widget to render properly.

The *storyRowTemplate* is a little more involved, using an outer div with the class `palm-row` to format the row, then each list row has both a title entry and a text entry:

```
<div class="palm-row" x-mojo-touch-feedback="delayed">
    <div class="palm-row-wrapper">
        <div id="storyTitle" class="title truncating-text #{unreadStyle}">
            #{title}
        </div>
        <div id="storySummary" class="news-subtitle truncating-text">
            #{text}
        </div>
    </div>
</div>
```

Each entry uses the `truncating-text` class, which will cause the entry to be automatically truncated at the list boundaries with ellipsis to indicate truncation. The templates `#{title}` and `#{text}` refer to the list items properties of those names that are substituted into the template.

The `#{unreadStyle}` template references another list items property that indirectly forces some styling specifically for the story titles that are not read. This demonstrates that property substitution can be used with any HTML content. Further on, we will apply some CSS styling to the classname used in the `unreadStyle` property.

Taken together, the scene's view or HTML files wrap the List widget with some specific styles to get the visual appearance shown in Figure 3–6. You should review the SDK's "User Interface Guidelines" for a complete discussion of Mojo styling, but to summarize briefly, there are three levels of styles at work in the `storyList` scene:

`palm-list`
: Is used in the `listTemplate` to drive spacing and the light separator rule that divides the list entries.

`palm-row`
: Wraps the div tag containing the list entry template to handle background styles and styling for highlight, selection, swipes, and other dynamic behavior. It can be modified with additional styles for `first`, `last`, `single`, and others (a complete list is provided in Appendix C).

`palm-row-wrapper`
: Also wraps the div tag containing the list entry template and adjusts spacing within `palm-row`.

Back to the example, to implement the feed list handling, add the *storyList-assistant.js*:

```
/*  StoryListAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Displays the feed's stories in a list, user taps display the
    selected story in the storyView scene. Major components:
    - Story View; push story scene when a story is tapped

    Arguments:
    - feedlist; Feeds.list array of all feeds
    - selectedFeedIndex; Feed to be displayed
*/

function StoryListAssistant(feedlist, selectedFeedIndex) {
    this.feedlist = feedlist;
    this.feed = feedlist[selectedFeedIndex];
    this.feedIndex = selectedFeedIndex;
    Mojo.Log.info("StoryList entry = ", this.feedIndex);
    Mojo.Log.info("StoryList feed = ", Object.toJSON(this.feed));
}


StoryListAssistant.prototype.setup =  function() {

// Setup story list with standard news list templates.
    this.controller.setupWidget("storyListWgt",
        {
            itemTemplate: "storyList/storyRowTemplate",
            listTemplate: "storyList/storyListTemplate",
            swipeToDelete: false,
            renderLimit: 40,
            reorderable: false
        },
```

```
        this.storyModel = {
            items: this.feed.stories
        }
    );

    this.readStoryHandler = this.readStory.bindAsEventListener(this);
    this.controller.listen("storyListWgt", Mojo.Event.listTap,
      this.readStoryHandler);

    //  Set title into header
    $("feedTitle").innerHTML=this.feed.title;
};

StoryListAssistant.prototype.activate =  function() {
    // Update list models
    this.storyModel.items = this.feed.stories;
    this.controller.modelChanged(this.storyModel);
};

StoryListAssistant.prototype.cleanup =  function() {
    // Remove event listeners
    this.controller.stopListening("storyListWgt", Mojo.Event.listTap,
        this.readStoryHandler);
};

// readStory - when user taps on displayed story, push storyView scene
StoryListAssistant.prototype.readStory = function(event) {
    Mojo.Log.info("Display selected story = ", event.item.title,
        "; Story index = ", event.index);
    Mojo.Controller.stageController.pushScene("storyView", this.feed,
      event.index);
};
```

When the scene is instantiated with a call to the `StoryListAssistant` function, the passed feed index assigns the selected feed to `this.feed`. The setup method is called before the scene is pushed and sets up the List widget: the templates are assigned and `renderLimit` is set to 40. You should use the default for your lists, but adjust it if necessary after testing.

renderLimit

The number of list elements that the List widget will render into the DOM at any one time is defined by `renderLimit`. You usually won't need to specify this, but if your list items are very short, the default of 20 might not be enough, as scrolling might overrun the framework's ability to fill the display list items.

For efficiency, the framework needs to limit the number of rendered list items to something reasonable. It can't just render all items, or there will be an impact on both memory and system performance. On the other hand, there must be enough items rendered to avoid having the list scrolling overrun the display list.

The list's model items are set to the input feed's `stories` array for display in the list, and `setupWidget` is called to instantiate the list. A listener is added for any taps on the list, and the handler, `readStory`, will push the `storyView` scene with that selected story entry.

In the `setup()` method, the list title is assigned to display in the header. You'll notice that we use the Prototype function `$()` to retrieve the header's element ID. This is safe to use in this context, but as you'll see in Chapter 10, it's not safe in multistage applications.

In the `activate()` method, we provisionally update the list's model in case reading the selected story changed the story's `unreadStyle` to read; we want to reflect changes in status immediately.

Next we have to change the *stage-assistant.js* to push the `storyList` scene instead of the `storyView` scene:

```
StageAssistant.prototype.setup = function() {
    // initialize the feeds model and update the feeds
    this.feeds = new Feeds();

    // Push the first scene
    this.controller.pushScene("storyList", this.feeds.list, 0);
};
```

For arguments, the `storyList` scene takes the feed list and an index value for the currently selected list. We're still using a single default list for now, so the index is set to 0.

Finally, add the new assistant to *sources.json*, then launch the application. The new scene with all the stories from the sample feed is shown in Figure 3-7.

Back to the News: Ajax requests

Chapter 6 covers Ajax requests more completely, but we'll look briefly at it here to enable dynamic feed lists. Now that we have a list, we're going to add the capability to load the list and update it through Ajax requests to the feed source.

Ajax requests are a common way of referring to use of the XMLHttpRequest object to make asynchronous HTTP transactions. The Prototype library built into Mojo provides an Ajax.Request object, which simplifies the XMLHttpRequest handling for many transactions.

These transactions provide a key part of building webOS applications by providing the core data services needed to build connected applications. You don't need to use the Prototype Ajax functions if you'd prefer to use XMLHttpRequest directly.

Figure 3-7. A storyList scene

Dynamic data is a very powerful and important capability that should be exploited by most applications. With the capability to update your application's data set, you are enabling the user with the most current and accurate information. Without this, the application loses value, as the degree of change is considerable over the course of hours, or even minutes in some cases.

You can write your own Ajax interfaces, but one reason that webOS includes the Prototype library is for its simple, powerful Ajax functions. We'll add the Ajax request to the *feeds.js* model, which will request feed data for our default New York Times feed. While the Ajax request is fairly simple, we need to process the RSS and Atom data that the application receives, and that's a bit more complicated.

We just need to add a URL for the Ajax request and set up some callback functions. See Chapter 6 for a full explanation of the arguments and properties used in Ajax.Request. Add a new method to *feeds.js*:

```
// updateFeedRequest - function called to setup and make a feed request
updateFeedRequest: function(currentFeed) {
    this.currentFeed = currentFeed;
    Mojo.Log.info("URL Request: ", this.currentFeed.url);

    var request = new Ajax.Request(currentFeed.url, {
        method: "get",
        evalJSON: "false",
        onSuccess: this.updateFeedSuccess.bind(this),
        onFailure: this.updateFeedFailure.bind(this)
    });
},
```

Ajax requests are asynchronous operations, with both success and error cases, and you'll need to create callback functions for each of these cases. The handler for the error case simply logs the error. Ajax requests return an HTTP status message, which we will convert to a readable format with Prototype's `Template` function, and then log the results:

```
// updateFeedFailure - Callback routine from a failed AJAX feed request;
//   post a simple failure error message with the http status code.
updateFeedFailure: function(transport) {
    // Prototype template to generate a string from the return status.xs
    var t = new Template("Status #{status} returned from newsfeed request.");
    var m = t.evaluate(transport);

    //    Post error alert and log error
    Mojo.Log.info("Invalid feed - http failure, check feed: ", m);
},
```

The handler for the successful case needs to process the feed before it can be used. In this case, we confirm the successful load by logging the returned status message, again using the `Template` function. Next, there's some code to handle when the feed data is returned as text-encoded XML; we can convert it to XML to enable processing.

The global function `ProcessFeed` is called to determine the feed format and extract the components that we need for our feed list. We'll cover this in a moment, but for now, note that it is called and returns with an explicit error status that, if equal to `error None`, means that the feed was processed successfully. We push the `storyList` scene with the processed feed in that case:

```
// updateFeedSuccess - Successful AJAX feed request (feedRequest);
//   uses this.feedIndex and this.list
updateFeedSuccess: function(transport) {

    var t = new Template({key: "newsfeed.status",
      value: "Status #{status} returned from newsfeed request."});
    Mojo.Log.info("Feed Request Success: ", t.evaluate(transport));

    // Work around due to occasional XML errors
    if (transport.responseXML === null && transport.responseText !== null) {
            Mojo.Log.info("Request not in XML format - manually converting");

// ** These next two lines are wrapped for book formatting only **
            transport.responseXML = new DOMParser().
              parseFromString(transport.responseText, "text/xml");
     }

    // Process the feed, passing in transport holding the updated feed data
    var feedError = this.processFeed(transport, this.feedIndex);

    // If successful processFeed returns News.errorNone,
    if (feedError !== News.errorNone)    {
        // There was a feed process error; unlikely, but could happen if the
        // feed was changed by the feed service. Log the error.
        if (feedError == News.invalidFeedError)    {
```

```
            Mojo.Log.info("Feed ", this.nameModel.value,
             " is not a supported feed type.");
        }
    }

    News.feedListChanged = true;

    // If NOT the last feed then update the feedsource and request next feed
    this.feedIndex++;
    if(this.feedIndex < this.list.length) {
        this.currentFeed = this.list[this.feedIndex];
        this.updateFeedRequest(this.currentFeed);
    } else {
        // Otherwise, this update is done. Reset index to 0 for next update
        this.feedIndex = 0;
        News.feedListUpdateInProgress = false;

    }
}
```

`this.processFeed()` is included in Appendix D if you're interested in how it works, but it's not shown here, since it doesn't directly affect the Mojo functions being presented. To summarize, `this.processFeed()` is passed an XML object and an index into the feed list, where it will put the processed feed. If there's no index argument, `this.processFeed()` will add the new feed to the end of the list.

For each of the supported formats, the title, text, and URL are extracted for each of the stories and the feed list is updated with the new feed data, the stories, and the `unreadCount`. If the feed isn't a well-formed Atom, RSS 1 (RDF), or RSS 2 format, it will return with an error, `News.invalidFeedError`.

We've added some logic to the end of `updateFeedSuccess()` to handle multiple feeds and to flag that the feed list has been changed. We'll come back to these in the next section as we expand News to handle multiple feeds.

Initiate the update with a call from within the stage assistant's setup method and add the global definitions needed for feed updates:

```
//  ---------------------------------------------------------------
//    GLOBALS
//  ---------------------------------------------------------------

//  News namespace
News = {};

// Constants
News.unreadStory = "unReadStyle";
News.versionString = "1.0";
News.errorNone =    "0";                    // No error, success
News.invalidFeedError = "1";                // Not RSS2, RDF (RSS1), or ATOM

//  Session Globals - not saved across app launches
News.feedListChanged = false;               // Triggers update to db
News.feedListUpdateInProgress = false;      // Feed update is in progress
```

```
StageAssistant.prototype.setup = function() {

    // initialize the feeds model and update the feeds
    this.feeds = new Feeds();

    // Update the news feed list
    this.feeds.updateFeedRequest(this.feeds.list[0]);

    // Push the first scene
    this.controller.pushScene("storyList", this.feeds.list, 0);

};
```

When the application is launched, it displays the default data in the top-level scene. If you tap a story to go to the story view, you'll see new stories, though. Popping the story view with a back gesture restores the story list view, but now with the updated stories. What's happening here?

Since Ajax requests are asynchronous, the initial story list view is pushed before the feed update is completed, but subsequent views are displayed after receiving the data. The right way to fix this is to update the `storyListWgt` model after the feed update is complete. You'll learn one technique for that in the next section, and a better one in Chapter 10, when we adapt the feed update process to a background application. The new `storyList` scene, when updated with a longer list of stories, is shown in Figure 3-8.

Figure 3-8. A storyList scene with updated stories

Back to the News: Adding a feed list

A News reader that handles one newsfeed isn't much use, so we're going to expand News to handle multiple feeds with another List widget. This one will present a list of newsfeeds for the user to select from before pushing the `storyList` scene with the selected list. We will also take advantage of the List widget's capability to reorder and delete list entries to enable some management of the newsfeeds.

We're still working from a default set of newsfeeds, but let's expand our feeds model by adding some popular news, sports, and technology feeds:

```
//  Default Feeds.list
    defaultList: [
            {
                title:"Huffington Post",
                url:"http://feeds.huffingtonpost.com/huffingtonpost/raw_feed",
                type:"atom", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Google",
                url:"http://news.google.com/?output=atom",
                type:"atom", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"New York Times",
                url:"http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml",
                type:"rss", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"MSNBC",
                url:"http://rss.msnbc.msn.com/id/3032091/device/rss/rss.xml",
                type:"rss", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"National Public Radio",
                url:"http://www.npr.org/rss/rss.php?id=1004",
                type:"rss", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Slashdot",
                url:"http://rss.slashdot.org/Slashdot/slashdot",
                type:"rdf", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Engadget",
                url:"http://www.engadget.com/rss.xml",
                type:"rss", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"The Daily Dish",
                url:"http://feeds.feedburner.com/andrewsullivan/rApM?format=xml",
                type:"rss", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Guardian UK",
                url:"http://feeds.guardian.co.uk/theguardian/rss",
                type:"rss", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Yahoo Sports",
                url:"http://sports.yahoo.com/top/rss.xml",
                type:"rss", numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"ESPN",
```

```
            url:"http://sports-ak.espn.go.com/espn/rss/news",
            type:"rss", numUnRead:0, newStoryCount:0, stories:[]
        },{
            title:"Ars Technica",
            url:"http://feeds.arstechnica.com/arstechnica/index?format=xml",
            type:"rss", numUnRead:0, newStoryCount:0, stories:[]
        }
    ],
```

We have to adjust the Ajax requests to make requests serially for each feed. We'll add a new function, `updateFeedList` to the feeds model:

```
// updateFeedList(index) - called to cycle through feeds. This is called
//   once per update cycle.
updateFeedList: function(index) {
    Mojo.Log.info("Feed Update Start");
    News.feedListUpdateInProgress = true;

    // request fresh copies of all stories
    this.currentFeed = this.list[this.feedIndex];
    this.updateFeedRequest(this.currentFeed);
},
```

Next, we'll create the feed list scene, which will display the list of feeds. Use `palm-generate` to create the scene and in the *feedList-scene.html* file, add a header titled 'Latest News' and the List widget declaration for `feedListWgt`:

```
<div id="feedListScene">
    <div id="feedListMain">

        <div id="feedList_view_header" class="palm-header left">
            Latest News
        </div>
        <div class="palm-header-spacer"></div>

        <!--    Feed List                                          -->
        <div class="palm-list">
            <div x-mojo-element="List" id="feedListWgt"></div>
        </div>

    </div>
</div>
```

To format the list, we need the list templates, which in this case are put into the *views/feedList* directory. First the container template, *feedListTemplate.html*:

```
<div class="palm-list">#{-listElements}</div>
```

and then *feedRowTemplate.html* to format the individual list entries:

```
<div class="palm-row" x-mojo-touch-feedback="delayed">
  <div class="palm-row-wrapper textfield-group">
    <div class="title">

      <div class="palm-dashboard-icon-container feedlist-icon-container">
        <div class="dashboard-newitem feedlist-newitem">
          <span class="unReadCount">#{numUnRead}</span>
```

```
        </div>
        <div id="dashboard-icon" class="palm-dashboard-icon feedlist-icon">
        </div>
      </div>

      <div class="feedlist-title truncating-text">#{title}</div>
      <div class="feedlist-url truncating-text">#{-url}</div>

    </div>
  </div>
</div>
```

Create the feed list assistant (*feedList-assistant.js*), which primarily completes the widget setup and adds event listeners in the file before calling the new `updateFeedList` method. All this is done within the assistant's setup method:

```
/*  FeedListAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Main scene for News app. Includes AddDialog-assistant for handling
    feed entry and then feedlist-assistant and supporting functions.

    Major components:
    - FeedListAssistant; manages feedlists
    - List Handlers - delete, reorder and add feeds

    Arguments:
    - feeds; Feeds object

*/

// ---------------------------------------------------------------------
//
// FeedListAssistant - main scene handler for news feedlists
//
function FeedListAssistant(feeds) {
    this.feeds = feeds;
}

FeedListAssistant.prototype.setup =  function() {

    // Setup the feed list, but it's empty
    this.controller.setupWidget("feedListWgt",
         {
            itemTemplate:"feedList/feedRowTemplate",
            listTemplate:"feedList/feedListTemplate",
            swipeToDelete:true,
            renderLimit: 40,
            reorderable:true
        },
        this.feedWgtModel = {items: this.feeds.list});

    // Setup event handlers: list selection, add, delete and reorder feed entry
    this.showFeedHandler = this.showFeed.bindAsEventListener(this);
```

```
    this.controller.listen("feedListWgt", Mojo.Event.listTap,
        this.showFeedHandler);
    this.listDeleteFeedHandler = this.listDeleteFeed.bindAsEventListener(this);
    this.controller.listen("feedListWgt", Mojo.Event.listDelete,
        this.listDeleteFeedHandler);
    this.listReorderFeedHandler = this.listReorderFeed.bindAsEventListener(this);
    this.controller.listen("feedListWgt", Mojo.Event.listReorder,
        this.listReorderFeedHandler);

    // Update the feed list
    this.feeds.updateFeedList();
};

// cleanup - always remove event listeners
FeedListAssistant.prototype.cleanup =  function() {
    Mojo.Log.info("FeedList cleaning up");

    // Remove event listeners
    this.controller.stopListening("feedListWgt",
        Mojo.Event.listTap, this.showFeedHandler);
    this.controller.stopListening("feedListWgt",
        Mojo.Event.listDelete, this.listDeleteFeedHandler);
    this.controller.stopListening("feedListWgt",
        Mojo.Event.listReorder, this.listReorderFeedHandler);

};
```

You'll see that we added both the `reorderable` and `swipeToDelete` properties to the `feedListWgt` list widget. A tap-and-hold on a list item will allow the user to move it to a new position in the list. A `Mojo.Event.listReorder` event is fired on the widget div, which includes the item being moved, as well as the old and new indexes. The indexes are passed as properties of the event object, `event.toIndex` and `event.fromIndex`.

Dragging items horizontally will invoke a special delete UI, allowing the user to confirm or cancel the operation. If confirmed, a `Mojo.Event.listDelete` event is fired on the widget div, which includes the item being removed, `event.item`, and its index, `event.index`.

We added event listeners for the `Mojo.Event.listDelete` and `Mojo.Event.listReorder`, and need to provide handlers for these events:

```
// ---------------------------------------------------------------------
// List functions for Delete, Reorder and Add
//
// listDeleteFeed - triggered by deleting a feed from the list and updates
// the feedlist to reflect the deletion
//
FeedListAssistant.prototype.listDeleteFeed =  function(event) {
    Mojo.Log.info("News deleting ", event.item.title, ".");

    var deleteIndex = this.feeds.list.indexOf(event.item);
    this.feeds.list.splice(deleteIndex, 1);
    News.feedListChanged = true;
};
```

```
// listReorderFeed - triggered re-ordering feed list and updates the
// feedlist to reflect the changed order
FeedListAssistant.prototype.listReorderFeed =  function(event) {
    Mojo.Log.info("com.palm.app.news - News moving ", event.item.title, ".");

    var fromIndex = this.feeds.list.indexOf(event.item);
    var toIndex = event.toIndex;
    this.feeds.list.splice(fromIndex, 1);
    this.feeds.list.splice(toIndex, 0, event.item);
    News.feedListChanged = true;
};
```

In both cases, the framework handles the on-screen changes, but you will need to reflect those changes in the feed model itself. Add listeners to receive the delete and reorder events, and you will receive the indexes for the changes through the event object. You then use these indexes to make the corresponding changes in the feed model.

The lists we set up for News are not using them, but there are other list manipulation options:

- If the `addItemLabel` property is specified, an additional item is appended to the list. Tapping it will cause a `Mojo.Event.listAdd` event to be fired on the widget div.
- Deleted items that are unconfirmed have a `deleted` property set in the model. You can specify the name of this property using the `deletedProperty` property, and `Mojo.Event.propertyChange` events will be sent when it is updated. If unspecified, the property `deleted` will be used. For dynamic lists, it is important for the application implementation to persist this value in a database. Otherwise, swiped items will be automatically undone when they are removed from the cache of loaded items.
- A better option than persisting the deleted property is using the `uniquenessProperty`. This is the name of an item model property that can be used to uniquely identify items. If specified, List will maintain a hash of swiped items instead of setting a deleted property, preventing the app from having to persist the deleted property.
- If the `dragDatatype` property is specified, users will be able to drag items to other lists with the same `dragDatatype` value. When this happens, the item's old list will receive a `Mojo.Event.listDelete` event, and the new list will get a `Mojo.Event.listAdd` event. In this case, the `Mojo.Event.listAdd` event will have the item and index properties specified, indicating that a specific item should be added at a specific location.

The other event handler, `showFeed()`, pushes the `storyList` scene when a `Mojo.Event.listTap` event is received, meaning that a feed has been tapped:

```
// -------------------------------------------------------------------------
// Show feed handler
//
// showFeed - triggered by tapping a feed in the this.feeds.list.
```

```
FeedListAssistant.prototype.showFeed = function(event) {
    Mojo.Controller.stageController.pushScene("storyList", this.feeds.list, event.index);
};
```

Before running the application, change the stage assistant to push the `feedList` scene:

```
    // Push the first scene
    this.controller.pushScene("feedList", this.feeds);
```

Change the stage controller to push the `feedList` scene, and when you run the application now, you'll see that it's starting to take the basic structure of the envisioned application. It has an initial scene that is a list of available feeds with a count of unread messages, which users can tap to view individual feeds and messages. We've made a lot of changes in this section, as the List widget has really opened up the application's feature set.

You'll notice that the list entry style is not complete, but we'll fix that with some CSS in *stylesheets/News.css*:

```
/* feedList styles */

.feedlist-title {
    line-height: 2.0em;
}

.feedlist-url {
    font-size: 14px;
    color: gray;
    margin-top: -20px;
    margin-bottom: -20px;
    line-height: 16px;
}

.feedlist-icon-container {
    height: 54px;
    margin-top: 5px;
}

.feedlist-icon {
    background: url(../images/list-icon-rssfeed.png) center no-repeat;
}

.feedlist-newitem   {
    line-height: 20px;
    height: 26px;
    min-width: 26px;
    -webkit-border-image: url(../images/feedlist-newitem.png) 4 10 4 10
        stretch stretch;
    -webkit-box-sizing: border-box;
    border-width: 4px 10px 4px 10px;
```

A few of these styles (`feedlist-icon-container`, `feedlist-icon`, and `feedlist-newitem`) are modified versions of the framework's standard dashboard styles. Those styles set up the icon and new items badge to the left of each feed. The other styles refine the positioning and appearance of the feed title and URL.

Now when you run the application the styling should look complete, but there is still a problem. As we saw with the `storyList` scene in the last section, the feed updates aren't reflected in the displayed list view until you tap a feed then return to the `feedList` scene.

We need to be able to update the list widget's model as each feed is updated. First, add activate and deactivate methods to the `feedList` scene:

```
// activate
FeedListAssistant.prototype.activate =  function() {
    this.feeds.registerListModel(this);

    if (News.feedListChanged === true)    {
        this.feedWgtModel.items = this.feeds.list;
        this.controller.modelChanged(this.feedWgtModel, this);
    }
};

// deactivate
FeedListAssistant.prototype.deactivate =  function() {
    Mojo.Log.info("FeedList deactivating");
    this.feeds.removeListModel(this.feedWgtModel);
};
```

In the `activate()` method, call to register this assistant with the feeds object so that it will update `this.feedWgtModel` when changes are made to the feed. Also add an update to the model for any activation of this scene. In this way, unread count changes are reflected whenever new stories are viewed in the `storyView` scene. In the `deactivate()` method, remove the registration whenever the `feedList` scene is replaced by another scene.

Then add these new methods to *feeds.js*, along with an `updateListModel()` method that will be called from within the feed update loop in `updateFeedSuccess()`:

```
// registerListModel(sceneAssistant) - called to register the list model for updates
//   as the underlying data changes.
registerListModel: function(sceneAssistant) {
    Mojo.Log.info("Model Registered");
    this.listAssistant = sceneAssistant;
},

// removeListModel() - called to remove the list model from updates
//   as the underlying data changes.
removeListModel: function() {
    Mojo.Log.info("Model Removed");
    this.listAssistant = undefined;
},
```

```
// updateListModel() - called to update the list.
updateListModel: function() {
    Mojo.Log.info("Model Updated");
    if (this.listAssistant !== undefined) {
       this.listAssistant.feedWgtModel.items = this.list;
       this.listAssistant.controller.modelChanged(this.listAssistant.feedWgtModel,
            this);
    }
},
```

Now when you run the application, the feed list widget is updated as the feed data is updated by the feeds object. You'll see the unread count (shown in the white badge on each feed) change to reflect the number of stories read in each feed and when fully loaded a view like the one in Figure 3-9.

Note that `modelChanged()` causes a full rendering of the widget. For small changes, it's better to use `noticeAddedItems()` or `noticeUpdatedItems()` to render only the changed elements.

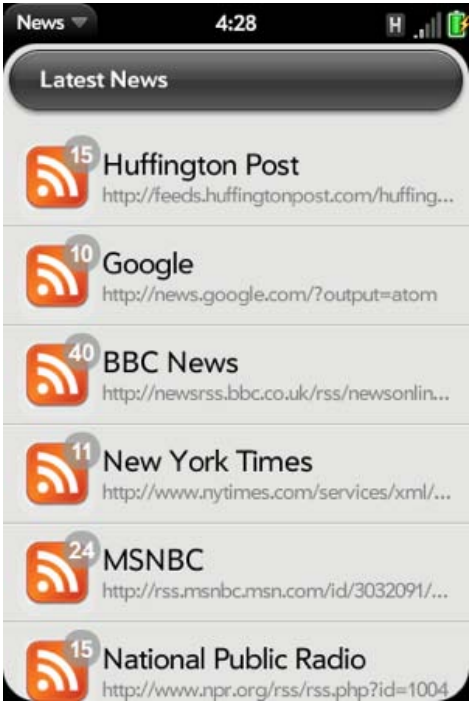


Figure 3-9. The feedList scene

We'll make one more change. The application updates the feed when launched, but it would be nicer to have the feeds update periodically. We'll set up an alarm during the stage assistant's setup method to fire after 15 minutes has elapsed, using the JavaScript `setTimeout()` method. We created a `setWakeup()` method to set the alarm and a `handleWakeup()` method as the callback when the alarm fires. The `handleWakeup()` method sets the next alarm and calls `this.feeds.updateFeedList()` to refresh all of the feeds:

```
//  ------------------------------------------------------
//  setup - all startup actions:
//    - Setup globals
//    - Initiate alarm for first feed update

StageAssistant.prototype.setup = function() {

    // initialize the feeds model and update the feeds
    this.feeds = new Feeds();

    // Set up first timeout alarm
    this.setWakeup();

    // Push the first scene
    this.controller.pushScene("feedList", this.feeds);

};

// ------------------------------------------------------------------------
// handleWakeup - called when wakeup timer fires; sets a new timer and calls
//   for a feed update cycle
StageAssistant.prototype.handleWakeup = function() {

    // Set next wakeup alarm
    this.setWakeup();

    // Update the feed list
    Mojo.Log.info("Update FeedList");
    this.feeds.updateFeedList();

};

// ------------------------------------------------------------------------
// setWakeup - called to setup the wakeup alarm for background feed updates
//   if preferences are not set for a manual update (value of 0)
StageAssistant.prototype.setWakeup = function() {

    if (News.feedUpdateInterval !== 0)    {
        var interval = News.feedUpdateInterval;
        News.wakeupTaskId =
          this.controller.window.setTimeout(this.handleWakeup.bind(this),
          interval);
        Mojo.Log.info("Set Interval Timer: ", interval);
    }

};
```

```
// ------------------------------------------------------------------------
// cleanup - clear the wakeup alarm for background feed updates if set
StageAssistant.prototype.cleanup = function() {
    if (News.wakeupTaskId !== 0)    {
       News.wakeupTaskId = this.controller.window.clearTimeout(News.wakeupTaskId);
       Mojo.Log.info("clear Interval Timer");
    }
};
```

Define the global `News.feedUpdateInterval` at the beginning of the stage assistant; we'll use it later when we add a preferences option to change the update interval. Don't forget to clear the timeout when the stage is closed, which in this case happens when the application is closed.

Using Widgets in Lists

You can define list entries to include other widgets, including other lists. The list's model is an object that includes an array of items, and each item entry may have properties that are referenced in the list's `itemTemplate`. You can declare widgets within the list's `itemTemplate`, using a name attribute to identify the widgets. In your setup method, you set up the list with a model object containing an array of objects for the list items. Each of these objects is used as the model for the widgets in the corresponding list item. After setting up the List widget itself, you should call `setupWidget()` once for each widget declaration in the item template. However, it's not necessary to specify the model when doing this, since it comes from the list items array.

For example, to create a list in which each list item or row has a text label and a toggle button, you define your list's `itemTemplate` with a template for the text and a declaration for the toggle button:

```
<div class="palm-row">
  <div class="palm-row-wrapper">
    #{text}
    <div name="listToggle" x-mojo-element="ToggleButton"></div>
  </div>
</div>
```

In the setup method, you define your list's model to include the toggle models:

```
this.firstModel = {text: "First", value: true};
this.secondModel = {text: "Second", value: true};
this.thirdModel = {text: "Third", value: true};
this.fourthModel = {text: "Fourth", value: true;

this.listModel = {items:[this.firstModel, this.secondModel, this.thirdModel,
  this.fourthModel]};
this.controller.setupWidget("myList", this.listAttr, this.listModel);
```

Then set up the toggle once with only an attributes object, as the model will be pulled from the list items above:

```
this.toggleAttr = { trueLabel: "On", falseLabel: "Off"};
this.controller.setupWidget("listToggle", this.toggleAttr);
```

Note that the widgets declared in the `itemTemplate` use a name attribute not an ID, because the same name can be used for each instantiation of the widget and an ID must be unique.

You can't set up a listener to the toggle, but you can listen to `Mojo.Event.propertyChange` on the list and have the model passed as an event property. For example:

```
this.controller.listen("myList", Mojo.Event.propertyChange,
    this.toggleChange.bindAsEventListener(this));
```

In your event listener, you would reference the toggle model this way:

```
MyAssistant.prototype.ToggleChange = function(event)  {
if (event.model.value === true) {
    ....
} else  {
    ....
}
};
```

In cases like this, multiple widgets must share the same model. All widgets allow you to specify `modelProperty` in their attributes to make it easier to use a shared model. For example, Toggle Button and List Selector both have a `modelProperty` property in their attributes.

The event handling is more complicated when the list items contain multiple subitems, including widgets. To deal with this, List supports `Mojo.Event.listTap` and `Mojo.Event.listChange` events. You can add event listeners to the widget div to listen to these list events, then analyze the event to determine which element in the List item is targeted by tracing the reference to the model object used for the particular list item that was clicked/changed, or examine the `event.target` property to see which element in the list item was affected. Also, you can add a `propertyChange` listener on the List div to get all `propertyChange` events from any widget in the list; check the "property" in the event to figure out what triggered it. The event also contains the model for the widget that sent the event.

More About Lists

There are several major features included with lists that aren't used with News lists, and there is another list widget: Filter List. We'll use Filter List in Chapter 5 to add a search list to News, but the other features will be briefly touched on here.

Dynamic lists

The List attributes can optionally include a callback function for supplying list items dynamically. You do not need to provide the items array objects at setup time; whenever the framework needs to load items (speculatively or for display), it will call the callback function `itemsCallback (listWidget, offset, limit)`, with the arguments described in Table 3-3.

Table 3-3. ItemsCallback arguments

Argument	Type	Description
`listWidget`	Object	The DOM node for the list widget requesting the `items`
`offset`	Integer	Index in the list of the first desired item model object (zero-based)
`limit`	Integer	The number of item model objects requested

It is understood that the requested data may not be immediately available. Once the data is available, the given widget's `noticeUpdatedItems()` method should be called to update the list. It's acceptable to call the `noticeUpdatedItems()` immediately, if desired, or any amount of time later. Lengthy delays may cause various scrolling artifacts, however. It should be called as `listWidget.mojo.noticeUpdatedItems(offset, items)`, using the arguments shown in Table 3-4.

Table 3-4. noticeUpdatedItems arguments

Argument	Type	Description
`offset`	Integer	Index in the list of the first object in `items`; usually the same as offset passed to the `itemsCallback`
`items`	Array	An array of the list item model objects that have been loaded for the list

Formatters and dividers

The formatters property is a simple hash of property names to formatter functions, like this:

```
{timeValue: this.myTimeFormatter, dayOfWeek: this.dayIndexToString, ... }
```

Before rendering the relevant HTML templates, the formatters are applied to the objects used for property substitution. The keys within the formatters hash are property names to which the formatter functions should be applied. The original objects are not modified, and the formatted properties are given new modified names so that the unformatted value is still accessible from inside the HTML template.

The divider function works similar to a data formatter function. It is called with the item model as the sole argument during list rendering, and it returns a label string for the divider. For example, the function `dividerAlpha` generates list dividers based on the first letter of each item:

```
dividerAlpha = function(itemModel) {
  return itemModel.data.toString()[0];
};
```

If you're defining your own template, you should insert the property `#{dividerLabel}` where you would want to have the label string inserted.

Text Fields

There is a legacy of great text-centric applications on Palm devices. The original Palm OS included a whole new writing system, Graffiti, to provide simple, effective tools for entering and editing text, and one of the Treō's hallmarks was a terrific "thumbable" keyboard and a system optimized for messaging and email applications. So naturally, Palm webOS has some powerful text features, including a simple text widget, to embed text in your applications.

This section will start with the Text Field (shown in Figure 3-10), the base text widget that supports all general text requirements: single-line or multi-line text entry, with common styles for labels, titles, headings, body text, line items, and item details. The editing tools include basic entry and deletion, symbol and alternate character sets, cursor movement, selection, cut/copy/paste, and auto text correction.

Figure 3-10. A Text Field widget example

In most cases, `TextField` will address your text needs, but there are three specialized widgets:

`PasswordField`
: Handles passwords or other confidential text input.

`FilterField`
: Supports type-down filters of an off-screen list or similar searchable data.

`RichTextEdit`
: A multi-line text field that supports simple text styles (bold, italic, and underline).

In all of the text widgets, the framework will handle all user interactions with the text field, returning the entered string when the field loses focus or the user keys *Enter* (where enabled). Mojo text fields are smart text fields by default. Autocapitalization and correction for common typing mistakes are performed on all fields unless explicitly disabled.

Smart Text Features

The Smart Text Engine (STE) refers to the automatic modification of user-entered text to allow quicker text input. When typing on the small keyboards that are usually

characteristic of mobile devices, users are more likely to make certain spelling mistakes. Furthermore, because text input for things like text messaging, notes, and contact information is often done in a hurry, users are more likely to forgo punctuation, forgo capitalization, and/or use common slang abbreviations (such as using "r" instead of "are" and "u" instead of "you").

The STE performs autocapitalization and autoreplacement. `Auto capitalization` will, by default, assert a Shift key state when it detects a punctuation character followed by a space during text entry, or can be set to force all caps or all lowercase. Autoreplacement works by checking each word against a file of substitution pairs, and, if found, a substitution is made.

Smart text is automatically enabled in all text fields except for password fields. If you want to disable smart text, you can set the `autoReplace` property to `false`, or you can set the `textCase` property to one of the following options:

`Mojo.Widget.steModeSentenceCase` (default)
`Mojo.Widget.steModeTitleCase`
`Mojo.Widget.steModeLowerCase`

`Emoticons` is another property, which will direct the STE to substitute bitmap images in place of common emoticon text strings, such as ☺ for :) among many others.

There are a number of ways to style text fields, depending on whether you are grouping fields together as you would for a form or using them singly or within other widgets. Chapter 7 has more information on styling text fields as well as other advanced styling topics.

Adding Text Fields to News

We only have one example of a text field in the News application: adding a newsfeed requires text fields to enter the feed URL and name. The Text Fields will be put below the list of feeds within the `feedList` scene and a Drawer widget will hide the Text Fields until triggered by the Add Feed action on the `feedListWgt` widget.

Drawers are container widgets that can be *open*, allowing child content to be displayed normally, or *closed*, keeping it out of view. The state of the drawer depends on a single model property, although there are also exposed widget functions (`toggleState`, `getOpenState`, and `setOpenState`) available for opening and closing a drawer.

Add the Text Field declarations within a styled `palm-group` into the *feedList-scene.html* file. We're going to wrap the Text Fields with the Drawer and several layers of styling div tags:

```
<div id='feedDrawer' x-mojo-element="Drawer">
    <div id="add-feed-title" class="palm-dialog-title">
        Add Feed
    </div>

    <div class="palm-list">
```

```
        <div class="palm-row first">
            <div class="palm-row-wrapper textfield-group"
              x-mojo-focus-highlight="true">
                <div class="title">
                   <div x-mojo-element="TextField" id="newFeedURL"></div>
                </div>
            </div>
        </div>
        <div class='palm-row last'>
            <div class="palm-row-wrapper textfield-group"
              x-mojo-focus-highlight="true">
                <div class="title">
                    <div x-mojo-element="TextField" id="newFeedName"></div>
                </div>
            </div>
        </div>
    </div>
    <div x-mojo-element="Button" id="okButton"></div>
    <div x-mojo-element="Button" id="cancelButton"></div>
</div>
```

At the beginning of the file there is a style class to create the "Add Feed" title. The next style class, `palm-list`, creates a list-style group with row dividers into which we'll put our text fields. We wrap each of the fields with `palm-row` and `palm-row-wrapper` classes and add a div with the `title` class to complete the styling. The text field widgets are declared within all those layers of styling classes. Button widgets are declared at the bottom to approve the feed entry and submit it for addition to the list, or to cancel the action and close the drawer.

Next, set up the Drawer, Text Fields, and Buttons in the setup method of the *feedList-assistant.js*:

```
//  Setup Drawer for add Feed; closed to start
this.controller.setupWidget('feedDrawer', {}, this.addDrawerModel={open: false});

// Set the add feed drawer title to Add Feed
var addFeedTitleElement = this.controller.get("add-feed-title");
addFeedTitleElement.innerHTML = "Add News Feed Source";

// Setup text field for the add new feed's URL
this.controller.setupWidget(
    "newFeedURL",
    {
          hintText: "RSS or ATOM feed URL",
          autoFocus: true,
          autoReplace: false,
          textCase: Mojo.Widget.steModeLowerCase,
          enterSubmits: false
    },
    this.urlModel = {value : ""});

// Setup text field for the new feed's name
this.controller.setupWidget(
    "newFeedName",
```

```
    {
          hintText: "Title (Optional)",
          autoReplace: false,
          textCase: Mojo.Widget.steModeTitleCase,
          enterSubmits: false
    },
    this.nameModel = {value : ""});

// Setup OK  & Cancel buttons
//   OK button is an activity button which will be active
//   while processing and adding feed. Cancel will just cancel the
//   action and close the scene
this.okButtonModel = {label: "OK", disabled: false};
this.controller.setupWidget("okButton", {type: Mojo.Widget.activityButton},
    this.okButtonModel);
this.okButtonActive = false;
this.okButton = this.controller.get("okButton");
this.checkFeedHandler = this.checkFeed.bindAsEventListener(this);
this.controller.listen("okButton", Mojo.Event.tap,
  this.checkFeedHandler);

this.cancelButtonModel = {label: "Cancel", disabled: false};
this.controller.setupWidget("cancelButton", {type: Mojo.Widget.defaultButton},
    this.cancelButtonModel);
this.closeAddFeedHandler = this.closeAddFeed.bindAsEventListener(this);
this.controller.listen("cancelButton", Mojo.Event.tap,
  this.closeAddFeedHandler);
```

The first `setupWidget` call creates the Drawer, which is initially closed. The next `setupWidget` creates the URL field with some hint text and setting focus to the field. The name field is set up with the hint text indicating that the field is optional—if not entered, we'll use the name provided in the feed after it's loaded.

The first button is set up with an OK label and declared as an activity button, which will be used to show activity while we are checking and loading the feed. A second button is set up to cancel the operation.

We still need a selector to open the Drawer. The List widget has an ideal feature to use for that selector, the Add Item option, which generates a `Mojo.Event.listAdd` event when tapped. Insert the `addItemLabel` property to the `feedListWgt` setup to enable a selector to add a new feed. You will see that new property added to our previous setup function, just below the `renderLimit` property:

```
this.controller.setupWidget("feedListWgt",
   this.feedWgtAttr = {
         itemTemplate: "feedList/feedRowTemplate",
         listTemplate: "feedList/feedListTemplate",
         swipeToDelete: true,
         renderLimit: 40,
         addItemLabel: "Add...",
         reorderable: true
   },
   this.feedWgtModel = {items: feedList});
```

Add a listener for the `Mojo.Event.listAdd` event and specify a handler to open the Drawer:

```
// addNewFeed - triggered by "Add..." item in feed list
FeedListAssistant.prototype.addNewFeed = function() {
    this.addDrawerModel.open = true;
    this.controller.modelChanged(this.addDrawerModel);
};
```

When the user taps the Add item at the end of the list, the `listAdd` event causes the `addNewFeed` handler to open the Drawer. From there, the feed's URL is entered, the Add Feed button tapped to generate a tap event on the button, and the `checkIt` handler called to process the feed. The drawer will stay open until a feed is completely added or the user taps the Cancel button.

Since we're demonstrating the Text Field widget, we haven't included all the code for `this.checkIt`, but you can refer to Appendix D, where the News application source is reproduced in it's entirety. Just know that the version of `this.checkIt` in Appendix D is built for use within a dialog, which is eventually where this Add Feed function will be handled. If you try to use it in this drawer case, you'll need to remove the `sceneAssistant` references for all the scene controller method calls.

The handlers will submit an Ajax request for the entered feed. If it's a valid feed, `ProcessFeed` will be called with the result and will add the processed feed to the end of the feed list. The Drawer is closed at the end if the feed is added successfully. Figure 3-11 shows the new `feedList` scene with the Drawer in the open position and the text fields.

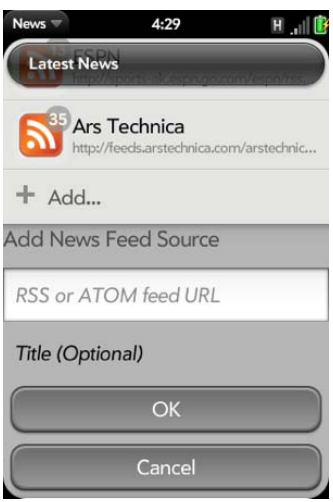


Figure 3-11. A feedList scene with a text field

Password Field

If you need a text field that will be used for passwords or some other type of confidential information, the Password Field provides many of the Text Field features, but masks the display. Any entered text is displayed as a bullet (•) character. As with the Text Field, the framework handles all of the editing logic within the field and generates a `Mojo.Event.propertyChange` event when the field has been updated. Figure 3-12 shows an example of a Password Field widget.

Figure 3-12. A Password Field widget example

Palm webOS Editing

The Palm Prē phone has a "slideout" keyboard, which was an incentive to include some powerful editing features with webOS. In addition to the advantages of keyboarding on thumbable keyboards, all text fields support trackball mode cursoring, smart deletion, and text selection.

Trackball mode (which users access by holding the Alt or Orange key while swiping) lets users use swipes to move the cursor across the text to the desired location, while text selection (holding the Shift key while swiping), will highlight selected text for deletion, replacement, or cut/copy/paste operations.

Smart deletion (holding the Shift key while deleting) will delete whole words at a time rather than one character at a time.

All this support comes with using Mojo's text fields, along with the Smart Text features discussed in this section.

Filter Field

If you require a text field to filter down the contents of an offline list, you can use the Filter Field. It can be applied to any case where you want to process the field contents and update on-screen elements based on the entered string.

Filter Field is hidden until displayed by the framework in response to the user entering text when there isn't focus on any text field. In other words, the filter field is given focus for any text input on scene where it is present and another text field hasn't been explicitly been given focus.

Along with displaying the field, the framework will call a provided filter function to handle the entered text after a specified delay. It's up to you to respond appropriately, but the framework will continue to display new text input and to call the filter function until the field is closed.

Rich Text Edit

There is a simple Rich Text Edit widget (see Figure 3-13), which is similar to a multiline text field, but also supports applying bold, italic, and underline styles to arbitrary runs of text within the field.

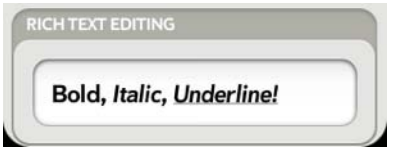


Figure 3-13. A Rich Text Edit widget example

To create support for this styling, enable the `RichTextEditItems` property in the Application menu (see Chapter 4 for information on the Application menu). The user will then be able to apply bold, italic, and underline style to the current text selection.

Events

The World Wide Web Consortium (W3C) HTML event model provides a way to respond to user actions. In the model, which is part of the HTML DOM, user actions can be associated with a DOM element. When an action occurs on the DOM element, the browser generates an event and invokes the JavaScript code subscribed to the element, for a particular event type. The W3C HTML event model defines standard event types, such as load, mouseover, click, and resize, corresponding to user actions.

The framework implements an event model very similar to the W3C HTML event model. One difference is that the framework defines event types at a higher level of abstraction, representing actions meaningful to the UI model and framework widgets.

Framework Event Types

Mojo defines unique event types supporting different parts of the UI system:

- System UI events, such as drag, flick, and hold
- Widget events, including `listTap`, `propertyChange`, and more

- Application UI events, such as `scrollStarting`, `stageActivate`, and `stageDeactivate`

You should refer to the API documentation, specifically Mojo.Event, for a complete list with descriptions and references to the event object properties and related information. For the System and Application UI event types, the meaning of the event depends on the context in which the event occurs, and you need to handle the event accordingly.

The framework also provides a way to define custom events and propagate events to an event handler through `Mojo.Event.make` and `Mojo.Event.send`.

Listening

When an event occurs, all code that is subscribed to handle the event is notified. You can subscribe to events on any DOM element by calling one of the following methods:

- `Mojo.Event.listen()` or `this.controller.listen()`
- `<DOM Element ID>.addEventListener()`
- `observe()`

These methods are roughly equivalent, differing only in call semantics. `Mojo.Event.listen` was created as a contingency for issues with either `addEventListener` or Prototype's `observe` method, but at this point, all work equally well with Mojo.

There is an issue with referencing elements by DOM ID. The Prototype `$` and `getElementById` won't work across Stage boundaries, so if you have a multistage application, you will need to use `this.controller.listen()` if you pass an element by DOM ID, or `this.controller.get()` when you want to retrieve an element by DOM ID.

As in the standard HTML model, events bubble up the DOM tree and the parent DOM element receives the events that occur on any child elements. For controls, this implies that you should observe events on the enclosing div element instead of on an element that is part of the control implementation.

The following code snippets show how to subscribe to events using `this.controller.listen`:

1. Define the HTML DOM element associated with an event and assign an ID to the element:

```
<div id="thanksButton">Thank you</div>
```

2. Provide a JavaScript method to handle the event:

```
MySceneAssistant.prototype.handleThanks = function() {
  this.sceneAssistant.outputDisplay.innerHTML = "Thanks";}
```

This is the handler specified when the user subscribes to the event. The method is invoked by the browser when the event occurs. You should provide the event handling logic appropriate for the event and application context.

3. Subscribe to the event, using the element ID and specifying the event handler method:

```
this.controller.listen("thanksButton", Mojo.Event.tap,
  this.handleThanks.bindAsEventListener(this));
```

You will typically need to use `bind` or `bindAsEventListener` on your event listeners. The JavaScript *this* keyword will be set to either the window or the DOM element when your event listener is called. You will want to use `bind` or `bindAsEventListener` to make sure that the *this* keyword will point to the same scene assistant instance that registered the handler.

stopListening

Use one of the following methods to remove your listener from events:

- `Mojo.Event.stopListening()` or `this.controller.stopListening()`
- `<DOM Element ID>.removeEventListener()`
- `stopObserving()`

You should use the method that corresponds to the method used to set up listening for the event; in other words, use the same Mojo method to stop listening that you used to initiate listening.

With any of these methods, you must use the exact handler reference used in the listen method call. In the above example, the handler was specified as `this.handleThanks.bindAsEventListener(this)`, which won't work in the `stopListening` method. Try this instead:

```
this.eventHandler = this.handleThanks.bindAsEventListener(this);
this.controller.listen("thanksButton", Mojo.Event.tap, this.eventHandler);

.
.
.

this.controller.stopListening("thanksButton", Mojo.Event.tap, this.eventHandler);
```

Note that if you include the `useCapture` argument when setting up your listener, you must also include it with the `stopListening` call in exactly the same way.

There are two ways to *leak* memory in JavaScript. The first is to have a circular reference between the closure of an event handler and the node that you call `addEventListener` on. Since the event won't be unregistered until the node is destroyed, and the node won't be destroyed until the JavaScript reference goes away, it causes a memory leak.

The second form of leaking is more straightforward, which is to have a persistent reference to a closure (like `setInterval` or a Mojo service subscription) that has a direct or indirect reference to a DOM node. Those objects and nodes will also live forever.

Using Events with Widgets

Many widgets dispatch events. Applications can use these to better leverage the functionality built into the controls. Events are generally dispatched to the widget's element, the div defined in the scene's view HTML that has the `x-mojo-element` attribute. Appendix B enumerates all the specific events propagated by each widget in options tables accompanying each widget's description.

Summary

Widgets are signature components provided by Mojo that enable your applications with a powerful UI that has the look and feel of webOS. Using common techniques, you can customize widget behavior and appearance around your specific needs by manipulating widget settings along with their corresponding events and styles.

In this chapter, we've looked at the widget design and covered the general methodology for declaring, instantiating, rendering, and updating widgets. The News application has been extended to include buttons, lists, and text fields, and we've covered each of those widget types in detail. We've also covered event handling and style overrides, and by now you should have a good idea how to use a widget within your application.

With these basic widgets, you can write some simple applications. But you will also need menus and dialog boxes, which we'll cover in the next chapter, to write meaningful, UI-complete applications. With what you've learned so far, however, it wouldn't hurt to write some sample applications to familiarize yourself with stages, scenes, widgets, and event handling. These basics will be used throughout any webOS application.

CHAPTER 4

Dialogs and Menus

Familiar components in every UI framework, dialog boxes and menus are used by almost all applications. Mojo's Dialog and Menu widgets provide the expected features, but have some unique additions. With custom dialogs, you can include any web content in a dialog box. Additionally, you can customize menus by scene and present them as either conventional drop-down lists or floating elements.

Dialogs and menus are both fundamental widgets, though more complex than the basic widgets covered in Chapter 3, and they are accessed and managed differently than other widgets. Dialogs are instantiated through controller functions rather than through `setupWidget`, and `showDialog` requires an assistant as one of its components.

Menus are instantiated by `setupWidget`, but use the *Commander Chain* to propagate menu commands between stage assistants and scene assistants. The Commander Chain is a model for propagating commands through the application, stage, and scene controllers, and is described in detail near the end of this chapter.

As in Chapter 3, we'll explore these widgets in the context of adding them into the News application. This will be accompanied by a general description and some screenshots.

Dialogs

You can use a Dialog widget to create a modal view for almost any purpose. A *custom dialog* is a conventional dialog, but it requires a widget assistant and an HTML template file. The Dialog widget is dynamically instantiated within a scene assistant, so there is a bit of overhead in using it both for you as a developer and at runtime. For errors, you should use the Error dialog. For presenting some simple options, use an Alert dialog. The simple built-in dialogs will be presented first, followed by a discussion of how to build custom dialogs with `showDialog`.

Error Dialog

You can post error messages in a modal dialog box with a fixed title of "Error," a customizable message, and a confirmation button. The Error dialog must be used only with errors, since you can't change the title; an example is shown in Figure 4-1.

You can post an Error dialog box with a single call:

```
Mojo.Controller.errorDialog("Invalid Feed", this.controller.window);
```

Figure 4-1. An Error dialog box

Logging Methods

Mojo includes logging methods to give you an efficient way to generate console output without degrading the performance of your application or the system. There are three log levels:

```
Mojo.Log.info();    // Mojo.Log.LOG_LEVEL_INFO = 20
Mojo.Log.warn();    // Mojo.Log.LOG_LEVEL_WARNING = 10
Mojo.Log.error();   // Mojo.Log.LOG_LEVEL_ERROR = 0
```

Only messages at or below the current logging level are generated. The current logging level is set as a configuration option in a new file, *framework_config.json*. To allow all log levels, set the `logLevel` property to `99`:

```
{
"logLevel": 99
}
```

For shipping code, do not set the limit above 0, as logging overhead will contribute to slow performance on the application and the system.

Unlike `console.log`, the arguments to `Mojo.Log` are passed individually to the log functions and only turned into strings if the message is actually printed to the console. Take the following code, for example:

```
Mojo.Log.info("I have", 3, "eggs.");
```

This would output:

```
I have 3 eggs.
```

There is also support for a limited number of formatting characters and for adding log methods to individual objects. Look at the following code:

```
var favoriteColor = 'blue';
Mojo.Log.info("My favorite color is %s.", favoriteColor);
```

The output would be:

```
My favorite color is blue.
```

You can use `%s`, `%d`, `%f`, `%i`, `%o`, and `%j`. The first four produce the same result; coercing the appropriate parameter to a string for logging. `%o` converts the parameter to a string using Prototype's `Object.inspect()`, while `%j` converts it using `Object.toJSON()`.

Alert Dialog

You can display a short message using an Alert dialog, with one or more HTML buttons presenting the selection options. This is the best option if you have either a message for the user, other than an error message, or want to present options that can be selected in the form of button selections:

```
this.controller.showAlertDialog({
    onChoose: function(value) {
        this.outputDisplay.innerHTML = "Alert result = " + value;
    },
    title: "Filet Mignon",
    message: "How would you like your steak done?",
    choices:[
      {label: "Rare", value: "rare", type: "affirmative"},
      {label: "Medium", value: "medium"},
      {label: "Overcooked", value: "overcooked", type: "negative"},
      {label: "Nevermind", value: "cancel", type: "dismiss"}
    ]});
```

This example presents four choices, as shown in Figure 4-2. Each button is labeled, with an optional button type corresponding to a `palm-button` class, and returns a value string.

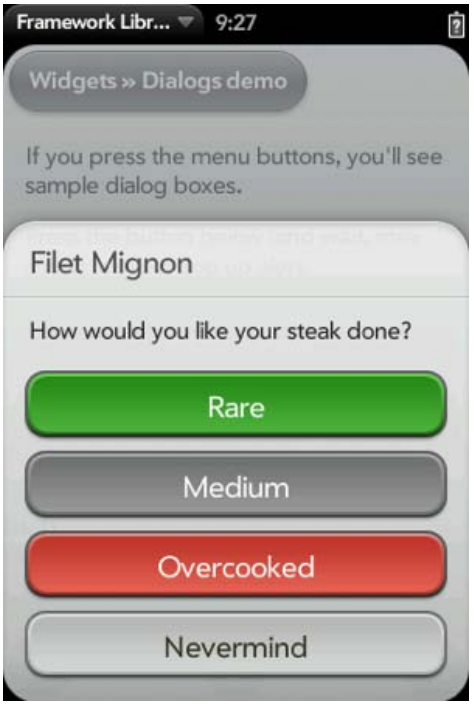


Figure 4-2. An Alert dialog box

Custom Dialogs

If the two simple dialogs don't meet your needs, you can use the `showDialog` function, which can display any type of content to the user in the form of a modal dialog box. You can put anything into a custom dialog that you'd put into a scene, meaning almost any web content or Mojo UI content.

Back to the News: Adding an Add Feed dialog

In the previous chapter, we added a Drawer in the `FeedListAssistant` to support the Add Feed feature. It would be better to put this type of feature in a dialog; we will create an Add Feed dialog with the `showDialog` function and move the code used in the Drawer into the dialog.

Begin by replacing the `addNewFeed` method in *feedlist-assistant.js* with a call to `showDialog()`:

```
//  addNewFeed - triggered by "Add..." item in feed list
FeedListAssistant.prototype.addNewFeed = function() {

   this.controller.showDialog({
```

```
        template: 'feedList/addFeed-dialog',
        assistant: new AddDialogAssistant(this, this.feeds)
    });
};
```

The arguments specify the dialog template and a reference to the assistant that handles the dialog. We create a new instance of the `AddDialogAssistant`, passing a reference to the `FeedListAssistant` and `this.feeds`, the feed object, and pass that in along with a reference to our `addFeed-dialog.html` template. The dialog template is simply an HTML template, but you should make use of some of the standard dialog box styles such as `palm-dialog-content`, `palm-dialog-title`, `palm-dialog-separator`, and `palm-dialog-buttons` to format and style your dialog boxes to fit in with webOS UI guidelines.

Create the HTML for the `addFeed-dialog` template by moving the code used in the previous chapter from *feedList-scene.html* to a new file, *views/feedList/addFeed-dialog.html*:

```
<div id="palm-dialog-content" class="palm-dialog-content">
    <div id="add-feed-title" class="palm-dialog-title">
        Add Feed
    </div>
    <div class="palm-dialog-separator"></div>
    <div class="textfield-group" x-mojo-focus-highlight="true">
        <div class="title">
           <div x-mojo-element="TextField" id="newFeedURL"></div>
        </div>
    </div>
    <div class="textfield-group" x-mojo-focus-highlight="true">
        <div class="title">
           <div x-mojo-element="TextField" id="newFeedName"></div>
        </div>
    </div>

    <div class="palm-dialog-buttons">
        <div x-mojo-element="Button" id="okButton"></div>
        <div x-mojo-element="Button" id="cancelButton"></div>
    </div>
</div>
```

The changes from the HTML used previously include the removal of the Drawer and the addition of the various `palm-dialog` styles and replacing the `palm-list` styling with the `textfield-group`.

The dialog assistant should be defined like a scene assistant with a creator function and the standard scene methods: setup, activate, deactivate, and cleanup.

Within a dialog assistant, you can set up widgets, push scenes, and generally do anything that you can do within a scene assistant. There is one major difference: the dialog assistant's controller is a widget controller so it doesn't have direct access to scene controller methods; instead the dialog assistant must use the calling scene assistant's scene controller methods such as `setupWidget`. To facilitate this, the `assistant` property

in the `showDialog` argument object passes the keyword `this` as an argument when calling the dialog's creator function.

To create the `AddDialogAssistant`, we'll move the code we used in the last chapter to generate the small form in the Drawer widget. Here that code is presented with some modifications in the `AddDialogAssistant`:

```
// ---------------------------------------------------------------------------
// AddDialogAssistant - simple controller for adding new feeds to the list
//   when the "Add..." is selected on the feedlist. The dialog will
//   allow the user to enter the feed's url and optionally a name. When
//   the "Ok" button is tapped, the new feed will be loaded. If no errors
//   are encountered, the dialog will close otherwise the error will be
//   posted and the user encouraged to try again.
//
function AddDialogAssistant(sceneAssistant, feeds) {
    this.feeds = feeds;
    this.sceneAssistant = sceneAssistant;

    this.title = "";
    this.url = "";
    this.feedIndex = null;
    this.dialogTitle = "Add News Feed Source";
}

AddDialogAssistant.prototype.setup = function(widget) {
        this.widget = widget;

        // Set the dialog title to either Edit or Add Feed
// ** These next two lines are wrapped for book formatting only **
        var addFeedTitleElement =
          this.sceneAssistant.controller.get("add-feed-title");
        addFeedTitleElement.innerHTML = this.dialogTitle;

        // Setup text field for the new feed's URL
        this.sceneAssistant.controller.setupWidget(
            "newFeedURL",
            {
                  hintText: "RSS or ATOM feed URL",
                  autoFocus: true,
                  autoReplace: false,
                  textCase: Mojo.Widget.steModeLowerCase,
                  enterSubmits: false
            },
            this.urlModel = {value : this.url});

        // Setup text field for the new feed's name
        this.sceneAssistant.controller.setupWidget(
            "newFeedName",
            {
                    hintText: "Title (Optional)",
                    autoReplace: false,
                    textCase: Mojo.Widget.steModeTitleCase,
                    enterSubmits: false
            },
```

```
            this.nameModel = {value : this.title});

        // Setup OK  & Cancel buttons
        //   OK button is an activity button which will be active
        //   while processing and adding feed. Cancel will just
        //   close the scene
        this.okButtonModel = {label: "OK", disabled: false};
        this.sceneAssistant.controller.setupWidget("okButton",
          {type: Mojo.Widget.activityButton},
            this.okButtonModel);
        this.okButtonActive = false;
        this.okButton = this.sceneAssistant.controller.get("okButton");
        this.checkFeedHandler = this.checkFeed.bindAsEventListener(this);
        this.sceneAssistant.controller.listen("okButton", Mojo.Event.tap,
          this.checkFeedHandler);

        this.cancelButtonModel = {label: "Cancel", disabled: false};
        this.sceneAssistant.controller.setupWidget("cancelButton",
          {type: Mojo.Widget.defaultButton},
          this.cancelButtonModel);
        this.sceneAssistant.controller.listen("cancelButton", Mojo.Event.tap,
          this.widget.mojo.close);
};

// checkFeed  - called when OK button is clicked
AddDialogAssistant.prototype.checkFeed = function() {

        if (this.okButtonActive === true)  {
            // Shouldn't happen, but log event if it does and exit
            Mojo.Log.info("Multiple Check Feed requests");
            return;
        }

        // Check entered URL and name to confirm that it is a valid feedlist
        Mojo.Log.info("New Feed URL Request: ", this.urlModel.value);

        // Check for "http://" on front or other legal prefix; any string of
        // 1 to 5 alpha characters followed by ":" is ok, else
             prepend "http://"
        var    url = this.urlModel.value;
        if (/^[a-z]{1,5}:/.test(url) === false)    {
            // Strip any leading slashes
            url = url.replace(/^\/{1,2}/,"");
            url = "http://"+url;
        }

        // Update the entered URL & model
        this.urlModel.value = url;
        this.sceneAssistant.controller.modelChanged(this.urlModel);

        this.okButton.mojo.activate();
        this.okButtonActive = true;
        this.okButtonModel.label = "Updating Feed";
        this.okButtonModel.disabled = true;
        this.sceneAssistant.controller.modelChanged(this.okButtonModel);
```

```
        var request = new Ajax.Request(url, {
            method: "get",
            evalJSON: "false",
            onSuccess: this.checkSuccess.bind(this),
            onFailure: this.checkFailure.bind(this)
        });

};

// checkSuccess - Ajax request failure
AddDialogAssistant.prototype.checkSuccess = function(transport) {
        Mojo.Log.info("Valid URL - HTTP Status", transport.status);

        // DEBUG - Work around due occasion Ajax XML error in response.
        if (transport.responseXML === null && transport.responseText !== null) {
                Mojo.Log.info("Request not in XML format - manually converting");
// ** These next two lines are wrapped for book formatting only **
                transport.responseXML = new    DOMParser().
                  parseFromString(transport.responseText, "text/xml");
         }

        var feedError = News.errorNone;

        //  If a new feed, push the entered feed data on to the feedlist and
        //  call processFeed to evaluate it.
        if (this.feedIndex === null) {
            this.feeds.list.push({title:this.nameModel.value,
                url:this.urlModel.value, type:"", value:false, numUnRead:0,
                stories:[]});
            // processFeed - index defaults to last entry
            feedError = this.feeds.processFeed(transport);
        }
        else    {
            this.feeds.list[this.feedIndex] = {title:this.nameModel.value,
                url:this.urlModel.value, type:"", value:false, numUnRead:0,
                stories:[]};
            feedError = this.feeds.processFeed(transport, this.feedIndex);
        }

        // If successful processFeed returns errorNone
        if (feedError === News.errorNone)    {
            // update the widget, save the DB and exit
            this.sceneAssistant.feedWgtModel.items = this.feeds.list;
// ** These next two lines are wrapped for book formatting only **
            this.sceneAssistant.controller.modelChanged(
              this.sceneAssistant.feedWgtModel);
            this.widget.mojo.close();
        }
        else    {
            // Feed can't be processed - remove it but keep the dialog open
            this.feeds.list.pop();
            if (feedError == News.invalidFeedError)    {
                Mojo.Log.warn("Feed ",
                    this.urlModel.value, " isn't a supported feed type.");
```

```
                var addFeedTitleElement = this.controller.get("add-feed-title");
                addFeedTitleElement.innerHTML = "Invalid Feed Type - Please Retry";
            }

            this.okButton.mojo.deactivate();
            this.okButtonActive = false;
            this.okButtonModel.label = "OK";
            this.okButtonModel.disabled = false;
            this.sceneAssistant.controller.modelChanged(this.okButtonModel);
        }
};

// checkFailure  - Ajax request failure
AddDialogAssistant.prototype.checkFailure = function(transport) {
        // Log error and put message in status area
        Mojo.Log.info("Invalid URL - HTTP Status", transport.status);
        var addFeedTitleElement = this.controller.get("add-feed-title");
        addFeedTitleElement.innerHTML = "Invalid Feed Type - Please Retry";
};

// cleanup  - close Dialog
AddDialogAssistant.prototype.cleanup = function() {
        // TODO - Cancel Ajax request or Feed operation if in progress
        this.sceneAssistant.controller.stopListening("okButton",
          Mojo.Event.tap, this.checkFeedHandler);
        this.sceneAssistant.controller.stopListening("cancelButton",
          Mojo.Event.tap, this.widget.mojo.close);
};
```

There were several changes made to the previous version with the Drawer widget to create this version in a dialog:

Scene Assistant Methods
: Change `this.controller.*` references to `this.sceneAssistant.controller.*` references, because the `AddDialogAssistant` must use the passed reference to the scene assistant for any scene controller methods.

Close
: Add `this.widget.mojo.close()` after successfully adding the feed in `checkOk`. You will have to directly close the dialog by calling the `close()` method on the dialog widget. Notice that the widget element is passed as an argument to the dialog assistant's setup method.

TextField Cleanup
: Remove the code that explicitly cleared the text fields on exit; it isn't needed, as the dialog scene is removed from the DOM entirely.

Swiping back in a default dialog box will close the dialog box, but a Cancel button is recommended for most dialog boxes to help novice users who may be confused by its absence. You can set the optional `preventCancel` to `true` in the `showDialog` call arguments to stop the back gesture from canceling the dialog box; by default,

`preventCancel` is set to false. Figure 4-3 shows the results of these changes and the Add Feed dialog.

Figure 4-3. An Add Feed dialog box

Menus

Mojo supports four types of menu widgets. Each is fairly unique, but they share some common design elements and can be used in similar ways. You should review the User Interface Guidelines to see how best to apply each menu type and for general information on designing menus for your application.

Application menu
: A conventional desktop-style menu that drops down from the top-left corner of the screen when the user taps in that area.

View menu
: Menus used across the top of the screen. They can be used as display headers or action buttons, to pop up submenus, or to toggle settings.

Command menu
: Used to set menus or (more typically) buttons across the bottom of the screen for actions, to pop up submenus, or to toggle settings.

Submenu
: Can be used in conjunction with the other menu types to provide more options, or can be attached to any element in the page.

Application, View, and Command menus are technically very similar: they use a single model definition with a menu items array, and are configured through `setupWidget()`. Menu selections generate commands, which are propagated to registered *commanders* through the Commander Chain. We'll cover these three widgets in the next section on Menu widgets.

The Submenu shares many of the model properties with Menu widgets, but is instantiated through a direct function call and is handled differently. The Submenu widget will be addressed fully in its own section later in the chapter.

The System UI includes another menu, called the *Connection menu*, which is similar to the Application menu in appearance and is anchored to the top-right of the screen. It is restricted for system use and is not available to applications.

Menu Widgets

Unlike all other widgets, Menu widgets are not declared in your scene view file, but are simply instantiated and handled from within your assistant. From a design perspective, Menu widgets float above other scene elements, attached to the scene's window rather than a point in the scene. Because of this, it wouldn't work for their positions to be determined within the HTML. They are in the DOM, so you can use CSS to style them, but the framework determines their positions according to predefined constraints and the individual menu's attributes and model properties.

A Menu widget is instantiated by a call to `setupWidget()`, specifying the menu type, attributes, and model. The menu types take the form `Mojo.Menu.type`, where type can be one of `appMenu`, `viewMenu`, or `commandMenu`.

Menus have just a few attribute properties that differ between the Application menu and the Command/View menus; they'll be described in the following sections. The model is primarily made up of the `items` array, which includes an object for each menu item and optional properties. Other than the items array there is simply a `visible` property to set the entire menu to invisible (`false`) or visible (`true`). If not present, the menu defaults to visible.

The major options are in the `items` array. You can include selectable items and groups at the top level of any menu, where groups allow you to specify a second level of selectable items. Items can have a `label` and an icon. Icons can specify either an application-supplied icon image (found at `iconPath`) or one of the framework's icons (using the `icon` property).

Each item includes a command value, which is propagated through the Commander Chain when the item is selected. This is a rather significant topic, which we'll touch on briefly here, but you should review the section "Commander Chain" on page 126 to get a full description.

Application menu

The Application menu appears in the upper-left corner of the screen when the user taps the left side of the status bar. It includes some system-defined and some application-defined actions, and is intended to have an application-wide scope for the most part. Figure 4-4 shows an example of an Application menu.

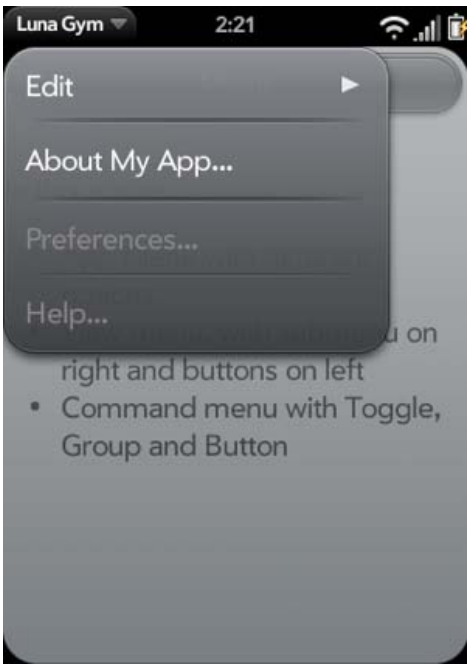


Figure 4-4. An Application menu

The Application menu contains a few required items: Edit (an item group including Cut, Copy, and Paste), Preferences, and Help; the latter items are disabled by default. You are free to add any other items to the menu, and to enable Preferences and/or Help by including command handlers to take the appropriate actions within your application.

Back to the News: Adding an Application menu

Now let's add an Application menu to News, with an "About News" item. Unlike our earlier example, we'll declare the Application menu attributes and model as global

variables, and add the `handleCommand()` method to the News stage assistant. This makes the Application menu available to all of the News scene assistants:

```
// Setup App Menu for all scenes; all menu actions handled in
//  StageAssistant.handleCommand()
News.MenuAttr = {omitDefaultItems: true};

News.MenuModel = {
    visible: true,
    items: [
        {label: "About News...", command: "do-aboutNews"},
        Mojo.Menu.editItem,
        Mojo.Menu.prefsItem,
        Mojo.Menu.helpItem
    ]
};
// ---------------------------------------
// handleCommand - called to handle app menu selections
//
StageAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {

            case "do-aboutNews":
                    var currentScene = this.controller.activeScene();
                    currentScene.showAlertDialog({
                      onChoose: function(value) {},
// ** These next two lines are wrapped for book formatting only **
                        title: "News – v#{version}".interpolate({
                          version: News.versionString}),
                        message: "Copyright 2009, Palm Inc.",
                        choices:[
                            {label:"OK", value:""}
                                ]
                      });
            break;
        }
    }
};
```

The following menu properties are unique to the Application menu:

`richTextEditItems`
: Can be set to true when you include a Rich Text Edit widget in your scene, and it will add the bold, italic, and underline styling items to the Edit menu.

`omitDefaultItems`
: Must be set to true when you want to remove or reorder the default items: Edit, Preferences, or Help.

By choosing `omitDefaultItems`, we must manually add back any of the default items in the model definition if they are to be displayed in the menu. If you want to change some but not all items, you can use system constants to replace the items that aren't changing. In the `News.MenuModel`, the default items `Mojo.Menu.editItem`, `Mojo.Menu.prefsItem`, and

`Mojo.Menu.helpItem` are all added back into the model to allow us to change the order of the items.

The `News.MenuAttr` declares that this menu will override the default items. The `News.MenuModel` puts the About News item at the top of the menu and by referencing the default items keeps them in the menu with the framework still handling them. Within the `handleCommand` method, the `do-aboutNews` command handler puts up an Alert Dialog with the About News information.

When the Application menu commands are propagated, they are handled by the stage-assistant, but the handlers need to be aware of the current scene. The local variable `currentScene` is set to the active scene controller at the beginning of `handleCommand`. From there, `currentScene` applies scene assistant functions, such as `showAlertDialog`, to whichever scene is currently displayed.

All this work has been done in *stage-assistant.js*, but the Application menu is actually displayed within the scenes. To configure and display the Menu widget, each scene assistant's setup method will include this `setupWidget()` call:

```
//  Setup Application Menu
this.controller.setupWidget(Mojo.Menu.appMenu, News.MenuAttr, News.MenuModel);
```

You can override the application-wide behavior for a specific scene by defining scene-specific application menu attributes or model before setting up the Application menu, and including a `handleCommand` method in that scene to handle the Application menu commands there. Don't forget to call `Mojo.Event.stopPropagation()` if you use any of the same commands used in your global Application menu. Figure 4-5 shows the Application menu and the resulting About box.

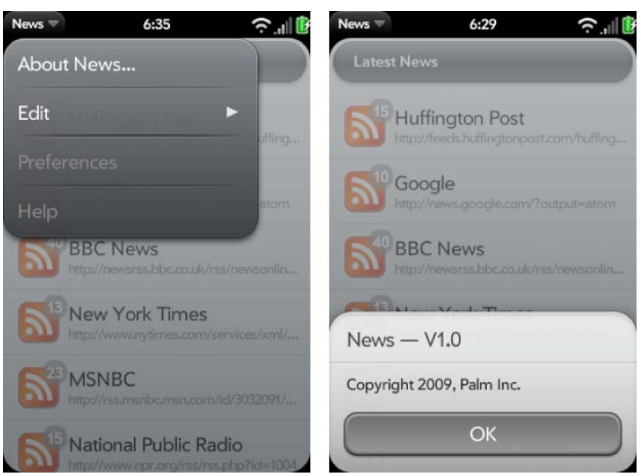


Figure 4-5. The News Application menu and About box

Be sure that you do not call `Mojo.Event.stopPropagation()` or `event.stop()` on events you do not handle. This is a common pitfall: stopping all events in `handleCommand`. This breaks a parts of the system UI such as back gestures and character picker.

By consolidating the Application menu declaration and handler in the stage assistant, it's easy to provide a common set of menu options across all of the scenes. As an example, let's add a Preferences scene to News.

By default, the Application Menu disables Preferences and Help. If you simply want to enable one or both commands, you can include a handler for `Mojo.Event.commandEnable` in the `handleCommand` method and call `Mojo.Event.stopPropagation()` for the command that you want to enable.

Back to the News: Adding preferences to News

In the previous chapter, we implemented the Ajax requests in *feedlist-assistant.js*, which retrieves the initial feed data. Let's extend that feature to periodically update the feeds, and we'll set the interval, the period between feed updates, in a preferences scene.

Create a Preferences scene using `palm-generate`:

```
palm-generate -t new_scene -p "name=preferences" com.palm.app.news
```

The scene's view file, *preferences-scene.html*, would look like this:

```
<div class="palm-page-header">
  <div class="palm-page-header-wrapper">
    <div class="icon news-mini-icon"></div>
    <div class="title">News Preferences</div>
  </div>
</div>

<div class="palm-group">
  <div class="palm-group-title"><span>Feed Updates</span></div>
  <div class="palm-list">
      <div class="palm-row">
         <div class="palm-row-wrapper">
            <div x-mojo-element="ListSelector" id="feedCheckIntervalList">
            </div>
         </div>
      </div>
    </div>
  </div>
</div>
```

The header is one of the framework style classes, `palm-page-header`, used on most preferences scenes. You'll also note the `icon` and the news-mini-icon styles, which allow us to add some CSS to insert a small news icon in the header. The icon must be added to the *images* directory at the News application's root level:

```
/* Header Styles  */
.icon.news-mini-icon {
    background: url(../images/header-icon-news.png) no-repeat;
    margin-top: 13px;
    margin-left: 17px;
}
```

After the header styling, you can see some list style classes followed by a List Selector widget declaration to pick the interval setting. The *preferences-assistant.js* will set up the List Selector and add a listener for selections using that List Selector. The handler, `feedIntervalHandler`, updates the global variable, `feedUpdateInterval`, after a selection is made:

```
/*  Preferences - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.
    Preferences - Handles preferences scene, where the user can:
        - select the interval for feed updates

    App Menu is disabled in this scene.

*/

function PreferencesAssistant() {

}

PreferencesAssistant.prototype.setup = function() {

    // Setup list selector for UPDATE INTERVAL
    this.controller.setupWidget("feedCheckIntervalList",
        {
            label: "Interval",
            choices: [
                {label: "Manual Updates",   value: 0},
                {label: "5 Minutes",        value: 300000},
                {label: "15 Minutes",       value: 900000},
                {label: "1 Hour",           value: 3600000},
                {label: "4 Hours",          value: 14400000},
                {label: "1 Day",            value: 86400000}
            ]
        },
        this.feedIntervalModel = {
            value : News.feedUpdateInterval
        });
```

```
    this.changeFeedIntervalHandler = this.changeFeedInterval.bindAsEventListener(this);
    this.controller.listen("feedCheckIntervalList",
      Mojo.Event.propertyChange, this.changeFeedIntervalHandler);

};

// Cleanup - remove listeners
PreferencesAssistant.prototype.cleanup = function() {
    this.controller.stopListening("feedCheckIntervalList",
        Mojo.Event.propertyChange, this.changeFeedIntervalHandler);
};

//    changeFeedInterval - Handle changes to the feed update interval
PreferencesAssistant.prototype.changeFeedInterval = function(event) {
    Mojo.Log.info("Preferences Feed Interval Handler; value = ",
        this.feedIntervalModel.value);
    News.feedUpdateInterval = this.feedIntervalModel.value;
};
```

The `feedUpdateInterval` is used by the stage assistant's `setWakup()` method to set the timer for the updates.

With the Preferences scene coded, we can hook it up by returning to the stage assistant and changing the `News.MenuModel` to override the default Preferences command:

```
News.MenuModel = {
    visible: true,
    items: [
        {label: "About News...", command: "do-aboutNews"},
        Mojo.Menu.editItem,
        {label: "Preferences...", command: "do-newsPrefs"},
        Mojo.Menu.helpItem
    ]
};
```

Next, we'll add a handler for `do-newsPrefs` in the `handleCommand` method to push the Preferences scene:

```
case "do-newsPrefs":
    this.controller.pushScene("preferences");
break;
```

When you run the application now, you'll see that the Preferences item is enabled and when selected brings up the new scene. Figure 4-6 shows the Application menu and the resulting Preferences scene.

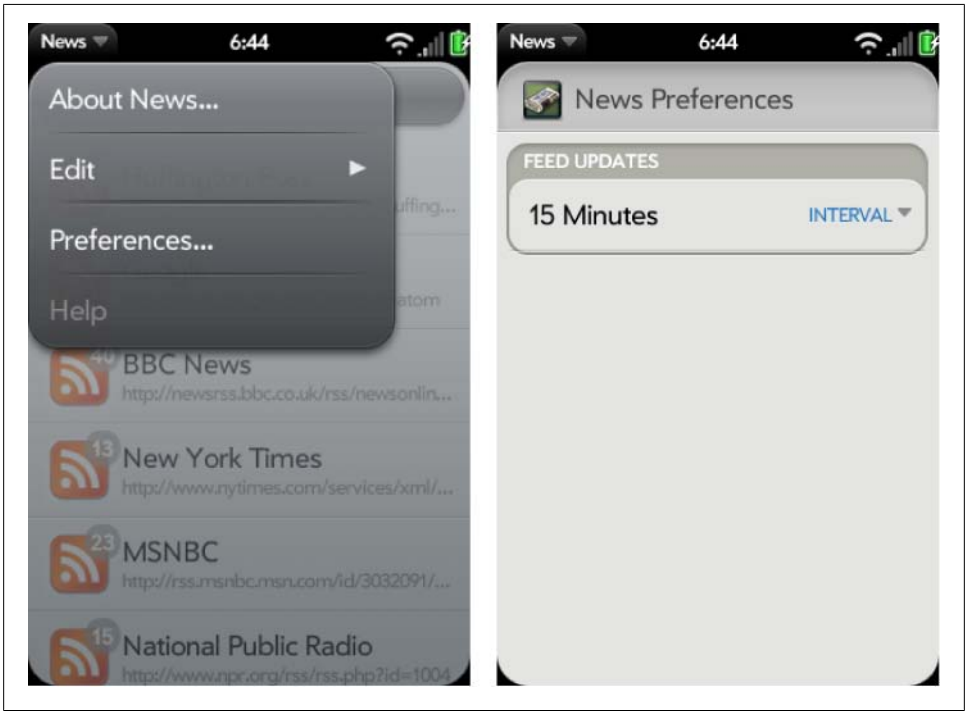


Figure 4-6. The News Application menu with a Preferences scene

You should also notice that we didn't have to modify any of the scene assistants, yet the Preferences option is available in every scene. This approach makes it simple to consolidate common Application menu handling throughout your application.

Our final Application menu example will demonstrate the command enable feature of the Commander Chain. We'll add a manual feed update feature to the Application menu by adding a new item to the `News.MenuModel` called "Update All Feeds":

```
News.MenuModel = {
    visible: true,
    items: [
        {label: "About News...", command: "do-aboutNews"},
        Mojo.Menu.editItem,
        {label: "Update All Feeds", checkEnabled: true, command: "do-feedUpdate"},
        {label: "Preferences...", command: "do-newsPrefs"},
        Mojo.Menu.helpItem
    ]
};
```

Because this command should be disabled whenever a feed update is in progress, a new property, `checkEnabled`, is set to `true`. This property will instruct the framework to propagate a `Mojo.Event.commandEnable` event through the commander chain anytime the menu is displayed. If any recipient calls `event.preventDefault()` in response, then the menu item is disabled.

Here's how this is handled in the stage assistant's `handleCommand()` method:

```
// ----------------------------------------
// handleCommand - called to handle app menu selections
//
StageAssistant.prototype.handleCommand = function(event) {
    if (event.type == Mojo.Event.commandEnable) {
        if (News.feedListUpdateInProgress && (event.command == "do-feedUpdate")) {
            event.preventDefault();
        }
    }

    else {

        if(event.type == Mojo.Event.command) {
            switch(event.command) {

                case "do-aboutNews":
                    var currentScene = this.controller.activeScene();
                    currentScene.showAlertDialog({
                            onChoose: function(value) {},
// ** These next two lines are wrapped for book formatting only **
                            title: "News – v#{version}".interpolate({
                              version: News.versionString}),
                            message: "Copyright 2009, Palm Inc.",
                            choices:[
                                {label:"OK", value:""}
                                        ]
                              });
                break;

                case "do-newsPrefs":
                    this.controller.pushScene("preferences");
                break;

                case "do-feedUpdate":
                    this.feeds.updateFeedList();
                break;
            }
        }
    }
};
```

At the top, we check for the `Mojo.Event.commandEnable` event, and if it is tied to a `do-feedUpdate` command and an update is in progress, we'll inhibit the menu item by calling `event.preventDefault()`. You can learn more about this in the section "Commander Chain" on page 126.

View menu

The View menu presents items as variable-sized buttons, either singly or in groups across the top of the scene. The items are rendered in a horizontal sequence starting from the left of the screen to the right. The button widths can be adjusted using the items `width` property, and the framework adjusts the space between the buttons

automatically. Use dividers or empty list items to influence the spacing to get specific layouts.

Typically, you would use the View menu for actionable buttons, buttons with an attached submenu, or header displays. You can group buttons together or combine actionable buttons with header information, as in the example shown in Figure 4-7.

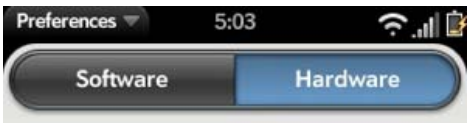


Figure 4-7. A View menu with buttons

Back to the News: Adding View menus

View menus allow us to style the `storyList` scene headers and to provide a simple way to switch between story feeds. We're going to change *storyList-assistant.js* to include a View menu with both Next Feed and Previous Feed menu buttons, and methods to push the new scene for either the next feed or previous feed when selected.

First, let's add the View menu. In this next code sample, the `this.feedMenuModel` is set up with three menu items that are all based on the feed that is displayed in this instance of the `storyList` scene:

- `FeedMenuPrev`, a local variable set to either the Previous menu item or an empty item if the feed is the first feed in the feedlist, meaning that the `selectedFeedIndex` is zero
- `FeedMenuNext`, a local variable set to either the Next menu item or an empty item if the feed is the last feed in the feedlist, meaning that the `selectedFeedIndex` is one less than the length of the feedlist
- A literal that displays the feed's title

The setup method starts with some conditional assignments to `feedMenuPrev` and `feedMenuNext` to deal with the boundary cases of the first and last feeds, then the View Menu widget is setup in a `setupWidget()` call.

Items that do not specify any visible attributes (such as `label` and `icon`), and are not groups, are treated as *dividers*. During layout of the menu buttons, all extra space is equally distributed to each of the dividers. If there are no dividers, any extra space is placed between the menu items, with the first and last menu items always aligned to the left and right of the scene. The boundary cases of the first feed and last feed will create dividers in `feedMenuPrev` and `feedMenuNext` to maintain the header's visual style and format.

Menu Icons

The Mojo framework includes a number of default icons that you can use on your View and Command menu buttons, or you can provide your own. Look at the standard styles prefixed with `palm-menu-icon`, which you can reference with the `icon` property, or define your own in your images folder, referencing them with the `iconPath` property. The News example uses one of each reference for illustration so you can see the two techniques.

Here are some guidelines for designing your own icons:

- Use PNG-24, which supports 8-bit alpha transparency.
- Menu icons are currently two frames in a 32 × 64 PNG, with the top frame as the normal state and the bottom frame as the pressed state.
- Each icon is approximately 24 × 24 pixels within the 32 × 32 frame.
- You can start with a monochrome glyph and style it with some Photoshop layer effects, although plain white would work.

You can look at the PNG files in the framework's *images* directory to see some examples of these icons.

```
/*  StoryListAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Displays the feed's stories in a list, user taps display the
    selected story in the storyView scene. Major components:
    - Setup view menu to move to next or previous feed
    - Story View; push story scene when a story is tapped

    Arguments:
    - feedlist; Feeds.list array of all feeds
    - selectedFeedIndex; Feed to be displayed
*/

function StoryListAssistant(feedlist, selectedFeedIndex) {
    this.feedlist = feedlist;
    this.feed = feedlist[selectedFeedIndex];
    this.feedIndex = selectedFeedIndex;
    Mojo.Log.info("StoryList entry = ", this.feedIndex);
    Mojo.Log.info("StoryList feed = " + Object.toJSON(this.feed));
}


StoryListAssistant.prototype.setup =  function() {
    // Setup scene header with feed title and next/previous feed buttons. If
    // this is the first feed, suppress Previous menu; if last, suppress Next menu
    var feedMenuPrev = {};
    var feedMenuNext = {};

    if (this.feedIndex > 0)    {
        feedMenuPrev = {
            icon: "back",
```

```
            command: "do-feedPrevious"
        };
    } else    {
        // Push empty menu to force menu bar to draw on left (label is the force)
        feedMenuPrev = {icon: "", command: "", label: "  "};
    }

    if (this.feedIndex < this.feedlist.length-1)    {
        feedMenuNext = {
            iconPath: "images/menu-icon-forward.png",
            command: "do-feedNext"
        };
    } else    {
        // Push empty menu to force menu bar to draw on right (label is the force)
        feedMenuNext = {icon: "", command: "", label: "  "};
    }

    this.feedMenuModel =     {
        visible: true,
        items:     [{
            items: [
                feedMenuPrev,
                { label: this.feed.title, width: 200 },
                feedMenuNext
            ]
        }]
    };

    this.controller.setupWidget(Mojo.Menu.viewMenu,
        { spacerHeight: 0, menuClass:"no-fade" }, this.feedMenuModel);

    // Setup App Menu
    this.controller.setupWidget(Mojo.Menu.appMenu, News.MenuAttr, News.MenuModel);

    // Setup story list with standard news list templates.
    this.controller.setupWidget("storyListWgt",
        {
            itemTemplate: "storyList/storyRowTemplate",
            listTemplate: "storyList/storyListTemplate",
            swipeToDelete: false,
            renderLimit: 40,
            reorderable: false
        },
        this.storyModel = {
            items: this.feed.stories
        }
    );

    this.readStoryHandler = this.readStory.bindAsEventListener(this);
    this.controller.listen("storyListWgt", Mojo.Event.listTap,
        this.readStoryHandler);
};
```

Continuing on with our example, we'll add the `handleCommand()` method after the `activate()` and `readStory()` methods, which are unchanged:

```
// handleCommand - handle next and previous commands
StoryListAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case "do-feedNext":
                this.nextFeed();
                break;
            case "do-feedPrevious":
                this.previousFeed();
                break;
        }
    }
};

// nextFeed - Called when the user taps the next menu item
StoryListAssistant.prototype.nextFeed = function(event) {
    Mojo.Controller.stageController.swapScene(
        {
            transition: Mojo.Transition.crossFade,
            name: "storyList"
        },
        this.feedlist,
        this.feedIndex+1);
};

// previousFeed - Called when the user taps the previous menu item
StoryListAssistant.prototype.previousFeed = function(event) {
    Mojo.Controller.stageController.swapScene(
        {
            transition: Mojo.Transition.crossFade,
            name: "storyList"
        },
        this.feedlist,
        this.feedIndex-1);
};
```

The `handleCommand` method is called for the Next and Previous commands and results in a `swapScene()` call to push the next scene. We discussed in Chapter 2 that `swapScene()` is similar to `pushScene()`, but rather than leaving the old scene on the scene stack, `swapScene()` pops it as part of the operation.

Figure 4-8 shows the News application's `storyList` with these changes.

Command menu

The Command menu items are presented at the bottom of the screen, but are similar to the View menu in most other ways. Items will include variable-sized buttons that you can combine into groups and in a horizontal layout from left to right. You can override the default positioning by including dividers to force an item to the right or the middle of the screen, or by including an item's entry with the `disable` property set to true. Typically, you would use the Command menu for actionable buttons, buttons with dynamic behavior, or for attaching a submenu to a button to give further options.

Figure 4-8. The News View menu and storyList scene

As with the View menu, you can adjust button widths from within the item's property width, and the framework adjusts the space between the buttons automatically (as shown in Figure 4-9).

Figure 4-9. A Command menu with buttons

You can also define toggle buttons or include buttons with other dynamic behavior (Figure 4-10).

Figure 4-10. A Command menu with toggles

If you'd like to group several items together, include an `items` array and the `toggleCmd`, which will be set by the framework to the command of the currently selected items in the nested items array. You can group buttons together or combine actionable buttons into a toggle group, as shown in Figure 4-11.

Figure 4-11. A Command menu with groups

Back to the News: Adding Command menus

We've been using buttons within the Story view to go back and forth between stories within a feed, but here we'll replace those buttons with Command menus. It's really straightforward now that we've covered the basics with the Application and View menus.

Change the *storyView-assistant.js* to include a command menu. Similar to the View menu, the Next and Previous buttons normally are assigned to generate `do-viewNext` or `do-viewPrevious` commands, except when the current story is the first or last story in the feed. The first part of the setup method will create the items array with the correct entries, then call `setupWidget()` to instantiate the menu. Since we're replacing the buttons that were in the scene, remove the listeners from the setup method (and the button declarations from the scene's view file).

Notice that the items are put into a menu group so that they are styled together. We use dividers on either side of the group to force the group to be centered in the scene. Notice the visual difference from the Application menu's grouping, where subitems are combined into an expanding item. With View and Command menus, the button groups are presented as an integrated view element:

```
/*  StoryViewAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.
```

```
    Passed a story element, displays that element in a full scene view and offers
    options for next story (right command menu button), and previous
    story (left command menu button)
    Major components:
    - StoryView; display story in main scene
    - Next/Previous; command menu options to go to next or previous story

    Arguments:
    - storyFeed; Selected feed from which the stories are being viewed
    - storyIndex; Index of selected story to be put into the view
*/

function StoryViewAssistant(storyFeed, storyIndex) {
    this.storyFeed = storyFeed;
    this.storyIndex = storyIndex;
}

// setup - set up menus
StoryViewAssistant.prototype.setup = function() {
       this.storyMenuModel = {
         items: [
            {},
            {items: []},
            {}
         ]};

        if (this.storyIndex > 0)    {
           this.storyMenuModel.items[1].items.push({
               icon: "back",
               command: "do-viewPrevious"
            });
        } else {
            this.storyMenuModel.items[1].items.push({
                icon: "", command: "",
                label: "  "
            });
        }

        if (this.storyIndex < this.storyFeed.stories.length-1)    {
            this.storyMenuModel.items[1].items.push({
                icon: "forward",
                command: "do-viewNext"
            });
        } else {
            this.storyMenuModel.items[1].items.push({
                icon: "", command: "",
                label: "  "
            });
        }
    this.controller.setupWidget(Mojo.Menu.commandMenu,
        undefined, this.storyMenuModel);

    // Setup App Menu
    this.controller.setupWidget(Mojo.Menu.appMenu, News.MenuAttr, News.MenuModel);
```

```
    // Update story title in header and summary
    var storyViewTitleElement = this.controller.get("storyViewTitle");
    var storyViewSummaryElement = this.controller.get("storyViewSummary");
    storyViewTitleElement.innerHTML = this.storyFeed.stories[this.storyIndex].title;
    storyViewSummaryElement.innerHTML = this.storyFeed.stories[this.storyIndex].text;

};
```

The activate method is unchanged, but we replace the button handlers with command handlers, as shown in this next code sample:

```
// -------------------------------------------------------------
// Handlers to go to next and previous stories, display web view
// or share via messaging or email.
StoryViewAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case "do-viewNext":
                Mojo.Controller.stageController.swapScene(
                    {
                        transition: Mojo.Transition.crossFade,
                        name: "storyView"
                    },
                    this.storyFeed, this.storyIndex+1);
                break;
            case "do-viewPrevious":
                Mojo.Controller.stageController.swapScene(
                    {
                        transition: Mojo.Transition.crossFade,
                        name: "storyView"
                    },
                    this.storyFeed, this.storyIndex-1);
                break;

        }
    }
};
```

That's it. Run the application and tap a feed and then a story to see the results (shown in Figure 4-12).

Submenus

Pop-up submenus can offer a transient textual list of choices to the user, typically off of another menu type or from a DOM element in the scene. Submenus accept standard menu models and some unique properties, but unlike the other menu types, Submenu does not use the Commander Chain for propagating selections. Instead, a callback is used to handle selections.

A modal list will appear with the label choices presented. When the user taps one, the `onChoose` callback function will be called (in the scope of the scene assistant) with the

Figure 4-12. A News Command menu and storyView scene

command property of the chosen item as an argument. If the user taps outside the pop-up menu, it's still dismissed and the `onChoose` function is called with `undefined` instead.

Back to the News: Adding a submenu

We will use a submenu to present options when the users taps the info button on the News feed list. For each feed, you can choose between Mark Read, Mark Unread, or Edit Feed. The first two options mark all the stories as either read or unread based on the selection, while the last option brings up the Add Feed dialog.

To bring up the submenu, we'll add an icon called `info`, to each list entry in the feed list, as shown in the `feedList` scene, to serve as an access point. The change is made in *feedRowTemplate.html* just before the entries for the `feedlist-title` and `feedlist-url`. The custom class will be used in *News.css* to load the icon's image while the framework classes will fix the position of the icon properly within the list row and align it to the right:

```
<div class="feedlist-info icon right" id="info"></div>
<div class="feedlist-title truncating-text">#{title}</div>
<div class="feedlist-url truncating-text">#{-url}</div>
```

Next, we'll modify the `showFeed()` method of *feedlist-assistant.js* to detect taps on the `info` icon. If it's a tap anywhere else, the `storyList` scene will be pushed as before.

Otherwise, the Submenu will be created, with an arguments list starting with `onChoose`, which specifies `popupHandler` to handle the user's menu selection. The other arguments include the `placeNear` property to locate the submenu near the tapped icon and the array of menu items. You'll notice that we save the `event.index` value by assigning it to `this.popupIndex` for use later in `popupHandler`. We'll need to reference the tapped list entry when it comes time to apply the action indicated by the Submenu selection:

```
// ---------------------------------------------------------------------
// Show feed and popup menu handler
//
// showFeed - triggered by tapping a feed in the this.feeds.list.
//   Detects taps on the unReadCount icon; anywhere else,
//   the scene for the list view is pushed. If the icon is tapped,
//   put up a submenu for the feedlist options
FeedListAssistant.prototype.showFeed = function(event) {
        var target = event.originalEvent.target.id;
        if (target !== "info") {
          Mojo.Controller.stageController.pushScene("storyList",
            this.feeds.list, event.index);
        }
        else  {
            var myEvent = event;
            var findPlace = myEvent.originalEvent.target;
            this.popupIndex = event.index;
            this.controller.popupSubmenu({
              onChoose:  this.popupHandler,
              placeNear: findPlace,
              items: [
                {label: "All Unread", command: "feed-unread"},
                {label: "All Read", command: "feed-read"},
                {label: "Edit Feed", command: "feed-edit"}
                ]
              });
        }
};
```

The handler, `popupHandler`, uses a switch statement to invoke the appropriate command handler. The command handlers for "All Unread" and "All Read" handle the actions for marking all stories in the selected feed as unread or read, updating the feed's `numUnRead`, and calling `modelChanged` to update the scene's displayed view. Once the actions are complete, the handler exits and the framework cleans up the display by removing the Submenu.

Targeting the submenu can sometimes be a little tricky. The framework will automatically place the submenu in the center of the window, but you can override it by setting `placeNear` to a specific DOM element. In our example, it's placed near the `info` icon, which was defined as the tap target for this submenu. It's a good idea to use fixed targets for menu placement:

```
// popupHandler - choose function for feedPopup
FeedListAssistant.prototype.popupHandler = function(command) {
        var popupFeed=this.feeds.list[this.popupIndex];
        switch(command) {
            case "feed-unread":
                Mojo.Log.info("Popup - unread for feed:",
                  popupFeed.title);

                for (var i=0; i<popupFeed.stories.length; i++ ) {
                    popupFeed.stories[i].unreadStyle = News.unreadStory;
                }
                popupFeed.numUnRead = popupFeed.stories.length;
                this.controller.modelChanged(this.feedWgtModel);
                break;

            case "feed-read":
                Mojo.Log.info("Popup - read for feed:",
                  popupFeed.title);
                for (var j=0; j<popupFeed.stories.length; j++ ) {
                    popupFeed.stories[j].unreadStyle = "";
                }
                popupFeed.numUnRead = 0;
                this.controller.modelChanged(this.feedWgtModel);
                break;

            case "feed-edit":
                Mojo.Log.info("Popup edit for feed:",
                  popupFeed.title);
                this.controller.showDialog({
                    template: "feedList/addFeed-dialog",
                    assistant: new AddDialogAssistant(this,
                        this.feeds, this.popupIndex)
                });
                break;

       }
};
```

For the "Edit Feed" choice, the handler uses the `AddDialogAssistant` to display the selected feed's URL and name so that they can be changed. A new argument, `this.popupIndex`, is added to the `AddDialogAssistant` call to enable the `AddDialogAssistant` and its methods, `checkIt` and `checkOk`, to look for an edit case. These changes are not shown here because they are not directly related to the Submenu, but you can look at the full News source in Appendix D to see where the changes were made.

Figure 4-13 shows a Submenu with these changes within the `feedList` scene.

Commander Chain

Mojo provides a model for propagating commands through the application, stage, and scene controllers called the Commander Chain. The chain is an array of handlers,

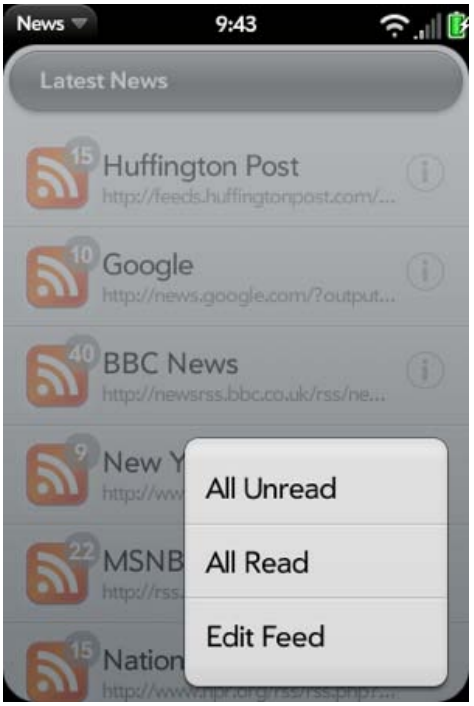


Figure 4-13. A News Submenu in a feedList scene

ordered like a stack. The handlers, or commanders, are put onto the chain in the order that they register themselves, and commands are propagated according to this order.

Commanders are registered implicitly by declaring a `handleCommand` method as a stage-assistant or scene-assistant method, or for dialogs, when instantiated. The framework always adds the App-Assistants to the end of the Stage-Controller chain at instantiation.

Commanders can register explicitly by calling the `pushCommander` method from either the stage controller or scene controller. The commander will be removed when the scene assistant is popped or the application is closed.

The chain is really a tree of chains (see Figure 4-14). There is a chain for each stage controller, and within each stage there is a chain for each scene controller. Commands are propagated starting with the most recent commander registered in the active scene controller's chain. After all commanders in the scene have been called, propagation continues with the most recent commander in the active stage controller chain through the rest of the chain. There are chains for each of the inactive stage controllers and scene controllers, but commands are not propagated to any inactive chains.

At any time, any commander can stop propagation by calling `event.stopPropagation()`. For example, a scene puts up a modal dialog box, so it's

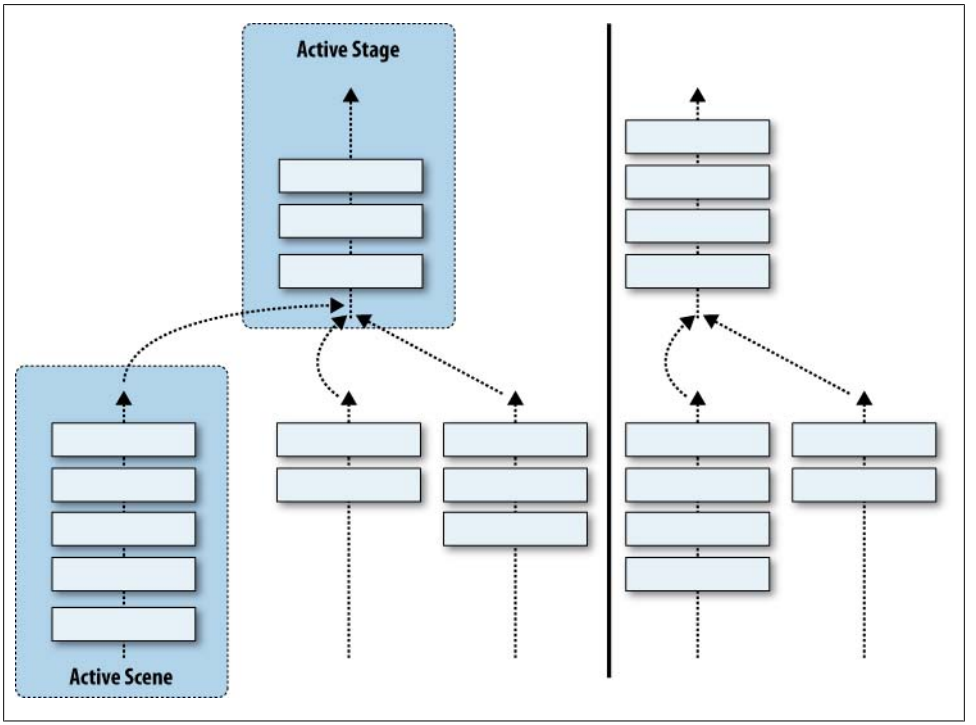


Figure 4-14. A Commander Chain propagation example

implicitly added to the chain. It will have the opportunity to handle a back event and stop propagation before it gets back to the scene that pushed the dialog. If not, the stage controller would see the back gesture and pop the scene, which is not the desired user experience.

Commanders can always remove themselves from the chain by calling the `removeCommander` method of either `StageController` or `SceneController`. For example:

```
this.controller.removeCommander(this);
```

There are four types of events that propagate through the chain:

`Mojo.Event.back`
: Indicates a back gesture.

`Mojo.Event.forward`
: Indicates a forward gesture.

`Mojo.Event.command`
: Is used for all menu commands.

`Mojo.Event.commandEnable`
: Is used to enable a menu item dynamically.

Command and Command Enable events are both discussed in the section "Menu Widgets" on page 107. The former is used when a menu command is selected, and the latter when a menu is created for any menu item that includes the property `commandEnable` set to true. If any commander wants to inhibit the menu command, it can call `event.preventDefault()` to do so. The framework uses this to inhibit the Edit functions in the Application menu when anything other than a text field is in focus.

A common application of the Commander Chain is the consolidation of the setup and handling of the Application menu into the stage controller. An example of this consolidation using the News application is shown in the section "Application menu" on page 108.

Summary

Dialogs and menus round out the basic widgets that most applications require, and with what you've learned so far, you should be able to write a meaningful application. In this chapter we covered the three dialog functions and the four types of menus, and used almost all of them in the News application as sample code.

The next chapter will cover the remainder of the widgets, but it would be good at this point to build some sample code using widgets and the UI model. With just what's been covered so far, you can build full-featured applications, but more importantly, the concepts learned here will be used throughout the rest of the book.

CHAPTER 5

Advanced Widgets

This chapter completes the review of the Mojo widgets with a look at indicators, pickers and viewers, the Filter List, and the Scroller. Not all applications will use these widgets, because they are each designed for specific uses, but the widgets are just as simple to work with as the widgets discussed in Chapters 3 and 4.

As with the two preceding chapters, each group will be reviewed in a summary form, then a specific case will be used as an example with the News application. Where a widget isn't used in an example, there will be a description and references to where you can find more information.

Indicators

Indicators are used to show that activity is taking place, even it it's not visible, and in some cases, to show some measure of the progress of the activity. Mojo has four indicator widgets, but they belong to two types:

- Activity indicator, or Spinner, which spins without showing progress
- Progress indicator, which shows both activity and progress

The spinner is the only activity indicator, but there are three progress indicator widgets:

- Progress Pill, a wide pill that is styled to match the View menu and the `palm-header` scene style
- Progress Bar, a narrow horizontal bar with a blue progress indicator
- Progress Slider, which is intended for streaming media playback applications

The Spinner widget is most appropriate when there isn't much space in the layout for an indicator or when the duration of the activity is hard to estimate. In other cases you should use a progress indicator; it's preferable because it gives the user a bounded sense of duration.

Spinners

Use a spinner to show that an activity is taking place. The framework uses a spinner as part of any activity button, and you'll see it used in the core applications. There are two sizes; the large spinner is 128 × 128 pixels, and the small spinner is 32 × 32 pixels. These sizes are optimized for the Palm Prē screen and may vary on other devices, but the spatial and visual characteristics will be maintained on other devices.

Back to the News: Adding a spinner for feed updates

There aren't any long operations in News other than the feed updates, which are asynchronous. We'll add a spinner to the feed list whenever an update is in progress, demonstrating a simple application of an indicator.

This will also demonstrate the technique for including widgets within a list entry, a powerful Mojo feature introduced in Chapter 3. You already know that you can use widgets to display dynamic data; by combining them into lists you can create complex UI controls with the widgets as building blocks. You may want to review Chapter 3 if you have questions after reading these next few paragraphs.

You can design list entries to include other widgets, including other lists, in almost the same way that you use widgets outside of lists. The differences are that the list's model includes the widgets' models, and that you declare widgets within the list's `itemTemplate`, using a name attribute to identify each widget.

In this example, a Spinner widget is included in each `feedListWgt` entry, which will be activated when the corresponding news feed is updated through an Ajax request. Start by adding the spinner declaration into the `feedListWgt`'s row template, *views/feedList/feedRowTemplate.html*:

```
<div class="feedlist-info icon right" id="info"></div>
<div x-mojo-element="Spinner" class="right" name="feedSpinner"</div>
<div class="feedlist-title truncating-text">#{title}</div>
<div class="feedlist-url truncating-text">#{-url}</div>
```

The new line is the div with the name `feedSpinner` and is simply a declaration of the Spinner widget. The syntax should start to seem familiar by now.

By including it into the `feedListWgt` list item's template, we have implicitly directed the List widget to insert a new spinner element in the DOM whenever it creates a new entry. It's the same as creating a spinner outside of a list, except in one major way: the spinner's model property must be part of the `feedListWgt`'s items array.

We still have to set up the Spinner widget, which we do in the setup method of *feedListassistant.js*, but we don't include a model in the call to `setupWidget()`, as that is assumed to be part of the `feedListWgt`'s items array:

```
    // Setup the feed list, but it's empty
this.controller.setupWidget("feedListWgt",
     {
        itemTemplate:"feedList/feedRowTemplate",
```

```
        listTemplate:"feedList/feedListTemplate",
        addItemLabel:"Add...",
        swipeToDelete:true,
        renderLimit: 40,
        reorderable:true
    },
    this.feedWgtModel = {items: this.feeds.list});

// Setup event handlers: list selection, add, delete and reorder feed entry
this.showFeedHandler = this.showFeed.bindAsEventListener(this);
this.controller.listen("feedListWgt", Mojo.Event.listTap,
    this.showFeedHandler);
this.addNewFeedHandler = this.addNewFeed.bindAsEventListener(this);
this.controller.listen("feedListWgt", Mojo.Event.listAdd,
    this.addNewFeedHandler);
this.listDeleteFeedHandler = this.listDeleteFeed.bindAsEventListener(this);
this.controller.listen("feedListWgt", Mojo.Event.listDelete,
    this.listDeleteFeedHandler);
this.listReorderFeedHandler = this.listReorderFeed.bindAsEventListener(this);
this.controller.listen("feedListWgt", Mojo.Event.listReorder,
    this.listReorderFeedHandler);

// Setup spinner for feedlist updates
this.controller.setupWidget("feedSpinner", {property: "value"});
```

Most of the `feedList` assistant's setup method is shown in this code sample; the only addition is the last line of code (and preceding comment). It sets up the spinner, naming the model property as `value`, but the model is not in the arguments list; the list's model, `this.feedWgtModel`, is used implicitly as the spinner's model.

The `feedList` default data is defined at the beginning of the *stage-assistant.js*. Add the value property to each default list entry, and set it to false. This is the spinner's model, and will start as false since there is no activity. You need to include this for every `feedList` entry, as in this example:

```
{
   title:"New York Times",
   url:"http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml",
   type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
},
```

There's one other place where a new feed list entry is created; in the `checkOk` method of *addDialog-assistant.js*:

```
//  If a new feed, push the entered feed data on to the feedlist and
//  call processFeed to evaluate it.
if (this.feedIndex === null) {
    this.feeds.list.push({title:this.nameModel.value,
        url:this.urlModel.value, type:"", value:false, numUnRead:0,
        stories:[]});
    // processFeed - index defaults to last entry
    feedError = this.feeds.processFeed(transport);
}
else    {
    this.feeds.list[this.feedIndex] = {title:this.nameModel.value,
```

```
            url:this.urlModel.value, type:"", value:false, numUnRead:0,
            stories:[]};
        feedError = this.feeds.processFeed(transport, this.feedIndex);
    }
```

The spinner is set up and integrated into the list's template and model. All that remains is to activate and deactivate the spinner at the right times. Those times are just before the Ajax request is made (spinner activated) and after the response is received (spinner deactivated) whether the request was successful or not.

There are four changes to make, all in the `feeds` model:

`updateFeedRequest`
: Activate before Ajax request

`updateFeedFailure`
: Deactivate

`updateFeedSuccess`
: Deactivate after processing new feed data, and activate before another feed update request is made

The sample below shows the changes to `updateFeedSuccess()`, which includes both an activate and a deactivate call:

```
// Process the feed, passing in transport holding the updated feed data
var feedError = this.processFeed(transport, this.feedIndex);

// If successful processFeed returns News.errorNone,
if (feedError !== News.errorNone)    {
    // There was a feed process error; unlikely, but could happen if the
    // feed was changed by the feed service. Log the error.
    if (feedError == News.invalidFeedError)    {
        Mojo.Log.info("Feed ", this.nameModel.value,
          " is not a supported feed type.");
    }
}

News.feedListChanged = true;
// Change feed update indicator & update widget
var spinnerModel = this.list[this.feedIndex];
spinnerModel.value = false;
this.updateListModel();

// If NOT the last feed then update the feedsource and request next feed
this.feedIndex++;
if(this.feedIndex < this.list.length) {
    this.currentFeed = this.list[this.feedIndex];

    // Request an update for the next feed but first
    // change the feed update indicator & update widget
    spinnerModel = this.list[this.feedIndex];
    spinnerModel.value = true;
    this.updateListModel();
```

```
        this.updateFeedRequest(this.currentFeed);
    } else {

        // Otherwise, this update is done. Reset index to 0 for next update
        this.feedIndex = 0;
        News.feedListUpdateInProgress = false;

    }
         .
         .
         .
```

In each case, we set the `spinnerModel.value` and call `this.updateListModel()` to update the model changing the state of the spinner. The spinner's model is accessed by referencing the `this.feedIndex`, the array index for the feed that is being updated, then setting that entry's value property to change just the spinner in that list entry.

Run the application, and the feeds will update one after another. If you wait for the feed interval to pass, they will update again. You will see a spinner appear between the feed title and the unread count badge on the left side, as shown in Figure 5-1, with the spinner on the BBC News feed item.

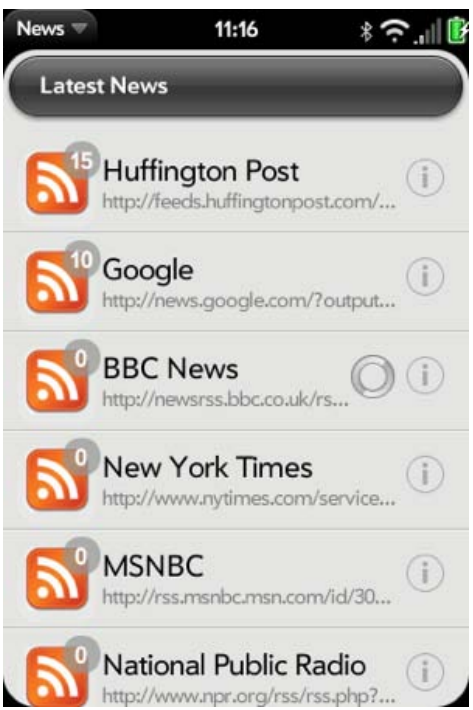


Figure 5-1. Spinner on News feed updates

Spinners only take up space when they're active. In some cases, if the feed title is long enough, you'll see the title truncated to accommodate the spinner, then resize to fill the vacated space after the spinner is deactivated. Elegant integration of indicators is the type of polish that makes an application appealing and easy to do with Mojo's widgets.

Progress Indicators

The progress pill is the most common progress indicator, and is styled to match the Mojo button and header styles. The other two indicators, progress bars and progress sliders, are more specialized, but are functionally derived from progress pills; you'll manage them in the same way.

Progress Pill

Use a Progress Pill widget (Figure 5-2) to show download progress when loading from a database, or anytime you initiate a long-running operation and have a sense of the duration.

Figure 5-2. A Progress Pill widget example

The indicator is designed to show a pill image that corresponds to the model's value property, where a value of 0 has no pill exposed and a value of 1 has the pill completely filling the container. Initialize the indicator's model value to 0, then progressively update the model property until it has a value of 1.

It's best to use an interval timer. At each interval callback, increase the progress indicator's value property and call the `updateModel` function. For example, start with the progress pill's value property set to 0 and set an interval timer for 600 ms. Assuming the planned operation will take about 3 seconds, you would increase the value property from 0 to 0.2 at the first update and again by 0.2 at each update thereafter:

```
if (this.progressCounter > 1) {
    // This operation is complete!
    this.completeProgress();
}
else {
    this.pillModel.progress = this.pillModel.progress + 0.2;
    this.controller.modelChanged(this.pillModel);
}
```

Progress Bar

The Progress Bar widget is exactly the same as the progress pill, except that you use `x-mojo-element="ProgressBar"` in your scene file. Otherwise, you code it and manage it just as you do the progress pill. Figure 5-3 shows a progress bar.

Figure 5-3. A Progress Bar widget example

In the default style, there isn't room on the bar for a title or image, but the properties are supported nonetheless.

Progress Slider

For media or other applications where you want to show progress as part of a tracking slider, the Progress Slider widget is an ideal choice. Combining the Slider widget with the progress pill, the behavior is fully integrated, but not all of the configuration options are represented. Figure 5-4 shows a progress slider.

Figure 5-4. A Progress Slider widget example

All of the slider properties are represented, and you configure the progress slider just as you would a Slider widget. You have a model property of `value`, which can be renamed through the attributes property. You manage it exactly as you would the progress pill, by progressively increasing the value property from 0 to 1.

Dynamic Widgets

All of our examples start with declaring a widget within an HTML scene and doing almost everything else in JavaScript. It is also possible to eliminate even the HTML declaration and create widgets dynamically from within JavaScript.

For example, you can use the following:

```
this.controller.setupWidget("my-widget", Attr, this.widgetModel);
```

Later, the element is added to the DOM:

```
this.target = this.controller.get("an-element");
this.target.innerHTML = "<div id='my-widget' x-mojo-element='List'></div>";
```

Then you have to call `this.controller.instantiateChildWidgets` to parse and apply the setup. Mojo provides a function that does this automatically, so you could instead write this to instantiate the widget:

```
this.controller.update($("an-element"), "<div id='my-id' x-mojo-element='List'></div>")
```

You can even be completely dynamic and generate everything at runtime. Call `setupWidget()` before generating the ID and injecting the widget's node into the DOM.

Widgets inside of lists are a little tricky, since they are automatically instantiated with the list. You can still create them dynamically by using the widget's name attribute and accommodating for their model as part of the list's model.

Scrollers

The Scroller widget provides the scrolling behavior in Mojo. A scroller is installed automatically in every scene, and you can have any number of additional scrollers anywhere in the DOM.

You can disable the scroller in a scene by setting the `disableSceneScroller` property to true in the scene arguments to `pushScene`.

In the current release of Mojo, you can select one of six scrolling modes, specified in the mode property of the widget's attributes:

`free`
: Allow scrolling along both the horizontal and vertical axes.

`horizontal`
: Allow scrolling only along the horizontal axis.

`vertical`
: Allow scrolling only along the vertical axis.

`dominant`
: Allow scrolling along the horizontal or vertical axis, but not both at once. The direction of the initial drag will determine the scrolling axis.

`horizontal-snap`
: In this mode, scrolling is locked to the horizontal axis, but snaps to points determined by the position of the block elements found in the model's `snapElements` property. As the scroller scrolls from snap point to snap point it will send a `propertyChange` event.

`vertical-snap`
: This mode locks scrolling to the vertical axis, and snaps to points determined by the elements in the `snapElements` property array.

Upon rendering, the widget targets its single child element for scrolling. If it has more than one child element, it will create a single div to wrap the child elements. It will never update this element, so if you replace the contents of a Scroller widget after it is

instantiated, scrolling might not work. Instead, put another block element inside the scroller and update its contents as needed.

The size of the scroller's target div, the child element, must be set in CSS. By default, the div will expand to the size of the contents. You must constrain the width, on a horizontal scroller, or the height, on a vertical scroller, within your CSS or the scroller will not function.

A Scroller widget will ignore any drag start event that doesn't indicate a valid scroll start for its mode setting, so you can nest scrollers if they don't conflict. For example, you can put a small horizontal scroller inside the default scene vertical scroller. This configuration will pass horizontal swipes to the horizontal scroller, but vertical swipes on the horizontal scroller, or any kinds of swipes outside the horizontal scroller, will be passed to the scene scroller.

Back to the News: Adding a featured feed Scroller

In this example, a rotating feature story will be added to the News application. This will present a title and story from the feed list for 5 seconds, after which time it will be replaced by another story. This rotating feature story will be displayed in a fixed-size area above the `feedListWgt` to allow for a stable view, but we'll attach a vertical scroller to allow users to read the full story if it is longer than the view, and we'll enclose it all in a drawer so that users can selectively enable or disable this view.

Start by modifying the `feedList` scene *(feedList-scene.html)* to add an icon to the `palm-header` to serve as a tap target to open and close the drawer. We're going to change the image to match the state of the drawer by using two different classes, `featureFeed-close` and `featureFeed-open`. We'll start with the drawer closed.

The Scroller is declared and encloses `featureStoryDiv`, which will be fixed to a specific height through CSS and filled by the story title and text. This is all placed above the `feedList` widget in the scene's layout.

Within the `featureStoryDiv`, define the `splashScreen` div to fill the space for the initial launch case where there are no stories to display. The `update-image` style will insert the News icon alongside a copyright notice for the application. We'll hide this div when there are stories to display; each story `title` and `text` will be inserted in the following divs, identified as `featureStoryTitle` and `featureStory`:

```
<div id="feedListScene">
    <div id="feedListMain">

        <!--    Rotating Feature Story                          -->
        <div id="feedList_view_header" class="palm-header left">
            Latest News
            <div id="featureDrawer" class="featureFeed-close"></div>
        </div>
```

```
        <div class="palm-header-spacer"></div>
        <div x-mojo-element="Drawer" id="featureFeedDrawer">
            <div x-mojo-element="Scroller" id="featureScroller" >
                <div id="featureStoryDiv" class="featureScroller">
                    <div id="splashScreen" class="splashScreen">
                        <div class="update-image"></div>
                        <div class="title">News v0.8<span>#{version}</span>
                            <div class="palm-body-text">
                                Copyright 2009, Palm®
                            </div>
                        </div>
                    </div>
                    <div id="featureStoryTitle" class="palm-body-title">
                    </div>
                    <div id="featureStory" class="palm-body-text">
                    </div>
                </div>
            </div>
        </div>

        <!--    Feed List                                          -->
        <div class="palm-list">
            <div x-mojo-element="List" id="feedListWgt"></div>
        </div>
    </div>
</div>
```

Set up the Drawer and the Scroller in the *feedList-assistant.js*. Set up a listener for taps to the new tap target in the header and then the Drawer widget with the state defined by a new global, `News.featureFeedEnable`. In Chapter 6, we'll add saved preferences and we will retain the drawer's state at that time.

The `featureScrollerModel` defines a single property, the scroller `mode`, in this case set to vertical, and the Scroller is set up with just its ID and model as arguments. The scroller simply responds to vertical swipes to scroll the content, and will generate scrolling events if you want to receive them, although you don't normally need to:

```
// Setup header, drawer, scroller and handler for feature feeds

    this.featureDrawerHandler = this.toggleFeatureDrawer.bindAsEventListener(this);
    this.controller.listen("featureDrawer", Mojo.Event.tap,
        this.featureDrawerHandler);

    this.controller.setupWidget("featureFeedDrawer", {},
      this.featureFeedDrawer = {open: News.featureFeedEnable});

    this.featureScrollerModel = {
        mode: "vertical"
        };

    this.controller.setupWidget("featureScroller", this.featureScrollerModel);
    this.readFeatureStoryHandler = this.readFeatureStory.bindAsEventListener(this);
    this.controller.listen("featureStoryDiv", Mojo.Event.tap,
        this.readFeatureStoryHandler);
```

```
    // If feature story is enabled, then set the icon to open
    if (this.featureFeedDrawer.open === true) {
        this.controller.get("featureDrawer").className = "featureFeed-open";
    } else  {
        this.controller.get("featureDrawer").className = "featureFeed-close";
    }
```

A listener is set up to handle taps in the feature story div, but it is unrelated to the Scroller. If users see a story they want to read further, they can scroll the story or tap it to go to the `storyView` scene with that story. And we'll finish by setting the `featureDrawer` element's `className` to match the state of the drawer; this is also a future provision for using saved preferences when the scene could be activated with the drawer in the open state.

The CSS completes the implementation by fixing the size of the Scroller div and formatting the contents. The first two rules support the tap target, either open or closed, and the drawer background. You'll have to add the appropriate images to the images folder of your application to reproduce this:

```
/* FeedList Header styles for feature drawer and selection */

.featureFeed-close {
    float:right;
    margin: 8px -12px 0px 0px;
    height:35px;
    width: 35px;
    background: url(../images/details-open-arrow.png) no-repeat;
}

.featureFeed-open {
    float:right;
    margin: 8px -12px 0px 0px;
    height:35px;
    width: 35px;
    background: url(../images/details-closed-arrow.png) no-repeat;
}

.palm-drawer-container {
    border-width: 20px 1px 20px 1px;
    -webkit-border-image: url(../images/palm-drawer-background-2.png)
        20 1 20 1 repeat repeat;
    -webkit-box-sizing: border-box;
    overflow: visible;
}

.featureScroller {
    height: 100px;
    width: 280px;
    margin-left: 20px;
}
```

The `palm-drawer-container` selector sets the drawer container's dimensions and applies a white background to contrast with the scene's main background. The

`featureScroller` style bounds the scroller's height because we have a vertical scroller; in the case of a horizontal scroller, we'd bound the width. If you aren't careful with your CSS, the scroller will not behave as expected.

The rest of the sample is related to handling the feature story. Create a new method, `toggleFeatureDrawer()`, to open or close the drawer. If the drawer is open, the method will close it by setting the drawer's model to false, and setting the div class to `feature Feed-close`. If the drawer is closed, the actions are reversed to open the drawer and we'll start the story rotation if it's not already running. The controller's `modelChanged()` method is called to signal the model changes to the drawer widget:

```
// toggleFeatureDrawer - handles taps to the featureFeed drawer. Toggle
//   drawer and icon class to reflect drawer state.
FeedListAssistant.prototype.toggleFeatureDrawer =  function(event) {
       var featureDrawer = this.controller.get("featureDrawer");
       if (this.featureFeedDrawer.open === true) {
           this.featureFeedDrawer.open = false;
           News.featureFeedEnable = false;
           featureDrawer.className = "featureFeed-close";
       } else {
           this.featureFeedDrawer.open = true;
           News.featureFeedEnable = true;
           featureDrawer.className = "featureFeed-open";

           // If there's some stories in the feedlist, then start
           // the story rotation even if the featureFeed is disabled as we'll use
           // the rotation timer to update the DB
           if(this.feeds.list[this.featureIndexFeed].stories.length > 0) {
               var splashScreenElement = this.controller.get("splashScreen");
               splashScreenElement.hide();
               this.showFeatureStory();
           }
       }
       this.controller.modelChanged(this.featureFeedDrawer);
 };
```

Create the `showFeed()` method to present the feature story and set up the timer for the next story. We set the timer default to 5 seconds (5000 milliseconds) in `News.featureStoryInterval`, another new global variable, added to *stage-assistant.js*. If the timer has been set, rotate the story by taking the next story in the current feed or the first story in the next feed if it's at the end of the current feed. We will also add some logic to strip URLs and other HTML from the title and text:

```
// ---------------------------------------------------------------------
// Feature story functions
//
// showFeatureStory - simply rotate the stories within the
// featured feed, which the user can set in their preferences.
FeedListAssistant.prototype.showFeatureStory = function() {

    // If timer is null, either initial story or restarting. Start with
    // previous story..
    if (News.featureStoryTimer === null)    {
```

```
    // ** These next two lines are wrapped for book formatting only **
        News.featureStoryTimer = this.controller.window.setInterval(
          this.showFeatureStory.bind(this), News.featureStoryInterval);
    }

    else {
        this.featureIndexStory = this.featureIndexStory+1;
        // ** These next two lines are wrapped for book formatting only **
        if(this.featureIndexStory >=
          this.feeds.list[this.featureIndexFeed].stories.length) {
            this.featureIndexStory = 0;
            this.featureIndexFeed = this.featureIndexFeed+1;
            if (this.featureIndexFeed >= this.feeds.list.length)    {
                this.featureIndexFeed = 0;
            }
        }
    }

    var summary = this.feeds.list[this.featureIndexFeed].stories[
       this.featureIndexStory].text.replace(/(<([^>]+)>)/ig,"");
    summary = summary.replace(/http:\S+/ig,"");
    var featureStoryTitleElement = this.controller.get("featureStoryTitle");
    // ** These next two lines are wrapped for book formatting only **
    featureStoryElement.innerHTML =
        unescape(this.feeds.list[this.featureIndexFeed].stories[
          this.featureIndexStory].title);
    var featureStoryElement = this.controller.get("featureStory");
    featureStoryElement.innerHTML = summary;

    // Because this is periodic and not tied to a screen transition, use
    // this to update the db when changes have been made

    if (News.feedListChanged === true)    {
        News.feedListChanged = false;

        this.feedWgtModel.items = this.feeds.list;
        this.controller.modelChanged(this.feedWgtModel, this);
    }

};

// readFeatureStory - handler when user taps on feature story;
//   will push storyView with the current feature story.
FeedListAssistant.prototype.readFeatureStory = function() {
    Mojo.Controller.stageController.pushScene("storyView",
      this.feeds.list[this.featureIndexFeed],
      this.featureIndexStory);
};
```

Following `showFeatureStory()` is `readFeatureStory()`, which simply pushes the `storyView` scene with the current feature story.

When you run the application, you'll see a different look, with the feature story now filling the top third of the `feedList` scene, as shown in Figure 5-5.

Figure 5-5. News with scrolling feature feed

There's still some cleanup needed. We must maintain `this.featureIndexFeed` during list reordering, and delete. You can see the changes made to the `listDeleteHandler()` and `listReorderHandler()` in the *feedList-assistant.js* in the News source listing in Appendix D.

Pickers

Pickers are used to present a common UI for selecting inputs in a variety of application scenarios. Mojo offers pickers for common objects such as a date, time, or number, or to select files.

The next section covers the first three pickers, since they are conventional widgets and are very similar to each other. After that, we will look at the File picker. It's accessed through function calls, and is actually implemented as a separate application wrapped with a framework interface.

Simple Pickers

The models for the date, time, and integer pickers are very similar. The pickers are declared within your scene's view file and wrapped with styling divs as shown here:

```
<div class="palm-group unlabeled">
  <div class="palm-list">
    <div id="DatePkrId" x-mojo-element="DatePicker"></div>
  </div>
</div>
```

This creates a picker that spans the width of the screen and is enclosed with an unlabeled group frame (as shown in Figure 5-6).

Figure 5-6. A Date Picker widget example

These pickers present choices as a linear sequence of values that wraps around; when you scroll to the end of the sequence, they simply continue back at the beginning. There's no way to override this behavior.

Date pickers

As shown in Figure 5-6, a date picker allows selection of month, day, and year values. The Date picker's model has a single property, named `date` by default, which should be assigned a JavaScript Date object. You can change the model property's name through the attributes `modelProperty`, and can assign an optional label that's displayed to the left of the picker. You can use the JavaScript functions `GetMonth()`, `GetDate()`, or `GetYear()` to extract those parts of the Date object that you need.

Time pickers

A Time picker is similar to a date picker, focusing on the time fields of the Date object and with an optional attributes property, `minuteInterval`, which defaults to the integer 5. As shown in Figure 5-7, a time picker allows selection of hours, minutes, and either A.M. or P.M. for time selection. The picker will suppress the AM/PM capsule if the 24-hour time format is selected in the user preferences or by the locale.

Figure 5-7. A Time picker widget example

You can use the JavaScript functions `GetHours()`, `GetMinutes()`, and `GetSeconds()` to extract what you need from the `Date` object.

Integer pickers

A simple number picker is included as the integer picker. Shown in Figure 5-8, the integer picker offers a selection between minimum and maximum integer values, both of which are specified by required `min` and `max` widget properties.

Figure 5-8. An Integer picker widget example

The integer picker is similar to the date and time pickers in all ways, except that its default model property is named `value`.

Back to the News: Adding an integer picker

News doesn't have many opportunities to use a picker, but we can add a picker to the Preferences scene to set the feed rotation period. There are arguably better options for handling this UI, so this isn't intended as a good UI design example, but as a way of demonstrating the coding for a Picker widget. By default, the interval is set to 5 seconds, but with this picker, the user can customize to any period between 1 and 60 seconds.

The Integer picker declaration is added to *preferences-scene.html* in front of the list selector widget for the update interval. Wrap it in a couple of divs with `palm-group` and `palm-list` styles:

```
<div class="palm-group">
    <div class="palm-group-title"><span>Feature Feed</span></div>
        <div class="palm-list">
            <div x-mojo-element="IntegerPicker" id="featureFeedDelay">
            </div>
        </div>
    </div>
</div>
```

The widget setup is included in the `setup` method of *preferences-assistant.js*, as shown next. The widget is set up with a range from 1 to 20 seconds and initialized to the current global interval, `News.featureStoryInterval`, which is in milliseconds. A listener is added for `Mojo.Event.propertyChange` events:

```
// Setup Integer Picker to pick feature feed rotation interval
this.controller.setupWidget("featureFeedDelay",
    {
        label:    "Rotation (in seconds)",
        modelProperty:    "value",
```

```
            min: 1,
            max: 20
        },
        this.featureDelayModel = {
            value : News.featureStoryInterval/1000
        });

    this.changeFeatureDelayHandler = this.changeFeatureDelay.bindAsEventListener(this);
    this.controller.listen("featureFeedDelay", Mojo.Event.propertyChange,
        this.changeFeatureDelayHandler)
```

The handler is added to the bottom of the preferences assistant, updating the global interval with the selected value, in milliseconds, and restarts the interval timer with the new value. The interval timer would be set by the `FeedListAssistant` upon activation or when the feed list is first updated:

```
//    changeFeatureDelay - Handle changes to the feature feed interval
PreferencesAssistant.prototype.changeFeatureDelay = function(event) {
    Mojo.Log.info("Preferences Feature Delay Handler; value = ",
        this.featureDelayModel.value);

    //  Interval is in milliseconds
    News.featureStoryInterval = this.featureDelayModel.value*1000;

    // If timer is active, restart with new value
    if(News.featureStoryTimer !== null) {
        this.controller.window.clearInterval(News.featureStoryTimer);
        News.featureStoryTimer = null;
    }
};
```

Figure 5-9 shows the new Preferences scene with the integer picker in place.

File Picker

WebOS devices have a *media partition*, a FAT32 file partition that is available to applications and is accessible to desktop operating systems, whether PC, Mac, or Linux, when the device is attached through a USB cable. This access mechanism is called USB mode.

The file picker presents a file browser that users can use to navigate the directory structure and optionally select a file. The file picker presents a flat listing of all files on disk, regardless of directory structure, and allows filtering by file type (such as file, image, audio, or video) Depending on the options provided by the calling application, the selected file will either be opened in an appropriate viewer or have its reference returned.

The file picker behaves like a full-screen widget, but isn't technically a widget. It is actually an application that is pushed into the current scene, similar to a viewer, maintaining the calling application's context.

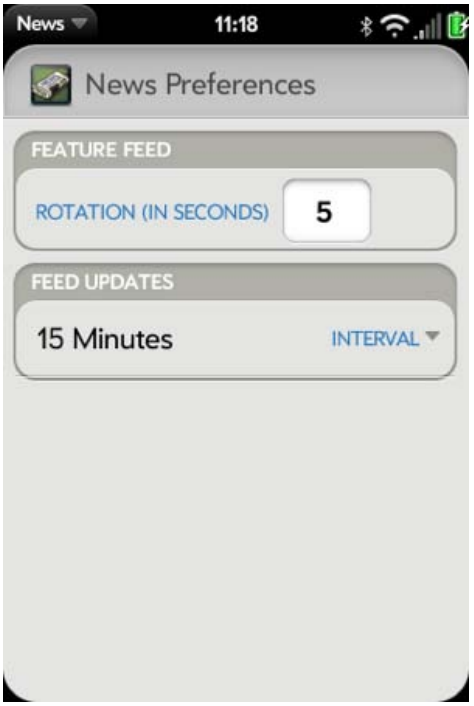


Figure 5-9. News with an integer picker

The presentation of the files will differ by file type. For example:

- Files: Name and icon
- Images: Thumbnail grid
- Audio and Video: Name and thumbnail

Figure 5-10 shows the file view, with the other view options presented as Command menu items across the bottom of the scene.

Advanced Lists

Lists were introduced in Chapter 3 with several extensive examples. Even so, some major list features weren't touched on. We'll take a look at some more advanced features here.

With all list widgets, you can intervene in the middle of the list rendering to provide some intermediate formatting to list items or to insert dividers between rows. After a brief review of those features, we will add a Filter List widget to News to implement a

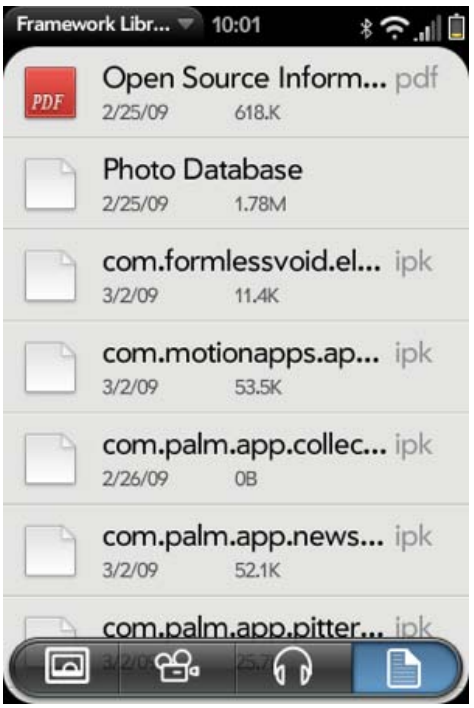


Figure 5-10. A file picker example

search feature. This is a good example of a dynamic list, something you can use in many different types of applications.

Formatters

The `formatters` property is a hash of property names to formatter functions, like this:

```
{timeValue: this.myTimeFormatter, dayOfWeek: this.dayIndexToString, ... }
```

Before rendering the relevant HTML templates, the formatter functions are applied to the objects used for property substitution. The keys within the `formatters` hash are property names to which the formatter functions should be applied.

The original objects are not modified, and the formatted properties are given modified new names so that the unformatted value is still accessible from inside the HTML template. Formatted values have the text "Formatted" appended to their names. In the example above, the HTML template could refer to `#{timeValueFormatted}` in order to render the output from the `myTimeFormatter()` function. Formatter functions receive the relevant property value as the first argument, and the appropriate model object or items element as the second.

Dividers

You can add dividers to your lists; they are particularly useful for long lists. You will specify the function name to the `dividerFunction` property and a template to `dividerTemplate`. If no template is specified, the framework will use the default, a single-letter alpha divider (*list-divider.html*, styled with the `palm-alpha-divider` class).

The divider function works similar to a data formatter function. It is called with the item model as the sole argument during list rendering, and it returns a label string for the divider.

Filter Lists

Use a Filter List widget when your list is best navigated with a search field, particularly one where you would like to instantly filter the list as each character is typed into the field. It is intended to display a variable length list of objects, built by a special callback function.

The widget includes a text field displayed above a list, where the list is the result of applying the contents of the text field through an application-specific callback function against some off-screen data source. The text field is hidden when empty, but it is given focus as soon as any key input is received. At the first keystroke, the field is displayed with the key input (after a delay; specified via the `delay` attribute—default is 300ms), and the framework calls the function specified by `filterFunction`.

The framework calls the `filterFunction`, which is similar to the `itemsCallback` function, in the base List widget (see Chapter 3) when data is needed for displaying list entries. You provide the `filterFunction`, with arguments for the list widget element, offset, and count, similar to `itemsCallback`, plus an additional argument, `filterString`.

It is understood that the requested data may not be immediately available. Once the data is available, the given widget's `noticeUpdatedItems()` method should be called to update the list. It's acceptable to call the `noticeUpdatedItems()` immediately if desired, or any amount of time later. Lengthy delays may cause various scrolling artifacts, however.

The filter list will display a spinner in the text field while the list is being built, and it is replaced with an entry count when done. To set the count properly, call the widget's `setCount(totalSubsetSize)`, where `totalSubsetSize` is the number of entries in the list. To set the list length, call `setLength(totalSubsetSize)`; the length is a dependency of some internal widget functions and needs to be set accurately.

Back to the News: Adding a search field

Search is one of the best applications for the Filter List widget. You can start typing on the `feedList` scene and quickly search all feeds for a keyword, viewing the results in the same list format used for the individual feed lists. It's simple to access and powerful.

The Filter List is declared and set up conventionally, but requires a filter function called to process the keyword entries and returns a list for display. In the News design, we're going to put the search results into a temporary list that is structured like the feed list. We will display it using the `storyList` assistant.

Because the Filter List search field is going into the `feedList` view, it will be necessary to hide the feed list and feature story when in search mode. Start by adding the Filter List declaration at the top of *feedList-scene.html* and wrap the rest of the scene in a div with `feedListMain` as its ID. This will be used later to hide the rest of the scene:

```
<!--    Search Field                                          -->
    <div id="searchFieldContainer">
        <div x-mojo-element="FilterList" id="startSearchField"></div>
    </div>

    <div id="feedListMain">
    <!--    Rotating Feature Story      -->
      .
      .
      .
```

Now set up the widget in *feedList-assistant.js*, reusing the `storyList` templates, preparing to format the search results in a `storyList` scene. Identify the filter function as `this.searchList`, and use a standard delay of 300 milliseconds. This is the default, so this step can be omitted, but you may need to tune the behavior, so it's not a bad idea to add it at the beginning:

```
// Setup the search filterlist and handlers;
this.controller.setupWidget("startSearchField",
    {
        itemTemplate: "storyList/storyRowTemplate",
        listTemplate: "storyList/storyListTemplate",
        filterFunction: this.searchList.bind(this),
        renderLimit: 70,
        delay: 300
    },
    this.searchFieldModel = {
        disabled: false
    });

this.viewSearchStoryHandler = this.viewSearchStory.bindAsEventListener(this);
this.controller.listen("startSearchField", Mojo.Event.listTap,
    this.viewSearchStoryHandler);
this.searchFilterHandler = this.searchFilter.bindAsEventListener(this);
this.controller.listen("startSearchField", Mojo.Event.filter,
    this.searchFilterHandler, true);
```

Add two listeners, one for the tap event and the other for a filter event. The filter event is used on Filter Field and Filter List only. It is sent after the defined delay once the filter field is activated, on the first character entry, and immediately when the field is cleared. Listening for this event allows you to do some pre- and postprocessing with the widget or the scene.

With News, we want to hide the rest of the scene when the first character is typed, and restore it when the field is cleared. To do this, add this new method in the `feedList` assistant; it simply calls the Prototype methods `hide()` and `show()` of the div element, `feedListMain`, in *feedlist-scene.html*:

```
// searchFilter - triggered by entry into search field. First entry will
//  hide the main feedList scene - clearing the entry will restore the scene.
//
FeedListAssistant.prototype.searchFilter = function(event)    {
    Mojo.Log.info("Got search filter: ", event.filterString);
    var feedListMainElement = this.controller.get("feedListMain");
    if (event.filterString !== "")    {
        //    Hide rest of feedList scene to make room for search results
        feedListMainElement.hide();
    }    else    {
        //    Restore scene when search string is null
        feedListMainElement.show();
    }

};
```

After the filter event is sent, the framework calls the function assigned to the `filterFunction` property in the Filter List widget's attributes; in this case, `this.searchList()`, which is added also to *feedList-assistant.js* and shown here:

```
// searchList - filter function called from search field widget to update the
//  results list. This function will build results list by matching the
//  filterstring to the story titles and text content, and then return the
//  subset of the list based on offset and size requested by the widget.
//
FeedListAssistant.prototype.searchList = function(filterString,
  listWidget, offset, count)    {

    var subset = [];
    var totalSubsetSize = 0;

    this.filter = filterString;

    // If search string is null, return empty list, otherwise build results list
    if (filterString !== "")    {

        // Search database for stories with the search string; push matches
        var items = [];

        // Comparison function for matching strings in next for loop
        var hasString = function(query, s) {
            if(s.text.toUpperCase().indexOf(query.toUpperCase())>=0) {
                return true;
            }
            if(s.title.toUpperCase().indexOf(query.toUpperCase())>=0) {
                return true;
            }
            return false;
        };
```

```
            for (var i=0; i<this.feeds.list.length; i++) {
                for (var j=0; j<this.feeds.list[i].stories.length; j++) {
                    if(hasString(filterString, this.feeds.list[i].stories[j])) {
                        var sty = this.feeds.list[i].stories[j];
                        items.push(sty);
                    }
                }
            }

            this.entireList = items;
            Mojo.Log.info("Search list asked for items: filter=",
                filterString, " offset=", offset, " limit=", count);

            // Cut down list results to just the window asked for by the widget
            var cursor = 0;
            while (true) {
                if (cursor >= this.entireList.length) {
                    break;
                }

                if (subset.length < count && totalSubsetSize >= offset) {
                    subset.push(this.entireList[cursor]);
                }
                totalSubsetSize++;
                cursor++;
            }
        }

        // Update List
        listWidget.mojo.noticeUpdatedItems(offset, subset);

        // Update filter field count of items found
        listWidget.mojo.setLength(totalSubsetSize);
        listWidget.mojo.setCount(totalSubsetSize);

    };
```

The function definition is `filterFunction (filterString, listWidget, offset, limit)`, using the arguments shown in Table 5-1.

Table 5-1. FilterFunction arguments

Argument	Type	Description
`filterString`	String	The contents of the filter field or the search string to be used
`listWidget`	Object	The DOM node for the list widget requesting the items
`offset`	Integer	Index in the list of the first desired item model object (zero-based)
`limit`	Integer	Count of the number of item model objects requested

Assuming the `filterString` isn't empty, which it shouldn't be, the first part of the method will do a primitive match against all the story titles and text content across all feeds. The results are pushed into the items array and assigned to `this.entireList` when complete.

The list is cut down to just the portion of the list that was requested by the `offset` and the `count`, and is assigned to `subset`, which is returned with the offset by calling `listWidget.mojo.noticeUpdatedItems(offset, items)`, using the arguments shown in Table 5-2. This is a method of `listWidget`, an argument passed by the framework.

Table 5-2. NoticeUpdatedItems arguments

Argument	Type	Description
`offset`	Integer	Index in the list of the first object in `items`; usually the same as the offset passed to the `itemsCallback`
`items`	Array	An array of the list item model objects that have been loaded for the list

Finish up with calls to `listWidget.mojo.setLength(totalSubsetSize)` and `listWidget.mojo.setCount(totalSubsetSize)` to set the list length and the results count for the counter displayed in the filter field.

With these changes, users can type at any time into the `feedList` scene to see the filter field display and the results presented in list form below, similar to what is shown in Figure 5-11.

Figure 5-11. News with a search filter list

Figure 5-12. News with a search story view

After the list is built, the tap event indicates a user selection of a list entry, just as it does for a conventional list. Since the search list is built as a stories array, News responds to a tap by creating a temporary feed and pushing the `storyView` scene with that feed. Here's the `viewSearchStory()` method:

```
// viewSearchStory - triggered by tapping on an entry in the search
//   results list  will push the storyView scene with the tapped story.
//
FeedListAssistant.prototype.viewSearchStory = function(event)    {
    var searchList = {
      title: "Search for: "+this.filter, stories: this.entireList
    };
    var storyIndex = this.entireList.indexOf(event.item);

    Mojo.Log.info("Search display selected story with title = ",
        searchList.title, "; Story index - ", storyIndex);
    Mojo.Controller.stageController.pushScene("storyView",
      searchList, storyIndex);

};
```

As shown in Figure 5-12, as well as viewing the selected story, users can tap Next and Previous to view each story in the results list.

Viewers

With Mojo, you can embed rich media objects within your scenes. There are widgets for a web object, a full screen image scroller, and partial support for HTML 5 audio and video tags for inclusion of audio and video objects.

WebView

To embed a contained web object, declare and instantiate a `WebView` widget. You can use it to render local markup or to load an external URL; as long as you can define the source as a reachable URL, you can use a `WebView` to render that resource.

Back to the News: Adding a web view

Tapping on a story will push a web view scene and load the original story's URL in that scene. This example is a simple use of Web view, where we create a new scene for the web page, but it's still within the News application's context.

Create a new scene, called webView, using `palm-generate`, then declare the widget in your scene view and configure it in your scene assistant before calling `setupWidget()`. The *storyWeb-scene.html* is just one line:

```
<div id="storyWeb" x-mojo-element="WebView"></div>
```

And there's not much more to the *storyWeb-assistant.js* to configure and set up the WebView widget:

```
/*  StoryWebAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Passed a story URL, displays that element in a full scene webview with
    a load indicator and reload button. Handles link selections within the
    view. User swipes back to return to the calling view.

    Major components:


    Arguments:
    - storyURL; Selected feed from which the stories are being viewed

*/


function StoryWebAssistant(storyURL) {
  //  Save the passed URL for inclusion in the webView setup
  this.storyURL = storyURL;
}

StoryWebAssistant.prototype.setup = function() {

    //  Setup up the webView widget
```

```
    this.controller.setupWidget("storyWeb", {url: this.storyURL},
        this.storyViewModel = {});

    //    Setup handlers for any links selected.
    this.linkClickedHandler = this.linkClicked.bindAsEventListener(this);
    this.controller.listen("storyWeb", Mojo.Event.webViewLinkClicked,
      this.linkClickedHandler);

    // Setup App Menu
    this.controller.setupWidget(Mojo.Menu.appMenu, News.MenuAttr, News.MenuModel);

};

StoryWebAssistant.prototype.cleanup = function(event) {
    this.controller.stopListening("storyWeb", Mojo.Event.webViewLinkClicked,
      this.linkClickedHandler);

};

//    linkClicked - handler for selected links, requesting new links to be opened
//    in same view
StoryWebAssistant.prototype.linkClicked = function(event) {
    Mojo.Log.info("Story Web linkClicked; event.url = ", event.url);
    var link = this.controller.get("storyWeb");
    link.mojo.openURL(event.url);
};
```

There are more options than what's shown here. You can set the virtual page used to render through the attributes properties `virtualpageheight` and `virtualpagewidth` and set the `minFontSize`. You can add listeners for many Mojo web events, including `webViewLoadProgress`, `webViewLoadStarted`, `webViewLoadStopped`, and `webViewLoadFailed` to intervene during any web page load. There are even more events than that; you can find a complete list of events and descriptions in the webOS SDK.

To get to the `webView`, `storyView` will be modified to add another command menu, this time to present a button on the lower-left of the scene to launch the `webView` scene:

```
this.storyMenuModel = {
        items: [
            {iconPath: "images/url-icon.png", command: "do-webStory"},
            {},
            {items: []},
            {},
            {}
        ]};

        if (this.storyIndex > 0)    {
            this.storyMenuModel.items[2].items.push({
              icon: "back",
              command: "do-viewPrevious"
            });
        } else {
            this.storyMenuModel.items[2].items.push({
              icon: "", command: "",
```

```
            label: "  "
          });
        }

        if (this.storyIndex < this.storyFeed.stories.length-1)    {
            this.storyMenuModel.items[2].items.push({
              icon: "forward",
              command: "do-viewNext"
            });
        } else {
           this.storyMenuModel.items[2].items.push({
              icon: "", command: "",
              label: "  "
           });
        }

    this.controller.setupWidget(Mojo.Menu.commandMenu, undefined,
 this.storyMenuModel);
```

Next, push the `storyWeb` scene in *storyView-assistant.js* by adding another command handler in `handleCommand()` after those for `do-viewNext` and `do-viewPrevious`:

```
case "do-webStory":
    Mojo.Log.info("View Story as a Web View Menu; url = ",
      this.storyFeed.stories[this.storyIndex].url);
    Mojo.Controller.stageController.pushScene("storyWeb",
      this.storyFeed.stories[this.storyIndex].url);
    break;
```

Load Indicator

Whether you're using a full-screen web view or adding to the end of your scene, you may want to use a loading indicator similar to the webOS browser. Even with a fast browser, many pages take a few seconds to load, and it's helpful to have some type of indication for the user.

There is a detailed sample in the Palm SDK, but here's a brief description of the design:

- Set up Command menu buttons for your indicator and Reload button; only one will be displayed at a time, but you'll switch them back and forth, so you need to set up both of them.
- Add listeners for the `loadStarted` and `loadStopped` events to switch the menu buttons and update the Command menu model.
- The body of the indicator handling is in a listener for `loadProgress`, which calculates the percent complete, using the `loadProgress event.progress` value, and invokes an update function to select the appropriate image based on the percent complete.
- That image is inserted into the DOM to act as the indicator.

As shown in Figure 5-13, the WebView widget is put into its own scene. It can also be declared within a scene so that a URL can be passed to it after it has been set up and

the scene is active. For example, an application such as Email, which might have to present HTML content, can declare and set up a widget without the `url` property defined:

```
this.controller.setupWidget("storyWeb", {}, this.storyViewModel = {});
```

And when the URL is available, call the widget's `openURL` method:

```
var webview =  this.controller.get("storyWeb");
webview.mojo.openURL(URL);
```

The web content will be displayed wherever the widget is declared within your scene; you can use another property, `topMargin`, to automatically scroll part of the scene to expose the top of the web view if that's useful.

Figure 5-13. A WebView widget example

Other Viewers

There are other viewers that you can add to your application: image view, audio players, and video players. `ImageView` is very similar to `WebView`, but the audio and video players are quite different. None of these other viewers is presented in depth here, but each is briefly described. There is a lot more information available in the webOS SDK.

Image view

Designed to view an image in full screen, with support for zooming and panning while optionally paging between additional images, the ImageView widget is configured much like the WebView widget. You can use an ImageView for displaying single images, but it is intended as a scrolling viewer, flicking left and right through a series of images. The example in Figure 5-14 shows an implementation of the ImageView widget, partially paged to the right.

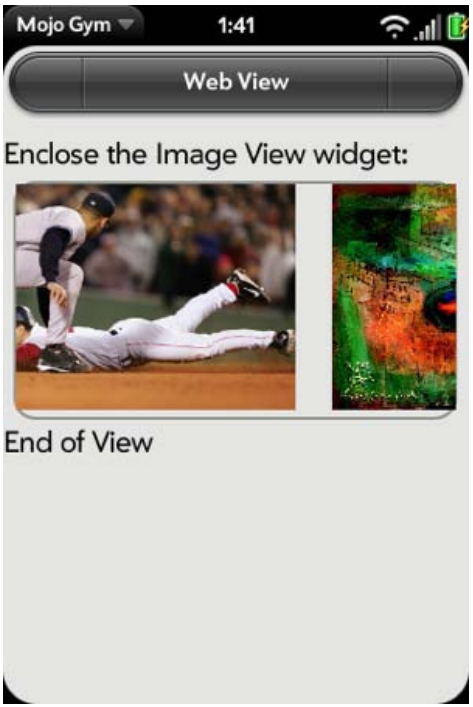


Figure 5-14. An ImageView widget example

Audio and video objects

There are application services supporting playback through the core audio and video applications, but for playback within your application, you can include audio and video objects that are based on the HTML 5 media extensions. You should use these objects when you want to maintain your application's context or play content directly within your scene. The application services, which are discussed in Chapter 8, are the best options when you just want to play a music track or some video.

The Audio object provides playback of audio based on the HTML 5 audio element definition. However, you must create the object using JavaScript in one of your assistants, as Mojo doesn't currently support creating objects directly through tags in HTML.

Audio objects are created from the Audio constructor in your assistant's setup method, and to play the audio, you set the source and event handlers before calling the object's play method. You can set up multiple audio objects to minimize delays when playing successive audio tracks.

Similarly, a Video object based on the HTML 5 video element definition provides video playback. As with the Audio object, you will play video after creating the Video object, setting the source and setting up event handlers.

Unlike audio, video playback requires coordination of the *video sink* through helper functions to `freezeVideo` and `activateVideo`. When an application is not in the foreground, it must release the video sink in case another active application needs to display video.

The media objects support multiple sources; both file-based and streaming sources are supported. You can learn more about the extent of the features and events supported by reviewing the HTML 5 spec at *www.whatwg.org/specs/web-apps/current-work/*.

Summary

With this discussion of advanced widgets, you've now learned all about Mojo's UI features and how to build a range of applications from simple to complex. This chapter covered indicators, widgets for showing activity and measuring progress; pickers and viewers; specialized widgets providing sophisticated interaction with specific data types; and a few advanced widgets, filter lists, and scrollers.

It's time to move on to storage and services that can extend your applications beyond UI and other functional areas that are more like native applications than web applications. Even without going further, you can build some very compelling and unique applications using Mojo's web development model, but the services will give your application some new and powerful options.

CHAPTER 6
Data

Access to local data storage is a signature feature of native application models. Web applications do not have access to local data storage other than browser cookies. Recently, there has been an effort to address these needs, particularly with the proposed HTML 5 APIs for structured client storage.

Palm webOS supports the HTML 5 Database APIs and provides two specific APIs for simple data creation and access:

Depot
: A wrapper on the HTML 5 APIs for simple object storage and retrieval.

Cookie
: A simplified interface to browser cookies, this is a single object store for small amounts of data.

You will have to evaluate your needs to determine which solution is the best.

- Cookies are best used for synchronous access to small amounts of data, such as preferences, version numbers, and other state information.
- Both HTML 5 databases and Depot are intended to support caches for offline access and to help with performance issues while accessing online data.
- Depot is recommended for storing simple objects without a schema design or manual transaction and query handling; otherwise, use an HTML 5 database.
- For disconnected applications that require a data store, an HTML 5 database is the best solution.

Whichever solution you select, it is critical that you provide some local caching for offline use, as stale data is better than no data for most applications. Conversely, you should provide a dynamic source for your data, as it's best to update the source data when the device is connected. Applications that can refresh their data or use online storage instead of the device storage will be more flexible in the long run.

In this chapter, both depot- and cookie-based storage will be covered in depth, with examples shown using the sample News application. The HTML 5 APIs will also be summarized with guidelines on using these APIs and where to find more information. Finally, we will revisit Prototype's Ajax functions. Chapter 3 included an example using the `Ajax.Request` method to update the news feeds. In this chapter, you will get some information on each of the Ajax methods and response handling.

Working with Cookies

The cookie is a well-known browser feature, created early on to store state or session information. Mojo cookies have a similar purpose, and are technically related to browser cookies, but with an object interface to simplify use by webOS applications. Mojo cookies typically store small amounts of data that will preserve application state and related information, such as preference settings.

Palm webOS creates a fake domain for running applications, based on each application ID. Cookies created by the application code are associated with that domain, so unlike browser cookies, they'll never be present as part of web requests to other services; they are strictly for local storage.

You should limit cookies to less than 4kB, but can have multiple cookies within an application if needed. You can remove cookies if they are no longer needed and the framework will delete an application's cookies if the application is removed from the device.

`Mojo.Model.Cookie(id)` opens the cookie that matches the ID argument or, if there is no match, creates a new cookie with that ID. There are three methods:

`get()`
: Retrieves the object stored in the cookie (if it exists) or returns `undefined`.

`put()`
: Updates the value of the named cookie with an optional date/time after which the object is deleted.

`remove()`
: Removes the named cookie and deletes the associated data.

The Cookie function and all of its methods are synchronous, unlike Depot and the database functions, making for a simpler calling interface and return handling.

Back to the News: Adding a Cookie

We'll use a `cookie` object to save the News preferences. Beyond the basics of creating the cookie and retrieving it on launch, we'll also add code to update the cookie when the preferences change.

Since the preferences are used in a few different places in the News application, we'll create the specific News cookie functions in *models/cookie.js*. The `News.Cookie` will have an `initialize()` function, which opens and gets the News cookie if it exists already, or creates the News cookie if not. The cookie's identifier doesn't need to be unique outside of this application:

```
/*  Cookie - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Handler for cookieData, a stored version of News preferences.
    Will load or create cookieData, migrate preferences and update cookieData
    when called.

    Functions:
    initialize - loads or creates newsCookie; updates preferences with contents
        of stored cookieData and migrates any preferences due version changes
    store - updates stored cookieData with current global preferences
*/

News.Cookie = ({

  initialize: function()  {
    // Update globals with preferences or create it.
    this.cookieData = new Mojo.Model.Cookie("NewsPrefs");
    var oldNewsPrefs = this.cookieData.get();
    if (oldNewsPrefs) {
      // If current version, just update globals & prefs
      if (oldNewsPrefs.newsVersionString == News.versionString)    {
        News.featureIndexFeed = oldNewsPrefs.featureIndexFeed;
        News.featureFeedEnable = oldNewsPrefs.featureFeedEnable;
        News.featureStoryInterval = oldNewsPrefs.featureStoryInterval;
        News.feedUpdateInterval = oldNewsPrefs.feedUpdateInterval;
        News.versionString = oldNewsPrefs.newsVersionString;
        News.notificationEnable = oldNewsPrefs.notificationEnable;
        News.feedUpdateBackgroundEnable = oldNewsPrefs.feedUpdateBackgroundEnable;
      } else {
        // migrate old preferences here on updates of News app
      }
    }

    this.storeCookie();

  },

  //  store - function to update stored cookie with global values
  storeCookie: function() {
    this.cookieData.put(    {
      featureIndexFeed: News.featureIndexFeed,
      featureFeedEnable: News.featureFeedEnable,
      feedUpdateInterval: News.feedUpdateInterval,
      featureStoryInterval: News.featureStoryInterval,
      newsVersionString: News.versionString,
      notificationEnable: News.notificationEnable,
```

```
            feedUpdateBackgroundEnable: News.feedUpdateBackgroundEnable
        });
    }

});
```

After creating the cookie in our sample, `this.cookie.get()` is called to retrieve it. If the cookie exists, the global preferences are updated with the stored values.

The second function will update the stored cookie with the current values in the global preferences. It's called from the `initialize()` method in the case where the cookie was first created and also as a future provision when preferences need to be migrated when a new version of the application changes the preferences.

The cookie should be first retrieved at application launch:

```
StageAssistant.prototype.setup = function() {

    // initialize the feeds model
    this.feeds = new Feeds();
    this.feeds.loadFeedDb();

    // load preferences and globals from saved cookie
    News.Cookie.initialize();

    // Set up first timeout alarm
    this.setWakeup();

    this.controller.pushScene("feedList", this.feeds);

};
```

The `storeCookie()` method is called anytime the preferences change. For example, the `deactivate()` method in *preferences-assistant.js* will update the cookie when the Preferences scene is popped:

```
// Deactivate - save News preferences and globals
PreferencesAssistant.prototype.deactivate = function() {
    News.Cookie.storeCookie();
};
```

The `cookie.remove()` method is straightforward, but you may not have any reason to use it at all. With News, the preferences are always retained unless the application is deleted from the device, in which case the storage is recovered by the system. If you are using cookies for temporary storage, you should remove them when they are no longer needed.

Working with the Depot

If you are only interested in a simple object store without any database queries or structure, Depot will likely meet your needs. You can store up to 1 MB of data in a

depot by default. Mojo provides a few simple functions that wrap the HTML 5 APIs to create, read, update, or delete a database.

`Mojo.Depot()` opens the depot that matches the name argument. If there is no match, it creates a new depot with that name. There are four methods:

`get()`
: Calls the provided `onSuccess` handler with the retrieved object (or null if nothing matches the key). `onFailure` is called if an error occurs in accessing the database.

`discard()`
: Removes the data associated with the key from the database.

`add()`
: Updates the value of the named object.

`removeAll()`
: Removes all data in the database by dropping the tables.

The Depot is simple to use. As with Cookie, you call Depot's constructor with a unique name to create a new store or open an existing one. Unlike Cookie, Depot calls are asynchronous, so you will do most of your handling in callback functions. Once opened, you can save and retrieve any JavaScript object. To simplify your data handling, the Depot function will flatten the object so that it can be stored using SQL.

You do need to keep this to simple objects, as Depot is not very efficient, and if you extend it to complex objects it can impact application performance and memory. JSON objects are recommended as the best-performing. Beyond that you'll have to experiment to see how the Depot performs with your application. Deep hierarchy, multiple object layers, array and object datatypes, and large strings are all characteristics of complex objects that may push the limits of the Depot capabilities.

Back to the News: Adding a Depot

The Depot will be used by News to store the feedlist for offline access of the stored feeds, to retain the user's feed choices, and to maintain the stories' unread state. Storing this state information will let us provide a better user experience at launch time and allow us to present the stored feeds quickly, without having to wait for a full sync from the various servers.

The Depot functions will be added to the Feeds data model through two new methods:

`loadFeedDb()`
: Loads the feed database depot or creates one using the default feed list if there isn't an existing depot.

`storeFeedDb()`
: Writes the contents of `Feeds.list` array to the database depot.

The database will be loaded once, when the application is first launched, so the stage assistant is modified:

```
StageAssistant.prototype.setup = function() {

    // initialize the feeds model
    this.feeds = new Feeds();
    this.feeds.loadFeedDb();
```

The load method simply opens the depot or creates one if it doesn't exist:

```
// loadFeedDb - loads feed db depot, or creates it with default list
// if depot doesn't already exist
loadFeedDb: function()  {
  // Open the database to get the most recent feed list
  // DEBUG - replace is true to recreate db every time; false for release
    this.db = new Mojo.Depot(
        {name:"feedDB", version:1, estimatedSize: 100000, replace: false},
        this.loadFeedDbOpenOk.bind(this),
        function(result) {
            Mojo.Log.warn("Can't open feed database: ", result);
        }
    );
},
```

Depot's constructor first takes an object, which must include a `name`. This is a required property that must be unique for this depot. The `version` indicates the desired database version, but any version can be returned. The `estimatedSize` advises the system on the potential size of the database in bytes, and the `replace` property indicates that if a depot exists with this name, it should be opened. Should the `replace` property be set to true, an existing depot will be replaced. The `replace` property is optional; if missing, it defaults to `false`.

The `loadFeedDbOpenOk` callback will handle both the case where the feedlist had been previously saved and the case when a new database is created. The function literal is used if there is a database error. The cause of such a failure could be either that the database exists but failed to open, or that it didn't exist and failed to be created.

In `loadFeedDbOpenOk`, attempt to retrieve the data with a call to the `get()` method. The first argument is a `key`, which must match the key used when the data was saved. The other two arguments are the success and failure callbacks.

If the request was successful, the callback function receives a single argument, an object with the returned data. In the following example, the success callback is `loadFeedDbGetSuccess`, which first tests the returned object, `fl`, for null. If `fl` is a valid object, it is assigned to feedlist and the update cycle is started to refresh the feeds:

```
// loadFeedDbOpenOk - Callback for successful db request in setup. Get stored
//   db or fallback to using default list
loadFeedDbOpenOk: function() {
    Mojo.Log.info("Database opened OK");
    this.db.simpleGet("feedList", this.loadFeedDbGetSuccess.bind(this),
      this.loadFeedDbUseDefault.bind(this));
```

```
},

// loadFeedDbGetSuccess - successful retrieval of db. Call
//  loadFeedDbUseDefault if the feedlist empty or null or initiate an update
//  to the list by calling updateFeedList.
loadFeedDbGetSuccess: function(fl) {
    if (fl === null) {
        Mojo.Log.warn("Retrieved empty or null list from DB");
        this.loadFeedDbUseDefault();

    } else {
        Mojo.Log.info("Retrieved feedlist from DB");
        this.list = fl;

        // If update, then convert from older versions

        this.updateFeedList();
    }
},
```

Should `get()` fail to retrieve any data, the sample code assumes that this is because we're creating a new database rather than opening an existing one, so we pass `loadFeedDbUseDefault` as the second (failure case) callback. The default feedlist is assigned and, again, the feed update cycle is started with a call to `this.updateFeedList()`:

```
// loadFeedDbUseDefault() - Callback for failed DB retrieval meaning no list
loadFeedDbUseDefault: function() {
  // Couldn't get the list of feeds. Maybe its never been set up, so
  // initialize it here to the default list and then initiate an update
  // with this feed list
    Mojo.Log.warn("Database has no feed list. Will use default.");
    this.list = this.getDefaultList();
    this.updateFeedList();
},
```

If the call to open the database fails, there is a database error and the failure callback is used. In this sample, a function literal is used to log the error. There should be some proper error handling added to notify the user and advise on recovery actions, but that's not shown here.

The `storeFeedDb()` method is much less involved, but it is critical to keep the data updated. The webOS application model and user experience rely on saving data as it is entered or received, without explicit actions by the user. The News application has several points at which the data needs to be saved, and each time, calling this method will do it:

```
// storeFeedDb() - writes contents of Feeds.list array to feed database depot
storeFeedDb: function() {
    Mojo.Log.info("FeedList save started");
    this.db.add("feedList", this.list,
            function() {Mojo.Log.info("FeedList saved OK");},
            this.storeFeedDbFailure);
},
```

```
    // storeFeedDbFailure(transaction, result) - handles save failure, usually an
    //   out of memory error
    storeFeedDbFailure: function(result) {
        Mojo.Log.warn("Database save error: ", result);
    },
```

The `add()` method accepts the `"feedList"` string as the depot key and writes `this.list` as the stored object. The success callback is a function literal that logs the transaction, but the failure callback is another method. This is where you'd want to put recovery logic for memory full conditions, for example.

These calls can be implemented at these points in the `FeedListAssistant`:

`AddDialogAssistant`
: A new feed has been added, so clearly an update is required to save that new feed and its contents.

`showFeatureStory()`
: This might seem odd, but since this is a periodic updater, a flag can be set indicating that the data has changed (for example, from a feed update, or perhaps the unread status has changed); performing the update here implements a lazy update of the depot.

`cleanup()`
: As a precaution, save the most recent version in case something has slipped through during execution.

News never deletes its Depot object, but you can use the `discard()` method to remove objects if needed. To be safe, you should only use this method after a successful open or create transaction, and you may want to include success and failure callbacks as a further precaution. The framework will remove an application's Depot objects when the application is deleted from the device.

HTML 5 Storage

Clearly, the Cookie and Depot objects are simplistic, and while attractive for casual data storage and caching, they won't fulfill the need for formal database support. To address that need, Palm webOS includes support for the HTML 5 Database object to create, update, and query an SQL database. Like the Depot, the HTML 5 database interfaces are asynchronous, requiring you to use callbacks for much of your database handling.

Creating Large Databases

By default, Palm webOS stores databases within a size-constrained local file store, which is the reason that HTML 5 databases are restricted to a 1MB maximum. The bulk of the storage on the Palm Prē is in the mass storage partition or USB accessible storage, reserved for media use.

You can store your HTML 5 database or Depot in the mass storage partition, where it is not subject to a maximum size other than by the amount of available memory. To do that, specify the database name with a prefix, as in:

```
openDatabase( "ext:mydbname", ... );
```

The "ext:" will be mapped by the system to the mass media partition on the Palm Prē and an appropriate mapping on future devices.

The HTML 5 specification includes extensions for structured client-side storage, including support for the Storage and Database objects. Palm webOS does not support the Storage object, a list of name/value pairs that grew out of Firefox's DOM Storage object, but it does support the Database object.

The `openDatabase()` method will create a new database or open an existing database, returning a Database object:

```
this.db = openDatabase("myDB", 1, "My DB", 10000);
```

The arguments are as follows (see Table 6-1).

Table 6-1. openDatabase arguments

Property	Required	Description
Name	Required	Database name
`version`	Optional	Target version, or undefined if any version is acceptable
`displayName`	Optional	Application defined, not used by webOS
`estimatedSize`	Optional	Informs webOS of intended size to prompt for any system constraints at creation rather than during use

The database `version` property is meant to represent the schema version, enabling smooth migration of the database forward through schema changes. If the application specifies a database name and a version number, both need to match an existing database for the database open to succeed; otherwise, a new database will be created.

Once the database is open, you can execute transactions against it, using either `transaction()` for read/write transactions or `readTransaction()` for read-only transactions. The transaction methods will specify one to three callbacks:

- Transaction callback
- Error callback
- Success callback

The transaction callback is the most important; it includes the transaction steps that you wish to execute using an `executeSQL()` method, which accepts an SQL query string as an argument along with success and error callbacks.

For example, the following code segment calls the transaction method with a literal function that includes two `executeSQL()` methods, the last of which specifies a success callback, `this.successHandler()`, and an error callback, `this.errorHandler()`:

```
MyAssistant.prototype.activate = function()    {
          .
          .
          .
this.db.transaction( (function (transaction) {
            transaction.executeSql('A BUNCH OF SQL', []);
            transaction.executeSql('MORE SQL', [], this.successHandler.bind(this),
               this.errorHandler.bind(this));
     }).bind(this);
          .
          .
          .
MyAssistant.prototype.successHandler = function(transaction, SQLResultSet) {
// Post processing with results

};

MyAssistant.prototype.errorHandler = function(transaction, error)  {
    Mojo.Log.Error('An error occurred',error.message);

    // Handle errors here
};
```

The success handler is passed the transaction object plus an `SQLResultSet` object as an argument. The attributes of the results object are described in Table 6-2.

Table 6-2. SQLResultSet object

Attributes	Description
`insertID`	Row ID of the row that was inserted into the database, or the last of multiple rows, if any rows were inserted
`RowsAffected`	Number of rows affected by the SQL statement
`SQLResultSetRowList`	Rows returned from query, if any

We've only touched on the basics of the HTML 5 database capabilities in this section. As a draft standard, HTML 5 will continue to evolve. You should review the formal SQL reference at *http://dev.w3.org/html5/webstorage/#databases* for more in-depth information. There's also a basic demo application at *http://webkit.org/demos/sticky-notes/index.html*.

Ajax

In Chapter 3, we added an Ajax request to the News application, transforming the application from a static data reader to a dynamic application serving up new stories from multiple feeds in the background. There is a huge difference in user experience

when your application can let the user know that something new exists and present it to her immediately.

You aren't required to use the Prototype functions. You can use the XMLHttpRequest object directly, and you will be required to if your data protocols are SOAP-based or if they are anything other than simple XML, JSON, or text-based web services. There are many references for XMLHttpRequest if you'd like to explore this more directly. Any fundamental JavaScript reference will give you an overview, but for more in-depth information, look for Ajax-specific references such as Anthony Holdener's *Ajax: The Definitive Guide* (O'Reilly). For a more basic introduction, you can review the tutorial at *https://developer.mozilla.org/en/XMLHttpRequest*, which, while focused on Firefox, is nonetheless a good introduction to XMLHttpRequest.

The Ajax class functions, which are a basic feature of the Prototype JavaScript library, are included with webOS because they encapsulate the lifecycle of an XMLHttpRequest object and handlers into a few simple functions. The next few pages will explore these functions to show you how Prototype can help you integrate dynamic data into your application.

Palm webOS applications are run from *file://* URLs and thus aren't restricted by the single-origin policy that makes mixing services from different websites difficult.

Ajax Request

Back in Chapter 3, we added an `Ajax.Request` object to News to sync the web feeds to the device, but didn't describe the object or the return handling in any detail. `Ajax.Request` manages the complete Ajax lifecycle, with support for callbacks at various points to allow you to insert processing within the lifecycle where you need to. In Chapter 3, we used `updateFeedRequest` to initiate the Ajax request:

```
// updateFeedRequest - function called to setup and make a feed request
updateFeedRequest: function(currentFeed) {
    Mojo.Log.info("URL Request: ", currentFeed.url);

    var request = new Ajax.Request(currentFeed.url, {
        method: "get",
        evalJSON: "false",
        onSuccess: this.updateFeedSuccess.bind(this),
        onFailure: this.updateFeedFailure.bind(this)
    });
},
```

An `Ajax.Request` is created using the `new` operator, the only way that a request can be generated, and initiates an XMLHttpRequest as soon as it is created. In this case, a `get` request is initiated on the URL defined in `currentFeed.url`, and two callbacks are

defined for success or failure of the request. You can also make `post` requests and define other callbacks, which map to the request lifecycle shown in Table 6-3.

Table 6-3. Ajax.Request callbacks by lifecycle stage

Callback	Lifecycle stage
`onCreate`	Created
`onUninitialized`	Created
`onLoading`	Initialized
`onLoaded`	Request Sent
`onInteractive`	Response being received (per packet)
`on###, onSuccess, onFailure`	Response Received
`onComplete`	Response Received

To clarify:

`onCreate`
: Is available only to responders and is not available as a property of `Ajax.Request`.

`on###`
: When specified (where `###` is an HTTP status code, such as on 403), it is invoked in place of `onSuccess` in the case of a success status, or `onFailure` in the case of a failure status.

`onComplete`
: Is called only after the previous potential callback—either `onSuccess` or `onFailure` (if specified)—is called.

Ajax requests are asynchronous by default, one of the virtues of Ajax. While `Ajax.Request` supports an override, the synchronous `XMLHttpRequest` has been disabled in webOS. Since the UI and applications run as part of a common process, this is necessary to preserve UI responsiveness. It may be supported in a later release when there is concurrency support.

There are more `Ajax.Request` options available. `Ajax.Request` is covered thoroughly in most Prototype references, including the Prototype 1.6 API reference available at *http://www.prototypejs.org/api*.

Ajax Response

An `Ajax.Response` object is passed as the first argument to any callback. In our sample, we have used the HTTP status codes under the status property and the `responseText` and `responseXML` properties. Table 6-4 provides a summary of the full list of properties.

Table 6-4. Ajax.Response properties

Property	Type	Description
`readyState`	Integer	Current lifecycle state: 0 = Uninitialized 1 = Initialized 2 = Request Sent 3 = Interactive 4 = Complete
`status`	Integer	HTTP status code for the request
`statusText`	String	HTTP status text corresponding to the code
`responseText`	String	Text body of the response
`responseXML`	XML or Document	If content type is application/xml, then XML body of the response; null otherwise
`responseJSON`	Object	If content type is application/json, then JSON body of the response; null otherwise
`headerJSON`	Object	Sometimes JSON is returned in the X-JSON header instead of response text; if not, this property is null
`request`	Object	Original request object
`transport`	Object	Actual XMLHttpRequest object

We haven't discussed JSON specifically, but it is an increasingly important tool for developers. Prototype includes some powerful JSON tools in `Ajax.Request`, which supports automatic conversion of JSON data and headers. If you're handling structured data, you should look at JSON, particularly for data interchange.

More Ajax

Prototype includes some additional functions for consolidating listeners for Ajax callbacks, called responders, and for updating DOM elements directly from an Ajax request.

Ajax responders

If you're doing multiple Ajax requests, you might find it useful to set up responders for common callbacks rather than setting callbacks with each request. This is particularly applicable to error handlers or activity indicators. The following example shows the use of setup responders to manage a spinner during feed updates and to handle Ajax request failures:

```
Ajax.Responders.register({
   onCreate: function()  {
       spinnerModel.value = true;
       this.controller.modelChanged(spinnerModel, this);
   }.bind(this),
```

```
        onSuccess: function(response)  {
            spinnerModel.value = false;
            this.controller.modelChanged(spinnerModel, this);
            this.process.Update(response);
        }.bind(this),

        onFailure: function(response) {
            this.spinnerModel.value = false;
            this.controller.modelChanged(spinnerModel, this);
            var status = response.status;
            Mojo.Log.info("Invalid URL - Status returned: ", status);
            Mojo.Controller.errorDialog("Invalid feed - http failure: "+status);
        }.bind(this)
    });
```

In this sample code, each responder is defined using the callback property and a function literal. The first, `onCreate`, is available only to responders and not in an `Ajax.Request` object. It starts the spinner, while the other two, `onSuccess` and `onFailure`, stop it while performing some appropriate postprocessing.

Responders can be unregistered, but you would need to avoid using function literals when you register them, as the previous example did. The responders would need to be defined first, then registered and unregistered with a reference to that definition.

Ajax.Updater and Ajax.PeriodicalUpdater

`Ajax.Updater` and `Ajax.PeriodicalUpdater` each make an Ajax request and update the contents of a DOM container or element with the response text. `Ajax.Updater` will perform an update request once, while `Ajax.PeriodicalUpdater` will perform the requests repeatedly, with a delay option to extend the interval between updates when responses are unchanged. Note that `Ajax.PeriodicalUpdater` uses JavaScript timers and won't wake the device.

Summary

Dynamic data is an important part of any Palm webOS application to keep the user connected and in touch, while local data is critical for offline access and a responsive user experience. In this chapter, we've looked at both topics and used the Depot and Cookie objects, along with Prototype's Ajax functions and the HTML 5 database APIs. Managing your data in an efficient way is as fundamental to a great user experience as the powerful UI functions.

CHAPTER 7
Advanced Styles

Most of this book has been concerned with developing code, with some design topics added in. That isn't due to a lack of design sophistication in the platform or any limit to the opportunity to design beautiful, effective applications, but is mostly a matter of focus.

This chapter is all about styling your applications, a broad design topic. You will learn advanced type-styling techniques, with additional background on the use of type and text within your applications. Images will be deconstructed so that you can learn how to position and size them within a scene's layout, and integrate them with other content. A well-designed application will integrate touches and gestures reliably; you'll learn how to optimize your application to handle touches elegantly within your visual design.

This is just a very small sample of the range of the design capabilities of webOS and the opportunities it presents. There are extensive resources available to learn more about styling or various visual and interaction design topics included in the SDK:

Human Interface guidelines
: The SDK includes an extended set of guidelines for application designers on everything from designing a great webOS application to technical guidelines for graphic designers.

Palm webOS Visual Style Guide
: A thorough review of the webOS design philosophy and how to design visual elements for your applications.

Style Matters
: An interactive sample application available in the SDK for learning and applying styles on webOS.

In addition, there is the Quick Reference Style Guide (included as Appendix C in this book). This guide will give you more detail on the topics in this chapter as well as additional styling information.

Typography

You read about text widgets in Chapter 3, including the available styling options. Some of these options are also available with any text element that you include in your scene's HTML. Plus, you can use Mojo's standard text styles or override those styles within your CSS. In this section, you'll see how to apply type styles as truncation and capitalization functions, as well as some basic alignment techniques.

Fonts

Prelude is the primary font family for webOS. Its warm and welcoming appearance belies its underlying strength and readability. Prelude's designer, David Berlow of the Font Bureau, says of the typeface, "We wanted something that just disappears on the device, becoming such an integral part of the Palm webOS design, you don't notice."

You have a choice of typestyles, as shown in Figure 7-1, which are built into the framework so that you get them by default.

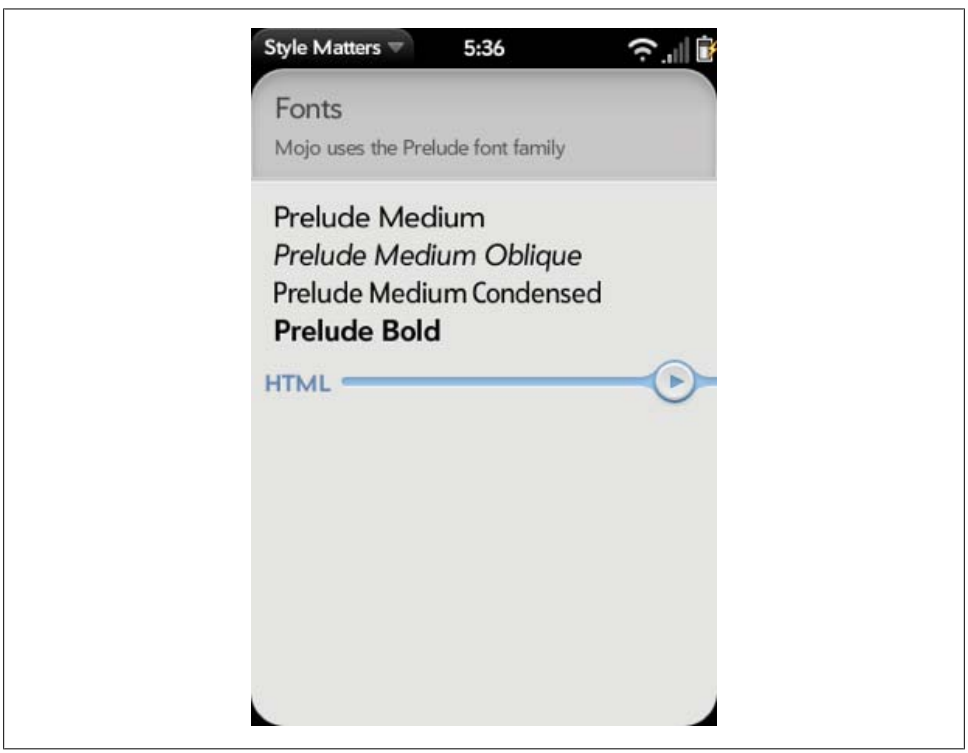


Figure 7-1. Prelude font family styles

Use the optional classes `condensed` or `oblique` with most text classes to modify the base class with a condensed or oblique font style.

Table 7-1 summarizes the available typestyles.

Table 7-1. Prelude typestyles

Typestyle	Technique
Prelude Medium	Provided with all text elements and classes by default
Prelude Medium Oblique	Add `oblique` class to any text element or class
Prelude Medium Condensed	Add `condensed` class to any text element or class
Prelude Medium Bold	Add <b> (bold) or tags around any string or element

Body text can be styled with a few basic text classes; some examples are shown in Figure 7-2.

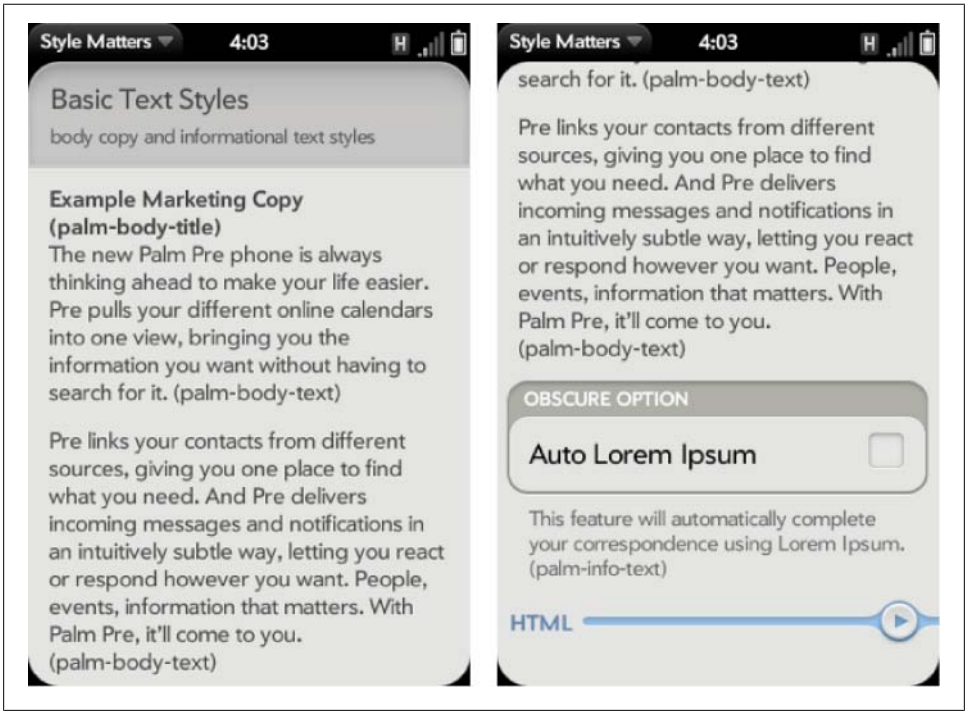


Figure 7-2. Text styles

This scene is created with HTML using the classes described in Table 7-2. The following excerpt shows the HTML for the title and text styles:

```
<div class="palm-text-wrapper">
    <div class="palm-body-title">
```

```
            Example Marketing Copy
        </div>

        <div class="palm-body-text">
            The new Palm Pre phone is always thinking ahead to make your life easier.
            Pre pulls your different online calendars into one view, bringing you the
            information you want without having to search for it.

        ...

        </div>
    </div>
```

Table 7-2. Text style classes

Class	Description
`palm-text-wrapper`	Use this wrapper to contain multiple divs of styled text for proper padding
`palm-body-title`	Title text
`palm-body-text`	Body text
`palm-info-text`	Caption text; commonly used with group boxes

Truncation

Text truncation is a standard feature of the Text Field widget, and an option for any HTML text element. You can add the `truncating-text` class to a conventional div element that contains a text string, and the string will be constrained to a single line and properly terminated with ellipses, as shown in Figure 7-3.

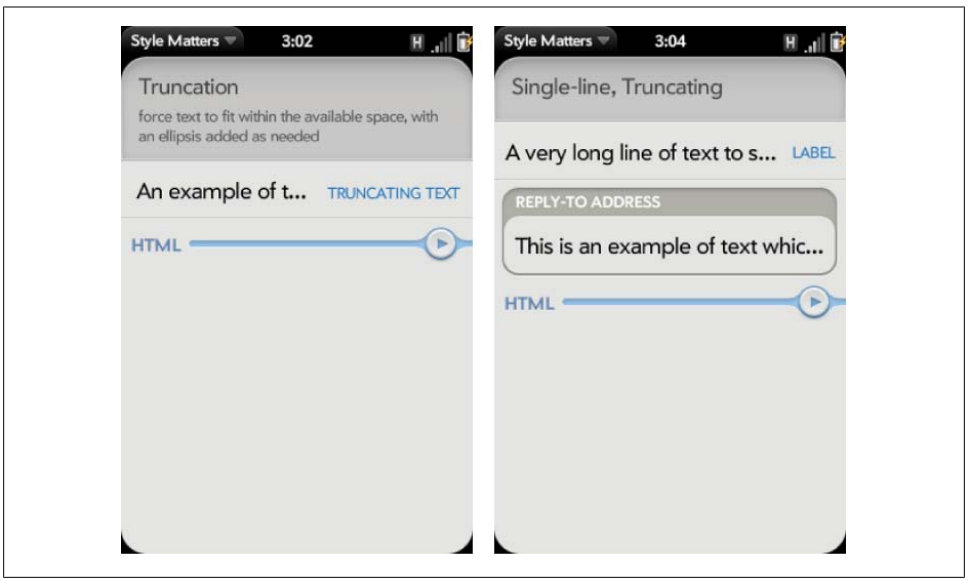


Figure 7-3. Text truncation

The leftmost example truncates a text string within a conventional div:

```
<div class="palm-list">
    <div class="palm-row first" x-mojo-touch-feedback="delayed">
        <div class="palm-row-wrapper">
            <div class="label">Truncating text</div>
            <div class="title truncating-text">
                An example of text
                which is so long you
                could not possibly fit
                it on a single line.
            </div>
        </div>
    </div>
</div>
```

In the righthand example, there are two text fields, each using single-line truncation, but within differently styled elements. The Text Field widget is composed of several elements, which have the `truncating-text` class assigned to them. You don't need to specify the class within your HTML; it's provided by default. If instead you want your text to wrap and your text field to grow vertically, set the `mulitline` property to `true`.

Capitalization

Some widgets and styles shift text strings to uppercase or apply capitalization. Specific framework styles will be capitalized by default. See Table 7-3 for a list of those styles.

Table 7-3. Classes with default capitalization

Class	Description
`capitalize`	Use to apply title-case capitalization to any text element
`palm-page-header`	Page header text element
`palm-dialog-title`	Error and alert dialog boxes by default, or when used in HTML
`palm-button`	Button label

Use the `un-capitalize` class to override autocapitalization in those styles.

The Text Field widget also performs capitalization by default, but it is controlled by the `textCase` property in the widget's attributes rather than through HTML class assignments. By default, `textCase` is set to `Mojo.Widget.steModeSentenceCase`, but you can set it to `Mojo.Widget.steModeTitleCase` for all caps, or to `Mojo.Widget.steModeLowerCase` to disable autocapitalization.

Vertical Alignment

There are a few techniques for vertically aligning text or elements within a div that you might find useful. For single lines of truncating text, set the text `line-height` equal to the div's height:

```
.single-line  {
  margin: 15px 0;
  padding: 0 15px;
  height: 50px;
  background: grey;
  line-height: 50px;
}
```

For multiple lines of text, specify equal amounts of padding to the div:

```
.multi-line {
  margin: 15px 0;
  padding: 15px;
  background: grey;
}
```

These two examples are shown in Figure 7-4; the top shaded box demonstrates the single-line alignment and the lower shaded box shows the multi-line alignment.

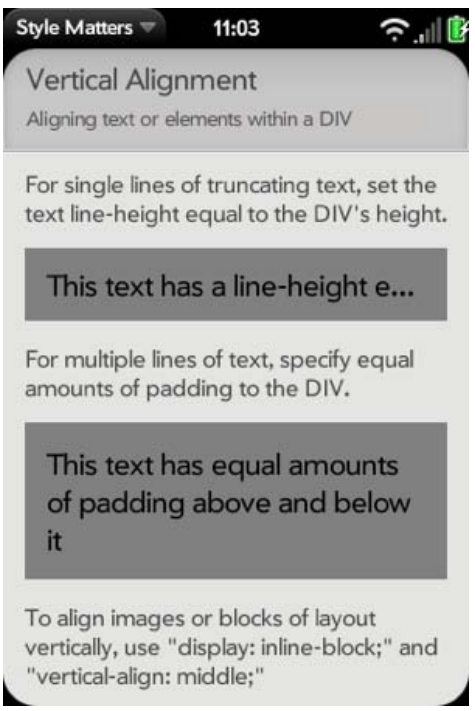


Figure 7-4. Vertical alignment examples

To align images or blocks of layout vertically, use `display: inline-block` and `vertical-align: middle` with your CSS rules for the specific div.

Images

This section will show you how to use images within your application, whether you want to reuse images provided with Mojo or create your own. There is a summary of the types of images provided with Mojo, and general guidelines on how to incorporate them into your application. You will need to refer to the SDK to see the individual images referenced here; there are too many to include here, and new ones are added with each SDK release.

You will also find design and technical guidelines here that will help you create your own custom images.

There are dozens of images provided with the Mojo framework. Look in the *framework/images* directory and you'll see a long list of PNG images, which are used as backgrounds, widget components, icons, and in various parts of the System UI. You are free to use any of the images in your application, but you should use them in a manner that is consistent with their use in the System UI or Palm applications.

A key UI principle is that consistency enhances ease of use. If your application uses visual images differently than what the user expects, your application will be perceived as harder to use.

Images are structured according to the intended use case:

Standard image
: A single image within a single file for conventional image use with `img` tags or similar cases.

Multistate image
: Multiple images within a single file used to combine multiple button states (e.g., pressed and unpressed states) or in a filmstrip animation sequence for activity indicators or similar cases.

9-tile image
: Using `webkit-border-image`, you can specify an image as the border of a div to create visually rich containers, buttons, dividers, and other dynamic images.

Standard Image

For any image, you should use a 24-bit per pixel RGB *png* with 8-bit transparency and 1-bit alpha channel whenever possible. Size the images according to how they will be rendered in the application. Image scaling is always a performance risk and will impact the user experience; avoid scaling images if you can.

Multistate Image

When displaying an animation or multistate button as the background of a div, combine your multiple states into a single image and change the background position to display the appropriate *frame* as desired. This negates the need to preload, eliminates flicker between the states, and conveniently keeps the assets together. Some examples of multistate images are shown in Figure 7-5.

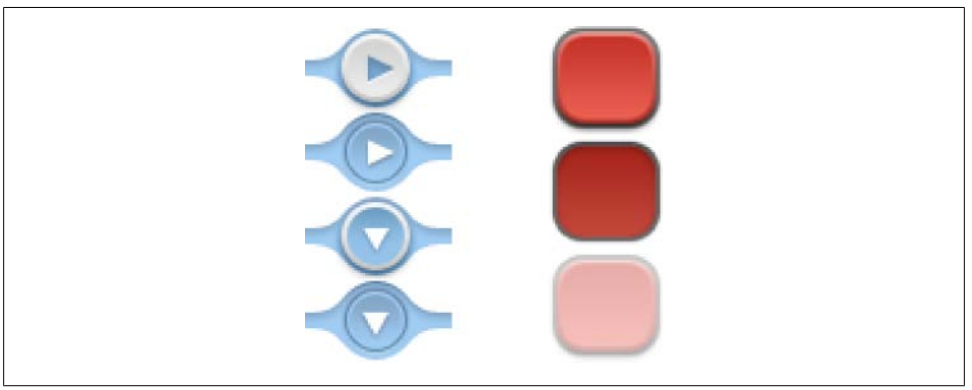


Figure 7-5. Multistate image examples

Multistate images are used for Push buttons, Menu buttons, Indicators, Toggle buttons, and Check Boxes, among other elements. It will be used anywhere you have a single image that needs to reflect state changes (e.g., unselected/highlighted/selected) or is part of an animation sequence.

9-Tile Image

Create single styles (with small, optimized images) that can accommodate variable length content and stretch horizontally to support any orientation or screen width using `webkit-border-image`. You would use this selector to divide an image into nine components (as shown in Figure 7-6) and use these components to render the border of the box. You can stretch or repeat the images to fill the space required with your image.

There are numerous examples of 9-tile images in the Mojo framework, including headers, borders, dividers, buttons, gradients, indicators, backgrounds, and icons. In the example shown in Figure 7-7, the `palm-group` is enclosed with a 9-tile image.

This particular image is handled within the framework with the following CSS:

```
.palm-group {
  margin: 7px 7px 0 7px;
  border-width: 40px 18px 18px 18px;
  -webkit-border-image: url(../images/palm-group.png) 40 18 18 18 repeat repeat;
}
```

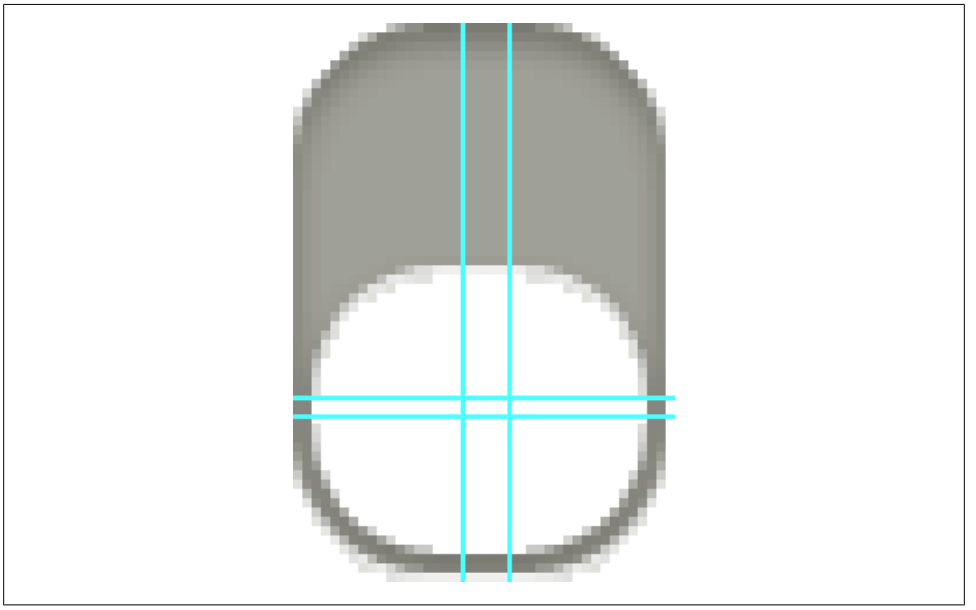


Figure 7-6. Parts of a 9-tile image

The image bounds are set as top (40px), right (18px), bottom (18px) and left (18px), followed by x- and y-transforms, usually `repeat` or `stretch`. It's important to set the `border-width` and the `webkit-border-image` bounds the same so that the image draws within the div bounds instead of outside them. `webkit-border-image` is based on the CSS 3 border-image standard, and you can find many resources for further information on this standard on the Web, but you can start with *www.css3.info/preview/border-image/*.

3-tile image

A 3-tile image is a 9-tile image with a zero border in one vector. You use a 3-tile image when you need an image that scales in one dimension only. Some examples are: radio button strips, dashboard containers, view menus, and page headers.

Create a 3-tile image by creating a 9-tile image and setting one dimension to 0, as shown below with the `palm-slider-background`:

```
.palm-slider-background {
  width: 250px;
  height: 7px;
  border-width: 0px 4px 0px 4px;
/** These next two lines are wrapped for book formatting only **/
  -webkit-border-image:
    url(../images/slider-background-3tile.png) 0 4 0 4 repeat repeat;
  margin: 6px 0px 0px 20px;
}
```

Figure 7-7. 9-tile image examples

Negative Margin

A div that functions as a container with a border image cannot use the space allocated to the border image for any content. The framework uses a technique called *negative margin* to reclaim that space so you can place content within the full width and height of the container.

The basic technique is to define a second div and place the content there instead of in the parent div that includes the border image. The child div has a wrapper class with a negative margin equal to the border width used in the parent div.

The framework uses this technique in numerous ways. One example is the submenu or list selector pop-up, where the container is defined using a `webkit-border-image` with a 24-pixel border:

```
.palm-popup-container {
  min-width: 180px;
  margin: 5px 0 0 0;
  padding: 0;
  z-index: 19950O;
  position: fixed;
  top: 80px;
  left: 20px;
```

```
    border-width: 24px;
    -webkit-box-sizing: border-box;
    -webkit-border-image: url(../images/palm-popup-background.png) 24 24
   24 24 stretch stretch;
  }
```

The border's width is reclaimed in the wrapper class to allow you to use the full width and height of the parent for content:

```
.palm-popup-wrapper {
  margin: -24px;
}
```

You use these styles in your HTML in this way:

```
<div class='palm-popup-container'>
  <div class='palm-popup-wrapper'>
    <!----- content is placed here  -------->
  </div>
</div>
```

Unsupported CSS properties

There are some properties common to Webkit that are not supported by Mojo, but there are some ways that you can work around those exclusions.

`webkit-border-radius`
: Instead of generating rounded corners dynamically, use `webkit-border-image` and specify an image with rounded corners.

`webkit-gradient`
: Instead of generating a gradient dynamically, create an image gradient and set it as the background of your body or div.

Touch

Since touch is the primary indicator of action, it is critical to style scenes optimally for `touchability`. Here are some strategies that you should consider for creating large, gapless hit targets.

Maximize Your Touch Targets

The elements in your scene may appear to be small and separate from each other, but their touch targets should be as large as possible. Touch targets in rows should be as tall as the row itself. You should maximize the size of the touch targets and there should be no gaps between targets.

Visual elements can be smaller than touch elements, so you might wrap the *image* div with a *touch* div. An example is the camera button, where the image is 80 × 60, but the div's width is set 20 pixels wider:

```
.capture-button {
  width: 80px;
  height: 80px;
  background: url(../images/menu-capture.png) top left no-repeat;
  position: absolute;
  left: 120px;
}
```

You can see in Figure 7-8 that there is no visible indication of the larger touch target.

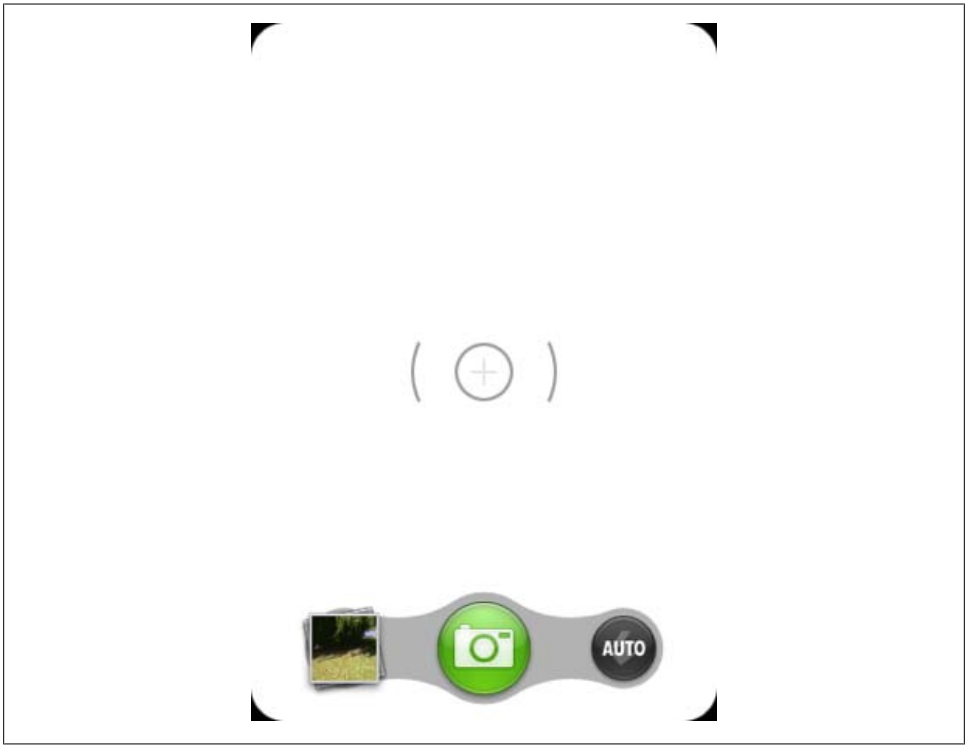


Figure 7-8. Camera button and touch target

Optimizing Touch Feedback

Use the `x-mojo-touch-feedback` attribute to make all touch targets reflect touches (in lieu of HTML focus attributes). Using conventional focus would result in highlights while dragging or scrolling items on the screen, while `x-mojo-touch-feedback` adds a delay on focus so that incidental touches won't cause a highlight. For example, in

the News application, we used a `momentary` tap highlight in *views/feedList/feedRowTemplate.html*:

```
<div class="palm-row" x-mojo-touch-feedback="delayed">
  <div class="palm-row-wrapper">
    <div class="icon right"><div class="unReadCount">#{numUnRead}</div></div>
    <div x-mojo-element="Spinner" class='feedSpinner' name='feedSpinner'</div>
    <div class="title truncating-text">#{title}</div>
  </div>
</div>
```

For items within scrollable content, as in the News feedlist, use `delayed` feedback. For fixed elements that don't scroll, `immediate` feedback is an option. Only use `immediatePersistent` or `delayedPersistent` if you require exacting control of when feedback is removed. To summarize, the values supported by `x-mojo-touch-feedback` are:

`immediate`
: Shows feedback immediately, stops showing it on finger up; for use with static items.

`delayed`
: Shows feedback after a short delay unless another gesture comes in to cancel the feedback; for use with scrollable items.

`immediatePersistent`
: Shows feedback immediately; feedback is not automatically cleared unless the user taps another item with `x-mojo-touch-feedback`; for use with static items.

`delayedPersistent`
: Shows feedback after a short delay unless another gesture comes in to cancel the feedback; feedback is not automatically cleared unless the user taps another item with `x-mojo-touch-feedback`; for use with scrollable items.

Passing Touches to the Target

In some cases, you might want to include an element that ignores touches, passing them through to a lower-level (in z-order) element. Mojo includes a custom CSS property:

```
-webkit-palm-target: ignore
```

This property will prevent an element from capturing touch events, allowing them to pass through to underlying elements.

Light and Dark Styles

If your application uses a light-colored background with dark text and controls, the default controls and text colors should work very well for you. If your application uses a dark-colored background with light text and controls, use the `palm-dark` controls and

text. Some of the applications that Palm ships on the phone use the `palm-dark` controls (e.g., Music and Videos). Figure 7-9 shows an example of `palm-light` and `palm-dark`.

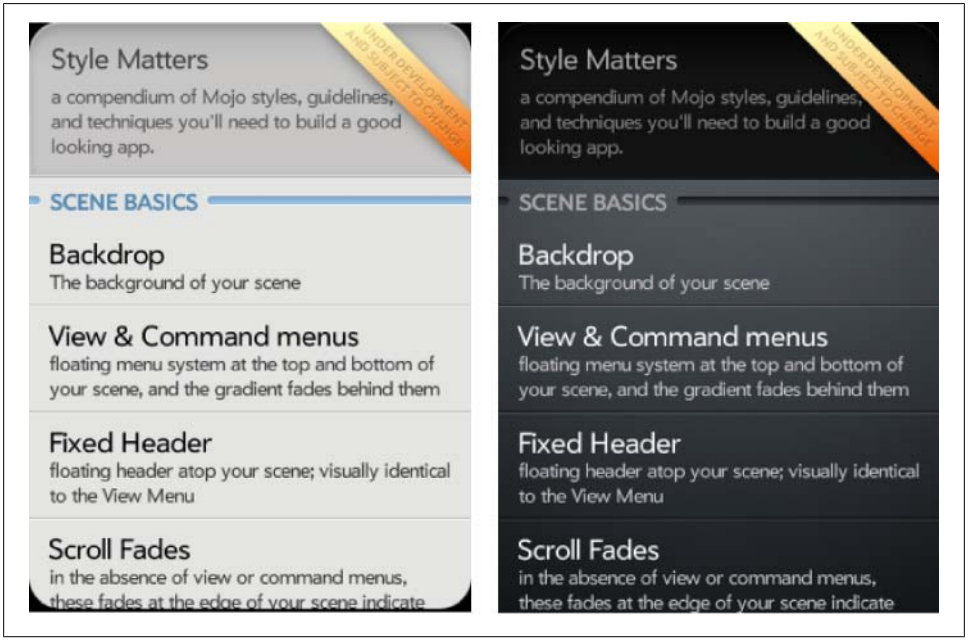


Figure 7-9. Light and dark styles

You can add the `palm-dark` class to your `body` element using JavaScript or add the `palm-dark` class to specific elements on the page (e.g., a Drawer widget). To change the `body` element, add the following within the main scene of your card stage:

```
var appController = Mojo.Controller.getAppController();
var stageController =  appController.getStageController(MainStageName);
var bodyElement = stageController.document.getElementsByTagName('body');

bodyElement[0].addClassName('my-dark-backdrop');
bodyElement[0].addClassName('palm-dark');
bodyElement[0].removeClassName('palm-default');
```

To constrain the style change to an individual scene, you can define an encompassing div with both `palm-scene` and `palm-dark` classes. You need to define the scene or body element with the class because submenus, dialog boxes, and other z-stacked elements will inherit the style from the scene.

If you don't use the dark styles, you can reduce your application load time by declaring that your application will use just the light styles. Do this by setting the theme property in your *appinfo.json* file:

```
"theme": "light"
```

Summary

This chapter highlights some of the widget and scene styles, and presents more information on using and creating images for use within custom styles or to override framework styles. Palm offers light and dark styles to distinguish the media applications from the productivity applications.

The Palm Mojo framework includes numerous styling options that you can use within your application. Many of these styles are provided automatically when you instantiate widgets or choose named palm style classes within your scenes. Refer to Appendix C for a complete list of available style classes and selectors, and for guidelines on how to apply them when working with widgets and scenes.

CHAPTER 8

Application Services

So far we've looked at UI elements and web service requests, but very little of what we've done is anything you couldn't do in a sufficiently capable web browser. What makes Mojo particularly powerful is its access to services that encapsulate both low-level hardware capabilities and higher-level data services that provide access to Palm webOS Synergy features, cloud services, and more. It is this access to the device and its services that lets developers build applications with the capabilities of a native application.

Most intriguing are the cloud-based services—emerging web services from many providers, including Palm—that provide limitless potential for added capability.

The Web as expressed through cloud service APIs is a platform in itself, and the webOS service architecture enables access to Palm cloud services. You can also access third-party cloud services or your own services through Ajax calls or other direct service interfaces, and build applications that integrate or mash up these services in unique ways.

In this chapter and the next, we'll explore a number of the available Mojo services. This chapter introduces the service architecture and presents the calling conventions that all Mojo services share. We will extend the News application to launch the web browser and to allow users to share stories over email or text messaging.

In Chapter 9, we conclude our discussion of the service layer by delving into cloud services and lower-level system services like Location services.

Using Services

All service calls are asynchronous operations. Each application service has a distinct service name, and exposes one or more named methods. Most application services will launch an application in its own card and will not return to the calling application. Device and cloud services will typically return some data or result to the calling application through a callback function defined by the calling application. Some services,

such as Location tracking, will return data in a series of calls to the callback function. You need to design your applications with this asynchronous interface in mind.

There are additional constraints when you are using services in background applications; these will be covered in Chapter 10. Background applications must moderate service use to conserve CPU and battery resources, plus with limited to no user interaction, the background application must handle service responses directly and use notifications and the dashboard to communicate with the user.

In many cases, the application should limit or stop service requests when minimized or in the background. For example, a game that takes accelerometer input should stop tracking the accelerometer when minimized. Without any visible display, there is no value in expending resources to collect that data.

Service Overview

Most services are Linux servers registered on the Palm bus, wrapped and accessed through the `Mojo.Service.Request` object. Application services are all accessed through a single service method, provided by the Application Manager, which routes the requests either implicitly based on resource or file type, or explicitly using the passed application ID. All other services are individually handled by the named service.

Use a `Mojo.Service.Request()` object for all service calls. For convenience, the `serviceRequest()` method is attached as a property to the scene controller, so a commonly used alternative is `this.controller.serviceRequest()`. The basic call includes a service name and `method`, with a method-specific `parameters` object:

```
this.controller.serviceRequest("palm://com.palm.serviceName", {
     method: "methodname"
     parameters: {},
     onSuccess: this.successHandler,
     onFailure: this.failureHandler
     }
);
```

Palm webOS uses the URI (Uniform Resource Identifier) scheme for identifying services, similar to the way a standard web URI is used. The service name is typically a string that begins with `palm://com.palm`, followed by a specific service name. The `method` defines the service method to use for this specific call, and `parameters` is a method-specific JSON object for passing arguments. For example, to get a GPS fix, make the following request in a scene assistant:

```
this.controller.serviceRequest("palm://com.palm.location", {
    method:"getCurrentPosition",
    parameters: {},
    onSuccess: this.onSuccessHandler,
    onFailure: this.onFailureHandler
    }
});
```

The string `palm://com.palm.location` is the service name and `getCurrentPosition` is the service method. There are optional parameters defined for this method, but this example is simply using the default settings. In general, parameters will vary from service method to service method.

Most service requests require callback functions to return results to the calling application. The `onSuccess` function is called if the service call is successful and may be called multiple times for service requests that result in a series of results, such as a request for Location tracking data instead of just a single fix. The `onFailure` function is called if the service call results in an error. Both callbacks include a single response object whose properties are service method dependent.

There are some conventions for response handling. All callbacks will be passed a single JSON object, and that will include some or all of the conventional properties described in Table 8-1, plus method-specific properties where appropriate.

Table 8-1. Response properties

Name	Description	Required
`returnValue`	`true` on success or `false` on failure of this request	Required
`errorCode`	The error code from the service when `returnValue` is false	Required
`errorText`	Description of the failure when `returnValue` is false	Required
`subscribed`	Set to `true` if a subscription request was successful	Optional

Mojo.Service.Request()

You can use `this.controller.serviceRequest()` within scenes where you would make most service requests. However, if you need to make a service request within your application assistant, you'll need to create a service request object.

A common case is a call to the Alarm service to wake up the application after an interval:

```
this.alarm = new Mojo.Service.Request("palm://com.palm.power/timeout", {
    method: "set",
    parameters: {
        key: "com.palm.app.news.update",
        in: feedUpdateInterval,
        uri: "palm://com.palm.applicationManager/open",
        params: {
            id: "com.palm.app.news",
            params: {action: "updateFeed"}
        }
    },
    onSuccess: this.onSuccessHandler,
    onFailure: this.onFailureHandler
});
```

In this example, the new request object is created and a service request issued, with the object stored as `this.alarm`.

The request references are managed by the scene when creating a service request using `this.controller.serviceRequest()`, and are removed upon completion of the request, unless the request has `subscribe: true`, in which case the requests are cleaned up when the scene is popped.

All requests made with `this.controller.serviceRequest()` are cleaned up when the scene is popped, meaning they are garbage collected and destroyed. If the subscription request needs to be retained beyond the lifetime of the scene, you will also need to use `Mojo.Service.Request()` to save the request object and manage the request yourself.

Remember that service requests are asynchronous, so they don't complete when you make the call; if there's a chance they will not be completed by the time the scene is popped, use `Mojo.Service.Request()`.

Application Manager

The Application Manager is a specific service that provides functions related to finding and launching applications. Applications launched through the Application Manager will open and maximize a new window for the targeted application while minimizing the current application window.

The Application Manager, through one or both of its service methods, provides access to most of the application services:

`open`
: Accepts a single argument, a formatted URI for a document you wish to display. The mime type of the referenced document is used to identify the appropriate application to handle the content indicated.

`launch`
: Launches the application indicated by the application ID argument passing any included parameters.

Open

Generally, the `open` method is used when you intend to display or process some targeted content, but you don't know the specific type of content or the best application available in the system to handle it. The Application Manager will use the content type to find the appropriate application to use for that content:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
     method:     "open",
     parameters: {
          target: "http://www.irs.gov/pub/irs-pdf/fw4.pdf"
     },
     onFailure: this.onFailureHandler
});
```

The target includes a command, the string up to and including the colon and forward slashes (://), and the resource. In this example, the command is `http://`, but before launching the browser, the Application Manager will retrieve the HTTP header and attempt to extract the resource type. Since the URI specifies a file target, the Application Manager will try to match the file type to the resource list and find a match with `com.palm.app.pdfviewer`. The file will be downloaded to `/media/internal` and the PDF View application will be launched with a file reference to the downloaded file.

If there's no header, the Application Manager will download the file anyway and try to match the file extension in the resource list. If there is a match, the associated application will be launched with the file reference as a launch parameter. If there's no match at this point, the Application Manager will exit and return an error code to the failure callback.

The same process is used when streaming audio or video formats, but instead of downloading the content and then launching the application, the launch is done first and the content URI is passed as an argument. The audio or video player handles the connection and data streaming in these cases; the data is never actually stored on device.

If the command is `file://`, it's a local file reference and the Application Manager will use the file extension to launch the associated application (if there is one).

Launch

There are many cases in which you already know which application you'd like to handle the request, and it's inconvenient to force the parameters into a URI format. In these cases, you'd want a command to launch a specific application and pass in parameters in the form that the application has specified. Here is an example of launching the Maps application to show a street address:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
     method:     "launch",
     parameters: {
         id: "com.palm.app.maps",
         params: {
             query: "950 W. Maude Ave, Sunnyvale,CA"}
         }
     }
});
```

The Command and Resource Table in Appendix B includes the full list of all supported content types and the application resource handlers. This list is very likely to change, so refer to the Palm Developer site for the most current information.

Cross-App Launch

In some cases, a Cross-App launch is used to keep the context of the calling application, but with a faster transition. The target application's scene is pushed directly in the current application's card stage, and when the target application is popped, it returns

results as arguments to the calling application's `activate()` method. You can learn more about the Cross-App launch by referring to the Camera and People Pickers, both of which use this technique and are covered later in this chapter.

Core Application Services

This first group of application services includes a core set of applications that provide basic functions for web browsing, phone calls, maps, camera, and photos. The browser will be used first, with an example using the News application, followed by brief descriptions of the other applications and how you call them from within your application.

Web

Earlier, a WebView widget was used to display the source URL for the News story, but launching the web browser application in a new card gives more flexibility to News. The browser application can be launched to its default launch view or to a specific URL.

Back to the News: Launching the browser

News will launch the browser to load a specified URL as a simple example of an application service. The WebView widget in News is replaced with a call to the Application Manager to launch the browser into a separate card. The command handler for `do-webStory` in the *storyView-assistant.js* is replaced with a new version that includes a single call to `this.controller.serviceRequest`, with the service name set to the Application Manager, or `palm://com.palm.applicationManager`. All Application Manager calls will start this way.

The second argument is an object literal that includes an `open` method and a `parameters` object that includes the application `id` property set to `com.palm.app.browser`, the browser's application ID, and a `params` object. The `params` object includes just the `target` URL retrieved from the story array entry. This is typical of an Application Manager `open` call and is used with most applications that can accept a URL parameter:

```
case "do-webStory":
    this.controller.serviceRequest("palm://com.palm.applicationManager", {
           method: "open",
           parameters: {
               id: "com.palm.app.browser",
               params: {
                   target: this.storyFeed.stories[this.storyIndex].url
               }
           }
    });
```

This change means that when the user taps a new Command menu button in the `storyView` scene, the browser will launch in a separate card with the contents of the story's originating URL displayed. You can find the code sample for the Command menu button changes in the section "Email and Messaging" on page 205.

This also eliminates the `storyWeb` scene, so you can remove the assistant and views from the News project and from *sources.json*.

Phone

The user must tap the dial button to approve any phone call that is placed. Your application can initiate the phone call by opening the Phone application, providing a dial string (as shown in Figure 8-1). The phone will be launched to the dial scene, with or without the dial string included:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
    method: "open",
    parameters: {
       target: "tel://4085556666"
       }
    }
});
```

Figure 8-1. Phone application launched with prepopulated number

Camera

From within your application, you can turn on the camera and present a simple interface to take pictures, with an option to save or delete the picture after it is captured. When called from within another application, the camera application will only take a single picture and will return a file reference to the calling application if the picture was saved.

You must use a Cross-App launch to call the Camera from your application. This requires that you call the `pushScene()` method just as with any scene push, but include scene arguments that indicate an application launch is required:

```
this.someAssistant.stageController.pushScene(
    { appId : "com.palm.app.camera", name: "capture" },
    { sublaunch : true }
);
```

When the picture is taken or canceled, control will be returned back to your scene with a call to the scene's `activate()` method, just as with any scene pop. However, unlike the typical scene lifecycle, there will be a response object passed as an argument to the `activate()` method:

```
CameraAssistant.prototype.activate = function(response){
    if (response)   {
        if (response.returnValue)  {
            this.showDialogBox("Picture Taken", response.filename);
        } else  {
            this.showDialogBox("No Picture", "");
        }
    } else {
         Mojo.Log.info("Picture not requested");
    }
};
```

Photos

The Photos application is limited to launching the application to the default view, where the user can choose between various albums and photos. All images stored on the device will be indexed and viewed this way:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
    method: "launch",
    parameters: {
        id:"com.palm.app.photos",
        params: {}
        }
    }
});
```

Maps

You can use the Maps application to display maps around specific locations defined by street address, latitude/longitude, or through a location query. The map can optionally include driving directions or additional local or business search results, and there is a choice of map type and zoom level:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
    method: "launch",
    parameters: {
    id: "com.palm.app.maps",
    params: {
        location: {lat: 37.759568992305134, lng: -122.39842414855957, acc: 1},
        query: "Pizza",
        }
    }
});
```

This example launches the Map application to show the pizza options around a section of San Francisco with an accuracy of a meter. This is very powerful when used with the Location service, which will be covered in the next chapter.

Palm Synergy Services

Palm Synergy integrates personal information from various sources on the Web and presents it in a single view so that users can see all in one place. Yet the information is maintained in such a way that users can keep things separate when they have to. The integration is at the visual or presentation layer, while the separation is maintained at the data layer.

The core Synergy applications are Contacts, Calendar, Email, and Messaging, but the concept is general enough that you can expect other applications to be supported over time. All Synergy applications can be launched through the Application Manager service, and Email and Messaging can be used to send messages with the user's approval, similar to the way the Phone application is used.

The Contacts and Calendar application service interfaces can do a bit more, allowing applications to add contacts or calendar events, distinguished by their own data sources. These features are designed for occasional use, serving the needs of applications that want to add single records rather than fully scaled sync solutions.

Account Manager

All Synergy applications require an established account before any other operations can take place. There is an implicit "Palm" account that all information created and stored on the device belongs to, but any other information must be provided by an application with an explicit account ID. The account determines all access permissions;

data belonging to an account can only be accessed by the application that owns that account.

The Account Manager includes methods to create, update, delete, or read accounts, as well as a method to list accounts. Here is an account create example:

```
this.controller.serviceRequest("palm://com.palm.accounts/crud", {
     method: "createAccount",
     parameters: {
         account:  {
            username: "myusername",
            domain: "mydomain",
            displayName: "My Name",
            icons: {
              "32x32": Mojo.appPath + "images/accountIcon.png",
              "48x48": Mojo.appPath + "images/stampIcon.png"
            },
            dataTypes: ["CONTACTS", "CALENDAR"],
            isDataReadOnly: false
         }
     },
     onSuccess: this.accountCreated.bind(this);
     onFailure: function(response)  {
         Mojo.Log.info("Account create failed; ", response.errorText}
     }
});
```

This method uses the service name `palm://com.palm.accounts/crud`, and has parameters that specify account properties. The `dataTypes` object declares the Synergy data sets used by the account: `CONTACTS` and/or `CALENDAR`. The `domain` property allows a single account to have different data sources within it. Domains are useful in limited cases, such as when a single application wants to maintain multiple sync sources.

Accounts may be a lot of overhead if your application has only an occasional need to add a contact or calendar event. In that case, you might choose to simply launch Contacts or Calendar and have the user enter the data directly. Plus both Contacts and Calendar support launch points to add or update individual records without creating a new account.

The `accountId` is used for subsequent Account Manager methods, and for access methods in Contacts and Calendar. It's also used in the other Account Manager methods.

The `listAccounts` method will list only accounts that belong to your application. It cannot retrieve information about accounts belonging to other applications.

Contacts and Calendar

Both Contacts and Calendar will allow applications to add information that will be merged into an integrated view. They don't allow applications to read, delete, or update any data that wasn't created by the same application.

The Contacts application has methods to create, read, update, and delete contacts, along with listing all contacts. In addition, there are methods to track changes to contacts to support applications that wish to optimize updating their data sources with changes made by the user on the device.

The following is an example of creating a contact entry:

```
this.controller.serviceRequest("palm://com.palm.contacts/crud", {
      method: "createContact",
      parameters: {
           accountId: this.accountId,
           contact: {
               firstName: "Harry",
               lastName: "Truman",
               companyName: "US Government",
               nickname: "POTUS 33"
            }
       },
       onSuccess: this.successEvent.bind(this),
       onFailure: this.failureHandler.bind(this)
});
```

Notice the use of `this.accountId`. This is returned when you create the account and is used for most of these Contacts or Calendar functions. The `contact` object defines the contents of the contact entry, and has many more property options than the few that are shown in the example.

Calendar requires that a new calendar is first created, and then entries for that calendar are created within it. This example creates a calendar in the first function and then, on success, creates an entry using the current date and time:

```
CalendarAssistant.prototype.createCalendar = function() {

        this.currentMethod = "Calendar - Create";
        this.controller.serviceRequest('palm://com.palm.calendar/crud', {
             method: 'createCalendar',
             parameters: {
                accountId: this.accountId,
                calendar: {
                    calendarId: "",
                    name: "My Events"
                }
            },
            onSuccess: this.createEvent.bind(this),
            onFailure: this.failureHandler.bind(this)
      });
};
```

```
CalendarAssistant.prototype.createEvent = function(response) {
        if (response)   {
            Mojo.Log.info("Calendar Create ", Object.toJSON(response));
            this.calendarId = response.calendarId;
        }
        this.currentMethod = "Event - Create";
        var currentTime = new Date();
        var startTime = currentTime.getTime();
        this.controller.serviceRequest('palm://com.palm.calendar/crud', {
             method: 'createEvent',
             parameters: {
                calendarId: this.calendarId,
                event: {
                   calendarId: this.calendarId,
                   subject: "Forecast",
                   startTimestamp: startTime,
                   endTimestamp: startTime + 3600000,
                   allDay: false,
                   note: "Cliff Notes",
                   location: "Bluff",
                   attendees: [],
                   alarm: "none"
                }
            },
            onSuccess: this.successEvent.bind(this),
            onFailure: this.failureHandler.bind(this)
      });
};
```

As with Contacts, there are methods to track changes to calendar events or event deletions.

People Picker

The People Picker is a special Contacts function that lets applications retrieve information from any Contacts entry. It won't allow direct access, but it will allow the user to select a specific contact and approve the transfer of that contact's details to the requesting application.

The People Picker is called through a Cross-App launch. As mentioned earlier, this technique pushes a scene from another application on your application's scene stack. This keeps your application context, meaning that the user won't see any card switch. To call the People Picker, use `pushScene()`:

```
PeoplePickerAssistant.prototype.getContact = function(event){
    this.contactRequest = true;
    this.controller.stageController.pushScene(
            { appId : "com.palm.app.contacts", name: "list" },
            { mode: "picker", message: "headerMessage" }
        );
};
```

People Picker presents the Contacts list scene with the filter field activated. Just as in the Contact's details scene, typing will filter down the list and eventually the user would either select a contact or cancel with a back gesture. After the user selects a contact, the details are returned as an argument to calling the scene's `activate` method:

```
PeoplePickerAssistant.prototype.activate = function(response){
    if (response)   {
        if (response.personId)  {
            this.showDialogBox("Contact Received", response.personId);
        } else  {
            this.showDialogBox("Contact Request Failed", "");
        }
    } else {
         Mojo.Log.info("No Contact Requested");
    }

};
```

An application might use this to get a contact's address for use in planning a trip, for example, or an IM address, or some other personal information; contact details are returned in `response.details`. You can optionally exclude contacts from the list by enumerating their contact IDs.

Email and Messaging

You can launch the Email and Messaging applications to their main views using the Application Manager launch method, but most applications will generally use these applications to send a message. For that, you will use the launch method to launch either of these applications to a Compose view and optionally populate some or all of the compose fields.

Back to the News: Sharing stories through email or messaging

News will be extended to share a story by either email or messaging to illustrate these service calls. This is best hooked into the `storyView` scene, but we'll start with the service calls before looking at how they are integrated into the scene.

Email is called with a prepopulated subject field using the `params.summary` property, and the shared URL in the message body, using `params.text`. You can also include one or more email addresses for any of the address fields (To, CC, and BCC), but in this example, the user would need to address the mail using the addressing widget in the email application:

```
// shareHandler - choose function for share submenu
StoryViewAssistant.prototype.shareHandler = function(command) {
        switch(command) {
            case "do-emailStory":
                this.controller.serviceRequest("palm://com.palm.applicationManager", {
                    method: "open",
                    parameters:  {
                        id: "com.palm.app.email",
```

```
                        params: {
                            summary: "Check out this News story...",
                            text: this.storyFeed.stories[this.storyIndex].url
                        }
                    }
                });
                break;
```

Messaging is very similar; for this example, the entire message is provided in `params.messageText`:

```
    case "do-messageStory":
                this.controller.serviceRequest("palm://com.palm.applicationManager", {
                    method: "open",
                    parameters: {
                        id: "com.palm.app.messaging",
                        params: {
            // ** These next two lines are wrapped for book formatting only **
                            messageText: "Check this out: "
                              +this.storyFeed.stories[this.storyIndex].url
                        }
                    }
                });
                break;
        }
    };
```

As with email, you can specify the recipient(s) as part of the call. See Appendix B for a complete list of the calling arguments.

To hook these calls into the scene, add a Command menu button to the bottom of the `storyView` scene, which creates a scene like that shown in Figure 8-2.

This sample code is used to generate the view shown in Figure 8-2:

```
// setup - set up menus
StoryViewAssistant.prototype.setup = function() {
    this.storyMenuModel = {
        items: [
            {iconPath: "images/url-icon.png", command: "do-webStory"},
            {},
            {items: []},
            {},
            {icon: "send", command: "do-shareStory"}
        ]};

        if (this.storyIndex > 0)    {
           this.storyMenuModel.items[2].items.push({
             icon: "back",
             command: "do-viewPrevious"
           });
        } else {
            this.storyMenuModel.items[2].items.push({
              icon: "", command: "",
              label: "  "
            });
```

```
    }

    if (this.storyIndex < this.storyFeed.stories.length-1)    {
        this.storyMenuModel.items[2].items.push({
          icon: "forward",
          command: "do-viewNext"
        });
    } else {
        this.storyMenuModel.items[2].items.push({
          icon: "",
          command: "", label: "  "
        });
    }

this.controller.setupWidget(Mojo.Menu.commandMenu, undefined, this.storyMenuModel);
```

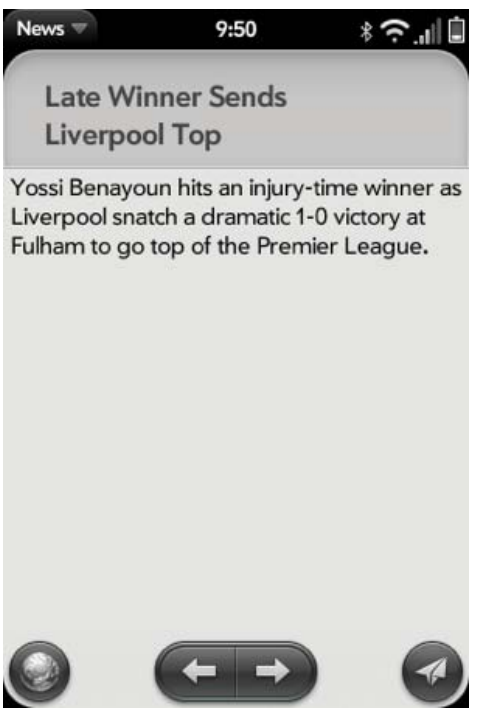


Figure 8-2. Story View with menu buttons for browser view and sharing

The Share button is the rightmost button and will present a pop-up when tapped. The Next/Previous button group in the center was covered in Chapter 4; they are used to navigate to the next and previous stories.

There is already a command handler included in this scene, so just add handlers for the new button and add a submenu to present the email and messaging options:

```
// -------------------------------------------------------------------------
// Handlers to go to next and previous stories, display web view
// or share via messaging or email.
StoryViewAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case "do-viewNext":
                Mojo.Controller.stageController.swapScene(
                    {
                        transition: Mojo.Transition.crossFade,
                        name: "storyView"
                    },
                    this.storyFeed, this.storyIndex+1);
                break;
            case "do-viewPrevious":
                Mojo.Controller.stageController.swapScene(
                    {
                        transition: Mojo.Transition.crossFade,
                        name: "storyView"
                    },
                    this.storyFeed, this.storyIndex-1);
                break;
            case "do-shareStory":
                var myEvent = event;
                var findPlace = myEvent.originalEvent.target;
                this.controller.popupSubmenu({
                    onChoose:  this.shareHandler,
                    placeNear: findPlace,
                    items: [
                        {label: "Email", command: "do-emailStory"},
                        {label: "SMS/IM", command: "do-messageStory"}
                        ]
                    });
                break;
             case "do-webStory":
                this.controller.serviceRequest("palm://com.palm.applicationManager", {
                       method: "open",
                       parameters: {
                           id: "com.palm.app.browser",
                           params: {
                               scene: "page",
                               target: this.storyFeed.stories[this.storyIndex].url
                           }
                       }
                });
                break;
          }
    }
};
```

The browser launch is handled directly in the command handler above. The service calls for email and messaging are in `shareHandler`, the submenu's callback, which is where this section started.

Viewers and Players

The media applications can be used to play streaming or file-based audio or video content. You can use the Application Manager's `open` method to handle common file types.

View File

There is no specific view file service; it's just the general case of using the Application Manager's `open` method, where the content target is unknown. As shown in the section "Application Manager" on page 196, simply call the Application Manager with a target value that refers to either web-based or file-based content:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
     method: "open",
     parameters: {
          target: "http:// crypto.stanford.edu/DRM2002/darknet5.doc"
     },
     onFailure: this.onFailureHandler
});
```

Any supported file type will be passed to the appropriate application for viewing, editing, or other supported handling.

Audio

The Music player is used to play or stream a file or other web-based content encoded in any supported audio format. Launch the Music player with the Application Manager's `open` method and a `target` property in the form `rtsp://`*`audio-file`*, where *`audio-file`* is a well-formed URI targeting a file encoded in a supported audio format. The target property can also point to a locally stored file, as shown in this example:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
    method: "open",
    parameters: {
        target: "file:///media/internal/World.mp3"
    }
});
```

Refer to the Command and Resource Handler table in Appendix B, which has a complete list of all supported audio file and mime types.

Video

The Video player is used to play or stream video content. Like the audio player, it can just be invoked through the Application Manager's `open` method and a `target` property in the form `rtsp://video-file`, where *`video-file`* is a well-formed URI targeting a file encoded in a supported video format.

There are some additional features when using the `launch` method, where you can specify a title or a thumbnail that is displayed while the video is loading:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
    method: "launch",
    parameters: {
        id: "com.palm.app.videoplayer",
        params:{
            target: "file: ///media/internal/Guitar.mp4",
            videoTitle: "Old Guitar"
        }
    }
});
```

Refer to the Command and Resource Handler table in Appendix B, which has a complete list of all supported video file and mime types.

Other Applications

The Application Manager service can launch any application, not just the core applications described in this chapter. However, it's limited at this time; to launch another application, you will need to know the application ID and the available parameters. Currently, webOS does not include dynamic registration for resource handlers or any broadcast services to allow you to determine which applications are available and which services they offer at runtime.

You can launch News in its current form with this call:

```
this.controller.serviceRequest("palm://com.palm.applicationManager", {
     method: "open",
     parameters: {
         id: "com.palm.app.news",
         params: {}
     }
});
```

News is launched as if from the Launcher to the `feedList` scene. If it is already launched, it will be maximized and put into the foreground view.

With the addition of an application assistant, an application is able to accept launch arguments through an explicit entry point, the `handleLaunch` method. Chapter 10 covers these topics in detail and explores the general use of launch requests.

Summary

Services extend the framework with access to the core applications, hardware-enabled features, and cloud services. In this chapter, the application services were described, including core applications, the Synergy applications, and the media players. Application services are mostly accessed through the Application Manager, a general command and resource handling service. System and cloud services will be covered in the next chapter.

The service architecture is accessed through `Mojo.Service.Request()`, which accepts a service name and a method name to route the request; service requests are always asynchronous operations.

CHAPTER 9
System and Cloud Services

System services are those that are enabled by hardware features or provided by the Linux OS. Hardware-enabled services include access to accelerometer data, location services, and connection status. The OS provides alarms, sounds, power management, properties, and time services.

As described in Chapter 8, the Mojo framework provides access to the system services, routing requests to the specified services and calling the application's callback functions with the service response. As shown in Figure 9-1, all system services are actually managed by Linux-resident server processes. The servers receive service requests from the application and send messages back. The messages are routed to the application through the specified callback functions, whether fulfilling the request or providing a failure indication.

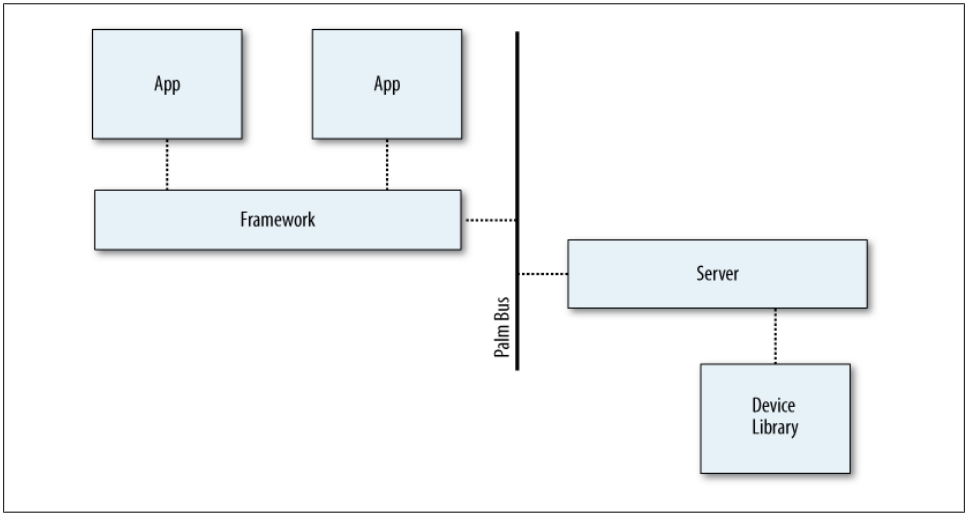


Figure 9-1. High-level service architecture

Cloud services are a form of web services. The initial cloud service is Mojo Messaging, an Extensible Messaging and Presence Prototocol (XMPP)-based messaging service for publish/subscribe notifications. It allows applications to send or receive notifications through the cloud to other collaborating clients and services. Over time there will be other cloud services that applications can leverage, extending the webOS platform further into the cloud.

System Services

This section describes each of the system services. Very few of the services apply to the News application, so most of the code samples in this section are simple ones to illustrate the service calls and handlers.

The system services are accessed through `Mojo.Service.Request()`, or the equivalent scene controller method `this.controller.serviceRequest()`:

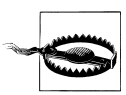
```
Mojo.Service.Request("palm://serviceName", options, requestOptions);
```

The arguments are described in Chapter 8, but briefly:

`serviceName`
: Is a URI-formatted string that uniquely specifies the service name.

`options`
: Includes the designated method, parameters, and callback functions.

`requestOptions`
: Is either set to true, requesting automatic subscription renewal on error, or an object with a `resubscribe` property and/or a `useNativeParser` property.



Be careful to prevent garbage collection of your service requests. Recall from Chapter 8 that requests made with `this.controller.serviceRequest()` are garbage-collected and destroyed when the scene is popped. You should use `Mojo.Service.Request()` to save the request object and handle termination of the request yourself to prevent garbage collection.

Service requests do not complete when you make the call; they are all asynchronous. If it's possible that the life of your request will outlast the life of the assistant when the call is made, you must use `Mojo.Service.Request()`.

Service Subscriptions

Most system services offer *subscriptions*, which provide the option of receiving service notifications each time there is an update. Set the `subscribe` property to `true` to register for subscription notification. For each update, subscribed requests will return to the specified `onSuccess` callback with a `subscribed` property set to `true`. For example:

```
this.controller.serviceRequest("palm://com.palm.connectionmanager", {
    method: "getStatus",
```

```
        parameters: {
           subscribe: true
        },
        onSuccess: this.handleResponse
    });
```

Subscriptions are typically used for:

- Registering for changes in status, such as with the Connection Manager, where a change from WiFi connection active to inactive would trigger a notification to any subscribed application.
- Receiving successive responses, such as with Location tracking, where each tracking fix is provided in a separate notification.

Be aware that all services will respond with an immediate callback to acknowledge the subscription registration request. In some cases, it will be the second callback that includes a response to the service request.

You can also request to have a subscription automatically renewed in the event the service is restarted or some other type of error occurs. Set the argument `requestOptions` to `true` to enable this renewal feature.

If the requested service fails (e.g., the location service is unavailable), the `onFailure` handler will be called. Periodically (at random intervals), the framework will attempt to reissue the command until the service comes back up. The `onSuccess` will be called only after the service has been restored.

Accelerometer

Applications can respond to high-level orientation, and shake events or elect to receive raw accelerometer data through acceleration events. You will use framework controller functions or event listeners to receive accelerometer events.

You can use the orientation events to rotate your application to track the orientation of the device between portrait and landscape modes. Shake events can be applied in creative ways as alternatives for user input, such as start/stop indicators. The raw acceleration data is useful for games or other applications that can integrate device movement directly into the application's interaction with the user.

Orientation changes

It's extremely simple to have your application respond to changes in screen orientation. Within your stage assistant or main scene assistant, set your application's card stage window to a *free* orientation through the stage controller's `setWindowOrientation()` method:

```
if (this.controller.stageController.setWindowOrientation) {
 this.controller.stageController.setWindowOrientation("free");
}
```

The system will rotate the stage window, following the orientation of the device between `up` (normal portrait), `right` (clockwise rotation from up), `down` (portrait, but the reverse of up), or `left` (counter-clockwise from up). You can use the same method to force an orientation with any of up, down, left, or right passed as the argument string.

There is a corresponding `getWindowOrientation()` method to retrieve the current stage window orientation.

If you want to take a specific action in response to a window's orientation change, you can add a listener to the stage window for the `orientationchange` event, and then respond to the change:

```
this.controller.listen(document, "orientationchange",
    this.handleOrientation.bindAsEventListener(this));
.
.
.
MyAssistant.prototype.handleOrientation(event)
     Mojo.Log.info("Orientation change position: ", event.position, " pitch: ",
       event.pitch, " roll: ", event.roll);
};
```

The new orientation data are passed as properties to the event object described in Table 9-1.

Table 9-1. Orientation change event properties

Property	Value	Description
`position`	Integer	Numeric value from 0 to 5, where:
		0 = face up
		1 = face down
		2 = up, or normal portrait
		3 = down, or reverse portrait
		4 = left, or landscape, left side down
		5 = right, or landscape, right side down
`roll`	Float	Righthanded rotation about the x-axis (in degrees)
`pitch`	Float	Righthanded rotation about the y-axis (in degrees)

Pitch and roll are absolute values with respect to the device being face up and flat on a table (in this position, they're guaranteed to be within [–90 degrees, 90 degrees]). Start with the device flat on the table in portrait mode with the bottom of the device facing you; pitch and roll in this position are both 0.

As you tilt the device toward you, pitch changes from 0 to –90 (when the device is completely vertical). Start again with the device flat: tilting the device away from you, the pitch increases from 0 to 90. Tilting to the right increases the roll from 0 to 90. Tilting left changes roll from 0 to –90.

When the device is face up and completely vertical the pitch is –90 degrees. As you continue to tilt the device toward you, the pitch goes to 0 (face down). The same is true of the roll. When you tilt the right side perpendicular with the ground, the roll will be 90. As you continue rotating (so that the device is face down), the roll decreases to 0.

With these definitions, you can see that there are two orientations where you'll get the same angles (one when face up and one when face down). You can use the z-axis to distinguish the two cases.

Shake

There are three events: `shakestart`, `shaking`, and `shakeend`. As you would expect, the start and end events are sent as soon as a shaking motion starts or stops. Shaking events are sent regularly while shaking continues, at the same rate as raw events, or 4Hz. The events include a `magnitude` property (in units of *gravitational acceleration* or *g*'s), where a large value indicates more vigorous shaking:

```
this.controller.listen(document, "shaking",
  this.handleShaking.bindAsEventListener(this));
.
.
.
MyAssistant.prototype.shaking(event)    {
     Mojo.Log.info("Shaking with magnitude: ", event.magnitude);
);
```

In practice, you may not need to listen to the shake start and end events unless you have some specific actions for those events. The shaking events will be sent as soon as the shaking motion commences, and will cease when the motion stops.

Raw acceleration

Detailed acceleration data is provided with each `acceleration` event, which are sent regularly whenever the device is in motion. While this information is accurate enough for games and similar dynamic applications, it won't be sufficient to allow inertial position tracking applications. Note that acceleration events are targeted at the window and do not bubble up to the containing context:

```
this.controller.listen(document, "acceleration",
    this.handleAcceleration.bindAsEventListener(this));
.
.
.
MyAssistant.prototype.acceleration(event)    {
     Mojo.Log.info("X: ", event.accelX, "; Y:", event.accelY,
       "; Z:", event.accelZ, "; time: ", event.time);
```

Acceleration events contain additional properties, listed in Table 9-2.

Table 9-2. Acceleration event properties

Property	Value	Description
`accelX`	Float	Acceleration along the x-axis (in g's)
`accelY`	Float	Acceleration along the y-axis (in g's)
`accelZ`	Float	Acceleration along the z-axis (in g's)

Alarms

You should use the JavaScript window methods `setInterval()` or `setTimeout()` to return to your application after a delay:

```
var wakeupFunction = function() {
        Mojo.Log.info("It's a wakeup call!");
};
window.setTimeout(wakeupFunction, 20000);
```

This is a lightweight delay timer, which executes a function after a specified period. In this example code, `wakeupFunction` is executed after a delay of 20 seconds. This type of alarm will work as long as the device is awake and where some imprecision is acceptable.

In all other situations, you will want to use the Alarm service, which is based on the device's real-time clock (RTC). Alarms are intended to wake applications while minimized or maximized, or to drive polling for dashboard applications. Alarms will:

- Accurately account for time changes; timeouts are accurately tracked across device sleep states, timezone changes, manual changes to time settings, or other changes that affect the displayed or perceived time on the device.
- Wake the device up from sleep; if needed, timers will wake up the device when the alarm fires.
- Fire after a specified delay or at a specified date and time.
- Make a specific service request when the alarm fires; commonly it will be an `applicationManager` service request to call the originating application's `handleLaunch` method, but can be any service call.

A good example of an alarm at work is using the Alarm service to wake an application up periodically:

```
this.controller.serviceRequest("palm://com.palm.power/timeout", {
    method: "set",
    parameters: {
        key: "com.palm.app.news.update",
        in: News.feedUpdateInterval,
        wakeup: "true",
        uri: "palm://com.palm.applicationManager/open",
        params: {
            id: "com.palm.app.news",
            params: {action: "updateFeed"}
```

```
        }
    },
    onSuccess: this.onSuccessHandler,
    onFailure: this.onFailureHandler
});
```

This example from News sets a *relative* alarm according to the requested `News.feedUpdateInterval` and requests that it wake up the device if it is asleep. When the alarm fires, it will use the Application Manager to launch News (even if News isn't running at the time) with a launch parameter indicating that this is an alarm for feed updates. You will learn more about handling launch events and using alarms with background applications in Chapter 10.

Another option is to set an alarm at a specific date and time:

```
this.controller.serviceRequest("palm://com.palm.power/timeout", {
    method: "set",
    parameters: {
        key: "com.palm.app.news.daily",
        at: "04-23-2009 03:30:00",
        wakeup: "true",
        uri: "palm://com.palm.applicationManager/open",
        params: {
            id: "com.palm.app.news",
            params: {action: "updateFeed"}
        }
    },
    onSuccess: this.onSuccessHandler,
    onFailure: this.onFailureHandler
});
```

In this example, News could be set to update the news feeds at 3:30 A.M. GMT. This *calendar* alarm uses the `at` property (in place of the `in` property for a relative alarm) to set the date and time for the alarm. By definition, you must set calendar alarms for each occurrence; there isn't a provision at this time for periodic or recurring alarms.

To clear the alarm, use the `clear` method:

```
this.controller.serviceRequest("palm://com.palm.power/timeout", {
    method: "clear",
    parameters: {
        key: "com.palm.app.news.daily",
    },
    onSuccess: this.onSuccessHandler,
    onFailure: this.onFailureHandler
});
```

The alarm's `key` property is used to clear the periodic alarm that was set with that property.

Some additional notes about alarms:

- Setting a relative alarm causes the device to wake and fire the alarm at a fixed time in the future. This is independent of time changes on the device by either the user

or an external source (such as network time or timezone changes). The maximum period for relative alarms is 24 hours and the minimum is 5 minutes, but calendar alarms do not have a 24-hour time limit.

- Alarms are preserved across reboots. If an alarm expires while the device is shut down, the alarm will fire when the device starts up again.
- If the alarm service call fails, one retry attempt will be made 30 seconds later.
- These alarms are coarse-grained alarms, so don't expect millisecond or even one-second resolution. At worst, an alarm may fire a few seconds from the intended time.

Because the Alarm service can wake up your application even while the device is asleep, the service is integral to running an application in the background. We'll use alarms to turn News into a background application in Chapter 10.

Connection Manager

Use the Connection Manager's `getStatus` method to get updates on the device connection status. Some applications need to manage data access based on the connection state or type of connection available:

```
this.controller.serviceRequest("palm://com.palm.connectionmanager", {
    method: "getStatus",
    parameters: {subscribe:true},
    onSuccess: this.onSuccessHandler,
    onFailure: this.onFailureHandler
});
```

The response will provide the connection status at the time of the call. Use the subscription option to register for updates each time the connection status changes. In every case, the `onSuccess` callback is made with a single response argument, an object describing the connection status. The return value is set to true and the connection properties are provided as described in Table 9-3.

Table 9-3. Response properties for connection status

Name	Description
`isInternetConnectionAvailable`	Set to `true` when a connection is present
`wifi`	Object describing the WiFi connection status, with properties for state (connected or disconnected), `ipAddress`, `ssid`, and `bssid`
`wan`	Object describing the WAN connection status, with properties for state, ipAddress, and type (`unknown`, `unusable`, `gprs`, `edge`, `umts`, `hsdpa`, `1x`, and `evdo`)
`btpan`	Object describing the Bluetooth Personal Area Network connection status, with properties for state, `ipAddress`, and `panUser`

Location Services

Palm webOS provides basic location services to get one or more location fixes. You can specify fixes by mode. In *automatic* mode, the system picks the most appropriate mode based upon your accuracy and response time requirements. Specific fix types include assisted GPS and Cell ID with or without WiFi ID.

Get current position

You can get the current position from the built-in GPS or through Cell ID or WiFi ID, depending on what's available. The simplest call uses all default settings:

```
this.controller.serviceRequest("palm://com.palm.location", {
    method: "getCurrentPosition",
    parameters: {},
    onSuccess: this.locationSuccess.bind(this),
    onFailure: this.locationFailure.bind(this)
});
```

The default settings will provide a new (not cached), medium accuracy (within 350 meters) fix within 5 to 20 seconds. Through different property settings, you can force greater or less accuracy, faster or slower response times, and agree to accept a cached fix. Typically, greater accuracy means a slower response to the request.

The Location service tries to get a fix with the requested accuracy. If it is not able to get a fix within the maximum allowed time, it returns the best match from the cache. If there is no entry in the cache, the service returns a timeout error.

If the Location service can get a fix, the `onSuccess` function will be called with a response object with the properties described in Table 9-4.

Table 9-4. Response properties for Location service position fix

Property	Description
`errorCode`	Set to zero if the request is successful; a nonzero value otherwise (see Table 9-5 for a complete list of error codes)
`timestamp`	The time (in milliseconds) when the location fix was created
`latitude`	A number representing the latitude in degrees of the location; valid range is −90.0, 90.0
`longitude`	A number representing the longitude in degrees of the location; valid range is −180.0, 180.0
`horizAccuracy`	Horizontal accuracy of the location in meters; if unknown, value is −1
`heading`	A number representing the compass azimuth in degrees; valid range is 0.0, 360.0; if unknown, value is −1
`velocity`	A number representing velocity in meters per second; if unknown, value is −1
`vertAccuracy`	Vertical accuracy of the location in meters; if unknown, value is −1

If there's an error, the `onFailure` function is called with a nonzero `errorCode`. A list of possible errors is provided in Table 9-5.

Table 9-5. Location service error codes

Error Code	Description
0	Success
1	Timeout
2	Position Unavailable
3	Unknown
4	GPS Permanent Error
5	Location Service Off
6	Permission denied: user hasn't agreed to location service terms of use

Tracking

Use the `startTracking` method to get a series of fixes. Tracking is effective for navigation applications or anywhere you want to update the device location over a period of time. A new tracking fix is provided about once a second:

```
this.trackingHandle = this.controller.serviceRequest("palm://com.palm.location", {
    method: "startTracking",
    parameters:{subscribe: true},
    onSuccess: this.trackingSuccess.bind(this),
    onFailure: this.trackingFailure.bind(this)
});
```

You need to subscribe to this service and save the request object so that you can cancel the tracking request when you are done with it:

```
this.trackingHandle.cancel();
```

The `onSuccess` function will be called with a response object that has the same properties provided with the `getCurrentPosition` method; they are described in Table 9-4. As with `getCurrentPosition`, you can get tracking fixes even without GPS, although at a reduced level of accuracy (Cell ID or WiFi ID are less accurate than GPS).

If there's an error, the `onFailure` function is called. You will continue to receive tracking fixes even after an error, since most errors are transient. In one case though, GPS Permanent Error, the error will persist, but you can still receive tracking data from Cell ID or WiFi ID.

If you receive a GPS Permanent Error, you can still receive location services through Cell ID and WiFi ID. In this case, the error will be reported with a callback to the specified onFailure function, but thereafter you will receive ongoing tracking fixes through the `onSuccess` callback.

Reverse location

An additional Location service provides you with a physical address when provided with a location described with latitude/longitude values:

```
this.controller.serviceRequest("palm://com.palm.location", {
    method: "getReverseLocation",
    parameters: {
        latitude: "37.7779",
        longitude: "-122.414"
    },
    onSuccess: this.reverseSuccess.bind(this),
    onFailure: this.reverseFailure.bind(this)
});
```

If successful, the `onSuccess` callback is passed a response object with a single address property. The address is two or three lines, each delimited by semicolons, where the street address is optional:

```
street address;
locality (eg. in the US, city, state, zipcode);
country
```

For example, this code sample would return the following string:

```
98th 8th St ;San Francisco, CA 94103 ;USA
```

Power Management

The device will automatically go to sleep after a period of inactivity, where inactivity is primarily defined as the absence of user interaction: gestures, touches, or keyboard input. The user can set a preference to trigger sleep after a minimum of 30 seconds or a maximum of 3 minutes. Some system services, such as audio or video playback will defer sleep, but if you need to keep the device awake, you can use the `activityStart()` and `activityEnd()` methods in the Power Management service.

You would use these service methods if your application performs an extended operation, such as syncing or downloading a lot of data, or if your application includes a passive viewing feature like a slide show. You will also use this in background applications where you need more than the few seconds allotted during an alarm wakeup.

Alert the Power Management service that you are starting an activity that will require the device to stay awake:

```
this.controller.serviceRequest("palm://com.palm.power/com/palm/power", {
    method: "activityStart",
    parameters: {
        id: "com.palm.app.news.update-1",
        duration_ms: '120000'
    },
    onSuccess: this.activitySuccess.bind(this),
    onFailure: this.activityFailure.bind(this)
});
```

Provide a unique ID, which should be your application ID with an activity name and an occurrence count. This recommended format will allow you to distinguish between requests and manage multiple requests if needed, but the only requirement is that the ID string be unique. The activity's expected duration is provided in milliseconds and cannot exceed 900,000 milliseconds (15 minutes).

The power management service will automatically terminate your activity request at the end of its duration or 15 minutes, whichever is shorter. You should notify the service when your activity completes, as every bit of power efficiency is important:

```
this.controller.serviceRequest("palm://com.palm.power/com/palm/power", {
    method: "activityEnd",
    parameters: {
        id: "com.palm.app.news.update-1"
    },
    onSuccess: this.activitySuccess.bind(this),
    onFailure: this.activityFailure.bind(this)
});
```

The only parameter is the `id` provided to the `activityStart()` method. Activities are not canceled when an application is closed, so you should use `activityEnd()` in your `cleanup()` method when there are any outstanding activity requests.

System Properties

Applications can request a named system property, which is currently limited to retrieving the unique device ID. Generally, the requested system property is named as a string to `parameter.key`; in this case, the device ID is named `com.palm.properties.nduid`:

```
this.controller.serviceRequest("palm://com.palm.preferences/systemProperties", {
    method: "getSysProperty" ,
    parameters:{"key": "com.palm.properties.nduid" },
    onSuccess: this.onSuccessHandler
    }
});
```

The `nduid` is a hardware-encoded device ID that is guaranteed to be unique. An example device ID is `a66690b3632bb592b29c6a15416717b9ee072b3f`.

On a successful callback, you will find the device ID as the value assigned to the key, `com.palm.properties.nduid`. In this example:

```
this.onSuccessHandler = function() {
    Mojo.Log.info("Success; nduid = ", response["com.palm.properties.nduid"]);
};
```

The device ID is the recommended way to uniquely identify the user or their device. The device ID is better than the phone number or device serial number because it is guaranteed to be unique to a specific device, yet it is difficult for others to use it to identify a specific user. The phone number is considered private user information and should never be used or transmitted with any user data. The serial number is printed on the device, making it easier to associate a device to a specific user.

System Services

The system is designed to expose a set of services enabling applications to access some general system settings. Currently, the only exposed service is one that provides the system time.

Make the request to the `getSystemTime` method, and optionally subscribe to notifications of changes to the time or timezone:

```
this.controller.serviceRequest("palm://com.palm.systemservice/time", {
    method: "getSystemTime",
    parameters: {subscribe: true},
    onSuccess: this.timeSuccess.bind(this),
    onFailure: this.timeFailure.bind(this)
});
```

Whether the subscribe property is set or not, the `onSuccess` function will be called initially with the response object as an argument. The properties are described in Table 9-6.

Table 9-6. Response properties for system time

Property	Description
`localTime`	The time for the current timezone in seconds
`offset`	Offset from UTC in minutes
`timezone`	Current timezone in the "TZ" environment variable format

In most cases, you can use the JavaScript Date object to get the current date and time. The `getSystemTime` service method gives you the timezone and allows you to subscribe to time changes, which may be important to your application. In most other cases, the Date object is a lightweight and more versatile source of date and time information.

System Sounds

The System Sounds service is used to create audio feedback for direct user actions. This might include button or keypad clicks, transition sounds, action audio, or any audible response to a user action. It's not intended for sustained audio, such as background audio or any lengthy playback. The call is limited to a static list of sounds and is not customizable.

The specified sounds will be played as soon as the call is received, with low latency. Call System Sounds using the `playFeedback` method, with the requested sound name as a string assigned to the `name` property:

```
this.controller.serviceRequest("palm://com.palm.audio/systemsounds", {
    method: "playFeedback",
    parameters:{name: "shutter",
        onSuccess: this.onSuccessHandler,
```

```
        onFailure: this.onFailureHandler
    }
});
```

The callbacks will receive the response object with `returnValue` set to true if the sound played successfully; otherwise, it will be set to false. If false is returned, the `errorText` property will include an indication of the type of error encountered.

The available sounds are enumerated in Appendix B, and you can find them on the Palm developer site.

Cloud Services

Palm webOS is designed around the needs of connected applications, including the deep integration of web or cloud services into the platform. The intention is to create a platform supporting not just client application UIs and services, but web services as well. This extends the platform beyond the boundaries of the device to the Web itself.

The initial webOS cloud service offering is Mojo Messaging, an XMPP-based messaging service supporting notifications from web services to the device and eventually between device applications and services. Push notifications are much more power-efficient and extend battery life. Applications will be notified when there is a service update, eliminating the need to poll. In addition, the Mojo Messaging architecture extends the messaging model to enable client applications and services to communicate with each other.

You will typically use Mojo Messaging following these basic steps:

1. Create an endpoint, to which a key is returned.
2. Share the key with the cloud service from which you need updates.
3. Subscribe to the endpoint with a callback to receive messages.
4. Wake when a message arrives from the cloud on the defined callback.

Start by creating a notification endpoint. This registers the receiving application with the messaging service and establishes a publishing key for notifications:

```
this.controller.serviceRequest("palm://com.palm.pubsubservice", {
    method: "createEndpoint",
    parameters: {
        endpoint: "com.palm.app.news.newstories",
        description: "When new stories are published, notify the News application"
    },
    onSuccess: this.createSuccess.bind(this),
    onFailure: this.createFailure.bind(this)
});
```

Share the publishing key with any service that would send notifications to the applications; typically this is done with an HTTP POST request.

You will retrieve the publishing key from the response object returned in the `onSuccess` case, as outlined in Table 9-7.

Table 9-7. Response properties for createEndpoint method

Property	Description
`endpoint`	Endpoint that was passed on the createEndpoint method
`publishKey`	The key to be used to publish to this endpoint

Subscribe for notifications published to the endpoint and renew the subscription after each notification:

```
this.controller.serviceRequest("palm://com.palm.pubsubservice", {
    method: "subscribe",
    parameters:  {
        endpoint: "com.palm.app.news.newstories",
        subscribe: true
    },
    onSuccess: this.notificationHandler.bind(this),
    onFailure: this.subscribeFailure.bind(this)
});
```

Whenever a notification comes into the device from the endpoint, it is dispatched to the application through a callback to the function defined as the `onSuccess` handler.

The Cloud services are in beta release at this time, so we aren't going to look at them in detail here. If you're interested in this class of service or the Messaging service in particular, you should view the latest information at *http://developer.palm.com*.

Summary

This chapter wraps up the presentation of the Services available on Palm webOS. There are a number of System services available. The accelerometer receives and responds to device orientation, shaking events, and raw acceleration data. Alarms wake up devices or make specified service requests after a delay or at a specified time. The Connection Manager checks connection status. Location services retrieve a device's current position and provide tracking capabilities. The Power Management service lets you override the device's sleep function in order to carry out long operations. System Properties allows you to request a device ID, and System Services provides the system time. Finally, System Sounds lets you associate audio feedback in direct response to user actions, such as typing. While few of these System services are used in the News application, specific code samples were given for most of the service calls.

Another type of service is the Cloud service, specifically Mojo Messaging, which is an XMPP messaging service. This service is essential for background applications that require notifications from a web service or applications that want to share information and events across a community.

CHAPTER 10

Background Applications

Until now, mobile and web applications have generally been limited to a single window, within which the user moves from view to view, reading content, performing tasks, and providing input in a serial fashion. With Palm webOS, mobile applications can anticipate the user's needs by using notifications while running in the background, and they can put common tasks into separate windows for quick access when needed.

Mojo includes a sophisticated notification system that supports banners and pop-ups, which allow you to display information subtly or get the user's attention with more urgent messages. In this chapter, you'll be introduced to notifications and dashboard summaries, with code examples of each to show you how to use them in your application.

Advanced applications are built around an application assistant, which coordinates the application's stages, handles background tasks and launch requests, and provides general command handling for the application. This structure lets you build multistage applications with secondary cards or dynamic dashboard stages, and run your application in the background, waking the device from sleep or across reboots of the device.

Even applications that don't wake the device will want to adapt their behavior when running in the background. There's no need to waste CPU cycles or battery to update the display or to frequently update data while the user is looking or working elsewhere. This chapter will give you the basic techniques for managing minimized card and dashboard stages, and will provide guidelines and best practices.

Stages

In Chapter 1, you were introduced to stages:

> A *stage* is similar to a conventional HTML window or browser tab. Applications can have one or more stages, but typically the primary stage will correspond to the application's card. Other stages might include a dashboard, a pop-up notification, or secondary cards for handling specific activities within the application. Refer to email as an example of a multistage application, where the main card holds the account lists, inbox and displays the email contents, but new emails are composed in a separate card to allow for switching

between compose and other email activities. Each card is a separate stage, but still part of a single application.

We haven't worked much with stages so far, but they are an essential part of the features discussed in this chapter. Each secondary card, dashboard summary, or pop-up notification is a separate window, and each window corresponds to a stage, with a stage controller that manages that window. Recall that each stage controller has a stack of scene controllers, with the topmost scene activated and in view within the stage's window.

Before we build each feature into the News application, we'll start with some general information about using stages.

Creating New Stages

All stages are created the same way, with a call to `createStageWithCallback()`, an application controller method, and a callback function, which at a minimum will push the first scene using the newly created stage controller:

```
var stageArguments = {name: "main", lightweight: true};
var pushMainScene = function(stageController) {
    stageController.pushScene("main");
};
this.controller.createStageWithCallback(stageArguments, pushMainScene, "card");
```

You can refer to the API documentation for the specifics of this call, but you should always do the following:

- Name the stage; the name identifies the stage and you will use the name later to determine whether the stage exists.
- Set the `lightweight` property to `true` (early on, Mojo included *heavyweight* and *lightweight* stages, but only *lightweight* stages are supported now).
- Specify the callback function.

The last argument, the stage type, defaults to `card`, so it is optional for this example. The complete set of stage types are:

```
Mojo.Controller.StageType = {
  popupAlert:  "popupalert",
  bannerAlert: "banneralert",
  dashboard:   "dashboard",
  card:        "card"
};
```

You can specify a stage assistant in `stageArguments`; otherwise, the stage will be created without a stage assistant.

Using Existing Stages

Often, you will want to create a stage only when it doesn't already exist. If the stage exists, you will likely want to put focus on the stage or update its contents. Use `getStageController()` to get the stage controller; a return value of `undefined` means that the stage doesn't exist, so you must create one. Otherwise, use the returned value as the stage controller to focus or update the existing stage:

```
// Look for an existing main stage by name.
var stageController = this.controller.getStageController("main");

if (stageController)    {
     stageController.window.focus();
} else {
     var pushMainScene = function(stageController) {
         stageController.pushScene("main");
     };
    var stageArguments = {name: "main", lightweight: true};
    Mojo.Controller.AppController.createStageWithCallback(stageArguments,
      pushMainScene, "card");
}
```

The `getStageController()` method will also return `undefined` when the stage controller has been created but is not available at the time of the call. Use `getStageProxy()` whenever you may be trying to access the stage close to where it is being created. The `getStageProxy()` method will still return undefined if the stage does not exist or hasn't been created at the time of the call. However, you can't use the returned object as a stage controller; the returned object simplify verifies the existence of the stage controller and can only be used with the `delegateToSceneAssistant()` method, which you'll learn about later in this chapter.

It can take as long as one second to create a stage in some instances. If your `get` request for the stage controller could occur within a second of the `create` request, you should use `getStageProxy()`.

Working with Stages

The News application uses a single card stage, which is created automatically when the user launches the application. For most of the advanced features, we will be creating and accessing stages directly and turning News into a multistage application. When working with multistage applications you should follow these guidelines:

- JavaScript must be loaded through *sources.json*. JavaScript cannot be loaded through script tags (other than the required script tag for *mojo.js*), so multistage applications will fail unless the source files are specified in *sources.json*.

- Specify `noWindow:true` in *appinfo.json*; applications with an app-assistant and multiple stages need to indicate that they will initially launch as a *no window* or background app, creating their own stages, or windows, explicitly.
- There is no support for Prototype's `$()` function; get elements with methods from the scene or widget controller.

Be careful about nested functions. With single stage applications, you could use the global `$()` function in a nested function, but now you must use:

```
var controller = this.controller
```

to put the appropriate scene controller in a local variable so it is visible to the nested function.

- The `Mojo.Controller.stageController` global is not supported; replace it with the stage controller property of the scene or widget controller.
- Do not use the `window` global; instead, use the window property of the stage, scene, or widget controller. You can still use the window global in the application assistant.
- Do not use the `document` global; instead, use the document property from the stage, scene, or widget controller, or the `ownerDocument` property of an element if all you have is an element reference.
- Don't use `document.viewport`; instead, use:

```
Mojo.View.getViewportDimensions(targetDocument);
```

You can follow these practices even when working with single-stage applications, though the convenience of using the framework to manage stage creation and the convenience of the prototype `$()` function are worth considering.

Notifications

You can post a *banner notification*, which appears subtly in the notification bar below the main window and is typically followed by a dashboard panel to allow for deferred action on the notification. For more urgent actions, you can use a *pop-up notification*, which slides up out of the notification bar and reduces the window size of the Card view or foreground card. All notifications and dashboards are *nonmodal*, meaning users can continue to interact with whatever is in the foreground view until they are ready to address the notifications or interact with the dashboard.

Banner Notifications

A banner includes an icon and a short message accompanied by an audible alert signaling the user to the presence of the banner. After a few seconds, the banner is removed. The banner is usually accompanied by a dashboard summary, which serves as a reminder of the notification and can provide additional details about the notification.

Back to the News: Banner notifications

Each time new stories are added during an update cycle, News will post a banner notification with the name of the feed and number of new stories, as shown in Figure 10-1. We'll add the code to actually post the notification in the `feedRequestSuccess()` method, but we'll add an initial condition that will inhibit notifications when they are turned off in the application's preferences scene. The code for the preferences scene change isn't shown, but you can see it in Appendix D:

```
// If successful processFeed returns News.errorNone,
if (feedError == News.errorNone)    {
    var appController = Mojo.Controller.getAppController();

    // Post a notification if new stories and application is minimized
    if (this.list[this.feedIndex].newStoryCount > 0)   {
        Mojo.Log.info("New Stories: ", this.list[this.feedIndex].title," : ",
          this.list[this.feedIndex].newStoryCount, " New Items");
        if (News.notificationEnable)   {
            var bannerParams = {
        // ** These next two lines are wrapped for book formatting only **
                messageText: this.list[this.feedIndex].title+": "
                  +this.list[this.feedIndex].newStoryCount+" New Items"
            };

            appController.showBanner(bannerParams, {},
             this.list[this.feedIndex].url);

        }
    }
} else     {

    // There was a feed process error; unlikely, but could happen if the
    // feed was changed by the feed service. Log the error.
    if (feedError == News.invalidFeedError)    {
        Mojo.Log.info("Feed ", this.nameModel.value, " is not a supported
 feed type.");
    }
} .....
```

Get the application controller, then call its `showBanner()` method. This will post a single line of text, truncated to fit the screen width, and a scaled version of the calling application's icon, shown immediately to the left of the message. The argument, `bannerParams`, includes the message string. It isn't necessary for News, but you can add

a `soundClass` property for an audible alert; currently, the only supported value for `soundClass` is `alerts`.

Figure 10-1. A banner notification

If the user taps on the banner as it is displayed in the notification bar, the framework will relaunch your application. This will be described in more detail in the following section on Advanced Applications, where you'll see how to use an explicit `handleLaunch()` method in the application assistant to receive these and other launch requests. In this simple form, the framework will activate the scene at the top of the News scene stack when the user taps the banner. The second argument to `showBanner()` is an empty object representing the launch parameters.

Even when there are no launch parameters, you still need to provide an object for them (in this case, an empty object). If no object is passed at all, tapping the notification will not launch your application.

You can include the optional third argument, `category`, to distinguish banner notifications within your application. Since banners are displayed for a fixed length of time (five seconds as of this writing), they can back up if multiple requests are made before

they can be displayed. If there is more than one banner notification within a named category, the framework will discard all but the last of them. If you are using banner notifications from different sources, you may want to identify them through a category.

Any notification that doesn't include a specific category belongs by default to the `banner` category.

Banners from separate applications are always distinct. Category is only needed when there are multiple banner categories within an application.

Minimized Applications

Generally, you should avoid using notifications when your application is *maximized*, meaning it is the current foreground card. Use banner notifications when not in focus, either *minimized* (with a card view but not the foreground card) or in the *background* (without a card stage). When maximized, your application should use on screen representations that show visible changes to dynamic data or dialog boxes for critical alerts or events.

You can receive events for maximize/minimize transitions by adding listeners to your stage controller's document for the `Mojo.Event.stageActivate` and `Mojo.Event.stageDeactivate` events. However, in this case, we'll use a stage controller method, `isActiveAndHasScenes()`, which returns `true` when the stage controller is the active stage controller and there are scenes displayed within the stage:

```
if (!Mojo.Controller.stageController.isActiveAndHasScenes()
 && News.notificationEnable) {
     var bannerParams = {
// ** These next two lines are wrapped for book formatting only **
         messageText: this.list[this.feedIndex].title+":
           "+this.list[this.feedIndex].newStoryCount+" New Items"
       };

       appController.showBanner(bannerParams, {action: "notification",
           index: this.feedIndex}, this.list[this.feedIndex].url);

}
```

With these changes, News will only post banner notifications when minimized. This is the recommended behavior for most applications.

Pop-up Notifications

Use a pop-up notification when you need to get the user's attention urgently. These notifications slide up from the Notification bar with a message and one or more selection options. For example, calendar events and incoming phone calls use pop-up

notifications; you can see a phone pop-up notification when you connect your phone to a computer using USB.

You shouldn't use pop-up notifications very often, as they are disruptive by design, taking up as much as half the screen. You can generate a pop-up by creating a stage and pushing a pop-up scene onto it:

```
var appController = Mojo.Controller.getAppController();
var pushPopup = function(stageController)  {
    stageController.pushScene('myPopup', "Hot off the presses!");
};
appController.createStageWithCallback({name: "popupStage",
  lightweight: true, height: 200}, pushPopup, 'popupalert');
```

Pop-up stages take the same stage arguments as the card and dashboard stage examples, with an optional `height` property. Indicate that this is a pop-up stage by using `'popupalert'` as the final argument. Typically, you specify the stage `name` and `lightweight` properties, and include the callback function as the second argument.

Specific to pop-ups is the option to set the `height` property. The default pop-up height is 200 pixels on the Palm Prē, but you can override it to a maximum of 400 pixels. It's recommended that you use the default at all times, but the flexibility is there if you have an unusual requirement that needs a different height.

Customize the pop-up notification in the scene assistant and views. As an example, you can generate the notification shown in Figure 10-2 by first creating a main pop-up scene in *views/popup/popup-scene.html*:

```
<div class="notification-panel">
    <div class="notification-container" x-palm-popup-content="">
        <div id="notification-icon" class="notification-icon"></div>
        <div id="info"></div>
    </div>
    <br>
    <br>
    <br>
    <div class="popupdiv">
        <div class="palm-button affirmative popupbutton" id="addButton">
          Ok
        </div>
        <div class="palm-button negative popupbutton" id="closeButton">
          Close
        </div>
    </div>
</div>
```

The attribute `x-palm-popup-content=""` is used within the pop-up scene div, which contains the main content for the pop-up. The System UI will use this special attribute to draw only the main content area of the pop-up in the lock screen when the display is turned on from sleep. The pop-up's buttons should be kept outside of this div, as they are not actionable when the device is locked.

Add a template named *popup/item-info.html*, for rendering the variable content into the `info` div above:

```
<div id='notification-title' class="notification-title">
  #{subject}
</div>
<div id='notification-subtitle' class="notification-subtitle">
  #{eventSubtitle}
</div>
```

Figure 10-2. A pop-up notification

You can see that we're using a lot of specific notification style classes to lay out the pop-up scene to follow the System UI conventions and to style the title and subtitle text. The framework doesn't load some of the common styles used with cards with dashboard or pop-up stages, so you may have to create your own in those cases. The styles in this example are from the Calendar application and are not provided by the framework. In this example, you would add these CSS rules to your application's CSS file:

```
/* Pop-up notifications */

    .notification-panel    {
            background: #000;
            color: #fff;
            overflow: hidden;
            padding: 15px;
            top: 0;
```

```
        width: 320px;
}

.notification-container               {
        width: 290px;
        height: 48px;
        padding: 0;
        position: fixed;
        top: 10px;
        display: table-cell;
        vertical-align: middle;
}

.notification-container .notification-icon    {
        width: 48px;
        height: 48px;
        margin-right: 5px;
        float: left;
        background: url(../images/dashboard-icon-news.png) top left no-repeat;
}

.notification-container .notification-title    {
        width: 230px;
        height: 18px;
        margin-top: 5px;
        margin-bottom: 3px;
        padding-bottom:5px;
        overflow: hidden;
        text-overflow: ellipsis;
        white-space: nowrap;
        color: #fff;
        font-size: 16px;
        font-weight: bold;
}

.notification-container .notification-subtitle    {
        margin-top:-5px;
        width: 230px;
        height: 18px;
        overflow: hidden;
        text-overflow: ellipsis;
        color: #fff;
        font-size: 14px;
        white-space: nowrap;
}

.notification-container .notification-title span,
.notification-container .notification-subtitle span {
        font-weight: normal;
}
```

A pop-up's scene assistant is a conventional scene assistant. In this case, we've created *popup-assistant.js* in the *assistants* directory. The assistant accepts a message string as an argument and inserts that message into the *popup/item-info.html* template:

```
function PopupAssistant(message) {
      this.message = message;
 }

PopupAssistant.prototype.setup = function() {
    this.update(this.message);

    this.okButton = this.controller.get("okButton");
    this.okHandler = this.handleOk.bindAsEventListener(this);
    Mojo.Event.listen(this.okButton, Mojo.Event.tap, this.okHandler);

    this.closeButton = this.controller.get("closeButton");
    this.closeHandler = this.handleClose.bindAsEventListener(this);
    Mojo.Event.listen(this.closeButton, Mojo.Event.tap, this.closeHandler);
};

PopupAssistant.prototype.update = function(message) {
    this.info = {eventSubtitle: message, subject: "News"};
    Mojo.Log.info("Popup Update");
    //  Use render to convert the object and its properties
    // along with a view file into a string containing HTML
    var renderedInfo = Mojo.View.render({
        object: this.info,
        template: 'popup/item-info'
    });

    var infoElement = this.controller.get('info');
    infoElement.innerHTML = renderedInfo;
};

PopupAssistant.prototype.handleOk = function(){
    Mojo.Log.info("Ok received");
};

PopupAssistant.prototype.handleClose = function(){
    this.controller.window.close();
};
PopupAssistant.prototype.cleanup = function(){
    Mojo.Event.stopListening(this.okButton, Mojo.Event.tap, this.okHandler);
    Mojo.Event.stopListening(this.closeButton Mojo.Event.tap, this.closeHandler);
};
```

You can do almost anything within a pop-up scene that you can do in any other scene, but you should limit your actions to simple messages and selections. Your pop-up assistant must close the window on exit to close the stage and remove it from the display. Don't forget to clean up any event listeners when the stage is closed.

Updating Pop Ups and Dashboard Panels

Pop-up notifications and dashboard panels are persistent; they aren't removed from the display until the user closes them. In some cases, they are displayed long enough that you may want to update the contents. For example, a calendar pop-up notification shows the next event on the calendar, but if that event passes while the notification is on the screen, a new calendar event should replace it.

You should prepare for updates by structuring your pop-up assistant with a method to refresh the content. Review the sample code under the section "Pop-up Notifications" on page 235 for an example using the `update()` method, noting that you can name the method whatever you'd like.

Before creating the pop-up stage, check to see whether the stage already exists and call the update method instead of creating a new stage:

```
var appController = Mojo.Controller.getAppController();
var message = this.bannerTextModel.value;

var popupStage = appController.getStageProxy("popup");
if(popupStage) {
    popupStage.delegateToSceneAssistant("update", message);
}
else {
    var pushPopup = function(stageController){
        stageController.pushScene("popup", message);
    };

    appController.createStageWithCallback({name: "popup",
        lightweight: true, height: 200}, pushPopup, "popupalert");
}
```

Use `getStageProxy("popup")`, which returns a proxy to the stage controller if the stage exists or is in the process of being created. Invoke the scene's method using `delegateToSceneAssistant()`, naming the method in the first argument. All other arguments are passed as arguments to the named method. It's safer to use `getStageProxy()` to avoid a condition in which the stage is being created before the stage controller exists.

Stage proxies are *not* general substitutes for the stage controller; the only valid use of a stage proxy is as an existence test and a call to the proxy's `delegateToSceneAssistant()` method. You can't reliably use it to call other stage controller methods.

Dashboards

You will usually create a dashboard panel following a banner notification posting as a reminder of the notification. Dashboards can also display ambient information or provide a dynamic window for background applications when you don't need a full card stage. Although constrained by size and UI convention, dashboard summaries are fully functional stages, within which you can push scenes and employ any part of the Mojo API, though not all of the Mojo styles are available outside of card stages.

The full framework CSS is not automatically loaded when creating pop-up or dashboard stages. Currently, some of the styles that you can use in a card stage or main application window are not available within noncard stages. This will be addressed in time, but for now you may have to copy some style properties and selectors to your application CSS from the framework CSS.

Dashboards are constrained windows that span the full screen width and about 10% of the screen height in portrait mode; on the Palm Prē, that's 320 pixels wide and 48 pixels high. But just as with card windows, you should lay out your dashboard windows to handle different widths for landscape modes or different screen sizes on future Palm webOS devices.

Back to the News: Adding a Dashboard Stage

You can create a dashboard stage similar to the general stage example shown earlier in this chapter, but you will declare it as a `dashboard` stage type. We'll add a dashboard stage to News in `updateFeedSuccess()` by following the banner notification with a call to `createStageWithCallback()`, passing a callback function that pushes *dashboard-assistant.js* with the feedlist and the current feed index to post the feed's title and the most recent new story headline. We'll also pass a global constant, `News.DashboardStageName`, which defines the dashboard stage name, set the *lightweight* property to `true`, and specify this as a *dashboard* stage:

```
if (feedError == News.errorNone)    {
    var appController = Mojo.Controller.getAppController();
    var stageController = appController.getStageController(News.MainStageName);
    var dashboardStageController =
      appController.getStageProxy(News.DashboardStageName);
    // Post a notification if new stories and application is minimized
    if (this.list[this.feedIndex].newStoryCount > 0)   {
        Mojo.Log.info("New Stories: ", this.list[this.feedIndex].title,
          " : ", this.list[this.feedIndex].newStoryCount, " New Items");
        if (!Mojo.Controller.stageController.isActiveAndHasScenes()
         && News.notificationEnable)   {
            var bannerParams = {
            // ** These next two lines are wrapped for book formatting only **
                 messageText: this.list[this.feedIndex].title+":
                    "+this.list[this.feedIndex].newStoryCount+" New Items"
            };

            appController.showBanner(bannerParams,
                {action: "notification", index: this.feedIndex},
                 this.list[this.feedIndex].url);

            // Create or update dashboard
            var feedlist = this.list;
            var selectedFeedIndex = this.feedIndex;

            if(!dashboardStageController) {
                Mojo.Log.info("New Dashboard Stage");
                var pushDashboard = function(stageController){
                    stageController.pushScene("dashboard", feedlist,
                      selectedFeedIndex);
                };
                appController.createStageWithCallback({
                    name: News.DashboardStageName,
                    lightweight: true
                  }, pushDashboard, "dashboard");
```

```
            }
            else {
                Mojo.Log.info("Existing Dashboard Stage");
                dashboardStageController.delegateToSceneAssistant("updateDashboard",
                  selectedFeedIndex);
            }
        }
    }
} else     {

    // There was a feed process error; unlikely, but could happen if the
    // feed was changed by the feed service. Log the error.
    if (feedError == News.invalidFeedError)    {
        Mojo.Log.info("Feed ", this.nameModel.value,
         " is not a supported feed type.");
    }
}
```

Before creating the stage, call `getStageProxy(News.DashboardStageName)`. If the stage exists, the proxy will be defined. Call the dashboard assistant's `updateDashboard()` method with just the current feed index to use the existing dashboard and update its contents with information about the latest feed update.

Dashboard scenes have scene assistants and view templates since they are often dealing with dynamic data but are working within a restricted view. Use the same techniques described in the section "Pop-up Notifications" on page 235 to render scenes and handle updates:

```
/*  Dashboard Assistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Responsible for posting that last feed with new stories,
    including the new story count and the latest story headline.

    Arguments:
    - feedlist; News feed list
    - selectedFeedIndex; target feed

    Other than posting the new story, the dashboard will call the
    News apps handleLaunch with a "notification" action when the
    dashboard is tapped, and the dashboard window will be closed.
*/

function DashboardAssistant(feedlist, selectedFeedIndex) {
    this.list = feedlist;
    this.index = selectedFeedIndex;
    this.title = this.list[this.index].title;
    this.message = this.list[this.index].stories[0].title;
    this.count = this.list[this.index].newStoryCount;
}

DashboardAssistant.prototype.setup = function() {
    this.displayDashboard(this.title, this.message, this.count);
```

```
    this.switchHandler = this.launchMain.bindAsEventListener(this);
    this.controller.listen("dashboardinfo", Mojo.Event.tap, this.switchHandler);

    this.stageDocument = this.controller.stageController.document;
    this.activateStageHandler = this.activateStage.bindAsEventListener(this);
    Mojo.Event.listen(this.stageDocument, Mojo.Event.stageActivate,
        this.activateStageHandler);
    this.deactivateStageHandler = this.deactivateStage.bindAsEventListener(this);
    Mojo.Event.listen(this.stageDocument, Mojo.Event.stageDeactivate,
        this.deactivateStageHandler);
};

DashboardAssistant.prototype.cleanup = function() {
    // Release event listeners
    this.controller.stopListening("dashboardinfo", Mojo.Event.tap,
 this.switchHandler);
    Mojo.Event.stopListening(this.stageDocument, Mojo.Event.stageActivate,
        this.activateStageHandler);
    Mojo.Event.stopListening(this.stageDocument, Mojo.Event.stageDeactivate,
        this.deactivateStageHandler);
};

// Update scene contents, using render to insert the object into an HTML template
DashboardAssistant.prototype.displayDashboard = function(title, message, count) {
    var info = {title: title, message: message, count: count};
    var renderedInfo = Mojo.View.render({object: info,
      template: "dashboard/item-info"});
    var infoElement = this.controller.get("dashboardinfo");
    infoElement.innerHTML = renderedInfo;
};

// Update dashboard scene contents - external method
DashboardAssistant.prototype.updateDashboard = function(selectedFeedIndex) {
    this.index = selectedFeedIndex;
    this.title = this.list[this.index].title;
    this.message = this.list[this.index].stories[0].title;
    this.count = this.list[this.index].newStoryCount;
    this.displayDashboard(this.title, this.message, this.count);
};
```

A conventional reminder dashboard is shown in Figure 10-3 with an icon on the left, a badge indicating the number of new stories, and the title and new story headline on the right.

All of the presentation for this dashboard is in the template, so the scene's view file is minimal. Included in *views/dashboard/dashboard-scene.html*:

```
<div id="dashboardinfo" class="dashboardinfo"></div>
```

A template is defined in *views/dashboard/item-info.html* with the icon container (`palm-dashboard-icon-container`), the badge (`dashboard-new-item` and the custom class `dashboard-icon-news`), the title area (`dashboard-title`), and the message area (`dashboard-text`):

```
<div class="dashboard-notification-module">
    <div class="palm-dashboard-icon-container">
        <div class="dashboard-newitem">
            <span>#{count}</span>
        </div>
        <div id="dashboard-icon" class="palm-dashboard-icon dashboard-icon-news">
        </div>
    </div>
    <div class="palm-dashboard-text-container">
        <div class="dashboard-title">
          #{title}
        </div>
       <div id='dashboard-text' class="palm-dashboard-text">#{message}</div>
    </div>
</div
```

Figure 10-3. A dashboard summary

These are all supported framework styles (most `dashboard` named styles are loaded by the framework), but we'll add the `dashboard-icon-news` in *News.css* to show the News icon, which is located in the News application's *images* directory:

```
.dashboard-icon-news {
  background: url(../images/dashboard-icon-news.png);
}
```

This is a simple notification reminder, but dashboard summaries can be fully dynamic application views. You can provide ambient information that is accessed intermittently, like weather, stocks, baseball scores, or any information that a user wants to track and occasionally will tap to get more details. Dashboard summaries can be more sophisticated, like traffic monitors that not only provide tracking information, but also generate notifications to warn of traffic problems on routes of interest.

You can use dashboard summaries specifically to provide the display window for a background application, such as a location-based service or other applications that provide status and an occasional notification when an event occurs. This is a particular type of background application called a *Dashboard Application* and is covered more fully in the section "Background Applications" on page 256.

Handling Minimize, Maximize, and Tap Events

Like card stages, you can add a listener to the dashboard stage controller's document window for `Mojo.Event.stageActivate` and `Mojo.Event.stageDeactivate` events. When the user taps the notification bar, the Dashboard view is opened and all dashboard stages become maximized. When the user taps away or gestures back, the Dashboard view is closed and all dashboard stages are minimized.

Add event listeners to the dashboard scene assistant's setup method:

```
this.stageDocument = this.controller.stageController.document;
this.activateStageHandler = this.activateStage.bindAsEventListener(this);
Mojo.Event.listen(this.stageDocument, Mojo.Event.stageActivate,
    this.activateStageHandler);
this.deactivateStageHandler = this.deactivateStage.bindAsEventListener(this);
Mojo.Event.listen(this.stageDocument, Mojo.Event.stageDeactivate,
    this.deactivateStageHandler);
```

Next, add handlers that will add a new feature to the dashboard. When the stage is activated, display the most recent story in the feed and start a three-second timer that, upon expiry, invokes the `showStory()` method to update the dashboard summary with a new story. The timer is reset so that the stories will rotate as long as the dashboard is activated. On deactivate, clear the timer:

```
DashboardAssistant.prototype.activateStage = function() {
    Mojo.Log.info("Dashboard stage Activation");
    this.storyIndex = 0;
    this.showStory();
};

DashboardAssistant.prototype.deactivateStage = function() {
    Mojo.Log.info("Dashboard stage Deactivation");
    this.stopShowStory();
};

// Tap to Dashboard should relaunch applications
DashboardAssistant.prototype.launchMain = function() {
    Mojo.Log.info("Tap to Dashboard");
```

```
    this.controller.serviceRequest('palm://com.palm.applicationManager',
     {
       method: "open",
       parameters: {
         id: "com.palm.app.news",
         params: {}
        }
      }
    );

    this.controller.window.close();
};

// showStory - rotates stories shown in dashboard panel, every 3 seconds.
//   Only displays unread stories
DashboardAssistant.prototype.showStory = function() {
    Mojo.Log.info("Dashboard Story Rotation", this.timer, this.storyIndex);

    this.interval = 3000;
    //  If timer is null, just restart the timer and use the most recent story
    //  or the last one displayed;
    if (!this.timer)    {
        this.timer = this.controller.window.setInterval(this.showStory.bind(this),
            this.interval);
    }

    // Else, get next story in list and update the story in the dashboard display.
    else {
        // replace with test for unread story
        this.storyIndex = this.storyIndex+1;
        if(this.storyIndex >= this.list[this.index].stories.length) {
            this.storyIndex = 0;
        }

        this.message = this.list[this.index].stories[this.storyIndex].title;
        this.displayDashboard(this.title, this.message, this.count);
    }
};

DashboardAssistant.prototype.stopShowStory = function() {
    if (this.timer) {
        this.controller.window.clearInterval(this.timer);
        this.timer = undefined;
    }
};
```

You will also want to handle taps to the Dashboard panel by calling your application's main entry point and closing the dashboard window, which closes the stage. In the code above, a listener is added to the `dashboardinfo` div for any tap events. The handler uses the Application Manager to call the News entry point.

In the next section, we'll cover handling launch requests in the application assistant. In those cases, you won't use the Application Manager service; instead, you will call the `handleLaunch()` method of the application controller.

Advanced Applications

We've pushed the simple model of single application stage far enough to incorporate services, notifications, and even dashboard stages, but we need to move to a more advanced model to access the rest of the features. An advanced application will have some or all of these characteristics:

- Use an application assistant as the main application entry point and for handling application initialization and coordination.
- Create a primary card stage when launched.
- Handle relaunch or remote launch requests through a defined `handleLaunch` method.
- Post banner notifications and maintain a dashboard panel for events while not in focus or in the background.
- Schedule wakeup requests through the Alarm service and handle the alarm callbacks in the background.

If you aren't clear on the application lifecycle or the role of the application assistant, you may want to review Chapter 2 before reading the rest of this chapter.

Back to the News: App Assistant

This chapter began with a list of guidelines for developing multistage applications. News needs to be cleaned up to conform to those guidelines, so before creating the app assistant, we'll make these changes to News:

1. Remove use of the global window object; change `window.setInterval()` to `this.controller.window.setInterval()`.
2. Use the local controller's `stageController` methods; instead of `Mojo.Controller.stageController` methods for `pushScene`, we'll use `swapScene`, as in this example in *storyView-assistant.js* in the `handleCommand` method:

```
case "do-viewNext":
Mojo.Controller.stageController.swapScene(
        {
            transition: Mojo.Transition.crossFade,
            name: "storyView"
        },
        this.storyFeed, this.storyIndex+1);
break;
```

3. Add the `noWindow` property to *appinfo.json*:

```
{
    "title": "News",
    "type": "web",
    "main": "index.html",
    "id": "com.palm.app.news",
```

```
    "version": "1.0.0",
    "vendor": "Palm",
    "noWindow" : "true",
    "icon": "icon.png",
    "theme": "light"
}
```

4. And add the app assistant to *sources.json*; remember that it must be the first entry:

```
[
  {
    "source": "app/assistants/app-assistant.js"
  },
  {
    "source": "app/assistants/stage-assistant.js"
  },

 .
 .
 .

]
```

We'll create a minimal application assistant first, and then flesh it out step by step so that you can see each part clearly. Initially, we'll simply move all the code from the stage assistant to *app-assistant.js*, removing the call to `this.controller.pushScene()` and adding the `handleLaunch` method to create the card stage and push the first scene:

```
/*  AppAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Responsible for app startup, handling launch points and updating news feeds.
    Major components:
    - setup; app startup including preferences, initial load of feed data
        from the Depot and setting alarms for periodic feed updates
    - handleLaunch; launch entry point for initial launch, feed update
        alarm, dashboard or banner tap
    - handleCommand; handles app menu selections

    Data structures:
    - globals; set of persistant data used throughout app
    - Feeds Model; handles all feedlist updates, db handling and default data
    - Cookies Model; handles saving and restoring preferences

    App architecture:
    - AppAssistant; handles startup, feed list management and app menu management
    - FeedListAssistant; handles feedList navigation, search, feature feed
    - StoryListAssistant; handles single feed navigation
    - StoryViewAssistant; handles single story navigation
    - PreferencesAssistant; handles preferences display and changes
    - DashboardAssistant; displays latest new story and new story count

*/

//  ------------------------------------------------------------
```

```
//    GLOBALS
//  ------------------------------------------------------------

//  News namespace
News = {};

// Constants
News.unreadStory = "unReadStyle";
News.versionString = "1.0";
News.MainStageName = "newsStage";
News.DashboardStageName = "newsDashboard";
News.errorNone =    "0";                    // No error, success
News.invalidFeedError = "1";                // Not RSS2, RDF (RSS1), or ATOM

// Global Data Structures

// Persistent Globals - will be saved across app launches
News.featureFeedEnable = false;              // Enables feed rotation
News.featureStoryInterval = 5000;           // Feature Interval (in ms)
News.notificationEnable = true;             // Enables notifcations
News.feedUpdateBackgroundEnable = false;    // Enable device wakeup
News.feedUpdateInterval = 900000;           // Feed update interval

//  Session Globals - not saved across app launches
News.feedListChanged = false;               // Triggers update to Depot db
News.feedListUpdateInProgress = false;      // Feed update is in progress
News.featureStoryTimer = null;              // Timer for story rotations
News.dbUpdate = "";                         // Default is no update
News.wakeupTaskId = undefined;              // Id for wakeup tasks

// Setup App Menu for all scenes; all menu actions handled in
//  AppAssistant.handleCommand()
News.MenuAttr = {omitDefaultItems: true};

News.MenuModel = {
    visible: true,
    items: [
        {label: "About News...", command: "do-aboutNews"},
        Mojo.Menu.editItem,
        {label: "Update All Feeds", checkEnabled: true, command: "do-feedUpdate"},
        {label: "Preferences...", command: "do-newsPrefs"},
        Mojo.Menu.helpItem
    ]
};

function AppAssistant (appController) {

}

//  ------------------------------------------------------
//  setup - all startup actions:
//    - Setup globals with preferences
//    - Set up application menu; used in every scene
//    - Open Depot and use contents for feedList
//    - Initiate alarm for first feed update
```

```
AppAssistant.prototype.setup = function() {

    // initialize the feeds model
    this.feeds = new Feeds();
    this.feeds.loadFeedDb();

    // load preferences and globals from saved cookie
    News.Cookie.initialize();

    // Set up first timeout alarm
    this.setWakeup();

};

//  ------------------------------------------------------
//  handleLaunch - called by the framework when the application is asked to launch
//    - First launch; create card stage and first first scene
//    - Update; after alarm fires to update feeds
//    - Notification; after user taps banner or dashboard
//
AppAssistant.prototype.handleLaunch = function (launchParams) {
    Mojo.Log.info("ReLaunch");

    var cardStageController = this.controller.getStageController(News.MainStageName);
    var appController = Mojo.Controller.getAppController();

    if (!launchParams)  {
        // FIRST LAUNCH
        // Look for an existing main stage by name.
        if (cardStageController) {
            // If it exists, just bring it to the front by focusing its window.
            Mojo.Log.info("Main Stage Exists");
            cardStageController.popScenesTo("feedList");
            cardStageController.activate();
        } else {
            // Create a callback function to set up the new main stage
            // once it is done loading. It is passed the new stage controller
            // as the first parameter.
            var pushMainScene = function(stageController) {
                stageController.pushScene("feedList", this.feeds);
            };
            Mojo.Log.info("Create Main Stage");
            var stageArguments = {name: News.MainStageName, lightweight: true};
            this.controller.createStageWithCallback(stageArguments,
                pushMainScene.bind(this), "card");
        }
    }
};

// ----------------------------------------
// handleCommand - called to handle app menu selections
//
     .
     .
```

```
    .
};
```

Handling Launch Requests

The framework calls the application assistant's `handleLaunch` method after the `setup` method on initial launch, and whenever a launch request is made to the application. If you don't define one, the framework attaches a default `handleLaunch` method, which calls your application assistant's `setup` method. Launch requests are made implicitly through the following:

- Taps to your application's banner notifications.
- Calls from other applications through an Application Manager service request.
- Alarms that wake up the application after a timeout.

By convention, you should also use this entry point for your own launch requests. For example, instead of using the Application Manager service request, launch your application after a tap to the Dashboard stage by calling the entry point directly from within the `handleTap` method in the Dashboard, passing an `action` property in the `launchParams` object:

```
DashboardAssistant.prototype.launchMain = function() {
    Mojo.Log.info("Tap to Dashboard");
    var appController = Mojo.Controller.getAppController();
    appController.assistant.handleLaunch({action: "notification",
      index: this.index});
    this.controller.window.close();
};
```

And we'll add a specific case in `handleLaunch` for this notification action following the conditional set up to handle the first launch. We use a case statement because we'll build on this to handle other launch actions:

```
//  ------------------------------------------------------
//  handleLaunch - called by the framework when the application is asked to launch
//    - First launch; create card stage and first first scene
//    - Update; after alarm fires to update feeds
//    - Notification; after user taps banner or dashboard
//
AppAssistant.prototype.handleLaunch = function (launchParams) {
    Mojo.Log.info("ReLaunch");

    var cardStageController = this.controller.getStageController(News.MainStageName);
    var appController = Mojo.Controller.getAppController();

    if (!launchParams)  {
        // FIRST LAUNCH
        // Look for an existing main stage by name.
        if (cardStageController) {
            // If it exists, just bring it to the front by focusing its window.
            Mojo.Log.info("Main Stage Exists");
```

```
            cardStageController.popScenesTo("feedList");
            cardStageController.activate();
        } else {
            // Create a callback function to set up the new main stage
            // once it is done loading. It is passed the new stage controller
            // as the first parameter.
            var pushMainScene = function(stageController) {
                stageController.pushScene("feedList", this.feeds);
            };
            Mojo.Log.info("Create Main Stage");
            var stageArguments = {name: News.MainStageName, lightweight: true};
            this.controller.createStageWithCallback(stageArguments,
                pushMainScene.bind(this), "card");
        }
    }
    else  {
        Mojo.Log.info("com.palm.app.news -- Wakeup Call", launchParams.action);
        switch (launchParams.action) {

        // NOTIFICATION
        case "notification"  :
             Mojo.Log.info("com.palm.app.news -- Notification Tap");
            if (cardStageController) {

                // If it exists, find the appropriate story list and activate it.
                Mojo.Log.info("Main Stage Exists");
                cardStageController.popScenesTo("feedList");
                cardStageController.pushScene("storyList", this.feeds.list,
                  launchParams.index);
                cardStageController.activate();
            } else {

                // Create a callback function to set up a new main stage,
                // push the feedList scene and then the appropriate story list
                var pushMainScene2 = function(stageController) {
                    stageController.pushScene("feedList", this.feeds);
                    stageController.pushScene("storyList", this.feeds.list,
                      launchParams.index);
                };
                Mojo.Log.info("Create Main Stage");
                var stageArguments2 = {name: News.MainStageName,
                  lightweight: true};
                this.controller.createStageWithCallback(stageArguments2,
                  pushMainScene2.bind(this), "card");
            }
        break;

        }
    }
};
```

The notification launch case is added to activate the News main stage with the `storyList` scene pushed to the selected feed. If the stage doesn't exist, it will be created before pushing the `storyList` scene. This type of launch request will launch the application if it's not already launched.

Looking at this a little more closely, if the stage exists, the News application's main stage is in the card view, so the scene stack is popped back to the `feedList` scene, which is always at the base of the scene stack for this application. The `storyList` scene is pushed with the feed that was displayed by the dashboard summary. If the main card stage does not exist (just the dashboard is running), the stage is created before pushing both the `feedList` and `storyList` scenes. It's important to set up the scene stack when launching to a scene that is normally not at the base of the stack.

This launch action is initiated by tapping either the dashboard summary (from the code sample shown immediately prior to this application assistant sample) and the banner notification. If you look at the *feed.js* method for `updateFeedSuccess()`, you'll see that the launch argument's `action` property is set to `"notification"`:

```
appController.showBanner(bannerParams, {action: "notification",
 index: this.feedIndex},
  this.list[this.feedIndex].url);
```

Sending and Considering Notifications

To facilitate communication between assistants, the Mojo framework supports an application specific notification chain. Any application assistant can pass a notification through the chain through a call to `SendToNotificationChain()` with a single hash parameter. The current stage and scene assistants have the opportunity to handle these notifications by including a `considerForNotification()` method.

To illustrate this, we'll send an update notification (`type: "update"`) through the chain, identifying that an update is in progress (`update : true`) or just completed (`update : false`), as well as the index of the affected feed (`feedIndex : this.feedIndex`). This is done in several places in *feeds.js*, wherever we were updating the `spinnerModel` and calling `this.updateListModel()`. Replace this code:

```
// Change feed update indicator & update widget
var spinnerModel = this.list[this.feedIndex];
spinnerModel.value = true;
this.updateListModel();
```

With this code:

```
// Notify the chain that there is an update in progress
Mojo.Controller.getAppController().sendToNotificationChain({
  type: "update",
  update: true,
  feedIndex: this.feedIndex
});
```

That code is used in three other places. The `update` property should be set to `true` wherever you were previously setting the `spinnerModel.value` to `true`, and `false` in the other cases. You can remove the `registerListModel()`, `removeListModel()`, and `updateListModel()` methods in *feeds.js* and remove the calls to those methods from the *feedList-assistant.js* `activate()` and `deactivate()` methods.

Scenes can receive notifications if you add a `considerForNotification()` method to the scene assistant. In the News example, we'll add this to the *feedList-assistant.js*:

```
// -------------------------------------------------------------------------
// considerForNotification - called by the framework when a notification
//  is issued; look for notifications of feed updates and update the
//  feedWgtModel to reflect changes, update the feed's spinner model
FeedListAssistant.prototype.considerForNotification = function(params){
    if (params && (params.type == "update"))    {
        this.feedWgtModel.items = this.feeds.list;
        this.feeds.list[params.feedIndex].value = params.update;
        this.controller.modelChanged(this.feedWgtModel);

        // If stories exist in the this.featureIndexFeed, then start the rotation
        //  if not already started
// ** These next two lines are wrapped for book formatting only **
        if ((this.feeds.list[this.featureIndexFeed].stories.length > 0) &&
         (News.featureStoryTimer === null)) {
            var splashScreenElement = this.controller.get("splashScreen");
            splashScreenElement.hide();
            this.showFeatureStory();
        }
    }
    return undefined;
};
```

This method is called on any notification, but on update notifications it will set the affected feed's spinner value to reflect whether an update is in progress or not, update the feed list, and start the feature story timer if needed.

In *storyList-assistant.js*, it's used to look for changes to the displayed feed:

```
// considerForNotification - called when a notification is issued; if this
// feed has been changed, then update it.
StoryListAssistant.prototype.considerForNotification = function(params){
    if (params && (params.type == "update"))    {
        if ((params.feedIndex == this.feedIndex) && (params.update === false)) {
            this.storyModel.items = this.feed.stories;
            this.controller.modelChanged(this.storyModel);
        }
    }
    return undefined;
};
```

Among the scenes, only the active scene's `considerForNotification()` method is called. That's followed by calls to the active stage assistant and finally the application assistant. Since the application assistant is always the last on the chain, it can process what remains once the other assistants have had their chance at the notification block.

Back to the News: Creating Secondary Cards

In this final example, we'll create a secondary card stage by adding the option to push a single feed into its own card. Add a "New Card" item to the pop-up submenu, which

displays when the user taps the unread count on a specific feed in the list. When that item is tapped, it will push that feed into a new card. Figure 10-4 shows the new submenu and the card view showing the main `feedList` and the secondary card.

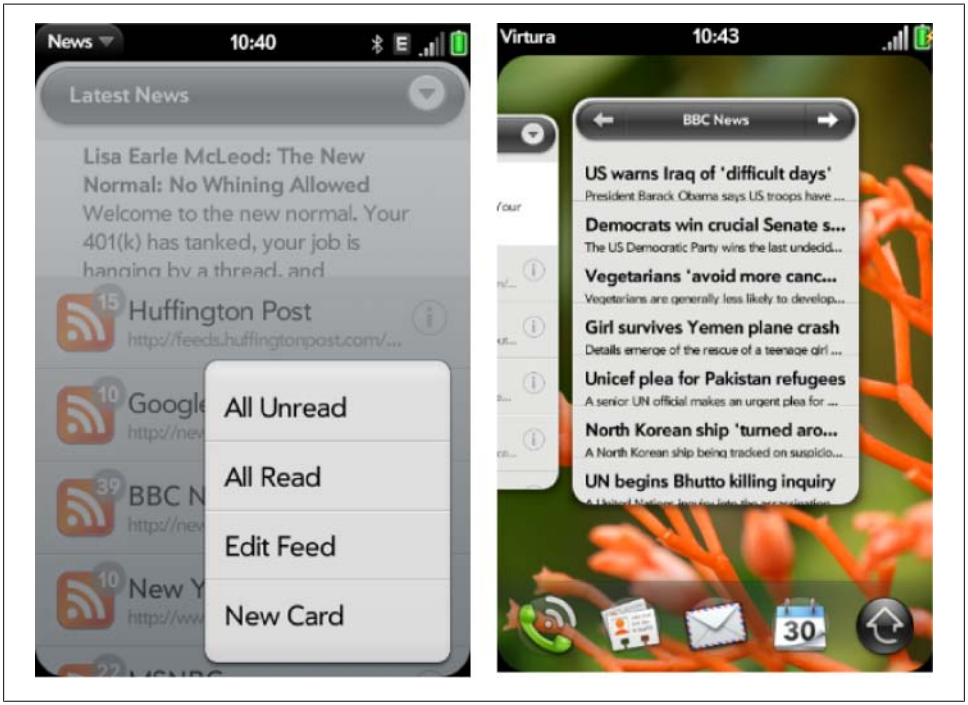


Figure 10-4. The secondary card

Secondary card stages are created like other stages. Here's the case statement from the `popupHandler` method in *feedList-assistant.js*, triggered by the New Card submenu selector:

```
case "feed-card":
    Mojo.Log.info("Popup tear off feed to new card:",
      popupFeed.title);

    var newCardStage = "newsCard"+this.popupIndex;
    var cardStage = this.appController.getStageController(newCardStage);
    var feedList = this.feeds.list;
    var feedIndex = this.popupIndex;
    if(cardStage) {
        Mojo.Log.info("Existing Card Stage");
        cardStage.popScenesTo();
        cardStage.pushScene("storyList", this.feeds.list, feedIndex);
        cardStage.activate();
    } else  {
        Mojo.Log.info("New Card Stage");
        var pushStoryCard = function(stageController){
```

```
            stageController.pushScene("storyList", feedList,
              feedIndex);
        };
        this.appController.createStageWithCallback({
            name: newCardStage, lightweight: true},
            pushStoryCard, "card");
    }
    break;
```

The stage name must be unique unless you plan to reuse the same stage for each card; in this example, we use the feed index to form part of the stage name to keep it unique. When reusing the stage, `popScenesTo()` is used with `pushScene()` and `activate()` methods to maximize the stage with the `storyList` scene.

To enable this command option, add another choice to the submenu in the `feedList` assistant's `showFeed()` method:

```
{label: "New Card", command: "feed-card"}
```

When users select this option by tapping the info icon on any feed in the feed list, a `storyList` scene will be pushed with the selected feed in its own card.

Background Applications

We've done most of the work to make a background application, and in some ways, News is already running in the background. Background applications could be any of these types:

Minimized application
: Has a card stage (or window), but is not in the foreground view.

Dashboard application
: Has a dashboard stage (or panel), but no card stage.

Background application
: Has neither a card nor a dashboard stage, but wakes periodically through an alarm and can issue notifications and create dashboards or cards when appropriate.

Minimized and dashboard applications should use conventional JavaScript timers, such as `setTimeout()` or `setInterval()`, to schedule recurring actions, for instance checking for new articles. In its current form, News uses `setTimeout()` and runs while minimized. You'll notice if you close the main card stage while there is a News dashboard panel, the application will run in the background, posting notifications when feeds are updated with new stories. It will not perform these updates once the device goes to sleep.

Background applications can run without a window, and can wake the device from sleep or across boots by using the Alarm service. Since the framework will close any application unless there is an open window, there isn't another option for this type of application.

We'll replace the `setTimeout()` timer with an alarm set in the `setWakeup()` method in *app-assistant.js*:

```
// ------------------------------------------------------------------------
// setWakeup - called to setup the wakeup alarm for background feed updates
//   if preferences are not set for a manual update (value of "00:00:00")
AppAssistant.prototype.setWakeup = function() {
        if (News.feedUpdateInterval !== "00:00:00")   {
            this.wakeupRequest =
              new Mojo.Service.Request("palm://com.palm.power/timeout"
              , {
                method: "set",
                parameters: {
                    "key": "com.palm.app.news.update",
                    "in": News.feedUpdateInterval,
                    "wakeup": News.feedUpdateBackgroundEnable,
                    "uri": "palm://com.palm.applicationManager/open",
                    "params": {
                        "id": "com.palm.app.news",
                        "params": {"action": "feedUpdate"}
                    }
                },
                onSuccess:  function(response){
                    Mojo.Log.info("Alarm Set Success", response.returnValue);
                    News.wakeupTaskId = Object.toJSON(response.taskId);
                },
                onFailure:  function(response){
                    Mojo.Log.info("Alarm Set Failure",
                        response.returnValue, response.errorText);
                }
            });
           Mojo.Log.info("Set Update Timeout");
        }
};
```

You might want to refer back to Chapter 9, where the Alarm service is reviewed in detail. In this case, we set up the alarm for the specified `News.feedUpdateInterval` and requested that the alarm wake the device by setting the `wakeup` property to `true`.

The Alarm service will not accept any relative alarm values of less than five minutes.

To field the update, we'll add another action handler in the application assistant's `handleLaunch` method:

```
switch (launchParams.action) {

        .
        .
        .
```

```
        // UPDATE FEEDS
        case "feedUpdate"  :
            // Set next wakeup alarm
            this.setWakeup();

            // Update the feed list
            Mojo.Log.info("Update FeedList");
            this.feeds.updateFeedList();
        break;
        }
    }
```

This is pretty straightforward, and will work as long as the device is awake when the alarm fires. But when it is sleeping, the application will only have a few seconds before the power management system will force the device back to sleep.

A background application that receives an alarm when the device is asleep will have less than five seconds before being shut down again. If you need more time than this, you should use `activityStart()` and `activityStop()` methods to prevent the device from sleeping until your activity has completed. Refer to Chapter 9 for more information on these service methods.

Five seconds isn't enough time for News to complete a full update of all the feeds during a single alarm wakeup cycle. But the update process is structured to resume with the next feed to be updated, so in these cases, News will do a full update over several cycles.

To allow the user full control of the application's background behavior, we'll add some additional preferences features:

Manual updates
: In addition to the update intervals from five minutes to one day, we'll add an option for manual updates only which will disable the background updates.

Wakeup enable/disable
: A toggle to turn off the option of waking up the device during background updates.

Notification enable/disable
: A toggle to turn notifications on or off. When set to off, the feed updates will be carried out, but without any notifications or dashboard updates.

If you're interested in seeing the final version of the application assistant, you should review the full code listing for *app-assistant.js* in Appendix D.

Guidelines for Background Applications

Background applications are very powerful, but they can also easily overuse resources and hurt the user experience. Most of these guidelines are common sense, but they are critical to making your application successful.

- When minimized, suspend application behavior that isn't necessary and lengthen polling cycles. Since the application is minimized, updates are not immediately visible, so work done to update the display is a waste of resources.
- When your application is minimized or in the background in any way, limit system calls, data connections, and similar requests. They consume CPU and power, and lengthy operations will impact the responsiveness of the maximized application.
- Always provide a way to close the application and terminate any background activity.
- Whenever possible, avoid waking up the device.
- Always give users the option of not waking the device; they should be able to run the application and experience some background activity when the device is awake, but not when the device sleeps.
- Set alarm and timeout intervals as long as possible.
- When using alarms in a recurring fashion, meaning after each alarm you set up a new alarm, always provide a simple way for the user to turn off the alarms or completely close the application so that it doesn't run indefinitely.
- Conserve power; poll as infrequently as possible. Space out your requests and implement degrading intervals that lengthen if your polling does not produce an event or data change.
- Tasks that are scheduled for when the device is in mass storage mode will not be suspended. You will need to detect when the device goes into mass storage mode and restart the task. If the timeout expires when the device is in mass storage mode, the callback will not be made.
- Limit notifications; use the dashboard to update state and status, and limit even banner notifications to important information.

Summary

Whether you're interested in building an advanced application or just want to add notifications to a basic application, you'll find some essential topics in this chapter. After a broad review of advanced multistage applications, with an introduction to notifications and dashboards, you learned that advanced applications are based on an application assistant, which can handle external launch requests and potentially run in the background. You were also shown how to customize your application's behavior when minimized, meaning switched out of the foreground view, and how to use the internal application notification chain to coordinate actions between assistants or share events and data.

With the techniques in this chapter, you should be able to move your application to the background, build a dashboard-only application, or use secondary card stages to support separate activities.

CHAPTER 11

Localization and Internationalization

The Palm webOS platform was designed from the beginning to be a world-ready system, from the choice of OS technologies through the UI design. While it may take some time to support all languages and regions, and to provide the application content to meet the needs of users in all locales, the framework has the basic support you need to build global applications.

In this chapter, you will get an overview of the framework's locale support and learn how to localize your application. We will localize the News application to Spanish and we will walk through each step of the localization process. In the final section, we'll cover some of the Internationalization APIs available in Mojo.

Users can switch languages and regions at runtime using a language preferences application, shown in Figure 11-1. Users can select from any of the languages and any of the regions, thereby creating any locale formed by those combinations.

The system does not dynamically switch languages; it must do a soft reset of the application environment, which closes any running applications and restarts the system UI with the newly selected locale.

Locales

Palm webOS defines a locale conventionally as a combination of language and region, and initially includes support for some Latin-1 languages and related regions. The first products will include all North and South American languages and regions as well as some of the Western European languages and regions.

Figure 11-1. A language preferences application

The choice of language indicates the primary localization, while the regional settings govern date formats, number formats, and similar types of data representation. You can mix any language and any region to create a locale. For example, *en_DE* is the English language and German regional settings. A complete list of supported languages and keyboard mappings is provided in Table 11-1, with some of the more common locales and regions.

Table 11-1. Supported Languages and Regions

Locale	Language	Region	Keyboard
en_US	English (en)	United States (US)	QWERTY
en_GB	English (en)	Great Britain (GB)	QWERTY
en_IE	English (en)	Ireland (IE)	QWERTY
es_US	Spanish (es)	United States (US)	QWERTY
es_ES	Spanish (es)	Spain (ES)	QWERTY
en_CA	English (en)	Canada (CA)	QWERTY
fr_CA	French (fr)	Canada (CA)	AZERTY

Locale	Language	Region	Keyboard
de_DE	German (de)	Germany (DE)	QWERTZ
it_IT	Italian (it)	Italy (IT)	QWERTY
fr_FR	French (fr)	France (FR)	AZERTY

The architecture is capable of supporting most single-byte and double-byte locales, but the initial release does not include the necessary fonts, input methods, and some of the text-processing utilities needed to fully support those locales. Additional support will be provided over time, but availability will depend upon regional business priorities. If you follow the techniques discussed in this chapter, your application should be ready to support those locales when available.

There are many more regions supported than those shown in Table 11-1, and additional regions are added frequently. You'll find the current list on the Palm developer site.

Character Sets and Fonts

The initial Palm webOS devices ship with a Unicode UTF-8 character set, primarily supported by the Prelude font, which was described from a style and design perspective in Chapter 7. Prelude supports the following character sets, as defined by Windows codepages:

- 1250 (Eastern Europe)
- 1251 (Cyrillic)
- 1252 (Latin 1 or Western Europe)
- 1253 (Greek)
- 1254 (Turkish)
- 1257 (Baltic)

Additional fonts are included to support conventional browser content. The browser fonts support the character sets named above, plus all the characters used for Japanese, Chinese, Korean, and Vietnamese.

Keyboards

The keyboards will track the available locales. For example, QWERTY, QWERTZ, and AZERTY configurations are provided with the locales as described in Table 11-1. This is not something that will be uniform across webOS devices, so while some form of keyboard or input method support is provided, you cannot expect a typical configuration.

Global Applications

For many developers, creating a global application means localizing your application to the locales in which it will be used, and supporting display formats for any locale. If you are diligent, you will use locale-sensitive formatting and sorting with text, number, percent, and currency strings, and will accommodate global requirements for addresses, phone numbers, and similar data.

But global applications should deal with regional and language requirements at a deeper level, driving the content and features themselves. You should:

- Use locale-specific content for default data or other data sources. In our example, the default News feeds should be locale-specific. The feeds of interest to the North American English speaker are quite different from those of a European Italian speaker or a South American Spanish speaker.
- Make your content appropriate to the user's location. Implement the spirit of the function, not just literal support. For example, a sports-oriented application needs to focus on popular sports based on regional popularity and interest.
- Use regional data servers; select web services or servers based on the user's locale or location.

Consider your application's feature set and the data content when thinking about your application as a product with a worldwide user base.

Localization

The system is localized for the languages and locales offered as options in the Language picker in any given release of Palm webOS. You can localize your application to any of the available locales and the correct localization will be selected when the application is launched. If your application doesn't support the selected locale, it will try to match just the language. If there is no match for the language, the base version will be selected.

To localize your application, create locale-named directories within your application to hold the localized files. The framework will look for a directory that matches the current locale and will automatically substitute the localized content for the content in the base version.

You can localize any of the following:

Application name and icon
: Each locale can have a separate version of the application's *appinfo.json*, which can be modified for that locale to customize any of the properties defined there.

JavaScript strings
: Any string appearing in JavaScript executed by the application can be extracted and localized by encapsulating the string with the `$L()` function.

HTML scene or template
: Each locale has a *views* directory, which is structured like the base version *views* directory and can contain any of the base version HTML files that you wish to modify for that locale.

HTML strings
: Any string appearing in an HTML file or template can be localized by creating a copy of the file and modifying the strings.

Typically, you will develop your application with a base version and once you reach UI freeze or the UI is complete, you will do a test localization with a pseudolocale to confirm that you have identified all the strings and that your application structure is correct.

You can localize to multiple languages in parallel, but once you've done your localization, you will need to manage code or UI changes. This may be difficult under the current structure, as the tools don't address change management, forcing you to manually migrate all changes.

It's good to do a pseudolanguage localization early to test localization readiness, but wait on actual localization until your UI is final or very close to it.

Localized Application Structure

Localized applications have an additional directory, *resources*, at the application's root directory, and within it are directories for each locale. Within *resources* are locale directories that include a localized version of *appinfo.json* and localization files for JavaScript string translation (*strings.json*), and localized versions of the application's view files. Figure 11-2 shows an example of an exploded *resources* directory for the News application, which has a single localization for United States Spanish.

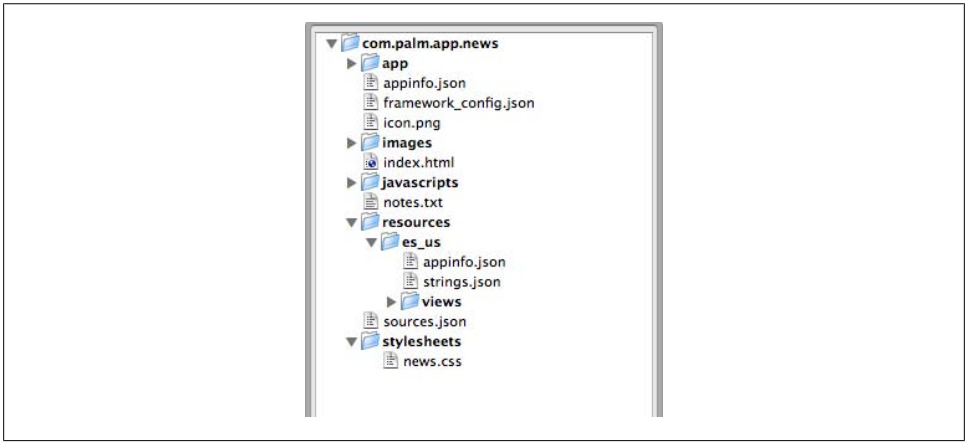


Figure 11-2. An example resources directory

You'll see that the first level of directories within *resources* is for locales. If the locale is set to *es_US*, the framework will look for content in the *resources/es_us* directory first. If that directory or the content (e.g., *appinfo.json*, specific strings in *strings.json*, HTML files in */views*) is not there, it will then default to the *app* directory contents.

You can create this structure manually by creating and naming the directories with the valid language and region codes. Table 11-1 defines the initially supported languages and regions, but check the SDK documentation for the current set of supported names. All of the language names are encoded as a two-letter ISO 639 language code and all of the region names are based on a two-letter ISO 3166 country code.

At the time of this writing, there are no tools provided in the SDK to assist with creating or maintaining localization content, but tools are planned. Check the SDK site for any new information on localization tools.

appinfo.json

Each locale directory will include a copy of the main *appinfo.json* file, modified for localization and for file path changes. For the News application, the title is changed to "Noticias" and the main and icon property values are modified to adjust the relative path for the location of this version of *appinfo.json*. The file path must reflect that this version of *appinfo.json* is located at *com.palm.app.news/resources/es_us/appinfo.json* and the target files are located at *com.palm.app.news*:

```
{
  "title": "Noticias",
  "type": "web",
  "main": "../../index.html",
  "icon": "../../icon.png",
  "id": "com.palm.app.news",
  "version": "1.0",
  "vendor": "Palm",
  "theme": "light",
  "noWindow": "true"
}
```

We only localized the application name here, but you could also do the following:

- Change the icon by pointing to a different icon image file. The *appinfo.json* file is encoded in UTF-8.
- Use a localized *index.html*, which might be useful for including different stylesheets for a given localization or for changing page layout based on writing direction or cultural reasons.

JavaScript Text Strings

The Mojo framework will dynamically substitute localized strings based on the current locale. Any JavaScript string can be localized; you should localize all text strings that

are displayed and are visible to the user, but not those that never appear. Examples of strings that should not be localized include logging information or information used strictly as internal data or arguments. Nor should you localize strings that don't include text, such as HTML templates that contain only variable definitions.

Identifying strings

You prepare for string extraction and translation by encapsulating any target string with the `$L()` function, as in this example from *app-assistant.js* in News, where the default application menu attributes and model are set up:

```
// Setup App Menu for all scenes; all menu actions handled in
//  AppAssistant.handleCommand()
News.MenuAttr = {omitDefaultItems: true};

News.MenuModel = {
    visible: true,
    items: [
        {label: $L("About News..."), command: "do-aboutNews"},
        Mojo.Menu.editItem,
        {label: $L("Update All Feeds"), checkEnabled: true, command: "do-feedUpdate"},
        {label: $L("Preferences..."), command: "do-newsPrefs"},
        Mojo.Menu.helpItem
    ]
};
```

Or in this case, from *preferences-assistant.js*, where the list selector widget is set up:

```
// Setup list selector for UPDATE INTERVAL
    this.controller.setupWidget("feedCheckIntervalList",
        {
            label: $L("Interval"),
            choices: [
                {label: $L("Manual Updates"),   value: "00:00:00"},
                {label: $L("5 Minutes"),        value: "00:05:00"},
                {label: $L("15 Minutes"),       value: "00:15:00"},
                {label: $L("1 Hour"),           value: "01:00:00"},
                {label: $L("4 Hours"),          value: "04:00:00"},
                {label: $L("1 Day"),            value: "23:59:59"}
            ]
        },
        this.feedIntervalModel = {
            value : News.feedUpdateInterval
        });
```

The framework will use any string that is encapsulated with `$L()` as a key in the appropriate `strings.json` file (based on the current locale) and substitute the resulting value (the localized string) in the original string's place.

In this next example, the string arguments to `Mojo.Log.warn()` are not encapsulated, but the arguments to `Mojo.Controller.errorDialog()` are:

```
Mojo.Log.warn("Can't open feed database: ", result);
Mojo.Controller.errorDialog($L("Can't open feed database : ")+ result);
```

Sometimes translations can change the position of variables within the strings. If a variable is inserted within a string, use templates to allow for changes in position. For example:

```
Mojo.Log.warn("Can't open feed database (#", result, "). All feeds will be reloaded.");

// ** These next two lines are wrapped for book formatting only **
var message = new Template($L("Can't open database (#{message}).
   Feeds will be reloaded."));
Mojo.Controller.errorDialog(message.evaluate({message: result}));
```

This is even more critical if you have strings where there are multiple variables. Localization can change not only the location, but also the order of variables within strings. For example:

```
"Not enough memory to #{action} the file #{fname}."
```

becomes (in Finnish):

```
"Liian vähän muistia tiedoston #{fname} #{action}."
```

Also, be careful about reusing keys where the same original string might have multiple translations in another language. For example, "save" can be translated in one context to "speichern" in German. In another, it could be translated as "retten." Likewise, the English word "add" can mean "append" in one context, and "sum" in another context. When using keys in different contexts, consider using different identifiers in your source documents, even if they both have the same localization into English.

Extracting strings

Once the strings are identified, you will extract them to a *strings.json* file, located in the root level of the locale directory.

You can manually extract the strings by scanning the file for `$L()` encapsulation and copying the strings into the key position in *strings.json*. If they aren't encapsulated by the `$L()` function, the framework will not perform the substitutions.

Palm recognizes that this is a very tedious process and intends to provide tool support to facilitate string extraction, but at the time of this writing, those tools are not yet available. Refer to the Palm developer site for more information about localization tools.

Localizing strings

The *strings.json* file is a conventional JSON file, encoded in UTF-8, with the base version of the string used as a key and the localized version of the string as the value:

```
"Original String" : "Localized String",
```

The News *resources/es/strings.json* is shown here:

```
{
   "#{status}" : "#{status}",
```

```
// ** These next four lines are wrapped for book formatting only **
"0##{title} : No New Items|1##{title} : 1 New Item|1>##{title} :
  #{count} New Items" :
  "0##{title} : No hay elementos nuevos|1##{title} : 1 elemento nuevo|1>##{title} :
  #{count} elementos nuevos",
"1 Day" : "1 día",
"1 Hour" : "1 hora",
"15 Minutes" : "15 minutos",
"4 Hours" : "4 horas",
"5 Minutes" : "5 minutos",
"About News..." : "Acerca de noticias...",
// ** These next three lines are wrapped for book formatting only **
"Add Feed DB save error : #{message}; can't save feed list." :
  "Error de base de datos al intentar agregar nueva fuente web :
  #{message}; no se puede guardar la lista de
    fuentes web.",
"Add News Feed Source" : "Añadir fuente web de noticias",
"Add..." : "Añadir...",
"Adding a Feed" : "Añadiendo una fuente web",
"All Read" : "Todas leídas",
"All Unread" : "Todas las no leídas",
"Can't open feed database: " : "No se puede abrir la base de datos de fuentes web: ",
"Cancel" : "Cancelar",
"Cancel search" : "Cancelar búsqueda",
"Check out this News story..." : "Leer esta noticia...",
"Check this out: " : "Mira esto: ",
"Copyright 2009, Palm Inc." : "Copyright 2009, Palm Inc.",
"Database save error: " : "Error al guardar en la base de datos: ",
"Edit a Feed" : "Editar una fuente web",
"Edit Feed" : "Editar fuente web",
"Edit News Feed" : "Editar una fuente web de noticias",
"Feature Feed" : "Fuente web destacada",
"Featured Feed" : "Fuente web destacada",
"Feature Rotation" : "Rotación de fuente web destacada",
"Feed Request Success:" : "Solicitud de fuente web lograda:",
"Feed Updates" : "Actualización de fuentes web",
"Help..." : "Ayuda...",
"Interval" : "Intervalo",
// ** These next two lines are wrapped for book formatting only **
"Invalid Feed - not a supported feed type" :
  "Fuente web no válida: no es un tipo de fuente web admitido",
"Latest News" : "Últimas noticias",
"Manual Updates" : "Actualizaciones manuales",
"Mark Read or Unread" : "Marcar leída o no leída",
"New Card" : "Tarjeta nueva",
"New features" : "Nuevas características",
"New Items" : "Elementos nuevos",
"News Help" : "Ayuda para noticias",
"News Preferences" : "Preferencias para noticias",
// ** These next two lines are wrapped for book formatting only **
"newsfeed.status" :
  "Estado #{status} devuelto desde solicitud de fuente web de noticias",
"OK" : "OK",
"Optional" : "Opcional",
"Preferences..." : "Preferencias...",
```

```
    "Reload" : "Cargar nuevamente",
    "Rotate Every" : "Girar cada",
    "Rotation (in seconds)" : "Rotación (en segundos)",
    "RSS or ATOM feed URL" : "Fuente web RSS o ATOM URL",
    "Search for:  #{filter}" : "Buscar:  #{filter}",
    "Show Notification" :"Mostrar aviso",
    "SMS/IM" : "SMS/IM",
    // ** These next two lines are wrapped for book formatting only **
    "Status #{status} returned from newsfeed request." :
      "La solicitud de fuente web de noticias indicó el estado #{status}.",
    "Stop" : "Detener",
    "Title" : "Título",
    "Title (Optional)" : "Título (Opcional)",
    "Update All Feeds" : "Actualizar todas las fuentes web",
    "Wake Device" : "Activar dispositivo",
    // ** These next two lines are wrapped for book formatting only **
    "Will need to reload on next use." :
      "Se tendrá que cargar de nuevo la próxima vez que se use."

}
```

If the original string is not appropriate as a key, the `$L()` function can be called with an explicit key:

```
$L("value":"Original String", "key": "string_key")
```

In this case, `string_key` is the key in *strings.json* and the translation of `Original String` is the value:

```
{
"string_key" : "Localized String"
}
```

Here's an example, again in News. The results template is defined with a key inside the `$L()` function:

```
AppAssistant.prototype.feedRequestSuccess = function(transport) {

  var t = new Template($L({key: "newsfeed.status",
    value: "Status #{status} returned from newsfeed request."}));
  Mojo.Log.info("com.palm.app.news - Feed Request Success: ",
    t.evaluate(transport));
```

And if you look back at the *strings.json*, you'll see this entry, where the key is used instead of the original string:

```
// ** These next two lines are wrapped for book formatting only **
  "newsfeed.status" :
    "Estado #{status} devuelto desde solicitud de fuente web de noticias",
```

Localizable HTML

The framework does not dynamically substitute localized text strings for HTML text strings. Instead you will create a copy of the HTML scenes and templates, manually substituting the localized strings. You do get some help identifying strings and

extracting them to *strings.json* to facilitate translation, but you'll need to manually copy the translated strings to your localized HTML files.

You don't have to limit the changes in the localized HTML files to localized text substitution. You may also make layout changes by locale; in some cases the translations may need some adjustments. And you can also include locale-specific CSS, which you can use in combination with your HTML to modify the presentation, either to accommodate text translations (for example, adjusting for significant changes in text length) or simply to address unique formatting requirements within a specific locale.

Identifying and extracting strings

You don't need to extract the strings from the HTML files, but instead you will copy any HTML file that includes a localized string into the views directory for each locale. The structure of the views folder will mirror the base version, but only the files with localized content will be included. For example, News will have only three localized HTML files:

views/feedList/addFeed-scene.html
views/feedList/feedList-assistant.html
views/preferences/preferences-assistant.html

As part of your localization workflow, you may want to extract the strings, and it doesn't hurt to include those strings in *strings.json*.

Localizers are very familiar with translating whole HTML files. You may want to just hand over your file to the localizers to get translated to various languages and not take time to extract the strings.

Localizing strings

After translation, the localized strings must be copied into the localized version of the HTML file. Using the previous example of *preferences-assistant.html*, the strings are translated to *es_US*:

```
<div class="palm-page-header">
    <div class="palm-page-header-wrapper">
        <div class="icon news-mini-icon"></div>
        <div class="title">Preferencias para noticias</div>
    </div>
</div>

<div class="palm-group">
    <div class="palm-group-title"><span>Fuente web destacada</span></div>
        <div class="palm-list">
            <div x-mojo-element="IntegerPicker" id="featureFeedDelay"></div>
        </div>
    </div>
</div>
```

```
<div class="palm-group">
    <div class="palm-group-title"><span>Actualización de fuentes web</span></div>
           <div class="palm-list">
            <div class="palm-row first">
                <div class="palm-row-wrapper">
                    <div x-mojo-element="ListSelector" id="feedCheckIntervalList">
                    </div>
                </div>
            </div>
            <div class="palm-row">
                <div class="palm-row-wrapper">
                    <div x-mojo-element="ToggleButton" id="notificationToggle">
                    </div>
                    <div class="title left">Mostrar aviso</div>
                </div>
            </div>
            <div class="palm-row last">
                <div class="palm-row-wrapper">
                    <div x-mojo-element="ToggleButton" id="bgUpdateToggle"></div>
                    <div class="title left">Activar dispositivo</div>
                </div>
            </div>
        </div>
    </div>
</div>
```

You can see the other HTML files along with all the localization changes made for News to support the *es_US* locale. When you launch News with that locale selected, you'll see something similar to Figure 11-3.

Figure 11-3. News localized to the es_US locale

Internationalization

The Mojo framework includes `Mojo.Format`, a set of locale-aware methods to assist you with formatting different types of text strings. The available methods are summarized in Table 11-2.

Table 11-2. Mojo.Locale and Mojo.Format methods

Method	Description
`Mojo.Locale.getCurrentLocale()`	Returns the currently set locale as an ISO 639-formatted string (e.g., en_us for US English)
`Mojo.Locale.getCurrentFormatRegion()`	Returns the currently set region as an ISO 639-formatted string (e.g., us for US English)
`Mojo.Format.formatDate()`	Formats the date object appropriately for the current locale
`Mojo.Format.formatRelativeDate()`	Formats the date object as with `formatDate()`, but returns yesterday/today/tomorrow if appropriate, or the day of the week if in the last week
`Mojo.Format.formatNumber()`	Converts a number to a string using the proper locale-based format for numbers and number separators
`Mojo.Format.formatCurrency()`	Converts a number representing an amount of currency to a string using the proper locale-based format for currency; does not do any currency rate conversion, just formatting
`Mojo.Format.formatChoice()`	Formats a choice list to handle things like replacement parameters with plurals
`Mojo.Format.formatPercent()`	Converts a number to a percent string using the proper locale-based format for percentages
`Mojo.Format.using12HrTime()`	Returns true if the current locale uses 12-hour time or false if 24-hour time
`Mojo.Format.getCurrentTimeZone()`	Returns current timezone

The behavior of some of the widgets should be influenced by the selected locale. Currently, the Time Picker widget will hide the AM/PM panel if the current locale defaults to a 24-hour time format. There will be further integration of locale-specific behavior over time; check the Palm Developer site for updates in this area.

Back to the News: Multilingual Formatting

The News' banner notification includes a phrase indicating the number of new stories, as shown in Figure 11-4.

Figure 11-4. News banner notification

But this isn't linguistically correct (that is, 1 new item will display as "1 New Items"), nor will it localize properly. What's needed is a way of expressing this in a conditional way that accounts for language differences. `Mojo.Format.formatChoice()` can address this need:

```
var bannerParams = {
    messageText: Mojo.Format.formatChoice(
        this.list[this.feedIndex].newStoryCount,
        $L("0##{title} : No New Items|1##{title} : 1 New Item|1>##{title} :
          #{count} New Items"),
        {title: this.list[this.feedIndex].title, count:
          this.list[this.feedIndex].newStoryCount}
    )
};

appController.showBanner(bannerParams, {action: "notification",
index: this.feedIndex},
  this.list[this.feedIndex].url);
```

The call to `formatChoice()` passes an integer representing the quantity, as well as a set of choices. In this case, the choices are:

- For a quantity of 0 then a string of No New Items
- For a quantity of 1, then a string of 1 New Item
- For a quantity greater than 1, then a string of #{count} New Items

The final argument includes the object that supplies the values to be substituted for the templates `#{title}` and `#{count}`.

Summary

Palm webOS is a world-ready operating system designed to support localized and internationalized applications. It supports conventional locales and ships with character sets, fonts, and keyboards required to support Latin-1 languages and regions. The localization architecture supports dynamic string substitution and provides basic internationalization APIs for regional formatting.

As with the rest of the framework, building global-ready applications is easy with Mojo and Palm webOS. If you haven't attempted to take an application beyond your own region or locale, then this is a great opportunity to expand your potential user base.

APPENDIX A

Palm webOS Developer Program

Palm is working collaboratively with the worldwide developer community to construct a vibrant, vital ecosystem around the Palm webOS platform to enable next-generation mobile application development. Together, Palm and the development community have an opportunity to revolutionize the design, development, and distribution of mobile applications in ways that will redefine the mobile device industry.

Through its developer website, webOSdev, Palm offers a wide range of resources to assure that developers of all kinds receive the assistance they need to build and market webOS applications. Palm webOSdev is the online gathering point for the worldwide mobile OS development community, providing developers the ability to contribute to the platform itself. Both the resources and community can be found through webOSdev at *http://developer.palm.com/*.

Philosophy

Palm webOS is the latest in a long line of technologies that deliver the Palm experience—fast, easy, reliable mobile devices and applications that allow people to manage extraordinary lives on the go. Palm mobile computing products—together with your applications—enable anytime, anywhere computing for consumers, mobile professionals, and enterprise users.

That's just a starting point. Building on its heritage of working together with developers to build the Palm experience, Palm is committed to working even more closely with the developer community. Together, we can open new, unexplored markets and previously unimaginable technological and business opportunities.

To realize its vision of a thriving mobile device development ecosystem, Palm is taking the following steps:

- Attracting developers of all kinds—consumer, enterprise, professional, educational, even hobbyists from all over the world—to explore webOS technologies and deliver innovative applications.

- Connecting members of the community through both online and traditional means.
- Providing the vehicles for developers to effectively and profitably distribute their applications to an ever-widening mobile user market.

Palm webOS: Open Platform, Open Community

Palm webOS development uses technologies like HTML, CSS, and JavaScript. Palm believes in an open model for building the developer ecosystem as well—one where the development community helps drive the overall direction of the platform.

Key to building an open developer ecosystem is a meeting point for the worldwide developer community, which Palm is building through the webOSdev website. Through user forums, blogs, and other to-be-delivered vehicles for community contributions, you can provide your own ideas, development techniques, questions, and comments. Palm views you as an integral partner in building the next-generation mobile development platform, and we'll marry code, concepts, and content from the community with our own to evolve the platform.

Benefits to the Developer

By participating in the Palm developer community, you can become a part of the next generation of mobile computing, helping to revolutionize the way people communicate Because webOS relies on well-established, flexible, robust technologies, you have all the tools you need to implement your ideas, and you'll be able to do develop more quickly, cheaply, and profitably.

Working with Palm, you'll be able to do the following:

- Speed your time-to-success with the webOS platform.
- Develop applications more quickly and with more innovative user functionality than with any other mobile OS platform.
- Help advance the state of the art of mobile application development.
- Play a part in building the new mobile computing business model, and take advantage of it.
- Work with a strong partner—Palm—to reach new markets.

Resources and Community

Palm plans full lifecycle support for the developer: from the moment you first consider the platform, through the development and distribution of your first application, then to the maturation of your application and its evolution to the next big thing. This can only work through an open partnership with the entire community. Palm provides an

initial offering of resources, and you use, comment on, and add to that offering, making it a living, breathing entity that's cared for and fed by the entire ecosystem.

Initially, Palm makes the following technical resources available through the webOSdev developer website:

- Palm webOS Mojo Software Development Kit
- Additional Palm development tools
- Palm webOS developer guide
- Palm webOS API documentation
- Human interface guide
- Code samples
- Access to support engineers

To facilitate a dialogue with the community, Palm also supports:

- Community support forums monitored by internal Palm experts
- Palm developer blog with news, perspective, and commentary for developers

To help you distribute and sell your applications and build your business, Palm is investing in the infrastructure to deliver:

- An application submission and signing process to certify your application
- The Palm App Catalog, with administrative features to help you manage the business of selling your application

What You Should Do

Download the Palm webOS Mojo SDK and start developing. Let us know what you're thinking, what's working, how we can help, and, most of all, contribute back to the community in all the ways available. Remember, your idea will become part of the future of mobile device development.

APPENDIX B

Quick Reference—Developer Guide

Throughout the book, you've seen sample code and examples of widgets and services. In this section, those interfaces are compiled in reference form to help you as you are coding or if you just want to look something up.

This is not a comprehensive list of all Mojo APIs; for that you should refer to the SDK site. There you will find the APIs mentioned in this book, including:

- Widgets
- Dialogs
- Menus
- Storage
- Services
- Controller APIs

Widgets

This section includes all of the Mojo widgets. Each widget section includes a brief description repeating some of the information from Chapters 3, 4, and 5, followed by an enumeration of the widget's attribute and model properties, relevant events, and public methods.

Button

Buttons are the most basic UI element, bounding an action to a region. When a button is pushed, it can change state but gracefully returns to the previous state, like a doorbell.

Attribute properties	Type	Description
type	String	Choices: Mojo.Widget.defaultButton Mojo.Widget.activityButton
disabledProperty	Boolean	Name of model property for disabled state
label	String	Displayed label
labelProperty	String	Model property name for label
Model properties	**Type**	**Description**
buttonClass	String	Style options are primary, secondary, dismissal, affirmative, or negative
label	String	Displayed label
disabled	Boolean	Default property that when true, disables the widget
Events		
Mojo.Event.tap		
Methods		
activate()		For an activity button, start the spinner
deactivate()		For an activity button, stop the spinner

Check Box

A Check Box widget is used to control and indicate a binary state value in one element.

Attribute properties	Type	Description
modelProperty	String	Name of model property for widget state
disabledProperty	String	Name of model property for disabled state
trueValue	String	Value to set modelProperty when widget state is true
falseValue	String	Value to set modelProperty when widget state is false
inputName	String	Identifier for the value of the check box; used when the widget is used in HTML forms
Model properties	**Type**	**Description**
value	Boolean	Current value of widget
disabled	Boolean	Default property that when true, disables the widget
Events		
Mojo.Event.propertyChange		

Date Picker

The date picker allows selection of month, day, and year values.

Attribute properties	Type	Description
label	String	Label displayed with the widget controls
labelPlacement	String	Choices are: Mojo.Widget.labelPlacementLeft Mojo.Widget.labelPlacementRight
modelProperty	String	Name of model property for date object; defaults to 'date'
month	Boolean	Set to true to display the month field in the widget
day	Boolean	Set to true to display the day field in the widget
year	Boolean	Set to true to display the year field in the widget
maxYear	Integer	Specify maximum year in year capsule if enabled (default 2099)
minYear	Integer	Specify minimum year in year capsule if enabled (default 1900)
Model properties	**Type**	**Description**
date	Date	Date object set to the widget value
Events		
None		

Drawer

Drawers are container widgets that can be open, allowing child content to be displayed normally, or closed, keeping it out of view.

Attribute properties	Type	Description
unstyled	Boolean	When set to true, prevents styles from being added, allowing the Drawer to be used just for open/close function
modelProperty	String	Name of model property for date object
Model properties	**Type**	**Description**
open	Boolean	Current state of the widget; set to true when open
Events		
Mojo.Event.propertyChange		
Methods		
setOpenState(open)		Sets the open state to open or closed
getOpenState()		Returns current value of open state
toggleState()		Change the drawer's open state to the opposite of what it is now

File Picker

```
Mojo.FilePicker.pickFile (params, stageController)
```

The File Picker presents a file browser that lets the user navigate the directory structure and optionally select a file. The File Picker lets users view and select files from the media partition, and allows filtering by file type (e.g., file, image, audio, or video).

Arguments	Type	Description
`params`	Object	Object containing information about the file to select as described below:
`onSelect`	Function	Function to call after selection is made; it will return an object of the following format: `{ fullPath: /full/path/of/selected/file' }`
`onCancel`	Function	Function to call after selection is made; when picker is canceled.
`kinds`	Array	Array of strings to allow (`image`, `audio`, `video`, `file`); default is support for all kinds; if only one kind is needed, kind: `mytype` can be used and override kinds
`defaultKind`	String	String; the view to go to; one of `image`, `audio`, `video` or `file`
`actionType`	String	`attach` and `open` are the only supported options; `open` is the default
`actionName`	String	Overrides the default string defined by actionType
`extensions`	Array	File extensions to filter in the files view
`stageController`	Object	The calling application's stage assistant
Events		
None		

Filter Field

The Filter Field can be applied to any case where you want to process the field contents and update on-screen elements based on the entered string.

Attribute properties	Type	Description
`delay`	Integer	Delay between key strokes for a filter event, in milliseconds
`disabledProperty`	String	Name of model property for disabled state
Model properties	**Type**	**Description**
`disabled`	Boolean	Default property that, when true, disables the widget
Events		
`Mojo.Event.filter`		
Methods		
`open()`		Open the widget
`close()`		Close the widget
`setCount(integer)`		Set the number to be shown in the results bubble in the Filter Field

Filter List

The Filter List combines a Filter Field and a List. It is intended to display a variable length list of objects, built by a special callback function that filters the list based on the contents of the filter field.

Attribute properties	Type	Description
`delay`	Integer	Delay between key strokes for a filter event, in milliseconds
`filterFunction`	Function	Function called to load items into the list as it is scrolled or filter changes; function definition is: `filterFunction(filterString, listWidget, offset, count)` where: `filterString`: set to the input string `listWidget`: this Filter Listwidget object `offset`: index of first displayed entry in listfs `count`; number of entries required from the function
`listTemplate`	String	File path relative to app folder for container template
`itemTemplate`	String	File path relative to app folder for item template
`addItemLabel`	String	If defined, a special "add" item will be appended to the list and taps on this will generate a `Mojo.Event.listTap`
`formatters`	Function	Object functions to format list entries based on model properties
`itemsProperty`	String	Model property for items list
`swipeToDelete`	Boolean	If true, list entries can be deleted with a swipe
`autoconfirmDelete`	String	If false, delete swipes will post a delete/undo button pair, otherwise deletes will be made immediately after swiping
`uniquenessProperty`	String	Name of an item model property which can be used to uniquely identify items; if specified, List will maintain a hash of swiped items instead of setting a deleted property
`preventDeleteProperty`	String	If specified, the item models will be checked for this property, and `swipeToDelete` will be ignored on that item if the item model's property is true
`reorderable`	Boolean	If true, list entries can be reordered by drag and drop
`dividerFunction`	Function	Function to create divider elements
`dividerTemplate`	Function	Function to format divider
`fixedHeightItems`	Boolean	If false, list widget will not apply optimizations for fixed height lists
`initialAverageRowHeight`	Integer	Initial value used for average height estimation
`renderLimit`	Integer	Max number of items to render at once; increase this if the UI overruns the list boundaries
`lookahead`	Integer	Number of items to fetch ahead when loading new items

Attribute properties	Type	Description
dragDatatype	String	Used for drag-and-drop reordering; if specified, will enable dragging of items from one list to another of the same data type
deletedProperty	String	Name of the item object property in which to store the deleted status of an item
nullItemTemplate	String	File path relative to app folder for template for items that are rendered before loading
emptyTemplate	String	File path relative to application folder for template for empty list
onItemRendered	Function	Called each time an item is rendered into the DOM with these arguments
Model properties	**Type**	**Description**
disabled	Boolean	Default property that, when true, disables the widget
Events		
Mojo.Event.listChange Mojo.Event.listTap Mojo.Event.listAdd Mojo.Event.listDelete Mojo.Event.listReorder Mojo.Event.filter		
Methods		
getList()		Get the List widget associated with this filter list
open()		Open the filter field associated with this filter list
close()		Close the filter field associated with this filter list
setCount(integer)		Set the number to be shown in the results bubble in the filter field

Image View

You can use an `ImageView` for displaying single images, but it is intended as a scrolling viewer, flicking left and right through a series of images.

Attribute properties	Type	Description
highResolutionLoad	Integer	Time to wait before switching photo to high res
noExtractFS	Boolean	Flag to prevent looking up a high res version
fslimitZoom	Boolean	Flag to prevent or limit zooming
Model properties	**Type**	**Description**
onLeftFunction	Function	Called after a left scroll and transition
onRightFunction	Function	Called after a right scroll and transition
Events		
Mojo.Event.imageViewChanged		

Methods	
`getCurrentParams()`	Return the current zoom level and focus [0,1]
`manualSize(width, height)`	Manually size the widget
`leftUrlProvided(url)`	Set the image for the left scroll
`rightUrlProvided(url)`	Set the image for the right scroll
`centerUrlProvided(url)`	Set the image for the center

Integer Picker

The Integer Picker offers a selection between minimum and maximum integer values, both of which are specified as properties.

Attribute properties	Type	Description
`label`	String	Label displayed with the widget controls

`labelPlacement`	String	Choices are:
		`Mojo.Widget.labelPlacementLeft`
		`Mojo.Widget.labelPlacementRight`

`modelProperty`	String	Name of model property for integer value; defaults to `'value'`
`min`	Integer	Minimum value of the widget
`max`	Integer	Maximum value of the widget
`padNumbers`	Boolean	Add padding to single digit numbers or not
Model properties	**Type**	**Description**
`value`	Integer	Current value of the widget
Events		
`Mojo.Event.propertyChange`		

List

List is the most common and possibly the most powerful Mojo widget. Objects are rendered into list items using provided HTML templates, and may be variable height and/or include other widgets.

Attribute properties	Type	Description
`listTemplate`	String	File path relative to app folder for container template
`itemTemplate`	String	File path relative to app folder for item template
`addItemLabel`	String	If defined, a special "add" item will be appended to the list and taps on this will generate a `Mojo.Event.listTap`
`formatters`	Function	Object functions to format list entries based on model properties
`itemsProperty`	String	Model property for items list

Attribute properties	Type	Description
`itemsCallback`	Function	Items will be loaded as needed by calling this function
`swipeToDelete`	Boolean	If true, list entries can be deleted with a swipe
`autoconfirmDelete`	String	If false, delete swipes will post a delete/undo button pair, otherwise deletes will be made immediately after swiping
`uniquenessProperty`	String	Name of an item model property which can be used to uniquely identify items; if specified, List will maintain a hash of swiped items instead of setting a deleted property
`preventDeleteProperty`	String	If specified, the item models will be checked for this property, and `swipeToDelete` will be ignored on that item if the item model's property is true
`reorderable`	Boolean	If true, list entries can be reordered by drag and drop
`dividerFunction`	Function	Function to create divider elements
`dividerTemplate`	Function	Function to format divider
`fixedHeightItems`	Boolean	If false, list widget will not apply optimizations for fixed height lists
`initialAverageRowHeight`	Integer	Initial value used for average height estimation
`renderLimit`	Integer	Max number of items to render at once; increase this if the UI overruns the list boundaries
`lookahead`	Integer	Number of items to fetch ahead when loading new items
`dragDatatype`	String	Used for drag-and-drop reordering; if specified, will enable dragging of items from one list to another of the same data type
`deletedProperty`	String	Name of the item object property in which to store the deleted status of an item
`nullItemTemplate`	String	File path relative to app folder for template for items that are rendered before loading
`emptyTemplate`	String	File path relative to application folder for template for empty list
`onItemRendered`	Function	Called each time an item is rendered into the DOM with these arguments
Model properties	**Type**	**Description**
`items`	Array	An array of objects to display in the list; required unless `itemsCallback` property is set as an attributes property
Events		

`Mojo.Event.listChange`
`Mojo.Event.listTap`
`Mojo.Event.listAdd`
`Mojo.Event.listDelete`
`Mojo.Event.listReorder`		

Methods		
`focusItem(itemModel, focusSelector)`		Focus the item designated by the item model; optionally pass in the `focusSelector` to focus a specific element within the item
`showAddItem(enable)`		Show the "add item" in the list

Methods	
`noticeUpdatedItems(offset, items)`	Causes the given items to be replaced and rerendered; items provided past the current end of the list will cause the length to grow; must pass an array
`noticeAddedItems(offset, items)`	Inserts the given array of items into the list at the given offset; if list items are dynamically loaded, this may cause some to be kicked out of the cache; calling this API will not cause a property-change event to be fired
`noticeRemovedItems(offset, items)`	Removes items from the list beginning at the given offset, and continuing for limit items; if list items are dynamically loaded, this may cause new ones to be requested; calling this API will not cause a property-change event to be fired
`getNodeByIndex(index)`	Return top level node for the list item of the given index; returns undefined if the item does not exist or is not currently rendered
`invalidateItems(offset, limit)`	Causes the given items to be reloaded (if currently loaded); if limit is unspecified, causes all items after `offset` to be invalidated
`getLoadedItemRange()`	Returns a hash with offset and limit properties indicating the range of currently loaded item models (or items that have been requested); this is sometimes used on the service side to optimize subscription data
`getMaxLoadedItems()`	Returns the maximum number of loaded items the list will maintain in its local cache
`setInitialSize(length)`	Call to set the initial size of the list, or after resetting the list state by calling `modelChanged();` this function will set the limit on what range of items may be requested, but subsequent changes to the list size should be made through `noticeAddedItems()` and `fsnoticeRemovedItems()`; this function has no affect when the list size != 0
`setLength(length)`	Call to set the overall length of the list; this function will set the range of items that may be requested, but will generally not invalidate any existing items or request any new ones; it may request new items when the currently loaded items window is either not full, or the length change causes the items window to move (the latter case can occur if the length change causes the window to be "out of bounds", or if it would ideally be positioned past the end of the list)
`setLengthAndInvalidate(length)`	Behaves like `setLength()`, except that all currently loaded items are invalidated; for lazily loaded lists, this API will result in a request for a whole window of items
`getLength()`	Returns the current length of the list
`revealItem(index, animate)`	Attempts to scroll the scene to reveal the item with the given index; may behave poorly when working with variable height list items that are not currently loaded, since you can't accurately predict the height of the final rendered content
`getItemByNode(node)`	Returns the item model associated with the list item containing the given node, if any; otherwise, returns undefined

List Selector

The List Selector enables the selection of one of many options, presented in a pop-up list in which there is no practical limit to the number of options presented. This is similar to the Submenu widget behavior.

Attribute properties	Type	Description
`modelProperty`	String	Name of model property for widget state
`disabledProperty`	String	Name of model property for disabled state
`multi-line`	Boolean	If true, long labels will wrap to the next line instead of being truncated
`label`	String	Display label
`labelPlacement`	String	`Mojo.Widget.labelPlacementRight` places label on right, value on left (default); `Mojo.Widget.labelPlacementLeft` places label on left, value on right
`choices`	Array	Array of selector descriptions, which must be in either the attributes or model object; each entry is required to be: `{label: 'string', value: value}` There must be at least two entries, and the number of entries defines the number of options presented in the widget
Model properties	**Type**	**Description**
`value`	Boolean	Current value of widget
`disabled`	Boolean	Default property that, when true, disables the widget
`choices`	Array	Array of selector descriptions, which must be in either the attributes or model object; each entry is required to be: `{label: 'string', value: value}` There must be at least two entries, and the number of entries defines the number of options presented in the widget
Events		
`Mojo.Event.propertyChange`		

Password Field

A text field used for passwords or some other type of confidential information, the Password Field provides many of the Text Field features, but masks the display. Any entered text is displayed as a bullet or "•" character.

Attribute properties	Type	Description
`modelProperty`	Boolean	Name of model property for widget value
`hintText`	String	Initially displayed string; supplanted by model value if supplied

Attribute properties	Type	Description
`inputName`	String	If supplied, the text area will have this DOM name so that when it is serialized, the property can be easily pulled out
`charsAllow`	Function	Function must return `true` to allow input character, or `false` if not allowed
`autoFocus`	Boolean	If true, field has focus on scene push
`modifierState`	String	Initial state of modifier keys for this field; can be: `Mojo.Widget.numLock` or `Mojo.Widget.capsLock`
`growWidth`	Boolean	Automatically grow field horizontally
`autoResizeMax`	String	Maximum width of field
`enterSubmits`	Boolean	If set, the Enter key will submit rather than newline; must be used with multi-line
`limitResize`	Boolean	Limit height resize (scrolls text rather than grow field)
`preventResize`	Boolean	There will be no resizing in any dimension
`holdToEnable`	Boolean	If the text field is disabled, tapping and holding and releasing will enable it; if disabled is not set, this is ignored

`focusMode`	String	Replace or Insert Mode; choices are:
		`Mojo.Widget.focusSelectMode`
		`Mojo.Widget.focusInsertMode`
		`Mojo.Widget.focusAppendMode`

`changeOnKeyPress`	Boolean	If true, sends a property change event on every character change to a field; otherwise only when field loses focus
`maxLength`	Integer	Maximum character length of field; does not apply to multi-line fields, where it will be ignored
`requiresEnterKey`	Boolean	Required Enter key to submit; other navigation will not submit contents of field
`holdToEdit`	Boolean	Tap and hold to focus/edit; tap only will be ignored
Model properties	**Type**	**Description**
`value`	Boolean	Plain-text value of the widget
Events		
`Mojo.Event.propertyChange`		
Methods		
`focus()`		Put focus on the input field
`blur()`		Remove focus from the input field
`getValue()`		Get the plaintext value of the widget
`setValue()`		Set the plaintext value of the widget
`getCursorPosition()`		Returns an option with: `{selectionStart: int, selectionEnd: int}` that describe the position of the cursor; if start is not equal to end, there is text selected

Methods	
`setCursorPosition(start, end)`	Sets the cursor position in the input portion of the text field

Progress Bar

Progress Bar displays a narrow horizontal bar with an incremental internal bar to show progress. Use a Progress Bar or Pill to show download progress, when loading from a database, or anytime you initiate a long-running operation and have a sense of the duration.

Attribute properties	Type	Description
`modelProperty`	String	Name of model property for widget value
`icon`	String	CSS class for icon to display on the bar
`iconPath`	String	File path relative to application folder for icon
Model properties	**Type**	**Description**
`value`	Integer	Value of the widget
`title`	String	Dynamic title to show on bar
`image`	String	File path relative to application folder for dynamic image to show on bar
Events		
`Mojo.Event.progressComplete`		
Methods		
`reset()`		Reset progress to 0
`cancelProgress()`		Stop the progress and freeze bar display in current state

Progress Pill

Progress Pill displays a broad horizontal bar with an incremental pill to show progress. Use a Progress Bar or Pill to show download progress, when loading from a database, or anytime you initiate a long-running operation and have a sense of the duration.

Attribute properties	Type	Description
`modelProperty`	String	Name of model property for widget value
`title`	String	Title to show on bar
`image`	String	File path relative to application folder for image to show on bar
`icon`	String	CSS class for icon to display on the bar
`iconPath`	String	File path relative to application folder for icon
Model properties	**Type**	**Description**
`value`	Integer	Value of the widget

Model properties	Type	Description
`title`	String	Dynamic title to show on bar
`image`	String	File path relative to application folder for dynamic image to show on bar
Events		
`Mojo.Event.progressIconTap` `Mojo.Event.progressComplete`		
Methods		
`reset()`		Reset progress to 0
`cancelProgress()`		Stop the progress and freeze bar display in current state

Progress Slider

For media or other applications in which you want to show progress as part of a tracking slider, the Progress Slider is an ideal choice. Combining the Slider widget with the Progress Pill, the behavior is fully integrated, but not all of the configuration options are represented.

Attribute properties	Type	Description
`sliderProperty`	String	Name of model property for slider position value
`progressProperty`	String	Name of model property for progress position value
`progressStartProperty`	Integer	Starting position of progress bar
`minValue`	Integer	Starting value, or leftmost value on the slider
`maxValue`	Integer	Ending value, or rightmost value on the slider
`round`	Boolean	If true, will round the value to the nearest integer
`updateInterval`	Integer	If set >0, the widget will send events every `updateInterval` seconds
Model properties	**Type**	**Description**
`value`	Integer	Value of the widget
Events		
`Mojo.Event.propertyChange` `Mojo.Event.progressComplete`		
Methods		
`reset()`		Reset progress to 0

Radio Button

The Radio Button presents each button as a labeled selection option in a horizontal array, where only one option can be selected at a time.

Attribute properties	Type	Description
`modelProperty`	String	Name of model property for widget state
`disabledProperty`	String	Name of model property for disabled state
`choices`	Array	Array of button descriptions; each entry is required to be: `{label: 'string', value: value}` The number of entries defines the number of buttons presented in the widget
Model properties	**Type**	**Description**
`value`	Boolean	Current value of widget
`disabled`	Boolean	Default property that, when true, disables the widget
Events		
`Mojo.Event.propertyChange`		

Rich Text Edit

There is a simple Rich Text Edit widget that behaves similar to a multi-line text field, but in addition supports applying Bold, Italic, and Underline styles to arbitrary runs of text within the field. To enable the styling, set the Application menu's `RichTextEditItems` property to `true`.

Attribute properties	Type	Description
None		
Model properties	**Type**	**Description**
`value`	Boolean	Current value of widget
Events		
None		

Scroller

The Scroller widget provides the scrolling behavior in Mojo. Scrollers can be applied to any div content and set to one of six scrolling modes.

Attribute properties	Type	Description
`mode`	String	Scrolling mode; one of `free`, `vertical`, `horizontal`, `dominant`, `vertical-snap`, or `horizontal-snap`
Model properties	**Type**	**Description**
`snapElements`	Array	Array of DOM elements used as snap points for horizontal or vertical scrolling
Events		
`Mojo.Event.propertyChange`		

Methods	
`revealTop(newTop)`	Jumps the scroll to reveal the top of the specified object at the top of the scroll area
`revealBottom()`	Jumps the scroll to reveal the bottom of the content being scrolled
`revealElement(Element)`	Jumps the scroll to reveal a specific DOM element
`scrollTo(x-coord, y-coord, animated, suppressNotification)`	Jumps the scroll to the specified x- and y-coordinates; set animated to true to animate the scroll, or if animated is false, set `suppressNotification` to true to prevent notifications to event listeners
`getState()`	Returns the current scroll state for use in a future call to `setState()`
`setState(scrollState, animate)`	Jumps the scroll to the value specified in `scrollState`; pass true to animate the scroll
`adjustBy(deltaX, deltaY)`	Adjusts the current scroll position by the given amount; safe to call from scroll listeners while animating; does not cause listeners to be notified of any changes
`scrollerSize`	Returns the size of the scroller's view port in pixels: `{height:nnn, width:nnn}`
`setMode(newMode)`	Set the mode of the scroller, which controls which drag directions causes scrolling; choices are `free`, `dominant`, `horizontal`, `horizontal-snap`, `vertical`, and `vertical-snap`
`getScrollPosition()`	Get the current position of the scroller; returns: `{left: nnn px, top: nnn px}`
`setSnapIndex(snapIndex, animate)`	Sets the snap index for a snap scroller and scrolls to the new position; pass true to animate

Slider

The Slider presents a range of selection options in the form of a horizontal slider with a control knob that can be dragged to the desired location.

Attribute properties	Type	Description
`modelProperty`	String	Name of model property for widget value
`minValue`	Integer	Starting value, or leftmost value on the slider
`maxValue`	Integer	Ending value, or rightmost value on the slider
`round`	Boolean	If true, will round the value to the nearest integer
`updateInterval`	Integer	If set >0, the widget will send events every `updateInterval` seconds
Model properties	**Type**	**Description**
`value`	Integer	Value of the widget
Events		
`Mojo.Event.propertyChange`		

Spinner

Use a Spinner to show that an activity is taking place. The framework uses a Spinner as part of any activity button, and you'll see it used in the core applications. There are two sizes: the large Spinner is 128 × 128 pixels, and the small Spinner is 32 × 32.

Attribute properties	Type	Description
`modelProperty`	String	Name of model property for widget state

`spinnerSize`	String	Choices are:
		`Mojo.Widget.spinnerLarge`
		`Mojo.Widget.spinnerSmall`

`superClass`	String	Specifies the CSS class name for the background image with a custom spinner
`startFrameCount`	Integer	With a custom spinner, this is set to the number of frames for the preloop animation
`mainFrameCount`	Integer	With a custom spinner, this is set to the number of frames for the main loop animation
`finalFrameCount`	Integer	With a custom spinner, this is set to the number of frames for the post-loop animation
`frameHeight`	Integer	Explicitly sets the height of the animation
`fps`	Integer	Frames per second of the main loop animation
Model properties	**Type**	**Description**
`spinning`	Boolean	Spinner state; set to true if spinning
Events		
`Mojo.Event.propertyChange`		
Methods		
`start()`		Start the spinner
`stop()`		Stop the spinner
`toggle()`		Change the spinner from start to stop or from stop to start

Text Field

The basic text widget that supports all general text requirements: single or multi-line text entry, with common styles for labels, titles, headings, body text, line items, and item details. The editing tools include basic entry and deletion, symbol and alternate character sets, cursor movement, selection, cut/copy/paste, and auto text correction.

Attribute properties	Type	Description
`modelProperty`	Boolean	Name of model property for widget value
`disabledProperty`	String	Name of model property for disabled state
`hintText`	String	Initially displayed string; supplanted by model value if supplied

Attribute properties	Type	Description
`inputName`	String	If supplied, the text area will have this DOM name so that when it is serialized, the property can easily be pulled out
`charsAllow`	Function	Function must return `true` to allow input character, or `false` if not allowed
`autoFocus`	Boolean	If true, field has focus on scene push
`modifierState`	String	Initial state of modifier keys for this field; can be: `Mojo.Widget.numLock` or `Mojo.Widget.capsLock`
`growWidth`	Boolean	Automatically grow field horizontally
`autoResizeMax`	String	Maximum width of field
`enterSubmits`	Boolean	If set, the Enter key will submit rather than newline; must be used with multi-line
`limitResize`	Boolean	Limit height resize (scrolls text rather than grow field)
`preventResize`	Boolean	There will be no resizing in any dimension
`holdToEnable`	Boolean	if the text field is disabled, tapping and holding and releasing will enable it; if disabled is not set, this is ignored

`focusMode`	String	Replace or Insert Mode; choices are:
		`Mojo.Widget.focusSelectMode`
		`Mojo.Widget.focusInsertMode`
		`Mojo.Widget.focusAppendMode`

`changeOnKeyPress`	Boolean	If true, sends a property change event on every character change to a field; otherwise only when field loses focus
`maxLength`	Integer	Maximum character length of field; does not apply to multi-line fields, where it will be ignored
`requiresEnterKey`	Boolean	Requires Enter key to submit; other navigation will not submit contents of field
`holdToEdit`	Boolean	Tap and hold to focus/edit; tap only will be ignored
`emoticons`	Boolean	Enable emoticons on this field
`autoReplace`	Boolean	Whether to enable the Smart Text Engine services

`textCase`	String	Options are:
		`Mojo.Widget.steModeSentenceCase`
		`Mojo.Widget.steModeTitleCase`
		`Mojo.Widget.steModeLowerCase`

Model properties	**Type**	**Description**
`value`	Boolean	Value of the widget
`disabled`	Boolean	Default property that when true, disables the widget
Events		
`Mojo.Event.propertyChange`		
Methods		
`focus()`		Put focus on the input field

Methods	
`blur()`	Remove focus from the input field
`getValue()`	Get the plain-text value of the widget
`setValue(String)`	Set the plain-text value of the widget
`getCursorPosition()`	Returns an option with: `{selectionStart: int, selectionEnd: int}` that describe the position of the cursor; if start is not equal to end, there is text selected
`setCursorPosition(start, end)`	Sets the cursor position in the input portion of the text field

Time Picker

The Time Picker enables selection of hours, minutes, and either A.M. or P.M. for time selection. The picker will suppress the A.M./P.M. capsule if the 24-hour time format is selected in the user preferences or by the locale.

Attribute properties	Type	Description
`label`	String	Label displayed with the widget controls

`labelPlacement`	String	Choices are:
		`Mojo.Widget.labelPlacementLeft`
		`Mojo.Widget.labelPlacementRight`

`modelProperty`	String	Name of model property for date object; defaults to `'time'`
`minuteInterval`	Integer	Interval between minute selections
Model properties	**Type**	**Description**
`time`	Date	Date object set to the widget value
Events		
`Mojo.Event.propertyChange`		

Toggle Button

The Toggle Button is another widget for displaying and controlling a binary state value. As with the Check Box, the Toggle Button will switch between two states each time it is tapped.

Attribute properties	Type	Description
`modelProperty`	String	Name of model property for widget state
`disabledProperty`	String	Name of model property for disabled state
`trueValue`	String	Value to set `modelProperty` when widget state is true
`trueLabel`	String	Label when widget state is true
`falseValue`	String	Value to set `modelProperty` when widget state is false

Attribute properties	Type	Description
falseLabel	String	Label when widget state is false
inputName	String	Identifier for the value of the check box; used when the widget is used in HTML forms
Model properties	**Type**	**Description**
value	Boolean	Current value of widget
disabled	Boolean	Default property that, when true, disables the widget
Events		
Mojo.Event.propertyChange		

Web View

To embed a contained web object, declare and instantiate a `WebView` widget. You can use it render local markup or to load an external URL; as long as you can define the source as a reachable URL, you can use a `WebView` to render that resource.

Attribute properties	Type	Description
virtualpageheight	Integer	The browser's virtual page height
virtualpagewidth	Integer	The browser's virtual page width
url	String	The initial URL to display
pageIdentifier	Function	The BrowserServer page identifier; this is used when the Browser Server instructs an application to open a new URL
minFontSize	Integer	The minimum font size that the browser will display
topMargin	Integer	The margin above the web view that is scrolled off the screen when a new page is loaded
cacheAdapter	Boolean	If true, cache this adapter, false if not, or undefined to not specify and use the browser-adapter default; default is undefined
interrogateClicks	Boolean	Use to call the host application for every hyperlink click via `Mojo.Event.webViewLinkClicked`
showClickedLink	Boolean	Styles clicked links with grey background and border
Model properties	**Type**	**Description**
None		
Events		
Mojo.Event.webViewLoadStarted Mojo.Event.webViewLoadProgress Mojo.Event.webViewLoadStopped Mojo.Event.webViewDownloadFinished Mojo.Event.webViewLinkClicked Mojo.Event.webViewTitleUrlChanged Mojo.Event.webViewTitleChanged Mojo.Event.webViewUrlChanged Mojo.Event.webViewCreatePage Mojo.Event.webViewTapRejected		

Events

```
Mojo.Event.webViewScrollAndScaleChanged
Mojo.Event.webViewEditorFocused
Mojo.Event.webViewUpdateHistory
Mojo.Event.webViewSetMainDocumentError
Mojo.Event.webViewServerConnect
Mojo.Event.webViewServerDisconnect
Mojo.Event.webViewResourceHandoff
Mojo.Event.webViewFirstPaintComplete
Mojo.Event.webViewUrlRedirect
Mojo.Event.webViewModifierTap
Mojo.Event.webViewMimeNotSupported
Mojo.Event.webViewMimeHandoff
```

Methods

`setTopMargin(margin)`	Set the top margin (in pixels)
`clearCache()`	Clear browser cache
`clearCookies()`	Clear browser cookies
`deleteImage(image)`	Delete the image file specified by the argument
`generateIconFromFile(src, dst, left, top, right, bottom)`	Generate a 64 × 64 pixel icon from a portion of a source file; the output icon will be given a drop shadow and sheen consistent with other launcher icons
`goBack()`	Go to the previous page in the user's browsing history
`goForward()`	Go to the next page in the user's browsing history
`openURL(url)`	Open the specified URL in the WebView
`reloadPage()`	Reload the currently loaded page
`resizeImage(src, dst, width, height)`	Resize the input file to the specified width/height and write the new image to the specified output file
`getHistoryState(onSuccess)`	Asynchronous; retrieves the current history state from the Browser server; will call `onSuccess` with results
`setBlockPopups(enable)`	Set to true to block pop-ups
`setAcceptCookies(enable)`	Set to false to disable cookies
`addUrlRedirect(urlRe, redirect, userData, type)`	Add a URL redirect; when the browser server navigates to a URL matching `urlRe` and `redirect` is true, it will not navigate to that URL, and will instead send a `Mojo.Event.webViewUrlRedirect` event
`addSystemRedirects(skippAppId)`	Read the command resource handler table and send down redirect handler commands to the browser server
`saveViewToFile(fname, left, top, width, height)`	Save the specified view frame (in pixels) to the specified file
`setEnableJavaScript(enable)`	Set to false to disable JavaScript
`stopLoad()`	Stop loading the current page
`clearHistory()`	Clear browser history
`setShowClickedLink(enable)`	Set to true to enable launch of clicked links

Dialogs

This section includes the three Mojo dialog APIs:

- Error dialog
- Alert dialog
- Custom dialog

Each entry includes a brief description, repeating some of the information from Chapter 4, followed by an enumeration of the arguments. Dialogs are accessed through a direct API so the format for this entry will be structured as an API entry.

Mojo.Controller.errorDialog()

This API is used to post error messages in a modal dialog box with a fixed title of "Error," a customizable message, and a confirmation button. The Error dialog must be used only with errors, since you can't change the title.

Arguments	Type	Description
`message`	String	Displayed message in a modal dialog
`window`	Element	Optional argument to specify the window to post the alert within; required in multistage applications

Mojo.Controller.SceneController.showAlertDialog()

You can display a short message using an Alert dialog, with one or more HTML buttons presenting the selection options. This is the best option if you have either a nonerror message for the user or want to present options in the form of button selections.

Arguments	Type	Description
`onChoose`	Function	Handler called when user makes a choice and the dialog is dismissed
`message`	String	Displayed message in a modal dialog
`title`	String	Title of the dialog box
`preventCancel`	Boolean	If true, back gesture or other alerts will not cancel the dialog box
`choices`	Array	Array of button descriptions; each entry is required to be: `{label: 'string', value: value, type: 'string', allowHTMLMessage: boolean}` The number of entries defines the number of buttons presented in the dialog box; the type property is set to one of the button classes (e.g., primary, secondary, affirmative, negative); allowHTMLMessage set to allow insertion of HTML if safe
`allHTMLMessage`	Boolean	If true, the message string will not have HTML escaped

Mojo.Controller.SceneController.showDialog()

The `showDialog` function can display any type of content to the user in the form of a modal dialog box. You can put anything into a dialog box that you'd put into a scene, meaning almost any web content or Mojo UI content.

Arguments	Type	Description
`template`	String	File path to HTML template containing content for the dialog box; rendered with properties from this model object
`assistant`	Object	The dialog assistant responsible for running the dialog box, which must implement methods (setup, activate, deactivate, cleanup, and `handleCommand`)
`preventCancel`	Boolean	If true, back gesture or other alerts will not cancel the dialog box

Menus

This section summarizes the four menu types:

- Application menu
- Command menu
- View menu
- Submenu

The three menu types are structured like the widgets in the previous section. Each entry includes a brief description, repeating some of the information from Chapter 4, followed by an enumeration of the widget's attribute and model properties, as well as relevant events. Unlike the other menu types, Submenu is accessed through a direct API, so the format for this entry will be structured as an API entry.

AppMenu

The Application menu appears in the upper-left corner of the screen when the user taps the left side of the status bar. It includes some system-defined and some application-defined actions, and is intended to have an application-wide scope for the most part.

Attribute properties	Type	Description
`omitDefaultItems`	Boolean	If true, default menu items will not be added to this menu
`richTextEditItems`	Boolean	If true, the Edit menu will also include Bold/Italics/Underline
Model properties	**Type**	**Description**
`label`	String	Currently not supported
`visible`	Boolean	Current visibility of this menu

Model properties	Type	Description
`Items`	Object	Object containing items for this menu, structured as:
`label`	String	User-visible label for this item, not rendered for groups
`icon`	String	CSS class for icon to display in this item
`iconPath`	String	Path to image to display in menu item, relative to application's directory
`width`	Integer	Calculated based on item's width; specifies the width in pixels of this menu item; overrides default calculations; ignored for groups
`items`	Boolean	If this is specified, this item is a group that visually ties the child items together
`toggleCmd`	Boolean	Only used when items is specified; specify this property to make this group a *toggle group*; this string is the command of currently selected choice item in this group, and this property is modified by the widget when a different choice is made
`command`	Boolean	Specify to make this item a choice; it will then respond to a user tap by sending a `Mojo.Event.command` through the commander chain with this string as the `command` property
`disabled`	Boolean	Menu choice is disabled when this is true; only used for items that also specify `command`
`submenu`	Boolean	Specify to make this item a submenu item; it will then respond to a user tap by displaying the named menu as a pop-up submenu
`template`	String	Path to HTML template for custom content to be inserted instead of a standard menu item; the template is rendered using this item model object for property substitution
`checkEnabled`	Boolean	If set to true, a `Mojo.Event.commandEnable` event will be sent through the commander chain each time this menu item is displayed or invoked via keyboard shortcut
Events		
`Mojo.Event.command` `Mojo.Event.commandEnable`		

Command Menu

The Command menu items are presented at the bottom of the screen. Items include variable-sized buttons that can be combined into groups, and in a horizontal layout from left-to-right.

Attribute properties	Type	Description
`spacerHeight`	Boolean	If specified, the spacer DIV associated with this menu will be the given height in pixels
`menuClass`	Boolean	Alternate CSS style; default is palm-default

Model properties	Type	Description
`label`	String	Currently not supported
`visible`	Boolean	Current visibility of this menu
`items`	Object	Object containing items for this menu, structured as:
`label`	String	User-visible label for this item, not rendered for groups
`icon`	String	CSS class for icon to display in this item
`iconPath`	String	Path to image to display in menu item, relative to application's directory
`width`	Integer	Calculated based on item's width; specifies the width in pixels of this menu item; overrides default calculations; ignored for groups
`items`	Boolean	If this is specified, this item is a group that visually ties the child items together
`toggleCmd`	Boolean	Only used when items is specified; specify this property to make this group a *toggle group*; this string is the command of currently selected choice item in this group, and this property is modified by the widget when a different choice is made
`command`	Boolean	Specify to make this item a choice; it will then respond to a user tap by sending a `Mojo.Event.command` through the commander chain with this string as the `command` property
`disabled`	Boolean	Menu choice is disabled when this is true; only used for items which also specify `command`
`submenu`	Boolean	Specify to make this item a submenu item; it will then respond to a user tap by displaying the named menu as a pop-up submenu
`template`	String	Path to HTML template for custom content to be inserted instead of a standard menu item; the template is rendered using this item model object for property substitution
`checkEnabled`	Boolean	If set to true, a `Mojo.Event.commandEnable` event will be sent through the commander chain each time this menu item is displayed or invoked via keyboard shortcut

Events
`Mojo.Event.command` `Mojo.Event.commandEnable`

View Menu

The View menu presents items as variable-sized buttons, either singly or in groups across the top of the scene. The items are rendered in a horizontal sequence starting from the left of the screen to the right.

Attribute properties	Type	Description
spacerHeight	Boolean	If specified, the spacer DIV associated with this menu will be the given height in pixels
menuClass	Boolean	Alternate CSS style; default is `palm-default`
Model properties	**Type**	**Description**
label	String	Currently not supported
visible	Boolean	Current visibility of this menu
items	Object	Object containing items for this menu, structured as:
label	String	User visible label for this item, not rendered for groups
icon	String	CSS class for icon to display in this item
iconPath	String	Path to image to display in menu item, relative to application's directory
width	Integer	Calculated based on item's width; specifies the width in pixels of this menu item; overrides default calculations; ignored for groups
items	Boolean	If this is specified, this item is a group that visually ties the child items together
toggleCmd	Boolean	Only used when items is specified; specify this property to make this group a *toggle group*; this string is the command of currently selected choice item in this group, and this property is modified by the widget when a different choice is made
command	Boolean	Specify to make this item a choice; it will then respond to a user tap by sending a `Mojo.Event.command` through the commander chain with this string as the `command` property
disabled	Boolean	Menu choice is disabled when this is true; only used for items which also specify `command`
submenu	Boolean	Specify to make this item a submenu item; it will then respond to a user tap by displaying the named menu as a pop-up submenu
template	String	Path to HTML template for custom content to be inserted instead of a standard menu item; the template is rendered using this item model object for property substitution
checkEnabled	Boolean	If set to true, a `Mojo.Event.commandEnable` event will be sent through the commander chain each time this menu item is displayed or invoked via keyboard shortcut
Events		
`Mojo.Event.command` `Mojo.Event.commandEnable`		

Submenu

```
Mojo.Controller.SceneController.popupSubmenu()
```

Pop-up submenus can offer a transient list of choices to the user, typically off of another menu entry or from a DOM element in the scene.

Arguments	Type	Description
`onChoose`	Function	Called when user makes a choice and the pop-up is dismissed
`placeNear`	Element	Used to position the pop-up menu near the triggering element
`toggleCmd`	Boolean	Causes the appropriate item to appear with a checkmark; supported in top-level model for pop-ups
`popupClass`	String	CSS class for the pop-up menu, referenced from the HTML templates
`scrimClass`	String	CSS class for the pop-up scrim; defaults to `submenu-popup`
`manualPlacement`	Boolean	If true, pop-up menu will not be placed automatically (centered, or near `placeNear` element)
`items`	Array	Array of choices; each choice is an object with the following properties:
`label`	String	Display name of choice
`command`	String	Command string passed to `onChoose` handler when selected
`secondaryIcon`	String	CSS class for a secondary icon to display, generally used for some kind of status, and appearing to the left of the menu item
`secondaryIconPath`	String	Just like `iconPath`, but for `secondaryIcon`
`chosen`	Boolean	Styles the item as the selected item in the group

Storage

This section describes the Cookie and Depot objects and methods. Each section includes a brief description, repeating some of the information from Chapter 6, followed by an enumeration of each object's methods with their arguments and return values. Both storage objects are accessed through a direct API, so the format for this entry will be structured as an API entry.

If you'd like more detailed information on the HTML 5 Database object, refer to the specification at *http://dev.w3.org/html5/webstorage/#databases*.

Mojo.Model.Cookie()

Mojo cookies are technically related to browser cookies, but with an object interface to simplify use by webOS applications. Mojo cookies typically store small amounts of data that will be used to preserve application state and related information, such as preference settings.

Calling the constructor will open the named cookie if it already exists, or if it doesn't exist, will create it.

Constructor		
`new Mojo.Model.Cookie(cookieName, optionalDocument)`		
Arguments	**Type**	**Description**
`cookieName`	String	Name for the cookie; has an application scope so uniqueness across applications is not a requirement
`optionalDocument`	Object	Document element to store cookie; defaults to current document
Methods		
`get()`		Returns the object stored in this cookie, or undefined if the cookie doesn't exist
`put(objectToStore, expirationDate)`		Updates the value of this cookie with the passed object with an optional date object to set an expiration date; if no expiration date is set, the cookie will not expire
`remove()`		Deletes the cookie

Mojo.Depot()

The Depot object is a wrapper on the HTML 5 APIs for simple object storage and retrieval. You can store up to 1 MB of data in a depot by default. Mojo provides by a few simple functions that wrap the HTML 5 APIs to create, read, update, or delete a database.

Calling the constructor will open the named depot if it already exists, or if it doesn't exist, will create it.

Constructor		
`new Mojo.Depot(options, onSuccess, onFailure)`		
Arguments	**Type**	**Description**
`options`	Object	Options for opening or creating a depot, structured as:
`name`	String	Name used to identify the underlying database; has an application scope; use `'ext:'` prefix to create Depot in the */media* partition
`version`	Integer	Version number used for the underlying database
`displayName`	String	Not currently supported; for future use
`estimatedSize`	Integer	Estimated size in bytes; used to assist framework in managing databases
`replace`	Boolean	If true, will replace the existing database if it exists; defaults to false
`onSuccess`	Function	Callback function that is called if the depot is successfully opened or created
`onFailure`	Function	Callback function that is called with an error string if an error occurs
Methods		
`add(key, objectToStore, onSuccess, onFailure)`		Function to add an object, `objectToStore`, identified by a `key`

Methods	
`get(key, onSuccess, onFailure)`	Gets the object identified by the `key` and returned as the single argument to the `onSuccess` function
`discard(key, onSuccess, onFailure)`	Removes the depot object associated with the key
`removeAll(onSuccess, onFailure)`	Removes everything in the depot

Services

This section describes the available application, device, and cloud service methods. Each service is briefly described, repeating some of the information from Chapters 8 and 9, with a listing of the service's available methods, arguments, and responses.

All services are accessed through:

```
Mojo.Service.Request(serviceName, {method:methodName,
parameters:{}, onSuccess:{}, onFailure:{}})
```

Each service entry includes:

- The `serviceName` in the form of a string such as `'palm://com.palm.name'`
- A description of each method, with the `methodName` and `parameters` properties
- The contents of the `response` object, which is provided as an argument in the callbacks to either the `onSuccess` or `onFailure` functions

Accounts

```
palm://com.palm.accounts/crud
```

The Accounts service provides an interface for interacting with the accounts system. To use the Synergy applications, you must provide an account ID as a parameter; this service provides access to those IDs.

Many of the methods will use some common objects.

Account		
Properties	**Type**	**Description**
`username`	String	Login credentials
`domain`	String	The account source
`accountId`	String	The account reference for use in the Synergy applications
`icons`	Object	Includes `largeIcon` and `smallIcon` properties whose values are file paths to the appropriate account icons
`dataTypes`	Object	A hash of strings indicating which applications apply to this account, either `"CONTACTS"` or `"CALENDAR"`

Account		
Properties	**Type**	**Description**
`isDataReadOnly`	Boolean	If true, data with this account is read-only

Method: listAccounts

Lists accounts created by this application.

Parameters	Type	Description
None		
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`list`	Array	List of `account` objects
`errorCode`	Integer	If `returnValue` is false, the `errorCode` provides the error number
`errorText`	String	An error message; only provided with `returnValue` set to false

Method: createAccount

Creates an account.

Parameters	Type	Description
`account`	Object	Account object specifying account data
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`accountId`	String	The account reference for use accessing the account or for services that are based on the account
`errorCode`	Integer	If `returnValue` is false, the `errorCode` provides the error number
`errorText`	String	An error message; only provided with `returnValue` set to false

Method: getAccount

Gets the details of an account.

Parameters	Type	Description
`accountId`	String	The account reference for use accessing the account or for services that are based on the account
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`account`	Object	Account object specifying account data

Response	Type	Description
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: updateAccount

Updates an account with revised data type, icons, or a change to read/write permissions.

Parameters	Type	Description
accountId	String	The account reference for use accessing the account or for services that are based on the account
displayName	String	The displayable account name
icons	Object	Includes largeIcon and smallIcon properties whose values are file paths to the appropriate account icons
isDataReadOnly	Boolean	If true, data with this account is read-only
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
accountId	String	The account reference for use accessing the account or for services that are based on the account
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: deleteAccount

Deletes account data for one or more data types.

Parameters	Type	Description
accountId	String	The account reference for use accessing the account or for services that are based on the account
dataTypes	Object	A hash of strings indicating which applications apply to this account, either "CONTACTS" or "CALENDAR"
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise there is an error and the account was not created
dataTypes	Object	A hash of strings indicating which data types were deleted
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Alarms

```
palm://com.palm.power/timeout
```

The Alarm service is based on the device's real-time clock (RTC). Alarms are intended to wake applications while minimized, maximized, or to drive polling for Dashboard applications.

Method: set

Sets an alarm to wake up the application.

Parameters	Type	Description
`at`	String	Create a calendar-based alarm in GMT; of the form: `'mm/dd/yyyy hh:mm:ss'`; either `at` or `in` is required
`in`	String	Create a relative alarm; of the form: `'hh:mm:ss'`, with a minimum alarm of five minutes
`uri`	String	URI of serviceName/methodName to be called when alarm fires
`wakeup`	Boolean	Wake up the device from sleep when set to true; set to false by default
`params`	Object	Parameter object to be sent along with service call; the contents are dependent on the service method defined by the `uri` parameter
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the alarm was successfully set; otherwise, there is an error and the alarm was not set
`key`	String	The key provided in the method call
`errorText`	String	An error message; only provided with `returnValue` set to false

Method: clear

Clears a previously set alarm.

Parameters	Type	Description
`key`	String	The key provided in the method call
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the alarm was successfully set; otherwise, there is an error and the alarm was not set
`key`	String	The key provided in the method call
`errorText`	String	An error message; only provided with `returnValue` set to false

Application Manager

```
palm://com.palm.applicationManager
```

The Application Manager service provides functions related to finding and launching applications. Applications launched through the Application Manager will open and maximize a new window for the targeted application while minimizing the current application window. This is the general case of specific application services. Currently supported are:

- Audio
- Browser
- Email
- Maps
- Messaging
- Phone
- Photos
- Video

For details on calling individual services, refer to the specific service in this section.

Method: open

The Application Manager will use the content type to find the appropriate application to use for that content. Refer to Tables B-1, B-2, and B-3 for the supported content types.

Parameters	Type	Description
`target`	String	A standard URI format of the form: `command://url`, where command is one of the supported commands, and the `url` is an argument string in conventional URL notation; if the command is a webOS application, the url argument string will be specified by that application
Response	**Type**	**Description**
None		

Table B-1. Supported file types

Extension	Mime type
htm	text/html
html	text/html
pdf	application/pdf
txt	application/txt
doc	application/doc
doc	application/msword
xls	appId":"com.palm.app.docviewer
xls	appId":"com.palm.app.docviewer
xls	application/vnd.ms-excel

Extension	Mime type
xls	application/x-excel
xls	application/x-msexcel
ppt	application/ppt
ppt	application/mspowerpoint
ppt	application/powerpoint
ppt	application/vnd.ms-powerpoint
ppt	application/x-mspowerpoint

Table B-2. Supported video formats

Extension	Mime type
mp4	video/mp4-generic
mp4	video/quicktime
mp4	video/mp4
mp4	video/mpeg4
m4v	video/mp4-generic
m4v	video/quicktime
m4v	video/mp4
m4v	video/mpeg4
3gp	video/3gp
3gpp	video/3gp
3g2	video/3gpp
3gpp2	video/3gpp
sdp	application/sdp

Table B-3. Supported audio formats

Extension	Mime type
3gp	audio/3gpp
3gpp	audio/3gpp
3ga	audio/3gpp
3gp	audio/3ga
3gpp	audio/3ga
3ga	audio/3ga
3g2	audio/3gpp2
3gp2	audio/3gpp2
sdp	audio/amr

Extension	Mime type
amr	audio/x-amr
mp3	audio/mpa
mp3	audio/mp3
mp3	audio/x-mp3
mp3	audio/x-mpg
mp3	audio/mpeg
mp3	audio/mpeg3
mp3	audio/mpg3
mp3	audio/mpg
mp4	audio/mp4
m4a	audio/mp4
m4a	audio/m4a
aac	audio/aac
aac	audio/x-aac
aac	audio/mpeg
aac	audio/mp4a-latm
wav	audio/wav
pls	audio/x-scpls
m3u	audio/mpegurl
m3u	audio/x-mpegurl

Method: launch

This method will launch a specific application and pass in parameters in the form that the application has specified.

Parameters	Type	Description
`id`	String	The application ID, defining the intended application
`params`	Object	JSON object containing the parameters for the target application and specified by the target application
Response		
None		

Audio

```
palm://com.palm.applicationManager
```

The Audio application can be launched through the Application Manager service.

Method: launch

Launches the Audio application to play or stream the file located at the target URI, downloading it first if not already on the device. If the URI is not specified, it will just launch the audio player to its normal starting scene.

Parameters	Type	Description
`id`	String	Set to `'com.palm.app.streamingmusicplayer'`
`params`	Object	Includes a single property:
`target`	String	URL of the form *rtsp://audio-file*, where audio-file is a well-formed URI targeting an audio file encoded in a supported video format
Response	**Type**	**Description**
None		

Calendar

```
palm://com.palm.calendar/crud
```

The Calendar API provides programmatic access to the Calendar application. It allows you to create, read, update, delete, and list calendars, events, and attendees. To use this API you must have an account created via the Accounts API. An account has many calendars, a calendar has many events, and an event has many attendees.

Many of the APIs will use the `calendar` or `event` objects as input or output argument.

Calendar		
Properties	**Type**	**Description**
`calendarId`	String	Immutable unique identifier for this calendar
`name`	String	Display name for this calendar
`externalId`	String	External ID for this calendar

Event		
Properties	**Type**	**Description**
`subject`	String	Title of the event
`startTimestamp`	String	Event start time (in milliseconds UTC)
`endTimestamp`	String	Event end time (in milliseconds UTC)
`allDay`	Boolean	True if all-day event
`note`	String	Optional note text
`location`	String	Location of the event
`alarm`	String	ISO 8601 duration format or none, all lowercase

Event		
Properties	**Type**	**Description**
`rrule`	String	RFC 2445 recurrence string, may only include RRULE, EXRULE, RDATE, or EXDATE
`rruleTZ`	String	`rrule` timezone object; this object is mandatory if an `rrule` is specified
`endValidity`	String	The end timestamp for nonrecurring events; for recurring events, the end timestamp of last occurrence or 0 if it repeats forever
`attendees`	Object	Event start time (in milliseconds UTC)
`externalId`	String	An `externalId` reference to this event and `parented`
`parentId`	String	This is an exception that needs to be linked to a parent recurring series
`originalStartTimestamp`	Integer	The original start timestamp of this event in the parent series

On methods that accept a `calendarId` as an input argument (`getCalendar`, `updateCalendar`, and `deleteCalendar`), you can alternately provide an `externalId` and an `accountId`.

Method: createCalendar

Creates a new calendar.

Parameters	Type	Description
`accountId`	String	The account reference for use in the Synergy applications
`calendar`	Object	Specify the `name` and optionally the `externalId`
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`errorCode`	Integer	If `returnValue` is false, the `errorCode` provides the error number
`errorText`	String	An error message; only provided with `returnValue` set to false
`calendarId`	String	Immutable unique identifier for this calendar

Method: getCalendar

Retrieves the named calendar.

Parameters	Type	Description
`calendarId`	String	Immutable unique identifier for this calendar
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created

Response	Type	Description
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
calendar	Object	Specify the name and optionally the externalId

Method: updateCalendar

Updates the named calendar.

Parameters	Type	Description
calendar	Object	Specify the name and optionally the externalId
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: deleteCalendar

Deletes the named calendar.

Parameters	Type	Description
calendarId	String	Immutable unique identifier for this calendar
Response	**Type**	**Description**
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: listCalendars

Lists all calendars created by the calling application; you will see only the calendars that you have created.

Parameters	Type	Description
accountId	String	The account reference for use in the Synergy applications
Response	**Type**	**Description**
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
calendars	Array	List of calendar objects associated with this account

Method: createEvent

Adds the provided event to the named calendar.

Parameters	Type	Description
calendarId	String	Immutable unique identifier for this calendar
event	Object	Details of the calendar event
trackChanges	Boolean	Marks this change so that it's tracked, and getChanges will include this record ID in its response; defaults to false
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
eventId	String	Immutable unique identifier for this event

Method: getEvent

Retrieves the details of the named event.

Parameters	Type	Description
eventId	String	Immutable unique identifier for this event
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
event	Object	Details of the calendar event

Method: updateEvent

Updates the event details of the named event.

Parameters	Type	Description
event	Object	Details of the calendar event
trackChanges	Boolean	Marks this change so that it's tracked, and getChanges will include this record ID in its response; defaults to false
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: deleteEvent

Deletes the named event.

Parameters	Type	Description
`eventId`	String	Immutable unique identifier for this event
`trackChanges`	Boolean	Marks this change so that it's tracked, and `getChanges` will include this record ID in its response; defaults to false
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`errorCode`	Integer	If `returnValue` is false, the `errorCode` provides the error number
`errorText`	String	An error message; only provided with `returnValue` set to false

Method: listEvents

Lists all events created by the calling application; you will only see events in calendars that you have created.

Parameters	Type	Description
`calendarId`	String	Immutable unique identifier for this calendar
`startTimestamp`	String	Start of search range (in milliseconds UTC)
`endTimestamp`	String	End of search range (in milliseconds UTC)
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`errorCode`	Integer	If `returnValue` is false, the `errorCode` provides the error number
`errorText`	String	An error message; only provided with `returnValue` set to false
`events`	Array	Array of `event` objects that fit the search criteria

Method: startTracking

Enables change tracking for an account. After this is called, all user-initiated changes to records belonging to this account will be returned by `getChanges`. Change tracking is incremental, and changes are forgotten each time `doneWithChanges` is called.

Parameters	Type	Description
`accountId`	String	The account reference for use in the Synergy applications
`trackerId`	String	A client-chosen handle if multiple independent change trackers are needed per account

Response	Type	Description
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: getChanges

Gets a list of calendar and event IDs for records in an account that have been changed or deleted by the user since the last time doneWithChanges was called.

Parameters	Type	Description
accountId	String	The account reference for use in the Synergy applications
trackerId	String	A client-chosen handle if multiple independent change trackers are needed per account

Response	Type	Description
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
token	String	A string to be passed to doneWithChanges after processing of this change set is complete
calendar	Object	Includes changed, an array of changed calendar objects (calendarId only), and deleted, an array of deleted calendar objects (calendarId and externalId)
events	Object	Includes changed, an array of changed event objects (eventId only), and deleted, an array of deleted event objects (eventId and externalId)

Method: doneWithChanges

Forgets all changes for an account. Future calls to getChanges will return only changes from this point forward.

Parameters	Type	Description
accountId	String	The account reference for use in the Synergy applications
trackerId	String	A client-chosen handle if multiple independent change trackers are needed per account
token	String	A string to be passed to doneWithChanges after processing of this change set is complete

Response	Type	Description
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Connection Manager

```
palm://com.palm.connectionmanager
```

Use the Connection Manager's getStatus method to get updates on the device connection status.

Method: getStatus

Gets connections status and optionally subscribes to connection notifications.

Parameters	Type	Description
subscribe	Boolean	Set to true for subscriptions, default is false
Response	**Type**	**Description**
returnValue	Boolean	If true, the status data was returned and the subscription was set if requested
isInternetConnectionAvailable	Boolean	If true, a data connection is available; could be through any of the transports
wifi	Object	An object with the WiFi status properties:
state	String	Set to either 'connected' or 'disconnected'
ipAddress	String	WiFi IP address
ssid	String	The SSID of the currently connected access point
bssid	String	The BSSID of the currently connected access point
wan	Object	An object with the WAN status properties:
state	String	Set to either 'connected' or 'disconnected'
ipAddress	String	WAN IP address
network	String	Set to 'unknown','unsusable','gprs','edge','umts', 'hsdpa','1x','evdo'
btpan	Object	An object with the Bluetooth PAN status properties:
state	String	Set to either 'connected' or 'disconnected'
ipAddress	String	Bluetooth IP address
panUser	String	Set to the name of the Bluetooth PAN client connected to the device
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Contacts

```
palm://com.palm.contacts/crud
```

The Contacts API provides programmatic access to the Contacts application. It allows you to create, read, update, delete, and list contacts entries. To use this API you must have an account created via the Accounts API.

Many of the APIs will use the `contacts` or `contactslice` objects as input or output arguments.

Contact		
Properties	**Type**	**Description**
`id`	String	Immutable unique identifier for this Contact record
`firstName`	String	
`lastName`	String	
`middleName`	String	
`displayText`	String	
`prefix`	String	
`suffix`	String	
`companyName`	String	
`jobTitle`	String	
`pictureLoc`	String	File path to picture location; 50 × 50 pixels rough size
`pictureLocSquare`	String	File path to picture location; 50 × 50 pixels rough size, but guaranteed to be square
`pictureLocBig`	String	File path to picture location; a full size image
`isAvatarLoc`	String	The original start timestamp of this event in the parent series
`birthday`	String	YYYYMMDD format
`anniversary`	String	YYYYMMDD format
`nickname`	String	
`spouse`	String	
`children`	String	
`notes`	String	
`phoneNumbers`	Array	Array of phone number strings
`emailAddresses`	Array	Array of email address strings
`imNames`	Array	Array of IM names strings
`addresses`	Array	Array of physical address strings
`urls`	Array	Array of urls strings
`customFields`	Array	Array of custom field strings

Contact		
Properties	**Type**	**Description**
`listPic`	String	*Input only*; the local file path of the picture to use as this contact's photo in lists; should be square, roughly 50 × 50 pixels
`incomingPic`	String	*Input only*; the local file path of the picture to use as this contact's high-quality photo
`extraDetails`	String	Generic overflow field for any metadata the third-party application wants to store (will not be displayed in the Contacts application)

Contact Slice		
Properties	**Type**	**Description**
`id`	String	Immutable unique identifier for this Contact record
`firstName`	String	
`lastName`	String	
`middleName`	String	
`displayText`	String	
`prefix`	String	
`suffix`	String	
`nickname`	String	
`companyName`	String	
`pictureLoc`	String	File path to picture location; 50 × 50 pixels rough size.
`pictureLocSquare`	String	File path to picture location; 50 × 50 pixels rough size, but guaranteed to be square

On methods that accept an `id` as an input argument (`get`, `update`, and `delete`), you can alternately provide an `externalId` and an `accountId`.

Method: createContact

Adds the provided record to contacts.

Parameters	Type	Description
`accountId`	String	Immutable unique identifier for this account
`contact`	Object	The contact to save
`externalId`	String	External ID for this contact record
`trackChanges`	Boolean	Marks this change so that it's tracked, and `getChanges` will include this record ID in its response; defaults to false

Response	Type	Description
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
id	String	Immutable unique identifier for this contact record

Method: getContact

Retrieves the details of the named contact.

Parameters	Type	Description
id	String	Immutable unique identifier for this contact
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
contact	Object	Details of the contact record

Method: updateContact

Updates the event details of the named contact.

Parameters	Type	Description
accountId	String	Immutable unique identifier for this account
contact	Object	The contact to save
id	String	Immutable unique identifier for this contact
trackChanges	Boolean	Marks this change so that it's tracked, and getChanges will include this record ID in its response; defaults to false
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: deleteContact

Deletes the named contact.

Parameters	Type	Description
accountId	String	Immutable unique identifier for this account
id	String	Immutable unique identifier for this contact
trackChanges	Boolean	Marks this change so that it's tracked, and getChanges will include this record ID in its response; defaults to false
Response	**Type**	**Description**
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: listContacts

Lists all contacts created by the calling application; you will see only Contact records that you have created.

Parameters	Type	Description
accountId	String	The account reference for use in the Synergy applications
offset	Integer	Start of range
limit	Integer	Number of contacts to return
filter	String	The filter string to search for; matches are done on the firstName, lastName, and companyName fields
Response	**Type**	**Description**
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false
list	Array	Array of contactslice objects

Method: startTracking

Enables change tracking for an account. After this is called, all user-initiated changes to records belonging to this account will be returned by getChanges. Change tracking is incremental, and changes are forgotten each time doneWithChanges is called.

Parameters	Type	Description
accountId	String	The account reference for use in the Synergy applications
trackerId	String	A client-chosen handle if multiple independent change trackers are needed per account
Response	**Type**	**Description**
returnValue	Boolean	If true, the account was successfully created; otherwise there is an error and the account was not created
errorCode	Integer	If returnValue is false, the errorCode provides the error number
errorText	String	An error message; only provided with returnValue set to false

Method: getChanges

Gets a list of contact IDs for records in an account that have been changed or deleted by the user since the last time `doneWithChanges` was called.

Parameters	Type	Description
`accountId`	String	The account reference for use in the Synergy applications
`trackerId`	String	A client-chosen handle if multiple independent change trackers are needed per account
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`errorCode`	Integer	If `returnValue` is false, the `errorCode` provides the error number
`errorText`	String	An error message; only provided with `returnValue` set to false
`token`	String	A string to be passed to `doneWithChanges` after processing of this change set is complete
`changed`	Array	An array of changed contact record objects (`id` only)
`deleted`	Array	An array of deleted contact record objects (`id` and `externalId`)

Method: doneWithChanges

Forgets all changes for an account. Future calls to `getChanges` will only return changes from this point forward.

Parameters	Type	Description
`accountId`	String	The account reference for use in the Synergy applications
`trackerId`	String	A client-chosen handle if multiple independent change trackers are needed per account
`token`	String	A string to be passed to `doneWithChanges` after processing of this change set is complete
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the account was successfully created; otherwise, there is an error and the account was not created
`errorCode`	Integer	If `returnValue` is false, the `errorCode` provides the error number
`errorText`	String	An error message; only provided with `returnValue` set to false

Email

```
palm://com.palm.applicationManager
```

The Email application can be launched through the Application Manager service.

Method: open

Launches the Email application to the compose scene, which will be addressed with the provided email address.

Parameters	Type	Description
`target`	String	A mailto URI formatted as specified in RFC2368
Response	**Type**	**Description**
None		

Method: launch

Launches the Email application to the compose scene.

Parameters	Type	Description
`id`	String	Set to `'com.palm.app.email'`
`params`	Object	Includes a single property:
`summary`	String	Text to display in the subject line
`text`	String	Text to display in the body of the email message
`recipients`	Array	Recipients array, including:
`value`	String	A properly formed email address of the recipient
`type`	String	Set to email
`role`	Integer	Numeric ID for type of recipient where 1=To, 2=CC, and 3=BCC
`contactDisplay`	String	Display name for the recipient
`attachments`	Array	Array of objects containing the property `fullPath` (the full path and file name) and the optional properties `displayName` and `mime Type`
`accountId`	String	The ID of the account from which to send the email message; default account will be used if not provided
Response	**Type**	**Description**
None		

Location Services

```
palm://com.palm.location
```

Palm webOS provides basic location services to get single or multiple location fixes.

Method: getCurrentPosition

You can get the current position sourced from the built in GPS, or through Cell ID or WiFi ID, depending on what's available.

Parameters	Type	Description
accuracy	Integer	Accuracy or precision of the fix: 1 for high precision, 2 for medium (default), and 3 for low
responseTime	Integer	Accuracy or precision of the fix: 1 for less than 5 seconds, 2 for less than 20 seconds (default), and 3 for greater than 20 seconds
maximumAge	Integer	Accept a cached position no older than maximumAge (in seconds); if 0 or not specified, a new fix will always be requested
Response	**Type**	**Description**
errorCode	Integer	Status of service request; if errorCode, a successful request and the following response properties will be available; otherwise, errorCode provides an error number to describe the failure
errorText	String	An error message; only provided with returnValue set to false
timestamp	Double	The time (in milliseconds) when the location fix was retrieved
latitude	Double	The latitude of the location in degrees
longitude	Double	The longitude of the location in degrees
horizAccuracy	Double	Horizontal accuracy (in meters) of the location fix
vertAccuracy	Double	Vertical accuracy (in meters) of the location fix
heading	Double	The compass azimuth (in degrees); set to –1 if unknown
velocity	Double	The velocity (in meters/second); set to –1 if unknown
altitude	Double	The altitude (in meters); set to –1 if unknown

Method: startTracking

Requests a continuous GPS fix by invoking the onSuccess callback with a new location object every time the service determines that the position of the device has changed. In case of error, the service will call the callback with error code set to some value greater than 0.

Parameters	Type	Description
subscribe	Boolean	Set to true for subscriptions, default is false
Response	**Type**	**Description**
errorCode	Integer	Status of service request; if errorCode, a successful request and the following response properties will be available; otherwise, errorCode provides an error number to describe the failure
errorText	String	An error message; only provided with returnValue set to false
timestamp	Double	The time (in milliseconds) when the location fix was retrieved
latitude	Double	The latitude of the location in degrees
longitude	Double	The longitude of the location in degrees
horizAccuracy	Double	Horizontal accuracy (in meters) of the location fix

Response	Type	Description
vertAccuracy	Double	Vertical accuracy (in meters) of the location fix
heading	Double	The compass azimuth (in degrees); set to –1 if unknown
velocity	Double	The velocity (in meters/second); set to –1 if unknown
altitude	Double	The altitude (in meters); set to –1 if unknown

Method: getReverseLocation

Requests a location for the given latitude and longitude.

Parameters	Type	Description
latitude	Double	The latitude of the location in degrees
longitude	Double	The longitude of the location in degrees
Response	**Type**	**Description**
errorCode	Integer	Status of service request; if `errorCode`, a successful request and the following response properties will be available; otherwise `errorCode` provides an error number to describe the failure
errorText	String	An error message; only provided with `returnValue` set to false
address	String	U.S. only, not available in other regions; formatted user-readable address with individual lines separated by a semicolon (;); typically two or three lines

Maps

```
palm://com.palm.applicationManager
```

The Maps application can be launched through the Application Manager service.

Method: open

Launches the Maps application to the map scene, which will be loaded from the results of a query if provided.

Parameters	Type	Description
target	String	URL matching the regular expression: `(.+\.)?google\.(com|[a-z]{2}|com?\.[a-z]{2})(/maps/m|/maps/m/.*)` Or alternately, the target string can be of the form `mapto:location`, where location is a well-formed location
Response	**Type**	**Description**
None		

Method: launch

Launches the Maps application to the map view scene.

Parameters	Type	Description
`id`	String	Set to `'com.palm.app.maps'`
`params`	Object	Includes:
`query`	String	Well-formed expression complying with the Google Map Parameters spec; options include address, latitude/longitude, location, and business search or driving directions
`zoom`	String	Zoom level: numeric value from 1 to 18, with 18 being max zoom in; max zoom level depends on the region (for some regions, zoom level 18 may not be available)
`location`	Object	If specified, map will search for `query` around this location as described by these properties:
		`lat` = latitude in degrees (float)
		`lng` = longitude in degrees (float)
		`acc` = accuracy in meters; optional (float)
		`age` = age of fix in seconds
`type`	String	Sets the display type, currently limited to m for map (default when not set) and k for satellite
`layer`	String	Activates selective overlays, currently limited to t for traffic or null for no overlay
Response	**Type**	**Description**
None		

Messaging

```
palm://com.palm.applicationManager
```

The Messaging application can be launched through the Application Manager service.

Method: launch

Launches the Messaging application to the chat scene.

Parameters	Type	Description
`id`	String	Set to `'com.palm.app.messaging'`
`params`	Object	Includes:
`personId`	String	Contacts ID for the recipient; only apply to the launching the chat view
`contactPointId`	String	Contacts entry ID for the contact method (i.e., mobile phone number, IM address); only apply to the launching the chat view; you can optionally pass a `contactPointId` to set the selected transport when you launch the chat

Parameters	Type	Description
`messageText`	String	The contents of the message
`composeAddress`	String	Mobile phone number; an alternative to the personID/contactPointId above; currently only supports one number
`attachment`	String	Path to a single attachment file; JPG images only
Response	**Type**	**Description**
None		

Phone

```
palm://com.palm.applicationManager
```

The Phone application can be launched through the Application Manager service.

Method: open

Launches the Phone application to the dial scene, prepopulating the dial string if provided.

Parameters	Type	Description
`target`	String	URL of the from `"tel://dialstring"`, where `dialstring` is part or all of a phone number; the number can contain pause and wait characters to indicate that DTMF tones should be sent after the call connects
Response	**Type**	**Description**
None		

Method: launch

Launches the Phone application to the dial scene.

Parameters	Type	Description
`id`	String	Set to `'com.palm.app.phone'`
Response	**Type**	**Description**
None		

Photos

```
palm://com.palm.applicationManager
```

The Photos application can be launched through the Application Manager service.

Method: launch

Launches the Photos application to its album scene.

Parameters	Type	Description
`id`	String	Set to `'com.palm.app.photos'`
Response	**Type**	**Description**
None		

System Properties

```
palm://com.palm.preferences/systemProperties
```

Applications can request a named system property, currently limited to a unique device ID.

Method: getSysProperties

Requests the named property, which is returned in a response object.

Parameters	Type	Description
`key`	String	Options: `"com.palm.properties.nduid"` (device ID)
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the key was retrieved and the value was returned
`errorText`	String	An error message; only provided with `returnValue` set to false

System Service

```
palm://com.palm.systemservice/time
```

The system is designed to expose a set of services allowing applications to access some general system settings.

Method: getSystemTime

Requests the system time, and if the subscribe property is true, it will receive notifications when the timezone changes and/or the system time changes by a significant amount (currently five minutes).

Parameters	Type	Description
`subscribe`	Boolean	Set to true for subscriptions, default is false
Response	**Type**	**Description**
`returnValue`	Boolean	If true, the key was retrieved and the value was returned

Response	Type	Description
errorCode	Integer	Status of service request; if errorCode, a successful request and the following response properties will be available; otherwise, errorCode provides an error number to describe the failure
localtime	Integer	The time for the current timezone (in seconds)
offset	Integer	Offset from the UTC (in minutes)
timezone	String	The current system timezone

System Sounds

```
palm://com.palm.audio/systemsounds
```

The System Sounds service is used to play audio feedback in response to user interaction, with low latency.

Method: playFeedback

Plays a system sound using the feedback stream class. Intended for UI feedback such as UI button clicks, keypad clicks, and similar sounds. The available sound names are enumerated in Table B-4.

Parameters	Type	Description
name	String	Set to the name of the sound to play (see Table B-4 for a complete list of sound names)

Response	Type	Description
returnValue	Boolean	If true, the key was retrieved and the value was returned
errorCode	Integer	Status of service request; if errorCode, a successful request and the following response properties will be available; otherwise, errorCode provides an error number to describe the failure
errorText	String	An error message; only provided with returnValue set to false

Table B-4. System sounds

Names		
appclose	browser_01	pageforward_01
back_01	card_01	shuffle_02
default_425hz	card_02	shuffle_03
delete_01	card_03	shuffle_04
discardingapp_01	card_04	shuffle_05
down2	card_05	shuffle_06
dtmf_0	dtmf_asterisk	shuffle_07

Names		
dtmf_1	dtmf_pound	shuffle_08
dtmf_2	error_01	shuffling_01
dtmf_3	error_02	shutter
dtmf_4	error_03	switchingapps_01
dtmf_5	focusing	switchingapps_02
dtmf_6	launch_01	switchingapps_03
dtmf_7	launch_02	tones_3beeps_otasp_done
dtmf_8	launch_03	unassigned
dtmf_9	pagebacwards	up2

Video

```
palm://com.palm.applicationManager
```

The Video application can be launched through the Application Manager service.

Method: launch

Launches the Video application to play or stream the file located at the target URI, downloading it first if not already on the device. If a URI is not specified, it will launch the video player to its normal starting scene.

Parameters	Type	Description
`id`	String	Set to `'com.palm.app.videoplayer'`
`params`	Object	Includes a single property:
`target`	String	URL of the form *rtsp://`video-file`*, where *`video-file`* is a well-formed URI targeting a video file encoded in a supported video format
`videoTitle`	String	The name of the video to be displayed to the user
`thumbURL`	String	The URL of a thumbnail to be displayed while the video is loading
Response	**Type**	**Description**
None		

Controller APIs

This section includes the application, stage, and scene controller methods used in the book. These are just a subset of the APIs available; refer to the Palm SDK for a complete list of the available APIs.

Each entry includes a very brief description, followed by an enumeration of the arguments used in the method call.

Mojo.Controller.AppController.createStageWithCallback (stageArguments, onCreate, optionalStageType)

Method to create a new stage and be called back when the stage is loaded.

Arguments	Type	Description
`stageArguments`	String or Object	If a string, the name of the new stage; if a stage exists with this name, its contents will be replaced; if an object, it must have a `name` property containing the stage name and may have an `assistantName` property to specify the stage assistant and a `height` property to specify the height of a pop-up alert
`onCreate`	Function	A function that is called once the new stage is fully loaded; it is passed the new stage controller as its first parameter
`optionalStageType`	String	The type of stage to create: `'card'`, `'dashboard'`, `'popup'`, or `'banner'`
Returns	**Type**	**Description**
None		

Mojo.Controller.AppController.getActiveStageController(stageType)

Method to return the first currently focused stage's controller.

Arguments	Type	Description
`stageType`	String	The type of stage to return: `'card'`, `'dashboard'`, `'popup'`, or `'banner'`
Returns	**Type**	**Description**
`stageController`	Object	The stage controller for the active stage

Mojo.Controller.AppController.getScreenOrientation()

Method to return the device orientation.

Arguments	Type	Description
`None`	String	
Returns	**Type**	**Description**
`orientation`	String	The orientation of the device: `'up'`, `'down'`, `'left'`, or `'right'`

Mojo.Controller.AppController.getStageController(stageName)

Method to get the stage controller for a stage.

Arguments	Type	Description
`stageName`	String	The name of the stage
Returns	**Type**	**Description**
`stageController`	Object	

Mojo.Controller.AppController.getStageProxy(stageName)

Function to get a controller or proxy object for a stage. Returns the stage controller if available, but if the stage is still in the process of being created, a proxy object will be returned instead. This proxy implements `delegateToSceneAssistant()`, and will delegate the calls as expected when the stage is available.

Arguments	Type	Description
`stageName`	String	The name of the stage
Returns	**Type**	**Description**
`stageProxy/ stageController`	Object	Returns the stage controller if available, but if the stage is still in the process of being created, a proxy object will be returned instead

Mojo.Controller.AppController.sendToNotificationChain (notificationData)

Sends the passed-in notification data to everyone in the commander stack of the focused window (usually the scene assistant, stage assistant, and application assistant), calling `considerForNotification()`, if present, on each.

Arguments	Type	Description
`notificationData`	Object	JSON object with payload of notification
Returns	**Type**	**Description**
None		

Mojo.Controller.AppController.showBanner(bannerParams, launchArguments, category)

Shows the message text from the `bannerParams` in the banner area. The `launchArguments` will be used to launch or relaunch the application if the banner is touched.

Arguments	Type	Description
`bannerParams`	String or Object	Can be a string, in which case it is simply message text, or an object with the following properties: `messageText` (text to display), `soundClass` (string containing the sound class to use), `soundFile` (partial or full path to a sound file to play), and `icon` (partial or full path to an icon to show)
`launchArguments`	Varies	Arguments sent to the application when it is launched or relaunched if the banner is touched
`category`	String	Value defined by the application; used if you have more than one kind of banner message; since banners are displayed for a fixed length of time (five seconds currently), they can back up if there are more requests made than time to display them—if there is more than one banner notification within a named category, the framework will discard all but the last of them

Returns	Type	Description
None		

Mojo.Controller.SceneController.listen(element, eventType, callback, onCapture)

Wrapper around `Mojo.Event.listen` that additionally will call `get()` on the element parameter if it is a string, converting it to a DOM node.

Arguments	Type	Description
`element`	String or Element	An element reference or DOM ID string identifying the target element
`eventType`	String	String identifier for the event type
`callback`	Function	Function object to be called when the event occurs
`onCapture`	Boolean	Pass true to listen during the capture phase, false to listen during bubbling

Returns	Type	Description
None		

Mojo.Controller.SceneController.get(elementId)

If the `elementId` is a string, calls `document.getElementById()` with that string and returns the result. Otherwise, it returns `elementId`.

Arguments	Type	Description
`elementid`	String	An element reference or DOM ID string identifying the target element

Returns	Type	Description
element	Element	Element referenced by elementId

Mojo.Controller.SceneController.removeRequest(request)

Removes a request from the scene's list of requests to manage; once a scene is popped this will no longer clear the request.

Arguments	Type	Description
request	Object	A request object, typically the one returned from a serviceRequest() call
Returns	**Type**	**Description**
None		

Mojo.Controller.SceneController.serviceRequest(url, options, resubscribe)

Creates a Palm service request that will be automatically cancelled when the scene is popped. The parameters are passed directly to new Mojo.Service.Request().

Arguments	Type	Description
url	String	URI-formatted service name
options	Object	Service-dependent arguments
resubscribe	Boolean	If set to true, will automatically resubscribe when receiving an error from the service
Returns	**Type**	**Description**
request	Object	A request object, important for requests that persist beyond scene's life or to be able to remove the request

Mojo.Controller.SceneController.setupWidget(name, attributes, model)

Registers the given attributes and model to be used with the widget of the given name. Called by scene assistants in their setup methods.

Arguments	Type	Description
name	String	Element ID for a div in which the widget is declared
attributes	Object	Widget attributes; specific contents are dependent on the widget
model	Object	Widget model; specific contents are dependent on the widget
Returns	**Type**	**Description**
None		

Mojo.Controller.SceneController.stopListening(element, eventType, callback, onCapture)

Wrapper around `Mojo.Event.stopListening` that additionally will call `get()` on the element parameter if it is a string, converting it to a DOM node.

Arguments	Type	Description
`element`	String or Element	An element reference or DOM ID string identifying the target element
`eventType`	String	String identifier for the event type
`callback`	Function	Function object passed in previous call to `listen()`
`onCapture`	Boolean	Pass `true` if it was listening during the capture phase; `false` if it was listening during bubbling
Returns	**Type**	**Description**
None		

Mojo.Controller.StageController.activate()

Activates this stage, similar to `window.focus()`. Causes card windows to be maximized.

Arguments	Type	Description
None		
Returns	**Type**	**Description**
None		

Mojo.Controller.StageController.activeScene()

Returns the currently active scene from this stage, if any. If no scenes are active, it returns undefined.

Arguments	Type	Description
None		
Returns	**Type**	**Description**
`sceneController`	Object	The scene controller for the active scene

Mojo.Controller.StageController.deactivate()

Deactivates this stage. Causes card windows to be minimized.

Arguments	Type	Description
None		

Returns	Type	Description
None		

Mojo.Controller.StageController.delegateToSceneAssistant (functionName)

Use to call a method on the assistant of the current scene of this stage. The first parameter is the name of the property that contains the function to call. The remaining parameters are passed to that function. The `this` keyword is bound to the scene assistant for this call; any additional arguments will be passed to the scene assistant's method.

Arguments	Type	Description
`functionName`	Function	Function to be called within the active scene
Returns	**Type**	**Description**
None		

Mojo.Controller.StageController.getScenes()

Returns an array of scene controllers currently on the stack.

Arguments	Type	Description
None		
Returns	**Type**	**Description**
`results`	Array	Array of scene controllers with the bottom scene on the stack at `result[0]`

Mojo.Controller.StageController.getWindowOrientation()

Gets the orientation of the stage's window.

Arguments	Type	Description
None		
Returns	**Type**	**Description**
`orientation`	String	The orientation of the device: `'up'`, `'down'`, `'left'`, or `'right'`

Mojo.Controller.StageController.popScene(returnValue, options)

Removes a scene from the scene stack, passing the return value to the newly revealed scene's `activate` method. Note that this is an asynchronous operation. Any additional arguments are also passed to the new scene.

Arguments	Type	Description
`returnValue`	Object	Passed to the newly activated scene
Returns	**Type**	**Description**
None		

Mojo.Controller.StageController.popScenesTo(targetScene, returnValue, options)

Removes scenes from the scene stack until the target scene is reached, or no scenes remain on the stack. `targetScene` may be either the `SceneController` for the desired scene, the scene DOM ID, or the scene name. If `targetScene` is undefined, all scenes will be popped. Intermediate popped scenes are not reactivated, nor is there any visual transition to signify their removal from the stack. This is an asynchronous operation.

Arguments	Type	Description
`targetScene`	String or Element	Name or DOM ID of scene to be activated
`returnValue`	Object	Passed to the newly activated scene
Returns	**Type**	**Description**
None		

Mojo.Controller.StageController.pushScene(sceneArguments)

Pushes a new scene; the Scene Lifecycle initial setup includes this function. This is an asynchronous operation.

Arguments	Type	Description
`sceneArguments`	String or Element	Either the name of the scene to push or an object with properties including the name of the scene and the ID to use as a DOM ID; all additional arguments are passed to the constructor of the next scene's assistant
Returns	**Type**	**Description**
None		

Mojo.Controller.StageController.setWindowOrientation(orientation)

Sets the orientation of the stage's window.

Arguments	Type	Description
`orientation`	String	The orientation of the device: `'free'`, `'up'`, `'down'`, `'left'` or `'right'`

Returns	Type	Description
None		

Mojo.Controller.StageController.swapScene(sceneArguments)

Pops the current scene and simultaneously pushes a new scene without activating and deactivating any underlying scenes. Note that this is an asynchronous operation.

Arguments	Type	Description
`sceneArguments`	String or Element	Either the name of the scene to push or an object with properties including the name of the scene and the ID to use as a DOM ID; all additional arguments are passed to the constructor of the next scene's assistant
Returns	**Type**	**Description**
None		

Mojo.Controller.StageController.topScene()

Returns the topmost scene from this stage.

Arguments	Type	Description
None		
Returns	**Type**	**Description**
`sceneController`	Object	The scene controller for the top scene

APPENDIX C

Quick Reference—Style Guide

Mojo's styling is automatically provided when you select the Mojo class names within your HTML, or use Mojo widgets. This appendix will help you determine the selectors or properties to change when you want to override the automatic styling. It's best to use this style guide along with the *StyleMatters* sample application provided with the SDK.

The guide is organized into several categories, including:

- Scene Basics
- List Basics
- Containers
- Dividers
- Panels
- Text
- Widgets

Each category is presented in a table with:

- A brief description
- The filename of the CSS style sheet where the style category is defined
- An HTML code sample using the style
- A list of the base selectors that you can use in your CSS to override the framework's styling
- A list of optional selectors
- Required child selectors, which must be embedded with the base selector for the style to be used effectively

These styles may be updated from time to time by Palm as part of an SDK update—you should check the SDK documentation to get the latest information.

Scene Basics

Backdrop

global-base.css

Change the background of an individual scene or every scene within the application.

Scene HTML		
`<div class="my-backdrop"></div>`		
Base selector	**Optional selectors**	**Required child**
`my-backdrop`		
`body`		

Notes:

- `my-backdrop` is a developer-defined selector (you can define any selector name); this technique allows each individual scene to have a unique background.
- Use the `body` selector to style the body element; a simple solution to style all of the scenes in your application.

Fixed Header

global-lists.css

Floating header atop your scene; visually identical to the View Menu.

Scene HTML		
`<div class="palm-header"></div>` `<div class="palm-header-spacer"></div>`		
Base selector	**Optional selectors**	**Required child**
`.palm-header`	`.left`	
	`.right`	
`.palm-header-spacer`		

Page Header

global-lists.css

The topmost element of the scrollable content; commonly used atop preference scenes.

Scene HTML

```
<div class="palm-page-header multi-line">
     <div class="palm-page-header-wrapper">
        <div class="icon"></div>
        <div class="title">
          My title
        </div>
     </div>
</div>
```

Base selector	Optional selectors	Required child
`.palm-page-header`	`.multi-line`	`.palm-page-header-wrapper`
	`.icon`	
`.palm-page-header-wrapper`		
`.palm-page-header-wrapper > .icon`		
`.palm-page-header-wrapper > .title`	`.left`	
	`.center`	
	`.right`	
	`.truncating-text`	

Notes:

- `palm-page-header-wrapper` is a child element with a specified margin to compensate for the padding effect of `-webkit-border-image`.
- `palm-page-header-wrapper > .icon` should be a 32 × 32px PNG centered on the div.

Scrim

global.css

A translucent layer used to obscure background UI when modal foreground UI is layered on top of the current scene.

Scene HTML

```
<div class="palm-scrim"></div>
```

Base selector	Optional selectors	Required child
`.palm-scrim`	`.app-menu`	
	`.dialog`	
	`.submenu-popup`	
	`.menu-panel`	
	`.picker-popup`	

Scroll Fades

In the absence of view or command menus, these fades at the edge of your scene indicate more content.

Scene HTML

```
<div class="my-scene-fade-top" x-mojo-scroll-fade="top"></div>
<div class="my-scene-fade-bottom" x-mojo-scroll-fade="bottom"></div>
```

Base selector	Optional selectors	Required child
None		

Notes:

- Use a high z-index (100,000) if you want your fades to draw above all else, and use the style `-webkit-palm-mouse-target: ignore;` to allow taps to pass through these fades to underlying content.

View/Command Menus

global-menus.css

Float menus at either the top or bottom of your scene, and the gradient fades behind them.

Scene HTML

None

Base selector	Optional selectors	Required child
`.palm-menu`		
`.palm-menu.view-menu`	`.palm-default`	
	`.palm-white`	
`.palm-menu.view-menu > .palm-menu-fade`		
`.palm-menu.command-menu`	`.palm-default`	
	`.palm-white`	
`.palm-menu.command-menu > .palm-menu-fade`		
`.palm-menu-button`	`.selected`	
`.palm-menu-spacer`		

List Basics

Add/Remove Rows

global-lists.css

The "add item" row appended to a list and "remove item" buttons for removable rows.

Scene HTML

None

Base selector	Optional selectors	Required child
`.list-item-add-button`		
`.list-item-remove-button`		

Lists and Rows

global-lists.css

Rows stacked vertically within lists, designed for legibility and touch interaction.

Scene HTML

```
<!-- Within Scene    -->
<div id="my-list" x-mojo-element="List"></div>

<!-- Within List template -->
<div class="palm-row"   x-mojo-touch-feedback="delayed">
     <div class="palm-row-wrapper">
       <!-- row content here -->
     </div>
</div>
```

Base selector	Optional selectors	Required child
`.palm-list`		
`.palm-row`	`.first`	`.palm-row-wrapper`
	`.last`	
	`.single`	
	`.no-divider`	
	`.no-separator`	
	`.disabled`	
`.palm-row-wrapper`	`.textfield-group`	
`.palm-row-wrapper > .title`	`.left`	
	`.right`	
	`.truncating-text`	

Notes:

- `.palm-row-wrapper`: child element with a specified margin to compensate for the padding effect of `-webkit-border-image`.

Separators

global-lists.css

Thin lines that visually separate rows.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.palm-row`		

Reordering Rows

global-lists.css

The space behind the reordered items and the item you're moving.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.palm-drag-spacer`		
`.palm-row.palm-reorder-element`		

Swipe to Delete

global-lists.css

The space revealed when you swipe to delete, which may contain confirmation buttons.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.palm-row.palm-swipe-delete`		
`.palm-row.palm-swipe-delete` `.palm-swipe-delete-button`		
`.palm-row.palm-swipe-delete` `.palm-swipe-undo-button`		

Touch Feedback

global-lists.css

Displaying alternate background images and content styling in response to user interaction.

Scene HTML

```
<div class="palm-row" x-mojo-touch-feedback="delayed">
  <div class="palm-row-wrapper">
    <!-- row content here -->
  </div>
</div>
```

Base selector	Optional selectors	Required child
`.palm-row.selected`		

Notes:

- When touched, rows using `x-mojo-touch-feedback` will display a selection graphic implemented using `-webkit-border-image` with a 9-tile image (41 × 49 pixels).
- For items within scrollable content, use `delayed` feedback. For fixed elements that don't scroll, `immediate` feedback is an option. Use `immediatePersistent` or `delayed Persistent` only if you require exacting control of when feedback is removed (which must be done manually).

Touch Feedback with Groups

global-lists.css

Displaying alternate background images and content styling in response to user interaction with rows within in a palm-group.

Scene HTML

```
<!--  First Row -->
<div class="palm-group">
    <div class="palm-row first" x-mojo-touch-feedback="delayed">
      <div class="palm-row-wrapper">
        <!-- row content here -->
      </div>
    </div>
</div>

<!--  Last Row -->
<div class="palm-group">
    <div class="palm-row last" x-mojo-touch-feedback="delayed">
      <div class="palm-row-wrapper">
        <!-- row content here -->
      </div>
    </div>
</div>
```

Scene HTML

```
<!--  Single Row -->
<div class="palm-group">
    <div class="palm-row single" x-mojo-touch-feedback="delayed">
      <div class="palm-row-wrapper">
        <!-- row content here -->
      </div>
    </div>
</div>
```

Base selector	Optional selectors	Required child
`.palm-group .palm-row.selected.first`		
`.palm-group .palm-row.selected.last`		
`.palm-group .palm-row.selected.single`		

Notes:

- The first row within a group requires a selection graphic with rounded top corners.
- The last row within a group requires a selection graphic with rounded bottom corners.
- A single row within a group requires a selection graphic with rounded corners.

Containers

Drawers

global-lists.css

Hide UI or lists with an area that animates open and closed.

Scene HTML

```
<div id="my-id" x-mojo-element="Drawer"></div>
```

Base selector	Optional selectors	Required child
`.palm-drawer-container`		`.palm-drawer-contents`
`.palm-drawer-contents`	`.label`	

Labeled Groups

global-lists.css

Visually group a list with a label.

Scene HTML

```
<div class="palm-group">
   <div class="palm-group-title">
    My Title
   </div>
   <div class="palm-list">
    <!-- rows -->
   </div>
  </div>
```

Base selector	Optional selectors	Required child
`.palm-group`		`.palm-group-title`

Unlabeled Groups

global-lists.css

Visually group a list without a label.

Scene HTML

```
<div class="palm-group unlabeled">
   <div class="palm-list">
    <!-- rows -->
   </div>
</div>
```

Base selector	Optional selectors	Required child
`.palm-group.unlabeled`		

Dividers

Alphabetical Dividers

global-dividers.css

Divide a scene or list of rows with a bold line containing a single character.

Scene HTML

```
<div class="palm-divider alpha">
  <div>#{dividerLabel}</div>
</div>
```

Base selector	Optional selectors	Required child
`.palm-divider.alpha`		

Collapsible Dividers

global-dividers.css

Divide a scene or list of rows with dividers that control corresponding drawers of content.

Scene HTML

```
<table class="palm-divider collapsible">
  <tr>
    <td class="left"></td>
    <td class="label">My label</td>
    <td class="line"></td>
    <td><div class="palm-arrow-closed arrow_button"></div></td>
    <td class="right"></td>
  </tr>
</table>
```

Base selector	Optional selectors	Required child
`.palm-divider.collapsible`	`.label`	
	`.line`	
	`.right`	

Labeled Dividers

global-dividers.css

Divide a scene or list of rows with a bold line and label.

Scene HTML

```
<table class="palm-divider labeled">
  <tr>
   <td class="left"></td>
   <td class="label">
    My title
   </td>
   <td class="right"></td>
  </tr>
</table>
```

Base selector	Optional selectors	Required child
`.palm-divider.labeled`	`.left`	
	`.right`	

Solid Dividers

global-dividers.css

Divide a scene or list of rows with a bold line.

Scene HTML

```
<div class="palm-divider"></div>
```

Base selector	Optional selectors	Required child
`.palm-divider`		

Panels

Dashboard Panel

global-notifications.css

A dashboard panel containing UI, custom messages, and images.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.palm-dashboard-icon-container`		
`.palm-dashboard-text-container`	`.selected`	
`.palm-dashboard-icon`	`.alert`	
`.palm-dashboard-text`		

Dialog

global.css and *global-buttons.css*

A modal panel pinned to the bottom of the scene.

Scene HTML

```
<div class="palm-dialog-content">
  <div class="palm-dialog-title">Title</div>
  <div class="palm-dialog-separator"></div>
  <div class="palm-dialog-message">Message</div>
</div>
<div class="palm-dialog-buttons">
  <!-- buttons -->
</div>
```

Base selector	Optional selectors	Required child
`.palm-dialog-content`		`.palm-dialog-title`
		`.palm-dialog-separator`
		`.palm-dialog-message`
`.palm-dialog-title`	`.un-capitalize`	
`.palm-dialog-buttons`		`.palm-button`
`.palm-dialog-separator`		
`.palm-dialog-message`		

Notes:

- Custom dialog box template with title, message, and buttons.

Menu Panel

global-menus.css

A pop-up panel containing UI or lists, floating under the view or command menu.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.palm-menu-panel`		`.palm-menu-panel-fade-top`
		`.palm-menu-panel-fade-bottom`
		`.palm-menu-panel-wrapper`
`.palm-menu-panel-wrapper`		`.palm-list`

Submenu

global-menus.css

A pop-up panel containing UI or lists, floating above all other scene UI.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.palm-popup`		
`.palm-popup-container`		
`.palm-popup-content`	`.popup-item-checkmark`	
	`.chosen`	
	`.palm-divider`	
`.palm-popup-icon`	`.left`	
	`.right`	

Text

Basic Text Styles

global.css

Body copy and informational text.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.palm-text-wrapper`		`.palm-body-text`
`.palm-body-text`		
`.palm-body-title`		
`.palm-info-text`	`.single`	

Notes:

- `.palm-text-wrapper`: use this wrapper to contain multiple divs of styled text for proper padding.

Capitalization

global.css

Some Mojo widgets and styles shift strings to uppercase or apply capitalization.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.capitalize`		
`.un-capitalize`		

Notes:

- Use `.capitalize` for CSS title case capitalization and `.un-capitalize` to override autocapitalization in buttons, dialog box titles, and page headers.

Fonts

global-base.css

Mojo uses the Prelude font family.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`body`	`.condensed`	
`button`	`.oblique`	
`input`		
`textarea`		

Truncation

global-base.css

Force text to fit within the available space, with an ellipsis added as needed.

Scene HTML		
None		
Base selector	**Optional selectors**	**Required child**
`.truncating-text`		

Widgets

Button

global-buttons.css

Scene HTML		

```
<div class="palm-button" x-mojo-touch-feedback="delayed">
  <div class="palm-button-wrapper">
    Button label
  </div>
</div>
```

Base selector	Optional selectors	Required child
`.palm-button`	`.primary`	`.palm-button-wrapper`
	`.secondary`	
	`.dismiss`	
	`.negative`	
	`.affirmative`	
	`.disabled`	
	`.selected`	

Check Box

global.css

Scene HTML		

```
<div id="my-id" class="left"   x-mojo-element="CheckBox"></div>
```

Base selector	Optional selectors	Required child
`.checkbox`	`.left`	
	`.right`	

Base selector	Optional selectors	Required child
	.disabled	
	.true	

Notes:

- Use the classes `.left` and `.right` on the widget when placing the check box into a `palm-row` with `title` text.

Date Picker

global-widget-mvpicker.css

Scene HTML		
`<div id="my-id" x-mojo-element="DatePicker"></div>`		
Base selector	**Optional selectors**	**Required child**
`.mv-picker-capsule`		

Integer Picker

global-widget-mvpicker.css

Scene HTML		
`<div id="my-id" x-mojo-element="IntegerPicker"></div>`		
Base selector	**Optional selectors**	**Required child**
`.mv-picker-capsule`		

Filter Field

global-textfields.css

Scene HTML		
`<div id="my-id"  x-mojo-element="FilterField"></div>`		
Base selector	**Optional selectors**	**Required child**
`.filter-field-container`	`.filter-field-activity-spinner`	
	`.search-term`	
	`.mag-glass-icon`	

List Selector

global-menu.css

Scene HTML		
`<div id="my-id"  x-mojo-element="ListSelector"></div>`		
Base selector	**Optional selectors**	**Required child**
`.palm-list-selector`	`.label`	
	`.right`	
	`.title`	
`.palm-popup-container`		
`.palm-row`	`.list-selector-triangle`	

Progress Pill

global.css

Scene HTML		
`<div id="my-id" x-mojo-element="ProgressPill"></div>`		
Base selector	**Optional selectors**	**Required child**
`.progress-pill-background`		
`.progress-pill-progress`		

Radio Button

global.css

Scene HTML		
`<div id="my-id" x-mojo-element="Radio"></div>`		
Base selector	**Optional selectors**	**Required child**
`.palm-radiobutton`	`.selected`	

Slider

global.css

Scene HTML		
`<div id='my-id' x-mojo-element="Slider"  class="palm-slider"></div>`		
Base selector	**Optional selectors**	**Required child**
`.palm-slider`		
`.palm-slider-button`	`.selected`	
`.palm-slider-background`		

Spinner

global.css

Scene HTML

```
<div id="my-id"   x-mojo-element="Spinner"></div>
```

Base selector	Optional selectors	Required child
.palm-activity-indicator-small		
.palm-activity-indicator-large		

Text Field

global-textfields.css

Scene HTML

```
<!-- Single Text Field -->
<div id="my-id"   x-mojo-element="TextField"></div>

<!-- Grouped Text Field -->
<div class="palm-row">
    <div class="palm-row-wrapper">
     <div class="textfield-group" x-mojo-focus-highlight="true">
      <div class=""title"">
            <div class="truncating-text" x-mojo-element="TextField"></div>
      </div>
      </div>
    </div>
</div>
```

Base selector	Optional selectors	Required child
.textfield-group	.focused	
	.title	
	.label	

Time Picker

global-widget-mvpicker.css

Scene HTML

```
<div id="my-id"   x-mojo-element="TimePicker"></div>
```

Base selector	Optional selectors	Required child
.mv-picker-capsule		

Toggle Button

global.css

Scene HTML		
`<div id="my-id"  x-mojo-element="ToggleButton"></div>`		
Base selector	**Optional selectors**	**Required child**
`.sliding-toggle-container`		
`.toggle-text`		
`.toggle-button`	`.true`	
	`.false`	

APPENDIX D

News Application Source Code

Source code from the News applications is used throughout the book to show examples of the Mojo APIs, widgets, and services as they might be used within an application. The complete source code is provided here so that you can see the full application. The source code and image files are also posted at *http://oreilly.com/catalog/9780596155254/*.

Some of the sample code used in the early chapters is replaced in later chapters. Only the final version the application is shown here.

News Application Directory Structure

Some of the JavaScript code has been broken to accommodate the book layout, and may not execute as broken. It's best to retrieve the code from the oreilly.com link above.

```
news
  app
    assistants
      app-assistant.js
      dashboard-assistant.js
      feedList-assistant.js
      preferences-assistant.js
      stage-assistant.js
      storyList-assistant.js
      storyView-assistant.js
    models
      cookie.js
      feeds.js
    views
      dashboard
        dashboard-scene.html
        item-info.html
```

```
    feedList
      feedRowTemplate.html
      feedListTemplate.html
      addFeed-dialog.html
      feedList-scene.html
    preferences
      preferences-scene.html
    storyList
      storyList-scene.html
      storyListTemplate.html
      storyRowTemplate.html
    storyView
      storyView-scene.html
appinfo.json
framework_config.json
icon.png
images
  cal-selector-header-gray.png
  dashboard-icon-news.png
  details-closed-arrow.png
  details-open-arrow.png
  feedlist-newitem.png
  filter-search-light-bg.png
  header-icon-news.png
  icon-rssfeed.png
  info-icon.png
  list-icon-rssfeed.png
  menu-icon-back.png
  menu-icon-forward.png
  menu-icon-web.png
  news-icon.png
  palm-drawer-background-2.png
  url-icon.png
index.html
resources
   es_us
     appinfo.json
     strings.json
     views
       feedList
         addFeed-dialog.html
         feedList-scene.html
       preferences
         preferences-scene.html
sources.json
stylesheets
  News.css
```

news/app/assistants/app-assistant.js

```
/*  AppAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Responsible for app startup, handling launch points and updating news feeds.
```

```
    Major components:
    - setup; app startup including preferences, initial load of feed data
        from the Depot and setting alarms for periodic feed updates
    - handleLaunch; launch entry point for initial launch, feed update
        alarm, dashboard or banner tap
    - handleCommand; handles app menu selections

    Data structures:
    - globals; set of persistant data used throughout app
    - Feeds Model; handles all feedlist updates, db handling and default data
    - Cookies Model; handles saving and restoring preferences
    App architecture:
    - AppAssistant; handles startup, feed list management and app menu management
    - FeedListAssistant; handles feedList navigation, search, feature feed
    - StoryListAssistant; handles single feed navigation
    - StoryViewAssistant; handles single story navigation
    - PreferencesAssistant; handles preferences display and changes
    - DashboardAssistant; displays latest new story and new story count
*/

//  ------------------------------------------------------------------
//    GLOBALS
//  ------------------------------------------------------------------

//  News namespace
News = {};

// Constants
News.unreadStory = "unReadStyle";
News.versionString = "1.0";
News.MainStageName = "newsStage";
News.DashboardStageName = "newsDashboard";
News.errorNone =    "0";                    // No error, success
News.invalidFeedError = "1";                // Not RSS2, RDF (RSS1), or ATOM

// Global Data Structures

// Persistent Globals - will be saved across app launches
News.featureFeedEnable = true;              // Enables feed rotation
News.featureStoryInterval = 5000;           // Feature Interval (in ms)
News.notificationEnable = true;             // Enables notifcations
News.feedUpdateBackgroundEnable = false;    // Enable device wakeup
News.feedUpdateInterval = "00:15:00";       // Feed update interval

//  Session Globals - not saved across app launches
News.feedListChanged = false;               // Triggers update to Depot db
News.feedListUpdateInProgress = false;      // Feed update is in progress
News.featureStoryTimer = null;              // Timer for story rotations
News.dbUpdate = "";                         // Default is no update
News.wakeupTaskId = 0;                      // Id for wakeup tasks

// Setup App Menu for all scenes; all menu actions handled in
//  AppAssistant.handleCommand()
News.MenuAttr = {omitDefaultItems: true};
```

```
News.MenuModel = {
    visible: true,
    items: [
        {label: $L("About News..."), command: "do-aboutNews"},
        Mojo.Menu.editItem,
        {label: $L("Update All Feeds"), checkEnabled: true,
          command: "do-feedUpdate"},
        {label: $L("Preferences..."), command: "do-newsPrefs"},
        Mojo.Menu.helpItem
    ]
};

function AppAssistant (appController) {

}

//  ------------------------------------------------------
//  setup - all startup actions:
//    - Setup globals with preferences
//    - Set up application menu; used in every scene
//    - Open Depot and use contents for feedList
//    - Initiate alarm for first feed update

AppAssistant.prototype.setup = function() {

    // initialize the feeds model
    this.feeds = new Feeds();
    this.feeds.loadFeedDb();

    // load preferences and globals from saved cookie
    News.Cookie.initialize();

    // Set up first timeout alarm
    this.setWakeup();

};

//  ------------------------------------------------------
//  handleLaunch - called by the framework when the application is asked to launch
//    - First launch; create card stage and first first scene
//    - Update; after alarm fires to update feeds
//    - Notification; after user taps banner or dashboard
//
AppAssistant.prototype.handleLaunch = function (launchParams) {
    Mojo.Log.info("ReLaunch");

    var cardStageController =
      this.controller.getStageController(News.MainStageName);
    var appController = Mojo.Controller.getAppController();

    if (!launchParams)  {
        // FIRST LAUNCH
        // Look for an existing main stage by name.
        if (cardStageController) {
            // If it exists, just bring it to the front by focusing its window.
```

```
            Mojo.Log.info("Main Stage Exists");
            cardStageController.popScenesTo("feedList");
            cardStageController.activate();
        } else {
            // Create a callback function to set up the new main stage
            // once it is done loading. It is passed the new stage controller
            // as the first parameter.
            var pushMainScene = function(stageController) {
                stageController.pushScene("feedList", this.feeds);
            };
            Mojo.Log.info("Create Main Stage");
            var stageArguments = {name: News.MainStageName, lightweight: true};
            this.controller.createStageWithCallback(stageArguments,
                pushMainScene.bind(this), "card");
        }
    }
    else  {
        Mojo.Log.info("com.palm.app.news -- Wakeup Call", launchParams.action);
        switch (launchParams.action) {

           // UPDATE FEEDS
           case "feedUpdate"  :
               // Set next wakeup alarm
               this.setWakeup();

              // Update the feed list
              Mojo.Log.info("Update FeedList");
              this.feeds.updateFeedList();
           break;

           // NOTIFICATION
           case "notification"  :
                Mojo.Log.info("com.palm.app.news -- Notification Tap");
               if (cardStageController) {

                   // If it exists, find the appropriate story list and activate it.
                   Mojo.Log.info("Main Stage Exists");
                   cardStageController.popScenesTo("feedList");
                   cardStageController.pushScene("storyList", this.feeds.list,
                       launchParams.index);
                   cardStageController.activate();
               } else {

                   // Create a callback function to set up a new main stage,
                   // push the feedList scene and then the appropriate story list
                   var pushMainScene2 = function(stageController) {
                      stageController.pushScene("feedList", this.feeds);
                      stageController.pushScene("storyList", this.feeds.list,
                          launchParams.index);
                   };
                   Mojo.Log.info("Create Main Stage");
                   var stageArguments2 = {name: News.MainStageName,
                     lightweight: true};
                   this.controller.createStageWithCallback(stageArguments2,
                     pushMainScene2.bind(this), "card");
```

```
                }
            break;

        }
    }
};

// ---------------------------------------
// handleCommand - called to handle app menu selections
//
AppAssistant.prototype.handleCommand = function(event) {
    var stageController = this.controller.getActiveStageController();
    var currentScene = stageController.activeScene();

    if (event.type == Mojo.Event.commandEnable) {
        if (News.feedListUpdateInProgress && (event.command == "do-feedUpdate")) {
            event.preventDefault();
        }
    }

    else {

        if(event.type == Mojo.Event.command) {
            switch(event.command) {

                case "do-aboutNews":
                    currentScene.showAlertDialog({
                       onChoose: function(value) {},
                       title: $L("News — v#{version}").interpolate
                         ({version: News.versionString}),
                       message: $L("Copyright 2008-2009, Palm Inc."),
                       choices:[
                           {label:$L("OK"), value:""}
                       ]
                     });
                break;

                case "do-newsPrefs":
                    stageController.pushScene("preferences");
                break;

                case "do-feedUpdate":
                    this.feeds.updateFeedList();
                break;
            }
        }
    }
};

// ---------------------------------------------------------------------
// setWakeup - called to setup the wakeup alarm for background feed updates
//   if preferences are not set for a manual update (value of "00:00:00")
AppAssistant.prototype.setWakeup = function() {
        if (News.feedUpdateInterval !== "00:00:00")   {
            this.wakeupRequest =
```

```
                new Mojo.Service.Request("palm://com.palm.power/timeout", {
                  method: "set",
                  parameters: {
                      "key": "com.palm.app.news.update",
                      "in": News.feedUpdateInterval,
                      "wakeup": News.feedUpdateBackgroundEnable,
                      "uri": "palm://com.palm.applicationManager/open",
                      "params": {
                          "id": "com.palm.app.news",
                          "params": {"action": "feedUpdate"}
                      }
                  },
                  onSuccess:  function(response){
                      Mojo.Log.info("Alarm Set Success", response.returnValue);
                      News.wakeupTaskId = Object.toJSON(response.taskId);
                  },
                  onFailure:  function(response){
                      Mojo.Log.info("Alarm Set Failure",
                          response.returnValue, response.errorText);
                  }
              });
             Mojo.Log.info("Set Update Timeout");
          }
};
```

news/app/assistants/dashboard-assistant.js

```
/*  Dashboard Assistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Responsible for posting that last feed with new stories,
    including the new story count and the latest story headline.

    Arguments:
    - feedlist; News feed list
    - selectedFeedIndex; target feed

    Other than posting the new story, the dashboard will call the
    News apps handleLaunch with a "notification" action when the
    dashboard is tapped, and the dashboard window will be closed.
*/

function DashboardAssistant(feedlist, selectedFeedIndex) {
    this.list = feedlist;
    this.index = selectedFeedIndex;
    this.title = this.list[this.index].title;
    this.message = this.list[this.index].stories[0].title;
    this.count = this.list[this.index].newStoryCount;
}

DashboardAssistant.prototype.setup = function() {
    this.displayDashboard(this.title, this.message, this.count);
    this.switchHandler = this.launchMain.bindAsEventListener(this);
    this.controller.listen("dashboardinfo", Mojo.Event.tap, this.switchHandler);
```

```
    this.stageDocument = this.controller.stageController.document;
    this.activateStageHandler = this.activateStage.bindAsEventListener(this);
    Mojo.Event.listen(this.stageDocument, Mojo.Event.stageActivate,
        this.activateStageHandler);
    this.deactivateStageHandler = this.deactivateStage.bindAsEventListener(this);
    Mojo.Event.listen(this.stageDocument, Mojo.Event.stageDeactivate,
        this.deactivateStageHandler);
};

DashboardAssistant.prototype.cleanup = function() {
    // Release event listeners
    this.controller.stopListening("dashboardinfo", Mojo.Event.tap,
        this.switchHandler);
    Mojo.Event.stopListening(this.stageDocument, Mojo.Event.stageActivate,
        this.activateStageHandler);
    Mojo.Event.stopListening(this.stageDocument, Mojo.Event.stageDeactivate,
        this.deactivateStageHandler);
};

DashboardAssistant.prototype.activateStage = function() {
    Mojo.Log.info("Dashboard stage Activation");
    this.storyIndex = 0;
    this.showStory();
};

DashboardAssistant.prototype.deactivateStage = function() {
    Mojo.Log.info("Dashboard stage Deactivation");
    this.stopShowStory();
};

// Update scene contents, using render to insert the object into an HTML template
DashboardAssistant.prototype.displayDashboard = function(title, message, count) {
    var info = {title: title, message: message, count: count};
    var renderedInfo = Mojo.View.render({
        object: info,
        template: "dashboard/item-info"
    });
    var infoElement = this.controller.get("dashboardinfo");
    infoElement.innerHTML = renderedInfo;
};

DashboardAssistant.prototype.launchMain = function() {
    Mojo.Log.info("Tap to Dashboard");
    var appController = Mojo.Controller.getAppController();
    appController.assistant.handleLaunch({action:"notification",
     index:this.index});
    this.controller.window.close();
};

// showStory - rotates stories shown in dashboard panel, every 3 seconds.
DashboardAssistant.prototype.showStory = function() {
    Mojo.Log.info("Dashboard Story Rotation", this.timer, this.storyIndex);

    this.interval = 3000;
```

```
    //    If timer is null, just restart the timer and use the most recent story
    //  or the last one displayed;
    if (!this.timer)    {
        this.timer = this.controller.window.setInterval(this.showStory.bind(this),
            this.interval);
    }

    // Else, get next story in list and update the story in the dashboard display
    else {
        // replace with test for unread story
        this.storyIndex = this.storyIndex+1;
        if(this.storyIndex >= this.list[this.index].stories.length) {
            this.storyIndex = 0;
        }

        this.message = this.list[this.index].stories[this.storyIndex].title;
        this.displayDashboard(this.title, this.message, this.count);
    }
};

DashboardAssistant.prototype.stopShowStory = function() {
    if (this.timer) {
        this.controller.window.clearInterval(this.timer);
        this.timer = undefined;
    }
};

// Update dashboard scene contents - external method
DashboardAssistant.prototype.updateDashboard = function(selectedFeedIndex) {
    this.index = selectedFeedIndex;
    this.title = this.list[this.index].title;
    this.message = this.list[this.index].stories[0].title;
    this.count = this.list[this.index].newStoryCount;
    this.displayDashboard(this.title, this.message, this.count);
};
```

news/app/assistants/feedList-assistant.js

```
/*  FeedListAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Main scene for News app. Includes AddDialog-assistant for handling
    feed entry and then feedlist-assistant and supporting functions.

    Major components:
    - AddDialogAssisant; Scene assistant for add feed dialog and handlers
    - FeedListAssistant; manages feedlists
    - List Handlers - delete, reorder and add feeds
    - Feature Feed - functions for rotating and showing feature stories
    - Search - functions for searching across the entire feedlist database

    Arguments:
    - feeds; Feeds object
```

```
*/


// ---------------------------------------------------------------------------
// AddDialogAssistant - simple controller for adding new feeds to the list
//   when the "Add..." is selected on the feedlist. The dialog will
//   allow the user to enter the feed's url and optionally a name. When
//   the "Ok" button is tapped, the new feed will be loaded. If no errors
//   are encountered, the dialog will close otherwise the error will be
//   posted and the user encouraged to try again.
//
function AddDialogAssistant(sceneAssistant, feeds, index) {
    this.feeds = feeds;
    this.sceneAssistant = sceneAssistant;

    //  If an index is provided then this is an edit feed, not add feed
    //  so provide the existing title, url and modify the dialog title
    if (index !== undefined) {
        this.title = this.feeds.list[index].title;
        this.url = this.feeds.list[index].url;
        this.feedIndex = index;
        this.dialogTitle = $L("Edit News Feed");
    }
    else {
        this.title = "";
        this.url = "";
        this.feedIndex = null;
        this.dialogTitle = $L("Add News Feed Source");
    }
}

AddDialogAssistant.prototype.setup = function(widget) {
        this.widget = widget;

        // Set the dialog title to either Edit or Add Feed
        var addFeedTitleElement =
          this.sceneAssistant.controller.get("add-feed-title");
        addFeedTitleElement.innerHTML = this.dialogTitle;

        // Setup text field for the new feed's URL
        this.sceneAssistant.controller.setupWidget(
            "newFeedURL",
            {
                  hintText: $L("RSS or ATOM feed URL"),
                  autoFocus: true,
                  autoReplace: false,
                  textCase: Mojo.Widget.steModeLowerCase,
                  enterSubmits: false
            },
            this.urlModel = {value : this.url});

        // Setup text field for the new feed's name
        this.sceneAssistant.controller.setupWidget(
            "newFeedName",
            {
```

```
                hintText: $L("Title (Optional)"),
                autoReplace: false,
                textCase: Mojo.Widget.steModeTitleCase,
                enterSubmits: false
            },
            this.nameModel = {value : this.title});

        // Setup OK  & Cancel buttons
        //   OK button is an activity button which will be active
        //   while processing and adding feed. Cancel will just
        //   close the scene
        this.okButtonModel = {label: $L("OK"), disabled: false};
        this.sceneAssistant.controller.setupWidget("okButton",
          {type: Mojo.Widget.activityButton}, this.okButtonModel);
        this.okButtonActive = false;
        this.okButton = this.sceneAssistant.controller.get("okButton");
        this.checkFeedHandler = this.checkFeed.bindAsEventListener(this);
        this.sceneAssistant.controller.listen("okButton", Mojo.Event.tap,
          this.checkFeedHandler);

        this.cancelButtonModel = {label: $L("Cancel"), disabled: false};
        this.sceneAssistant.controller.setupWidget("cancelButton",
          {type: Mojo.Widget.defaultButton}, this.cancelButtonModel);
        this.sceneAssistant.controller.listen("cancelButton", Mojo.Event.tap,
          this.widget.mojo.close);
};

// checkFeed  - called when OK button is clicked implying a valid feed URL
// has been entered.
AddDialogAssistant.prototype.checkFeed = function() {

        if (this.okButtonActive === true)  {
            // Shouldn't happen, but log event if it does and exit
            Mojo.Log.info("Multiple Check Feed requests");
            return;
        }

        // Check entered URL and name to confirm a valid and supported feedlist
        Mojo.Log.info("New Feed URL Request: ", this.urlModel.value);

        // Check for "http://" on front or other prefix; assume any string of
        // 1 to 5 alpha characters followed by ":" is ok, else prepend "http://"
        var url = this.urlModel.value;
        if (/^[a-z]{1,5}:/.test(url) === false)    {
            // Strip any leading slashes
            url = url.replace(/^\/{1,2}/,"");
            url = "http://"+url;
        }

        // Update the entered URL & model
        this.urlModel.value = url;
        this.sceneAssistant.controller.modelChanged(this.urlModel);

        // If the url is the same, then assume that it's just a title change,
        // update the feed title and close the dialog. Otherwise update the feed.
```

```
        if (this.feedIndex && this.feeds.list[this.feedIndex].url ==
          this.urlModel.value) {
            this.feeds.list[this.feedIndex].title = this.nameModel.value;
            this.sceneAssistant.feedWgtModel.items = this.feeds.list;
            this.sceneAssistant.controller.modelChanged(
              this.sceneAssistant.feedWgtModel);
            this.widget.mojo.close();
        }
        else {

            this.okButton.mojo.activate();
            this.okButtonActive = true;
            this.okButtonModel.label = "Updating Feed";
            this.okButtonModel.disabled = true;
            this.sceneAssistant.controller.modelChanged(this.okButtonModel);

            var request = new Ajax.Request(url, {
                method: "get",
                evalJSON: "false",
                onSuccess: this.checkSuccess.bind(this),
                onFailure: this.checkFailure.bind(this)
            });
        }
};

// checkSuccess - Ajax request success
AddDialogAssistant.prototype.checkSuccess = function(transport) {
    Mojo.Log.info("Valid URL - HTTP Status", transport.status);

    // DEBUG - Work around due occasional Ajax XML error in response.
    if (transport.responseXML === null && transport.responseText !== null) {
            Mojo.Log.info("Request not in XML format - manually converting");
            transport.responseXML =
              new DOMParser().parseFromString(transport.responseText, "text/xml");
     }

    var feedError = News.errorNone;

    //  If a new feed, push the entered feed data on to the feedlist and
    //  call processFeed to evaluate it.
    if (this.feedIndex === null) {
        this.feeds.list.push({title:this.nameModel.value, url:this.urlModel.value,
            type:"", value:false, numUnRead:0, stories:[]});
        // processFeed - index defaults to last entry
        feedError = this.feeds.processFeed(transport);
    }
    else    {
        this.feeds.list[this.feedIndex] = {title:this.nameModel.value,
          url:this.urlModel.value,
            type:"", value:false, numUnRead:0, stories:[]};
        feedError = this.feeds.processFeed(transport, this.feedIndex);
    }

    // If successful processFeed returns errorNone
    if (feedError === News.errorNone)    {
```

```
        // update the widget, save the DB and exit
        this.sceneAssistant.feedWgtModel.items = this.feeds.list;
        this.sceneAssistant.controller.modelChanged(this.sceneAssistant.feedWgtModel);
        this.feeds.storeFeedDb();
        this.widget.mojo.close();
    }
    else    {
        // Feed can't be processed - remove it but keep the dialog open
        this.feeds.list.pop();
        if (feedError == News.invalidFeedError)    {
            Mojo.Log.warn("Feed ",
                this.urlModel.value, " isn't a supported feed type.");
            var addFeedTitleElement = this.controller.get("add-feed-title");
            addFeedTitleElement.innerHTML = $L("Invalid Feed Type - Please Retry");
        }

        this.okButton.mojo.deactivate();
        this.okButtonActive = false;
        this.okButtonModel.buttonLabel = "OK";
        this.okButtonModel.disabled = false;
        this.sceneAssistant.controller.modelChanged(this.okButtonModel);
    }
};

// checkFailure  - Ajax request failure
AddDialogAssistant.prototype.checkFailure = function(transport) {
    // Log error and put message in status area
    Mojo.Log.info("Invalid URL - HTTP Status", transport.status);
    var addFeedTitleElement = this.controller.get("add-feed-title");
    addFeedTitleElement.innerHTML = $L("Invalid Feed Type - Please Retry");
};

// cleanup  - remove listeners
AddDialogAssistant.prototype.cleanup = function() {
    this.sceneAssistant.controller.stopListening("okButton", Mojo.Event.tap,
        this.checkFeedHandler);
    this.sceneAssistant.controller.stopListening("cancelButton", Mojo.Event.tap,
        this.widget.mojo.close);
};



//    ------------------------------------------------------------------------
//
//    FeedListAssistant - main scene handler for news feedlists
//
function FeedListAssistant(feeds) {
    this.feeds = feeds;
    this.appController = Mojo.Controller.getAppController();
    this.stageController = this.appController.getStageController(News.MainStageName);
}

FeedListAssistant.prototype.setup =  function() {

    // Setup App Menu
```

```
this.controller.setupWidget(Mojo.Menu.appMenu, News.MenuAttr, News.MenuModel);

// Setup the search filterlist and handlers;
this.controller.setupWidget("startSearchField",
    {
        itemTemplate: "storyList/storyRowTemplate",
        listTemplate: "storyList/storyListTemplate",
        filterFunction: this.searchList.bind(this),
        renderLimit: 70,
        delay: 350
    },
    this.searchFieldModel = {
        disabled: false
    });

this.viewSearchStoryHandler = this.viewSearchStory.bindAsEventListener(this);
this.controller.listen("startSearchField", Mojo.Event.listTap,
    this.viewSearchStoryHandler);
this.searchFilterHandler = this.searchFilter.bindAsEventListener(this);
this.controller.listen("startSearchField", Mojo.Event.filter,
    this.searchFilterHandler, true);

// Setup header, drawer, scroller and handler for feature feeds

this.featureDrawerHandler = this.toggleFeatureDrawer.bindAsEventListener(this);
this.controller.listen("featureDrawer", Mojo.Event.tap,
    this.featureDrawerHandler);

this.controller.setupWidget("featureFeedDrawer", {},
  this.featureFeedDrawer = {open:News.featureFeedEnable});

this.featureScrollerModel = {
    scrollbars: false,
    mode: "vertical"
    };

this.controller.setupWidget("featureScroller", this.featureScrollerModel);
this.readFeatureStoryHandler =
  this.readFeatureStory.bindAsEventListener(this);
this.controller.listen("featureStoryDiv", Mojo.Event.tap,
    this.readFeatureStoryHandler);

        // If feature story is enabled, then set the icon to open
if (this.featureFeedDrawer.open === true) {
    this.controller.get("featureDrawer").className = "featureFeed-open";
} else  {
    this.controller.get("featureDrawer").className = "featureFeed-close";
}

// Setup the feed list, but it's empty
this.controller.setupWidget("feedListWgt",
     {
        itemTemplate:"feedList/feedRowTemplate",
        listTemplate:"feedList/feedListTemplate",
        addItemLabel:$L("Add..."),
```

```
            swipeToDelete:true,
            renderLimit: 40,
            reorderable:true
        },
        this.feedWgtModel = {items: this.feeds.list});

    // Setup event handlers: list selection, add, delete and reorder feed entry
    this.showFeedHandler = this.showFeed.bindAsEventListener(this);
    this.controller.listen("feedListWgt", Mojo.Event.listTap,
        this.showFeedHandler);
    this.addNewFeedHandler = this.addNewFeed.bindAsEventListener(this);
    this.controller.listen("feedListWgt", Mojo.Event.listAdd,
        this.addNewFeedHandler);
    this.listDeleteFeedHandler = this.listDeleteFeed.bindAsEventListener(this);
    this.controller.listen("feedListWgt", Mojo.Event.listDelete,
        this.listDeleteFeedHandler);
    this.listReorderFeedHandler = this.listReorderFeed.bindAsEventListener(this);
    this.controller.listen("feedListWgt", Mojo.Event.listReorder,
        this.listReorderFeedHandler);

    // Setup spinner for feedlist updates
    this.controller.setupWidget("feedSpinner", {property: "value"});

    // Setup listeners for minimize/maximize events
    this.activateWindowHandler = this.activateWindow.bindAsEventListener(this);
    Mojo.Event.listen(this.controller.stageController.document,
      Mojo.Event.activate, this.activateWindowHandler);
    this.deactivateWindowHandler = this.deactivateWindow.bindAsEventListener(this);
    Mojo.Event.listen(this.controller.stageController.document,
      Mojo.Event.deactivate, this.deactivateWindowHandler);

    // Setup up feature story index to first story of the first feed
    this.featureIndexFeed = 0;
    this.featureIndexStory = 0;
};

// activate - handle portrait/landscape orientation, feature feed layout and rotation
FeedListAssistant.prototype.activate =  function() {

    // Set Orientation to free to allow rotation
    if (this.controller.stageController.setWindowOrientation) {
        this.controller.stageController.setWindowOrientation("free");
    }

    if (News.feedListChanged === true)    {
        this.feedWgtModel.items = this.feeds.list;
        this.controller.modelChanged(this.feedWgtModel, this);
        this.controller.modelChanged(this.searchFieldModel, this);

        // Don't update the database here; it's slow enough that it lags the UI;
        // wait for a feature story update to mask the update effect
    }

    // If there's some stories in the feed list, then start
    // the story rotation even if the featureFeed is disabled as we'll use
```

```
    // the rotation timer to update the DB
    if(this.feeds.list[this.featureIndexFeed].stories.length > 0) {
        var splashScreenElement = this.controller.get("splashScreen");
        splashScreenElement.hide();
        this.showFeatureStory();
    }
};

// deactivate - always turn off feature timer
FeedListAssistant.prototype.deactivate =  function() {
    Mojo.Log.info("FeedList deactivating");
    this.clearTimers();
};

// cleanup - always turn off timers, and save this.feeds.list contents
FeedListAssistant.prototype.cleanup =  function() {
    Mojo.Log.info("FeedList cleaning up");

    // Save the feed list on close, as a precaution; shouldn't be needed;
    //don't wait for results
    this.feeds.storeFeedDb();

    // Clear feature story timer and activity indicators
    this.clearTimers();

    // Remove event listeners
    this.controller.stopListening("startSearchField", Mojo.Event.listTap,
        this.viewSearchStoryHandler);
    this.controller.stopListening("startSearchField", Mojo.Event.filter,
        this.searchFilterHandler, true);
    this.controller.stopListening("featureDrawer", Mojo.Event.tap,
        this.featureDrawerHandler);
    this.controller.stopListening("feedListWgt", Mojo.Event.listTap,
        this.showFeedHandler);
    this.controller.stopListening("feedListWgt", Mojo.Event.listAdd,
        this.addNewFeedHandler);
    this.controller.stopListening("feedListWgt", Mojo.Event.listDelete,
        this.listDeleteFeedHandler);
    this.controller.stopListening("feedListWgt", Mojo.Event.listReorder,
        this.listReorderFeedHandler);
    Mojo.Event.stopListening(this.controller.stageController.document,
      Mojo.Event.activate, this.activateWindowHandler);
    Mojo.Event.stopListening(this.controller.stageController.document,
      Mojo.Event.deactivate, this.deactivateWindowHandler);
};

FeedListAssistant.prototype.activateWindow = function() {
    Mojo.Log.info("Activate Window");
    this.feedWgtModel.items = this.feeds.list;
    this.controller.modelChanged(this.feedWgtModel);

    // If stories exist in the this.featureIndexFeed, start the rotation
    // if not started
    if ((this.feeds.list[this.featureIndexFeed].stories.length > 0) &&
     (News.featureStoryTimer === null)) {
```

```
        var splashScreenElement = this.controller.get("splashScreen");
        splashScreenElement.hide();
        this.showFeatureStory();
    }
};

FeedListAssistant.prototype.deactivateWindow = function() {
    Mojo.Log.info("Deactivate Window");
    this.clearTimers();
};

// -------------------------------------------------------------------------
// List functions for Delete, Reorder and Add
//
// listDeleteFeed - triggered by deleting a feed from the list and updates
// the feedlist to reflect the deletion
//
FeedListAssistant.prototype.listDeleteFeed =  function(event) {
    Mojo.Log.info("News deleting ", event.item.title,".");

    var deleteIndex = this.feeds.list.indexOf(event.item);
    this.feeds.list.splice(deleteIndex, 1);
    News.feedListChanged = true;

    // Adjust the feature story index if needed:
    // - feed that falls before feature story feed is deleted
    // - feature story feed itself is deleted (default back to first feed)
    if (deleteIndex == this.featureIndexFeed)    {
        this.featureIndexFeed = 0;
        this.featureIndexStory = 0;
    } else    {
        if (deleteIndex < this.featureIndexFeed)    {
            this.featureIndexFeed--;
        }
    }
};

// listReorderFeed - triggered re-ordering feed list and updates the
// feedlist to reflect the changed order
FeedListAssistant.prototype.listReorderFeed =  function(event) {
    Mojo.Log.info("com.palm.app.news - News moving ", event.item.title,".");

    var fromIndex = this.feeds.list.indexOf(event.item);
    var toIndex = event.toIndex;
    this.feeds.list.splice(fromIndex, 1);
    this.feeds.list.splice(toIndex, 0, event.item);
    News.feedListChanged = true;

    // Adjust the feature story index if needed:
    // - feed that falls after featureIndexFeed is moved before it
    // - feed before is moved after
    // - the feature story feed itself is moved
    if (fromIndex > this.featureIndexFeed && toIndex <= this.featureIndexFeed) {
        this.featureIndexFeed++;
    } else {
```

```
      if (fromIndex < this.featureIndexFeed && toIndex > this.featureIndexFeed) {
          this.featureIndexFeed--;
      }    else    {
            if (fromIndex == this.featureIndexFeed)    {
                this.featureIndexFeed = toIndex;
            }
      }
    }
};

// addNewFeed - triggered by "Add..." item in feed list
FeedListAssistant.prototype.addNewFeed = function() {

        this.controller.showDialog({
            template: "feedList/addFeed-dialog",
            assistant: new AddDialogAssistant(this, this.feeds)
        });

};

// ------------------------------------------------------------------------------
// clearTimers - clears timers used in this scene when exiting the scene
FeedListAssistant.prototype.clearTimers = function()    {
    if(News.featureStoryTimer !== null) {
        this.controller.window.clearInterval(News.featureStoryTimer);
        News.featureStoryTimer = null;
    }

    // Clean up any active update spinners
    for (var i=0; i<this.feeds.list.length; i++) {
            this.feeds.list[i].value = false;
    }
    this.controller.modelChanged(this.feedWgtModel);

};

// ---------------------------------------------------------------------
// considerForNotification - called by the framework when a notification
//  is issued; look for notifications of feed updates and update the
//  feedWgtModel to reflect changes, update the feed's spinner model
FeedListAssistant.prototype.considerForNotification = function(params){
    if (params && (params.type == "update"))    {
        this.feedWgtModel.items = this.feeds.list;
        this.feeds.list[params.feedIndex].value = params.update;
        this.controller.modelChanged(this.feedWgtModel);

        // If stories exist, start the rotation if not started
        if ((this.feeds.list[this.featureIndexFeed].stories.length > 0) &&
         (News.featureStoryTimer === null)) {
            var splashScreenElement = this.controller.get("splashScreen");
            splashScreenElement.hide();
            this.showFeatureStory();

        }
    }
```

```
    return undefined;
};

// ------------------------------------------------------------------------
// Feature story functions
//
// showFeatureStory - simply rotate the stories within the
// featured feed, which the user can set in their preferences.
FeedListAssistant.prototype.showFeatureStory = function() {

    // If timer is null, either initial story or restarting. Start with
    // previous story..
    if (News.featureStoryTimer === null)    {
        News.featureStoryTimer =
          this.controller.window.setInterval(this.showFeatureStory.bind(this),
            News.featureStoryInterval);
    }

    else {
        this.featureIndexStory = this.featureIndexStory+1;
        if(this.featureIndexStory >=
          this.feeds.list[this.featureIndexFeed].stories.length) {
            this.featureIndexStory = 0;
            this.featureIndexFeed = this.featureIndexFeed+1;
            if (this.featureIndexFeed >= this.feeds.list.length)    {
                this.featureIndexFeed = 0;
            }
        }
    }

    var summary = this.feeds.list[this.featureIndexFeed].
      stories[this.featureIndexStory].text.replace(/(<([^>]+)>)/ig,"");
    summary = summary.replace(/http:\S+/ig,"");
    var featureStoryTitleElement = this.controller.get("featureStoryTitle");
    featureStoryTitleElement.innerHTML =
      unescape(this.feeds.list[this.featureIndexFeed].
      stories[this.featureIndexStory].title);
    var featureStoryElement = this.controller.get("featureStory");
    featureStorySummaryElement.innerHTML = summary;

    // Because this is periodic and not tied to a screen transition, use
    // this to update the db when changes have been made

    if (News.feedListChanged === true)    {
        this.feeds.storeFeedDb();
        News.feedListChanged = false;
    }

};

// readFeatureStory - handler when user taps on feature story; will push storyView
//  with the current feature story.
FeedListAssistant.prototype.readFeatureStory = function() {
    this.stageController.pushScene("storyView",
      this.feeds.list[this.featureIndexFeed], this.featureIndexStory);
```

```
};

// toggleFeatureDrawer - handles taps to the featureFeed drawer. Toggle
//   drawer and icon class to reflect drawer state.
FeedListAssistant.prototype.toggleFeatureDrawer =  function(event) {
       var featureDrawer = this.controller.get("featureDrawer");
       if (this.featureFeedDrawer.open === true) {
           this.featureFeedDrawer.open = false;
           News.featureFeedEnable = false;
           featureDrawer.className = "featureFeed-close";
       } else {
           this.featureFeedDrawer.open = true;
           News.featureFeedEnable = true;
           featureDrawer.className = "featureFeed-open";
       }
       this.controller.modelChanged(this.featureFeedDrawer);
       News.Cookie.storeCookie();                // Update News saved preferences
};

// ---------------------------------------------------------------------
// Search Functions
//
// searchFilter - triggered by entry into search field. First entry will
//  hide the main feedList scene - clearing the entry will restore the scene.
//
FeedListAssistant.prototype.searchFilter = function(event)    {
    Mojo.Log.info("Got search filter: ", event.filterString);
    var feedListMainElement = this.controller.get("feedListMain");
    if (event.filterString !== "")    {
        //    Hide rest of feedList scene to make room for search results
        feedListMainElement.hide();
    }    else    {
        //    Restore scene when search string is null
        feedListMainElement.show();
    }

};

// viewSearchStory - triggered by tapping on an entry in the search results
// list  will push the storyView scene with the tapped story.
//
FeedListAssistant.prototype.viewSearchStory = function(event)    {
    var searchList = {title: $L("Search for: ")+this.filter,
      stories: this.entireList};
    var storyIndex = this.entireList.indexOf(event.item);

    Mojo.Log.info("Search display selected story with title = ",
        searchList.title, "; Story index - ", storyIndex);
    this.stageController.pushScene("storyView", searchList, storyIndex);

};

// searchList - filter function called from search field widget to update the
//  results list. This function will build results list by matching the
//  filterstring to the story titles and text content, and then return the
```

```
//  subset of the list based on offset and size requested by the widget.
//
FeedListAssistant.prototype.searchList = function(filterString, listWidget,
   offset, count)    {

    var subset = [];
    var totalSubsetSize = 0;

    this.filter = filterString;

    //    If search string is null, return empty list, else build results
    if (filterString !== "")    {

        // Search database for stories with the search string; push matches
        var items = [];

        // Comparison function for matching strings in next for loop
        var hasString = function(query, s) {
            if(s.text.toUpperCase().indexOf(query.toUpperCase())>=0) {
                return true;
            }
            if(s.title.toUpperCase().indexOf(query.toUpperCase())>=0) {
                return true;
            }
            return false;
        };

        for (var i=0; i<this.feeds.list.length; i++) {
            for (var j=0; j<this.feeds.list[i].stories.length; j++) {
                if(hasString(filterString, this.feeds.list[i].stories[j])) {
                    var sty = this.feeds.list[i].stories[j];
                    items.push(sty);
                }
            }
        }

    this.entireList = items;
    Mojo.Log.info("Search list asked for items: filter=",
        filterString, " offset=", offset, " limit=", count);

    // Cut down results to just the window asked for by the widget
    var cursor = 0;
        while (true) {
            if (cursor >= this.entireList.length) {
                break;
            }

            if (subset.length < count && totalSubsetSize >= offset) {
                subset.push(this.entireList[cursor]);
            }
            totalSubsetSize++;
            cursor++;
        }
    }
```

```
    // Update List
    listWidget.mojo.noticeUpdatedItems(offset, subset);

    // Update filter field count of items found
    listWidget.mojo.setLength(totalSubsetSize);
    listWidget.mojo.setCount(totalSubsetSize);

};

// ------------------------------------------------------------------------------
// Show feed and popup menu handler
//
// showFeed - triggered by tapping a feed in the this.feeds.list.
//   Detects taps on the unReadCount icon; anywhere else,
//   the scene for the list view is pushed. If the icon is tapped,
//   put up a submenu for the feedlist options
FeedListAssistant.prototype.showFeed = function(event) {
        var target = event.originalEvent.target.id;
        if (target !== "info") {
          this.stageController.pushScene("storyList", this.feeds.list,
            event.index);
        }
        else  {
            var myEvent = event;
            var findPlace = myEvent.originalEvent.target;
            this.popupIndex = event.index;
            this.controller.popupSubmenu({
              onChoose:  this.popupHandler,
              placeNear: findPlace,
              items: [
                {label: $L("All Unread"), command: "feed-unread"},
                {label: $L("All Read"), command: "feed-read"},
                {label: $L("Edit Feed"), command: "feed-edit"},
                {label: $L("New Card"), command: "feed-card"}
                ]
              });
        }
};

// popupHandler - choose function for feedPopup
FeedListAssistant.prototype.popupHandler = function(command) {
        var popupFeed=this.feeds.list[this.popupIndex];
        switch(command) {
            case "feed-unread":
                Mojo.Log.info("Popup - unread for feed:", popupFeed.title);

                for (var i=0; i<popupFeed.stories.length; i++ ) {
                    popupFeed.stories[i].unreadStyle = News.unreadStory;
                }
                popupFeed.numUnRead = popupFeed.stories.length;
                this.controller.modelChanged(this.feedWgtModel);
                break;

            case "feed-read":
                Mojo.Log.info("Popup - read for feed:", popupFeed.title);
```

```
                for (var j=0; j<popupFeed.stories.length; j++ ) {
                    popupFeed.stories[j].unreadStyle = "";
                }
                popupFeed.numUnRead = 0;
                this.controller.modelChanged(this.feedWgtModel);
                break;

            case "feed-edit":
                Mojo.Log.info("Popup edit for feed:", popupFeed.title);
                this.controller.showDialog({
                    template: "feedList/addFeed-dialog",
                    assistant: new AddDialogAssistant(this, this.feeds,
                      this.popupIndex)
                });
                break;

            case "feed-card":
                Mojo.Log.info("Popup tear off feed to new card:",
                  popupFeed.title);

                var newCardStage = "newsCard"+this.popupIndex;
                var cardStage = this.appController.getStageController(newCardStage);
                var feedList = this.feeds.list;
                var feedIndex = this.popupIndex;
                if(cardStage) {
                    Mojo.Log.info("Existing Card Stage");
                    cardStage.popScenesTo();
                    cardStage.pushScene("storyList", this.feeds.list, feedIndex);
                    cardStage.activate();
                } else  {
                    Mojo.Log.info("New Card Stage");
                    var pushStoryCard = function(stageController){
                        stageController.pushScene("storyList", feedList, feedIndex);
                    };
                    this.appController.createStageWithCallback({
                          name: newCardStage,
                          lightweight: true
                        },
                        pushStoryCard, "card");
                }
                break;

        }
};
```

news/app/assistants/preferences-assistant.js

```
/*  Preferences - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Preferences - Handles preferences scene, where the user can:
      - select the featured feed rotation interval
      - select the interval for feed updates
      - enable or disable background feed notifications
```

```
      - enable or disable device wakeup

    App Menu is disabled in this scene.

*/

function PreferencesAssistant() {

}

PreferencesAssistant.prototype.setup = function() {

    // Setup Integer Picker to pick feature feed rotation interval
    this.controller.setupWidget("featureFeedDelay",
        {
            label:    $L("Rotation (in seconds)"),
            modelProperty:    "value",
            min: 1,
            max: 20
        },
        this.featureDelayModel = {
            value : News.featureStoryInterval/1000
        });

    this.changeFeatureDelayHandler =
      this.changeFeatureDelay.bindAsEventListener(this);
    this.controller.listen("featureFeedDelay", Mojo.Event.propertyChange,
        this.changeFeatureDelayHandler);

    // Setup list selector for UPDATE INTERVAL
    this.controller.setupWidget("feedCheckIntervalList",
        {
            label: $L("Interval"),
            choices: [
                {label: $L("Manual Updates"),   value: "00:00:00"},
                {label: $L("5 Minutes"),        value: "00:05:00"},
                {label: $L("15 Minutes"),       value: "00:15:00"},
                {label: $L("1 Hour"),           value: "01:00:00"},
                {label: $L("4 Hours"),          value: "04:00:00"},
                {label: $L("1 Day"),            value: "23:59:59"}
            ]
        },
        this.feedIntervalModel = {
            value : News.feedUpdateInterval
        });

    this.changeFeedIntervalHandler =
      this.changeFeedInterval.bindAsEventListener(this);
    this.controller.listen("feedCheckIntervalList", Mojo.Event.propertyChange,
        this.changeFeedIntervalHandler);

    // Toggle for enabling notifications for new stories during feed updates
    this.controller.setupWidget("notificationToggle",
        {},
        this.notificationToggleModel = {
```

```
            value: News.notificationEnable
        });
    this.changeNotificationHandler =
      this.changeNotification.bindAsEventListener(this);
    this.controller.listen("notificationToggle", Mojo.Event.propertyChange,
        this.changeNotificationHandler);

    // Toggle for enabling feed updates while the device is asleep
    this.controller.setupWidget("bgUpdateToggle",
        {},
        this.bgUpdateToggleModel = {
            value: News.feedUpdateBackgroundEnable
        });

    this.changeBgUpdateHandler =
      this.changeBgUpdate.bindAsEventListener(this);
    this.controller.listen("bgUpdateToggle", Mojo.Event.propertyChange,
        this.changeBgUpdate);
};

// Deactivate - save News preferences and globals
PreferencesAssistant.prototype.deactivate = function() {
    News.Cookie.storeCookie();
};

// Cleanup - remove listeners
PreferencesAssistant.prototype.cleanup = function() {
this.controller.stopListening("featureFeedDelay",
        Mojo.Event.propertyChange, this.changeFeatureDelayHandler);
    this.controller.stopListening("feedCheckIntervalList",
        Mojo.Event.propertyChange, this.changeFeedIntervalHandler);
    this.controller.stopListening("notificationToggle",
        Mojo.Event.propertyChange, this.changeNotificationHandler);
    this.controller.stopListening("bgUpdateToggle",
        Mojo.Event.propertyChange, this.changeBgUpdate); };

//    changeFeatureDelay - Handle changes to the feature feed interval
PreferencesAssistant.prototype.changeFeatureDelay = function(event) {
    Mojo.Log.info("Preferences Feature Delay Handler; value = ",
        this.featureDelayModel.value);

    //  Interval is in milliseconds
    News.featureStoryInterval = this.featureDelayModel.value*1000;

    // If timer is active, restart with new value
    if(News.featureStoryTimer !== null) {
        this.controller.window.clearInterval(News.featureStoryTimer);
        News.featureStoryTimer = null;
    }
};

//    changeFeedInterval    - Handle changes to the feed update interva;
PreferencesAssistant.prototype.changeFeedInterval = function(event) {
    Mojo.Log.info("Preferences Feed Interval Handler; value = ",
        this.feedIntervalModel.value);
```

```
    News.feedUpdateInterval = this.feedIntervalModel.value;
};

//    changeNotification - disables/enables notifications
PreferencesAssistant.prototype.changeNotification = function(event) {
    Mojo.Log.info("Preferences Notification Toggle Handler; value = ",
        this.notificationToggleModel.value);
    News.notificationEnable = this.notificationToggleModel.value;
};

//    changeBgUpdate - disables/enables background wakeups
PreferencesAssistant.prototype.changeBgUpdate = function(event) {
    Mojo.Log.info("Preferences Background Update Toggle Handler; value = ",
        this.bgUpdateToggleModel.value);
    News.feedUpdateBackgroundEnable = this.bgUpdateToggleModel.value;
};
```

news/app/assistants/storyList-assistant.js

```
/*  StoryListAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Displays the feed's stories in a list, user taps display the
    selected story in the storyView scene. Major components:
    - Setup view menu to move to next or previous feed
    - Search filter; perform keyword search within feed list
    - Story View; push story scene when a story is tapped
    - Update; handle notifications if feedlist has been updated

    Arguments:
    - feedlist; Feeds.list array of all feeds
    - selectedFeedIndex; Feed to be displayed
*/

function StoryListAssistant(feedlist, selectedFeedIndex) {
    this.feedlist = feedlist;
    this.feed = feedlist[selectedFeedIndex];
    this.feedIndex = selectedFeedIndex;
    Mojo.Log.info("StoryList entry = ", this.feedIndex);
    Mojo.Log.info("StoryList feed = " + Object.toJSON(this.feed));
}


StoryListAssistant.prototype.setup =  function() {
    this.stageController = this.controller.stageController;
    // Setup scene header with feed title and next/previous feed buttons. If
    // this is the first feed, suppress Previous menu; if last, suppress Next menu
    var    feedMenuPrev = {};
    var    feedMenuNext = {};

    if (this.feedIndex > 0)    {
        feedMenuPrev = {
            icon: "back",
            command: "do-feedPrevious"
```

```
    };
} else    {
    // Push empty menu to force menu bar to draw on left (label is the force)
    feedMenuPrev = {icon: "", command: "", label: "  "};
}

if (this.feedIndex < this.feedlist.length-1)    {
    feedMenuNext = {
        iconPath: "images/menu-icon-forward.png",
        command: "do-feedNext"
    };
} else    {
    // Push empty menu to force menu bar to draw on right (label is the force)
    feedMenuNext = {icon: "", command: "", label: "  "};
}

this.feedMenuModel =     {
    visible: true,
    items:     [{
        items: [
            feedMenuPrev,
            { label: this.feed.title, width: 200 },
            feedMenuNext
        ]
    }]
};

this.controller.setupWidget(Mojo.Menu.viewMenu,
    { spacerHeight: 0, menuClass:"no-fade" }, this.feedMenuModel);

// Setup App Menu
this.controller.setupWidget(Mojo.Menu.appMenu, News.MenuAttr, News.MenuModel);

// Setup the search filterlist and handlers;
this.controller.setupWidget("storyListSearch",
    {
        itemTemplate: "storyList/storyRowTemplate",
        listTemplate: "storyList/storyListTemplate",
        filterFunction: this.searchList.bind(this),
        renderLimit: 70,
        delay: 350
    },
    this.searchFieldModel = {
        disabled: false
    });

this.viewSearchStoryHandler = this.viewSearchStory.bindAsEventListener(this);
this.controller.listen("storyListSearch", Mojo.Event.listTap,
    this.viewSearchStoryHandler);
this.searchFilterHandler = this.searchFilter.bindAsEventListener(this);
this.controller.listen("storyListSearch", Mojo.Event.filter,
    this.searchFilterHandler, true);

// Setup story list with standard news list templates.
this.controller.setupWidget("storyListWgt",
```

```
        {
            itemTemplate: "storyList/storyRowTemplate",
            listTemplate: "storyList/storyListTemplate",
            swipeToDelete: false,
            renderLimit: 40,
            reorderable: false
        },
        this.storyModel = {
            items: this.feed.stories
        }
    );

    this.readStoryHandler = this.readStory.bindAsEventListener(this);
    this.controller.listen("storyListWgt", Mojo.Event.listTap,
        this.readStoryHandler);
};

StoryListAssistant.prototype.activate =  function() {
    // Update list models in case unReadCount has changed
    this.controller.modelChanged(this.storyModel);
};

StoryListAssistant.prototype.cleanup =  function() {
    // Remove event listeners
    this.controller.stopListening("storyListSearch", Mojo.Event.listTap,
        this.viewSearchStoryHandler);
    this.controller.stopListening("storyListSearch", Mojo.Event.filter,
        this.searchFilterHandler, true);
    this.controller.stopListening("storyListWgt", Mojo.Event.listTap,
        this.readStoryHandler);
};

// readStory - when user taps on displayed story, push storyView scene
StoryListAssistant.prototype.readStory = function(event) {
    Mojo.Log.info("Display selected story = ", event.item.title,
        "; Story index = ", event.index);
    this.stageController.pushScene("storyView", this.feed, event.index);
};

// handleCommand - handle next and previous commands
StoryListAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case "do-feedNext":
                this.nextFeed();
                break;
            case "do-feedPrevious":
                this.previousFeed();
                break;
        }
    }
};

// nextFeed - Called when the user taps the next menu item
StoryListAssistant.prototype.nextFeed = function(event) {
```

```
    this.stageController.swapScene(
        {
            transition: Mojo.Transition.crossFade,
            name: "storyList"
        },
        this.feedlist,
        this.feedIndex+1);
};

// previousFeed - Called when the user taps the previous menu item
StoryListAssistant.prototype.previousFeed = function(event) {
    this.stageController.swapScene(
        {
            transition: Mojo.Transition.crossFade,
            name: "storyList"
        },
        this.feedlist,
        this.feedIndex-1);
};

// searchFilter - triggered by entry into search field. First entry will
//   hide the main storyList scene and clearing the entry will restore the scene.
StoryListAssistant.prototype.searchFilter = function(event)    {
    var storyListSceneElement = this.controller.get("storyListScene");
    if (event.filterString !== "")    {
        //    Hide rest of storyList scene to make room for search results
        storyListSceneElement.hide();
    }    else    {
        //    Restore scene when search string is null
        storyListSceneElement.show();
    }

};

// viewSearchStory - triggered by tapping on an entry in the search results list.
StoryListAssistant.prototype.viewSearchStory = function(event)    {
    var searchList =
      {title: $L("Search for: #{filter}").interpolate({filter: this.filter}),
      stories: this.entireList};

    var storyIndex = this.entireList.indexOf(event.item);

    this.stageController.pushScene("storyView", searchList, storyIndex);

};

// searchList - filter function called from search field widget to update
// results list. This function will build results list by matching the
// filterstring to story titles and text content, and return the subset
// of list based on offset and size requested by the widget.t.
StoryListAssistant.prototype.searchList = function(filterString, listWidget,
  offset, count)    {

    var subset = [];
    var totalSubsetSize = 0;
```

```
    this.filter = filterString;

    // If search string is null, then return empty list, else build results list
    if (filterString !== "")    {

        // Search database for stories with the search string
        // and push on to the items array
        var items = [];

        // Comparison function for matching strings in next for loop
        var hasString = function(query, s) {
            if(s.text.toUpperCase().indexOf(query.toUpperCase())>=0) {
                return true;
            }
            if(s.title.toUpperCase().indexOf(query.toUpperCase())>=0) {
                return true;
            }
            return false;
        };

        for (var j=0; j<this.feed.stories.length; j++) {
            if(hasString(filterString, this.feed.stories[j])) {
                var sty = this.feed.stories[j];
                items.push(sty);
            }
        }

        this.entireList = items;

        Mojo.Log.info("Search list asked for items: filter=", filterString,
            " offset=", offset, " limit=", count);

        // Cut down the list results to just the window asked for by the widget
        var cursor = 0;
        while (true) {
            if (cursor >= this.entireList.length) {
                break;
            }

            if (subset.length < count && totalSubsetSize >= offset) {
                subset.push(this.entireList[cursor]);
            }
            totalSubsetSize++;
            cursor++;
        }
    }

    // Update List
    listWidget.mojo.noticeUpdatedItems(offset, subset);

    // Update filter field count of items found
    listWidget.mojo.setLength(totalSubsetSize);
    listWidget.mojo.setCount(totalSubsetSize);
};
```

```
// considerForNotification - called when a notification is issued; if this
// feed has been changed, then update it.
StoryListAssistant.prototype.considerForNotification = function(params){
    if (params && (params.type == "update"))    {
        if ((params.feedIndex == this.feedIndex) && (params.update === false)) {
            this.storyModel.items = this.feed.stories;
            this.controller.modelChanged(this.storyModel);
        }
    }
    return undefined;
};
```

news/app/assistants/storyView-assistant.js

```
/*  StoryViewAssistant - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Passed a story element, displays that element in a full scene view and offers
    options for next story (right command menu button), previous story (left
    command menu button) and to launch story URL in the browser (view menu) or
    share story via email or messaging. Major components:
    - StoryView; display story in main scene
    - Next/Previous; command menu options to go to next or previous story
    - Web; command menu option to display original story in browser
    - Share; command menu option to share story by messaging or email

    Arguments:
    - storyFeed; Selected feed from which the stories are being viewed
    - storyIndex; Index of selected story to be put into the view
*/

function StoryViewAssistant(storyFeed, storyIndex) {
    this.storyFeed = storyFeed;
    this.storyIndex = storyIndex;
}

// setup - set up menus
StoryViewAssistant.prototype.setup = function() {
    this.stageController = this.controller.stageController;

    this.storyMenuModel = {
        items: [
            {iconPath: "images/url-icon.png", command: "do-webStory"},
            {},
            {items: []},
            {},
            {icon: "send", command: "do-shareStory"}
        ]};

        if (this.storyIndex > 0)    {
           this.storyMenuModel.items[2].items.push({
             icon: "back",
             command: "do-viewPrevious"
```

```
            });
        } else {
            this.storyMenuModel.items[2].items.push({
              icon: "",
               command: "",
               label: "  "
             });
        }

        if (this.storyIndex < this.storyFeed.stories.length-1)    {
            this.storyMenuModel.items[2].items.push({
              icon: "forward",
              command: "do-viewNext"}
            );
        } else {
            this.storyMenuModel.items[2].items.push({
              icon: "",
              command: "",
              label: "  "
            });
        }

    this.controller.setupWidget(Mojo.Menu.commandMenu, undefined,
      this.storyMenuModel);

    // Setup App Menu
    this.controller.setupWidget(Mojo.Menu.appMenu, News.MenuAttr, News.MenuModel);

    // Update story title in header and summary
    var storyViewTitleElement = this.controller.get("storyViewTitle");
    var storyViewSummaryElement = this.controller.get("storyViewSummary");
    storyViewTitleElement.innerHTML = this.storyFeed.stories[this.storyIndex].title;
    storyViewSummaryElement.innerHTML = this.storyFeed.stories[this.storyIndex].text;

};

// activate - display selected story
StoryViewAssistant.prototype.activate = function(event) {
    Mojo.Log.info("Story View Activated");

    // Update unreadStyle string and unReadCount in case it's changed
    if (this.storyFeed.stories[this.storyIndex].unreadStyle == News.unreadStory) {
        this.storyFeed.numUnRead--;
        this.storyFeed.stories[this.storyIndex].unreadStyle = "";
        News.feedListChanged = true;
    }

};

// ------------------------------------------------------------------------
// Handlers to go to next and previous stories, display web view
// or share via messaging or email.
StoryViewAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
```

```
            case "do-viewNext":
                this.stageController.swapScene(
                    {
                        transition: Mojo.Transition.crossFade,
                        name: "storyView"
                    },
                    this.storyFeed, this.storyIndex+1);
                break;
            case "do-viewPrevious":
                this.stageController.swapScene(
                    {
                        transition: Mojo.Transition.crossFade,
                        name: "storyView"
                    },
                    this.storyFeed, this.storyIndex-1);
                break;
            case "do-shareStory":
                var myEvent = event;
                var findPlace = myEvent.originalEvent.target;
                this.controller.popupSubmenu({
                    onChoose:  this.shareHandler,
                    placeNear: findPlace,
                    items: [
                        {label: $L("Email"), command: "do-emailStory"},
                        {label: $L("SMS/IM"), command: "do-messageStory"}
                        ]
                    });
                break;
             case "do-webStory":
                this.controller.serviceRequest(
                  "palm://com.palm.applicationManager", {
                      method: "open",
                      parameters: {
                          id: "com.palm.app.browser",
                          params: {
                            target: this.storyFeed.stories[this.storyIndex].url
                          }
                      }
                  });
                break;
          }
    }
};

// shareHandler - choose function for share submenu
StoryViewAssistant.prototype.shareHandler = function(command) {
        switch(command) {
            case "do-emailStory":
                this.controller.serviceRequest(
                  "palm://com.palm.applicationManager", {
                      method: "open",
                      parameters:  {
                          id: "com.palm.app.email",
                          params: {
                           summary: $L("Check out this News story..."),
```

```
                        text: this.storyFeed.stories[this.storyIndex].url
                    }
                }
            });
            break;
        case "do-messageStory":
            this.controller.serviceRequest(
              "palm://com.palm.applicationManager", {
                  method: "open",
                  parameters: {
                      id: "com.palm.app.messaging",
                      params: {
                          messageText: $L("Check this out: ")
                            +this.storyFeed.stories[this.storyIndex].url
                      }
                  }
            });
            break;
    }
};
```

news/app/models/cookies.js

```
/*  Cookie - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    Handler for cookieData, a stored version of News preferences.
    Will load or create cookieData, migrate preferences and update cookieData
    when called.

    Functions:
    initialize - loads or creates newsCookie; updates preferences withcontents
        of stored cookieData and migrates any preferences due version changes
    store - updates stored cookieData with current global preferences
*/

News.Cookie = ({

  initialize: function()  {
    // Update globals with preferences or create it.
    this.cookieData = new Mojo.Model.Cookie("comPalmAppNewsPrefs");
    var oldNewsPrefs = this.cookieData.get();
    if (oldNewsPrefs) {
      // If current version, just update globals & prefs
      if (oldNewsPrefs.newsVersionString == News.versionString)    {
        News.featureFeedEnable = oldNewsPrefs.featureFeedEnable;
        News.featureStoryInterval = oldNewsPrefs.featureStoryInterval;
        News.feedUpdateInterval = oldNewsPrefs.feedUpdateInterval;
        News.versionString = oldNewsPrefs.newsVersionString;
        News.notificationEnable = oldNewsPrefs.notificationEnable;
        News.feedUpdateBackgroundEnable = oldNewsPrefs.feedUpdateBackgroundEnable;
      } else {
        // migrate old preferences here on updates of News app
      }
```

```
    }

    this.storeCookie();

  },

  //  store - function to update stored cookie with global values
  storeCookie: function() {
    this.cookieData.put(    {
      featureFeedEnable: News.featureFeedEnable,
      feedUpdateInterval: News.feedUpdateInterval,
      featureStoryInterval: News.featureStoryInterval,
      newsVersionString: News.versionString,
      notificationEnable: News.notificationEnable,
      feedUpdateBackgroundEnable: News.feedUpdateBackgroundEnable
    });
  }

});
```

news/app/models/feeds.js

```
/*  Feeds - NEWS

    Copyright 2009 Palm, Inc.  All rights reserved.

    The primary data model for the News app. Feeds includes the primary
        data structure for the newsfeeds, which are structured as a list of lists:

    Feeds.list entry is:
      list[x].title           String    Title entered by user
      list[x].url             String    Feed source URL in unescaped form
      list[x].type            String    Feed type: either rdf, rss, or atom
      list[x].value           Boolean   Spinner model for feed update indicator
      list[x].numUnRead       Integer   How many stories are still unread
      list[x].newStoryCount   Integer   For each update, how many new stories
      list[x].stories         Array     Each entry is a complete story

    list.stories entry is:
       stories[y].title       String    Story title or headline
       stories[y].text        String    Story text
       stories[y].summary     String    Story text, stripped of markup
       stories[y].unreadStyle String    Null when Read
       stories[y].url         String    Story url

    Methods:
    initialize(test) - create default and test feed lists
    getDefaultList() - returns the default feed list as an array
    getTestList() - returns both the default and test feed lists as a single array
    loadFeedDb() - loads feed database depot, or creates default feed list
        if no existing depot
    processFeed(transport, index) - function to process incoming feeds that are
        XML encoded in an Ajax object and stores them in the Feeds.list. Supports
        RSS, RDF and Atom feed formats.
    storeFeedDb() - writes contents of Feeds.list array to feed database depot
```

```
    updateFeedList(index) - updates entire feed list starting with this.feedIndex.
*/

var Feeds = Class.create ({

    //  Default Feeds.list
    defaultList: [
            {
                title:"Huffington Post",
                url:"http://feeds.huffingtonpost.com/huffingtonpost/raw_feed",
                type:"atom", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Google",
                url:"http://news.google.com/?output=atom",
                type:"atom", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"BBC News",
                url:"http://newsrss.bbc.co.uk/rss/newsonline_world_edition/
                  front_page/rss.xml",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"New York Times",
                url:"http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"MSNBC",
                url:"http://rss.msnbc.msn.com/id/3032091/device/rss/rss.xml",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"National Public Radio",
                url:"http://www.npr.org/rss/rss.php?id=1004",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Slashdot",
                url:"http://rss.slashdot.org/Slashdot/slashdot",
                type:"rdf", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Engadget",
                url:"http://www.engadget.com/rss.xml",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"The Daily Dish",
                url:"http://feeds.feedburner.com/andrewsullivan/rApM?format=xml",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Guardian UK",
                url:"http://feeds.guardian.co.uk/theguardian/rss",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"Yahoo Sports",
                url:"http://sports.yahoo.com/top/rss.xml",
                type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
            },{
                title:"ESPN",
                url:"http://sports-ak.espn.go.com/espn/rss/news",
```

```
        type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
    },{
        title:"Ars Technica",
        url:"http://feeds.arstechnica.com/arstechnica/index?format=xml",
        type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
    },{
        title:"Nick Carr",
        url:"http://feeds.feedburner.com/roughtype/unGc",
        type:"rss", value:false, numUnRead:0, newStoryCount:0, stories:[]
    }
],

// Additional test feeds
testList: [
    {
        title:"Hacker News",
        url:"http://news.ycombinator.com/rss",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
        title:"Ken Rosenthal",
        url:"http://feeds.feedburner.com/foxsports/rss/rosenthal",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
        title:"George Packer",
        url:"http://www.newyorker.com/online/blogs/georgepacker/rss.xml",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
        title:"Palm Open Source",
        url:"http://www.palmopensource.com/tmp/news.rdf",
        type:"rdf", value:false, numUnRead:0, stories:[]
    },{
        title:"Baseball Prospectus",
        url:"http://www.baseballprospectus.com/rss/feed.xml",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
        title:"The Page",
        url:"http://feedproxy.google.com/time/thepage?format=xml",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
        title:"Salon",
        url:"http://feeds.salon.com/salon/index",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
        title:"Slate",
        url:"http://feedproxy.google.com/slate?format=xml",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
     title:"SoSH",
        url:"http://sonsofsamhorn.net/index.php?act=rssout&id=1",
        type:"rss", value:false, numUnRead:0, stories:[]
    },{
        title:"Talking Points Memo",
        url:"http://feeds.feedburner.com/talking-points-memo",
        type:"atom", value:false, numUnRead:0, stories:[]
    },{
```

```
            title:"Whatever",
            url:"http://scalzi.com/whatever/?feed=rss2",
            type:"rss", value:false, numUnRead:0, stories:[]
        },{
            title:"Baseball America",
            url:"http://www.baseballamerica.com/today/rss/rss.xml",
            type:"rss", value:false, numUnRead:0, stories:[]
        },{
            title:"Test RDF Feed",
            url:"http://foobar.blogalia.com/rdf.xml",
            type:"rdf", value:false, numUnRead:0, stories:[]
        },{
            title:"Daily Kos",
            url:"http://feeds.dailykos.com/dailykos/index.html",
            type:"rss", value:false, numUnRead:0, stories:[]
        }
    ],
// initialize - Assign default data to the feedlist
initialize: function(test)  {
    this.feedIndex = 0;
    if (!test)  {
        this.list = this.getDefaultList();
    } else {
        this.list = this.getTestList();
    }
},

// getDefaultList - returns the default feed list as an array
getDefaultList: function() {
    var returnList = [];
    for (var i=0; i<this.defaultList.length; i++)   {
        returnList[i] = this.defaultList[i];
    }

    return returnList;
},

// getTestList - returns the default and tests feeds in one array
getTestList: function() {
    var returnList = [];
    var defaultLength = this.defaultList.length;
    for (var i=0; i<defaultLength; i++)   {
        returnList[i] = this.defaultList[i];
    }

    for (var j=0; j<this.testList.length; j++)   {
        returnList[j+defaultLength] = this.testList[j];
    }

    return returnList;
},

// loadFeedDb - loads feed db depot, or creates it with default list
// if it doesn't already exist
loadFeedDb: function()  {
```

```
  // Open the database to get the most recent feed list
  // DEBUG - replace is true to recreate db every time; false for release
    this.db = new Mojo.Depot(
        {name:"feedDB", version:1, estimatedSize: 100000, replace: false},
        this.loadFeedDbOpenOk.bind(this),
        function(result) {
            Mojo.Log.warn("Can't open feed database: ", result);
        }
    );
},

// dbOpenOK - Callback for successful db request in setup. Get stored db or
// fallback to using default list
loadFeedDbOpenOk: function() {
    Mojo.Log.info("Database opened OK");
    this.db.simpleGet("feedList", this.loadFeedDbGetSuccess.bind(this),
      this.loadFeedDbUseDefault.bind(this));
},

// loadFeedDbGetSuccess - successful retrieval of db. Call
//  useDefaultList if the feedlist empty or null or initiate an update
//  to the list by calling updateFeedList.
loadFeedDbGetSuccess: function(fl) {

    Mojo.Log.info("Database Retrieved OK");
    if (fl === null) {
        Mojo.Log.warn("Retrieved empty or null list from DB");
        this.loadFeedDbUseDefault();

    } else {
        Mojo.Log.info("Retrieved feedlist from DB");
        this.list = fl;

        // If update, then convert from older versions

        this.updateFeedList();
    }
},

// loadFeedDbUseDefault() - Callback for failed DB retrieval meaning no list
loadFeedDbUseDefault: function() {
  // Couldn't get the list of feeds. Maybe its never been set up, so
  // initialize it here to the default list and then initiate an update
  // with this feed list
    Mojo.Log.warn("Database has no feed list. Will use default.");
    this.list = this.getDefaultList();
    this.updateFeedList();
},

// processFeed (transport, index) - process incoming feeds that
//   are XML encoded in an Ajax object and stores them in Feeds.list.
//   Supports RSS, RDF and Atom feed formats.
processFeed: function(transport, index) {
  // Used to hold feed list as it's processed from the Ajax request
```

```
  var listItems = [];
// Variable to hold feed type tags
  var feedType = transport.responseXML.getElementsByTagName("rss");

  if (index === undefined)    {
      //    Default index is at end of the list
      index = this.list.length-1;
  }

  // Determine whether RSS 2, RDF (RSS 1) or ATOM feed
  if (feedType.length > 0)    {
      this.list[index].type = "rss";
  }
  else    {
      feedType = transport.responseXML.getElementsByTagName("RDF");
      if (feedType.length > 0)    {
          this.list[index].type = "RDF";
      }
      else {
          feedType = transport.responseXML.getElementsByTagName("feed");
          if (feedType.length > 0)    {
              this.list[index].type = "atom";
          }
          else {

          // If none of those then it can't be processed, set an error code
          // in the result, log the error and return
          Mojo.Log.warn("Unsupported feed format in feed ",
              this.list[index].url);
          return News.invalidFeedError;
          }
      }
  }

  // Process feeds; retain title, text content and url
  switch(this.list[index].type) {
  case "atom":
      // Temp object to hold incoming XML object
      var atomEntries = transport.responseXML.getElementsByTagName("entry");
      for (var i=0; i<atomEntries.length; i++) {
          listItems[i] = {
              title: unescape(atomEntries[i].getElementsByTagName("title").
                item(0).textContent),
              text: atomEntries[i].getElementsByTagName("content").
                item(0).textContent,
              unreadStyle: News.unreadStory,
              url: atomEntries[i].getElementsByTagName("link").
                item(0).getAttribute("href")
          };

      // Strip HTML from text for summary and shorten to 100 characters
      listItems[i].summary = listItems[i].text.replace(/(<([^>]+)>)/ig,"");
      listItems[i].summary = listItems[i].summary.replace(/http:\S+/ig,"");
      listItems[i].summary = listItems[i].summary.replace(/#[a-z]+/ig,"{");
      listItems[i].summary =
```

```
      listItems[i].summary.replace(/(\{([^\}]+)\})/ig,"");
    listItems[i].summary =
      listItems[i].summary.replace(/digg_url .../,"");
    listItems[i].summary = unescape(listItems[i].summary);
    listItems[i].summary = listItems[i].summary.substring(0,101);
    }
    break;

case "rss":
    // Temp object to hold incoming XML object
    var rssItems = transport.responseXML.getElementsByTagName("item");
    for (i=0; i<rssItems.length; i++) {

        listItems[i] = {
            title: unescape(rssItems[i].getElementsByTagName("title").
              item(0).textContent),
            text: rssItems[i].getElementsByTagName("description").
              item(0).textContent,
            unreadStyle: News.unreadStory,
            url: rssItems[i].getElementsByTagName("link").
              item(0).textContent
        };

    // Strip HTML from text for summary and shorten to 100 characters
    listItems[i].summary = listItems[i].text.replace(/(<([^>]+)>)/ig,"");
    listItems[i].summary = listItems[i].summary.replace(/http:\S+/ig,"");
    listItems[i].summary = listItems[i].summary.replace(/#[a-z]+/ig,"{");
    listItems[i].summary =
      listItems[i].summary.replace(/(\{([^\}]+)\})/ig,"");
    listItems[i].summary =
      listItems[i].summary.replace(/digg_url .../,"");
    listItems[i].summary = unescape(listItems[i].summary);
    listItems[i].summary = listItems[i].summary.substring(0,101);
    }
    break;

case "RDF":
    // Temp object to hold incoming XML object
    var rdfItems = transport.responseXML.getElementsByTagName("item");
    for (i=0; i<rdfItems.length; i++) {

        listItems[i] = {
            title: unescape(rdfItems[i].getElementsByTagName("title").
              item(0).textContent),
            text: rdfItems[i].getElementsByTagName("description").
              item(0).textContent,
            unreadStyle: News.unreadStory,
            url: rdfItems[i].getElementsByTagName("link").
              item(0).textContent
        };

    // Strip HTML from text for summary and shorten to 100 characters
    listItems[i].summary = listItems[i].text.replace(/(<([^>]+)>)/ig,"");
    listItems[i].summary = listItems[i].summary.replace(/http:\S+/ig,"");
    listItems[i].summary = listItems[i].summary.replace(/#[a-z]+/ig,"{");
```

```
            listItems[i].summary =
              listItems[i].summary.replace(/(\{([^\}]+)\})/ig,"");
            listItems[i].summary =
              listItems[i].summary.replace(/digg_url .../,"");
            listItems[i].summary = unescape(listItems[i].summary);
            listItems[i].summary = listItems[i].summary.substring(0,101);
            }
            break;
        }

        // Update read items by comparing new stories with stories last
        // in the feed. For all old stories, use the old unreadStyle value,
        // otherwise set unreadStyle to News.unreadStory.
        // Count number of unread stories and store value.
        // Determine if any new stories when URLs don't match a previously
        // downloaded story.
        //
        var numUnRead = 0;
        var newStoryCount = 0;
        var newStory = true;
        for (i = 0; i < listItems.length; i++) {
            var unreadStyle = News.unreadStory;
            var j;
            for (j=0; j<this.list[index].stories.length; j++ ) {
                if(listItems[i].url == this.list[index].stories[j].url) {
                    unreadStyle = this.list[index].stories[j].unreadStyle;
                    newStory = false;
                }
            }

            if(unreadStyle == News.unreadStory) {
                numUnRead++;
            }

            if (newStory) {
                newStoryCount++;
            }

            listItems[i].unreadStyle = unreadStyle;
        }

        // Save updated feed in global feedlist
        this.list[index].stories = listItems;
        this.list[index].numUnRead  = numUnRead;
        this.list[index].newStoryCount = newStoryCount;

        // If new feed, the user may not have entered a name; if so, set the
        // name to the feed title
        if (this.list[index].title === "")    {
            //    Will return multiple hits, but the first is the feed name
            var titleNodes = transport.responseXML.getElementsByTagName("title");
            this.list[index].title = titleNodes[0].textContent;
        }
        return News.errorNone;
    },
```

```
// storeFeedDb() - writes contents of Feeds.list array to feed database depot
storeFeedDb: function() {
    Mojo.Log.info("FeedList save started");
    this.db.simpleAdd("feedList", this.list,
            function() {Mojo.Log.info("FeedList saved OK");},
            this.storeFeedDBFailure);
},


// storeFeedDbFailure(transaction, result) - handles save failure, usually an
//    out of memory error
storeFeedDbFailure: function(result) {
    Mojo.Log.warn("Database save error: ", result);
},

// updateFeedList(index) - called to cycle through feeds. This is called
//   once per update cycle.
updateFeedList: function(index) {
    News.feedListUpdateInProgress = true;

    // request fresh copies of all stories
    this.currentFeed = this.list[this.feedIndex];
    this.updateFeedRequest(this.currentFeed);
},

// feedRequest - function called to setup and make a feed request
updateFeedRequest: function(currentFeed) {
    Mojo.Log.info("URL Request: ", currentFeed.url);

    // Notify the chain that there is an update in progress
    Mojo.Controller.getAppController().sendToNotificationChain({
      type: "update", update: true, feedIndex: this.feedIndex});

    var request = new Ajax.Request(currentFeed.url, {
        method: "get",
        evalJSON: "false",
        onSuccess: this.updateFeedSuccess.bind(this),
        onFailure: this.updateFeedFailure.bind(this)
    });
},

// updateFeedFailure - Callback routine from a failed AJAX feed request;
//   post a simple failure error message with the http status code.
updateFeedFailure: function(transport) {
    // Prototype template to generate a string from the return status.
    var t = new Template(
      $L("Status #{status} returned from newsfeed request."));
    var m = t.evaluate(transport);

    //    Post error alert and log error
    Mojo.Log.info("Invalid feed - http failure, check feed: ", m);

    // Notify the chain that this update is complete
    Mojo.Controller.getAppController().sendToNotificationChain({
```

```
      type: "update", update: false, feedIndex: this.feedIndex});
},

// updateFeedSuccess - Successful AJAX feed request (feedRequest);
//   uses this.feedIndex and this.list
updateFeedSuccess: function(transport) {

    var t = new Template($L({key: "newsfeed.status",
      value: "Status #{status} returned from newsfeed request."}));
    Mojo.Log.info("Feed Request Success: ", t.evaluate(transport));

    // Work around due to occasional XML errors
    if (transport.responseXML === null && transport.responseText !== null) {
        Mojo.Log.info("Request not in XML format - manually converting");
        transport.responseXML =
          new DOMParser().parseFromString(transport.responseText, "text/xml");
     }

    // Process the feed, passing in transport holding the updated feed data
    var feedError = this.processFeed(transport, this.feedIndex);

    // If successful processFeed returns News.errorNone
    if (feedError == News.errorNone)    {
        var appController = Mojo.Controller.getAppController();
        var stageController =
          appController.getStageController(News.MainStageName);
        var dashboardStageController =
          appController.getStageProxy(News.DashboardStageName);

        // Post a notification if new stories and application is minimized
        if (this.list[this.feedIndex].newStoryCount > 0)   {
            Mojo.Log.info("New Stories: ",
              this.list[this.feedIndex].title," : ",
              this.list[this.feedIndex].newStoryCount, " New Items");
            if (!stageController.isActiveAndHasScenes() &&
              News.notificationEnable) {
                var bannerParams = {
                    messageText: Mojo.Format.formatChoice(
                        this.list[this.feedIndex].newStoryCount,
                        $L("0##{title} : No New Items|
                            1##{title} : 1 New Item|
                            1>##{title} : #{count} New Items"),
                        {
                          title: this.list[this.feedIndex].title,
                          count: this.list[this.feedIndex].newStoryCount
                        }
                    )
                };

                appController.showBanner(bannerParams, {
                    action: "notification",
                    index: this.feedIndex
                  },
                  this.list[this.feedIndex].url);
```

```
            // Create or update dashboard
            var feedlist = this.list;
            var selectedFeedIndex = this.feedIndex;

            if(!dashboardStageController) {
                Mojo.Log.info("New Dashboard Stage");
                var pushDashboard = function(stageController){
                    stageController.pushScene("dashboard", feedlist,
                      selectedFeedIndex);
                };
                appController.createStageWithCallback({
                      name: News.DashboardStageName,
                      lightweight: true
                    },
                    pushDashboard, "dashboard");
            }
            else {
                Mojo.Log.info("Existing Dashboard Stage");
                dashboardStageController.delegateToSceneAssistant(
                  "updateDashboard", selectedFeedIndex);
            }
        }
    }
} else     {

    // There was a feed process error; unlikely, but could happen if the
    // feed was changed by the feed service. Log the error.
    if (feedError == News.invalidFeedError)    {
        Mojo.Log.info("Feed ", this.nameModel.value,
          " is not a supported feed type.");
    }
}

// Notify the chain that this update is done
Mojo.Controller.getAppController().sendToNotificationChain({
  type: "update", update: false, feedIndex: this.feedIndex});
News.feedListChanged = true;

// If NOT the last feed then update the feedsource and request next feed
this.feedIndex++;
if(this.feedIndex < this.list.length) {
    this.currentFeed = this.list[this.feedIndex];

    // Notify the chain that there is a new update in progress
    Mojo.Controller.getAppController().sendToNotificationChain({
        type: "update",
      update: true,
      feedIndex: this.feedIndex
    });

    // Request an update for the next feed
    this.updateFeedRequest(this.currentFeed);
} else {

    // Otherwise, this update is done. Reset index to 0 for next update
```

```
            this.feedIndex = 0;
            News.feedListUpdateInProgress = false;

        }
    }
});
```

news/app/assistants/views/dashboard/dashboard-scene.html

```
<div id="dashboardinfo" class="dashboardinfo"></div>
```

news/app/assistants/views/dashboard/item-info.html

```
<div class="dashboard-notification-module">
    <div class="palm-dashboard-icon-container">
        <div class="dashboard-newitem">
            <span>#{count}</span>
        </div>
         <div id="dashboard-icon" class="palm-dashboard-icon dashboard-icon-news">
         </div>
    </div>
    <div class="palm-dashboard-text-container">
         <div class="dashboard-title">
            #{title}
         </div>
        <div id="dashboard-text" class="palm-dashboard-text">#{message}</div>
    </div>
</div
```

news/app/assistants/views/feedList/addFeed-dialog.html

```
<div id="palm-dialog-content" class="palm-dialog-content">
    <div id="add-feed-title" class="palm-dialog-title">
        Add Feed
    </div>
    <div class="palm-dialog-separator"></div>
    <div class="textfield-group" x-mojo-focus-highlight="true">
        <div class="title">
           <div x-mojo-element="TextField" id="newFeedURL"></div>
        </div>
    </div>
    <div class="textfield-group" x-mojo-focus-highlight="true">
        <div class="title">
           <div x-mojo-element="TextField" id="newFeedName"></div>
        </div>
    </div>

    <div class="palm-dialog-buttons">
        <div x-mojo-element="Button" id="okButton">
        <div x-mojo-element="Button" id="cancelButton">
    </div>
</div>
```

news/app/assistants/views/feedList/feedList-scene.html

```
<div id="feedListScene">

    <!--    Search Field                                        -->
    <div id="searchFieldContainer">
        <div x-mojo-element="FilterList" id="startSearchField"></div>
    </div>

    <div id="feedListMain">

        <!--    Rotating Feature Story                          -->
        <div id="feedList_view_header" class="palm-header left">
            Latest News
            <div id="featureDrawer" class="featureFeed-close"></div>
        </div>
        <div class="palm-header-spacer"></div>
        <div x-mojo-element="Drawer" id="featureFeedDrawer">
            <div x-mojo-element="Scroller" id="featureScroller" >
                <div id="featureStoryDiv" class="featureScroller">
                    <div id="splashScreen" class="splashScreen">
                        <div class="update-image"></div>
                        <div class="title">News v0.8<span>#{version}</span>
                            <div class="palm-body-text">Copyright 2009, Palm®</div>
                        </div>
                    </div>
                    <div id="featureStoryTitle" class="palm-body-title">
                    </div>
                    <div id="featureStory" class="palm-body-text">
                    </div>
                </div>
            </div>
        </div>

        <!--    Feed List                                        -->
        <div class="palm-list">
            <div x-mojo-element="List" id="feedListWgt"></div>
        </div>
    </div>
</div>
```

news/app/assistants/views/feedList/feedListTemplate.html

```
<div class="palm-list">#{-listElements}</div>
```

news/app/assistants/views/feedList/feedRowTemplate.html

```
<div class="palm-row" x-mojo-touch-feedback="delayed">
  <div class="palm-row-wrapper textfield-group">
    <div class="title">

      <div class="palm-dashboard-icon-container feedlist-icon-container">
        <div class="dashboard-newitem feedlist-newitem">
          <span class="unReadCount">#{numUnRead}</span>
```

```
        </div>
        <div id="dashboard-icon" class="palm-dashboard-icon feedlist-icon">
        </div>
      </div>

      <div class="feedlist-info icon right" id="info"></div>
      <div x-mojo-element="Spinner" class="right" name="feedSpinner"</div>
      <div class="feedlist-title truncating-text">#{title}</div>
      <div class="feedlist-url truncating-text">#{-url}</div>

    </div>
  </div>
</div>
```

news/app/assistants/views/preferences/preferences-scene.html

```
<div class="palm-page-header">
    <div class="palm-page-header-wrapper">
        <div class="icon news-mini-icon"></div>
        <div class="title">News Preferences</div>
    </div>
</div>

<div class="palm-group">
    <div class="palm-group-title"><span>Feature Feed</span></div>
        <div class="palm-list">
            <div x-mojo-element="IntegerPicker" id="featureFeedDelay"></div>
        </div>
    </div>
</div>

<div class="palm-group">
    <div class="palm-group-title"><span>Feed Updates</span></div>
           <div class="palm-list">
            <div class="palm-row first">
                <div class="palm-row-wrapper">
                    <div x-mojo-element="ListSelector" id="feedCheckIntervalList">
                    </div>
                </div>
            </div>
            <div class="palm-row">
                <div class="palm-row-wrapper">
                       <div x-mojo-element="ToggleButton" id="notificationToggle">
                       </div>
                       <div class="title left">Show Notification</div>
                </div>
            </div>
            <div class="palm-row last">
                <div class="palm-row-wrapper">
                       <div x-mojo-element="ToggleButton" id="bgUpdateToggle">
                       </div>
                       <div class="title left">Wake Device</div>
                </div>
            </div>
        </div>
```

```
    </div>
</div>
```

news/app/assistants/views/storyList/storyList-scene.html

```
<div class="palm-header-spacer"></div>
<div id="storyListScene" class="storyListScene">
    <div x-mojo-element="List" id="storyListWgt" ></div>
</div>
<div class="storyList-filter">
    <div x-mojo-element="FilterList" id="storyListSearch" class="palm-list"></div>
</div>
```

news/app/assistants/views/storyList/storyListTemplate.html

```
<div class="palm-list">#{-listElements}</div>
```

news/app/assistants/views/storyList/storyRowTemplate.html

```
<div class="palm-row" x-mojo-touch-feedback="delayed">
    <div class="palm-row-wrapper">
        <div id="storyTitle" class="title truncating-text #{unreadStyle}">
            #{title}
        </div>
        <div id="storySummary" class="news-subtitle truncating-text">
            #{summary}
        </div>
    </div>
</div>
```

news/app/assistants/views/storyView/storyView-scene.html

```
<div id="storyViewScene">
    <div class="palm-page-header multi-line">
        <div class="palm-page-header-wrapper">
            <div id="storyViewTitle" class="title left">
            </div>
        </div>
    </div>
    <div class="palm-text-wrapper">
        <div id="storyViewSummary" class="palm-body-text">
        </div>
    </div>
</div>
```

news/appinfo.json

```
{
    "title": "News",
    "type": "web",
    "main": "index.html",
    "id": "com.palm.app.news11-1",
```

```
    "version": "1.0.0",
    "vendor": "Palm",
    "noWindow" : "true",
    "icon": "icon.png",
    "theme": "light"
}
```

news/framework_config.json

```
{
    "logLevel": "0",
    "timingEnabled": "true"
}
```

news/index.html

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <title>News</title>
  <script src="/usr/palm/frameworks/mojo/mojo.js" type="text/javascript"
      x-mojo-version="1"></script>
  <link href="stylesheets/News.css" media="screen" rel="stylesheet"
      type="text/css"/>
</head>

<body>
</body>

</html>
```

news/resources/es_us/appinfo.json

```
{
    "title": "Noticias",
    "type": "web",
    "main": "../../index.html",
    "id": "com.palm.app.news",
    "version": "1.0.0",
    "vendor": "Palm",
    "noWindow" : "true",
    "icon": "../../icon.png",
    "theme": "light"
}
```

news/resources/es_us/strings.json

```
{
   "#{status}" : "#{status}",
```

```
"0##{title} : No New Items|1##{title} : 1 New Item|1>##{title} :
  #{count} New Items" :    "0##{title} : No hay elementos nuevos|1##{title} :
  1 elemento nuevo|1>##{title} : #{count} elementos nuevos",
"1 Day" : "1 día",
"1 Hour" : "1 hora",
"15 Minutes" : "15 minutos",
"4 Hours" : "4 horas",
"5 Minutes" : "5 minutos",
"About News..." : "Acerca de noticias...",
"Add Feed DB save error : #{message}; can't save feed list." :
  "Error de base de datos al intentar agregar nueva fuente web : #{message};
  no se puede guardar la lista de fuentes web.",
"Add News Feed Source" : "Añadir fuente web de noticias",
"Add..." : "Añadir...",
"Adding a Feed" : "Añadiendo una fuente web",
"All Read" : "Todas leídas",
"All Unread" : "Todas las no leídas",
"Can't open feed database: " :
  "No se puede abrir la base de datos de fuentes web: ",
"Cancel" : "Cancelar",
"Cancel search" : "Cancelar búsqueda",
"Check out this News story..." : "Leer esta noticia...",
"Check this out: " : "Mira esto: ",
"Copyright 2009, Palm Inc." : "Copyright 2009, Palm Inc.",
"Database save error: " : "Error al guardar en la base de datos: ",
"Edit a Feed" : "Editar una fuente web",
"Edit Feed" : "Editar fuente web",
"Edit News Feed" : "Editar una fuente web de noticias",
"Feature Feed" : "Fuente web destacada",
"Featured Feed" : "Fuente web destacada",
"Feature Rotation" : "Rotación de fuente web destacada",
"Feed Request Success:" : "Solicitud de fuente web lograda:",
"Feed Updates" : "Actualización de fuentes web",
"Help..." : "Ayuda...",
"Interval" : "Intervalo",
"Invalid Feed - not a supported feed type" :
  "Fuente web no válida: no es un tipo de fuente web admitido",
"Latest News" : "Últimas noticias",
"Manual Updates" : "Actualizaciones manuales",
"Mark Read or Unread" : "Marcar leída o no leída",
"New Card" : "Tarjeta nueva",
"New features" : "Nuevas características",
"New Items" : "Elementos nuevos",
"News Help" : "Ayuda para noticias",
"News Preferences" : "Preferencias para noticias",
"newsfeed.status" :
  "Estado #{status} devuelto desde solicitud de fuente web de noticias",
"OK" : "OK",
"Optional" : "Opcional",
"Preferences..." : "Preferencias...",
"Reload" : "Cargar nuevamente",
"Rotate Every" : "Girar cada",
"Rotation (in seconds)" : "Rotación (en segundos)",
"RSS or ATOM feed URL" : "Fuente web RSS o ATOM URL",
"Search for:  #{filter}" : "Buscar:  #{filter}",
```

```
    "Show Notification" :"Mostrar aviso",
    "SMS/IM" : "SMS/IM",
    "Status #{status} returned from newsfeed request." :
      "La solicitud de fuente web de noticias indicó el estado #{status}.",
    "Stop" : "Detener",
    "Title" : "Título",
    "Title (Optional)" : "Título (Opcional)",
    "Update All Feeds" : "Actualizar todas las fuentes web",
    "Wake Device" : "Activar dispositivo",
    "Will need to reload on next use." :
      "Se tendrá que cargar de nuevo la próxima vez que se use."

}
```

news/resources/es_us/views/feedList/addFeed-dialog.html

```
<div id="palm-dialog-content" class="palm-dialog-content">
    <div id="add-feed-title" class="palm-dialog-title">
        Añadir fuente web
    </div>
    <div class="palm-dialog-separator"></div>
    <div class="textfield-group" x-mojo-focus-highlight="true">
        <div class="title">
           <div x-mojo-element="TextField" id="newFeedURL"></div>
        </div>
    </div>
    <div class="textfield-group" x-mojo-focus-highlight="true">
        <div class="title">
           <div x-mojo-element="TextField" id="newFeedName"></div>
        </div>
    </div>

    <div class="palm-dialog-buttons">
        <div x-mojo-element="Button" id="okButton">
        <div x-mojo-element="Button" id="cancelButton">
    </div>
</div>
```

news/resources/es_us/views/feedList/feedList-scene.html

```
<div id="feedListScene">

    <!--    Search Field                                          -->
    <div id="searchFieldContainer">
        <div x-mojo-element="FilterList" id="startSearchField"></div>
    </div>

    <div id="feedListMain">

        <!--    Rotating Feature Story                            -->
        <div id="feedList_view_header" class="palm-header left">
            Últimas noticias
            <div id="featureDrawer" class="featureFeed-close"></div>
        </div>
```

```
        <div class="palm-header-spacer"></div>
        <div x-mojo-element="Drawer" id="featureFeedDrawer">
            <div x-mojo-element="Scroller" id="featureScroller" >
                <div id="featureStoryDiv" class="featureScroller">
                    <div id="splashScreen" class="splashScreen">
                        <div class="splashImage"></div>
                        <div class="splashText">
                        Noticias v0.8<span>#{version}</span>
                            <div class="splashBody">Copyright 2009, Palm®</div>
                        </div>
                    </div>
                    <div id="featureStoryTitle" class="palm-body-title">
                    </div>
                   <div id="featureStory" class="palm-body-text">
                </div>
              </div>
           </div>
        </div>

        <!--    Feed List                                           -->
        <div class="palm-list">
            <div x-mojo-element="List" id="feedListWgt"></div>
        </div>
    </div>
</div>
```

news/resources/es_us/views/preferences/preferences-scene.html

```
<div class="palm-page-header">
    <div class="palm-page-header-wrapper">
        <div class="icon news-mini-icon"></div>
        <div class="title">Preferencias para noticias</div>
    </div>
</div>

<div class="palm-group">
    <div class="palm-group-title"><span>Fuente web destacada</span></div>
        <div class="palm-list">
            <div x-mojo-element="IntegerPicker" id="featureFeedDelay"></div>
        </div>
    </div>
</div>

<div class="palm-group">
    <div class="palm-group-title"><span>Actualización de fuentes web</span></div>
           <div class="palm-list">
            <div class="palm-row first">
                <div class="palm-row-wrapper">
                    <div x-mojo-element="ListSelector" id="feedCheckIntervalList">
                    </div>
                </div>
            </div>
            <div class="palm-row">
                <div class="palm-row-wrapper">
                     <div x-mojo-element="ToggleButton" id="notificationToggle">
```

```
                    </div>
                    <div class="title left">Mostrar aviso</div>
                </div>
            </div>
            <div class="palm-row last">
                <div class="palm-row-wrapper">
                    <div x-mojo-element="ToggleButton" id="bgUpdateToggle"></div>
                   <div class="title left">Activar dispositivo</div>
                </div>
            </div>
        </div>
    </div>
</div>
```

news/sources.json

```
[
  {
    "source": "app/assistants/app-assistant.js"
  },
  {
    "source": "app/assistants/stage-assistant.js"
  },
  {
    "source": "app/assistants/dashboard-assistant.js",
    "scenes": "dashboard"
  },
  {
    "source": "app/assistants/feedList-assistant.js",
    "scenes": "feedList"
  },
  {
    "source": "app/assistants/preferences-assistant.js",
    "scenes": "preferences"
  },
  {
    "source": "app/assistants/storyList-assistant.js",
    "scenes": "storyList"
  },
  {
    "source": "app/assistants/storyView-assistant.js",
    "scenes": "storyView"
  },
  {
    "source" : "app/models/cookies.js"
  },
  {
    "source" : "app/models/feeds.js"
  }
]
```

news/stylesheets/News.css

```
/*    News CSS

    Copyright 2009 Palm, Inc.  All rights reserved.

    App overrides of palm scene and widget styles.
*/

/* Contrains storyView content to width of scene */
img    {
    max-width:280px;
}


/* Header Styles  */
.icon.news-mini-icon {
    background: url(../images/header-icon-news.png) no-repeat;
    margin-top: 13px;
    margin-left: 17px;
}


/* FeedList Header styles for feature drawer and selection */
.featureFeed-close {
    float:right;
    margin: 8px -12px 0px 0px;
    height:35px;
    width: 35px;
    background: url(../images/details-open-arrow.png) no-repeat;
}

.featureFeed-open {
    float:right;
    margin: 8px -12px 0px 0px;
    height:35px;
    width: 35px;
    background: url(../images/details-closed-arrow.png) no-repeat;
}.palm-drawer-container {
    border-width: 20px 1px 20px 1px;
    -webkit-border-image:
      url(../images/palm-drawer-background-1.png) 20 1 20 1 repeat repeat;
    -webkit-box-sizing: border-box;
    overflow: visible;
}


/* Feature Feed styles */
.featureScroller {
    height: 100px;
    width: 280px;
    margin-left: 20px;
}
```

```
/* feedList styles */
.palm-row-wrapper.textfield-group {
    margin-top: 5px;
}

.feedlist-title {
    line-height: 2.0em;
}

.feedlist-url {
    font-size: 14px;
    color: gray;
    margin-top: -20px;
    margin-bottom: -20px;
    line-height: 16px;
}

.feedlist-info {
    background: url(../images/info-icon.png) center center no-repeat;
}

.feedlist-icon-container {
    height: 54px;
    margin-top: 5px;
}

.feedlist-icon {
    background: url(../images/list-icon-rssfeed.png) center no-repeat;
}

.feedlist-newitem   {
    line-height: 20px;
    height: 26px;
    min-width: 26px;
    -webkit-border-image:
      url(../images/feedlist-newitem.png) 4 10 4 10 stretch stretch;
    -webkit-box-sizing: border-box;
    border-width: 4px 10px 4px 10px;
}

.unReadCount {
    color: white;
}


/* Story List styles */
.news-subtitle {
        padding: 0px 14px 0px 14px;
        font-size: 14px;
        margin-top: -10px;
        line-height: 16px;
}

.palm-row-wrapper > .unReadStyle    {
    font-weight: bold;
```

```
}
.storyList-filter .filter-field-container {
    top: 48px;
    left: 0px;
    position: fixed;
    width: 100%;
    height: 48px;
    border-width: 26px 23px 20px 23px;
    -webkit-border-image:
      url(../images/filter-search-light-bg.png) 26 23 20 23 repeat repeat;
    -webkit-box-sizing: border-box;
    z-index: 11002;
}

/* Splash Screen image  */
.update-image    {
    background: url(../images/news-icon.png) center center no-repeat;
    float: left;
    height: 58px;
    width: 58px;
    margin-left: -3px;
}

/* dashboard styles */

.dashboard-icon-news {
    background: url(../images/dashboard-icon-news.png);
}
```

Index

Symbols

A

We’d like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the Author

Mitch Allen is CTO of software at Palm, Inc., where he has worked in various positions for nine years, starting with building and leading the software team at Handspring, which conceived and developed the Treō smartphone. Mitch contributed to the early architecture of the webOS platform and led the development team through the initial design stage, and as a result is intimately familiar with the capabilities of the platform and tools. Previously, Mitch spent 15 years developing image and text-processing systems at Kodak and Agfa Compugraphic; after this time, he worked at Apple. He holds a degree in math and computer science from the University of New Hampshire.

He is currently part of the team developing the webOS SDK and tools, and is working with initial webOS developers.

Colophon

The animal on the cover of *Palm webOS* is a luna moth (*Actias luna*). Luna moths usually live in North American regions filled with black cherry, maple, hickory, willow, and other trees with leaves that can feed their young.

Upon hatching, luna moth caterpillars will wander aimlessly along the plants they were born upon and befriend other recently born caterpillars. But after passing through subsequent stages of larval development, the caterpillars' gregarious temperaments change, and they become loners as they prepare for pupation.

Before spinning and entering their thin cocoons, luna moth caterpillars will expel excess water and other fluids from their bodies. Once cocooned, the caterpillars will pupate for approximately two weeks, after which they will emerge in daylight with wet, crumpled wings. Although their wings take only 20 minutes to dry, luna moths will wait until nighttime to fly, as they have also metamorphosed into entirely nocturnal creatures.

While the caterpillars will munch on the leaves of the plants they were born upon, luna moths begin and end their adulthoods mouthless. But this trait does not disable them: they have no need for food, as they have also lost their digestive tracts. Though other insects will forage for food shortly after birth, luna moths exist only to find a mate and produce another generation.

Female luna moths attract mates by releasing pheromones from their abdomens; males detect these pheromones via their hairy antennae (and, because males and females both possess lime-green wings, a close inspection of a moth's antennae is an easy way to determine gender, as the male's antennae are hairier than the female's). Luna moths typically mate after midnight, and females will lay 100 to 300 eggs on the undersides of leaves just hours later, in the evening. The insect's short lifespan necessitates an accelerated reproduction schedule—adult luna moths live no longer than a week.

Luna moths have inspired many: Luna Moth is the name of a character in Michael Chabon's novel *The Amazing Adventures of Kavalier and Clay* (Picador), and Vladimir Nabokov, who was also an accomplished lepidopterist, has described the insect admiringly in his writings. Crafters have also paid homage to the insect's vivid wings with products ranging from shawls to stained glass. Luna was also Palm, Inc.'s code name for the webOS application environment, including the Mojo framework.

The cover image is from *Dover's Animals*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.