

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



Palm[®] webOS[™]

O'REILLY[®]

Mitch Allen

Table of Contents

Copyright	1
Introduction	1
Overview of webOS	5
.....	5
User Interface	7
Mojo Application Framework	15
Palm webOS Architecture	22
Software Developer Kit (SDK)	24
Application Basics	26
Getting Started	27
News	31
Controllers	49
Widgets	52
All About Widgets	53
Using Widgets	56
Buttons and Selectors	56
Lists	61
Events	87
Summary	90
Dialogs & Menus	90
Dialogs	91
Menus	99
Commander Chain	117
Summary	119
Advanced Widgets	119
Indicators	119
Scroller	126
Pickers	133
Advanced Lists	138
Viewers	144
Summary	149
Data	149
Using Cookies	150
Working with the Depot	152
HTML 5 Storage	155
AJAX	157
Summary	161
Advanced Styles	161
Typography	162
Images	168
Touch	173
Light & Dark Styles	175
Summary	176
Application Services	177
Using Services	177
Core Application Services	182
Palm Synergy™ Services	185
Viewers & Players	192
Other Applications	193
Summary	194
System & Cloud Services	194
System Services	195
Cloud Services	205

Summary	206
Background Applications	207
Stages	207
Notifications	210
Dashboards	219
Advanced Applications	223
Background Applications	236
Summary	240
Localization and Internationalization	240
Locales	241
Localization	243
Internationalization	252
Summary	254
News Application Source Code	254
.....	255

Copyright

Copyright © Palm, Inc., 2009. All rights reserved.

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://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The vulpine phalanger and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Introduction

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 have 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 towards 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 mash-ups 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 that can deliver a superior user interface on highly optimized hardware while leveraging web services and applications stand to gain.

P.1.

P.1.1. Mobile Web Challenges

But the challenge for the 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 phone. 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 user interface model, 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 then 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 are not able

to 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.

P.1.2. 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 enabling you to build applications using common web languages and tools but 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 but while similar to web applications, webOS applications are actually native applications. This application model enables you to use the same languages and tools that you use to build web content today to build powerful mobile applications.

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.

P.1.3. 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, almost always with the user even though not always 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 the user of upcoming calendar events or incoming emails are common, but webOS applications can notify the user about traffic on the route to their next appointment, or monitors the user's social network feeds. A webOS movie guide allows you to find movies within your 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 that type of platform and shows you how to build those next generation applications and with them, the new web—the mobile web.

P.1.4. 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 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 was written 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.

The book then is not a comprehensive reference, but more of a guided tutorial. It covers all the basics for creating and building an application, 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. The 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 from other sources such as Douglas Crockford's outstanding

JavaScript: The Good Parts or the comprehensive **JavaScript: The Definitive Guide** by David Flanagan.

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 multi-tasking 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 touchpanel though doesn't require one. Because the user interface and application model are built around a web browser, 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, CSS and JavaScript that you'd use to develop web applications. Palm has extended the standard web development environment through a JavaScript framework that gives standardized UI widgets, and access to selected device hardware and services.

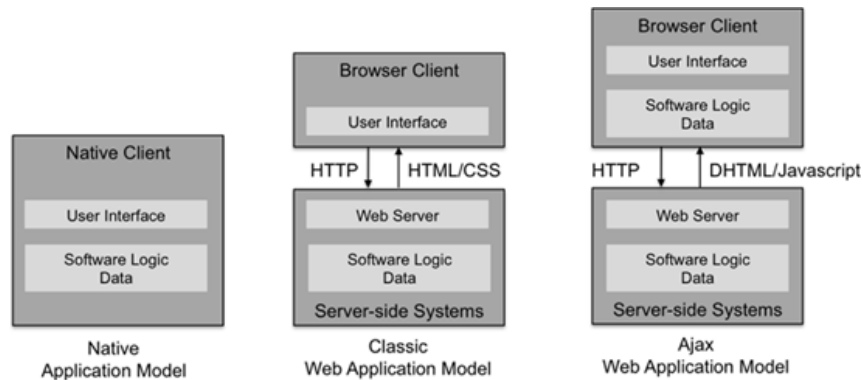
The user experience is optimized for launching and managing multiple applications at once. WebOS is designed around multi-tasking, and makes it utterly simple to run background applications, to switch between applications in a single step, and to easily handle interruptions and events without losing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

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

1.1.

1.1.1. 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 user interface 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 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. As a result, Ajax applications are able to deliver a richer and more responsive user experience. Some of the best examples of this richer experience are the map applications, which enable users to pan and zoom while asynchronously retrieving needed tiles from the web server.

Ajax applications have some significant advantages over embedded applications. They can be more easily deployed and updated through the same search and access techniques used for web pages. 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.

1.1.2. 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 user interfaces. The framework also includes event handling, notification services and a multi-tasking 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 HTML5 storage functions, and you can access data from some of webOS's 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 User Interface (UI) System Manager anchored by the open source Webkit core. The System Manager provides a full range of system user interface features including: navigation, application launching and lifecycle management, event management and notifications, system status, local and web searches, and rendering 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 information 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, browser, email and messaging. Other applications are included in the initial release, such as a camera, photo viewer, audio/video player and map application, but the full application suite for a given webOS device will vary depending on the model and carrier configuration.

1.2. User Interface

Palm webOS is designed for mobile, battery-operated devices with limited though variable screen sizes, and a touch-driven user interface. 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.

1.2.1. Navigation

Navigation is based upon 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 checkbox or selecting an object.

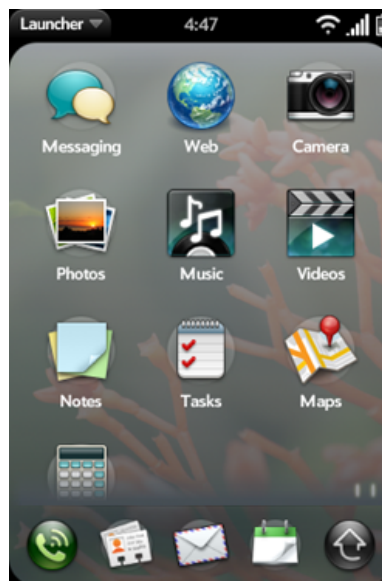
- *back* (the inverse of *open*). This feature looks like the opposite of a tap: the item compresses down to its summary in the containing context where it belongs. Typically, it reverses a view transition, as going from a child view to a parent view.
- *scroll* (flick and quick drags are used to scroll through lists and other views.

Beyond this, you can learn to pan, zoom, drag & 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.

1.2.2. Launcher

When you turn 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, a search can be initiated 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.

Figure 1-2. Quick Launch bar and Launcher



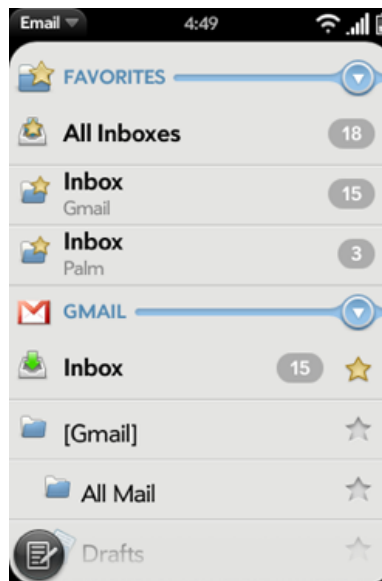
The launched application takes over the available screen becoming the *foreground* application; the application's view replaces the wallpaper image and the Quick Launch bar is dismissed. The status bar remains and is always visible except for full screen mode, which

is available to applications such as the video player, or others that request it. This sequence is fluid and smooth, as you will see with all webOS transitions.

1.2.3. Card View

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

Figure 1-3. Email Application

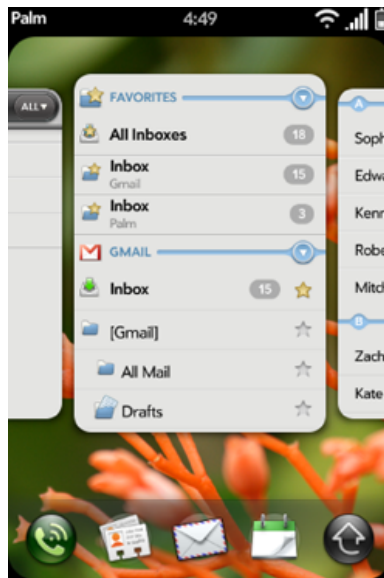


But running one application at a time, or performing one *activity* at a time, can be terribly restrictive, and inefficient. Palm webOS was designed to make it easy to work on more than one thing at a time. **Simply pressing the Center button brings up a new view,** the *Card view*, an example of which is shown in Figure 1-4. From the Card view, you can switch to another activity simply by scrolling to and tapping the card representing the activity. Alternately, you can launch another application from the Quick Launch bar.

The Card view was inspired by the way one handles a deck of cards. Cards can be fanned out to see what card is where. Within a deck of cards, any single card can be selected or removed with a simple gesture, or moved to a new location by slipping it between adjacent cards. The webOS Card view can be manipulated 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.

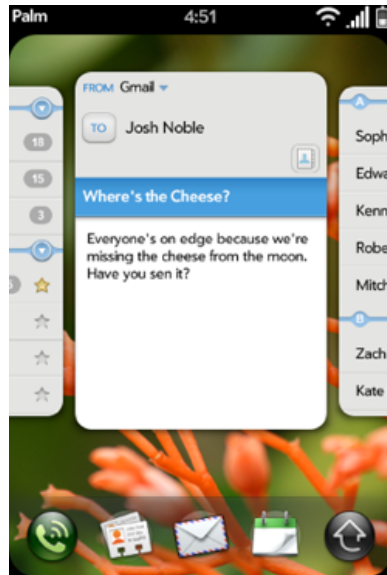
Figure 1-4. Card view with Email app and other apps



We've introduced the term *activity*, which needs further explanation. Often by design, you will work on one activity at a time with many applications, but with some applications it is more natural to work on several activities in parallel. A common email activity is writing a new email, but in the middle of writing that email, you may want to return to the inbox to look up some information in another email or perhaps read an urgent mail that has just arrived.

With a webOS device, the draft email has its own card separate from the email inbox card. In fact, you can have as many draft emails, each in their own card, as you need; each is considered a separate activity and independently accessible. Switching between emails is as simple as switching between applications and your data is safe, as it is always saved. [Figure 1-5](#) shows the Card view with the Email application's inbox card and a draft email compose card.

Figure 1-5. Card view with Email Application and New Email



1.2.4. Notifications and the Dashboard

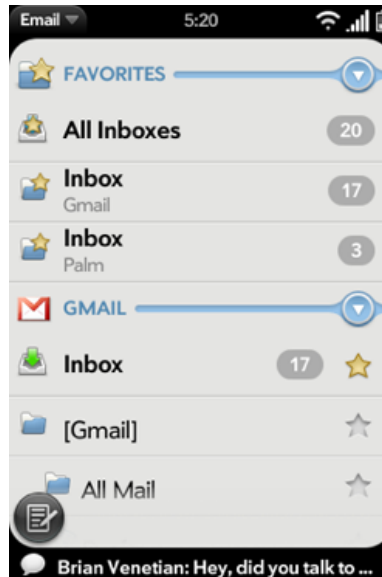
What happens to the current foreground application when you switch to a new application? The previous application is not closed but continues to run as a *background* application. Background applications can get events, read and write data, access services, repaint themselves and are generally not restricted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

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

- *Popup*. Non-modal dialogs which are of fixed height and include at least one button to dismiss the dialog
- *Banner*. Non-modal icon and single non-styled string of text

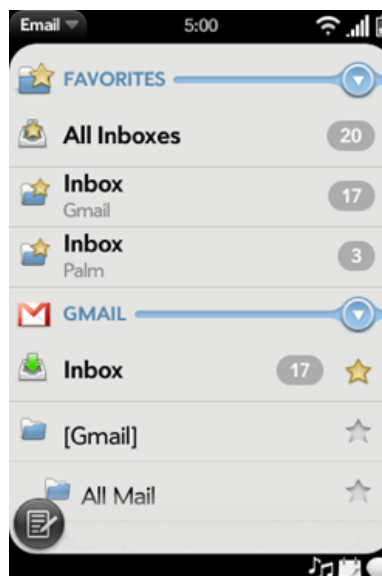
Popup notifications are disruptive, appropriate for incoming phone calls, calendar alarms, navigation notifications and other time sensitive or urgent notifications. Users are forced to take action with pop-ups or explicitly clear them but since they are not modal, users are not required to respond immediately.

Figure 1-6. Banner Notification



Banner notifications are displayed in a slow crawl 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 selectively 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 1-7. Notification Icons

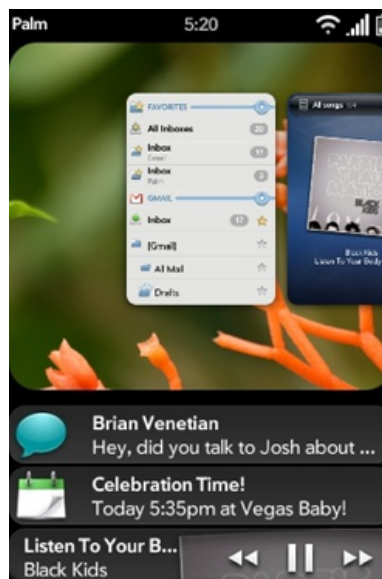


At any time, the user can tap the Notification bar, which brings up the *dashboard*, shown in [Figure 1-8](#). Notifications that are not cleared will display their current status within a dashboard *panel*.

Dashboard panels though are more than just summaries of notifications but are dynamic views enabling 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.

The Notification Bar and Dashboard manage interruptions and events, keeping you abreast of changes in information without disrupting your current activities. It may help to think of them as an event-driven model for viewing and managing your world, while the Card view provides you with task-oriented navigation tools. The combination gives you a few powerful tools from which you can quickly track and access what you need when you need it.

Figure 1-8. Dashboard



Headless 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.

You will tend to use the Card view to switch between tasks, launch applications and otherwise perform activities. Dashboard is used to monitor your world, to see what's changed or what events have taken place, which will often drive new activities.

1.2.5. User Interface Principles

There are some foundational principles or values that support the overall webOS user experience; application designers can exploit these same principles to more deeply integrate the application into the overall device experience and enhance the user's experience. Developers can rely on the framework to provide most of what is required at an implementation level, but the application design should anticipate these needs.

Here are the key principles to keep in mind while designing your application:

- *Physical metaphors* are reinforced through direct interaction with application objects and features, instant response to actions, followed by smooth display and object transitions with physics-based scrolling and other movement. For example, objects are deleted by flicking off screen and editing is in place without auxiliary dialogs or scenes.
- Maintain a *sense of place* with repeatable actions, reversible actions, stable object placement and visual transitions taking the user from one place to the next.
- Always display *up-to-date data*, which requires both pushing and pulling the latest data onto the device so that the user is never looking at stale data when more recent data is available. But this also means managing on-device caches so that when the device is out of coverage or otherwise off-line, 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 while efficient for experienced users.
- *Minimize the steps* for all common functions; put frequently executed commands on the screen, the next most frequent under the menus. Avoid preferences and settings where possible and where not, keep them minimal.
- *Don't block the user*; don't use a modal control when the same function can be carried out non-modally.
- *Be consistent*; help the user 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 strongly 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, to smoothly transitioning display changes, and object transitions.

You can find a lot more on the user interface guidelines and design information in the Palm webOS SDK under the Design Guide. 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.

1.3. Mojo Application Framework

A webOS application is similar to a web application based on standard HTML, CSS, and JavaScript, but the application lifecycle is different. Applications are run within the UI System Manager, 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 are delivered as a JavaScript framework, called Mojo, which supports common application-level functions, UI widgets, access to built-in applications and their data, and native services. To build full-featured webOS applications, many developers will also leverage HTML5 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 registering for events and DOM handling among many other great features.

The framework provides a specific structure for applications to follow 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.

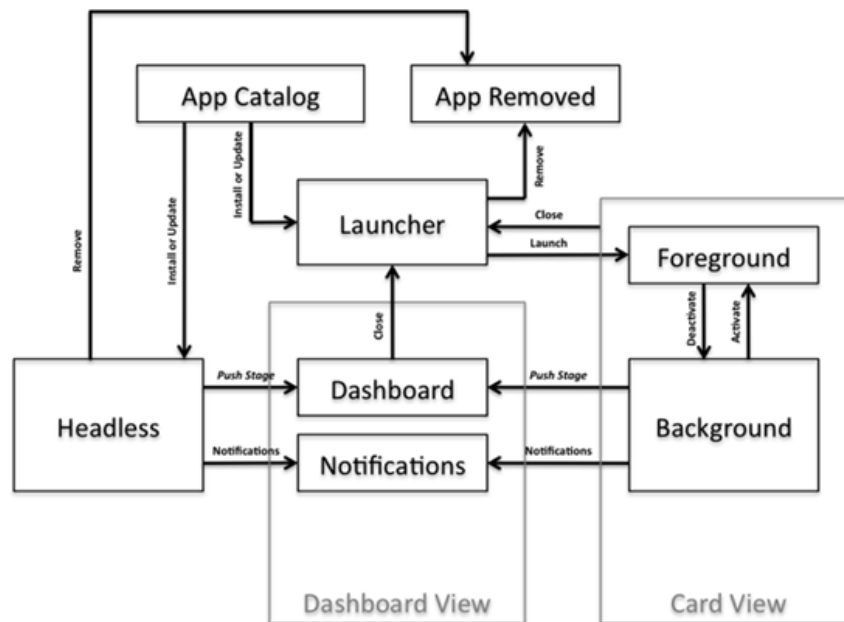
You will get a more extensive overview of Mojo in [Chapter 2](#), and details on widgets, services and styles starting in [Chapter 3](#). For now, you should know that the framework includes:

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

1.3.1. Anatomy of a webOS Application

Outside of the built-in applications, webOS applications are deployed over the web. They can be found in Palm's *App Catalog*, an application distribution service, built into all webOS devices and available to all registered developers. The basic lifecycle stages are illustrated in [Figure 1-9](#).

Figure 1-9. Application Stages



Downloading an application to the device initiates installation of the app provided that it has been validly signed. After installation the application will appear in the Launcher. If it is a *headless* application, then a card is not required and instead the application can utilize just a dashboard and communicate to the user through notifications. Headless applications typically include a simple card based preferences scene to initiate the application and configure its settings. Note that headless applications require at least one visible stage at all times (either a card, dashboard or alert) to not be shut down.

Other applications are launched from the launcher into the foreground and may be switched between foreground and background by the user. 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.

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

The user can opt to 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.

1.3.1.1. 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 a declarative HTML structure similar to a conventional HTML window or browser tab. Palm webOS 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, or other cards associated with different activities within the application. You should refer back to the earlier example of the Email application where the main card held the Email inbox and another card held a draft Email. Each email card is a separate stage, but part of the same 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 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, 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 stageController functions specifically designed to assist you, which you can find in detail in [Chapter 2](#), Application Basics, and [Chapter 3](#), UI Widgets.

1.3.1.2. 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 app. In addition, all applications will have an *index.html* file, an *icon.png* for application's Launcher icon, and an *app* folder, which provides a directory structure for *assistants* and *views*.

By convention, if the app has images, other javascript or application-specific CSS, then these 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 stage and scene assistants to perform the application's setup functions and to activate the first scene. From this point, the application would be 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 apps, such as a Calculator, which the user can launch, interact with and then set aside or close;
- Headless apps, 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 apps like a social-networking app, which provides a card for interaction or viewing and a dashboard giving status;
- Complex multi-stage apps like Email, which can have an Inbox card, one or more Compose cards, along with a dashboard showing email status. When all the cards are closed, Email will run headless to continue to sync email and post notifications as new emails arrive.

1.3.1.3. Events

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

The UI Widgets have a number of custom events, which 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 works within the DOM event model but includes support for listening to and generating custom Mojo event types and is more strict with parameters; Mojo checks parameters to confirm that they are properly defined and typed.

The webOS Service functions work a bit differently, with registered callbacks instead of DOM-style events, and are covered starting in [Chapter 7](#). The event-driven model isn't conventional to web development, but has been part of modern OS application design and derives from that.

1.3.1.4. Storage

Mojo supports the HTML5 database functions directly and provides high-level functions to support simple Create, Read, Update or Delete (CRUD) operations on local databases. Through these Mojo Depot functions, you can create a local database and add, delete or retrieve records individually or as a set. It's expected that you'd use databases for storage of application preferences, or cache data for faster access on application launch or for use when the device is disconnected.

1.3.2. UI Widgets

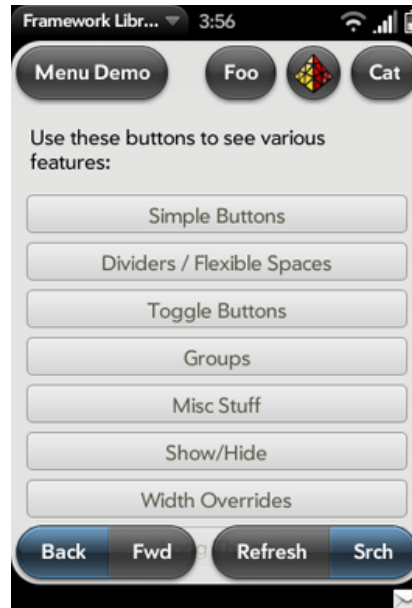
Supporting webOS's user interface are UI *Widgets* and a set of standard styles for use with the widgets and within your scenes. Mojo defines default styles for scenes and for each of the widgets. You get the styles simply by declaring and using the widgets, and you can also override the styles either 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 other widgets, including other lists, icons and images.

There are *Simple Widgets*, including buttons, checkboxes, sliders, indicators, and containers. The *Text Field* widget includes text entry and editing functions, including selection, cut/copy/paste, and text filtering. A Text Field can be used singly or in groups or in conjunction with a List widget.

Menu widgets can be used within specified areas on the screen; at the top and bottoms are the *View* and *Command* menus and they 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. Each of the various menu types is shown in [Figure 1-10](#).

Figure 1-10. Application Menu Types



For *Notifications*, you can choose from a Popup Notification or a Banner Notification, both of which post notifications for applications in the notification bar.

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

1.3.2.1. Using Widgets

A widget is declared 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 is required to reference the widget from your Javascript and must be unique.

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 = { property : '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 setup `my-toggle` passing a set of attributes, `toggle`, and a data model, `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, you would select `#my-checkbox` in your CSS and apply the desired styling (or use `.checkbox` to override the styling for all checkboxes in your app). For example, to 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 at this point. [Chapter 3](#) and [Chapter 4](#) describe each of the widgets and styles in complete detail.

1.3.3. Services

Even limiting yourself to just webOS's System UI, application model and UI widgets, developers would have some unique opportunities for building web applications, particularly with the notification and dashboards. But they'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 (Create, Read, Update and Delete) functions for the calendar, contacts and todo 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 controller function, `serviceRequest`**. The request passes a JSON object specific to the called service and specifying callbacks for success and failure of the service request.

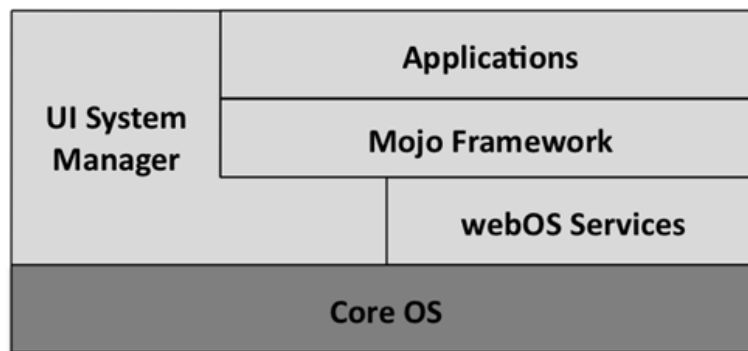
Starting with [Chapter 7](#), you'll find a full description of the general model and handling techniques as well as enumeration of all the services and the specifics for using each one.

1.4. Palm webOS Architecture

The 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 the various 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



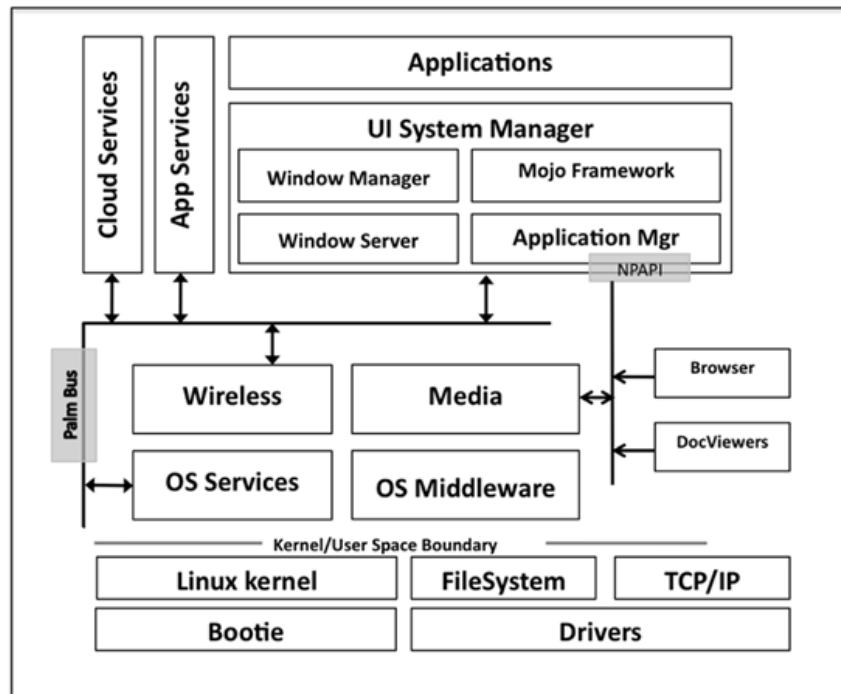
This overview is included as background in case you want to have an idea of how webOS works—this information is not needed to build applications so you can skip it if you aren't interested.

1.4.1. Application Environment

The application runtime environment is managed by the UI system manager, which also presents the System UI manipulated by the user. The framework provides access to the UI Widgets and the Palm Services. Supporting this environment is the Core OS environment, an embedded Linux OS with some custom sub-systems 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](#) describes 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 sub-systems. Let's take a brief look at how this all works together.

The UI System Manager or UI SysMgr, is responsible for almost everything in the system that is user visible. 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 apps, the status bar and the Launcher. The Application Manager runs in a single process, schedules and manages each of the running applications, and handles all rendering through interfaces to the Graphics sub-system and on-device storage through interfaces to SQLite.

Applications rely on the framework for their UI features set and for services access. 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.

1.4.2. 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 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 and WiFi networks when available, switches connections dynamically, prioritizing WiFi connections when both are available. EVDO or UMTS telephony and WAN data is supported depending upon the particular device model. Palm webOS also supports most standard Bluetooth profiles and provides simple pairing services. The Bluetooth sub-system is tightly integrated with audio routing to dynamically handle audio paths based upon user preferences and peripheral availability.

The media server is based upon *gststreamer* and includes support for numerous audio and video codecs, 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.

1.5. Software Developer Kit (SDK)

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 register as a Palm developer and download the Palm Software Developer Kit (SDK). The SDK includes the development tools, sample code, the Mojo Framework, along with access to the Palm Developer Wiki, where developers will find formal and informal training materials, tutorial and reference documentation. Palm also provides registered developers with direct technical support by email or through interaction in a hosted developer forum.

1.5.1.

1.5.1.1. Development Tools

The Palm Developer Tools (PDT) are installed from the SDK and include targets for Linux, Windows (XP/Vista) and Mac OS X. The tools enable you to create a new Palm project using sample code and framework defaults, search reference documentation, debug your app in the webOS emulator or an attached Palm device, and publish an application. [Chapter 2](#) includes more details about the tools in Palm's SDK and third-party tools, but you'll find a brief summary in [Table 1-1](#) below.

Table 1-1. Palm Developer Tools

Tools	Major Features
SDK Bundle Installer	Installs all webOS tools & SDK for 3 rd party editors
Emulator	Desktop Emulator and Device Manager
Command-Line Tools	Create New Project
	Install & Launch in Desktop Emulator or Device
	Open Inspector/Debugger Window
	Package & Sign App

The tools can be installed and accessed as command-line tools on every platform and include some bundles for integration into popular HTML editors and as a plug-in to Eclipse and Aptana Studio, a popular Javascript/HTML/CSS editor for Eclipse. Refer to Palm's Developer portal for the most current list of supported editors and tool bundles.

1.5.1.2. Mojo Framework and Sample Code

The SDK installation includes a copy of the 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 with debugging 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 give you some starter applications to review and leverage as you choose.

1.5.1.3. Developer Portal

Your main entry point is <http://developer.palm.com/>, which is where Palm hosts the Developer Portal. The portal provides access to everything that you might need to build webOS applications, including access to the SDK, all development tools, and documentation and training materials. Various support services are available, including developer forums, the virtual device lab for testing on webOS devices, bug reporting and direct support options.

The Developer Portal provides 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 App Catalog application. Applications need to be signed for installation on a webOS device, and through the portal you can access the signing tools and related support.

1.5.2. Summary

In this introductory chapter, you were introduced to webOS, Palm's next generation operating system. The following chapters will cover each of these topics in far more detail but this chapter should have helped you understand 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 app can be published and 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 be headless. Or consider an application that can open new windows for each new activity, enabling the user to multi-task 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 given 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 UI 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 HTML user interface, interacts with page elements and issues AJAX calls to web servers. The UI can be styled with CSS—either to make your application look and feel consistent with Palm's style guidelines or to make your own.

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 life cycle. 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 making user interface elements, database queries, and other 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 user interface 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 app. Along the way, we will go into detail on how to use the framework and apply the different APIs, widgets and styles.

2.1. Getting Started

You'll find everything you need to get started developing Palm webOS applications at the Palm Developer Portal (<http://developer.palm.com>). You'll need to sign up as a Palm developer then 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.

Installation will put a copy of the Palm Development Tools (PDT) and the Palm Mojo framework into a project directory (as shown in [Table 2-1](#)).

Table 2-1. Palm Development Tools Install Directory

Platform	Location
Mac OS X	<code>/opt/PalmSDK/Current/</code>
Windows XP/Vista	<code>c:\palm</code>
Linux	<code>/opt/Palm</code>

The installer will give you the option of installing different tool bundles. The Palm Development Tools package includes a collection of command-line tools, and on all platforms you can run any of these tools from a terminal command-line. 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, so the tool references and examples are either shown in that context or the command line. All of the bundles are fairly similar so you should be able to translate the examples pretty easily if you are developing on either Windows or Linux, or with a different editor. The command-line option for the tools will also be shown in the examples and is the same on every platform.

2.1.1. Creating your App

Palm webOS applications have a standard directory structure with naming conventions and some required files. To enable you to 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:

```
> palm-generate AppName
```

Or you can run it with a menu command in any of the editing tools supported in the SDK. In TextMate, the command is **Bundles→Mojo→Generate App**.

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

```
AppName
-> app
  -> assistants
    -> first-assistant.js
    -> second-assistant.js
    -> ...
  -> models
    -> model1.js
    -> model2.js
    -> ...
  -> views
    -> first
      -> first-scene.html
    -> second
      -> second-scene.html
    -> ...

-> appinfo.json
-> icon.png
-> images
  -> image1_
  -> image2_
  -> ...
-> 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 based on the Model-View-Controller (MVC) pattern.

In the `assistants` directory of your application there are scene assistants. 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.

Any application data models reside in the `models` directory. Note that applications may not have need of any models, if they manipulate only web-based data or define their data inline.

All layout files are located in the `views` directory of your application. A scene assistant has one main HTML view file, which provides the static 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 cascading style sheets (CSS) 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` object 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 in other ways serves as a unique application identifier.

Table 2-2. `appinfo.json` properties and values

Property	Values	Required	Description
<code>title</code>	Any	Yes	Name of application as it appears in Launcher and in app window
<code>type</code>	<code>web</code>	Yes	Conventional application
<code>main</code>	<code>index.html</code>	Yes	Application entry point; defaults to <code>index.html</code>
<code>id</code>	Any	Yes	Must be unique for each application
<code>version</code>	<code>x.y</code>	Yes	Application version number
<code>noWindow</code>	<code>true</code> <code>false</code>	No	Headless application; defaults to <code>false</code>
<code>icon</code>	file path	No	Application's launcher icon; defaults <code>icon.png</code>

Property	Values	Required	Description
minicon	file path	No	Notification icon; defaults to <i>miniicon.png</i>
category	Any	No	Default category for application

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 pixels by 64 pixels, encoded as a PNG with 24 bit/pixel RGB and 8 bits alpha. The icon's image should be about 56x56 pixels within the PNG bounds.

NOTE

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 looks like this:

```
[
  {
    "source": "app\\assistants\\app-assistant.js"
  },
  {
    "source": "app\\assistants\\stage-assistant.js"
  },
  {
    "source": "app\\assistants\\first-assistant.js",
    "scenes": "first"
  },
  {
    "source": "app\\assistants\\second-assistant.js",
```

```

    "scenes": "second"
  },
]

```

The app-assistant file path comes first, followed by the stage-assistant and the scene file paths after that. The scene file paths can be in any order but must include both the source and scene properties. 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 test libraries under `tests`. The required elements are:

- The *app* directory and everything within it
- *index.html* as the application launch point
- *appinfo.json*

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

2.1.2. Testing and Debugging

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

So you'll want to use the webOS Emulator with integrated JavaScript debugger and DOM inspector. Unlike the other Palm 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. You can launch the emulator directly and it will bring up a window that looks like a Palm Pre, or you can push the current version of your application to the emulator with the command **Bundles→Mojo→Run**, or you can use command line tools, `palm-package` to package your application and `palm-install` to run it on the emulator.

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

When you enroll in the Palm Developer program, you'll have an opportunity to generate signing certificates to support both on-device testing and publishing your applications. If you didn't create those certificates when you enrolled then you can go back to the portal when you're ready. However, you will need the certificates before you can test on device.

2.2. News

After the core webOS applications, one of the first apps built for webOS was the **News app**. In August 2008, the webOS platform was far enough along that we 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 app 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 app 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 addicting.

An RSS reader is a useful application and although simple to write, it will use 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 app up bit by bit throughout the book to demonstrate how to write a webOS app 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 what is not covered.

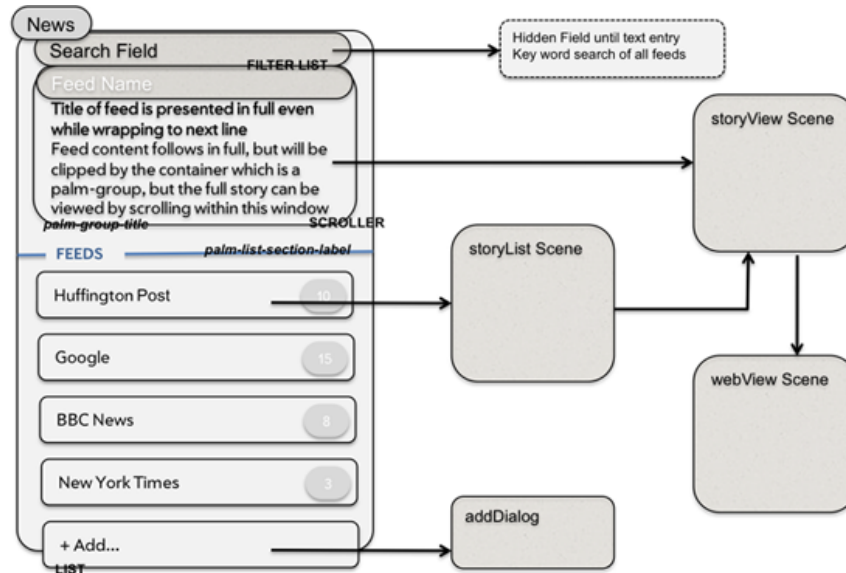
News manages a list of newsfeeds, periodically updating the feeds with new stories. The major features are:

- You 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 off-line viewing.
- The original and full story can be viewed in the browser.
- Keyword searches over the entire feed database.
- Stories are shared over email or messaging.
- A local news feed is compiled using current location for news, events and related local information.

2.2.1. 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 [Figure 2-1](#).

Figure 2-1. Part of the News wireframes



This shows the main scene of the News app 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 that is another base style called *palm-list-section-label* that divides the featured story area from the feed *list* widget below it. At the top of the scene is a search field built with a *filterlist* widget which has an attribute directing the framework to hide the field until some text is keyed, at which point the entered text will be used in a keyword search of the entire feed database.

NOTE

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.

The application supports other scenes, which can be brought 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 as well as for any dialogs, such as the dialog in this scene that adds new feeds to the list.

Chapter 1 included 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 re-arranged. New feeds are added 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, which you may find helpful. It covers broad user experience guidelines for designing webOS applications and includes technical details that are essential for graphics 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 Pre, 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 48px.

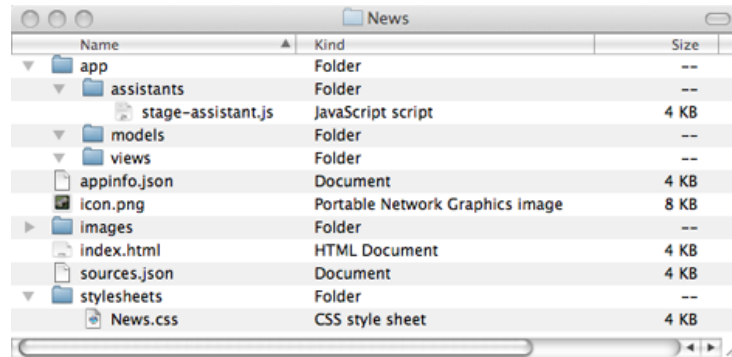
The minimum font size for lower case text is 16px and 14px for all caps.

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

2.2.2. Create the News App

We'll start by repeating the steps covered in the last section to create a new application project. Select **Bundles**→**Mojo**→**Generate App**, but this time name the app *News* and click OK. You'll see an initial app structure (as shown in [Figure 2-2](#)).

Figure 2-2. Creating a new Palm webOS Application



As discussed in the last section, 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, we left the app title as *News* but put it 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>
  <script src="app/assistants/stage-assistant.js" type="text/javascript"></script>

  <!-- application stylesheet should 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>
```

2.2.3. Customizing the Launcher Icon and App ID

While not strictly necessary, each app should have a unique launcher icon. A custom graphic was generated and replaced the generic *icon.png* in the *News* directory. If you don't want to draw a custom graphic, you can re-purpose 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 app on the Macintosh.

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've used `palm`) or some other unique string. We've also modified 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",
  "icon": "icon.png"
}
```

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

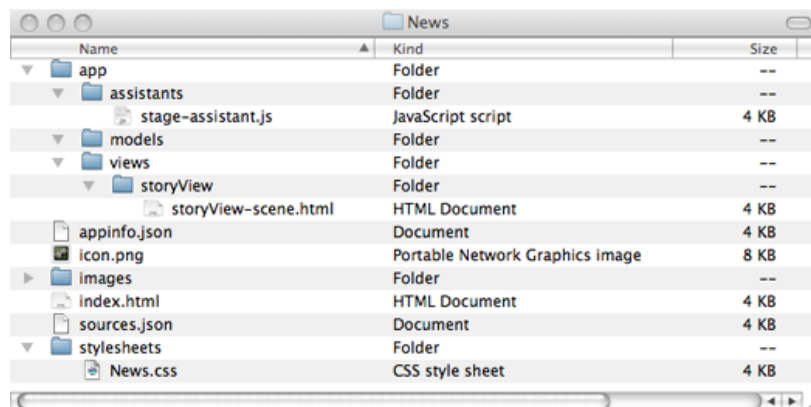
2.2.4. Adding the First Scene

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

2.2.4.1. 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. Start with the view file; create a directory with the scene name, `storyView`, and within it, an empty HTML file with the scene name followed by `-scene.html` (`storyView-scene.html`). 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 the structure quite a bit for your application. Place your scene's html file where you

want it under the app directory. As an example, you can put a scene called scene-one into:

```
app/views/example/scene-one/scene-one-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({id:"mojo-scene-scene-one",
name:"scene-one", sceneTemplate:"example/scene-one/scene-one-scene"});
```

This is useful for code organization purposes; particularly with large apps or where you might want re-use code between 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 DIVs 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 id="test" class="palm-page-header-wrapper">
      <div id="storyViewTitle" class="title left">#{title}</div>
    </div>
  </div>
  <div id="storyViewSummary" class="itemFull">#{text}</div>
</div>
```

You'll notice that `storyViewTitle` is wrapped by two DIVs that 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 of more 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).

To complete the scene, we need to add the scene assistant and modify the stage assistant to push the new scene.

2.2.4.2. The Scene Assistant

Mojo requires that the scene assistant's name match the view name, so create the scene assistant called *storyView.js* in the *News/assistants* directory. The assistant includes a function definition along with the four standard methods:

- setup
- activate
- deactivate
- cleanup

If any are missing, then the base class controller methods will be used, which are essentially empty functions. The function naming is important; the assistant's name should be the same as the filename with the removal of any delimiters and a capital letter to start, as in the example below.

```
// StoryViewAssistant()
//
// Displays the story element in a full scene view
//

function StoryViewAssistant() {

}

StoryViewAssistant.prototype.setup = function() {

};

StoryViewAssistant.prototype.activate = function(event) {
    $("storyViewTitle").innerHTML = "Green Inc.: Cities
    Target Lending to Speed Energy Projects";
    $("storyViewSummary").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.";
};

StoryViewAssistant.prototype.deactivate = function(event) {

};

StoryViewAssistant.prototype.cleanup = function(event) {

};
```

The `setup` method is invoked but there isn't anything to do here until we start adding UI elements. Before the scene is pushed, the `activate` method is called and it's here that the news feed title and text are put into the DIVs and into the scene.

Prototype's function `$` is used to return the element id for the title and summary DIVs. Mojo includes Prototype in webOS, and uses it throughout the Mojo framework and the webOS core applications. While Prototype is not an official part of webOS, it is bundled with webOS and you're free to use it within your applications..

2.2.4.3. Pushing the Scene

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

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

Notice that just 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 script tag into *index.html* to load *storyView-assistant.js*:

```
.....  
<title>News</title>  
  <script src="/usr/palm/frameworks/mojo/mojo.js" type="text/javascript"  
  x-mojo-version="1"></script>  
  <script src="app/assistants/stage-assistant.js" type="text/javascript"></script>  
  <script src="app/assistants/storyView-assistant.js"  
  type="text/javascript"></script>  
.....
```

Now run the app by selecting **Run** from the TextMate Mojo Bundle Menu, or by launching the emulator directly. See [Figure 2-4](#) and [Figure 2-5](#).

Figure 2-4. An unstyled first scene.

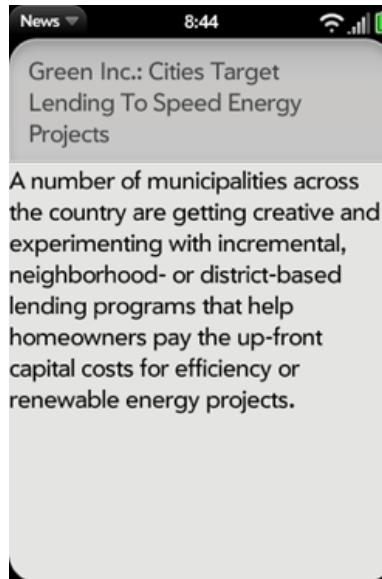
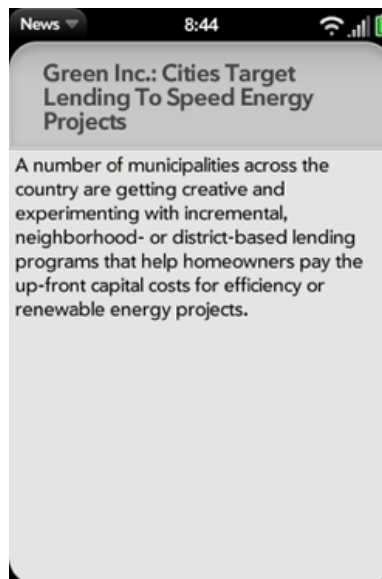


Figure 2-5. A styled first scene.



As shown in [Figure 2-4](#), it's clear that some styling is required in both the title and summary DIVs to provide some margin on the left and right sides and to fit the title text inside the header. It would also look better to have the title in a bold font-weight.

2.2.4.4. Styling the Scene

The *News.css* file is empty at this point, so simply add the following to fix up the styling:


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

.palm-page-header-wrapper    {
    padding-top: 5px;
    padding-left: 12px;
    padding-right: 10px;
    font-weight: bold;
}

.itemFull {
    vertical-align: top;
    font-size: 12pt;
    font-weight: normal;
    text-align: left;
    padding-right: 5px;
    padding-left: 5px;
}
```

There are two styling approaches here. In the first case, the style override is applied to the base style class `palm-page-header-wrapper`, which works as long as that base style isn't used in another place where the style changes will conflict. If you're worried about that, you can add another class to the element you wish to style,. 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 assign unique IDs and override the DIV style by id.

Base Style Overrides

It is often necessary to make style adjustments to get the right appearance for your application, so don't be afraid to dive into the CSS. Web Inspector includes an excellent DOM inspector to help you see what style definition is in and make the necessary adjustments.

In the second case, the styling is applied to an app-specific DIV class, which is standard CSS. You can use any standard CSS styling techniques with webOS apps—the only potential issues are where you wrap an element that has a Mojo class style within a custom style. You'll have to experiment to see what does and doesn't work in these cases.

2.2.5. Base Styles

Applying styles is an important topic so we'll return to it a few times. [Chapter 7](#) is devoted entirely to Advanced Styles, covering the use of widget styles in depth along with 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 do what you need, as it will make your applications feel like the other native webOS applications, and your users will know how to interact with them more

intuitively, based on their experience with other apps. 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.

In this section, we will give a brief overview of the webOS styles and provide some basic techniques that you can use. It's beyond the scope of this book to describe the base 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 *Styling User Guide* in the Palm SDK.

2.2.5.1. 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 then you shouldn't override these styles. These style definitions can be found in *globals.css*.

2.2.5.2. Scene Styles

The most basic style elements are those that are 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](#).

Table 2-3. Scene Styles

Style Class Name	Description
palm-page-header	Scene style with header that adjusts to text dimension; can be single or multi-line
palm-header	Scene style with pill at top; single-line only
palm-header-spacer	Keeps other scene elements from folding under for 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 these styles as is or adjust the style properties in your CSS. This is 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.

2.2.5.3. Widget Styles

Each widget is designed with a base style. In the next two chapters, we'll cover widgets in detail including the base styles and properties. Generally speaking, however, you will be able to override the widgets style in your application CSS and you can refer to the style

definitions in the framework's CSS files to guide you. [Table 2-4](#) cross-references the widget styles with the framework's CSS files.

Table 2-4. Widget Style Definitions by CSS File

Widget Styles	CSS File Location
Buttons	<i>global-buttons.css, global-buttons-dark.css</i>
Dialogs	<i>globals.css, global-dark.css</i>
Drawers	<i>global-lists.css, global-lists-dark.css</i>
Indicators	<i>globals.css, global-dark.css</i>
Lists	<i>global-lists.css, global-lists-dark.css</i>
Menus	<i>global-menus.css, global-menus-dark.css</i>
Notifications	<i>global-notifications.css</i>
TextFields	<i>global-textfields.css, global-textfields-dark.css</i>
Pickers	<i>global-menus.css, global-menus-dark.css</i>

2.2.6. App 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 function. The AppAssistant is used to handle launch arguments from the system and to setup stages if the application requires more than the main Card view or has non-standard 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; most simple single stage applications don't. But as you'll see in [Chapter 10](#), the AppAssistant is important for background applications and to apps with multiple stages.

NOTE

In [Chapter 10](#) we'll add the AppAssistant to News and explore its use in detail.

The first required controller instance is the StageAssistant. 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).

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, invoke first the `setup` and

then the `activate` method of the Scene Assistant. The `setup` method is used to instantiate widgets, setup event listeners, and perform 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 replaced 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. Apps that can support multiple activities, such as an email app that supports viewing an email list, as well as composing potentially multiple emails, should structure the app stages around activities.

2.2.7. 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 arrays to store the newsfeeds and stories. The stage-assistant is expanded to define `feedList`, a global array of newsfeed objects, each of which in turn includes an array of stories. The following code sample includes a breakdown of these data objects and expands the definition of our sample newsfeed with four sample stories.

```
// Globals - used throughout the News application
//

var feedList = [];           // News feed list

// Feedlist entry is:
// feedList[x].title         String Title entered by user
// feedList[x].url           String Feed source URL in unescaped form
// feedList[x].type          String Feed type: either rdf
// (rss1), rss (rss2) or atom
// feedList[x].numUnRead     Integer How many stories are still unread
// feedList[x].stories       Array Each entry is a complete story:
//   feedList[x].stories[y].title String Story title or headline
//   feedList[x].stories[y].text  String Story text
//   feedList[x].stories[y].unreadStyle String Style to apply when unread
//   feedList[x].stories[y].url   String Story url
//

// Push default feeds onto list

feedList.push({title:"New York Times",
  url:"http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml", type:"rss",
  numUnRead:0, stories:[]});

feedList[0].stories.push({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");

feedList[0].stories.push({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"});

feedList[0].stories.push({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"});

feedList[0].stories.push({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"});

var curFeedIndex = 0;      // Tracks current index for feedlist updates
var curFeedSource = feedList[0];

// Constants

var unreadFormatting = "unreadStyle";      // Prepend to story
    titles when unread

function StageAssistant () {

}

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

```

In the StageAssistant's 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. Otherwise there is risk of namespace collisions and unexpected behavior when running your web app; you don't know what other apps or sites may also be running at the same time.

Palm webOS is different. Each 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 `$$L`, 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 memory optimization).

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 controller, making them available to each scene method.

```
// StoryViewAssistant(storyFeed, storyIndex)
//
// 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).

function StoryViewAssistant(storyFeed, storyIndex) {

    // Save the passed arguments for use in the scene.
    //
    this.storyFeed = storyFeed;
    this.storyIndex = storyIndex;
}

StoryViewAssistant.prototype.setup = function() {

    // Hide Previous Button if first story, and Next Button if last one

    if (this.storyIndex > 0) {
        this.controller.listen('previousStory', Mojo.Event.tap,
            this.previousStory.bindAsEventListener(this));
    } else {
        $('previousStory').hide();
    }

    if (this.storyIndex < this.storyFeed.stories.length-1) {
        this.controller.listen('nextStory', Mojo.Event.tap, this.
            nextStory.bindAsEventListener(this));
    } else {
        $('nextStory').hide();
    }
}

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

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

StoryViewAssistant.prototype.activate = function(event) {

    $("storyViewTitle").innerHTML = this.storyFeed.stories[this.storyIndex].title;
    $("storyViewSummary").innerHTML = this.storyFeed.stories[this.storyIndex].text;
};
```

```

    if (this.storyFeed.stories[this.storyIndex].unreadStyle === unreadFormatting) {
        this.storyFeed.numUnread--;
        this.storyFeed.stories[this.storyIndex].unreadStyle = "";
    }
};

StoryViewAssistant.prototype.deactivate = function(event) {

};

StoryViewAssistant.prototype.cleanup = function(event) {

};

```

In the `setup` method, listeners are set up for `previousStory` and `nextStory`. These are button elements which, when tapped, will cause a new scene to be pushed with the appropriate story. The scene that is pushed is this same `storyView` scene, showing some of the flexibility of the scene model.

SceneController and Events

Already you may have noticed that we have created properties of the assistant prototype for data used by the assistant. It's a useful way to manage data that has is limited to a particular instance of the assistant. To access the data in the assistant's event handlers, you need to bind the assistant's controller instance to your event listener. The code examples throughout the book use Prototype's `bindAsEventListener` but you aren't limited to that approach.

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

```

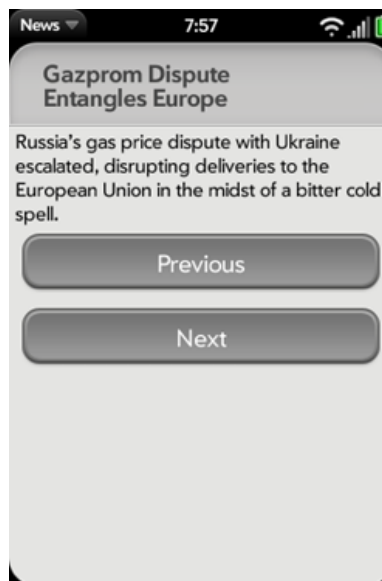
<button id="previousStory" class='palm-button'>Previous</button>
<button id="nextStory" class='palm-button'>Next</button>

```

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 app and push the buttons to go from one story to the next then back again and you should see views like 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, then 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 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`:

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

StoryboardAssistant.prototype.nextStory = function(event) {
    Mojo.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 app. It's hard to see how important this is given that this is currently such a simple app, but it will become clearer as we go forward.

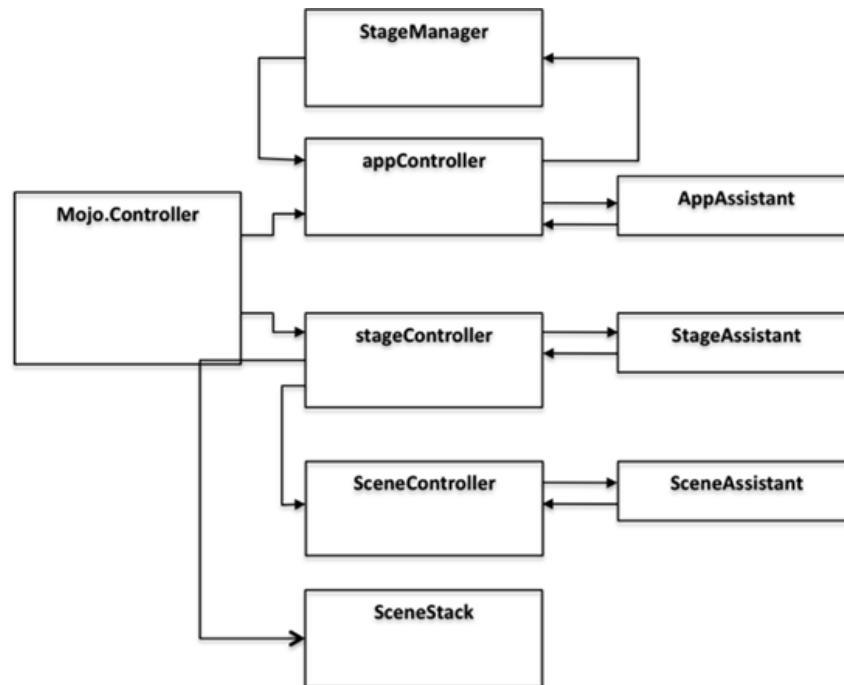
This is as much as we're going to do with News in this chapter, so if you're eager for more hands-on information, you can skip to the next chapter at this point. The remaining sections of this chapter cover more advanced topics that will help you understand the underlying framework design, but they aren't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

2.3. Controllers

So far we've used two controller classes (`stageController` and `sceneController`) and referred to a third (`appController`). All three classes are sub-classes of `Mojo.Controller`, which is mostly used as a namespace. The assistants that we've created are associated with their respective controller classes and rely heavily on the methods in those classes.

Figure 2-7 shows a class diagram showing the relationship between these classes and includes a couple of additional classes that support the controllers. `StageManager` is a class that works cooperatively the `AppController` to manage the active stages. Similarly, `SceneStack` works with the `StageController` to provide methods for managing the scene stack.

Figure 2-7. A controller class diagram



`AppController` and the use of stages within an application will be covered in-depth in [Chapter 9](#), when we cover notifications and background applications.

NOTE

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

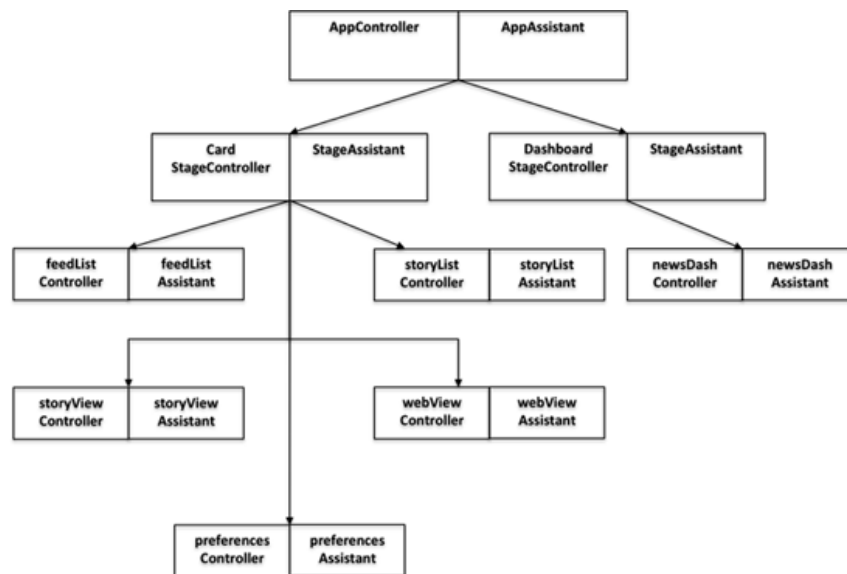
2.3.1. Controllers and Assistants

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 actually a delegate of the controller. The assistant has a `controller` property set to a reference of the associated controller, which is used to directly calling the controller's methods. The assistant defines methods of its own 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 2-8](#).

Figure 2-8. A News controller/assistant diagram



2.3.2. Scene Stack

We've already mentioned `pushScene` and `popScene`, but there are other methods that you will use to manage the scenestack. Though they are available in both classes, you should access the methods through `Mojo.Controller.StageController` and not through `Mojo.SceneStack`. Refer to the SDK documentation for a complete and up-to-date list of available `SceneStack` methods, but 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 and 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.

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

2.3.3. 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 hard-coded 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 user interface feature set, delivered in a collection of dynamic widgets. Mojo widgets are configured with `sceneController` methods and custom CSS styles and managed through Mojo events, briefly introduced to you in [Chapter 1](#).

With widgets you can build static and dynamic lists, or employ various button controls, selectors, and text fields. You can choose from several kinds of menus and dialogs, 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 would declare the widget within a scene's view file, then configure and setup 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 by giving 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 in each category to show you how to apply them in your applications.

We've included excerpts from the webOS Developer Guide in a number of spots to round out the tutorial we are presenting. Having this extra information in the context of the tutorial application might be helpful to you. You should still refer to the SDK's API documentation for details on the interfaces and calling functions.

3.1. 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. Widgets is a term widely used within web development but Mojo widgets are different than other widgets. Mojo widgets have a defined behavior but with 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, where the app specifies HTML templates that largely define the widget's structure.

3.1.1. Declaring Widgets

Widgets are declared in HTML as empty divs, as shown:

```
<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 either:

1. When a scene is pushed and the scene's view HTML includes widgets.
2. When a widget is specified in an HTML template used by another widget.

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

3.1.2. Setting Up a Widget

Before a widget is inserted into the scene, it must be set up. This should be done in the scene assistant's `setup` method, by calling `SceneController.setupWidget()`. You need to provide three arguments to this call (as 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 having dynamic attributes

For example, a `ToggleButton` would be setup this way:

```
this.toggle = { trueValue:'on', trueLabel:'On', falseValue:'off', falseLabel:'Off' };
this.toggleModel = { value:'on', disabled:false };
this.controller.setupWidget('my-toggle', this.toggle, 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 in the case that 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 a `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 is primarily to allow widgets to be easily used within lists. 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 on Lists later in this chapter.

3.1.3. 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 app (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, if you disabled the Toggle Button in our example:

```
this.toggleModel.disable = 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. Note that calling `modelChanged()` with an entirely new model object will not update the model. Instead there would be no change, since the specified model would not be used by any existing widget; no notifications will be generated or received.

The second argument identifies what 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 is an indirect way to update a widget's copy of its model; if you need to directly change the model then 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.

Here's a sample, following the 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);
```

3.1.4. 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.scrollingStarted` for the Scroller widget.

You should set up event listeners in the scene assistant's setup method when you setup 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 an 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 listener with code like this:

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

You can look at the last section in this chapter, Events, which covers the entire event model for more information, or consult the `Mojo.Event` API reference in the Palm SDK.

3.2. 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 1.0, 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, CheckBox, RadioButton, ToggleButton, ListSelector, Slider
Lists	List, FilterList, GridList
Dialogs & Containers	Dialog, ShowDialog, ErrorDialog, Drawer, Scroller
Text Fields	TextField, FilterField, PasswordField, RichTextEdit
Menus	AppMenu, CommandMenu, ViewMenu, SubMenu
Pickers	Date, File, Integer, Time
Viewers	ImageView, WebView, 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. While we won't use every widget that Mojo offers, we'll use one from every category and that will guide you on how to use any of the other related widgets.

Widgets are declared in the HTML scene, setup and rendered it from within the JavaScript assistant and can be styled through CSS. You've also seen that all widgets have an attribute object which 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.

3.3. Buttons and Selectors

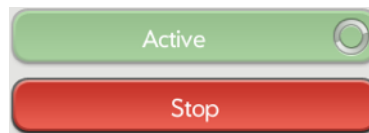
Buttons and Selectors are the simplest Mojo widgets. In the All About Widgets section previously, you saw how to use a toggle button; all the other buttons and selectors work in a very similar way. In this section, we'll work directly with the Button widget, adding one to the News application, and touch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

3.3.1. Button

You can use simple HTML buttons styled as widgets; for many use cases these will work quite well. Button *widgets*, by contrast, can have dynamic behavior, particularly the inclusion of a spinner to show activity and for you to manage the button's label through stages of an 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 button is pushed, it can change state but gracefully returns to the previous state, like a doorbell. Buttons can be styled as objects or unstyled and are usually labeled in some way with text or an image. Mojo buttons can be disabled, and can be configured to show an activity indicator.

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. As we did in [Chapter 2](#) for switching scenes, you declare HTML buttons in your view file using conventional HTML notation:

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

By assigning the button's class to `.palm-button` 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.

Override the style for either type of button in your CSS, if desired. The most typical style modification is to adjust the button width, which by default is designed to be 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.

3.3.1.1.

3.3.1.1.1. Adding a Button to News

At the end of [Chapter 2](#), we had added a scene to News 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 to show you how to add a simple widget. If you think you are already clear on how this works you can skip ahead to the next 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 id="test" class="palm-page-header-wrapper">
      <div id="storyViewTitle" class="title left">#{title}</div>
    </div>
  </div>
  <div id="storyViewSummary" class="itemFull">#{text}</div>
</div>
<div x-mojo-element="Button" id="previousStory"></div>
<div x-mojo-element="Button" id="nextStory"></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's, so we don't have to change the listener setup functions at all.

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

  // Hide Previous Button if first story, and Next Button if last one

  if (this.storyIndex > 0) {
    this.controller.setupWidget("previousStory",
      this.attributes = {
        disabledProperty: 'disabled'
      },
      this.model = {
        buttonLabel : "Previous",
        buttonClass: '',
        disabled: false
      });

    this.controller.listen('previousStory', Mojo.Event.tap,
      this.previousStory.bindAsEventListener(this));
  } else {
    $('previousStory').hide();
  }

  if (this.storyIndex < this.storyFeed.stories.length-1) {
    this.controller.setupWidget("nextStory",
      this.attributes = {
        disabledProperty: 'disabled'
      },
      this.model = {
        buttonLabel : "Next",
        buttonClass: '',
        disabled: false
      });
  }
};
```

```

        this.controller.listen('nextStory', Mojo.Event.tap,
            this.nextStory.bindAsEventListener(this));
    } else {
        $('nextStory').hide();
    }
};

```

The rest of the code stays the same. When you run this version of the app, it behaves the same as the previous version but we are using button widgets instead of the HTML button.

3.3.2. Selectors

The simple selectors will be used in other parts of the News application, shown in later examples. For now, the next few sections will briefly describe each of the selectors and how they can be used in your application.

3.3.2.1. Check Box

A Check Box widget (see [Figure 3-2](#)) is used to control and indicate a binary state value in one element.

Figure 3-2. A Check Box widget.



Tapping a Check Box 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.

3.3.2.2. Toggle Button

The Toggle Button is another widget for displaying and controlling a binary state value. As with the Check Box, the Toggle Button (see [Figure 3-3](#)) will switch between two states each time it is tapped.

Figure 3-3. A Toggle Button widget.



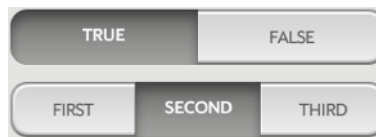
3.3.2.3. Radio Button

If you need a single widget to select from among multiple choices while also showing selection status, then a Radio Button (see [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 2 and 5 states given the typical screen size for a webOS device, but the framework won't limit you.

Figure 3-4. Radio Button widgets.



3.3.2.4. 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 popup list in which there is no practical limit to the number of options presented. It is similar to the Submenu widget behavior. [Figure 3-5](#) shows an example of the List Selector widget.

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, then you would define the array as a property in the widget's attributes. If the choices are subject to change, then attach them as a model property instead.

List Selectors in Forms

To group List Selector as you might do in a form, use the `.palm-group` unlabeled DIV class followed by `.palm-list` DIV class, then individual selector DIVs containing ListSelector 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>
</div>
```

```

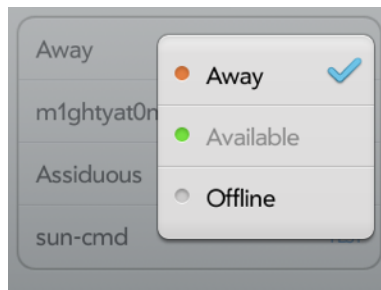
</div>
<div class="palm-row last">
  <div id="timeSelector" x-mojo-element="ListSelector"></div>
</div>
</div>

```

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 forms processing.

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 next section, Lists, for more information.

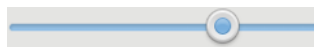
Figure 3-5. A List Selector widget example.



3.3.2.5. 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 can be dragged to the desired location. You must specify minimum (left-most) and maximum (right-most) values for the slider and you can optionally indicate intermediate choices which will trigger additional behavior. [Figure 3-6](#) shows an example of the Slider widget.

Figure 3-6. A Slider Widget example.



3.4. 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 flicks 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 but it was met and the rest of the framework took shape around their 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 widgets, GridList and FilterList, are derived from the List widget. They share many common features, but are designed around more specialized use cases. Our sample application will make use of the List and the Filter List but you won't learn about FilterList until Chapter 5, [Chapter 5](#).

3.4.1. List Widget

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 multiline text, images or other widgets. Lists can be static, where the list items are provided to the widget directly as an array, or they can be dynamic, where 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 apps 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.

3.4.1.1. 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, and hook it up to an Ajax call to get the real news feed 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, *storyList-scene.html*, declare a list widget under a `palm-header` which includes a `span` tag to which we'll later assign the list's title. The div with the `id='storyListWgt'` is the list widget declaration.

```
<div class='palm-header'>
  <div class='palm-header-center'><span id='feedTitle'></span>
</div>
<div class="palm-header-spacer"></div>
<div class="palm-list">
  <div id="storyListWgt" x-mojo-element="List"></div>
</div>
```

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

All List widgets are all built using HTML templates to layout 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.

NOTE

In Mojo, pathnames are relative to where the file containing the path is located, not from where `index.html` is located.

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 without any container. The `listTemplate` if present 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>
```

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-tap-highlight="momentary">
  <div id="storyTitle" class="listTitle truncating-text #{unreadStyle}">#{title}</div>
  <div id="storyText" class="listText truncating-text">#{text}</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 items properties of those names which are substituted into the template.

The `#{unreadStyle}` template references another items property which forces some styling specifically for the story titles that are not read. This example demonstrates that templates can be used with div attributes as well as content. In our CSS further on, there will be some styling applied to `unreadStyle`.

The `feedList` wraps the List widget with some specific styles to get the visual appearance shown in [Figure 3-7](#) and [Figure 3-8](#). You should review the SDK's UI & Style Guidelines for a complete discussion of Mojo styling but to summarize briefly, there are three levels of styles at work in the News `feedList`:

- `.palm-group` forms the outer frame to enclose all the list elements, and has an optional `.palm-group-title` to define the title of the group

- `.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

Back to the example, to implement the `feedList`, add the *storyList-assistant.js*:

```
//
// StoryListAssistant - Displays the feed's stories in a list,
// user taps display the
// selected story in the storyView scene.
//
// Arguments:
// selectedFeed      Feed to be displayed
//

function StoryListAssistant(selectedFeedIndex) {
  this.feed = feedList[selectedFeedIndex];
  this.feedIndex = selectedFeedIndex;
}

StoryListAssistant.prototype.setup = function() {

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

```

    },
    this.storyModel = {
        items: this.feed.stories
    }
};

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

StoryListAssistant.prototype.activate = function() {

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

    // Update story list model in case unreadCount has changed
    this.controller.modelChanged(this.storyModel);

};

// readStory - handler when user taps on displayed story, push that story to
// the storyView scene
StoryListAssistant.prototype.readStory = function(event) {

    Mojo.Controller.stageController.pushScene("storyView", this.feed, event.index);
};

StoryListAssistant.prototype.deactivate = function(event) {
};

StoryListAssistant.prototype.cleanup = function(event) {
};

```

When the scene is instantiated with a call to the `StoryListAssistant` function, the passed feed index is used to assign 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 twice its default of 20. For your lists, you should use the default but after testing adjust it if needed.

renderLimit

The number of list elements that will be rendered by the list widget into the DOM at any one time is defined by `renderLimit`. It usually shouldn't need to be specified, but if your list items are very short then 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, but there needs to be enough to avoid have the list scrolling overrun the display list.

Ideally, this would be calculated automatically, but it's currently not, so you may need to adjust this if you are seeing empty spaces on your lists when scrolling.

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.

During activate, the list title is assigned to display in the header, and 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.

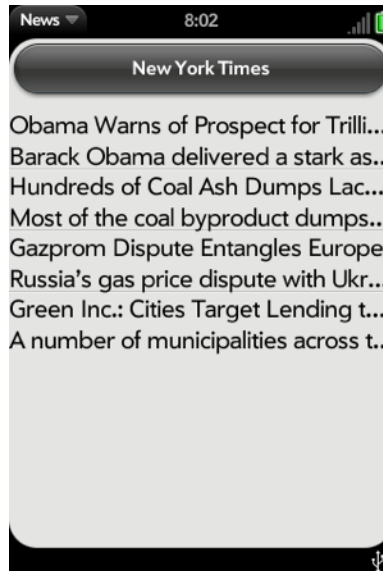
Lastly, we have to change the *stage-assistant.js* to push the `storyList` scene instead of the `storyView` scene:

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

You'll note that the `curFeedIndex` argument is still used but not `curFeedSource` which was dropped. The `feedList` is the feed source now, using all the stories in the selected feed.

Finally load the new assistant in `index.html`, then launch the application. The new scene with all the stories from the sample feed is shown in [Figure 3-7](#).

Figure 3-7. A StoryList scene.



It's not very pretty, so add some CSS to fix up the sizing and spacing and look at the improvements shown in [Figure 3-8](#).

```
.listTitle {
    padding-top: 10px;
    padding-right: 5px;
    padding-left: 5px;
    font-size: 14pt;
    text-align: left;
    font-weight: normal;
}

.listText {
    vertical-align: bottom;
    padding-bottom: 10px;
    padding-right: 10px;
    padding-left: 5px;
    font-size: 10pt;
    height: 20px;
    font-weight: normal;
    text-align: left;
    overflow-x: hidden;
    overflow-y: hidden;
}

.unReadStyle {
    font-weight: bold;
}
```

Figure 3-8. A StoryList Scene with styling adjustments.



3.4.1.2. Using *sources.json*

We just added our second scene and as the application grows, we want to be more aware of load times. Now is a good time to add a *sources.json* file and specify the scene files so that they are loaded as each scene is pushed rather than at application launch time. Make this change by creating the *sources.json* file in the app's root directory if it's not already there.

```
[
  {
    "source": "app\\controllers\\stage-assistant.js"
  },
  {
    "source": "app\\controllers\\storyList-assistant.js",
    "scenes": "storyList"
  },
  {
    "source": "app\\controllers\\storyView-assistant.js",
    "scenes": "storyView"
  }
]
```

Remove the script loading tags for these files from *index.html*, but leave the script load for *mojo.js*. From here whenever adding a new scene to the application, add the corresponding entry to *sources.json*.

3.4.1.3. Back to the News: Ajax Requests

[Chapter 6](#) covers Ajax more completely, but we'll introduce 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.

Dynamic data is a very powerful and important capability which should be exploited by most applications. With the ability to update your application's data set, you are enabling the user with the most current and most 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 in a new scene named `feedList`, which will request feed data for our default New York Times feed, then push `storyList` with the updated data. While the Ajax request is fairly simple, we need to process the RSS & ATOM data that we receive and that's a bit more complicated.

Start by adding the `feedList` scene, again using **palm-generate**:

```
palm-generate -t new_scene -p "name=feedList" .
```

Don't forget to update **`sources.json` with the new scene**. We just need a URL for the Ajax request and set up some callback functions. Add a new function to the `feedList` prototype:

```
// feedRequest - function called to setup and make a feed request
//
// Uses prototype's Ajax.Request and sets up callback routines:
//   onSuccess      feedRequestSuccess
//   onFailure      feedRequestFailure
//
FeedListAssistant.prototype.feedRequest = function(curFeed) {
    var request = new Ajax.Request(curFeed.url, {
        method: 'get',
        evalJSON: 'false',
        onSuccess: this.feedRequestSuccess.bind(this),
        onFailure: this.feedRequestFailure.bind(this)
    });
};
```

Then call it from within the `feedList` setup method.

```
FeedListAssistant.prototype.setup = function() {
    // Set feedsInitialized to false so that activate knows that it's
    // being launched or pushed from the background
    //
    this.feedsInitialized = false;

    // Start the Ajax Request
    //
    this.feedRequest(curFeedSource);
};
```

Because Ajax requests are asynchronous operations, with both success and error cases, you'll need to create callback functions for each of these cases. The handler for the error case simply needs logging the error; later we'll post an alert. Ajax requests return an HTTP

status message, which we convert to a readable format with Prototype's **Template** function and log the results.

```
//    feedRequestFailure
//
//    Callback routine from a failed AJAX feed request (feedRequest);
//    post a simple failure error message with the http status code.
//
FeedListAssistant.prototype.feedRequestFailure = function(transport) {
    //    Use the Prototype template object to generate a string
    //    from the return status.
    //
    var t = new Template("Status #{status} returned from newsfeed request.");
    var m = t.evaluate(transport);
    Mojo.Log.info(".....", "Invalid feed - http failure (" + 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 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 `feedList`. 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 `errorNone` means that the feed was processed successfully. We push the `storyList` scene with the processed feed in that case.

```
//    feedRequestSuccess
//
//    Callback routine from a successful AJAX feed request (feedRequest); use globals:
//    curFeedIndex    Index for the feed being updated
//    feedList        Holds the processed feeds, will have the current feed
//                    updated at exit; has old version of feed at entry
//
FeedListAssistant.prototype.feedRequestSuccess = function(transport) {
    var    feedError = errorNone;    //    Return variable for processing feed

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

    //    DEBUG - Work around
    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, identifying the current feed index and passing in the
    //    transport object holding the updated feed data
    //
    feedError = ProcessFeed(transport, curFeedIndex);

    //    If successful processFeed returns errorNone
    if (feedError == errorNone)    {
        Mojo.Controller.stageController.pushScene("storyList", curFeedIndex);
    } else    {
```

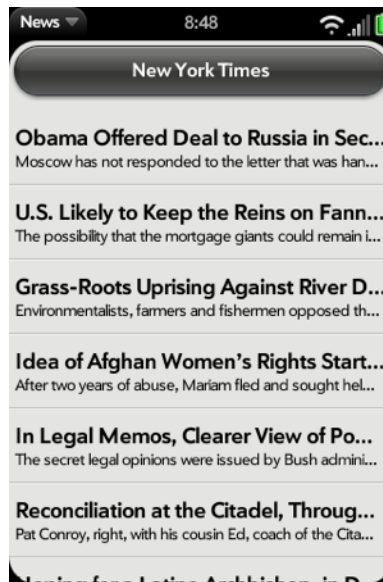
```
// There was a feed process error
if (feedError == errorUnsupportedFeedType) {
    Mojo.Log.info(".....", "Feed ", this.nameModel.value,
        " is not a supported type (#", errorUnsupportedFeedType, ").");
}
};
```

`ProcessFeed` is included in [Appendix A](#) 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 here. Just to summarize, `ProcessFeed` is passed an XML object and an index into `feedList` where it will put the processed feed. If there's no index argument then `ProcessFeed` will add the new feed to the end of the list.

For each of the supported formats, the title, text and URL is extracted for each of the stories and the `feedList` 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, then it will return with an `errorUnsupportedFeedType`.

When the application is launched, there's a new top level scene that briefly displays before receiving the feed data from the Ajax request, processing it then pushing the new `storyList` scene with a longer list of stories as shown in [Figure 3-9](#).

Figure 3-9. A StoryList scene with updated stories.



3.4.1.4. Back to the News: Adding a FeedList

A News reader that handles one news feed 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 add more to our `feedList` at the beginning of the Stage Controller with some popular news, sports and technology feeds.

```
// Push default feeds onto list; these will
// get overwritten by what's stored in the database but these will be used otherwise

feedList.push({title:"Huffington Post",
  url:"http://feeds.huffingtonpost.com/huffingtonpost/raw_feed",
  type:"atom", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"Google",
  url:"http://news.google.com/?output=atom", type:"atom", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"BBC News",
  url:"http://newsrss.bbc.co.uk/rss/newsonline_world_edition/
front_page/rss.xml", type:"rss",
  acFreq:0, numUnRead:0, stories:[]});
feedList.push({title:"New York Times",
  url:"http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml",
  type:"rss", acFreq:0, numUnRead:0, stories:[]});
feedList.push({title:"MSNBC",
  url:"http://rss.msnbc.msn.com/id/3032091/device/rss/rss.xml",
  type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"National Public Radio",
  url:"http://www.npr.org/rss/rss.php?id=1004", type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"Slashdot",
  url:"http://rss.slashdot.org/Slashdot/slashdot", type:"rdf", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"Engadget",
  url:"http://www.engadget.com/rss.xml", type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"The Daily Dish",
  url:"http://feeds.feedburner.com/andrewsullivan/rApM?format=xml",
  type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"Guardian UK",
  url:"http://feeds.guardian.co.uk/theguardian/rss", type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"Yahoo Sports",
  url:"http://sports.yahoo.com/top/rss.xml", type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"ESPN",
  url:"http://sports-ak.espn.go.com/espn/rss/news", type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
feedList.push({title:"Ars Technica",
  url:"http://feeds.arstechnica.com/arstechnica/BAaf", type:"rss", acFreq:0,
  numUnRead:0, stories:[]});
```

Next, we'll set up the `feedList` widget in both our scene's view file and the assistant. In the *feedList-scene.html* file, add a header and the widget declaration:

```

<div class='palm-header'>
  <div class='palm-header-center'>News</div>
</div>
<div class="palm-header-spacer"></div>
<div class="palm-list">
  <div id="feedListWgt" x-mojo-element="List"></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-tap-highlight="momentary">
  <div class="palm-row-wrapper">
    <div class="icon right"><div class="unreadCount">#{numUnread}</div></div>
    <div class="title truncating-text">#{title}</div>
  </div>
</div>

```

Complete the widget setup by calling `this.controller.setupWidget` and adding event listeners in the *feedList-assistant.js* file at the beginning of the Setup method:

```

FeedListAssistant.prototype.setup = function() {

  //   Setup the feed list
  //
  this.controller.setupWidget("feedListWgt",
    this.feedWgtAttr = {
      itemTemplate:"feedlist/feedRowTemplate",
      listTemplate:"feedlist/feedListTemplate",
      swipeToDelete:true,
      renderLimit: 40,
      reorderable:true
    },
    this.feedWgtModel = {items: feedList});

  //   Setup event handlers list selection, add feed, delete feed
  //   and reorder feed list
  //
  this.controller.listen('feedListWgt', Mojo.Event.listTap,
    this.showFeed.bindAsEventListener(this));
  this.controller.listen('feedListWgt', Mojo.Event.listDelete,
    this.listDeleteHandler.bindAsEventListener(this));
  this.controller.listen('feedListWgt', Mojo.Event.listReorder,
    this.listReorderHandler.bindAsEventListener(this));

  //   Set feedsInitialized to false so that activate knows that it's being
  //   launched or pushed from the background
  //
  this.feedsInitialized = false;

  //   Start the Ajax Request sequence
  //
  this.updateFeedList();
};

```

We have to adjust the Ajax Requests to make requests serially for each feed. We'll add a new function, which you can see called at the end of the setup method. The new function, `updateFeedList` will start the Ajax Requests at the first `feedList` entry.

```
//    updateFeedList (FeedListAssistant) - called to
//    cycle through the feeds either on activate or
//    after the interval timer has fired. This is split from feedRequest
//    so that the latter can be called on each feed. This is called once per
//    update cycle.
//

FeedListAssistant.prototype.updateFeedList = function() {
    // request fresh copies of all stories
    curFeedIndex = 0;
    curFeedSource = feedList[curFeedIndex];
    this.feedRequest(curFeedSource);
};
```

Then update the `feedRequestSuccess` method in two places. In the last version, after getting a successful Ajax request, we pushed the processed `feedList` to the `storyList` scene. This time after checking for `feedError == errorNone`, we update the `feedWgtModel.items` with the new feed data and call the `modelChanged` method to update the list widget with the new data.

We've also extended this function at the end to increment `curFeedIndex` and making another Ajax request if the `feedList` hasn't been completely updated.

```
//    feedRequestSuccess
//
//    Callback routine from a successful AJAX feed request (feedRequest); use globals:
//    curFeedIndex    Index for the feed being updated
//    feedList        Holds the processed feeds, will have the current
//                    feed updated at exit; has old version of feed at entry
//

FeedListAssistant.prototype.feedRequestSuccess = function(transport) {

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

    //    DEBUG - Work around
    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, identifying the current feed index and passing in
    //    the transport object holding the updated feed data
    //
    feedError = ProcessFeed(transport, curFeedIndex);

    //    If successful processFeed returns errorNone
    if (feedError == errorNone)    {

        //    Update the feedList model object with the updated feed data in feedList
        this.feedWgtModel.items = feedList;
        this.controller.modelChanged(this.feedWgtModel);

    } else    {
        //    There was a feed process error
        if (feedError == errorUnsupportedFeedType)    {
            Mojo.Log.info(".....", "Feed ", this.nameModel.value,
                " is not a supported feed type. (" + errorUnsupportedFeedType + ").");
        }
    }
}
```

```

//      Increment the index. If this is NOT the last feed then
//      update the feedsource and request to retrieve the next feed
//
curFeedIndex++;
if(curFeedIndex < feedList.length) {
    curFeedSource = feedList[curFeedIndex];
    this.feedRequest(curFeedSource);
} else {

    //      Otherwise, this update is done. Reset index to 0 for next update;
    //      if the timer is null, the set the interval to feedUpdateInterval
    //
    curFeedIndex = 0;
    this.feedsInitialized = true;
}
};

```

These changes will allow us to display a list of feeds that are updated serially, but before we are done, we need to have handlers for the delete and reorder events. We setup the event listeners back in the `feedList` assistant's setup method, but not the handlers.

```

//      listDeleteHandler - triggered by deleting a feed from
//      the list and updates the feedList to reflect the deletion
//
FeedListAssistant.prototype.listDeleteHandler = function(event) {

    var deleteIndex = event.index;
    feedList.splice(deleteIndex, 1);
    this.feedWgtModel.items = feedList;
};

//      listReorderHandler- triggered re-ordering feed list and
//      updates the feedList to reflect the changed order
//
FeedListAssistant.prototype.listReorderHandler = function(event) {

    var fromIndex = event.fromIndex;
    var toIndex = event.toIndex;
    feedList.splice(fromIndex, 1);
    feedList.splice(toIndex, 0, event.item);
    this.feedWgtModel.items = feedList;
};

```

In both cases, the framework handles the on screen changes, but you will need to reflect those changes in `feedList` itself. Add listeners to receive the delete and re-order events and you will receive the indices for the changes through the event object. You would then use these indices to make the corresponding changes in the `feedList`.

The `feedList` attributes includes the `reorderable` and `swipeToDelete` properties. A tap-and-hold on a list item will allow it to be moved 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 operation to be confirmed or cancelled. 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`.

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

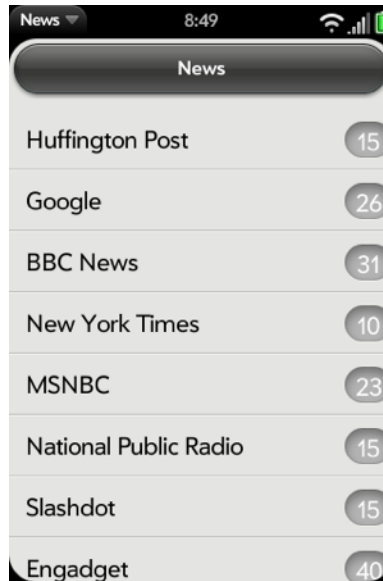
- If the `addItemLabel` property is specified, then an additional item is appended to the list. Tapping it will cause a `Mojo.Event.listAdd` event to be fired on the widget div.
- Deleted items which are unconfirmed have a `deleted` property set in the model. The name of this property can be specified 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 can be important for the app 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.
- If the `dragDatatype` property is specified, then items will be allowed to be dragged 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.

When you run the application now, it's starting to take the basic structure of the envisioned application with an initial scene that is a list of available feeds with a count of unread messages, which can be tapped 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.

Before finishing this change, style the `unreadCount` class in the **feedlistRowTemplate.html** with some CSS and add an background image. [Figure 3-10](#) shows the `feedList` scene with the style improvements.

```
.unreadCount {
  background: url(..images/unread-background.png) no-repeat ;
  width: 40px;
  height: 50px;
  position: relative;
  margin-top: 5px;
  top: 6px;
  padding-top: 3px;
  vertical-align: middle;
  text-align: center;
  color: white;
}
```

Figure 3-10. The feedList Scene with style improvements



Using Widgets in Lists

You can define list entries to include other widgets, including other lists. The List's model is an object which includes an array of items, and each item entry may have properties that are referenced by 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 that is an array of properties, at least one of which is the declared widget's models, then call `setupWidget()` once to setup up once for each widget declaration included in the template..

For example, to create a list where each list item or row has a text label and a Toggle Button, you would 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">
    <span>#{text}</span>
    <div name="listToggle" x-mojo-element="ToggleButton"></div>
  </div>
</div>
```

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

```
this.listModel = {items:[
  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.controller.setupWidget('myList', this.listAttr, this.listModel);
```

and then setup 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` must 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 will get 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 are forced to share the same model. Many widgets allow you to specify their `modelProperty` in their attributes to make it easier to use a shared model. For example, Toggle Button and List Selector both have a `modelProperty` attributes property.

The event handling is more complicated where the list items contain multiple items including widgets. To deal with this, List supports `Mojo.Event.listTap` and `Mojo.List.change` 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.

3.4.2. More About Lists

There are several major features included with Lists that aren't used with News lists, and there are two other list widget types as well: `GridList` and `FilterList`. `FilterList` will be used in [Chapter 5](#) to add a search list to News but the other features will be briefly touched on here.

3.4.2.1. 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 as described in [Table 3-3](#).

Table 3-3. ItemsCallback Arguments

Argument	Type	Description
<code>listWidget</code>	Object	The DOM node for the list widget requesting the items
<code>offset</code>	Integer	Index in the list of the first desired item model object (zero-based)
<code>limit</code>	Integer	Count of 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 arguments as shown in [Table 3-4](#).

Table 3-4. noticeUpdatedItems Arguments

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

3.4.2.2. Formatters & 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, this 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 then you should insert the property `{dividerLabel}` where you would want to have the label string inserted.

3.4.2.3. GridList

This Grid List presents items in a grid layout, with most of the optional attributes that you have in the base List widget. [Figure 3-11](#) shows an example of Grid List.

Figure 3-11. A GridList widget example.



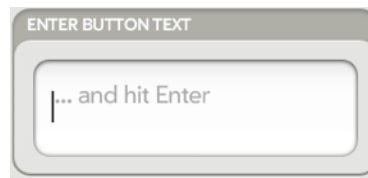
3.4.3. Text Fields

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 Treo's hallmark was a terrific thumb-able 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 3-12](#)), the base 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.

Figure 3-12. A Text Field widget example



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

- `PasswordField` to handle passwords or other confidential text input.
- `FilterField` to support type-down filters of an offscreen list or similar searchable data.
- `RichTextEdit` a multi-line text field that support 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. Auto capitalization and correction for common typing mistakes is performed on all fields unless explicitly disabled.

Smart Text Features

The Smart Text Engine (STE) refers to the automatic modification of user entered text in order to allow quicker text input. When typing on a small keyboard that is usually characteristic of a mobile device, a user is more likely to make certain spelling mistakes. Furthermore, because text input for things like SMS, notes, and contact info is often done in a hurry, a user is more likely to forgo typing punctuations, using capitalization, and/or use common slang abbreviations for words (e.g. "r" instead of "are", "u" instead of "you", etc.)

The STE performs auto-capitalization and auto-replacement. `STE-autocap` automatically asserts a Shift key state when it detects a punctuation character, followed by a space during text entry. Auto-replacement works by checking each word typed 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 apply any one of the special attributes in your Text Field div:

```
x-palm-disable-ste-all = "true";      // disables ALL Smart Text features
x-palm-disable-auto-replace = "true";  // disables just autoreplace
x-palm-disable-auto-cap = "true";     // disables just autocap
```

Or you can set the Text Field's attributes `textReplacement` property to false.

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 Advance Styling topics.

3.4.3.1.

3.4.3.1.1. Adding Text Fields to News

We only have one example of a text field in the News application: adding a news feed requires text fields to enter the feed URL and name. The Text Fields will be put below the `feedList` within the same scene and a Drawer widget will hide the Text Fields until triggered by the Add Feed action on the `feedList` 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 (`getOpenState` and `setOpenState`) available for manually opening & 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 divs:

```
<!-- Adding text field within a drawer and group box -->
<div id='feedDrawer' x-mojo-element="Drawer">
  <div class="palm-group">
    <div class="palm-group-title">

      <!-- Title and error status for invalid feeds -->
      <span id="add-feed-title" x-mojo-loc="">Add News Feed Source</span>

    </div>
```



```

<div class="palm-group unlabeled">
  <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 class="label">URL</div>
          <div id="newFeedURL" x-mojo-element="TextField" align="left"></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 class="label">Title</div>
          <div id="newFeedName" x-mojo-element="TextField" align="left"></div>
        </div>
      </div>
    </div>

  </div>
</div>

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

```

At the beginning of the file are some style classes to create a bordered group with a title 'Add News Feed Source'. The next style class `.palm-group unlabeled` and the following `.palm-list`, combine to create a bordered 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 title and label classes to complete the styling. The text field widgets are declared within all those layers of styling classes. A button widget is declared at the bottom to approve the feed entry and submit it for addition to the list.

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

```

// Setup Drawer for add Feed; closed to start
//
this.controller.setupWidget('feedDrawer', {property:'myOpen
Property'}, this.feedDrawerModel={myOpenProperty:false});

// Setup text field for the new feed's URL
//
this.controller.setupWidget(
  "newFeedURL",
  this.urlAttributes = {
    property: "value",
    hintText: "RSS or ATOM feed",
    focus: true,
    limitResize: true,
    textReplacement: false,
    enterSubmits: false
  },
  this.urlModel = {value : ""});

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

```

```

        property: "value",
        hintText: "Optional",
        limitResize: true,
        textReplacement: false,
        enterSubmits: false
    },
    this.nameModel = {value : ""});

//    Setup button and event handler
//
this.controller.setupWidget("okButton",
    this.attributes = {
    },
    this.model = {
        buttonLabel: "Add Feed",
        buttonClass: "addFeedButton",
        disabled: false
    });

this.controller.listen('okButton', Mojo.Event.tap, this.checkIt.
bindAsEventListener(this));

```

The first `setupWidget` call creates the Drawer, initially closed. The next `setupWidget` creates the URL field with some hint text, setting focus to the field and constraining the field from growing (extra text will scroll horizontally). The name field is setup the same way with the hint text indicating that the field is optional – we'll fill in the name if not entered from the contents of the feed.

The button is set up with an 'Add Feed' label and a custom class so we can adjust it's size in our CSS and setup a listener for the button tap to invoke the handler, `checkIt`.

```

.addFeedButton {
    width: 263px;
}

```

However, you still need a selector to open the Drawer. The List widget has a perfect feature, 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});

```

You will need to 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 and invokes the AddDialog
//  Assistant defined above.

```

```
//
FeedListAssistant.prototype.addNewFeed = function() {

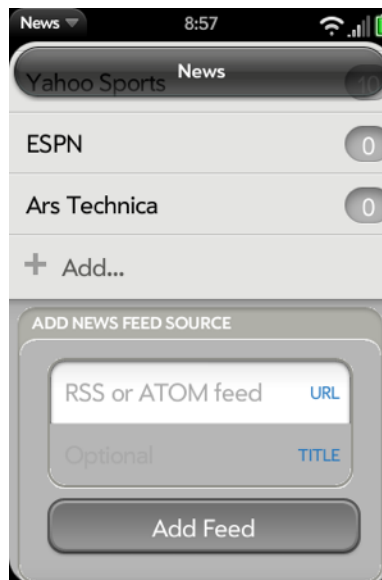
    this.feedDrawerModel.myOpenProperty = !this.feedDrawerModel.myOpenProperty;
    this.controller.modelChanged(this.feedDrawerModel, this);
};
```

When the Add... item is tapped 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 which generates a tap event on the button and the `checkIt` handler called to process the feed.

Since we're demonstrating the Text Field widget, we haven't included all the code for `this.checkIt`, but you can refer to the [Appendix A](#) where the News application source is reproduced in its entirety. 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 `feedList`. The Drawer is closed at the end if the feed is added successfully.

[Figure 3-13](#) shows the new `feedList` scene with the Drawer in the open position and the text fields.

Figure 3-13. A Feedlist scene with a text field.

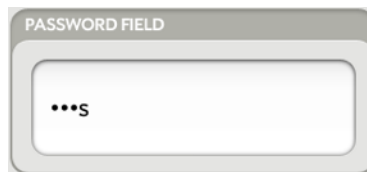


3.4.3.2. Password Field

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, or "." 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 3-14](#) shows an example of a Password Field.

Figure 3-14. A Password Field widget example.



Palm webOS Editing

The Palm Pre 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 thumb-able keyboard, all text fields support trackball mode cursoring, smart deletion and text selection.

Trackball mode (hold the Alt or Orange key while swiping) enables you to use swipes to move the cursor across the text to the location you need, while text selection (hold the Shift key while swiping), will highlight selected text for deletion, replacement or cut/copy/paste operation.

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

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

3.4.3.3. 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.

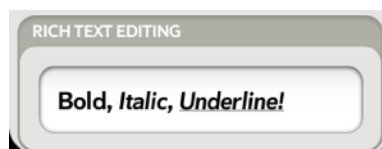
Filter Field is hidden until displayed by the framework in response to the user entering text when there isn't focus on any text field. In other words, the filter field by gets focus for any text input on scene where it is present and another text field hasn't been explicitly been given focus.

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 calling the filter function until the field is closed.

3.4.3.4. Rich Text Edit

There is a simple Rich Text Edit widget (see [Figure 3-15](#)) that behaves exactly like a multiline text field, but in addition supports applying Bold, Italic and Underline styles to arbitrary runs of text within the field.

Figure 3-15. A Rich Text Edit widget.



To enable the styling, enable the `RichTextEditMenu` property in the AppMenu (see [Menus](#) for information on the App Menu). This enables the Bold, Italic and Underline menu options, which when selected will apply that style to the current text selection if it is in a RichTextEdit widget.

3.5. Events

The 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.

3.5.1. 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 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 event handler through `Mojo.Event.make` and `Mojo.Event.send`.

3.5.2. Listening

When an event occurs, all code that is subscribed to handle the event is notified of the event occurrence. 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 mostly 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 multi-stage 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.

```
<button id='thanksButton'>Thank you</button>
```

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));
```

Binding *this* as Event listener

You will typically need to use `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 `bindAsEventListener` to make sure that the *this* keyword will point to the same scene assistant instance that registered the handler.

3.5.3. 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 Mojo method to stop listening if you used the Mojo method 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 also, 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.

3.5.4. 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. In Appendix B, all the specific events propagated by each widget are enumerated in options tables accompanying each widget's description.

3.6. Summary

Widgets are signature components provided by Mojo enabling your applications with 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 described 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 generally use a widget within your application.

With these basic widgets, you can write some simple applications. But you will need Menus and Dialogs, which we 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 & Menus

Familiar components in every UI framework, Dialogs and Menus are used by almost all applications. Mojo's Dialog and Menu widgets provide the expected features but have some unique additions. Dialogs can be built as child scenes, enabling you to include any web content in a dialog, and menus can be customized by scene and presented in both conventional dropdown presentations or as floating elements.

Dialogs and Menus are both fundamental widgets, though more complex than the basic widgets covered in [Chapter 3](#), and each 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. Mojo provides a model for propagating commands through the app, stage and scene controllers called the *Commander Chain*, which is described in detail near the end of this chapter.

As in [Chapter 3](#), these widgets will be presented by adding them into the News application accompanied by a general description and some screenshots.

4.1. Dialogs

You can use a dialog to create a modal view for almost any purpose. A `Custom Dialog` is a conventional dialog, but requires its own scene, which means declaring an assistant and scene view. The Dialog scene is pushed as a child scene to the scene assistant that invokes it, 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`.

4.1.1. Error Dialog

Post error messages in a modal dialog 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](#). It is called as a function and the only argument is a string that's posted for the user under a dialog entitled "Error" and a single OK button.

Figure 4-1. An Error Dialog.



4.1.1.1. Back to the News: Add an Error Dialog

You can post an Error Dialog in response to any Ajax failures after a sync feed request, by adding a call to `errorDialog` at the end of the `feedRequestFailure` function in *feedlist-assistant.html*:

```
// feedRequestFailure
//
// Callback routine from a failed AJAX feed request (feedRequest);
// post a simple failure error message with the http status code.
//
FeedListAssistant.prototype.feedRequestFailure = function(transport) {

    // Use the Prototype template object to generate a string from the return status.
    //
    var t = new Template("Status #{status} returned from newsfeed request.");
    var m = t.evaluate(transport);

    // Log error & post dialog
    //
    Mojo.Log.info("Invalid feed - http failure (" + m);
    Mojo.Controller.errorDialog("Invalid feed - http failure (" + m);
};
```

This dialog displays the Ajax error code appended to a static error message. In News, you can put a similar Error Dialog in the `ProcessFeed` function (also in *feedlist-assistant.html*) when encountering an unsupported feed format.

Logging Methods

Mojo includes logging methods to give you an efficient way to generate console output while not 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 *appinfo.json*. To allow all log levels, you would 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. For example:

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

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. For example:

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

Would output:

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

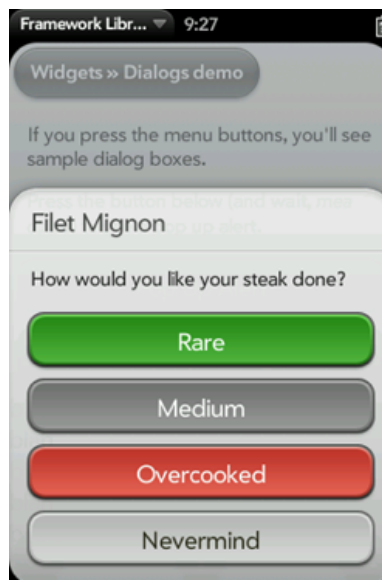
You can use `%s`, `%d`, `%f`, `%i`, `%o` and `%j`. The first four all produce the same result; coercing the appropriate parameter to a string for logging. `%o` converts the parameter to a string using Prototype's `Object.inspect()`, while `%j` converts it using `Object.toJSON()`.

In desktop browsers, these messages are frequently adorned based on the level. For system messages, text and some delimiters are added to make the message stand out.

4.1.2. 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 non-error message for the user or present options that can be selected in the form of button selections. An example of this dialog is shown in [Figure 4-2](#).

Figure 4-2. An Alert Dialog.



4.1.3. Custom Dialogs

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

4.1.3.1. Back to the News: Add Feed Dialog

In the last chapter, we added a drawer in the feedlist assistant to support the Add Feed feature. It would be better to put this type of feature in a dialog; we will create a 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 and invokes the AddDialog
// Assistant defined above.
//
FeedListAssistant.prototype.addNewFeed = function() {

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

The arguments specify the dialog template and a reference to the assistant that manages the dialog. We create a new instance of the `AddDialogAssistant`, passing a reference to the feedlist assistant, and pass that in along with a reference to our `addFeed-dialog` template. The dialog template is simply an HTML template, but you should make use of some of the standard dialog styles such as `.palm-dialog-box`, `.palm-dialog-title` and `.palm-dialog-content` to format and style your dialogs 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 the feedlist view file to a new file, *views/feedList/addFeed-dialog.html*.

```
<div>
  <span id="add-feed-title" class="palm-dialog-title">Add News Feed Source</span>

  <div class="palm-group unlabeled">
    <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 class="label">URL</div>
            <div id="newFeedURL" x-mojo-element="TextField" align="left"></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 class="label">Title</div>
            <div id="newFeedName" x-mojo-element="TextField" align="left"></div>
          </div>
        </div>
      </div>
    </div>
  </div>

  <div x-mojo-element="Button" id="okButton"></div>
</div>
```

The only changes from the HTML used previously are the removal of the `Drawer` and the addition of the enclosing `.palm-group` style divs.

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 setup widgets, push scenes and generally do anything that you can do within a scene assistant. There is one major difference: the dialog assistant isn't extended with sceneController methods; instead the dialog assistant must use the calling scene assistant's 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.
// Invoked by the FeedListAssistant when the "Add..." list item is selected
// on the feed list.
//
// The dialog displays two text fields (URL and Name) and an OK button.
// Either the user enters a feed URL and a name, followed by OK or a
// back swipe to close. If OK, the feed header is checked through an Ajax
// request and if valid, the feed updated and dialog closed. If an error,
// the posted in place of title and dialog remains open.
// Swipe back cancels and returns back to the FeedListAssistant.
//

function AddDialogAssistant(sceneAssistant) {

    this.sceneAssistant = sceneAssistant;
}

AddDialogAssistant.prototype.setup = function(widget) {

    this.widget = widget;

    // Setup text field for the new feed's URL
    //
    this.sceneAssistant.controller.setupWidget("newFeedURL",
        this.urlAttributes = {
            property: "value",
            hintText: "RSS or ATOM feed",
            focus: true,
            limitResize: true,
            textReplacement: false,
            enterSubmits: false
        },
        this.urlModel = {value : ""});

    // Setup text field for the new feed's name
    //
    this.sceneAssistant.controller.setupWidget("newFeedName",
        this.nameAttributes = {
            property: "value",
            hintText: "Optional",
            limitResize: true,
            textReplacement: false,
            enterSubmits: false
        },
        this.nameModel = {value : ""});

    // Setup button and event handler
    //
    this.sceneAssistant.controller.setupWidget("okButton",
```



```

        this.attributes = {},
        this.model = {
            buttonLabel: "Add Feed",
            buttonClass: "addFeedButton",
            disabled: false
        });

        Mojo.Event.listen($('okButton'), Mojo.Event.tap,
            this.checkIt.bindAsEventListener(this));
    };

    // -----
    // Add Feed Functions
    //
    //      checkIt - called when Add feed OK button is clicked
    //
    AddDialogAssistant.prototype.checkIt = function() {

        //      Check for 'http://' on front or other legal prefix
        //      assume any string of 1 to 5 alpha characters followed
        //      by ':' is legal, otherwise prepend "http://"
        //
        var url = this.urlModel.value;

        if (/^[a-z]{1,5}:/ .test(url) === false) {
            url = url.replace(/^\/{1,2}/, "");
            url = "http://" + url;
        }

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

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

    //
    //      checkOK - called when the Ajax request is successful.
    //
    AddDialogAssistant.prototype.checkOK = function(transport) {

        //      DEBUG - log the result
        //
        var t = new Template("Status #{status} returned from newsfeed request.");
        var success = "Feed Request Success: " + t.evaluate(transport);

        Mojo.Log.info(success);

        //      Work around when XML comes back in text 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 = errorNone;

        //      Push the entered feed onto feedlist and call processFeed to evaluate it.
        //      This is mainly to record the user's TITLE entry before processing
        //
        feedList.push({title: this.nameModel.value, url: this.urlModel.value, type: "",
            acFreq: 0, numUnRead: 0, stories: []});
        feedError = ProcessFeed(transport);
    }

```

```

//      If successful processFeed returns errorNone
//
if (feedError === errorNone) {
    this.sceneAssistant.feedWgtModel.items = feedList;
    this.sceneAssistant.controller.modelChanged
(this.sceneAssistant.feedWgtModel);
    this.widget.mojo.close();
}
else {
    feedList.pop();
    if (feedError == errorUnsupportedFeedType) {
        Mojo.Log.info("Feed ", this.urlModel.value,
            " isn't a recognized feed type (#", errorUnsupportedFeedType, ")");
        $("add-feed-title").innerHTML =
            "Invalid Feed - not a supported feed type";
    }
}
};

AddDialogAssistant.prototype.checkFailure = function(transport) {
    //      Use the Prototype template object to generate a string from the return
    //
    var t = new Template($L("#{status}"));
    var m = t.evaluate(transport);

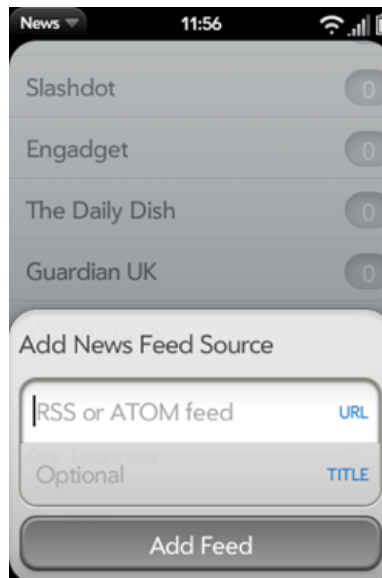
    //      Log error and put message in status
    //
    Mojo.Log.info("Invalid feed (Status ", m, " returned).");
    $("add-feed-title").innerHTML = "Invalid feed - http failure (" + m + ")";
};

```

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

- **Cancel:** remove the Cancel button and the `cancelIt` method, as the back gesture will be used to cancel instead of a button.
- **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 those methods.
- **Close:** add a `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 will close the dialog, so there is no need for a cancel button. You may want to keep a Cancel button in most dialogs for 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; by default `preventCancel` is set to `false`. [Figure 4-3](#) shows the results of these changes and the Add Feed dialog.

Figure 4-3. The Add Feed dialog.

4.2. Menus

Mojo supports four types of menu widgets. Each is fairly different in design, 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 into your application. Briefly:

- **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. Can be used as a display header, on action buttons, to popup a submenu or toggle a setting.
- **Command Menu:** used to set menus or more typically buttons across the bottom of the screen for actions, to popup a submenu or toggle a setting.
- **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 Menu, View Menu and Command Menu are technically very similar: they use a single model definition with a menu items array, and are instantiated 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 after the section on Menu Widgets.

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 not available to applications.

4.2.1. 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 position to be determined within the HTML. They are in the DOM so you can use CSS to style them, but the framework determines their position according to pre-defined 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 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 have 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 either specify one of the framework's icon using the `icon` property, or an application supplied icon image, found at `iconPath`.

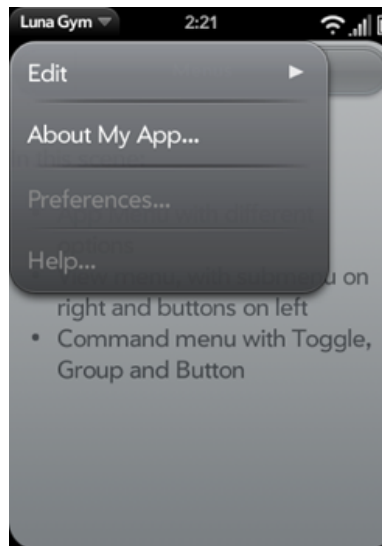
Each item includes a command value, which is propagated through the *commander chain* when this item is selected. This is a rather significant topic, which we'll touch on briefly here, but you should review the Commander Chain section later in this chapter to get a full description.

4.2.1.1. 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 the Application Menu.

Figure 4-4. An Application Menu widget.



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.

4.2.1.2. Back to the News: Adding an Application Menu

Now let's add an Application Menu to News, which will initially point to 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.

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

  //   Setup Application Menu with an About entry
  //

  newsMenuAttr = {omitDefaultItems: true};

  newsMenuModel = {
    visible: true,
    items: [
      {label: "About News...", command: 'do-aboutNews'},
      Mojo.Menu.editItem,
      Mojo.Menu.prefsItem,
      Mojo.Menu.helpItem
    ]
  };

  this.controller.pushScene("feedList");
}
```

```

};

//    handleCommand - Setup handlers for menus:
//
StageAssistant.prototype.handleCommand = function(event) {
    var currentScene = this.controller.activeScene();
    if(event.type == Mojo.Event.command) {
        switch(event.command) {

            case 'do-aboutNews':
                currentScene.showAlertDialog({
                    onChoose: function(value) {},
                    title: "News - v1.0",
                    message: "Copyright 2008-2009, Palm Inc.",
                    choices:[
                        {label: "OK", value:""}
                    ]
                });
                break;

        }
    }
};

```

These menu properties are unique to the Application Menu:

- `richTextEditItems` can be set to true when you include a `RichTextEdit` widget in your scene, and it will add the styling items, Bold, Italic and Underline to the Edit menu.
- `omitDefaultItems` must be set to true when you want to enable Preferences or Help, or if you chose to disable Edit.

If you choose `omitDefaultItems` then you must put the items back with your own definitions. If you want to override some but not all items, you can use system constants to replace the items that aren't changing. In the `newsMenuModel`, the default items `Mojo.Menu.editItem`, `Mojo.Menu.prefsItem` and `Mojo.Menu.helpItem`.

The `newsMenuAttr` declares that this menu will override the default items, and the `newsMenuModel` 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 as an About Box.

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.

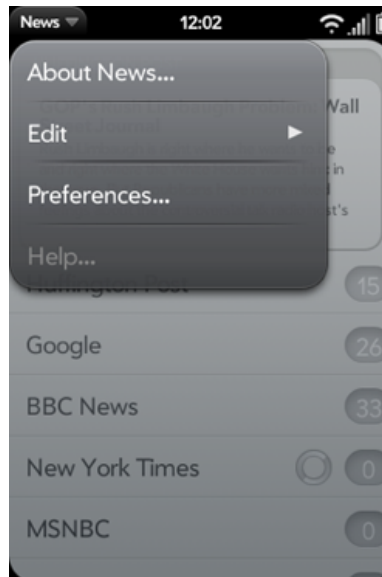
All this work has been done in *stage-assistant.js*, but the Application Menu is actually displayed within the scenes. To configure and display the menu widget, each scene assistant's `setup` method will include this `setupWidget()` call:

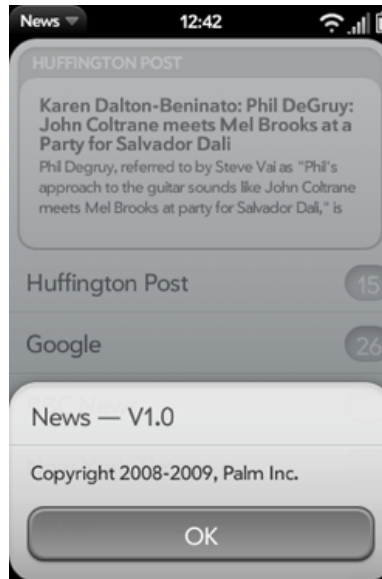
```
// Setup Application Menu  
this.controller.setupWidget(Mojo.Menu.appMenu, newsMenuAttr, newsMenuModel);
```

You can override the application-wide behavior in a specific scene by defining scene-specific app 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 both the Application Menu and the resulting About Box.

Figure 4-5. The News Application menu and About Box.





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.

4.2.1.3. Back to the News: Override Default Preferences

In the last 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 the preference 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="header-icon news"></div>
  <div class="header-text">
    <span id="newsPrefsTitle" class="feedTitleSpan">News</span>
  </div>
</div>

<div class="palm-list">
  <div class="palm-row">
    <div id="feedCheckIntervalList" x-mojo-element="ListSelector"></div>
  </div>
</div>
```

The header is one of the framework style classes, `.palm-page-header`, the standard for a preferences scene. You'll also note the `.header-icon` style and the `news` style

addition enabling us to add some CSS to specify a small news icon in the header. The icon must be added to the *images* directory at the News application's root level.

```
.palm-page-header .header-icon.news {
  background: url(../images/header-icon-news.png) no-repeat;
  font-weight: bold;
}
```

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 setup 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.

```
function PreferencesAssistant() {
}

PreferencesAssistant.prototype.setup = function() {

  //      Setup list selector for feed interval
  //
  this.controller.setupWidget('feedCheckIntervalList',
    { label: "Interval",
      choices: [
        {label: "1 Minute",      value: 60000},
        {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 : feedUpdateInterval
    });

  this.controller.listen('feedCheckIntervalList', Mojo.Event.propertyChange,
    this.feedIntervalHandler.bindAsEventListener(this));
};

PreferencesAssistant.prototype.feedIntervalHandler = function(event) {
  feedUpdateInterval = this.feedIntervalModel.value;
};
```

The *feedUpdateInterval* is used to set the timer for the updates, but refer to the full News application in [Appendix A](#) to see how we handled the update in the *feedlist-assistant.js*.

With the Preferences scene coded, we can hook it up by returning to the stage assistant and changing the *newsMenuModel* to override the default Preferences command:

```
//
//      Setup Application Menu with preferences entry
//

newsMenuAttr = {omitDefaultItems: true};

newsMenuModel = {
  visible: true,
  items: [
    {label: "About News...", command: 'do-aboutNews'},
```

```

        Mojo.Menu.editItem,
        { label: "Preferences...", command: 'do-newsPrefs' },
        Mojo.Menu.helpItem
    ]
};

this.controller.pushScene("feedList");
};

```

And adding a handler for `do-newsPrefs` in the `handleCommand` method to push the preferences scene:

```

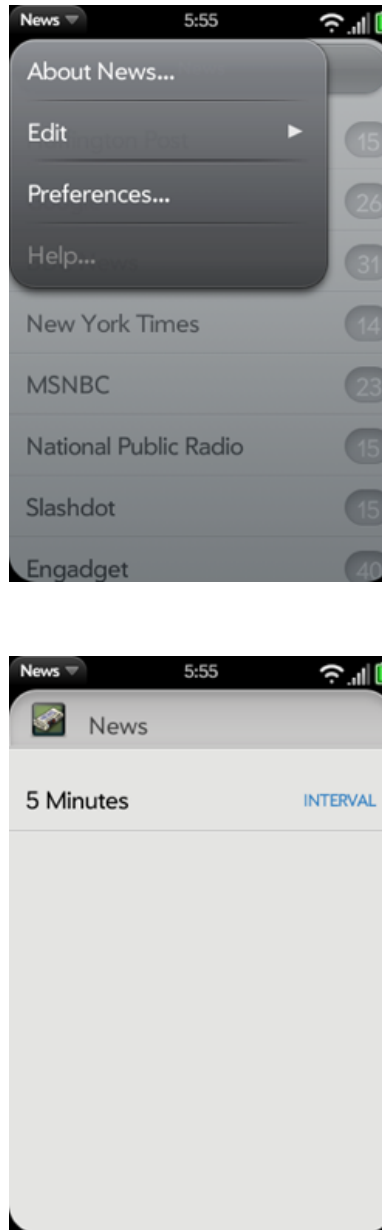
//    handleCommand - Setup handlers for menus:
//
StageAssistant.prototype.handleCommand = function(event) {
    var currentScene = Mojo.Controller.stageController.activeScene();
    if(event.type == Mojo.Event.command) {
        switch(event.command) {

            case 'do-aboutNews':
                currentScene.showAlertDialog({
                    onChoose: function(value) {},
                    title: "News - v1.0",
                    message: "Copyright 2008-2009, Palm Inc.",
                    choices:[
                        {label:"OK", value:""}
                    ]
                });
                break;
            case 'do-newsPrefs':
                Mojo.Controller.stageController.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.



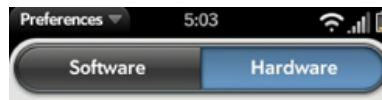
You should also notice that we didn't have to modify any of the scene assistants, yet the Preferences option is available in every scene. This approach makes it simple to consolidate common Application Menu handling throughout your application.

4.2.1.4. 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 from within the items property width, and the framework adjusts the space between the buttons automatically, as shown

in [Figure 4-7](#). Use dividers, or empty list items to influence the spacing to get specific layouts.

Figure 4-7. A View menu with buttons.



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](#).

4.2.1.5. 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 a next feed and a previous feed menu button, and methods to push the new scene for either the next feed or previous feed when selected.

First, let's add the view menu. Part way through this next code sample, the `feedMenuModel` is setup with three menu items:

- `feedMenuPrev` – a local variable set to either the Previous menu item or an empty item if the `selectedFeedIndex` points to the first feed in the `feedList`
- `feedMenuNext` – a local variable set to either the Next menu item or an empty item if the `selectedFeedIndex` points to the last feed in the `feedList`
- A literal object that will be used to display the `selectedFeed` 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, then any extra space is placed in 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 visual style and format.

Menu Icons

The Mojo framework includes a number of default icons that you can use as icons in your View and Command menu buttons, or you can provide your own. Look at the standard styles prefixed with `.palm-menu-icon` which can be referenced 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.

There are some guidelines if you're designing your own icons:

- Use PNG-24, which supports 8-bit alpha transparency.
- Menu icons are currently two frames in a 32x64 PNG, with the top frame as the normal state, and the bottom frame as the pressed state.
- Each icon is approximately 24x24 pixels within the 32x32 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 - Displays the feed's stories in
// a list, user taps display the
// selected story in the storyView scene.
//
// Arguments:
// selectedFeed      Feed to be displayed

function StoryListAssistant(selectedFeedIndex) {
    this.feed = feedList[selectedFeedIndex];
    this.feedIndex = selectedFeedIndex;
}

StoryListAssistant.prototype.setup = function() {

    // Setup scene header with feed title and next/previous feed buttons
    // If first feed, suppress Previous menu; if last feed, suppress Next menu
    //

    var feedMenuPrev = {};
    var feedMenuNext = {};

    if (this.feedIndex > 0) {
        feedMenuPrev = {
            icon: 'back',
            command: 'do-feedPrevious'
        };
    } else {
        feedMenuPrev = {icon: "", command: '', label: " "};
    }

    if (this.feedIndex < feedList.length-1) {
        feedMenuNext = {
            iconPath: 'images/menu-icon-forward.png',
            command: 'do-feedNext'
        }
    }
}
```

```

    };
    } else {
        feedMenuNext = {icon: "", command: '', label: " "};
    }

    // Define the model with next and previous menu items, and a title
    // area for the feed title where the width is set to creating a
    // style header
    this.feedMenuModel =
    {
        visible: true,
        items: [{
            items: [
                feedMenuPrev,
                { label: this.feed.title, width: 210 },
                feedMenuNext
            ]
        }]
    };

    // Setup the View Menu
    this.controller.setupWidget(Mojo.Menu.viewMenu,
        { spacerHeight: 0, menuClass:'no-fade' },
        this.feedMenuModel);

    // Setup Application Menu
    this.controller.setupWidget(Mojo.Menu.appMenu, newsMenuAttr, newsMenuModel);

    // Setup story list with standard news list
    templates, and listener for story taps
    //
    this.controller.setupWidget("storyListWgt",
        this.storyAttr = {
            itemTemplate: "storyList/storyRowTemplate",
            listTemplate: "storyList/storyListTemplate",
            swipeToDelete: false,
            renderLimit: 40,
            reorderable: false
        },
        this.storyModel = {
            items: this.feed.stories
        }
    );

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

```

Continuing on with our example, we'll add the `handleCommand` method after the `activate` and `readStory` methods, which are unchanged.

```

StoryListAssistant.prototype.activate = function() {
    // Unread count may have changed
    this.controller.modelChanged(this.storyModel);
};

// readStory - handler when user taps on displayed story,
// push story to the storyView scene
StoryListAssistant.prototype.readStory = function(event) {

    Mojo.Controller.stageController.pushScene("storyView", this.feed, event.index);
};

// -----
// Setup handlers for command menu
StoryListAssistant.prototype.handleCommand = function(event) {

```

```

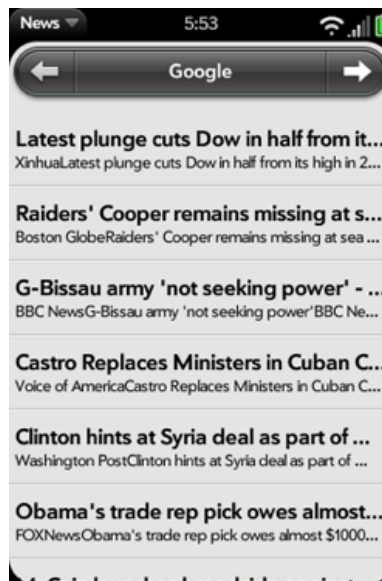
if(event.type == Mojo.Event.command) {
    switch(event.command) {
        case 'do-feedNext':
            Mojo.Controller.stageController.swapScene("storyList", this.feedIndex+1);
            break;
        case 'do-feedPrevious':
            Mojo.Controller.stageController.swapScene("storyList", this.feedIndex-1);
            break;
    }
}
};

```

The `handleCommand` method is called for a next and previous command, and invokes a `swapScene()` method to push the next scene. We mentioned in [Chapter 2](#) that `swapScene()` is similar to `pushScene()`, but rather than leaving the old scene on the scene stack, it pops it as part of the operation. It also has a different transition than `pushScene()` or `popScene()`, which might be an issue for you; if so you can always push and pop the scenes with individual calls.

With these changes, the News app's `storyList` scene is shown in [Figure 4-8](#).

Figure 4-8. The News View menu and `storyList` scene.



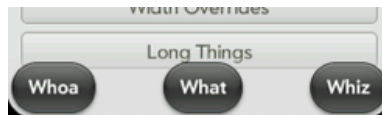
4.2.1.6. Command Menu

The Command Menu items are presented at the bottom of the screen, but similar to the View Menu in most other ways. Items will include variable sized buttons that can be combined into groups, and in a horizontal layout from left to right. You can override positioning by including dividers to force an item to the right or the middle of the screen, or by including an items entry with the `disable` property set to `true`. Typically you would

use the Command menu when using actionable buttons, buttons with dynamic behavior, or attaching a Submenu to a button to give further options.

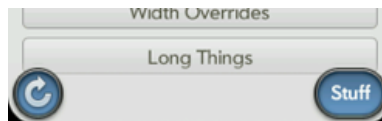
As with the View Menu, button widths can be adjusted from within the items property width, and the framework adjusts the space between the buttons automatically as shown in [Figure 4-9](#).

Figure 4-9. A Command menu with buttons.



[Figure 4-10](#) shows that you can define toggle buttons or include buttons with other dynamic behavior.

Figure 4-10. A Command menu with toggles.



If you'd like to group several items together, you would 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.



4.2.1.7. Back to the News: Adding Command Menus

We've been using HTML buttons within the Story View to go back and forth between stories within a feed, but here we'll replace them with Command Menus. It's really straightforward now that we've covered the basics with the App 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 HTML 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 sub-items are combined into an expanding item. With View and Command Menus the button groups are presented as an integrated view element.

```
// StoryViewAssistant(storyFeed, storyIndex)
//
// 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).

function StoryViewAssistant(storyFeed, storyIndex) {

    // Save the passed arguments for use in the scene.
    //
    this.storyFeed = storyFeed;
    this.storyIndex = storyIndex;
}

StoryViewAssistant.prototype.setup = function() {

    // Setup command menu for next and previous story as a menu group.
    // If first story, suppress Previous menu; if last story, suppress Next menu

    this.storyMenuAttr = {items: [{visible: false}, {items: []}, {visible: false}]};

    if (this.storyIndex > 0) {
        this.storyMenuAttr.items[1].items.push({icon:
"back", command: 'do-viewPrevious'});
    } else {
        this.storyMenuAttr.items[1].items.push({icon: "",
command: '', label: " "});
    }

    if (this.storyIndex < this.storyFeed.stories.length-1) {
        this.storyMenuAttr.items[1].items.push({icon: "forward",
command: 'do-viewNext'});
    } else {
        this.storyMenuAttr.items[1].items.push({icon: "",
command: '', label: " "});
    }

    this.controller.setupWidget(Mojo.Menu.commandMenu, undefined, this.storyMenuAttr);

    // Setup Application Menu
    this.controller.setupWidget(Mojo.Menu.appMenu, newsMenuAttr, newsMenuModel);
};

StoryViewAssistant.prototype.activate = function(event) {

    // Update story title in header and summary

    this.controller.update$("storyViewTitle"),
```

```

    this.storyFeed.stories[this.storyIndex].title);

    this.controller.update($("storyViewSummary"),
    this.storyFeed.stories[this.storyIndex].text);

    // Update unreadStyle string and unreadCount in case it's changed

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

```

The activate method is unchanged, but we replace the button handlers with command handlers as shown next.

```

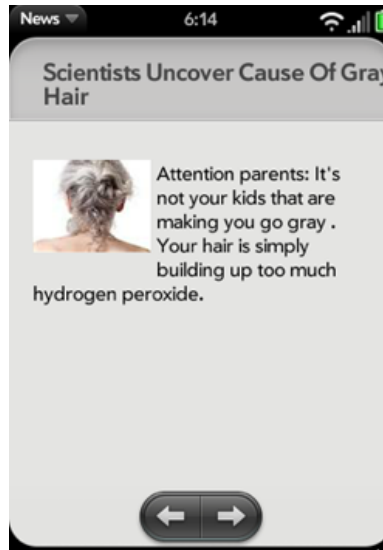
// Handlers to go to next and previous stories

StoryViewAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case 'do-viewNext':
                Mojo.Controller.stageController.swapScene("storyView",
                this.storyFeed, this.storyIndex+1);
                break;
            case 'do-viewPrevious':
                Mojo.Controller.stageController.swapScene("storyView", this.storyFeed,
                this.storyIndex-1);
                break;
        }
    }
};

```

That's it. Run the application; tap a feed and then a story to see the results (as shown in [Figure 4-12](#)).

Figure 4-12. A News Command menu and storyView scene.



4.2.2. Submenus

Popup submenus can be used to offer a transient textual list of choices to the user, typically off of another menu type, or from a DOM element in the scene. It accepts 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 command property of the chosen item as an argument. If the user taps outside the popup menu, it's still dismissed, and the `onChoose` function is called with **undefined** instead.

4.2.2.1. Back to the News: Adding a Submenu

We will use a submenu to present options when the unread count is tapped on the News feedList. For each feed you can choose between Mark Read, Mark Unread or Edit Feed. The first two options mark all the stories 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 modify the `showFeed()` method of *feedlist-assistant.js* to detect taps on the `unreadCount` icon on the right side of the list entry. If it's a tap anywhere else, then 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.

```
// showFeed - triggered by tapping a feed in the feedList.
// 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.className;
    if (target !== "unreadCount") {
        this.clearTimers();
        Mojo.Controller.stageController.pushScene("storyList", event.index);
    }
    else {
        this.popupIndex = event.index;
        this.controller.popupSubmenu({
            onChoose: this.popupHandler,
            placeNear: event.target,
            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. It handles the actions for marking all stories in the selected feed as unread or read, updates the feed's `numUnRead` and calls `modelChanged` to update the scene's displayed view. Once the actions are taken, the handler returns with the framework cleaning up the display by removing the Submenu and returning the feedlist scene to the full foreground view.

```
// popupHandler - choose function for feedPopup
//
FeedListAssistant.prototype.popupHandler = function(command) {

    var popupFeed=feedList[this.popupIndex];
    switch(command) {
        case 'feed-unread':
            for (var i=0; i<popupFeed.stories.length; i++) {
                popupFeed.stories[i].unreadStyle = unreadStoryFormatting;
            }
            popupFeed.numUnRead = popupFeed.stories.length;
            this.controller.modelChanged(this.feedWgtModel);
            break;

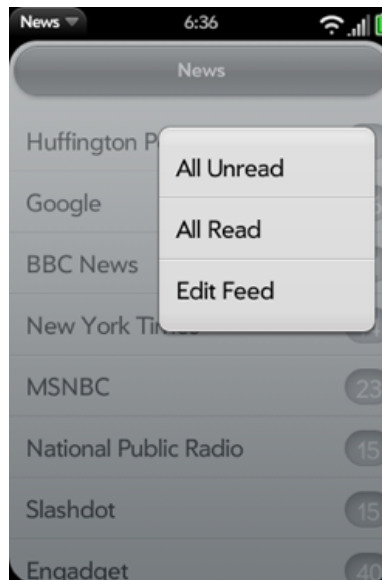
        case 'feed-read':
            for (var j=0; j<popupFeed.stories.length; j++) {
                popupFeed.stories[j].unreadStyle = "";
            }
            popupFeed.numUnRead = 0;
            this.controller.modelChanged(this.feedWgtModel);
            break;

        case 'feed-edit':
            this.controller.showDialog({
                template: 'feedList/addFeed-dialog',
                assistant: new AddDialogAssistant(this, this.popupIndex)
            });
            break;
    }
};
```

For the last option, the `AddDialogAssistant` is re-used. 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 A](#) to see where the changes were made.

With these changes, [Figure 4-13](#) shows a Submenu within the feedlist scene.

Figure 4-13. A News Submenu in a feedList scene.



4.3. Commander Chain

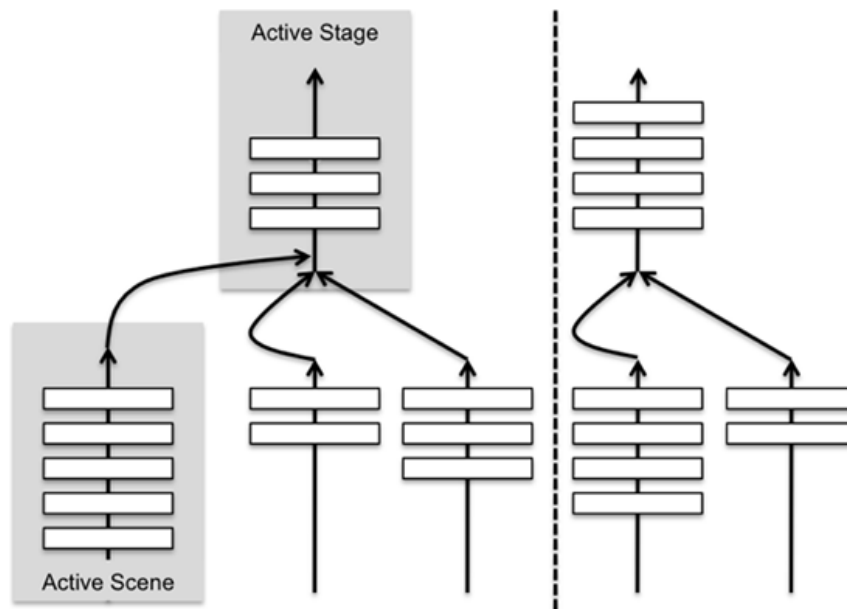
Mojo provides a model for propagating commands through the app, stage and scene controllers called the *Commander Chain*. The chain is an array of handlers, ordered like a stack. The handlers, or commanders, are put onto the chain in the order that they register themselves and commands are propagated 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 scene controller, and then within each stage there is a chain for each scene controller. Commands are propagated starting with the first commander registered in the active scene controller's chain. After all commanders in the scene have been called, then propagation continues with the first 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.

Figure 4-14. A Commander Chain propagation.



At any time, any commander can stop propagation by calling `Event.stopPropagation`. For example, a scene puts up a modal dialog, so it's 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 three types of events that propagate through the chain:

- `Mojo.Event.back`; this event indicates a back gesture.

- `Mojo.Event.command`; used for all menu commands.
- `Mojo.Event.commandEnable`; used to enable a menu item dynamically.

Command and Command Enable events are both discussed under the Menu Widgets section. 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 `stopPropagation` 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 was covered in the Application Menu section using the News application.

4.4. Summary

Dialogs and Menus round out the basic widgets that most applications require, and with what you've learned in [Chapter 2](#), [Chapter 3](#) and [Chapter 4](#) 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 in the last two chapters 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 use cases, but the widgets are just as simple to work with as the widgets already discussed in [Chapter 3](#) and [Chapter 4](#).

As with the two preceding chapters, each group will be reviewed in a summary form, and then a specific case will be used as an example with the News application. Where the widget isn't used in an example, we'll include a description and references to where you can find more information.

5.1. Indicators

Indicators are used to show that activity is taking place even if it's not visible, and in some cases, to show some measure of the progress that the activity is making. 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 which 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 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 to give the user a bounded sense of duration when possible.

5.1.1. 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 x 128 pixels, and the small Spinner is 32 x 32. These sizes are optimized for the Palm Pre screen and may vary on other devices but the spatial and visual characteristics will be maintained on other devices.

5.1.1.1. Back to the News: Add 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 `feedList` 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 a sidebar 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 that [Chapter 3](#) sidebar 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 the example, a Spinner widget is included in each `feedList` entry, which will be activated when the corresponding news feed is updated through an Ajax Request. Start by adding the Spinner declaration into the `feedList`'s entry template, *feedList/feedRowTemplate.html*:

```
<div class="palm-row" x-mojo-tap-highlight="momentary">
  <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>
```

The new line is the `div` between the `div`s containing the `numUnread` and the `title` property references.

```
<div x-mojo-element="Spinner" class='feedSpinner' name='feedSpinner'</div>
```

This is simply a declaration of the Spinner widget; the syntax should start to seem familiar by now.

By including it into the `feedList`'s template, we have implicitly directed the list widget to insert a new Spinner element in the DOM whenever it creates a new `feedList` 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 `feedList`'s items array.

We still have to setup the Spinner widget, which we do in the setup method of *feedList-assistant.js*, but we don't include a model in the call to `setupWidget()` as that is assumed to be part of the `feedList` items array.

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

// Setup event handlers list selection, add feed, delete
// feed and reorder feed list
//
this.controller.listen('feedListWgt', Mojo.Event.listTap,
  this.showFeed.bindAsEventListener(this));
this.controller.listen('feedListWgt', Mojo.Event.listAdd,
  this.addNewFeed.bindAsEventListener(this));
this.controller.listen('feedListWgt', Mojo.Event.listDelete,
  this.listDeleteHandler.bindAsEventListener(this));
this.controller.listen('feedListWgt', Mojo.Event.listReorder,
```

```

        this.listReorderHandler.bindAsEventListener(this));

// 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, but 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, `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:

```

feedList.push({title:"New York Times",
    url:"http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml", type:"rss",
    value:false, numUnRead:0, stories:[]});

```

There's one other place where a new `feedList` 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. Otherwise, it's an existing
// feed that's been updated so change it in place.
//
if (this.feedIndex === null) {
    feedList.push({title:this.nameModel.value, url:this.urlModel.value,
        type:"", value:false, numUnRead:0, stories:[]});
    feedError = ProcessFeed(transport);
}
else {
    feedList[this.feedIndex] = {title:this.nameModel.value, url:this.urlModel.value,
        type:"", value:false, numUnRead:0, stories:[]};
    feedError = ProcessFeed(transport, this.feedIndex);
}

```

The Spinner is setup 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 places to change, all in the `feedList` assistant:

- `feedRequest`, activate before Ajax Request.
- `feedRequestFailure`, deactivate.
- `feedRequestSuccess`, deactivate after processing new feed data, and activate before another feed update request is made.

The sample below shows the changes to `feedRequestSuccess`, which includes both an activate and a deactivate call:

```

// Process the feed, identifying the current feed index and
// passing in the transport object holding the updated feed data
//
feedError = ProcessFeed(transport, curFeedIndex);

// If successful processFeed returns errorNone, otherwise there
// was a problem with the feed
//
.
.
.

// Turn off feed update indicator
var spinnerModel = feedList[curFeedIndex];
spinnerModel.value = false;
this.controller.modelChanged(spinnerModel, this);

// Increment the index. If this is NOT the last feed then update the
// feedsource and request to retrieve the next feed
//
curFeedIndex++;
if(curFeedIndex < feedList.length) {
    curFeedSource = feedList[curFeedIndex];

    // Turn on indicator feed update
    spinnerModel = feedList[curFeedIndex];
    spinnerModel.value = true;
    this.controller.modelChanged(spinnerModel, this);

    this.feedRequest(curFeedSource);
} else {

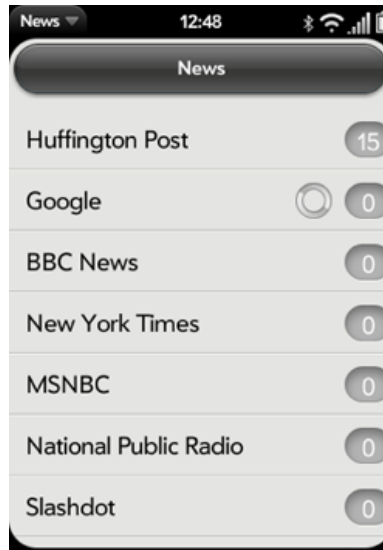
    // Otherwise, this update is done. Reset index to 0 for next update;
    // if the timer is null, the set the interval to feedUpdateInterval
    // which is set in Preferences
    .
    .
    .

```

In each case, we set the `spinnerModel.value` and call `this.controller.modelChanged` to update the model changing the state of the Spinner. The model is accessed by referencing the `curFeedIndex`, 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 and 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 unreadcount icon on the right side, as shown in [Figure 5-1](#), with the Spinner on the Google 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 get truncated to accommodate the Spinner, then reflow 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.

5.1.2. 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 Bar and Progress Slider, are more specialized, but are functionally derived from Progress Pill and you'll manage them in the same way.

5.1.2.1. Progress Pill

Use a Progress 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. An example of the Progress Pill is shown in [Figure 5-2](#).

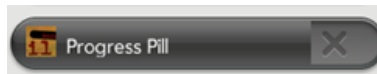
The indicator is designed to show a Pill image that corresponds to the model's value property, where a value of 0 has no Pill exposed and a value of 1 has the Pill completely filling the container. Initialize the indicator's model value to 0, then progressively update the model property until it has 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.completeProgress();
}
else {
    this.lgProgressPillModel.progress = this.lgProgressPillModel.progress + 0.2;
    this.controller.modelChanged(this.lgProgressPillModel);
}

```

Figure 5-2. A Progress Pill example.

5.1.2.2. Progress Bar

The Progress Bar is exactly the same as the Progress Pill except that you use "x-mojoelement="ProgressBar" in your scene file. Otherwise, you would code it and manage it just as you do the Progress Pill. [Figure 5-3](#) shows the Progress Bar widget.

Figure 5-3. The Progress Bar widget.

In the default style, there isn't room on the bar for a title or image, but the properties are supported nonetheless.

5.1.2.3. Progress Slider

For media, or other applications where you want to show progress as part of a tracking slider, then 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. [Figure 5-4](#) shows the Progress Slider widget.

Figure 5-4. The Progress Slider widget.

All of the Slider properties are represented and you configure the Progress Slider just as you would a Slider widget. You only have the attribute properties title and image from the Progress Pill and you have a model property of `value`, which can be renamed through the attributes property. You manage it exactly as you would with 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:

```
this.controller.setupWidget('my-widget', Attr, this.widgetModel);
```

Later the element is added to the DOM:

```
$('#anElement').update('<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.

5.2. Scroller

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.

NOTE

Note: 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:

1. `free`: allow scrolling along both the horizontal and vertical axis.
2. `horizontal`: allow scrolling only along the horizontal axis.
3. `vertical`: allow scrolling only along the vertical axis.
4. `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.
5. `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.
6. `vertical-snap`: similarly, this mode locks scrolling to the vertical axis, but 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.

NOTE

Note: 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 their contents and without constraining the width, on a horizontal scroller, or the height, on a vertical scroller, the scroller will not work.

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 may put a small horizontal scroller inside the default scene vertical scroller. This configuration will cause horizontal drags targeting the horizontal scroller to scroll the horizontal scroller, but vertical drags on the horizontal scroller, or any kinds of drags outside the horizontal scroller to target the scene scroller.

5.2.1.

5.2.1.1. Back to the News: Add Featured Feed Scroller

In this example, a rotating feature story will be added to the News application. This will present the title and story of one of the stories in the featured feed for five 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 `feedList` to allow for a stable view, but we'll attach a vertical scroller to allow for the full story to be read even if it is longer than the view.

Create the feature story on the `feedList` scene by replacing the `.palm-header` in *feedList-scene.html* with a `.palm-group` enclosing a Scroller declaration and two divs to insert a feature story title and text. You should be familiar with the `.palm-group` style which includes a title. In this case **Featured Feed** is a placeholder for the feed title, which will be inserted once the featured feed is selected.

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.

```

<!-- Rotating Feature Story -->
<div class="palm-group">
  <div class="palm-group-title">
    <span id="featuredFeedLabel" x-mojo-loc=''>Featured Feed</span>
  </div>

  <div x-mojo-element="Scroller" id="featureScroller" class="featureScroller">
    <div id="featureStoryDiv">
      <div id="featureStoryTitle" class="featureTitle">#{title}</div>
      <div id="featureStory" class="featureSummary">#{text}</div>
    </div>
  </div>
</div>

<!-- Feed List -->

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

```

Setup the Scroller in the *feedList-assistant.js*. The model defines a single property, the scroller mode, in this case vertical, and the Scroller is setup 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 however you don't normally need to.

```

// Setup scroller and handler for feature feeds, which will trigger a read story
//
this.featureScrollerModel={
  mode: "vertical"
};

this.controller.setupWidget('featureScroller', this.featureScrollerModel);
this.controller.listen('featureStoryDiv', Mojo.Event.tap,
  this.readFeatureStory.bindAsEventListener(this));

```

A listener is set up to handle taps in the feature story `div`, but it is unrelated to the Scroller. If the user sees a story they want to read further, they can scroll the story or tap it to go to the `storyView` scene with that story.

The CSS completes the implementation by fixing the size of the Scroller `div` and formatting the contents. The `.featureScroller` style bounds the Scroller's height, because we have a vertical scroller; in the case of a horizontal scroller, you'd bound the width. If you aren't careful with your CSS, the Scroller will not behave as expected.

```
.featureScroller {
    height: 100px;
    margin-bottom: 10px;
}

.featureTitle {
    padding-top: 0px;
    padding-right: 0px;
    padding-left: 0px;
    font-size: 12pt;
    text-align: left;
    font-weight: bold;
    color: black;
}

.featureSummary {
    vertical-align: top;
    padding-bottom: 0px;
    padding-right: 0px;
    padding-left: 0px;
    font-size: 10pt;
    font-weight: normal;
    text-align: left;
    color: black;
}
```

The `.featureTitle` and `.featureSummary` style the contents of the story presenting styles consistent with the `storyList` and `storyView` scenes.

The rest of the sample is related to handling the feature story. We'll set up some global variables to control the feed and story selection in the *stage-assistant.js*:

```
var featureFeedEnable = true;           // If true, feature feed rotation is enabled
var featureIndexFeed = 0;               // Tracks current feature feed
// which can be set in preferences
var featureIndexStory = 0;              // Tracks story for rotating story
var featureStoryTimer = null;           // Interval timer for rotating features story
var featureStoryInterval = 5000;       // Interval between feature story changes
```

And create the `showFeed` to present the feature story and setup the timer for the next story. We set the timer default to 5 seconds (5000 milliseconds) in `featureStoryInterval`.

```
// showFeatureStory - simply rotate the stories within the featured feed
//
FeedListAssistant.prototype.showFeatureStory = function() {

    // If timer is null, either initial story or restarting so start with
    // previous story. Otherwise, get next story in list before displaying.
    // Set up this way, featureIndex will reflect currently displayed story.
    //
    if (featureStoryTimer === null) {
        featureStoryTimer = window.setInterval(this.showFeature
        Story, featureStoryInterval);
    }
}
```

```

else {
    featureIndexStory = featureIndexStory+1;
    if(featureIndexStory >= feedList[featureIndexFeed].stories.length) {
        featureIndexStory = 0;
    }
}

// Extract summary from story text, stripping HTML tags and links if any and
// unescape the title
var story = feedList[featureIndexFeed].stories[featureIndexStory]
var summary = story.text.replace(/(<([>]+)>)/ig, "");
summary = summary.replace(/http:\S+/ig, "");
$("featureStory").innerHTML = summary;
$("featureStoryTitle").innerHTML = unescape(story.title);

};
// readFeatureStory - handler when user taps on feature story;
// will push storyView with the current feature story.
//
FeedListAssistant.prototype.readFeatureStory = function() {

    this.clearTimers();
    Mojo.Controller.stageController.pushScene("storyView",
    feedList[featureIndexFeed],
    featureIndexStory);
};

```

Following `showFeatureStory` is `readFeatureStory`, which simply stops the interval timers and 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.



We can refine this feature by including some preference settings. Feature feed rotation can be selectively enabled and the rotated feed can be a preference setting rather than just the first feed on the list as we are showing it now.

Add a Toggle Button to the *preferences.assistant.js* to enable or disable the feature, and a List Selector to select the featured feed when enabled. The added code is the middle set of divs enclosed with the `.palm-group`. The two widgets are wrapped with the `.palm-group` style, a `.palm-list` style, and individual list row styles.

```
<div class="palm-page-header">
  <div class="header-icon news"></div>
  <div class="header-text">
    <span id="newsPrefsTitle" class="feedTitleSpan">News</span>
  </div>
</div>

<div class="palm-group">
  <div class="palm-group-title" <span x-mojo-loc="">Feature Feed</span></div>
  <div class="palm-list">
    <div class="first row">
      <div class="palm-row-wrapper">
        <div id="featureFeedToggle" class="featureFeed
Toggle" x-mojo-element="ToggleButton">
          </div>
          <span class="featureFeedToggleLabel">Enable Feature Feed</span>
        </div>
      </div>
      <div class="last row">
        <div class="palm-row-wrapper">
          <div id="featureFeedList" x-mojo-element="ListSelector"></div>
        </div>
      </div>
    </div>
  </div>
</div>

<div class="palm-list">
  <div class="palm-row">
    <div id="feedCheckIntervalList" x-mojo-element="ListSelector"></div>
  </div>
</div>
```

Then add the setup and listener functions to the setup methods of the preferences assistant. The Toggle Button is set up with just the default settings and the value set to the current `featureFeedEnable` state, either true or false. A listener is set up, which will be covered next, to process the toggle changes.

The List Selector is loaded with the current feed titles in `feedList`, by pulling the titles out of `feedList` and putting them into the choices array. Any existing feed can be selected as the feature feed, and the current selection, `featureIndexFeed`, is set as the initial value of the List Selector.

```
// Toggle for enabling/disabling feature feed
//
this.controller.setupWidget('featureFeedToggle', {},
  this.featureFeedToggleModel = {value: featureFeedEnable} );
this.controller.listen('featureFeedToggle', Mojo.Event.propertyChange,
  this.featureFeedToggleHandler.bindAsEventListener(this));
```

```
// Setup list selector for FEATURED FEEDLIST choices and a
// handler for when it is changed;
// Pull out feedList titles for use in selector
//
var x;
var choices = [{label: "", value: ""}];

for (x=0; x<feedList.length; x++) {
    choices[x] = {label: feedList[x].title, value: x};
}

// Now set up listselector and listener for list selections
//
this.controller.setupWidget('featureFeedList',
    { label: "Feature Feed",
      choices: choices
    },
    this.featureFeedListModel = {
      value: featureIndexFeed,
      disabled: false
    });

this.controller.listen('featureFeedList', Mojo.Event.propertyChange,
    this.featureFeedListHandler.bindAsEventListener(this));
```

Now add the listeners to the end of *preferences-assistant.js*. The `featureFeedToggleHandler` will set `featureFeedEnable`, either enabling or disabling the feature feed rotation and the List Selector. You'll notice that the List Selector, `featureFeedList`, will be enabled or disabled according to the value of `featureFeedEnable` so that the user is reminded that the feature is disabled if they try to select an alternate feed.

```
// featureFeedToggleHandler - disables/enables feature feed
//
PreferencesAssistant.prototype.featureFeedToggleHandler = function(event) {

    featureFeedEnable = this.featureFeedToggleModel.value;
    // Change the feature feed status
    featureIndexStory = 0; // Restart story rotation
    if (featureFeedEnable === true) {
        this.featureFeedListModel.disabled = false;
        this.controller.modelChanged(this.featureFeedListModel, this);
    } else {
        this.featureFeedListModel.disabled = true;
        this.controller.modelChanged(this.featureFeedListModel, this);
    }
};

// featuredFeedListHandler
//

PreferencesAssistant.prototype.featureFeedListHandler = function(event) {
    featureIndexFeed = this.featureFeedListModel.value; // Update the featured feed
    featureIndexStory = 0; // Restart story rotation
};
```

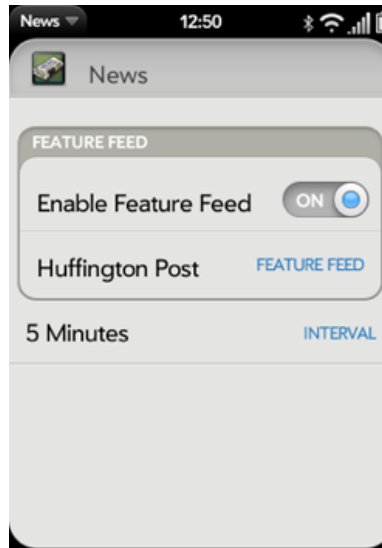
The `featureFeedListHandler` will assign the selected feed to be the featured feed and restart the story rotation at the beginning. A small amount of CSS is needed to clean up the alignment of the toggle and the label; once that's implemented, you will see the scene shown in [Figure 5-6](#) when you tap the preferences menu item.

```
.featureFeedToggleLabel {
    position: relative;
```

```
padding-left: 14px;
margin-top: 80px;
}

.featureFeedToggle {
  position: relative;
  padding-bottom: 9px;
}
```

Figure 5-6. News with feature feed preferences.



There's still some cleanup needed. A selectable feature story must be factored into list reordering, and list delete must be updated to account for the `featureIndexFeed`. There are more efficient designs than what's done here, but for this handling of the feature story, it's necessary to make those adjustments. You can see the changes made to the `listDeleteHandler` and `listReorderHandler` in the *feedList-assistant.js* in News source listing in Appendix B.

5.3. Pickers

Pickers are used to present a common user interface for selecting inputs in a variety of application scenarios. Mojo offers pickers for common objects such as a date, time or a number, or to select files.

The next section, Simple Pickers, covers the first three pickers since they are conventional widgets and very similar to each other. After that, we look at the File Picker. It's accessed through function calls, and is actually implemented as a separate application wrapped with a framework interface.

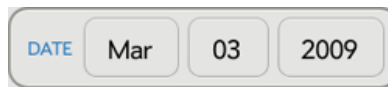
5.3.1. Simple Pickers

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 5-7](#)).

Figure 5-7. The Date Picker.



The pickers present their choices as a linear sequence of values that wraps around; when you scroll to the end of the sequence, it simply continues back at the beginning. There's no way to override this behavior at this time.

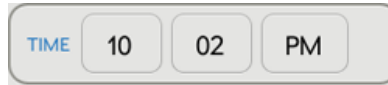
5.3.1.1. Date Picker

As shown in [Figure 5-7](#), the 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.

5.3.1.2. Time Picker

The Time Picker is similar to the 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 5-8](#), the Time Picker enables selection of hours, minutes and either AM or PM, 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 5-8. The Time Picker.

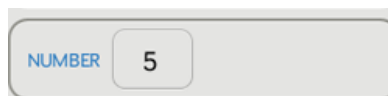


You can use the JavaScript functions `GetHours()`, `GetMinutes()` and `GetSeconds()` to extract what you need from the `Date` object.

5.3.1.3. Integer Picker

A simple number picker is included as the Integer Picker. Shown in [Figure 5-9](#), 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 5-9. Integer Picker



The integer picker is similar to the Date and Time pickers in all ways other than that its default model property is named `value`.

5.3.1.4. Back to the News: Add Integer Picker

News doesn't have a 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 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* after the widgets initially included, similarly wrapped within styling divs. You'll notice that we have adjusted the `row` style on the List Selector to keep the first and last elements within the `first row` and `last row` style. All additional entries will use `row` styles.

```
<div class="palm-group">
  <div class="palm-group-title" <span x-mojo-loc="">Feature Feed</span></div>
  <div class="palm-list">
    <div class="first row">
      <div class="palm-row-wrapper">
        <div id="featureFeedToggle" class="featureFeed
Toggle" x-mojo-element="ToggleButton">
          </div>
          <span class="featureFeedToggleLabel">Enable Feature Feed</span>
        </div>
      </div>
    <div class="row">
      <div class="palm-row-wrapper">
        <div id="featureFeedList" x-mojo-element="ListSelector"></div>
      </div>
    </div>
  </div>
</div>
```

```

    <div class="last row">
      <div class="palm-row-wrapper">
        <div id="featureFeedDelay" x-mojo-element="IntegerPicker"></div>
      </div>
    </div>
  </div>
</div>
</div>

```

The widget setup is included in the `setup` method of *preferences-assistant.js*, as shown next. The widget is setup with a range from 1 to 60 seconds and initialized to the current interval, which is in milliseconds. A listener is added for `propertyChange` events.

```

// Setup Integer Picker to pick feature feed rotation interval
//
var featureDelayAttr = {
  label: 'Interval',
  modelProperty: 'value',
  min: 1,
  max: 60
};
this.featureDelayModel = {
  value : featureStoryInterval/1000
};
this.controller.setupWidget('featureFeedDelay', feature
DelayAttr, this.featureDelayModel);
this.controller.listen('featureFeedDelay', Mojo.Event.propertyChange,
  this.featureDelayChange.bindAsEventListener(this));

```

The handler is added to the bottom the preferences assistant, updating the global interval with the selected value, in milliseconds, and restarts the interval timer with the new value.

```

// featureDelayChange - Handle changes to the interval
//
PreferencesAssistant.prototype.featureDelayChange = function(event) {

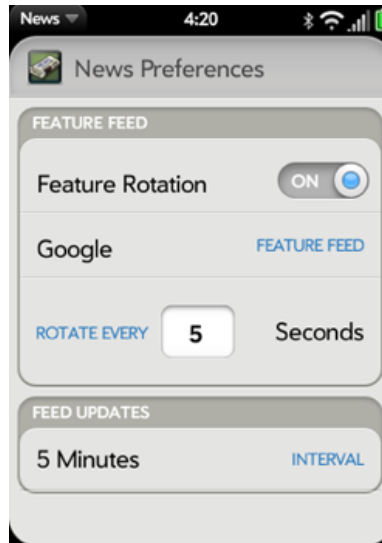
  // Interval is in milliseconds
  featureStoryInterval = this.featureDelayModel.value*1000;

  // If timer is active, restart with new value
  if(featureStoryTimer !== null) {
    window.clearInterval(featureStoryTimer);
    featureStoryTimer = null;
  }
};

```

Figure 5-10 shows what the new preferences scene looks like with the Integer Picker in place.

Figure 5-10. News with Integer Picker.



5.3.2. File Picker

WebOS devices have a Media Partition, a FAT32 file partition that is available to applications, and 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, with which the user can 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, video, etc.) Depending upon the options provided by the calling application, the selected file will either be opened in an appropriate viewer, or its reference returned.

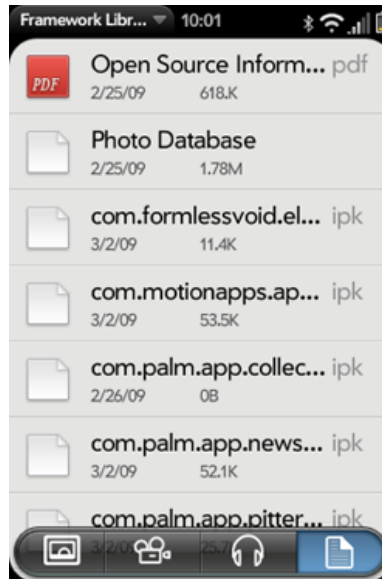
The File Picker behaves like a full screen widget, but isn't technically a widget. It is actually an application that is pushed into the current scene similar to a viewer, maintaining the calling application's context.

The presentation of the files will differ by file type. For example:

- Files[md]Name and icon.
- Images[md]Thumbnail grid.
- Audio and Video[md]Name and thumbnail.

Figure 5-11 shows the file view, with the other view options presented as command menu items across the bottom of the scene.

Figure 5-11. The File Picker.



5.4. 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.

The next section adds a Filter List to News to implement a search feature, which will provide a good example of a dynamic list. In addition, the list widget allows you to intervene in the middle of the list rendering to provide some intermediate formatting to list items or to insert dividers between rows.

5.4.1. 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 their first argument, and the appropriate model object or items element as their second.

5.4.2. Dividers

You can add dividers to your lists, particularly useful for long lists. You will specify the function name to the `dividerFunction` property and a template to `dividerTemplate`. If there is no template specified then 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.

5.4.3. Filter List

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 has 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 and the framework calls the function specified by `filterFunction`.

The framework calls the `filterFunction`, like 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, then replaces it 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.

5.4.3.1. Back to the News: Add 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, presenting 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 setup 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 `feedList` and 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 `feedList` and feature story when in search mode. Start with 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.

```
<div id="searchFieldContainer">
  <div x-mojo-element="FilterList" id="startSearchField"></div>
</div>

<div id="feedListMain">
  <!-- Rotating Feature Story -->
  .
  .
  .
```

Now setup the widget in *feedList-assistant.js*, re-using 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 350 milliseconds. This is the default so could have been omitted but you may need to tune the behavior so it's not a bad idea to add it at the beginning.

```
this.controller.setupWidget("startSearchField",
  this.searchFieldAttr = {
    itemTemplate: 'storyList/storyRowTemplate',
    listTemplate: 'storyList/storyListTemplate',
    filterFunction: this.searchList.bind(this),
    renderLimit: 70,
    delay: 350
  },
  this.searchFieldModel = {
    disabled: false
  });

this.controller.listen('startSearchField', Mojo.Event.listTap,
  this.viewSearchStory.bindAsEventListener(this));
this.controller.listen('startSearchField', Mojo.Event.filter,
  this.searchFilter.bind(this), true);
```

Add two listeners, one for the tap event and the other for a filter event. The filter event is used on Filter Fields and Filter Lists only. It is sent as soon as the filter field is activated, on the first character entry, and when the field is cleared. Listening for this event allows you to do some pre and post processing with the widget or the scene.

With News, you want to hide the rest of the scene when the first character is typed and restore it when the field is cleared. To do that, add this new method in the `feedList` assistant; it simply calls the `hide()` and `show()` methods of the `div` element, `feedListMain`, in *feedlist-scene.html*:

```
// searchFilter (FeedListAssistant) - triggered by entry into search field.
// First entry will hide the main feedList scene and clearing the entry
// will restore the scene.
//
FeedListAssistant.prototype.searchFilter = function(event) {
  if (event.filterString !== "") {
    $("feedListMain").hide();
  } else {
    $("feedListMain").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 that corresponds to the offset and size requested by the widget.
//
FeedListAssistant.prototype.searchList = function(filterString,
listWidget, offset, count) {

  var subset = []; // Ultimately the results list
  var totalSubsetSize = 0; // Used to set counter in Filter Field

  // If search string is null, then return empty list,
  // otherwise 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 i=0; i<feedList.length; i++) {
      for (var j=0; j<feedList[i].stories.length; j++) {
        if(hasString(filterString, feedList[i].stories[j])) {
          var sty = feedList[i].stories[j];
          items.push(sty);
        }
      }
    }
    this.entireList = items;

    // Cut down the list results to just the window
    // asked for by the filterList 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 as shown in [Table 5-1](#).

Table 5-1. FilterFunction Arguments List

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 assigned to `subset`, which is returned with the offset by calling `listWidget.mojo.noticeUpdatedItems(offset, items)`, using the arguments as shown in [Table 5-2](#). This is a method of `listWidget` an argument passed by the framework.

Table 5-2. NoticeUpdatedItems Arguments List

Argument	Type	Description
offset	Integer	Index in the list of the first object in 'items'; usually the same as the offset passed to the <code>itemsCallback</code>
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, you can type at any time into the `feedList` scene and you'll see the filter field display and the results presented in list form below, similar to what is shown in [Figure 5-12](#).

Figure 5-12. News with a search filter list.



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 searchList
//
FeedListAssistant.prototype.viewSearchStory = function(event) {
    var searchList = {title: "Search for: "+this.filter, stories: this.entireList};
    var storyIndex = this.entireList.indexOf(event.item);

    Mojo.Controller.stageController.pushScene("storyView", searchList, storyIndex);
};
```

As shown in [Figure 5-13](#), besides viewing the selected story, you can tap next and previous to view each story in the results list.

Figure 5-13. News with a search story view.

5.5. 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.

5.5.1. WebView

To embed a contained web object, you 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.

5.5.1.1. Back to the News: Add a Web View

Tapping on a story will push a webView 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 setup the *webView* widget.

```
// StoryWebAssistant(storyURL)

//
// Passed a URL and displays the corresponding story or link in a webview,
// handling any link selections within the view. User swipes back to
// return to the calling view.
//
// storyURL      unescaped form of URL

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.controller.listen('storyWeb', Mojo.Event.webViewLinkClicked,
    this.linkClicked.bindAsEventListener(this));

  // Setup App Menu
  this.controller.setupWidget(Mojo.Menu.appMenu, newsMenuAttr, newsMenuModel);
};
```

There are more options than what's been shown. 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 more events than that; you can find a complete list and descriptions in the webOS SDK.

To get to the *webView*, *storyView* will be modified to respond to taps on the story body or links in the story itself. These listeners will be added in the setup method and removed during cleanup:

```
// Setup handlers for taps to the story body and links within the story
this.clickHandler = this.webStory.bindAsEventListener(this);
this.controller.listen("storyViewSummary", "click", this.clickHandler, true);
this.controller.listen("storyViewScene", Mojo.Event.tap, this.clickHandler);
```

Then push the *storyWeb* scene after resolving whether to push the story's originating URL or a link within the story. If you're interested in the details of that, you can look at the *webStory* method of *storyView* in [Appendix A](#).

Load Indicator

Whether you're using a full screen WebView 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 they both need to be setup.
- 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 upon the percent complete.
- That image is inserted into the DOM to display as the indicator.

As shown in [Figure 5-14](#), 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 setup and the scene is active. For example, an application such as email, which might have to present HTML content, can declare and setup 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 webView if that's useful.

Figure 5-14. The webView widget.



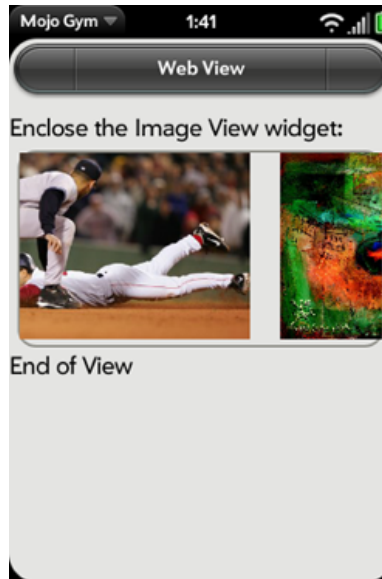
5.5.2. Other Viewers

There are other viewers that you can add to your application: Image View, Audio and Video Players. ImageView is very similar to WebView, but the Audio and Video players are quite different. None of these other viewers are presented in depth, but each is briefly described in this section. There is a lot more information available in the webOS SDK.

5.5.2.1. Image View

Designed to view an image full screen with support for zooming and panning while optionally paging between additional images, the ImageView widget is configured similar to the WebView. 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 5-15](#) shows the ImageView widget, partially paged to the right.

Figure 5-15. The Image View widget.



5.5.2.2. Audio & 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 best when being outside of the app in a media context enhances the media playback experience.

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 would set the source and event handlers before calling the objects 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. Just as with the audio object, you will play video after creating the Video object, setting the source and setting up event handlers.

Unlike the audio object, 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 <http://www.whatwg.org/specs/web-apps/current-work/>.

5.6. Summary

With this review 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 store and retrieval.
- Cookie—a simplified interface to browser cookies, a single object store for small amounts of data.

You will have to evaluate your needs to see which solution is the most optimal.

- 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 with accessing online data.
- Depot can be used if you are storing simple objects and don't need a schema design or to handle transactions or queries; 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 off-line 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. Lastly, 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.

6.1. Using 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 be used to 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 4k bytes, 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 this cookie if it exists or returns `undefined`.
- `put()` updates the value of the named with an optional date/time after which to delete the object.
- `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.

6.1.1. Back to the News: Using a Cookie

We'll add 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.

At application launch, open or create the `this.cookie` object in the *stage-assistant.js* using the constructor function with a single argument, a unique *cookie id*. The following example uses `NewsPrefs`, a form of the *app id* but you can use any unique string.

```
//Get preferences cookie, or create it
this.cookie = new Mojo.Model.Cookie("NewsPrefs");
var oldNewsPrefs = this.cookie.get();
if (oldNewsPrefs) {
    if (oldNewsPrefs.newsVersionString == "1.0"){
        featureIndexFeed = oldNewsPrefs.featureIndexFeed;
        featureFeedEnable = oldNewsPrefs.featureFeedEnable;
        featureStoryInterval = oldNewsPrefs.featureStoryInterval;
        feedUpdateInterval = oldNewsPrefs.feedUpdateInterval;
        updateDialog = oldNewsPrefs.updateDialog;
    } else {
        this.cookie.put() {
            featureIndexFeed: featureIndexFeed,
            featureFeedEnable: featureFeedEnable,
            feedUpdateInterval: feedUpdateInterval,
            featureStoryInterval: featureStoryInterval,
            newsVersionString: newsVersionString
        }
    }
}
});
```

After creating the cookie in our sample, `this.cookie.get()` is called to retrieve it if it exists. If the cookie exists, then the global preferences are updated with the stored values. If it doesn't exist, the cookie is stored to preserve the current settings, using `this.cookie.put()` and without specifying an expiration date. If the expiration date is not specified then the cookie will not expire. If the specified date is earlier than the current date, then the cookie will be deleted.

The cookie should be updated when the preferences change. Modify the `deactivate` method in *preferences-assistant.js* to update the cookie when the preferences scene is popped.

```
// Deactivate - update cookie
//
NewsPrefsAssistant.prototype.deactivate = function() {

    this.stageAssistant.cookie.put(
        {
            featureIndexFeed: this.featureFeedListModel.value,
            featureFeedEnable: this.featureFeedToggleModel.value,
            feedUpdateInterval: this.feedIntervalModel.value,
            newsVersionString: newsVersionString,
```



```
        featureStoryInterval: featureStoryInterval
    });
};
```

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, then you should remove them when they are no longer needed.

6.2. Working with the Depot

If you are only interested in a simple object store without any database queries or structure, then Depot will likely meet your needs. You can store up to 5 megabytes of data in a depot. Mojo provides by a few simple functions that wrap the HTML5 APIs to create, read, update or delete a database.

`Mojo.Depot()` opens the depot that matches the `name` argument or, if there is no match, creates a new depot with that name. There are three methods:

- `simpleGet()` calls the provided `success` method, passing the object retrieved if it exists, or `failure` if no object matches the `key`.
- `simpleAdd()` updates the value of the named object.
- `removeAll()` removes the named depot and deletes the associated data.

The Depot is simple to use. As with Cookies, you use the Depot constructor to create a new Depot or open an existing one, providing a unique key to identify the Depot. Unlike with Cookies, Depot calls are asynchronous so you will do most of your handling in callback functions.

Once opened, you can save and retrieve JSON formatted data; the Depot function will convert between JSON and SQL to make it simple for you. It's a great solution for a simple object cache for offline data access.

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. At this point, there aren't very specific guidelines on complexity so you'll have to experiment to see how the Depot performs with your application. Deep hierarchy, multiple object layers, array and object datatypes or large strings are all characteristics of complex objects that may push the limits of the Depot capabilities.

6.2.1. Back to the News: Add a Depot to News

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 store unread state. Storing this state information will let us provide a better user experience at launch time, by allowing us to present the stored feeds quickly, without having to wait for a full sync from the various servers. Since Depot is asynchronous, start the database operations in the `feedList` assistant's setup method:

```
// Open the database to get the most recent feed list
//
db = new Mojo.Depot({name:"feedDB", version:1, estimatedSize:25000, replace: false},
  this.dbOpenOK.bind(this), this.dbOpenFail.bind(this));
```

The first argument is a JSON 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 `estimated size` 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 then an existing depot will be replaced. The `replace` property is optional; if missing, it defaults to false.

The `dbOpenOK` callback will handle either the case where the `feedList` had been previously saved, or where this is a new database. The `dbOpenFail` callback will be called if there is a database error. The cause of such a failure could be either that the database failed to open but exists, or that it didn't exist and failed to be created.

In `dbOpenOK`, attempt to retrieve the data with a call to the `simpleGet` 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, a JSON object with the returned data. In the following example, the success callback is `updateList`, which first tests the returned object, `fl`, for null or an empty object. If `fl` is a valid object, it is assigned to `feedList` and `feedList` widget's model is updated as well to update the display, and the featured feed rotation is started if there are stories in the featured feed.

```
// dbOpenOK - Callback for successful DB request in setup. Does two things:
// Get feedlist from DB: call updateList if successful,
// otherwise useDefaultList for default set of feeds
//
FeedListAssistant.prototype.dbOpenOK = function() {

  Mojo.Log.info(".....", "Database opened OK");
  db.simpleGet("feedList", this.updateList.bind(this),
    this.useDefaultList.bind(this));

};

// updateList - Callback for successful DB retrieval of feedlist. Call
// useDefaultList if the feedlist empty or null or initiate an update
```

```
//    to the list by calling updateFeedList.
//
FeedListAssistant.prototype.updateList = function(fl) {

    Mojo.Log.info(".....","database size: " , Object.values(fl).size());

    if (Object.toJSON(fl) == "{}" || fl === null) {
        Mojo.Log.info("Retrieved empty or null list from DB");
        featureIndexFeed = 0;
        this.useDefaultList();
    } else {
        Mojo.Log.info(".....","Retrieved feedlist from DB");
        feedList = fl;
        this.feedWgtModel.items = feedList;
        this.controller.modelChanged(this.feedWgtModel);

        //    If stories exist in the featureIndexFeed, then start the rotation
        //
        if(featureIndexFeed < feedList[featureIndexFeed].stories.length) {
            $("featuredFeedLabel").innerHTML = feedList[featureIndexFeed].title;
            this.showFeatureStory();
        }

        this.updateFeedList();
    }
};
```

Should the `simpleGet` 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 `useDefaultList` as the second (failure case) callback. The default `feedList` is assigned and another Depot method is used, this time `simpleAdd`, to save the default data. There are success and failure callback functions for this call as well. In this case they are provided as literal functions to log the save results and post an error dialog to the user in the fail case.

```
//    useDefaultList - Callback for failed DB retrieval meaning no list so use default.
//
FeedListAssistant.prototype.useDefaultList = 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.
    //
    db.simpleAdd("feedList", feedList,
        function() {Mojo.Log.info(".....","feedList saved OK");},
        function(transaction,result) {
            Mojo.Log.warn("Database save error (#", result.message,
                ") - can't save feed list. Will need to reload on next use.");
            Mojo.Controller.errorDialog("Database save error (#" +
                result.message + ") - can't save feed list. Will need to reload on next use.");
        });
    Mojo.Log.warn("Database has no feed list. Will use default.");
    this.updateFeedList();
};
```

If the call to open the database fails, it's because of a database error with no way to recover. The error is logged and the user notified with an error dialog.

```
//    dbOpenFail - Callback for failed DB open during setup. Alerts user and continues.
//
```

```
FeedListAssistant.prototype.dbOpenFail = function(transaction, result) {
    Mojo.Log.warn("Can't open feed database (#", result.message,
        "). All feeds will be reloaded.");
    Mojo.Controller.errorDialog("Can't open feed database
        (#" + result.message + "). All feeds will be reloaded.");
};
```

This covers the launch events involving the depot data. The rest of the handling is related to updating the depot after the data has changed. This is critical as 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 it will be done with this call:

```
db.simpleAdd("feedList", feedList,
    function() {Mojo.Log.info(".....", "feedList saved OK");},
    function(transaction, result) {
        Mojo.Log.warn("Database save error (#", result.message,
            ") - can't save feed list. Will need to reload on next use.");
        Mojo.Controller.errorDialog("Database save error (#" +
            result.message + ") - can't save feed list. Will need to reload on next use.");
    });
```

These calls would be implemented at these points in the application:

- **AddDialog:** a new feed has been added so clearly an update is required to save that new feed and contents.
- **showFeatureStory:** this might seem odd, but since this is a periodic updater, a flag can be set indicating that the data has changed from a feed update, or unread stories being read, then performing the update here for 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 `removeAll()` method if that's something you need:

```
db.removeAll();
```

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.

6.3. HTML 5 Storage

Clearly the Cookie and Depot functions 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.

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, but does support the Database object.

The `openDatabase()` method will create a new database or open an existing database, returning a database object.

```
db = openDatabase(name:"myDB", version:"1", displayName:"My DB", estimatedSize"10000");
```

The arguments are each a property/value pair with an argument list (see [Table 6-1](#)).

Table 6-1. openDatabase Arguments List

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 then both need to match an existing database for the database open to succeed, or an error is generated. If the database doesn't exist, then it will be created.

Once the database is open, you are able to 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() {
    .
    .
    .
    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, which includes
    };

    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

NOTE

Note: Following the guidelines contained in the HTML 5 Draft Recommendation, webOS imposes a 5-megabyte limit to HTML 5 databases for any individual application. This limit and the database support will likely change over time, so check the Palm SDK site for the latest information.

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.

6.4. 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 them immediately.

You aren't required to use the Prototype functions. You can use the *XMLHttpRequest* object directly and will need 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". For a more basic introduction, you can review the tutorial at <http://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 help you see how Prototype can help you integrate dynamic data into your application.

NOTE

Note: Palm webOS applications are run from *file://* URLs and thus aren't restricted by the single-origin policy that makes mixing services from different web sites difficult.

6.4.1. 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 `feedRequest` to initiate the Ajax request:

```
// feedRequest - function called to setup and make a feed request
//
// Uses prototype's Ajax.Request and sets up callback routines:
//   onSuccess      feedRequestSuccess
//   onFailure      feedRequestFailure
//
```

```
FeedListAssistant.prototype.feedRequest = function(curFeed) {

    var request = new Ajax.Request(curFeed.url, {
        method: 'get',
        evalJSON: 'false',
        onSuccess: this.feedRequestSuccess.bind(this),
        onFailure: this.feedRequestFailure.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 `curFeed.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 life cycle shown in [Table 6-3](#).

Table 6-3. Ajax.Request callbacks by life cycle stage

Callback	Lifecycle Stage
<code>onCreate</code>	Created
<code>onUninitialized</code>	Created
<code>onLoading</code>	Initialized
<code>onLoaded</code>	Request Sent
<code>onInteractive</code>	Response being received (per packet)
<code>on###</code> , <code>onSuccess</code> , <code>onFailure</code>	Response Received
<code>onComplete</code>	Response Received

To clarify:

- `onCreate` is available only to Responders and not available as a property of `Ajax.Request`.
- When `on###` is specified (where `###` is an HTTP status code, e.g. `on403`), 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 a more `Ajax.Request` options available. If you're interested in learning more, `Ajax.Request` is covered thoroughly in most Prototype references, including the Prototype 1.6 API reference available from www.prototypejs.org.

6.4.2. 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
<code>readyState</code>	Integer	Current lifecycle state: 0 = Uninitialized, 1 = Initialized, 2 = Request Sent, 3 = Interactive, 4 = Complete
<code>status</code>	Integer	HTTP status code for the request
<code>statusText</code>	String	HTTP status text corresponding to the code
<code>responseText</code>	String	Text body of the response
<code>responseXML</code>	XML or Document	If content type is application/xml, then XML body of the response; null otherwise
<code>responseJSON</code>	Object	If content type is application/json, then JSON body of the response; null otherwise
<code>headerJSON</code>	Object	Sometimes JSON is returned in the X-JSON header instead of response text; if not this property is null
<code>request</code>	Object	Original request object
<code>transport</code>	Object	Actual XMLHttpRequest object

We haven't discussed JSON (JavaScript Object Notation) 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.

6.4.3. 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.

6.4.3.1. 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);
  },
});
```



```

    onSuccess: function(response) {
        spinnerModel.value = false;
        this.controller.modelChanged(spinnerModel, this);
        this.processUpdate(response)
    },

    onFailure: function(response) {
        this.spinnerModel.value = false;
        this.controller.modelChanged(spinnerModel, this);
        var status = response.status;
        Mojo.Log.info(".....", "Invalid URL (Status ", status, " returned).");
        Mojo.Controller.errorDialog("Invalid feed - http failure (" + status + ")");
    }
});

```

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 post-processing.

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.

6.4.3.2. `Ajax.Updater` & `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 decay option to extend the interval between updates when responses are unchanged. Note that `Ajax.PeriodicalUpdater` uses JavaScript timers and won't wake the device.

6.5. 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 covered both topics, showing you how to use the Depot and Cookie objects, along with Prototype's Ajax functions and the HTML 5 Database APIs to provide what you need. 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 mostly a matter of focus.

This chapter will focus on look and feel. 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 they are positioned and sized within a scene's layout, and how to 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 for you. There are extensive resources available to learn more about styling or various visual and interaction design topics:

- 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 for learning and applying styles on webOS.
- 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.

7.1. Typography

You read about text widgets in [Chapter 3](#), including the available styling options. Some of those 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 and some basic alignment techniques.

7.1.1. Fonts

Prelude is the primary typeface used in the Pre. 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. 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.

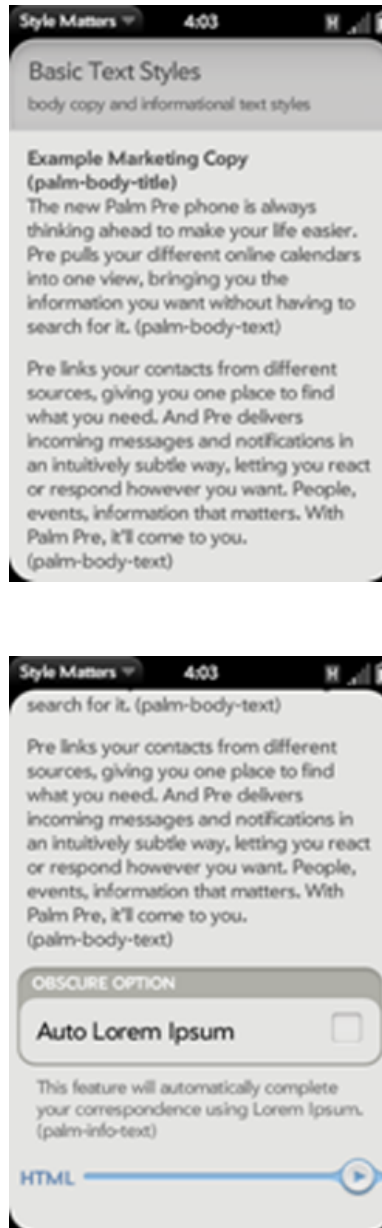
[Table 7-1](#) summarizes the available typestyles.

Table 7-1. Typestyles

Typestyle	Technique
Prelude Medium	Provided with all text elements and classes by default
Prelude Medium Oblique	Add <code>oblique</code> class to any text element or class
Prelude Medium Condensed	Add <code>condensed</code> class to any text element or class
Prelude Medium Bold	Add <code>bold</code> or <code>strong</code> style to any text element

Body text can be styled with a few basic text classes. You can find a full description of the CSS selectors in the Basic Text Styles section of Appendix C, "Quick Reference – Style Guide," but an example is 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 <nobr>(palm-body-title)</nobr>
  </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. <nobr>(palm-body-text)</nobr>
  </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

7.1.2. 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.



In the left-most example, you are truncating a text string within a conventional div:

```
<div class="palm-list">
  <div class="palm-row first" x-mojo-tap-highlight="momentary">
    <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>
</div>
```

In the other example, there are two text fields, each using single-line truncation but within different formats. 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 you want to disable truncation, set the `multiline` property to `true` to request that the widget expand the text field instead of truncating in response to a line wrap.

7.1.3. 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
<code>capitalize</code>	Use <code>capitalize</code> to CSS title-case capitalization on any text element

Class	Description
palm-page-header	Page Header text element
palm-dialog-title	Dialog title for error and alert dialogs by default, or when used in HTML
palm-button	Button label

Use the `un-capitalize` class to override auto-capitalization 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 auto-capitalization.

7.1.4. 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](#), with the top box demonstrating the single line alignment and the lower box 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.

7.2. Images

This section will help you use images within your application, whether you want to re-use images provided with Mojo or create your own images. There will be 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 so many that we didn't include them in this book, and new ones are added with each SDK release.

You will also find design and technical guidelines here that will help you, if you're looking to 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 consistently with the way they are used in the System UI or Palm applications.

NOTE

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.** Single image within a single file for conventional image use with `img` tags or similar cases.
- **Multi-State Image.** Multiple images within a single file used to combine multiple button states (e.g. pressed and un-pressed button 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.

7.2.1. 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.

7.2.2. Multi-State Image

When displaying an animation or multi-state 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, eliminating flicker between the states and conveniently keeps the assets together. Some examples of multi-state images are shown in [Figure 7-5](#).

Figure 7-5. Multi-state image examples.



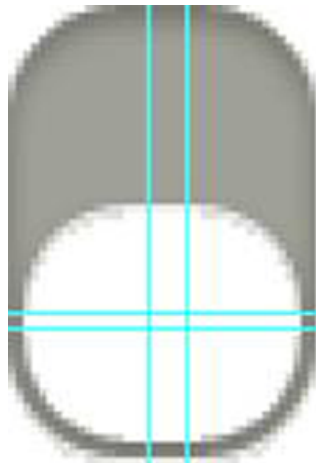


Multi-state images are used for Pushbuttons, Menu buttons, Indicators, Toggles and Checkboxes among other elements. It will be used anywhere you have a single image that needs to reflect state changes (e.g. unselected/highlighted/selected) or is part of an animation sequence.

7.2.3. 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.

Figure 7-6. Parts of a 9-tile 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.

Figure 7-7. 9-tile image examples.



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;
}
```

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 of them. If you're not familiar with `webkit-border-image`, it is based on the CSS3 `border-image` standard and you can find many resources for further information on this standard on the web.

7.2.3.1. 3-Tile Image

A 3-tile image is a 9-tile image with a zero border in one vector. You would use a 3-tile image when you need an image that scales in one dimension only. Some examples are: radio button strip, dashboard containers, view menus and page headers.

Create a 3-tile image by creating a 9-tile image and setting one dimension to zero, as shown below with the `palm-slider-background`:

```
.palm-slider-background {
  width: 250px;
  height: 7px;
  border-width: 0px 4px 0px 4px;
  -webkit-border-image: url(../images/slider-background-3tile.png)
  0 4 0 4 repeat repeat;
  margin: 6px 0px 0px 20px;
}
```

Negative Margin

A div that functions as a container with a border image is unable to use the space allocated to the border image for any content. The framework uses a technique called Negative Margin to reclaim that space to enable content to be placed within the full width and height of the container.

The basic technique is to define a second div, and place the content there rather than 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 popup, 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: 199500;
  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;
}
```

And the border's width is reclaimed in the wrapper class to enable you to place content to full width and height of the parent:

```
.palm-popup-wrapper {
  margin: -24px;
}
```

You would 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>
```

7.2.3.2. 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.

7.3. Touch

Since touch is the primary indicator of action, it is critical that your scenes are styled optimally for `touchability`. Here are some strategies that you should consider for creating large, gapless hit targets.

7.3.1. Maximize Your Touch Targets

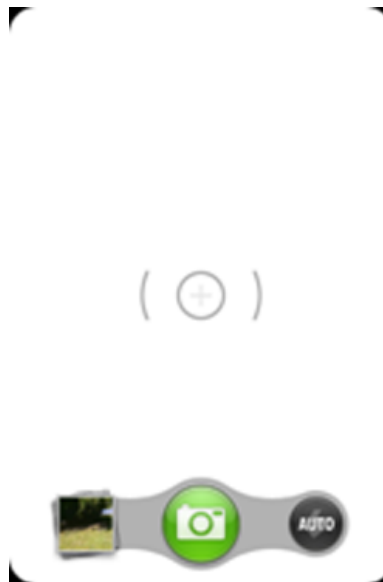
The elements in your scene may appear to be small and separate from each other, but their touch targets should be as large as possible. Elements in rows should be as tall as the row itself and extend (when adjacent) to the very edge of the scene.

Ensure that the touchable elements in your scene are as large as possible. Touch targets in rows should be as tall as the row itself, and should butt up against each other, such that there's no dead-space in between them. You should maximize the size of the touch targets (elements in a row should be full height) and there should not be any gaps between targets.

Visual elements can be smaller than touch elements, so you might wrap the *image* div with a *touch* div. An example with the camera button, where the image is 80x60, 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;
}
```

Yet 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.

7.3.2. Optimizing Touch Feedback

Use the `x-mojo-tap-highlight` attribute so that 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-tap-highlight` adds a delay on focus so that incidental touches won't cause a highlight. For example, in the News app, we used a `momentary` tap highlight in *views/feedList/feedRowTemplate.html*:

```
<div class="palm-row" x-mojo-tap-highlight="momentary">
  <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 `momentary` feedback. For fixed elements that don't scroll, `immediate` feedback is an option. Use `persistent` only if you require exacting control of when feedback is removed. This must be done manually.

7.3.3. 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.

7.4. Light & Dark Styles

If your application uses a light-colored background with dark text/controls, the default controls and text colors should work very well for you. If your application uses a dark-colored background with light text/controls, use the `palm-dark` styled controls and text. Some of the applications Palm ships on the phone use the `palm-dark` controls (Music, 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 (for example, a drawer widget). To change the body element, 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 have the scene or body element defined with the class because submenus, dialogs and any other z-stacked element will inherit 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 by setting the `theme` property in your *appinfo.json* file:

```
"theme": "light"
```

7.5. Summary

In this chapter, some of the widget and scene styles were highlighted, and you were presented more information on using and creating images within styles, text styles, use of dividers. You were also introduced here to Palm's light and dark styles, two different style moods that Palm has used to distinguish the media applications from the productivity applications.

The Palm Mojo framework includes numerous styling options for you to use within your application. Many of those 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, selectors and guidelines on how to apply them in working with widgets and scenes.

Chapter 8. Application Services

So far we've looked at user interface elements and web service requests, but very little of what we've done is something you couldn't do in a sufficiently capable web browser. What makes Mojo particularly powerful is its access to Application Services that encapsulate both low-level hardware capabilities and higher-level data services that provide access to Palm webOS Synergy™ features, cloud services, and beyond. 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, which 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 the next two chapters 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, SMS and Instant Messaging.

In [Chapter 9](#) we conclude our discussion of the service layer by delving into cloud services and lower-level system services like GPS.

8.1. 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 GPS 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, which 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.

8.1.1. 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
},
requestOptions
);
```

Palm webOS uses the URI 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, you would 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 should the service call be successful and may be called multiple times for service requests that result in a series of results, such as a request for GPS tracking data instead of just a single fix. The `onFailure` function is called should the service call result 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
<code>returnValue</code>	True on success or false on failure of this request	Required
<code>errorCode</code>	The error code from the service when <code>returnValue</code> is false	Required
<code>errorText</code>	Description of the failure when <code>returnValue</code> is false	Required
<code>subscribed</code>	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 App Assistant, then you'll need to create a service request object.

A common case is a call to the Alarm service to wakeup 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 on 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, then 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 then use `Mojo.Service.Request()`.

8.1.2. 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 launch to handle the content indicated.
- *Launch* launches the application indicated by the application id argument passing any included parameters.

8.1.2.1. Open

Generally, the `open` method is used where 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 '://', 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 PDFViewer 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 that matches, then again the associated application will be launched with the file reference as a launch parameter. If there's no match at this point then 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 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://` then it's a local file reference and the Application Manager will use the file extension to launch the associated application if there is one.

8.1.2.2. Launch

There are many cases where 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 this case, you'd want a command to launch a specific app, 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('mojo://com.palm.applicationManager', {
  method: 'launch',
  parameters: {
    id: "com.palm.app.maps",
    params: {
      query: "950 W. Maude Ave, Sunnyvale,CA"
    }
  }
});

```

The Command & 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.

8.1.3. Cross-App Launch

In some cases, a Cross-App launch is used to keep the context of the calling application, with a faster transition. The target application's scene is pushed directly in 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 Picker, both of which use this technique and are covered later in this chapter.

8.2. Core Application Services

This first group of application services includes a core set of applications that provide basic functions for the 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 would call them from use within your application.

8.2.1. Browser

Earlier, a WebView widget was used to display the source URL for the News story, but launching the full Browser in a new card gives more flexibility to the application. The Browser application can be launched to its default launch view or to a specific URL.

8.2.1.1. Back to the News: Launch 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 `webStory` method of *story-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 app 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.

```
// webStory
//
```

```
// Launch the Browser when user taps a story; will launch original story URL
// or a tapped link within a webView
//
StoryViewAssistant.prototype.webStory = function(event) {
    this.controller.serviceRequest('palm://com.palm.applicationManager', {
        method: 'open',
        parameters: {
            id: 'com.palm.app.browser',
            params: {
                target: this.storyFeed.stories[this.storyIndex].url
            }
        }
    });
};
```

When you make this change and tap on 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 on Email & Messaging a bit later in the chapter.

8.2.2. Phone

The user must tap the dial button to approve of 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 pre-populated number.



8.2.3. 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 *sceneArguments* 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");
  }
};
```


8.2.4. 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: {}
  }
});
```

8.2.5. Maps

You can use the Map 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.

8.3. Palm Synergy™ Services

Palm Synergy integrates your personal information from various sources on the web and presents it in a single view so that you can see all in one place. Yet the information is maintained in such a way that you can keep things separate when you 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 could 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 that the Phone application is used.

The Contacts and Calendar application service interfaces can do a bit more, enabling 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.

8.3.1. 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: {
    username: "myusername",
    domain: "mydomain",
    displayName: "My Name",
    icons: {largeIcon: '../accountIcon.png', smallIcon: '../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 enables a single account to have different data sources within it. A domain/account combination is treated as a unique data source. Domains are useful in limited cases, such as where 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.

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.

NOTE

Note: the `listAccounts` method will list only accounts that belong to your application. It cannot be used to retrieve information about accounts belonging to other applications.

8.3.2. Contacts & Calendar

Both Contacts and Calendar will allow applications to add information that will get 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 device.

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`, which would have had to be saved in the account creation success handler. 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.

8.3.3. People Picker

The People Picker is a special Contacts function to enable applications to retrieve information from any Contacts entry. It won't allow direct access, but 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, you will 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 select

a contact or cancel with a back gesture. After the user selects a contact, the details are returned as an argument to the 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 calculating a trip, for example, or an IM address, or some other private information. You can optionally exclude contacts from the list by enumerating their contact ids.

8.3.4. Email & Messaging

You can launch the Email and Messaging applications to their main view, 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.

8.3.4.1. Back to the News: Share 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 pre-populated 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: 'launch',
        parameters: {
          id: 'com.palm.app.email',
          params: {
            summary: 'Check out this News story...',
            text: this.storyFeed.stories[this.storyIndex].url
          }
        }
      });
      break;
```

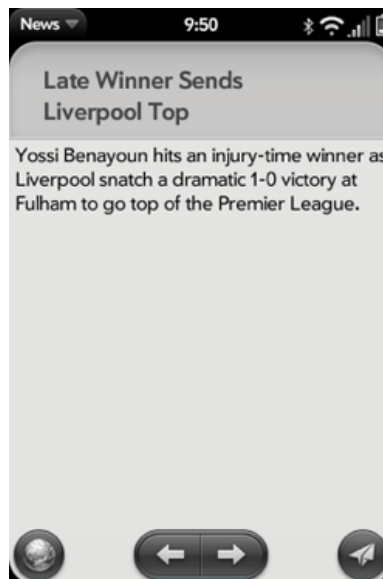
Messaging is very similar; for this example, the entire message is provided in `params.messageText`:

```
case 'do-messageStory':
    this.controller.serviceRequest('palm://com.palm.applicationManager', {
        method: 'launch',
        parameters: {
            id: 'com.palm.app.messaging',
            params: {
                messageText: "Check 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 combined with the browser command that was added earlier give you a scene like that shown in [Figure 8-2](#).

Figure 8-2. Story View with menu buttons for browser view and sharing.



This sample code is used to generate the view shown in [Figure 8-2](#):

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

    this.storyMenuAttr = {
```

```

        items: [
            {iconPath: "images/menu-icon-web.png", command: 'do-webStory'},
            {},
            {items: []},
            {},
            {icon: "send", command: 'do-shareStory'}
        ]];

        if (this.storyIndex > 0) {
            this.storyMenuAttr.items[2].items.push({icon: "back",
command: 'do-viewPrevious'});
        } else {
            this.storyMenuAttr.items[2].items.push({icon: "",
command: '', label: " "});
        }

        if (this.storyIndex < this.storyFeed.stories.length-1) {
            this.storyMenuAttr.items[2].items.push({icon: "forward",
command: 'do-viewNext'});
        } else {
            this.storyMenuAttr.items[2].items.push({icon: "", command:
'', label: " "});
        }

        this.controller.setupWidget(Mojo.Menu.commandMenu, undefined, this.storyMenuAttr)

```

The *web* button is added with a custom menu icon, stored within the *images* directory at the root level of the News directory. The *share* button is the rightmost button and will present a popup when tapped. The Next/Previous button group in center was covered in [Chapter 4](#); they are used to navigate to the next and previous story.

There is already a `commandHandler` included in this scene for the Next and Previous buttons, so just add handlers for the new buttons; for the Share button, add a sub-menu to present the Email and Messaging options.

```

StoryViewAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case 'do-viewNext':
                Mojo.Controller.stageController.swapScene("storyView",
                    this.storyFeed, this.storyIndex+1);
                break;
            case 'do-viewPrevious':
                Mojo.Controller.stageController.swapScene("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.

8.4. Viewers & Players

The media applications can be used to play stream or file-based audio or video content and the Application Manager's `open` method can be used to handle common file types.

8.4.1. 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 earlier Application Manager section, 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.

8.4.2. Audio

The Music player is used to play or stream file or 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 "<http://audio-file>", "<https://audio-file>" or "<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 Appendix B for the Command and Resource Handler Table, which has a complete list of all supported audio file and mime types.

8.4.3. Videos

Similarly, 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 "<http://video-file>", "[https:// video-file](https://video-file)" or "`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 Appendix B for the Command and Resource Handler Table, which has a complete list of all supported video file and mime types.

8.5. Other Applications

The Application Manager service can be used to launch any application not just the core applications described in this chapter. 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 what applications are available and what 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 it were launched 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.

8.6. 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 all 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.

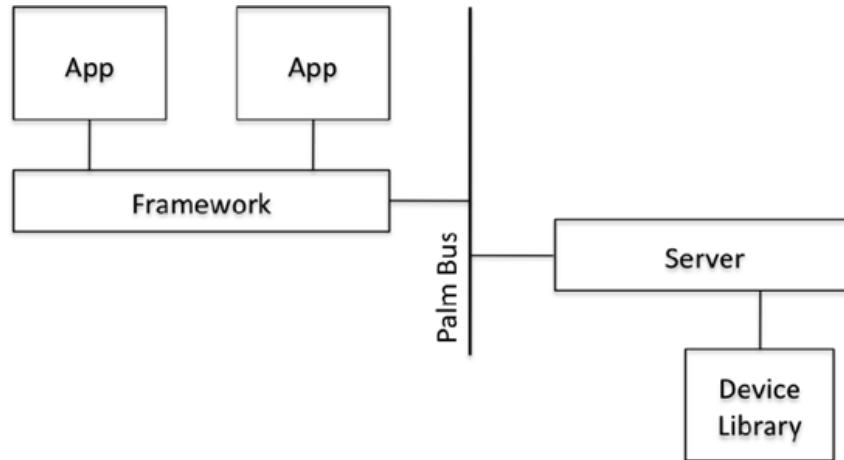
The service architecture is accessed through `Mojo.Service.Request()`, which accepts a service name and a method name to route the request; service requests are always asynchronous operations.

Chapter 9. System & Cloud Services

System services are those that are enabled by hardware features or provided by the Linux operating system. Hardware-enabled services include access to 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 service 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 XMPP-based messaging service for publish/subscribe notifications. It is designed for 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.

9.1. System Services

This section will describe 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 more detail in [Chapter 8](#), but briefly:

- `serviceName`—a URI formatted string uniquely specifying the service name.
- `options`—an object including the designated method, parameters and callback functions.
- `requestOptions`—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 if needed. 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 assistant where the call is made, then you must use `Mojo.Service.Request()`.

Service Subscriptions

Most System services offer subscriptions, the option of receiving service notifications each time there is an update. Set the `subscribe` property to true to register for subscription notification. Subscribed requests will return to the specified `onSuccess` callback with a `subscribed` property set to true, on each update. 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 GPS 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 that a subscription is automatically renewed in the event the service is restarted or some other type of error. Set the argument `requestOptions` to true to enable this renewal feature.

If the service requested fails (for example, the location service is unavailable), the `onFailure` handler will be called. Periodically (random intervals), the

framework will attempt to re-issue the command, until the service comes back up. The `onSuccess` will be called only after the service has been restored.

9.1.1. 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 the example code above, `wakeupFunction` is executed after a delay of 20 seconds. This type of alarms 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 for use to wake applications while minimized, maximized or to drive polling for Dashboard applications. Alarms will:

- Accurately account for time changes; timeouts are accurately tracked across device sleep, time zone changes, manual changes to time settings or other changes that affect the displayed or perceived time on the device.
- Wake up the device from sleep; if needed, these timers will wake up the device when the alarm fires.
- Fire after a delay, or at a specified date and time; alarms can be set for an interval, or to fire at a predetermined 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 common use case is an application using the Alarm service to wake itself up periodically.

```
this.controller.serviceRequest('palm://com.palm.power/timeout', {
    method: 'set',
    parameters: {
        key: 'com.palm.app.news.update',
        in: 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 `feedUpdateInterval` and requests that it wakeup the device if asleep. When the alarm fires, it will use the Application Manager to launch News (even if News isn't launched 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 the [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 AM local time. 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, calendar alarms need to be set up for each occurrence; there isn't a provision at this time for periodic or recurring alarms.

In case you need 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 whether by the user or from an external source (e.g. network time or time zone 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 a reboot. If an alarm expired while the device was shutdown, 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 alarm are coarse-grained alarms, so don't expect millisecond or even 1-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. Alarms will be used to turn News into a background application but we'll wait until [Chapter 10](#) to show those examples.

9.1.2. Connection Manager

Use the Connection Manager's `getStatus` method to get updates on the device connection status. Some applications need to manage their data access based on the presence of a connection or type of connection that is present.

```
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 is changed. 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 9-1](#).

Table 9-1. Response properties for connection status

Name	Description
<code>isInternetConnectionAvailable</code>	Set to <code>true</code> when a connection is present
<code>wifi</code>	Object describing the WiFi connection status, with properties for state ("connected", "disconnected"), <code>ipAddress</code> , <code>ssid</code> & <code>bssid</code>
<code>wan</code>	Object describing the Wide Area Network connection status, with properties for state, <code>ipAddress</code> and type ("unknown", "unusable", "gprs", "edge", "umts", "hsdpa", "1x", "evdo")
<code>btpan</code>	Object describing the Bluetooth Personal Area Network connection status, with properties for state, <code>ipAddress</code> and <code>panUser</code>

9.1.3. Location Services

Palm webOS provides basic location services to get a single or multiple location fixes. Fixes can be specified by mode, either automatic, meaning the system picks the most appropriate mode based upon your accuracy and response time requirements, or by a specific fix type including assisted GPS, Cell ID with or without WiFi ID.

9.1.3.1. Get Current Position

You can get the current position sourced 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 lesser accuracy, faster or slower response times, and agree to accept a cached fix. Typically greater accuracy means a slower response to the request.

Location service tries to get the fix of requested accuracy. If it is not able to get one within the maximum allowed time then it returns the best match from the cache. If there is no entry in the cache then it returns a timeout error.

If there is a fix can be acquired, the `onSuccess` function will be called with a response object with the properties described in [Table 9-2](#).

Table 9-2. Response Properties for Location Service Position Fix

Property	Description
errorCode	Set to zero if the request is successful; a non-zero value otherwise. See Table 9-5 for a complete list of error codes.
timestamp	The time (in milliseconds) when this location fix was created.
latitude	Number representing the latitude in degrees of the location. Valid range: -90.0, 90.0.
longitude	Number representing the longitude in degrees of the location. Valid range: -180.0, 180.0.
horizAccuracy	Horizontal accuracy of the location in meters; if unknown, value is -1
heading	Number representing the compass azimuth in degrees. Valid range 0.0, 360.0; if unknown, value is -1
velocity	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 non-zero `errorCode`. A list of possible errors is provided in [Table 9-3](#).

Table 9-3. Location Error Codes

Error Code	Description
0	Success
1	Timeout
2	Position Unavailable

Error Code	Description
3	Unknown
4	GPS Permanent Error
5	Location Service Off
6	Permission denied: user hasn't agreed to location service terms of use

9.1.3.2. Tracking

Use the `startTracking` method to get a series of fixes. Tracking is effective for navigation applications or anywhere you would 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, then 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.

NOTE

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 on-going tracking fixes through the `onSuccess` callback.

9.1.3.3. 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 semi-colons, where the street address is optional:

```
street address;
locality (eg. in the US, city, state, zipcode);
country
```

For example the code sample above would return the following string:

```
98th 8th St ;San Francisco, CA 94103 ;USA
```

9.1.4. Power Management

The device will automatically go into sleep after a period of inactivity, where inactivity is primarily defined as 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 need to 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 a background application 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 is recommended as your app ID with an activity name and an occurrence count. The 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 or 15 minutes.

The power management service will automatically terminate your activity 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 currently canceled when an application is closed, so you should use `activityEnd()` in your `cleanup()` method when there are any outstanding activity requests.

9.1.5. 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: 'getProperty',
  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 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 than the `nduid`.

9.1.6. 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 time zone:

```
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 9-4](#).

Table 9-4. Response Properties for System Time

Property	Description
<code>localTime</code>	The time for the current time zone in seconds.
<code>offset</code>	Offset from UTC in minutes.
<code>timezone</code>	Current time zone in the "TZ" environment variable format.

JavaScript Date Object

In most cases, you can use the JavaScript Date object to get the current date and time. The `getSystemTime` service method gives you the time zone if you need 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.

9.1.7. System Sounds

The System Sounds service is used to create feedback audio for direct user actions. This might include button or keypad clicks, transition sounds, action audio or any audible response to a direct 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 or false if not. In the false case, the `errorText` property includes an indication the type of error encountered.

The available sounds are enumerated in Appendix B and can be found on the Palm developer site.

9.2. 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 UI and services, but one that includes 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 the Mojo Messaging following these basic steps:

- Create an endpoint, to which a key is returned.
- Share the key with the cloud service from which you need updates.
- Subscribe to the endpoint with a callback to receive messages.
- 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 described in [Table 9-5](#):

Table 9-5. 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 cover them in detail. If you're interested in this class of service or the Messaging service in particular, then you should look to see the latest information at <http://developer.palm.com>.

9.3. Summary

This chapter wrapped up the presentation of the Services available on Palm webOS. We covered the System Services in detail, showing examples using the Location services for retrieving the current position or tracking; checking connection status; setting a wakeup Alarm; getting the device ID and the System's date and time. While few of these are used in the News application, there were specific code samples for most of the service calls.

The chapter finished with an overview of the initial Cloud Service, Mojo Messaging and 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, 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. You've seen a lot about the user experience that enables these features; in this chapter you'll learn how to build your applications to take advantage of them.

Mojo includes a sophisticated notification system, supporting banners and popups to give you options to display information subtly or to get the user's attention with more urgent messages. In this chapter, you'll be introduced to notifications and dashboard panels, with code examples of each to show you how to use them in your application.

Advanced applications are based on 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 enables you to build multi-stage applications with secondary cards or dynamic dashboard stages, and to 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 customize 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 dashboards stages and provide guidelines on the best practices.

10.1. 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 popup notification, or secondary cards for handling specific activities within the application. Refer to email as an example of a multi-stage 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 panel or popup notifications is a separate window, and each window corresponds to a stage, with a stage controller that manages that window. You'll recall that each stage controller has a stack of scene controllers with the topmost scene activated and in view within the stage's window.

We'll show you examples of each feature by building it into the News application as we have done in previous chapters. But before that we'll start with some general information about using stages.

10.1.1. Creating New Stages

All stages are created the same way, with a call to `createStageWithCallback()`, an `AppController` method, and a callback function, which at a minimum will push the first scene using the newly created `stageController`.

```
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:

- Name the stage; the name identifies the stage and you will use the name later to determine if the stage exists or not.
- 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 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` but if not the stage will be created with a default stage assistant.

10.1.2. Using Existing Stages

Often you will want to create a stage only when it doesn't already exist. If the stage exists then 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 would create one. Otherwise, you will 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(stage
Arguments, pushMainScene, "card");
}
```

But `getStageController()` will also return `undefined` when the stage controller has been created but is not available at the time of the call. You will want to 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, what's returned is a proxy for the controller and can't be used as a stage controller.

NOTE

It can take as long as a 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, then you should use `getStageProxy()`.

10.1.3. Working with Stages

The News application uses a single card stage, which is created automatically when launching the application. For most of the advanced features, we will be creating and accessing stages directly and turning News into a multi-stage application. When working with multi-stage applications you should follow these guidelines:

- JavaScript loading through *sources.json* is required; multi-stage applications will fail unless the source files are specified in *sources.json* as loading JavaScript through script tags will not work (other than the required script tag for *mojo.js*).

- 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 *no window* app, creating their own stages, or windows, explicitly.
- No support for Prototype's `$()` function; get elements with methods from the scene or widget controller.

NOTE

Be careful about nested functions, as before they could easily access the global `$()` function, but now will need to use a specific method. You can 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)`.

NOTE

Note that 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.

10.2. Notifications

You can post a *banner notification*, which subtly appears 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 *popup notification*, which slides up out of the notification bar and partially obscures the card view or foreground card. All notifications and dashboards are non-modal, meaning the user can continue to interact with whatever is in the foreground view until they are ready to address the notifications or interact with the dashboard.

10.2.1. 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. By convention the banner is usually accompanied by a dashboard panel, which serves as a reminder of the notification and can provide additional details about the notification.

10.2.1.1. 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](#). Start by modifying the global function `ProcessFeed()` to track new stories:

```
var numUnRead = 0;
var newStoryCount = 0;
var newStory = true;

for (i = 0; i < listItems.length; i++) {
    var storyUnread = unreadFormatting;
    var title = listItems[i].title;
    var j;
    for (j=0; j<feedList[index].stories.length; j++ ) {
        if(listItems[i].url == feedList[index].stories[j].url) {
            storyUnread = feedList[index].stories[j].unreadStyle;
            title = feedList[index].stories[j].title;
            newStory = false;
        }
    }

    if(storyUnread == unreadFormatting) {
        numUnRead++;
    }

    if (newStory) {
        newStoryCount++;
    }

    listItems[i].unreadStyle = storyUnread;
    listItems[i].title = title;
}

feedList[index].stories = listItems;
feedList[index].numUnRead = numUnRead;
feedList[index].newStoryCount = newStoryCount;
```

And add an additional property, *newStoryCount*, to the *feedList* entry:

```
var feedList = []; // News feed list, used throughout the application
//
// feedlist entry is:
// feedList[x].title String Title entered by user
// feedList[x].url String Feed source URL in unescaped form
// feedList[x].type String Feed type: either rdf (rss1),
// rss (rss2) or atom
// feedList[x].value Boolean Spinner model for feed update indicator
// feedList[x].numUnRead Integer How many stories are still unread
// feedList[x].newStoryCount Integer For each update, how many new stories
// feedList[x].stories Array Each entry is a complete story:
// feedList[x].stories[y].title String Story title or headline
// feedList[x].stories[y].text String Story text
// feedList[x].stories[y].summary String Summary for
// list view, stripped of markup
// feedList[x].stories[y].unreadStyle String Style to apply
```

```

    when unread; null when Read
    //      feedList[x].stories[y].url      String      Story url

    //      Push default feeds onto list; these will get overwritten by what's stored in the
    //      database but these will be used otherwise

    feedList.push({title:"Huffington Post",
        url:"http://feeds.huffingtonpost.com/huffingtonpost/raw_feed",
        type:"atom", value:false,
        numUnRead:0, newStoryCount:0, stories:[]});

```

These changes enable us to determine when to post a notification. We'll add the code to actually post the notification in the `feedRequestSuccess()` method:

```

//      If successful processFeed returns errorNone, otherwiset
//      there was a problem with the feed

if (feedError == errorNone) {
    this.feedWgtModel.items = feedList;
    this.controller.modelChanged(this.feedWgtModel);

    var currentFeed = feedList[curFeedIndex];
    // If this is the featured feed and there are stories,
    then start feature rotation
    if (curFeedIndex == featureIndexFeed && featureStoryTimer === null){

        $("featuredFeedLabel").innerHTML = feedList[featureIndexFeed].title;
        this.showFeatureStory();
    }

    // Post a banner notification if there are new stories
    var appController = Mojo.Controller.getAppController();
    if (feedList[curFeedIndex].newStoryCount > 0) {
        var bannerParams = {
            messageText: currentFeed.title+" :
"+currentFeed.newStoryCount+" New Items"
        };
        appController.showBanner(bannerParams, {}, 'newsUpdates');
    }
} else ....

```

Get the App Controller, and then call its `showBanner()` method. This will post a single line of text, which is 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 App Assistants, where you'll see how to use an explicit `handleLaunch()` method in the App 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 banner is tapped. The second argument to `showBanner()` is an empty object representing the launch parameters.

NOTE

You need to provide at least an empty object for the launch parameters when there are no launch parameters. If no object is passed at all, then 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 (5 seconds as of this writing), 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. If you are using banner notifications from different sources then 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.

NOTE

Banners from separate applications are always distinct. Category is only needed when there are multiple banner categories within an application.

10.2.2. Minimized Applications

Generally, you should avoid using notifications when your application is *maximized* meaning it is the foreground card currently displayed on screen. 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 usually use on screen representations that show visible changes to dynamic data, or dialogs 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.

```
// Setup listeners for minimize/maximize events
this.windowState = 'maximized';
var stageDocument = this.controller.stageController.document;
Mojo.Event.listen(stageDocument, Mojo.Event.stageActivate,
    this.activateWindow.bindAsEventListener(this);
Mojo.Event.listen(stageDocument, Mojo.Event.stageDeactivate,
    this.activateWindow.bindAsEventListener(this);

// Set feedsInitialized to false so that activate knows that it's
// being launched or pushed from the background
this.feedsInitialized = false;

// Start the Ajax Request sequence
this.updateFeedList();
};

FeedListAssistant.prototype.activateWindow = function() {
    this.windowState = "maximized";
    this.feedWgtModel.items = feedList;
    this.controller.modelChanged(this.feedWgtModel);

    // If there's some stories in the feature
    IndexFeed, then start the story rotation
    if(feedList[featureIndexFeed].stories.length > 0) {
        $("featuredFeedLabel").innerHTML = feedList[featureIndexFeed].title;
        this.showFeatureStory();
    }
};

FeedListAssistant.prototype.deactivateWindow = function() {
    this.windowState = 'minimized';
    this.clearTimers();
};
```

Now you can add another condition before posting a banner notification in the `feedRequestSuccess()` method:

```
// Post a banner notification if there are new stories
var appController = Mojo.Controller.getAppController();
if ((feedList[curFeedIndex].newStoryCount > 0)
    && (this.windowState == 'minimized')) {
    var bannerParams = {
        messageText: currentFeed.title+" : "+currentFeed.
newStoryCount+" New Items",
        soundClass: "alerts"
    };
    appController.showBanner(bannerParams, {}, 'newsUpdates');
}
```

With these changes, News will not post banner notifications when maximized, only when minimized. This is the recommended behavior for most applications.

10.2.3. Popup Notifications

Use a Popup 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 popup notifications and you'll see it when you connect your phone to a computer using USB.

Popup notifications shouldn't be used very often, as they are disruptive by design, taking up as much as half the screen. You will generate a popup by creating a stage and pushing a popup 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');
```

Popup stages take the same stage arguments as the card and dashboard stages examples, with an optional property `height` property. Indicate that this is a popup stage through use of `'popupalert'` as the last argument. As in the general case, you specify the stage name and `lightweight` properties and include the callback function as the second argument.

Specific to popups is the option to set the `height` property. The default popup height is 200 pixels on the Palm Pre, 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 requires a different height.

You will customize the popup 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 popup 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">
        <button class="palm-button affirmative popupbutton" id="addButton">Ok</button>
        <button class="palm-button negative popupbutton" id="close
Button">Close</button>
    </div>
</div>

```

The attribute `x-palm-popup-content=""` is used within the popup scene div which contains the main content for the popup. The System UI will use this special attribute to draw only the main content area of the popup in the lock screen when the display is turned on from sleep. The popup's buttons should be kept outside of this div, as they are not actionable when the device is locked.

And a template for rendering the variable content, named *popup/item-info.html*:

```

<div id='notification-title' class="notification-title"> #{subject}</div>
<div id='notification-subtitle' class="notification-subtitle">#{eventSubtitle}</div>

```

Figure 10-2. A popup notification.



You can see that we're using a lot of specific notification style classes, to layout the popup scene to follow the System UI conventions and to style the title and subtitle text. The framework doesn't load some of common styles used with cards with dashboard or popup stages, so you may have to create your own in those cases. The styles in this popup 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:


```

/* Popup 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 popup's scene assistant is a conventional scene assistant. In this case, we've created *popup-assistant.js* in the *assistants* directory:

```

function PopupAssistant(message) {
    this.message = message;
}

PopupAssistant.prototype.setup = function() {
    this.update(this.message);

    var okButton = this.controller.get("okButton");

```

```

    Mojo.Event.listen(okButton, Mojo.Event.tap, this.handleOk
    .bindAsEventListener(this));

    var closeButton = this.controller.get("closeButton");
    Mojo.Event.listen(closeButton, Mojo.Event.tap, this.handleClose.
    bindAsEventListener(this));
};

PopupAssistant.prototype.update = function(message) {
    this.info = {eventSubtitle: message, subject: "News"};

    // 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'});

    // Insert the HTML into the DOM, replacing the existing content. */
    var infoElement = this.controller.get('info');
    infoElement.update(renderedInfo);
};

PopupAssistant.prototype.handleOk = function(){
    Mojo.Log.info("Popup Ok received");
};

PopupAssistant.prototype.handleClose = function(){
    this.controller.window.close();
};

```

You can do most anything within a popup scene that you can do in any other scene, but you should limit your actions to simple messages and selections. Your popup assistant must close the window on exit to close the stage and remove it from the display.

Updating Popups and Dashboard Panels

Popup 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 popup notification shows the next event on the calendar, but if that event passes while the notification is on the screen, then a new calendar event should replace it.

You should prepare for updates by structuring your popup assistant with a method to refresh the content. Review the sample code under Popup Notifications for an example using an `update()` method, but you can name the method whatever you'd like.

Before creating the popup stage, check to see if 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. And 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 where the stage is being created and the stage controller doesn't yet exist.

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.

10.3. Dashboards

You will usually create a *dashboard panel* following a *banner notification* posting, as a reminder of the notification. Dashboards can also be used to 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 by UI convention, Dashboard panels 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.

NOTE

The full framework CSS is not automatically loaded when creating Popup or Dashboard stages. Currently some of the styles that you can use in a card stage or main application window are not available within non-card 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 20% of the screen height in portrait mode; on the Palm Pre that is 320 pixels wide and 48 pixels high. But just as with card windows, you should layout your dashboard windows to be able to

handle different widths for landscape modes or different screen sizes on future Palm webOS devices.

10.3.1. Back to the News: Adding a Dashboard Stage

You can create a dashboard stage similar to the general stage example showed earlier in this chapter, but you will declare it as a dashboard stage type. We'll add a dashboard to News in `feedRequestSuccess()` by following the banner notification with a call to `createStageWithCallback()` passing a callback function that pushes *dashboard-assistant.js* with the current feed title, the new story count and the most recent new story headline. We'll also pass a global constant, *DashboardStageName*, which defines the dashboard stage name, set the *lightweight* property to true and specify this as a *dashboard* stage.

```
// Post a banner notification and create or update Dashboard if there are new stories
var appController = Mojo.Controller.getAppController();
var dashboardStageController = appController.getStageProxy(DashboardStageName);

if ((feedList[curFeedIndex].newStoryCount > 0) && (this.window
State == 'minimized')) {
    var bannerParams = {
        messageText: currentFeed.title+" : "+currentFeed.newStoryCount+" New Items",
        soundClass: "alerts"
    };
    appController.showBanner(bannerParams, {}, 'newsUpdates');

    // Create or update dashboard
    var title = feedList[curFeedIndex].title;
    var message = feedList[curFeedIndex].stories[0].title;
    var count = feedList[curFeedIndex].newStoryCount;

    if(dashboardStageController) {
        dashboardStageController.delegate
ToSceneAssistant("updateDashboard", title,
        message, count);
    }
    else {
        var pushDashboard = function(stageController){
            stageController.pushScene('dashboard', title, message, count);
        };
        appController.createStageWithCallback({name:
DashboardStageName, lightweight: true},
        pushDashboard, 'dashboard');
    }
}
```

Before creating the stage, call `getStageProxy(DashboardStageName)`, and if the stage exists the proxy will be defined. Call the dashboard assistant's `updateDashboard()` method with the feed title, new story count and story title to use the existing dashboard and update its contents.

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 *Popup Notifications* to render scenes and handle updates:

```
function DashboardAssistant(title, message, count) {
    this.title = title;
}
```

```

        this.message = message;
        this.count = count;
    }

    DashboardAssistant.prototype.setup = function() {
        this.updateDashboard(this.title, this.message, this.count);
    };

    // Update scene contents
    DashboardAssistant.prototype.updateDashboard = function(title, message, count) {

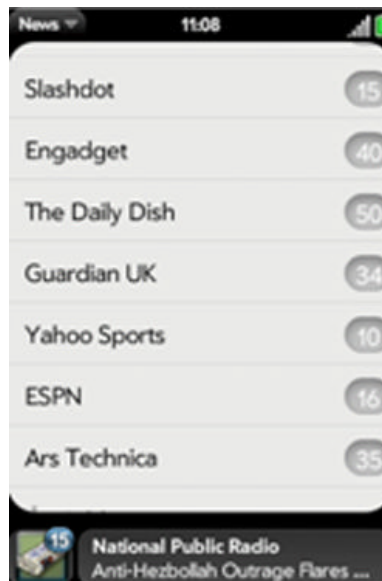
        var info = {title: title, message: message, count: count};

        // 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: info, template:
        'dashboard/item-info'});
        var infoElement = this.controller.get('dashboardinfo');
        infoElement.update(renderedInfo);
    };

```

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 to the right.

Figure 10-3. A dashboard panel.



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`), 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"></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>
```

These are all supported framework styles (most `dashboard` named styles are loaded by the framework), but we'll override the `palm-dashboard-icon` in *News.css* to show the News icon, which is located in the News application's *images* directory.

```
.palm-dashboard-icon {
  background: url(..images/dashboard-icon-news.png);
}
```

This is a simple notification reminder, but dashboard panels can be fully dynamic application views. You can provide ambient information that is intermittently accessed, like weather, stocks, baseball scores, or any information that a user wants to track and occasionally will tap to get more details. They 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.

Dashboard panels can be used 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 on Background Applications.

10.3.2. 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.

```
DashboardAssistant.prototype.setup = function() {
  this.updateScene();
  var switchButton = this.controller.get("dashboardinfo");
  Mojo.Event.listen(switchButton, Mojo.Event.tap,
```

```

this.launchMain.bindAsEventListener(this));

var stageDocument = this.controller.stageController.document;
Mojo.Event.listen(stageDocument, Mojo.Event.stageActivate,
    this.activateWindow.bindAsEventListener(this));
Mojo.Event.listen(stageDocument, Mojo.Event.stageDeactivate,
    this.deactivateWindow.bindAsEventListener(this));

// Update scene contents
DashboardAssistant.prototype.updateScene = function() {
    var info = {message: this.message, count: this.count};
    var renderedInfo = Mojo.View.render({object: info, template:
        'dashboard/item-info'});

    var infoElement = this.controller.get('dashboardinfo');
    infoElement.update(renderedInfo);
};

DashboardAssistant.prototype.activateWindow = function() {
    Mojo.Log.info("..... Dashboard Assistant - Activate Window");
};

DashboardAssistant.prototype.deactivateWindow = function() {
    Mojo.Log.info("..... Dashboard Assistant - Deactivate Window");
};

DashboardAssistant.prototype.launchMain = function() {
    this.controller.serviceRequest('palm://com.palm.applicationManager',
        {
            method: 'open',
            parameters: {
                id: 'com.palm.app.news',
                params: {}
            }
        });
    this.controller.window.close();
};

```

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, "Advanced Applications," we'll cover the handling launch requests in the application assistant. In that case, you wouldn't use the application manager service but would call the `handleLaunch()` method of the application controller.

10.4. 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 maintains a dashboard panel for events while not in focus or while in the background.
- Schedule wakeup requests through the Alarm service and handles 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.

10.4.1. Back to the News: App Assistant

This chapter began with a list of guidelines for developing multi-stage 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:

- Replace instances of `$()` with `this.controller.get()`; in several places, we had been using the `$()` function to get an element, so we'll replace them as with this instance in *storyView-assistant.js*:

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

- Remove use of the global window object; change `window.setInterval()` to `this.controller.window.setInterval()`.
- Use the local controller's `stageController` methods; instead of `Mojo.Controller.stageController` methods for `pushScene`, `swapScene` as in this example in *storyView-assistant.js* in the `handleCommand` method:

```
case 'do-viewNext':
    this.controller.stageController.swapScene("storyView",
        this.storyFeed, this.storyIndex+1);
    break;
```

Next, add the `noWindow` property to *appinfo.json*:

```
{
  "title": "News",
  "type": "web",
  "main": "index.html",
  "icon": "icon.png",
  "id": "com.palm.app.news10-3",
  "version": "1.0",
  "vendor": "Palm",
  "logLevel": 99,
  "timingEnabled": true,
  "noDeprecatedStyles": "true",
  "theme": "light",
  "noWindow": "true"
}
```


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.

```
var MainStageName = "newsStage";           // Main Card stage name
var DashboardStageName = "newsDashboard";   // Dashboard stage name

.
.
.

AppAssistant.prototype.handleLaunch = function (launchParams) {
  var cardStageProxy = this.controller.getStageProxy(MainStageName);
  var cardStageController = this.controller.getStageController(MainStageName);
  var appController = Mojo.Controller.getAppController();
  var dashboardStage = this.controller.getStageProxy(DashboardStageName);

  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("..... News - Main Stage Exists");
      cardStageController.popScenesTo("feedList");
      cardStageController.window.focus();
    } 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');
      };
      var stageArguments = {name: MainStageName, lightweight: true};
      this.controller.createStageWithCallback(stageArguments,
        pushMainScene.bind(this), "card");
    }
  }
};

// handleCommand - Setup handlers for menus:
//
.
.
.
};
```

10.4.2. 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 `AppAssistant`'s `setup` method. Launch requests are made implicitly by:

- 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. As an 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() {
    var appController = Mojo.Controller.getAppController();
    appController.assistant.handleLaunch({action: "notification"});
    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.

```
AppAssistant.prototype.handleLaunch = function (launchParams) {
    var cardStageProxy = this.controller.getStageProxy(MainStageName);
    var cardStageController = this.controller.getStageController(MainStageName);
    var appController = Mojo.Controller.getAppController();
    var dashboardStage = this.controller.getStageProxy(DashboardStageName);

    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.
            cardStageController.window.focus();
        } 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');
            };
            var stageArguments = {name: MainStageName, lightweight: true};
            this.controller.createStageWithCallback(stageArguments,
                pushMainScene.bind(this), "card");
        }
    }

    else {
        switch (launchParams.action) {

            // NOTIFICATION
            case 'notification' :
                Mojo.Log.info("..... News -- Notification Tap");
                if (cardStageController) {
```

```

        // If it exists, bring the feedList scene to the front by
        // popping any other scenes and focusing the stage window.
        cardStageController.popScenesTo("feedList");
        cardStageController.window.focus();
    } 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 pushMainScene2 = function(stageController) {
            stageController.pushScene('feedList');
        };
        Mojo.Log.info("..... News - Create Main Stage");
        var stageArguments2 = {name: MainStageName, lightweight: true};
        this.controller.createStageWithCallback(stageArguments2,
            pushMainScene2.bind(this), "card");
    }
    break;
}
}
};

```

10.4.3. Back to the News: Handling `feedList` updates in the AppAssistant

One major benefit of structuring your application around an application assistant is that you can use it for global tasks or tasks that are independent of the user interface. Launch handling is a specific case of this, but now we'll apply it in a more significant way by putting the database and `feedList` updates in the application assistant. Once this is done, we can take the final step of enabling background updates.

Previously, both the database handling and the `feedList` updates were built into the `feedList` scene. We'll move the code pretty much intact from *feedList-assistant.js* to *app-assistant.js*, so that the application assistant now looks like this:

```

function AppAssistant () {
}

AppAssistant.prototype.setup = function() {
    this.cookie = new Mojo.Model.Cookie("comPalmAppNewsPrefs");
    var oldNewsPrefs = this.cookie.get();
    if (oldNewsPrefs && oldNewsPrefs.newsVersionString) {
        // If current version, just update globals & prefs
        if (oldNewsPrefs.newsVersionString == newsVersionString) {
            featureIndexFeed = oldNewsPrefs.featureIndexFeed;
            featureFeedEnable = oldNewsPrefs.featureFeedEnable;
            featureStoryInterval = oldNewsPrefs.featureStoryInterval;
            feedUpdateInterval = oldNewsPrefs.feedUpdateInterval;
            newsVersionString = oldNewsPrefs.newsVersionString;
        }
    }

    this.cookie.put({
        featureIndexFeed: featureIndexFeed,
        featureFeedEnable: featureFeedEnable,
        feedUpdateInterval: feedUpdateInterval,
        featureStoryInterval: featureStoryInterval,
        newsVersionString: newsVersionString
    });

    //
    // Setup App Menu with preferences entry
    //

    newsMenuAttr = {omitDefaultItems: true};

```

```

newsMenuModel = {
    visible: true,
    items: [
        {label: "About News...", command: 'do-aboutNews'},
        Mojo.Menu.editItem,
        { label: "Preferences...", command: 'do-newsPrefs' },
        Mojo.Menu.helpItem
    ]
};

// Open the database to get the most recent feed list
// DEBUG - replace is true to recreate
databases every time; switch to false for release
//
db = new Mojo.Depot(
    {name:"feedDB", version:1, estimatedSize:100, replace: false},
    this.dbOpenOk.bind(this),
    function(transaction, result) {
        Mojo.Log.warn("Can't open feed database (#", result.message,
            "). All feeds will be reloaded.");
        Mojo.Controller.errorDialog("Can't
open feed database (#" + result.message + ").");
    }
);

};

// -----
// Database Functions

// dbOpenOK - Callback for successful DB request in setup. Does two things:
// Get feedlist from DB: call updateList if successful,
// otherwise useDefaultList for default set of feeds
//

AppAssistant.prototype.dbOpenOk = function() {

    Mojo.Log.info(".....","Database opened OK");
    db.simpleGet("feedList", this.updateList
    .bind(this), this.useDefaultList.bind(this));
};

// updateList - Callback for successful DB retrieval of feedlist. Call
// useDefaultList if the feedlist empty or null or initiate an update
// to the list by calling updateFeedList.
//
AppAssistant.prototype.updateList = function(fl) {

    Mojo.Log.info(".....","database size: " , Object.values(fl).size());

    if (Object.toJSON(fl) == "{}" || fl === null) {
        Mojo.Log.warn("Retrieved empty or null list from DB");
        featureIndexFeed = 0;
        this.useDefaultList();
    } else {
        Mojo.Log.info(".....","Retrieved feedlist from DB");
        feedList = fl;
        this.updateFeedList();
    }
};

// useDefaultList - Callback for
// failed DB retrieval meaning no list so use default.
//

AppAssistant.prototype.useDefaultList = 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.
    //
    db.simpleAdd("feedList", feedList,

```

```

        function() {Mojo.Log.info(".....","feedList saved OK");},
        function(transaction,result) {
            Mojo.Log.warn("Database save error (#", result.message,
                ") - can't save feed list. Will need to reload on next use.");
            Mojo.Controller.errorDialog("Database
save error (#" + result.message + ")");
        });

        Mojo.Log.warn("Database has no feed list. Will use default.");
        this.updateFeedList();
    };

AppAssistant.prototype.activateWindow = function() {
    Mojo.Log.info("..... News App Assistant - Activate Window");
    this.windowState = 'maximized';
};

AppAssistant.prototype.deactivateWindow = function() {
    Mojo.Log.info("..... News App Assistant - Deactivate Window");
    this.windowState = 'minimized';
};

AppAssistant.prototype.handleLaunch = function (launchParams) {
    Mojo.Log.info("..... News -- ReLaunch");

    var cardStageProxy = this.controller.getStageProxy(MainStageName);
    var cardStageController = this.controller.getStageController(MainStageName);
    var appController = Mojo.Controller.getAppController();
    var dashboardStage = this.controller.getStageProxy(DashboardStageName);

    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("..... News - Main Stage Exists");
            cardStageController.window.focus();
        } 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');
                Mojo.Event.listen(stageController
.document, Mojo.Event.stageActivate,
                    this.activateWindow.bind(this));
                Mojo.Event.listen(stageController
.document, Mojo.Event.stageDeactivate,
                    this.deactivateWindow.bind(this));
                this.windowState = 'maximized';
            };
            Mojo.Log.info("..... News - Create Main Stage");
            var stageArguments = {name: MainStageName, lightweight: true};
            this.controller.createStageWithCallback(stageArguments,
                pushMainScene.bind(this), "card");
        }
    }
    else {
        switch (launchParams.action) {

            // NOTIFICATION
            case 'notification' :
                Mojo.Log.info("..... News -- Notification Tap");
                if (cardStageController) {
                    // If it exists, bring the feedList scene to the front by
                    // popping any other scenes and focusing the stage window.
                    cardStageController.popScenesTo("feedList");
                    cardStageController.window.focus();
                } 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 pushMainScene2 = function(stageController) {
            stageController.pushScene('feedList');
            Mojo.Event.listen(stageController.
document, Mojo.Event.stageActivate,
            this.activateWindow.bind(this));
            Mojo.Event.listen(stageController.
document, Mojo.Event.stageDeactivate,
            this.deactivateWindow.bind(this));
            this.windowState = 'maximized';
        };
        Mojo.Log.info("..... News - Create Main Stage");
        var stageArguments2 = {name: MainStageName, lightweight: true};
        this.controller.createStageWithCallback(stageArguments2,
            pushMainScene2.bind(this), "card");
    }
    break;
}
};

//    handleCommand - Setup handlers for menus:
//

AppAssistant.prototype.handleCommand = function(event) {
    var stageController = this.controller.getActiveStageController();
    var currentScene = stageController.activeScene();
    if(event.type == Mojo.Event.command) {
        switch(event.command) {

            case 'do-aboutNews':
                currentScene.showAlertDialog({
                    onChoose: function(value) {},
                    title: "News - v1.0",
                    message: "Copyright 2008-2009, Palm Inc.",
                    choices:[
                        {label: "OK", value:""}
                    ]
                });
                break;
            case 'do-newsPrefs':
                stageController.pushScene("preferences", this.cookie);
                break;
        }
    }
};

// -----
// Feed Update Functions

//    updateFeedList - called to cycle through the feeds either on activate or
//    after the interval timer has fired. This is split from feedRequest
//    so that the latter can be called on each feed. This is called once per
//    update cycle.
//

AppAssistant.prototype.updateFeedList = function() {
    // request fresh copies of all stories
    curFeedIndex = 0;
    curFeedSource = feedList[curFeedIndex];
    this.feedRequest(curFeedSource);
};

//    feedRequest - function called to setup and make a feed request
//
//    Uses prototype's Ajax.Request and sets up callback routines:
//    onSuccess      feedRequestSuccess
//    onFailure      feedRequestFailure
//

AppAssistant.prototype.feedRequest = function(curFeed) {
    Mojo.Log.info(".....","URL Request: ", curFeed.url);

```

```

        // Turn on indicator feed update

var request = new Ajax.Request(curFeed.url, {
    method: 'get',
    evalJSON: 'false',
    onSuccess: this.feedRequestSuccess.bind(this),
    onFailure: this.feedRequestFailure.bind(this)
});

};

//    feedRequestFailure
//
//    Callback routine from a failed AJAX feed request (feedRequest);
//    post a simple failure error message with the http status code.
//
AppAssistant.prototype.feedRequestFailure = function(transport) {

    //    Use the Prototype template object to
    generate a string from the return status.
    //
    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 (", m, "). Please check URL and retry");
    Mojo.Controller.errorDialog("Invalid feed - http failure (" + m + ").");

    // Turn off feed update indicator

};

//    feedRequestSuccess
//
//    Callback routine from a successful AJAX
feed request (feedRequest); use globals:
//    curFeedIndex          Index for the feed being updated
//    feedList              Holds the processed feed
//

AppAssistant.prototype.feedRequestSuccess = function(transport) {

    var    feedError = errorNone;
    //    Return variable for processing feed

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

    //    DEBUG - Work around due to Ajax XML error
    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, identifying the current feed index
    feedError = ProcessFeed(transport, curFeedIndex);

    // If successful processFeed returns error
    None, otherwiset there was a problem with the feed

    if (feedError == errorNone) {
        // Post a banner notification and create
        or update Dashboard if there are new stories
        var currentFeed = feedList[curFeedIndex];
        var appController = Mojo.Controller.getAppController();
    }

```

```

        var dashboardStageController = appController
        .getStageProxy(DashboardStageName);

        if ((feedList[curFeedIndex].newStoryCount
> 0) && (this.windowState == 'minimized')) {
            var bannerParams = {
                messageText: currentFeed.title+"
: "+currentFeed.newStoryCount+" New Items",
                soundClass: "alerts"
            };
            appController.showBanner(bannerParams,
{action: 'notification'}, 'newsUpdates');

            // Create or update dashboard
            var title = feedList[curFeedIndex].title;
            var message = feedList[curFeedIndex].stories[0].title;
            var count = feedList[curFeedIndex].newStoryCount;

            if(dashboardStageController) {
                dashboardStageController.delegate
ToSceneAssistant("updateDashboard", title,
                    message, count);
            }
            else {
                var pushDashboard = function(stageController){
                    stageController.pushScene('dash
board', title, message, count);
                };
                appController.createStageWithCallback({
                    name: DashboardStageName,
                    lightweight: true
                },
                pushDashboard,
                'dashboard'
            );
        }
    }
    else
    {
        // There was a feed process error; unlikely, but could happen
        if (feedError == errorUnsupportedFeedType) {
            Mojo.Log.info("Feed ", this.nameModel.value, " unsupported feed type");
        }
    }

    // Turn off feed update indicator

    // Increment the index. If this is NOT the last feed then update
    // the feedsource and request to retrieve the next feed
    curFeedIndex++;
    if(curFeedIndex < feedList.length) {
        curFeedSource = feedList[curFeedIndex];

        // Turn on indicator feed update

        this.feedRequest(curFeedSource);
    } else {

        // Otherwise, this update is done. Reset index to 0 for next update; if
        // the timer is null, the set the interval to feedUpdateInterval
        curFeedIndex = 0;
        feedListChanged = true;

        // Set a timer to update the feeds after the prescribed interval,
        // if it's not already running
        if (feedUpdateTimer === null) {
            feedUpdateTimer = this.controller.window
.setInterval(this.updateFeedList.bind(this),
                feedUpdateInterval);
        }
    }
};

```


You'll notice a few other changes to the assistant code. We added listeners to track the minimize and maximize state of the `MainStageName` window; calling `Mojo.Event.Listen()` as part of the stage create callback. The app assistant's `windowState` property determines whether a banner notification and dashboard are generated after new stories are retrieved.

Also we removed the updates to the spinner models through `this.controller.modelChanged()`. You aren't able to directly update the models but instead need a way of communicating within an application between assistants when there is an update in progress and allowing the scene assistant to decide how best to show it.

10.4.4. Sending and Considering Notifications

To facilitate communication between assistants, the Mojo framework supports a localized 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.

```
var params = {type: 'update', update: true, feedIndex: curFeedIndex };
Mojo.Controller.getAppController().sendToNotificationChain(params);
```

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'`) and the index of the affected feed (`feedIndex : curFeedIndex`):

Scenes can receive notifications by adding 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 specific feed's spinner model with the state of the update.
FeedListAssistant.prototype.considerForNotification = function(params){
    if (params && (params.type == 'update')) {
        this.feedWgtModel.items = feedList;
        var spinnerModel = feedList[params.feedIndex]
        spinnerModel.value = params.update;
        this.controller.modelChanged(this.feedWgtModel);

        // If stories exist in the featureIndexFeed, then start the rotation
        // if not already started
        var currentFeed = feedList[featureIndexFeed];
        if ((featureIndexFeed < currentFeed.stories.length) &&
            (featureStoryTimer === null)) {
            var featureLabelElement = this.controller.get("featuredFeedLabel");
            this.controller.update(featureLabelElement, currentFeed.title);
            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, will update the feedList, and will start the feature story timer if needed.

In *storyList-assistant.js*, it's used to look for changes to the displayed feed:

```

// considerForNotification - called by the framework
// 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;
};

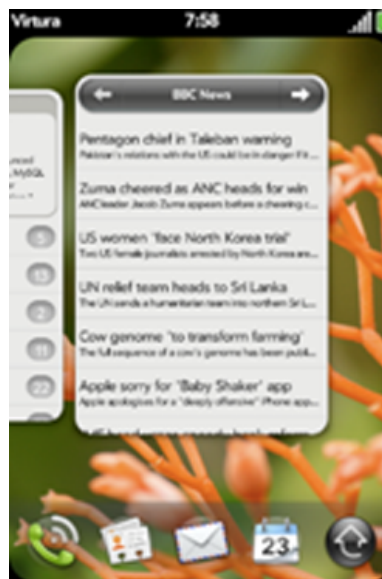
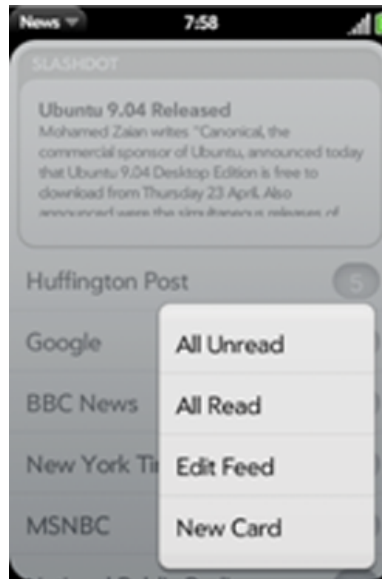
```

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 app 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.

10.4.5. Back to the News: Creating Secondary Cards

In this last example, we'll create a secondary card stage, by adding the option to push a single feed into it's own card. Add a **New Card** item to the popup 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 that feed into a new card. [Figure 10-4](#) shows both 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':
    var newCardStage = "newsCard"+this.popupIndex;
    var cardStage = this.appController.getStageController(newCardStage);
    var feedIndex = this.popupIndex;
    if(cardStage) {
        cardStage.popScenesTo();
        cardStage.pushScene('storyList', feedIndex);
        cardStage.activate();
    } else {
        var pushStoryCard = function(stageController){
```

```

        stageController.pushScene('storyList', feedIndex);
    };
    this.appController.createStageWithCallback({name: newCardStage,
    lightweight: true},
        pushStoryCard, 'card');
    }
    break;

```

The stage name must be unique unless you plan to re-use 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. Note that when re-using the stage, `popScenesTo()` is used with `pushScene` and `activate` to maximize the stage with the `storyList` scene.

10.5. Background Applications

We've done most of the work to make a background application, and in some ways, News already is running in the background. Background applications could be considered to be any of these types:

- *Minimized application* is an application that has a card stage (or window) but is not in the foreground view.
- *Dashboard application* is an application that has a dashboard stage (or panel) but no card stage.
- *Background application* is an application that 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 can and 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 `setInterval()` 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 `setInterval()` timer with an Alarm set at the end of the `feedRequestSuccess()` method in *app-assistant.js*:

```

var wakeupSuccess = function(response){
    Mojo.Log.info("Alarm Set Success", response.returnValue);
    wakeupTaskId = Object.toJSON(response.taskId);
};

var wakeupFailure = function(response){
    Mojo.Log.info("Alarm Set Failure", response.returnValue, response.errorText);

```

```
};

this.wakeupRequest = new Mojo.Service.Request('palm://com.palm.power/timeout', {
  method: 'set',
  parameters: {
    'key': 'com.palm.app.news.update',
    'in': feedUpdateInterval,
    'wakeup': 'true',
    'uri': 'palm://com.palm.applicationManager/open',
    'params': {
      'id': 'com.palm.app.news',
      'params': {'action': 'feedUpdate'}
    }
  },
  onSuccess: wakeupSuccess,
  onFailure: wakeupFailure
});
```

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 `feedUpdateInterval` and requested that the alarm wake the device by setting the `wakeup` property to `true`.

NOTE

The Alarm service will not accept any relative alarm values of less than 5 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' :
    this.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.

NOTE

A background application that receives an alarm when the device is asleep will have less than 5 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, Alarms](#), 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. So, we will change the update process to update as many feeds as possible in each cycle with two changes:

- Modify the `updateFeedList()` method to start with the next feed instead of always starting with the first feed.
- Set up the alarms at the beginning of the cycle rather than the end, so we can just let the update process fail at whatever point the device is powered down. If the full update cycle completes, which it will when the device is awake, then it will be stopped at that point.

The alarm setup is moved into the application assistant's `setup()` method and in the `feedUpdate` action handler:

```
// UPDATE FEEDS
case 'feedUpdate' :
    // Set up next timeout alarm
    this.wakeupRequest = new Mojo.Service.Request('palm://com.palm.power/timeout', {
        method: 'set',
        parameters: {
            'key': 'com.palm.app.news.update',
            'in': feedUpdateInterval,
            'wakeup': 'true',
            'uri': 'palm://com.palm.applicationManager/open',
            'params': {
                'id': 'com.palm.app.news',
                'params': {'action': 'feedUpdate'}
            }
        },
        onSuccess: function(response){
            Mojo.Log.info("..... News - Alarm Set Success", response.returnValue);
            wakeupTaskId = Object.toJSON(response.taskId);
        },
        onFailure: function(response){
            Mojo.Log.info("..... News - Alarm Set Failure", response.returnValue,
                response.errorText);
        }
    });

    this.updateFeedList();
    break;
```

Lastly, 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 5 minutes to 1 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 on or off notifications, although when off the feed updates are still being done, just not with 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 A](#).

Guidelines for Background Applications%

Background applications are very powerful, but they can also easily overuse resources and hurt the user experience. Most of the 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 as 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.
- Avoid waking up the device whenever possible.
- Always give the user the option of not waking the device; they should be able to run the application, 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 app so that it doesn't run indefinitely.
- Conserve power; poll as infrequently as possible. Space out your requests and implement degrading intervals that lengthen the longer 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 that the device goes into mass storage mode and restart the task in that case. 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.

10.6. Summary

Whether you're interested in building an advanced application or just want to add notifications to a basic application, this chapter covered some essential topics. There was a broad review of advanced multi-stage 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 learned 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 last section, we'll cover some of the Internationalization APIs available in Mojo.

Users are able to switch languages and regions at runtime using a Language preferences application, shown in [Figure 11-1](#). You are able to select from any of the languages and any of the regions, enabling you to create any locale formed by those combinations.

Figure 11-1. A language preference application.

The system can't 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.

11.1. 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.

The choice of language indicates the primary localization; while the regional settings govern date formats, number formats and similar types of data representation. A complete list of supported languages and keyboard mappings is provided in [Table 11-1](#), with some of the more common locales and regions. You can mix any language and any region to create *en_DE* for example for an English language system but German regional settings.

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

Locale	Language	Region	Keyboard
es_ES	Spanish (es)	Spain (ES)	QWERTY
es_MX	Spanish (es)	Mexico (MX)	QWERTY
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.

11.1.1.

11.1.1.1. Character Sets and Fonts

The initial Palm webOS devices will ship with a Latin-1 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)

Additional fonts are included to support conventional browser content. The browser fonts support the character sets name above plus all the characters used for Japanese, Chinese, Korean and Vietnamese.

11.1.1.2. 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 you can expect that while some form of keyboard or input method support is provided, you cannot expect a minimum configuration.

Global Applications

For many developers, creating a global application means localizing your application to the locales that it will be used in 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 is quite different than 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.

11.2. 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, then the base version will be selected.

To localize your application, you will 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 if found 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 would do a test localization with a pseudo-locale, to confirm that you have all the strings identified and that your application structure is correct.

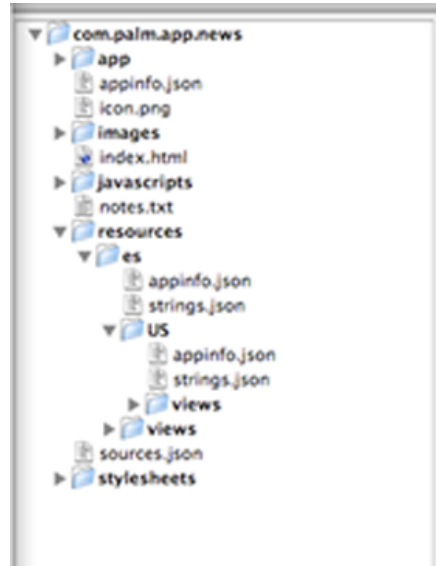
You can localize to multiple languages in parallel, but once you've done your localization, you will need to manage any code or UI changes. This may be difficult under the current structure, as the tools don't address change management, forcing you to manually migrate all changes.

It's good to do a pseudo-language localization early, to test localization readiness, but wait on actual localization until your UI is final or very close to it.

11.2.1. 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 and HTML 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 language directories and at the second level you'll find region directories. If the locale is set to *es_US*, then the framework will look for content in the *resources/es/US* directory first. If that directory is not there or the particular content (e.g. *appinfo.json*, specific strings in *strings.json*, html files in */views*) is not there, it will look for *resources/es* and then default to the *app* directory contents.

For any given language, there can be zero or more region directories. If the region directory is missing, then the framework will still use the content in the languages directory when that language has been selected as part of the current locale no matter which region is selected. In the example shown in [Figure 11-2](#), if the *es_ES* were the current locale then the framework will look in *resources/es* for content before defaulting to the *app* directory contents.

This means that you do need to put the localized content in the languages directory, which in the [Figure 11-2](#) example is in *resources/es*. You may not need to put content into the regions directories at all.

You can create this structure manually, by simply 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 their 2-letter ISO 639 language code, and all of the region names are based on their 2-letter ISO 3166 country codes.

At the time that this is being written, 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.

11.2.2. appinfo.json

Each locale directory will include a copy of the main *appinfo.json* file, modified for the 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/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",
  "logLevel": 99,
  "timingEnabled": true,
  "noDeprecatedStyles": "true",
  "theme": "light",
  "noWindow": "true"
}
```

We only localized the application name, but you could also:

- Change the icon by pointing to a different icon image file should that be appropriate. 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 for cultural reasons.

11.2.3. JavaScript Text Strings

Any JavaScript string can be localized. You should localize all text strings that are displayed and are visible to the user, but you should not localize strings that never appear, such as 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 with just variable definitions.

11.2.3.1. 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 AppMenu attributes and model are set up:

```
// Setup App Menu with preferences entry
//
newsMenuAttr = {omitDefaultItems: true};

newsMenuModel = {
  visible: true,
```

```

    items: [
      {label: $L("About News..."), command: 'do-aboutNews'},
      Mojo.Menu.editItem,
      { label: $L("Preferences..."), command: 'do-newsPrefs' },
      Mojo.Menu.helpItem
    ]
  };

```

Or in this case, from *preferences-assistant.js*, where the list selector widget is setup:

```

//      Setup list selector for feed 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 : feedUpdateInterval
});

```

In this next example, the strings 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.message,
  "). All feeds will be reloaded.");
Mojo.Controller.errorDialog($L("Add Feed DB save error: ") + result.message);

```

Use templates to avoid splitting strings, as was done in the preceding example. Translations can change the position of variables within the strings. So the above example should be:

```

Mojo.Log.warn("Can't open feed database (#", result.message,
  "). All feeds will be reloaded.");

var message = new Template($L("Can't open feed database ({message}). All
feeds will be reloaded.));
Mojo.Controller.errorDialog(message.evaluate({message: result.message}));

```

This is even more critical should 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: Two usages of a word in English might require two different translations in some languages. 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.

11.2.3.2. 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 then 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 that this is written those tools are not yet available. Refer to the Palm developer site for more information about localization tools.

11.2.3.3. 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",
```

As an example, the News *resources/es/strings.json* is shown here:

```
{
  "#{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 search" : "Cancelar búsqueda",
  "Check out this News story..." : "Leer esta noticia...",
  "Check this out: " : "Mira esto: ",
  "Copyright 2008-2009, Palm Inc." : "Copyright 2008-2009, Palm Inc.",
  "Database save error (#" : "Error al guardar en la base de datos (#",
  "Edit Feed" : "Editar fuente web",
  "Edit News Feed" : "Editar una fuente web de noticias",
  "Email" : "Correo electrónico",
```



```

"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 with this release" : "Nuevas características de esta versión",
"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" : "Fuente web RSS o ATOM",
"Search for: #{filter}" : "Buscar: #{filter}",
"Show Notification" : "Mostrar aviso",
"SMS/IM" : "SMS/IM",
"newsfeed.status" : "La solicitud de fuente web de noticias indicó
el estado #{status}.",
"Status #{status} returned from newsfeed request." : "La solicitud
de fuente web de noticias indicó el estado #{status}.",
"Stop" : "Detener",
"Title" : "Título",
"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."
}

```

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:

```
"newsfeed.status" : "Estado #{status} devuelto desde  
solicitud de fuente web de noticias",
```

11.2.4. 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 can be used in combination with your HTML to modify the presentation either to accommodate text translations (e.g. adjusting for significant changes in text length) or simply to address unique formatting requirements within a specific locale.

11.2.4.1. 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*.

NOTE

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.

11.2.4.2. 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 class="palm-row first">
      <div class="palm-row-wrapper">
        <div id="featureFeedToggle" x-mojo-element="ToggleButton"></div>
        <div class="title left">Rotación de fuente web destacada</div>
      </div>
    </div>
    <div class="palm-row">
      <div class="palm-row-wrapper">
        <div id="featureFeedList" x-mojo-element="ListSelector"></div>
      </div>
    </div>
    <div id="featureFeedDelay" class="featureFeedDelay" x-mojo-element="IntegerPicker"></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 id="feedCheckIntervalList" x-mojo-element="ListSelector"></div>
      </div>
    </div>
    <div class="palm-row">
      <div class="palm-row-wrapper">
        <div id="notificationToggle" x-mojo-element="ToggleButton"></div>
        <div class="title left">Mostrar aviso</div>
      </div>
    </div>
    <div class="palm-row last">
      <div class="palm-row-wrapper">
        <div id="bgUpdateToggle" x-mojo-element="ToggleButton"></div>
        <div class="title left">Activar dispositivo</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 what is shown in [Figure 11-3](#).

Figure 11-3. News localized to the `es_US` locale.



11.3. 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 11-1](#).

Table 11-2. Mojo.Format Methods

Method	Description
<code>Mojo.Format.getCurrentLocale()</code>	Returns the currently set locale as an ISO 639-formatted string (e.g. 'en_us' for US English).
<code>Mojo.Format.formatDate()</code>	Formats the date object appropriately for the current locale
<code>Mojo.Format.formatRelativeDate()</code>	Formats the data object as with <code>formatDate()</code> but returns yesterday/today/tomorrow if appropriate
<code>Mojo.Format.formatNumber()</code>	Converts a number to a string using the proper locale-based format for numbers and number separators
<code>Mojo.Format.formatCurrency()</code>	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.
<code>Mojo.Format.formatChoice()</code>	Formats a choice list to handle things like replacement parameters with plurals
<code>Mojo.Format.formatPercent()</code>	Converts a number to a percent string using the proper locale-based format for percentages
<code>Mojo.Format.using12HrTime()</code>	Returns true if the current locale uses 12 hour time or false if 24 hour time.
<code>Mojo.Format.getCurrentTimeZone()</code>	Returns current timezone

The behavior of some of the widgets should be influenced by the selected locale. Currently the `TimePicker` 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.

11.3.1.

11.3.1.1. Back to the News: Multilingual formatting

The News' banner notification includes a phrase indicating the number of new stories, as shown in [Figure 11-3](#).

Figure 11-4. News banner notification.

But this isn't linguistically correct (e.g. 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()` is intended to address this need.

```
var bannerParams = {
  messageText: Mojo.Format.formatChoice(
    feedList[curFeedIndex].newStoryCount,
    $L("0##{title} : No New Items|1##{title} : 1 New Item|>##{title}
  : #{count} New Items"),
  {title: feedList[curFeedIndex].title, count: feedList[curFeedIndex].newStoryCount}
)
```

```
};  
  
appController.showBanner(bannerParams, {action: 'notification'}, 'newsUpdates');
```

The call to `formatChoice()` passes an integer representing the quantity, and 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 last argument includes the object that supplies the values to be substituted for the templates `#{title}` and `#{count}`.

11.4. Summary

Palm webOS is a world-ready operating system designed to support localized and internationalized applications. This chapter provided an overview of locales, character sets, fonts and keyboards, and covered the localization architecture, tools and techniques. The internationalization APIs were briefly summarized.

Just 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. News Application Source Code

NOTE

This appendix is listed as "Appendix A" on Safari, but in the final, published version of the book, will be listed as "Appendix D". The appendixes A, B, and C from the final published version of the book will not be released on Safari Rough Cuts due to the fact that they will consist of topical coverage of the SDK that won't be in final, releasable form until roughly the time of the book's actual publication.

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 on <http://oreilly.com/catalog/9780596155254/>.

Note that the some of the sample code used in the early chapters is replaced in later chapters. Only the final version the application is shown here.

A.1.

A.1.1.

A.1.1.1. News Application Directory Structure

```

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
  icon.png
  images
    cal-selector-header-gray.png
    dashboard-icon-news.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-1.png
    url-icon.png
  index.html
  resources
    es
      appinfo.json
      strings.json
      views
        feedList
          addFeed-dialog.html
          feedList-scene.html
        preferences
          preferences-scene.html
  sources.json

```

```

stylesheets
News.css

```

A.1.1.2. 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
- database functions; set of methods to handle depot loading on startup
- update functions; set of methods to handle feed updates
- handleCommand; handles app menu selections

Data structures:
- db; depot object to save database
- feedList; list object used for main feedlist in feedList-assistant
- globals; set of persistent data used throughout app

App architecture:
- AppAssistant; handles startup, feed list management and app menu management
- FeedListAssistant; handles feedList navigation, feedList 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 = "0.81";
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.featureIndexFeed = 0; // Feature feed
News.featureStoryInterval = 5000; // Feature Interval (in ms)
News.notificationEnable = true; // Enables notifications
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");
    }
};

```

A.1.1.3. 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:
- title; News feed title
- count: Number of new stories
- message; Latest new story headline

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.update(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.
    // 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;
    }
};

// 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.1.1.4. 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:
- AddDialogAssistant; 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.update(this.dialogTitle);

    // Setup text field for the new feed's URL
    this.sceneAssistant.controller.setupWidget(
        "newFeedURL",
        {
            hintText: $L("RSS or ATOM feed URL"),
            focus: true,
            limitResize: 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)"),
            limitResize: true,
            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 is an HTML button
    // which will just cancel the action and close the scene
    this.sceneAssistant.controller.setupWidget('okButton',
        {type : Mojo.Widget.activityButton },
        this.okButtonModel = {buttonLabel : $L("OK"), disables: false});
    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.sceneAssistant.controller.setupWidget('cancelButton', {},
        {buttonLabel : $L("Cancel")});
    this.cancelHandler = this.cancel.bindAsEventListener(this);
    this.sceneAssistant.controller.listen("cancelButton", Mojo.Event.tap,
        this.cancelHandler);
};

// 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}:/i.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.buttonLabel = "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) {
    // Prototype template object generates a string from return status
    var t = new Template($L("#{status}"));
    var m = t.evaluate(transport);
    Mojo.Log.info("Valid URL (Status ", m, " returned).");

    // 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");
        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.update($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) {
    // Prototype template object generates a string from return status
    var t = new Template($L("#{status}"));
    var m = t.evaluate(transport);

    // Log error and put message in status area
    Mojo.Log.info("Invalid URL (Status ", m, " returned).");
    var addFeedTitleElement = this.controller.get("add-feed-title");
    addFeedTitleElement.update($L("Invalid URL - Please Retry."));
};

// cancel - close Dialog
AddDialogAssistant.prototype.cancel = 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.cancelHandler);
    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
    var featureFeedTitle = this.controller.get("featureFeedTitle");
    this.controller.update(featureFeedTitle,
        this.feeds.list[News.featureIndexFeed].title);

    this.featureFeedHandler = this.updateFeatureFeed.bindAsEventListener(this);
    this.controller.listen("feedList_view_header", Mojo.Event.tap,
        this.featureFeedHandler);

    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);

    // 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
    this.featureIndexStory = 0;
};

```

```

// activate - handle orientation, feature feed layout and rotation, and show
// the info dialog for the first launch after an app update.
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.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 News.featureIndexFeed, 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[News.featureIndexFeed].stories.length > 0) {
        var splashScreenElement = this.controller.get("splashScreen");
        splashScreenElement.hide();
        this.showFeatureStory();

        // Update the feature header
        var featureFeedTitle = this.controller.get("featureFeedTitle");
        this.controller.update(featureFeedTitle,
            this.feeds.list[News.featureIndexFeed].title);
    }

};

// 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;
    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("feedList_view_header",
        Mojo.Event.tap, this.featureFeedHandler);
    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 News.featureIndexFeed, then start the rotation
    if ((this.feeds.list[News.featureIndexFeed].stories.length > 0) &&
        (News.featureStoryTimer === null)) {
        var splashScreenElement = this.controller.get("splashScreen");
        splashScreenElement.hide();
        this.showFeatureStory();

        // Update the feature header
        var featureFeedTitle = this.controller.get("featureFeedTitle");
        this.controller.update(featureFeedTitle,
            this.feeds.list[News.featureIndexFeed].title);
    }
};

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("News deleting "+event.item.title+".");

    var deleteIndex = this.feeds.list.indexOf(event.item);
    this.feeds.list.splice(deleteIndex, 1);
    this.feedWgtModel.items = this.feeds.list;
    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 == News.featureIndexFeed) {
        News.featureIndexFeed = 0;
        this.featureIndexStory = 0;
        // Update View header with new feature story
        var featureFeedTitle = this.controller.get("featureFeedTitle");
        this.controller.update(featureFeedTitle,
            this.feeds.list[News.featureIndexFeed].title);
    } else {
        if (deleteIndex < News.featureIndexFeed) {
            News.featureIndexFeed--;
        }
    }
};

// listReorderFeed - triggered re-ordering feed list and updates the
// feedlist to reflect the changed order
FeedListAssistant.prototype.listReorderFeed = function(event) {
    Mojo.log("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);
    this.feedWgtModel.items = this.feeds.list;
    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 > News.featureIndexFeed && toIndex <= News.featureIndexFeed) {
        News.featureIndexFeed++;
    }
};

```

```

    } else {
        if (fromIndex < News.featureIndexFeed && toIndex > News.featureIndexFeed) {
            News.featureIndexFeed--;
        } else {
            if (fromIndex == News.featureIndexFeed) {
                News.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 in the News.featureIndexFeed, then start the rotation
        if ((this.feeds.list[News.featureIndexFeed].stories.length > 0) &&
            (News.featureStoryTimer === null)) {
            var splashScreenElement = this.controller.get("splashScreen");
            splashScreenElement.hide();
            this.showFeatureStory();

            // Update the feature header
            var featureFeedTitle = this.controller.get("featureFeedTitle");
            this.controller.update(featureFeedTitle, this.feeds.list
[News.featureIndexFeed].title);
        }
    }
    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[News.featureIndexFeed].stories.length) {
                this.featureIndexStory = 0;
            }
        }

        var summary = this.feeds.list[News.featureIndexFeed].stories[
            this.featureIndexStory].text.replace(/(<([^\>]+)>)/ig, "");
        summary = summary.replace(/http:\S+/ig, "");
        var featureStoryTitleElement = this.controller.get("featureStoryTitle");
        this.controller.update(featureStoryTitleElement,
            unescape(this.feeds.list[News.featureIndexFeed].stories[
                this.featureIndexStory].title));
        var featureStoryElement = this.controller.get("featureStory");
        this.controller.update(featureStoryElement, 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[News.featureIndexFeed], this.featureIndexStory);
    };

    // updateFeatureFeed - handles taps to the featureFeed header. If selector
    // is tapped, then display popup to allow for selection of new feed or
    // to disable rotation. Otherwise, toggle drawer.
    FeedListAssistant.prototype.updateFeatureFeed = function(event) {
        var target = event.target.id;
        if (target !== "featureFeedHitTarget") {
            if (this.featureFeedDrawer.open === true) {
                this.featureFeedDrawer.open = false;
                News.featureFeedEnable = false;
            } else {
                this.featureFeedDrawer.open = true;
                News.featureFeedEnable = true;
            }
            this.controller.modelChanged(this.featureFeedDrawer);
            // Update News saved preferences
            News.Cookie.storeCookie();
        }
        else {
            var myEvent = event;
            var findPlace = myEvent.target.id;

            // Setup popup for FEATURED FEEDLIST choices.
            // Pull out feedList titles for use in selector
            var choices = [{label: "", value: ""}];
            for (var x=0; x<this.feeds.list.length; x++) {
                choices[x] = {label: this.feeds.list[x].title, command: x};
            }

            // Now set up popup for feed selections;
            this.controller.popupSubmenu({
                onChoose: function(value) {

```

```

        News.featureIndexFeed = value;
        this.featureIndexStory = 0;
        News.feedListChanged=true;
        var featureFeedTitle = this.controller.get("featureFeedTitle");
        this.controller.update(featureFeedTitle, this.feeds.
list[News.featureIndexFeed].title);

        if (this.featureFeedDrawer.open === false) {
            this.featureFeedDrawer.open = true;
            this.controller.modelChanged(this.featureFeedDrawer);
            News.featureFeedEnable = true;
        }

        News.Cookie.storeCookie();          // Update News saved preferences
    },
    toggleCmd: News.featureIndexFeed,
    placeNear: findPlace,
    manualPlacement:true,
    popupClass:"featureFeed-selector-popup",
    items: choices
    });
    }
};

// -----
// 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, then return empty list, else 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 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);

};

// -----
// 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;
    }
};

```


A.1.1.5. news/app/assistants/preferences-assistant.js

```

/* Preferences - NEWS

Copyright 2009 Palm, Inc. All rights reserved.

Preferences - Handles preferences scene, where the user can:
- enable or disable the featured feed element on the FeedList Scene
- select the featured feed when enabled
- 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 interval;
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;
};

```

A.1.1.6. 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;
};

```

A.1.1.7. 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);

};

// activate - display selected story
StoryViewAssistant.prototype.activate = function(event) {
    Mojo.Log.info("Story View Activated");

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

    // 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: {
                        scene: "page",
                        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;
    }
};

```

A.1.1.8. 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 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("comPalmAppNewsPrefs");
    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
    });
}

});

```

A.1.1.9. 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/rougthtype/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:"McCovey Chronicles",
        url:"http://feedproxy.google.com/sportsblogs/mccoveychronicles.xml",
        type:"atom", 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: 100, replace: false},
            this.loadFeedDbOpenOk.bind(this),
            function(transaction, result) {
                Mojo.Log.warn("Can't open feed database: ", result.message);
            }
        );
    },

    // 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 size: " , Object.values(fl).size());

    if (Object.toJSON(fl) == "{}" || 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
        if (News.dbUpdate == "0.4") {
            for (var i=0; i<this.list.length; i++) {
                for (var j=0; j<this.list[i].stories.length; j++) {
                    if (this.list[i].stories[j].unreadStyle == "<b>") {
                        this.list[i].stories[j].unreadStyle = News.unreadStory;
                    }
                }
            }
            News.dbUpdate="";
        }
        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(transaction,result) {
    Mojo.Log.warn("Database save error: ", result.message);
    /* if (result.message == News.dbOutOfMemoryError) {
        // replace contents of text bodies with the summaries and try again
        for (var i = 0; i < this.list.length; i++) {
            var stories = this.list[i].stories
            for (var j=0; j<stories.length; j++ ) {
                stories[j].text = stories[j].summary;
            }
        }
    }
    */
},

// 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.notificationEnabled) {
            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;
}
}
});

```

A.1.1.10. news/app/assistants/views/dashboard/dashboard-scene.html

```
<div id="dashboardinfo" class="dashboardinfo"></div>
```

A.1.1.11. 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>

```

A.1.1.12. 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>

```

A.1.1.13. 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="featureFeed_source" class="featureFeed-source">
                <div id="featureFeedTitle" class="featureFeed-selector">None</div>
                <div id="featureFeedHitTarget" class="featureFeed-hit-target"></div>
            </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>
                        <div class="palm-body-text">Copyright 2009, Palm®</div>
                    </div>
                </div>
                <div id="featureStoryTitle" class="palm-body-title">
                    #{title}
                </div>
                <div id="featureStory" class="palm-body-text">
                    #{text}
                </div>
            </div>
        </div>
    </div>

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

```

A.1.1.14. news/app/assistants/views/feedList/feedListTemplate.html

```

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

```

A.1.1.15. news/app/assistants/views/feedList/feedRowTemplate.html

```

<div class="palm-row" x-mojo-tap-highlight="momentary">
    <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>

```

A.1.1.16. 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"
class="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>
</div>

```

A.1.1.17. news/app/assistants/views/storyList/storyList-scene.html

```

<div class="palm-header-spacer"></div>
<div id="storyListScene" class="storyListScene">
    <div class="palm-list">
        <div x-mojo-element="List" id="storyListWgt" ></div>
    </div>
</div>
<div class="storyList-filter">
    <div x-mojo-element="FilterList" id="storyListSearch" class="palm-list"></div>
</div>

```

A.1.1.18. news/app/assistants/views/storyList/storyListTemplate.html

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

A.1.1.19. news/app/assistants/views/storyList/storyRowTemplate.html

```
<div class="palm-row" x-mojo-tap-highlight="momentary">
  <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>
```

A.1.1.20. 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">
        #{title}
      </div>
    </div>
    <div id="storyViewSummary" class="palm-text-wrapper">
      <div class="palm-body-text">
        #{text}
      </div>
    </div>
  </div>
</div>
```

A.1.1.21. 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",
  "logLevel": "99",
  "timingEnabled": "true",
  "noDeprecatedStyles": "true",
  "theme": "light"
}
```

A.1.1.22. 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>
```

A.1.1.23. news/resources/es/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",
  "logLevel": "99",
  "timingEnabled": "true",
  "noDeprecatedStyles": "true",
  "theme": "light"
}
```

A.1.1.24. news/resources/es/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",
  "Are My Feeds Saved?" : "¿Se guardan mis fuentes web?",
  "Can I Change the Rotation?" : "¿Puedo cambiar la rotación?",
  "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."
}

```

A.1.1.25. news/resources/es/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 class="textfield-group" x-mojo-focus-highlight="true">
      <div class="title">
        <div x-mojo-element="TextField" id="newFeedName"></div>
      </div>
    </div>
  </div>
  <div class="palm-dialog-buttons">
    <div x-mojo-element="Button" id="okButton">
      <div x-mojo-element="Button" id="cancelButton">
    </div>
  </div>
</div>

```

A.1.1.26. news/resources/es/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>
  </div>
</div>

```

```

        <div id="featureFeed_source" class="featureFeed-source">
            <div id="featureFeedTitle" class="featureFeed-selector">
                Nada
            </div>
            <div id="featureFeedHitTarget" class="featureFeed-hit-target">
            </div>
        </div>
    </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">
                #{title}
            </div>
            <div id="featureStory" class="palm-body-text">
                #{-text}
            </div>
        </div>
    </div>
</div>
</div>

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

```

A.1.1.27. news/resources/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"
            class="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>
</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>

```

A.1.1.28. 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\\cookie.js"
  },
  {
    "source" : "app\\models\\feeds.js"
  }
]

```

A.1.1.29. news/stylesheets/News.css

```

/*    News CSS

    Copyright 2009 Palm, Inc.  All rights reserved.

    App overrides of palm scene and widget styles.
*/

/* Constrains 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-source {
    position: relative;
    float: right;
    margin: 8px -16px 0px 0px;
    font-size: 14px;
    text-transform: uppercase;
    font-weight: bold;
    height: 35px;
    line-height: 33px;
    border-width: 0px 22px 0px 16px;
    -webkit-border-image:
        url(../images/cal-selector-header-gray.png) 0 22 0 16 stretch stretch;
    -webkit-box-sizing: border-box;
}
.featureFeed-hit-target {
    position: absolute;
    top: -10px;
    left: -35px;
    right: -25px;
    height: 52px;
}
.featureFeed-selector {
    margin-left: -5px;
    padding-right: 2px;
    background: transparent;
    display: inline-block;
    min-width: 12px;
    max-width: 120px;
    text-overflow: ellipsis;
    white-space: nowrap;
    overflow: hidden;
}
.featureFeed-selector-popup {
    top: 38px;
    right: -3px;
    left: 120px;
}
.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);
}

```