



Cool projects that will push your skills to the limit

PhoneGap 2.x Mobile Application Development

Create exciting apps for mobile devices using PhoneGap

HOTSHOT

Kerri Shotts

www.it-ebooks.info

[**PACKT**] open source[®]
PUBLISHING community experience distilled

PhoneGap 2.x

Mobile Application

Development

HOTSHOT

Create exciting apps for mobile devices using PhoneGap

Kerri Shotts



BIRMINGHAM - MUMBAI

PhoneGap 2.x Mobile Application Development HOTSHOT

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2013

Production Reference: 1070213

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-940-3

www.packtpub.com

Cover Image by Siddharth Ravishankar (sidd.ravishankar@gmail.com)

Credits

Author

Kerri Shotts

Project Coordinator

Abhishek Kori

Reviewers

Steve Husting

Johnathan Iannotti

Tony Pye

Proofreader

Sandra Hopper

Indexers

Hemangini Bari

Tejal Soni

Acquisition Editor

Aarthi Kumaraswamy

Graphics

Aditi Gajjar

Lead Technical Editor

Sweny M. Sukumaran

Production Coordinator

Manu Joseph

Technical Editor

Nitee Shetty

Cover Work

Manu Joseph

About the Author

Kerri Shotts has been programming since she learned BASIC on her Commodore 64. She earned her degree in Computer Science and has worked as a Test Engineer and Database Administrator. Now, a Technology Consultant, she helps her clients with custom websites, apps (desktop and mobile), and more. When not at the computer, she enjoys photography and taking care of her aquariums.

About the Reviewers

Steve Husting has been involved in the creation of iPhone and Android apps for several years. During daytime he helps manage the website for his employer's company and at night he creates iPhone and Android apps to publish his Christian works for the public. He keeps up a hybrid app blogging website, <http://iphonedevlog.wordpress.com>, to publish his findings and store his notes about all things, PhoneGap/Cordova-related. He met the author of this book on the PhoneGap Google Groups forum, and was deeply appreciative of the breadth of her knowledge and ability to convey it to others.

Johnathan Iannotti is a software engineer and geek on an epic journey of life. He's loved technology since he was young, writing Atari BASIC programs and surfing BBS's into the morning hours. Over the years, his passion for technology evolved with the web and later into mobile apps built with both native and web technologies.

His experience spans the financial, healthcare, and military industries. He has held positions with the federal government, digital agencies, medical manufacturers, EMR providers, and financial institutions throughout North America. He is also a Combat Veteran having served multiple tours of duty and almost a decade in the U.S. Army.

He spends his free time innovating, creating Arduino gadgets, mobile apps, and riding his motorcycle through the Texas Hill Country.

You can follow him on Twitter @notticode or visit his website at www.johnforhire.com.

Tony Pye has over 10 years of experience in producing web-based solutions. He strives to keep ahead of rapidly evolving web technologies in order to be able to offer innovative solutions.

His passion is for matching the business goals with innovative use of technology. As head of digital production at INK Digital Agency, guiding clients through the complex digital world and integrating digital marketing with internal business systems is his specialty.

Liaising with the creative and user experience team members, meeting clients, presenting ideas, and help define goals is just part of his normal day at INK.

Some of the solutions he has helped to produce have delivered exciting results for companies including Ballymore, Morrisons, Renault, Tarmac, Aviva, LA fitness, and the University of Leeds.

He has also worked on a number of other books as the technical reviewer, including *Pro HTML5 Programming*, Apress and *The Definitive Guide to HTML5 WebSocket*, Apress (not yet published).

I'd like to thank my beautiful wife for her support and patience with me during the often long nights. Her fantastic coffee-making skills were certainly put to great use. Thanks darling!

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Project 1: Let's Get Local!	9
What do we build?	9
Designing the app – UI/interactions	14
Designing the data model	20
Implementing the data model	23
Implementing the start view	34
Implementing our game view	41
Implementing the end view	54
Putting it all together	58
Game Over..... Wrapping it up	67
Can you take the HEAT? The Hotshot Challenge	68
Project 2: Let's Get Social!	69
What do we build?	69
Designing the app – UI/interaction design	72
Designing the app – the data model	76
Implementing the data model	77
Configuring the plugins	86
Implementing the social view	93
Implementing the tweet view	101
Game Over..... Wrapping it up	108
Can you take the HEAT? The Hotshot Challenge	108
Project 3: Being Productive	109
What do we build?	109
Designing the user interface	110
Designing the data model	113
Implementing the data models	115

Table of Contents

Implementing documents view	125
Implementing the file view	136
Game Over..... Wrapping it up	141
Can you take the HEAT? The Hotshot Challenge	141
Project 4: Let's Take a Trip	143
What do we build?	143
Designing our UI and the look and feel	145
Designing our data model	149
Implementing our data model	151
Changing our document manager	157
Implementing our map view	159
Game Over..... Wrapping it up	168
Can you take the HEAT? The Hotshot Challenge	169
Project 5: Talking to Your App	171
What do we build?	171
Designing the user interface and the look and feel	172
Designing the data model	175
Implementing the data model	177
Implementing gesture support	185
Implementing the main view	194
Game Over..... Wrapping it up	206
Can you take the HEAT? The Hotshot Challenge	206
Project 6: Say Cheese!	207
What do we build?	207
Designing the user interface and the look and feel	209
Designing the data model	211
Implementing the document view	213
Implementing the image view	230
Game Over..... Wrapping it up	233
Can you take the HEAT? The Hotshot Challenge	234
Project 7: Let's Go to the Movies!	235
What do we build?	235
Preparing for the video thumbnail plugin	237
Implementing the video thumbnail plugin for iOS	240
Implementing the video thumbnail plugin for Android	247
Integrating with the video thumbnail plugin	251
Implementing recording and importing of video	253
Implementing video playback	256

Table of Contents

Game Over..... Wrapping it up	259
Can you take the HEAT? The Hotshot Challenge	260
Project 8: Playing Around	261
What do we build?	261
Designing the game	263
Implementing the options view	266
Generating levels	271
Drawing to the canvas	276
Keeping up	279
Performing updates	280
Handling touch-based input	284
Handling the accelerometer	286
Game Over..... Wrapping it up	289
Can you take the HEAT? The Hotshot Challenge	291
Project 9: Blending In	293
What do we build?	293
Installing the plugins	294
Adding the navigation bar	298
Adding the tab bar	304
Adding the ActionSheet	308
Adding the message box	310
Adding the picker	312
Adding the e-mail composer	314
Game Over..... Wrapping it up	316
Can you take the HEAT? The Hotshot Challenge	316
Project 10: Scaling Up	317
What do we build?	317
Designing the scaled-up UI	318
Implementing the scaled-up UI	321
Designing the split-view UI	326
Implementing the split-view UI	328
Game Over..... Wrapping it up	337
Can you take the HEAT? The Hotshot Challenge	338
Appendix A: Quick Design Pattern Reference	339
The navigation list	340
The grid	341
Carousel 1	342
Carousel 2	343

Table of Contents —————

The login screen	344
The sign-up form	346
The table	347
The list of choices	349
Doing things in bulk	350
Searching	351
Some things to keep in mind	353
Summary	354
Appendix B: Installing ShareKit 2.0	355
Index	365

Preface

Developing apps for mobile devices can be done using many different approaches and languages. Many apps are developed natively; meaning that they are developed in Java, Objective C, or some other language natively understood by the SDK available for the device. While native development allows the greatest flexibility and performance, the problem arises when you want to move an app from one platform to another: suddenly you're writing the app nearly from scratch. Then if you want to move to another platform, the same thing occurs. There's got to be a better way!

All current mobile platforms support the idea of *web apps*. These are applications coded entirely in HTML and JavaScript. For simple apps, or for apps that don't need to interact with the device's capabilities, this works just fine. But the moment you need to access the file system, work with the camera, and so on, you start needing more access to the device.

This is where PhoneGap comes in. PhoneGap (built on Cordova) wraps your HTML and JavaScript with just enough of a native app to let your web app feel more at home on the device. This wrapper is different for each platform, but exposes common capabilities in a consistent way. This helps you to write less code across multiple platforms.

Since PhoneGap is wrapping your HTML and your JavaScript in a native shell, you also gain the ability to submit your app to the platform's app store—something you can't do with just a simple web app. Keep in mind, though, that most app stores want your app to look and feel something like a native app, and some are more strict than others when it comes to how the app should look and feel. Furthermore, don't just wrap your existing website that is hosted on some other server—many app stores will reject these kind of apps. Your app needs to have local HTML and JavaScript that supports the UI and interacts with the device.

Which is where this book comes in. While we're using PhoneGap as the shell and the interface to some of the more interesting device abilities, we're not just repeating the PhoneGap documentation, either. Instead, there are full apps; each one designed to take advantage of one or more features of the device, yes, but also fully functioning apps.

This book will hopefully show you how to take PhoneGap and make interesting, even exciting apps that are also cross platform. While we focus only on iOS and Android, the techniques within this book can be easily extended to BlackBerry, Windows Phone, and others with minor modifications.

What this book covers

Project 1, Let's Get Local!, introduces us to PhoneGap with a simple quiz game. We'll also be introduced to app localization by making the game available in English as well as in Spanish. We'll also be introduced to the simple framework we'll be using for the rest of the book.

Project 2, Let's Get Social!, helps us develop a simple social app that displays feeds from select Twitter accounts. We'll cover plugin installation into PhoneGap, for when we need access to native functionality that PhoneGap doesn't supply.

Project 3, Being Productive, introduces us to an app, which like most apps needs to work with the file system for persistent data storage. It is a simple note-taking app and will allow us to fully explore creating, renaming, copying, and deleting files.

Project 4, Let's Take a Trip, helps us build an app that records the location of a user over a given period of time. This will require access to the GPS functionality of the device. We'll also need to use Google Maps. We'll build further on the file management introduced in *Project 3*.

Project 5, Talk to Your App, helps us create an app that will record voice memos, and allow the user to play them back at will. Along the way we'll integrate with PhoneGap's audio capture and playback APIs.

Project 6, Say Cheese!, covers how to display thumbnails in a memory-efficient manner as display and capture of media is critically important in most apps. We'll also interface with the device's camera and photo gallery.

Project 7, Let's Go to the Movies!, is much like *Project 6*, only here we're dealing with video. We'll be introduced to playing video on iOS and Android (each very different), and we'll also be tasked with recording video. Finally, we'll write our first plugin to extract a thumbnail from the video for display in the app.

Project 8, Playing Around, introduces us to a simple game that uses the HTML5 canvas to play a simple game, as there are plenty of apps that do something important, sometimes we just want to have fun. We'll also work with the device's accelerometer.

Project 9, Blending In, takes an app previously developed and applies native components on to it so that it looks and feels more like a native app, because sometimes we just want to blend in. While this project is tailored specifically for iOS, you can use the concepts for other platforms.

Project 10, Scaling Up, introduces us to the concept of detecting a tablet, since so far each app in this book has been tailored for a phone-sized device but tablets are quite prolific as well. We will also get acquainted with the common design patterns used to scale our app up to a tablet-sized device.

Appendix A, Quick Design Pattern Reference, covers some of the common design patterns used in mobile apps.

Appendix B, Installing ShareKit 2.0, covers all the steps necessary to integrate ShareKit 2.0 with your projects, because integrating it with iOS can get a bit painful at times.

What you need for this book

To build/run the code supplied for the book, the following software is required (divided by platform where appropriate):

	Windows	Linux	OS X
For iOS Apps			
IDE			XCode 4.5+
OS			OS X 10.7+
SDK			iOS 5+
For Android Apps			
IDE	Eclipse 4.x Classic	Eclipse 4.x Classic	Eclipse 4.x Classic
OS	XP or newer	Any modern distro supporting Eclipse and Android SDK—Ubuntu, RHEL, and so on.	OS X 10.6+ (probably works on lower versions)
Java	1.6 or higher	1.6 or higher	1.6 or higher
SDK	Version 15+	Version 15+	Version 15+
For All Platforms			
Apache Cordova / PhoneGap	2.2	2.2	2.2
Plugins	Current	Current	Current
Version Control*	Git (Appdx B)	Git (Appdx B)	Git (Appdx B)

* Used only for installation of the ShareKit 2.0 plugin in *Appendix B*.

Websites that can be useful for downloads are as follows:

- ▶ Xcode: <https://developer.apple.com/xcode/>
- ▶ iOS SDK: <https://developer.apple.com/devcenter/ios/index.action>
- ▶ Eclipse: <http://www.eclipse.org/downloads/packages/eclipse-classic-421/junior1>
- ▶ Android SDK: <http://developer.android.com/sdk/index.html>
- ▶ Apache Cordova/PhoneGap: <http://phonegap.com/download>
- ▶ Plugins: <https://github.com/phonegap/phonegap-plugins>
- ▶ Git: <http://git-scm.com/downloads>

Who this book is for

This book is for any developer who has a good sense of how to develop with HTML and JavaScript but wants to move into mobile app development. The developer should know how to write HTML and have a reasonable understanding of JavaScript. The developer should also be comfortable with setting up a development environment such as Eclipse or Xcode.

This book is also for any native developer who is looking for a way to create apps that can span multiple platforms with limited modifications. PhoneGap is a great tool with which you can build a single HTML/JavaScript codebase that works across many platforms.

The examples in this book specifically use PhoneGap 2.2.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

What do we build?

This section explains what you will build.

What does it do?

This section explains what the project will achieve.

Why is it great?

This section explains why the project is cool, unique, exciting, and interesting. It describes what advantage the project will give you.

How are we going to do it?

This section explains the major tasks required to complete your project.

- ▶ Task 1
- ▶ Task 2
- ▶ Task 3
- ▶ Task 4, and so on

What do I need to get started?

This section explains any pre-requisites for the project, such as resources or libraries that need to be downloaded, and so on.

Task 1

This section explains the task that you will perform.

Getting Ready

This section explains any preliminary work that you may need to do before beginning work on the task.

Getting on with it

This section lists the steps required in order to complete the task.

What did we do?

This section explains how the steps performed in the previous section allow us to complete the task.

What else do I need to know

The extra information in this section is relevant to the task.

Preface

In this book, you will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `addTranslation` method adds a translation to a specific locale."

A block of code is set as follows:

```
self.addAnswer = function( theAnswer )
{
    self.answers.push ( theAnswer );
    return self;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<div class="navigationBar">
    <div id="gameView_title"></div>
    <button class="barButton backButton"
        id="gameView_backButton" style="left:10px"></button>
</div>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "To the left of the text is a **Back** button."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

Project 1

Let's Get Local!

There are a lot of languages in the world, and chances are good that you want your app to have the widest possible use and distribution, which means that you will need to provide your app with the ability to be multilingual. This can be a thorny task; a lot goes in to localization, translations, currency formats, date formats, and so on. But thankfully, there are some very smart people out there who have already worked through a lot of the pain involved. Now it's up to us to put that work to good use.

What do we build?

The project that we will create is a simple game entitled *Quiz Time!* The game will essentially ask the player ten random questions in their native language and then tally and present their score when the game is finished. At the end, the app will ask the user if they want to try again as well.

The app itself will serve to introduce you to creating mobile apps using a simple framework named **YASMF (Yet Another Simple Mobile Framework)**. There are a multitude of fantastic frameworks out there (jQuery Mobile, jQuery Touch, iUI, Sencha Touch, and so on.), but the point of this book isn't to show you how to use a particular framework; rather, the point is to show you how to use PhoneGap to do some amazing things. The framework you choose to use ultimately doesn't really matter that much—they all do what they advertise—and our using a custom framework isn't intended to throw you off-kilter in any fashion. The main reason for using this particular custom framework is that it's very lightweight and simple, which means the concepts it uses will be easy to transfer to any framework. For more information regarding the framework, please visit <https://github.com/photokandyStudios/YASMF/wiki>.

The app itself will also serve as a foundation to creating localized apps in the future. Localization is absolutely critical to get right, even in the beginning stages of development, which is why we start with it here, and why we assign it such importance. In essence, this first project is intended to make the rest of your app development career easier.

What does it do?

As an app, Quiz Time! is pretty simple. There are only three screens, and only one of them is remotely complex. The game has ten built-in questions that it will randomly ask of the player. If the question is correct, the player is notified, and their score is increased by an arbitrarily large number. This is to show that we correctly handle the display of numbers in the player's locale. If the question is incorrect, we also notify the user, and then decrement their score. If they get enough questions wrong, they'll end up in negative territory, which is another great test for our localization skills.

Once the game is over, we'll display the score and the date to the player, along with the opportunity to try again. If the player does elect to try again, we'll reset everything and start the game over.

Why is it great?

You'll be primarily learning two things: building a simple game in PhoneGap, and localizing that app from the very beginning. A lot of projects forget about localization until near the end of the project, and then the poor developers find out that it is very difficult to shoehorn localization in after most of the project has already been developed. For example, the space assigned to some text might turn out to be too small for certain languages, or the images used as buttons or other widgets might not be large enough to hold the localized text. The app itself might crash in a certain language because it didn't expect to receive any non-English characters. By implementing localization at the start of your app development, you'll be saving yourself a lot of effort down the road, even if the first release of your app is only localized to one locale.

You'll often see the word Cordova in our code examples in this book. PhoneGap was recently acquired by Adobe and the underlying code was given to the Apache Incubator project. This project is named Cordova, and PhoneGap utilizes it to provide its various services. So if you see Cordova, it really means the same thing for now.

How are we going to do it?

We're going to follow the typical development cycle: design, implement, and test the app. Our design phase won't just include the user interface, but also the data model, that is, how our questions are stored and retrieved. The implementation will focus on our three stages of the app: the *start* view, the *game* view, and the *end* view. After implementation, we'll test the app not only to make sure it properly handles localization but also to make sure that the game works correctly.

Here's the general outline:

- ▶ Designing the app, UI/interactions
- ▶ Designing the data model
- ▶ Implementing the data model
- ▶ Implementing the start view
- ▶ Implementing the game view
- ▶ Implementing the end view
- ▶ Putting it all together

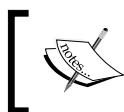
What do I need to get started?

First, be sure to download the latest version of PhoneGap from <http://phonegap.com/download>, currently 2.2.0 (as this was being written), and extract it to the appropriate directory. (For example, I use /Applications/phonegap/phonegap220.) Make sure that you have also installed the appropriate IDEs (Xcode for iOS development and Eclipse for Android development).

Next, download the latest version of the YASMF framework from <https://github.com/photokandyStudios/YASMF/downloads>, and extract it anywhere. (For example, I used my Downloads folder.)

If you want a copy of the projects for this book in order to look at, or to avoid the following project-creation steps, you can download them at <https://github.com/photokandyStudios/phonegap-hotshot>.

Next, you need to create a project for the various platforms you intend to support. Here's how we create both projects at once on Mac OS X. The commands should translate to Linux and Android-only projects with a little modification, and the same should apply to creating Android projects on Windows with some additional modification. For the following steps, consider \$PROJECT_HOME to be the location of your project, \$PHONEGAP_HOME to be the location where you installed PhoneGap, and \$YASMF_DOWNLOAD to be the location where you extracted the YASMF framework.



The following steps are just the steps that I use when setting up a project. You can, of course, structure it however you would like, but you'll need to make any modifications with regards to the file references and the likes on your own.

The following steps assume that you have downloaded PhoneGap (Cordova) 2.2.0. If you download a more recent version, the following steps *should* work with minimal modification:



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

1. Use the following code snippet:

```
mkdir $PROJECT_HOME
cd $PROJECT_HOME
mkdir Android iOS www
cd $PHONEGAP_HOME/lib/android/bin
./create $PROJECT_HOME/Android/QuizTime com.phonegapshot.
QuizTime QuizTime
cd $PHONEGAP_HOME/lib/ios/bin
./create $PROJECT_HOME/iOS com.phonegapshot.QuizTime QuizTime
cd $PROJECT_HOME
mkdir www/cordova
cp Android/QuizTime/assets/www/cordova-2.2.0.js www/cordova/
cordova-2.2.0-android.js
cp iOS/www/cordova-2.2.0.js www/cordova/cordova-2.2.0-ios.js
cd Android/QuizTime/assets
rm -rf www
ln -s ../../www
cd ../../ios
rm -rf www
ln -s ../www
cd ..
cd www
cp -r $YASMF_DOWNLOAD/framework .
mkdir images models views style
cd ..
cd Android/QuizTime/src/com/phonegapshot/QuizTime
edit QuizTime.java
Change "index.html" to "index_android.html"
Save the file.
cd $PROJECT_HOME/iOS/QuizTime
```

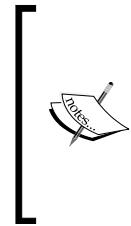
2. Edit Cordova.plist.

3. Search for UIWebViewBounce.
4. Change the <true/> tag just below it to <false/>.
5. Search for ShowSplashScreenSpinner.
6. Change the <true/> tag just below it to <false/>.
7. Search for ExternalHosts.
8. Remove the <array/> line and replace it with "<array>", "<string>*</string>" and "</array>". This isn't always something you'd want to do for a production app, but as it allows for our apps to access the Internet with no restrictions, it's good for testing purposes.
9. Save the file.
10. Start Eclipse.
11. **Navigate to File | New | Project...**
12. Select **Android Project**.
13. Click on **Next >**.
14. Select the **Create project from existing source** option.
15. Click on **Browse**.
16. Navigate to \$PROJECT_HOME/Android/QuizTime/.
17. Click on **Open**.
18. Click on **Next >**.
19. Uncheck and re-check the highest **Google APIs** entry. (For some reason, Eclipse doesn't always keep the correct SDK version when doing this, so you may have to go back after the project is created and reset it. Just right-click any directory, **Configure Build Paths...** and go to the **Android** section. Then you can re-select the highest SDK.)
20. Click on **Next >**.
21. Change the **Minimum SDK** value to 8.
22. Click on **Finish**.
23. Start Xcode.
24. Navigate to **File | Open....**
25. Navigate to the project in \$PROJECT_HOME/ios.
26. Click on **Open**.
27. At this point you should have Xcode and Eclipse open with the project. Close both; we'll be using our favorite editor for now.

When the project is created, the following directory structure results:

- ▶ /Android: The Android project
- ▶ /iOS: The iOS project
- ▶ /www
 - /cordova: We'll place the PhoneGap support libraries here.
 - /framework: Our framework will be in this directory.
 - /cultures: Any localization configuration will be placed here.
The framework comes with en-US.
 - /images: All of our images will be in this directory.
 - /views: All of our views will be here.
 - /models: All of our data models will be here.
 - /style: Any custom CSS we need to use will live here.

Once you've created the project, you also need to download the jQuery/Globalize repository from <https://github.com/jquery/globalize>. There's a lot of content there, but we're most interested in the lib directory. Copy the globalize.culture.en-US.js and globalize.culture.es-ES.js files to the www/framework/cultures directory. (If you want, feel free to copy other culture files as well, if you want to try your hand at localizing in a language you know.)



If you are using Eclipse, you must make absolutely certain that all the files you use in the www directory are set to the proper encoding. The easiest way to do this is to right-click on the assets directory, click on Properties, and then click on Other. Select the UTF-8 option from the drop-down list and click on Apply. If you don't do this, it is entirely possible that some of your localized content will not be displayed correctly.

Designing the app – UI/interactions

In this first task we'll be designing the look and feel of the application as well as specifying the interactions between the various elements in the user interface and the player. For most of this task you can use pencil and paper or a graphics editor, though at some point you'll need a graphics editor such as Adobe Photoshop or GIMP in order to create some of the resources you'll need for the app.

Getting on with it

The difficulty when it comes to designing an app that can run on more than one platform is that each platform has many different ideas when it comes to how things should look on the screen. There are several ways to approach this; they are discussed as follows:

- ▶ You can build the user interface for the majority of your market, and use the exact interface on all the other devices (but be careful; this will often lead to poor reviews).
- ▶ You could decide to customize the app's user interface for each device. This often requires a significant amount of work to accomplish and get it *just right*, but it can be very rewarding, especially when the end user has no idea the app wasn't written just for their own platform.
- ▶ Or you could create a platform-agnostic look and feel. This is the direction we'll take in this app. The interface would be reasonably at home on iOS and Android devices. That's not to say that the appearance will be identical on both devices; it won't, but it will be similar while incorporating some of the platform-specific notions as well.

Before we go too much further, we need to get out our pencil and paper and sketch out an idea of how we want our app to look. It should be similar to the following screenshot:



Our start view is a fairly simple view. At the top of the view we will have a navigation bar containing our app's title. In other views, this bar would often have other buttons in it, including one to go back to the previous view. At the bottom of the view, we will have a toolbar which will contain buttons relevant to the current view.

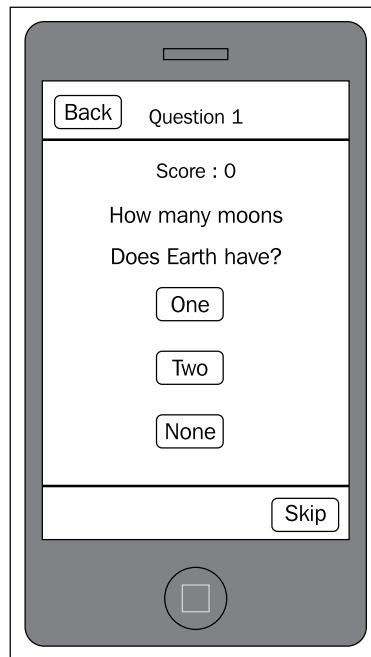
The app's title will be an image containing the title of the application. This image should be made using a font that's fun and stylistic. The image will be appropriately localized.

We'll have one button in the toolbar: a **Start** button. The text needs to be localized.

Below the navigation bar is the content area. Here we describe what the app will do. We won't have anything terribly fancy here; our space is limited, especially since we are restricted to the phone's screen size. In the future, we'll talk about how to allow content to scroll, but for now we'll keep it short and simple.

We do want to add a little bit of pizazz to the view, so we'll add a color splash to the background. You could make this anything you want, we'll go with rays of color shooting up from the bottom.

Our game view looks like the following screenshot:



Our game view is the most complex view we have in this app. Let's start outward and work in.

At the top, our navigation bar will indicate the current question number. This lets the player know how many questions they've answered. To the left of the text is a **Back** button. If clicked, it should take the player back to the start view.

At the bottom, our toolbar contains a single button: **Skip**. This button will allow the player to skip any question they don't want to answer. For now, we won't assign any penalty to skipping a question, but you could always add a score deduction or something worse if you wanted to do so. If you removed the button entirely, it would be wise to remove the toolbar as well.

In the middle is our content area, the most complex portion of the view; at the top we have the player's score, which needs to be localized. Below it is the question being asked, again, properly localized.

Below the question, we have several buttons; these need to be generated dynamically based on the question being asked. Not every question will have three answers; there may be some with two answers or some with four or more. The answers themselves also need to be properly localized.

Tapping a button will check to see if the button is labeled with the correct answer. If it is, we'll display a nice notice and increment the score. If it isn't, we'll indicate such and decrement the score.

After a question is answered or skipped, we'll get a new question and display it on the screen. Then, after ten questions have been answered, we'll end the game. The end view looks like the following screenshot:



The end view is similar to the start view in that it isn't terribly complex, but it does have a little more going on. It needs to properly display the final score and permit the player to play the game again.

The navigation bar contains the text **Results** and also a **Back** button. If tapped, it is the same thing as starting the game all over again.

The toolbar contains the **Try Again?** button. If tapped, it also starts the game again.

In the content area, we display a message containing the final score, and the date when it was achieved.

Of course, all of the content on the view needs to be properly localized. Numbers are hard enough; dates are even worse. It's a good thing that we have jQuery/Globalize to fall back on, or we'd have to do the hard work of localizing the date ourselves.

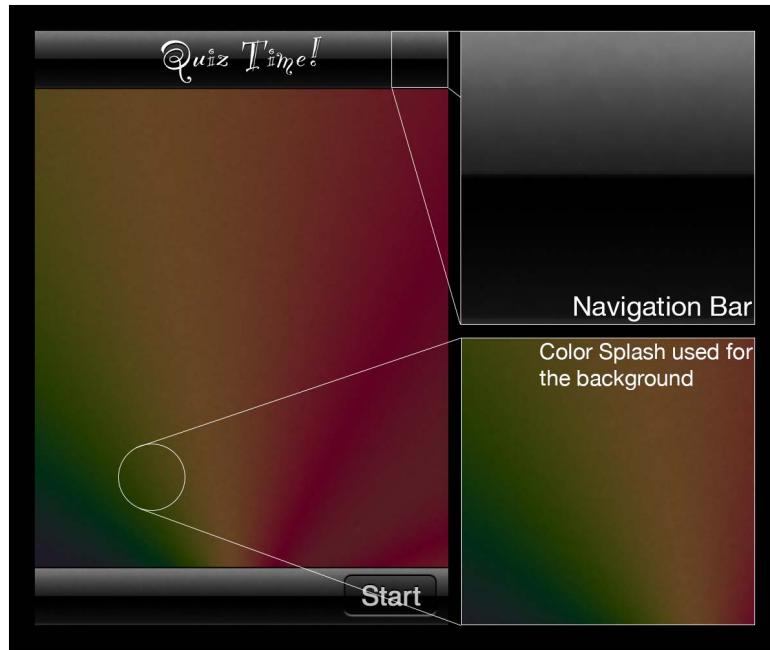
Now that we've sketched the user interface, it's time to start building some of the resources we'll need in our app. Open up your graphics editor and build a template of what any one of the views would look like. What we're doing here is determining what parts of the display will need to have images generated, and what parts will be able to be text or CSS-generated.

It's not super critical that you have the exact dimensions of any specific device. After all, the app can run on many different devices, each of which has a different screen size. We'll use 640 x 920 px, which just happens to be the available area on the screen for an iPhone 4 with a Retina display.



You do need to design using a high-enough resolution to get Retina-quality assets out of the design. That is, if you expect an icon to be 32 x 32 px, you will actually want it to be 64 x 64 px. Now whether you build on an exact size is up to you, but it's best to target the device you think will get the most use.

Here's the final template we're using:



There's a little bit of a texture there. While it's possible to do this in CSS, it's easiest to use images instead. The texture itself is tileable and so it can adapt to any screen size. The navigation bar should be placed in the `images` directory and named `NavigationBar.png`.

Notice the title? While this could also be handled by CSS and adding the font to your app, that gets into a lot of sticky licensing issues. Instead, we'll use an image of it, which means the font itself will never get distributed. The title should be placed in the `images` directory and named `AppTitle-enus.png`. The Spanish version (which should read *¡Examen Tiempo!*) should be named `AppTitle-eses.png`.

The background will also be an image, though you could likely approximate it with CSS (though getting the texture there would be a bit painful). Since we're supporting many platforms and screen sizes, the image approach is the best. This image should be saved in the `images` directory and named `Background.jpg`.

We'll build the app so that the image stretches to fill the screen. There will be some minor distortion, of course, but since this image is just a color splash, it doesn't really matter. (Other options include creating the background at various resolutions, or to create a tileable background that fills easily to any resolution.)

The button, on the other hand, is easy to build in CSS, and it's easy enough to get right on many platforms. In the worst case, the button won't be quite as shiny or rounded, but it'll still convey that it should be touched.

The middle area is where everything else will go, the player's score, the current question, the answers to the question, and so on. Since all of that is easily achievable with HTML, CSS, and JavaScript, we're not going to worry about putting those elements into the template.

What did we do?

In this task we designed our user interface and also spelled out the interaction between the various views and widgets. We indicated what parts we knew would need to be localized (everything!) and then drew up a pretty version of it in our favorite graphics editor. From this version we can splice the various elements that need to be saved as images while also identifying what portions can be rendered with HTML, CSS, and JavaScript.

Designing the data model

The data model is very important to get right: this is how we'll store our questions, the answers to those questions, and which answer is the correct answer for each of the questions. We'll also define how we should interact with the model, that is, how do we get a question, ask it if the answer is correct, and so forth.

Getting ready

Let's get out our pencil and paper again, or if you'd prefer, a diagramming tool that you're comfortable with. What we're really trying to do in this step is to come up with the properties the model needs in order to store the questions, and the interactions it will need in order to properly do everything we're asking of it.

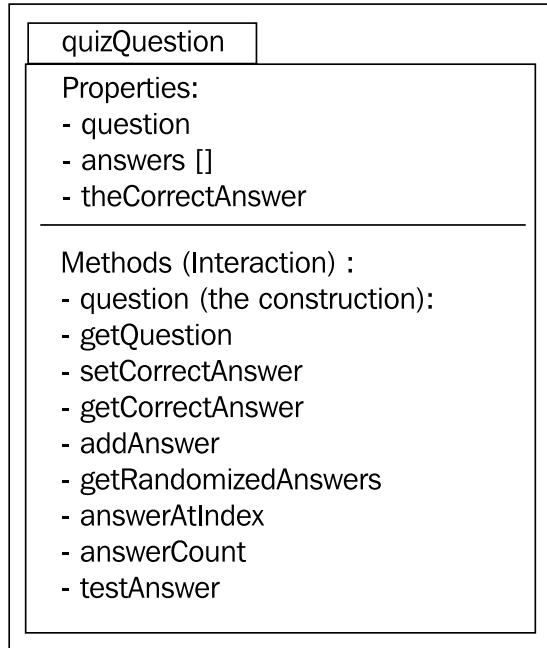
Getting on with it

We'll essentially have two data models: a single question, and a collection of questions. Let's start with what the question model should do:

- ▶ Store the actual question
- ▶ Have a list of all the possible answers
- ▶ Know the correct answer
- ▶ Set the question when created
- ▶ Return the question when asked
- ▶ Add an answer to its list of answers

- ▶ Return the list of answers when asked (in a random order)
- ▶ Set the correct answer
- ▶ Give the correct answer when asked
- ▶ Return a specific answer in the list when asked
- ▶ Check if a given answer is correct
- ▶ Return the number of answers

We can indicate this by creating a simple diagram as follows:

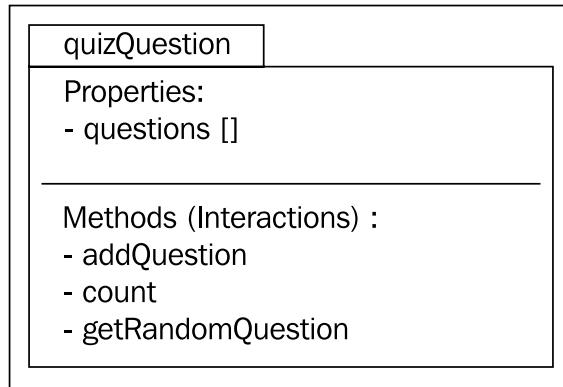


Our collection of questions should:

- ▶ Have a list of all the questions
- ▶ Be able to add a question to that list
- ▶ Return the total number of questions in the list
- ▶ Return a random question from the list

Let's Get Local! —————

The diagram covering these points would look like the following screenshot:



Having both of the models defined, let's come up with the questions we're going to ask, as well as the answers that will go along with them (for the full list of questions, see chapter1/www/models/quizQuestions.js in the download for this book):

#	English	Spanish
1	What is the color of the Sun?	¿Cuál es el color del Sol?
	Green	Verde
	White	Blanco
	Yellow (correct)	Amarillo (correct)
2	What is the name of the fourth planet?	¿Cuál es el nombre del cuarto planeta?
	Mars (correct)	Marzo (correct)
	Venus	Venus
	Mercury	Mercurio

With the design of our model complete, and the questions we're going to ask, this task is complete. Next we'll write the code to implement the model.

What did we do?

In this task we designed two data models: a single question and a collection of questions. We also determined the questions we were going to ask, along with their localized variants.

Implementing the data model

We'll be creating two JavaScript files in the `www/models` directory named `quizQuestion.js` and `quizQuestions.js`. The file `quizQuestion.js` will be the actual model: it will specify how the data should be formatted and how we can interact with it. `quizQuestions.js` will contain our actual question data.

Getting on with it

Before we define our model, let's define a namespace where it will live. This is an important habit to establish since it relieves us of having to worry about whether or not we'll collide with another function, object, or variable of the same name.

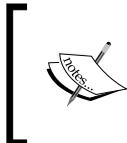
While there are various methods used to create a namespace, we're going to do it simply using the following code snippet:

```
// quizQuestion.js

var QQ = QQ || {};
```

Now that our namespace is defined, we can create our `question` object as follows:

```
QQ.Question = function ( theQuestion )
{
    var self = this;
```



Note the use of `self`: this will allow us to refer to the object using `self` rather than using `this`. (Javascript's `this` is a bit nuts, so it's always better to refer to a variable that we know will always refer to the object.)

Next, we'll set up the properties based on the diagram we created from step two using the following code snippet:

```
self.question = theQuestion;
self.answers = Array();
self.correctAnswer = -1;
```

Let's Get Local! —————

We've set the `self.correctAnswer` value to `-1` to indicate that, at the moment, any answer provided by the player is considered correct. This means you can ask questions where all of the answers are right.

Our next step is to define the methods or interactions the object will have. Let's start with determining if an answer is correct. In the following code, we will take an incoming answer and compare it to the `self.correctAnswer` value. If it matches, or if the `self.correctAnswer` value is `-1`, we'll indicate that the answer is correct:

```
self.testAnswer = function( theAnswerGiven )
{
    if ((theAnswerGiven == self.correctAnswer)
        || (self.correctAnswer == -1))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

We're going to need a way to access a specific answer, so we'll define the `answerAtIndex` function as follows:

```
self.answerAtIndex = function ( theIndex )
{
    return self.answers[ theIndex ];
}
```

To be a well-defined model, we should always have a way of determining the number of items in the model as shown in the following code snippet:

```
self.answerCount = function ()
{
    return self.answers.length;
}
```

Next, we need to define a method that allows an answer to be added to our object. Note that with the help of the return value, we return ourselves to permitting daisy-chaining in our code:

```
self.addAnswer = function( theAnswer )
{
    self.answers.push ( theAnswer );
    return self;
}
```

In theory we could display the answers to a question in the order they were given to the object. In practice, that would turn out to be a pretty boring game: the answers would always be in the same order, and chances would be pretty good that the first answer would be the correct answer. So let's give ourselves a randomized list using the following code snippet:

```
self.getRandomizedAnswers = function ()
{
    var randomizedArray = Array();
    var theRandomNumber;
    var theNumberExists;

    // go through each item in the answers array
    for (var i=0; i<self.answers.length; i++)
    {

        // always do this at least once
        do
        {
            // generate a random number less than the
            // count of answers
            theRandomNumber = Math.floor ( Math.random() *
                self.answers.length );
            theNumberExists = false;

            // check to see if it is already in the array
            for (var j=0; j<randomizedArray.length; j++)
            {
                if (randomizedArray[j] == theRandomNumber)
                {
                    theNumberExists = true;
                }
            }

            // If it exists, we repeat the loop.
        } while ( theNumberExists );

        // We have a random number that is unique in the
        // array; add it to it.
        randomizedArray.push ( theRandomNumber );
    }

    return randomizedArray;
}
```

Let's Get Local! —————

The randomized list is just an array of numbers that indexes into the `answers []` array. To get the actual answer, we'll have to use the `answerAtIndex()` method.

Our model still needs a way to set the correct answer. Again, notice the return value in the following code snippet permitting us to daisy-chain later on:

```
self.setCorrectAnswer = function ( theIndex )
{
    self.correctAnswer = theIndex;
    return self;
}
```

Now that we've properly set the correct answer, what if we need to ask the object what the correct answer is? For this let's define a `getCorrectAnswer` function using the following code snippet:

```
self.getCorrectAnswer = function ()
{
    return self.correctAnswer;
}
```

Of course, our object also needs to return the question given to it whenever it was created; this can be done using the following code snippet:

```
self.getQuestion = function()
{
    return self.question;
}
```

That's it for the `question` object. Next we'll create the container that will hold all of our questions using the following code line:

```
QQ.questions = Array();
```

We could go the regular object-oriented approach and make the container an object as well, but in this game we have only one list of questions, so it's easier to do it this way.

Next, we need to have the ability to add a question to the container, this can be done using the following code snippet:

```
QQ.addQuestion = function (theQuestion)
{
    QQ.questions.push ( theQuestion );
}
```

Like any good data model, we need to know how many questions we have; we can know this using the following code snippet:

```
QQ.count = function ()  
{  
    return QQ.questions.length;  
}
```

Finally, we need to be able to get a random question out of the list so that we can show it to the player; this can be done using the following code snippet:

```
QQ.getRandomQuestion = function ()  
{  
    var theQuestion = Math.floor (Math.random() * QQ.count());  
    return QQ.questions[theQuestion];  
}
```

Our data model is officially complete. Let's define some questions using the following code snippet:

```
// quizQuestions.js  
//  
// QUESTION 1  
//  
QQ.addQuestion ( new QQ.Question ( "WHAT_IS_THE_COLOR_OF_THE_SUN?" )  
    .addAnswer( "YELLOW" )  
    .addAnswer( "WHITE" )  
    .addAnswer( "GREEN" )  
    .setCorrectAnswer ( 0 ) );
```

Notice how we attach the `addAnswer` and `setCorrectAnswer` methods to the new question object. This is what is meant by daisy-chaining: it helps us write just a little bit less code.

You may be wondering why we're using upper-case text for the questions and answers. This is due to how we'll localize the text, which is next:

```
PKLOC.addTranslation ( "en", "WHAT_IS_THE_COLOR_OF_THE_SUN?", "What is  
the color of the Sun?" );  
PKLOC.addTranslation ( "en", "YELLOW", "Yellow" );  
PKLOC.addTranslation ( "en", "WHITE", "White" );  
PKLOC.addTranslation ( "en", "GREEN", "Green" );  
  
PKLOC.addTranslation ( "es", "WHAT_IS_THE_COLOR_OF_THE_SUN?", "¿Cuál  
es el color del Sol?" );
```

Let's Get Local! —————

```
PKLOC.addTranslation ( "es", "YELLOW", "Amarillo" );
PKLOC.addTranslation ( "es", "WHITE", "Blanco" );
PKLOC.addTranslation ( "es", "GREEN", "Verde" );
```

The questions and answers themselves serve as keys to the actual translation. This serves two purposes: it makes the keys obvious in our code, so we know that the text will be replaced later on, and should we forget to include a translation for one of the keys, it'll show up in uppercase letters.

PKLOC as used in the earlier code snippet is the namespace we're using for our localization library. It's defined in `www/framework/localization.js`. The `addTranslation` method is a method that adds a translation to a specific locale. The first parameter is the locale for which we're defining the translation, the second parameter is the key, and the third parameter is the translated text.

The `PKLOC.addTranslation` function looks like the following code snippet:

```
PKLOC.addTranslation = function (locale, key, value)
{
    if (PKLOC.localizedText [locale])
    {
        PKLOC.localizedText [locale] [key] = value;
    }
    else
    {
        PKLOC.localizedText [locale] = {};
        PKLOC.localizedText [locale] [key] = value;
    }
}
```

The `addTranslation` method first checks to see if an array is defined under the `PKLOC.localizedText` array for the desired locale. If it is there, it just adds the key/value pair. If it isn't, it creates the array first and then adds the key/value pair. You may be wondering how the `PKLOC.localizedText` array gets defined in the first place. The answer is that it is defined when the script is loaded, a little higher in the file:

```
PKLOC.localizedText = {};
```

Continue adding questions in this fashion until you've created all the questions you want. The `quizQuestions.js` file contains ten questions. You could, of course, add as many as you want.

What did we do?

In this task, we created our data model and created some data for the model. We also showed how translations are added to each locale.

What else do I need to know?

Before we move on to the next task, let's cover a little more of the localization library we'll be using. Our localization efforts are split into two parts: translation and data formatting.

For the translation effort, we're using our own simple translation framework, literally just an array of keys and values based on locale. Whenever code asks for the translation for a key, we'll look it up in the array and return whatever translation we find, if any. But first, we need to determine the actual locale of the player, using the following code snippet:

```
// www/framework/localization.js

PKLOC.currentUserLocale = "";

PKLOC.getUserLocale = function()
{
```

Determining the locale isn't hard, but neither is it as easy as you would initially think. There is a property (`navigator.language`) under WebKit browsers that is technically supposed to return the locale, but it has a bug under Android, so we have to use the `userAgent`. For WP7, we have to use one of three properties to determine the value.

Because that takes some work, we'll check to see if we've defined it before; if we have, we'll return that value instead:

```
if (PKLOC.currentUserLocale)
{
    return PKLOC.currentUserLocale;
}
```

Next, we determine the current device we're on by using the `device` object provided by Cordova. We'll check for it first, and if it doesn't exist, we'll assume we can access it using one of the four properties attached to the `navigator` object using the following code snippet:

```
var currentPlatform = "unknown";
if (typeof device != 'undefined')
{
    currentPlatform = device.platform;
}
```

We'll also provide a suitable default locale if we can't determine the user's locale at all as seen in the following code snippet:

```
var userLocale = "en-US";
```

Let's Get Local! —————

Next, we handle parsing the user agent if we're on an Android platform. The following code is heavily inspired by an answer given online at <http://stackoverflow.com/a/7728507/741043>.

```
if (currentPlatform == "Android")
{
    var userAgent = navigator.userAgent;

    var tempLocale = userAgent.match(/Android.*([a-zA-Z]{2})-[a-zA-Z]{2}\)/);
    if (tempLocale)
    {
        userLocale = tempLocale[1];
    }
}
```

If we're on any other platform, we'll use the `navigator` object to retrieve the locale as follows:

```
else
{
    userLocale = navigator.language ||
                 navigator.browserLanguage ||
                 navigator.systemLanguage ||
                 navigator.userLanguage;
}
```

Once we have the locale, we return it as follows:

```
PKLOC.currentUserLocale = userLocale;
return PKLOC.currentUserLocale;
}
```

This method is called over and over by all of our translation codes, which means it needs to be efficient. This is why we've defined the `PKLOC.currentUserLocale` property. Once it is set, the preceding code won't try to calculate it out again. This also introduces another benefit: we can easily test our translation code by overwriting this property. While it is always important to test that the code properly localizes when the device is set to a specific language and region, it often takes considerable time to switch between these settings. Having the ability to set the specific locale helps us save time in the initial testing by bypassing the time it takes to switch device settings. It also permits us to focus on a specific locale, especially when testing.

Translation of text is accomplished by a convenience function named `__T()`. The convenience functions are going to be our only functions outside of any specific namespace simply because we are aiming for easy-to-type and easy-to-remember names that aren't arduous to add to our code. This is especially important since they'll wrap every string, number, date, or percentage in our code.

The `__T()` function depends on two functions: `substituteVariables` and `lookupTranslation`. The first function is defined as follows:

```
PKLOC.substituteVariables = function ( theString, theParms )  
{  
    var currentValue = theString;  
  
    // handle replacement variables  
    if (theParms)  
    {  
        for (var i=1; i<=theParms.length; i++)  
        {  
            currentValue = currentValue.replace("%" + i, theParms[i-1]);  
        }  
    }  
  
    return currentValue;  
}
```

All this function does is handle the substitution variables. This means we can define a translation with `%1` in the text and we will be able to replace `%1` with some value passed into the function.

The next function, `lookupTranslation`, is defined as follows:

```
PKLOC.lookupTranslation = function ( key, theLocale )  
{  
    var userLocale = theLocale || PKLOC.getUserLocale();  
  
    if ( PKLOC.localizedText[userLocale] )  
    {  
        if ( PKLOC.localizedText[userLocale] [key.toUpperCase()] )  
        {  
            return PKLOC.localizedText[userLocale] [key.toUpperCase()];  
        }  
    }  
  
    return null;  
}
```

Let's Get Local!

Essentially, we're checking to see if a specific translation exists for the given key and locale. If it does, we'll return the translation, but if it doesn't, we'll return `null`. Note that the key is always converted to uppercase, so case doesn't matter when looking up a translation.

Our `__T()` function looks as follows:

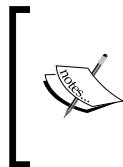
```
function __T(key, parms, locale)
{
    var userLocale = locale || PKLOC.getUserLocale();
    var currentValue = "";
```

First, we determine if the translation requested can be found in the locale, whatever that may be. Note that it can be passed in, therefore overriding the current locale. This can be done using the following code snippet:

```
if (! (currentValue=PKLOC.lookupTranslation(key,
                                              userLocale)) )
{
```

Locales are often of the form `xx-yy`, where `xx` is a two-character language code and `yy` is a two-character character code. My locale is defined as `en-US`. Another player's might be defined as `es-ES`.

If you recall, we defined our translations only for the language. This presents a problem: the preceding code will not return any translation unless we defined the translation for the language and the country.



Sometimes it is critical to define a translation specific to a language and a country. While various regions may speak the same language from a technical perspective, idioms often differ. If you use an idiom in your translation, you'll need to localize them to the specific region that uses them, or you could generate potential confusion.

Therefore, we chop off the country code, and try again as follows:

```
userLocale = userLocale.substr(0,2);

if (! (currentValue=PKLOC.lookupTranslation(key, userLocale)) )
{
```

But we've only defined translations for English (`en`) and Spanish(`es`)! What if the player's locale is `fr-FR` (French)? The preceding code will fail, because we've not defined any translation for the `fr` language (French). Therefore, we'll check for a suitable default, which we've defined to be `en-US`, American English:

```
userLocale = "en-US";
if (! (currentValue=PKLOC.lookupTranslation(key, userLocale)) )
{
```

Of course, we are now in the same boat as before: there are no translations defined for `en-US` in our game. So we need to fall back to `en` as follows:

```
userLocale = "en";
if (! (currentValue=PKLOC.lookupTranslation(key, userLocale)) )
{
```

But what happens if we can't find a translation at all? We could be mean and throw a nasty error, and perhaps you might want to do exactly that, but in our example, we're just returning the incoming key. If the convention of capitalizing the key is always followed, we'll still be able to see that something hasn't been translated.

```
    currentValue = key;
}
}
}
}
```

Finally, we pass the `currentValue` parameter to the `substituteVariables` property in order to process any substitutions that we might need as follows:

```
return PKLOC.substituteVariables( currentValue, parms
);
}
```

Implementing the start view

To create our view, we need to create the file for it first. The files should be called `startView.html`, and should live under the `www/views` directory. The view we're creating will end up looking like the following screenshot for iOS:



For Android (localized to Spanish), the view will be as follows:



Before we actually create the view though, let's define the structure of our view. Depending upon the framework in use, the structure of a view can be vastly different. For the YASMF framework, our view will consist of some HTML that will depend on some pre-defined CSS, and some JavaScript defined below that same HTML. You could easily make the case that the JavaScript and inline styles should be separated out as well, and if you wish, you can do so.

The HTML portion for all our views will be of the following form:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div id="theView_AppTitle"></div>
    <button class="barButton backButton"
      id="theView_backButton" style="left:10px" ></button>
  </div>
  <div class="content avoidNavigationBar avoidToolBar"
    id="theView_anId">
  </div>
  <div class="toolBar">
    <button class="barButton" id="theView_aButton"
      style="right:10px"></button>
  </div>
</div>
```

As you can see, there's no visible text anywhere in this code. Since everything must be localized, we'll be inserting the text programmatically via JavaScript.

The `viewBackground` class will be our view's container: everything related to the view's structure is defined within. The style is defined in `www/framework/base.css` and `www/style/style.css`; the latter is for our app's custom styles.

The `navigationBar` class indicates that the `div` class is just a navigation bar. For iOS users, this has instant meaning, but it should be pretty clear to everyone else: this bar holds the title of the view, as well as any buttons that serve for navigation (such as a **back** button). Notice that the **title** and **back** button both have `id` values. This value makes it easy for us to access them in our JavaScript later on. Notice also that we are namespacing these `id` values with the view name and an underscore; this is to prevent any issues with using the same `id` twice.

The next `div` class is given the class of `content avoidNavigationBar avoidToolBar`; this is where all the content will go. The latter two classes specify that it should be offset from the top of the screen and short enough to avoid both the navigation bar (already defined) and the toolbar (coming up).

Finally, the toolbar is defined. This is a bar much like the navigation bar, but is intended to hold buttons that are related to the view. For Android this would be commonly shown near or at the top of the screen, while for iPhone and WP7 display this bar is at the bottom. (iPad, on the other hand, would display this just below the navigation bar or on the navigation bar. We'll worry about that in *Project 10, Scaling Up.*)

Below this HTML block, we'll define any templates we may need for localization, and then finally, any JavaScript we need.

Getting on with it

With these pointers in mind, let's create our start view, which should be named `startView.html` in the `www/views` directory as follows:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div id="startView_AppTitle"></div>
  </div>
  <div class="content avoidNavigationBar avoidToolBar"
    id="startView_welcome">
  </div>
  <div class="toolBar">
    <button class="barButton" id="startView_startButton"
      style="right:10px"></button>
  </div>
</div>
```

The preceding code looks almost exactly like our view template defined earlier except that we're missing a back button. This is due to the fact that the first view we display to the user doesn't have anything to go back to, so we omit that button. The `id` values have also changed to include the name of our view.

None of these define what our view will look like, though. To determine that, we need to override our framework styles in `www/framework/base.css` by setting them in `www/style/style.css`.

First, to define the look of `navigationBar`, we use the glossy black bar from our template defined earlier in this project as follows:

```
.navigationBar
{
  background-image: url(../images/NavigationBar.png);
  color: #FFF;
  background-color: transparent;
}
```

The toolbar is defined similarly as follows:

```
.toolBar
{
    background-image: url(../images/ToolBar.png);
}
```

The view's background is defined as follows:

```
.viewBackground
{
    background-image: url(../images/Background.jpg);
    background-size: cover;
}
```

That's everything needed to make our start view start to look like a real app. Of course, there's a lot of pre-built stuff in `www/framework/base.css`, which you're welcome to analyze and reuse in your own projects.

Now that we've defined the view and the appearance, we need to define some of the view's content. We're going to do this by using a couple of hidden `div` elements that have the locale attached to their `id` values, as follows:

```
<div id="startView_welcome_en" class="hidden">
<h2>PhoneGap-Hotshot Sample Application</h2>
<h3>Chapter 1: Let's Get Local!</h3>
<p>This application demonstrates localization
    between two languages, based on your device's
    language settings. The two languages implemented
    are English and Spanish.</p>
</div>

<div id="startView_welcome_es" class="hidden">
<h2>Ejemplo de aplicación de PhoneGap-Hotshot</h2>
<h3>Capítulo 1: Let's Get Local!</h3>
<p>Esta aplicación muestra la localización
    entre los dos idiomas, sobre la base de su dispositivo de
    la configuración de idioma. Las dos lenguas aplicadas
    son Inglés y Español.</p>
</div>
```

These two `div` elements are classed `hidden` so that they won't be visible to the player. We'll then use some JavaScript to copy their content to the content area inside the view. Easier than using the `__T()` and `PKLOC.addTranslation()` functions for all that text, isn't it?

Next comes the JavaScript as follows:

```
<script>
    var startView = $ge("startView") || {};
```

Our first act is to put all our script into a namespace. Unlike most of our other namespace definitions, we're actually going to piggyback onto the "startView" element (which the astute reader will notice has not been defined yet; that'll be near the end of this project). While the element is a proper DOM element, it also serves as a perfect place for us to attach to, as long as we avoid any of the cardinal sins of using the DOM method names as our own, which, I promise, we won't do.

You might be wondering what `$ge` does. Since we're not including any JavaScript framework like jQuery, we don't have a convenience method to get an element by its ID. jQuery does this with the `$(...)` method, and because you might actually be using jQuery along with the framework we're using, I chose to use the `$ge()` method, short for `get element`. It's defined in `www/framework/utility.js` like the following code snippet and all it does is act as a shortened version of `document.getElementById`:

```
function $ge ( elementId )
{
    return document.getElementById ( elementId );
}
```

Getting back to our start view script, we define what needs to happen when the view is initialized. Here we *hook* into the various buttons and other interface elements that are in the view, as well as localize all the text and content as follows:

```
startView.initializeView = function ()
{
    PKLOC.addTranslation ("en", "APP_TITLE_IMAGE",
        "AppTitle-enus.png");
    PKLOC.addTranslation ("es", "APP_TITLE_IMAGE",
        "AppTitle-eses.png");

    startView.applicationTitleImage =
        $ge("startView_AppTitle");
    startView.applicationTitleImage.style.backgroundImage =
        "url('./images/" + __T("APP_TITLE_IMAGE") + "')";
```

This is our first use of the `__T()` function. This is how we can properly localize an image. The `APP_TITLE_IMAGE` key is set to point at either the English version or the Spanish version of the title image, and the `__T()` function returns the correct one based on our locale.

```
PKLOC.addTranslation ("en", "START", "Start");
PKLOC.addTranslation ("es", "START", "Comenzar");

startView.startButton = $ge("startView_startButton");
startView.startButton.innerHTML = __T("START");
```

Now we've properly localized our start button, but how do we make it do anything? We use a little function defined in `www/framework/ui-core.js` called `PKUI.CORE.addTouchListener()` as follows:

```
PKUI.CORE.addTouchListener( startView.startButton,
    "touchend", startView.startGame );
```

Finally, we need to display the correct `welcome` text in the content area using the following code snippet:

```
var theWelcomeContent = $geLocale("startView_welcome");
$ge("startView_welcome").innerHTML =
    theWelcomeContent.innerHTML;
}
```

We now introduce another convenience function: the `$geLocale()` function. This function acts like the `$ge()` function, except that it assumes there will be a locale appended to the ID of the element we're asking for. It's defined in the same file (`utility.js`) and looks like the following:

```
function $geLocale ( elementId )
{
    var currentLocale = PKLOC.getUserLocale();
    var theLocalizedElementId = elementId + "_" + currentLocale;
    if ($ge(theLocalizedElementId)) { return
        $ge(theLocalizedElementId); }

    theLocalizedElementId = elementId + "_" +
        currentLocale.substr(0,2);
    if ($ge(theLocalizedElementId)) { return
        $ge(theLocalizedElementId); }

    theLocalizedElementId = elementId + "_en-US";
    if ($ge(theLocalizedElementId)) { return
        $ge(theLocalizedElementId); }

    theLocalizedElementId = elementId + "_en";
```

```
if ($ge(theLocalizedElementId)) { return  
    $ge(theLocalizedElementId); }  
return $ge( elementId );  
}
```

Much like our `__T()` function, it attempts to find an element with our full locale attached (that is, `_xx-YY`). If it can't find it, it tries `_xx`, and here it should succeed if our locale is English- or Spanish-speaking. If it isn't, we'll then look for `_en-US`, and if that isn't found, we'll look for `_en`. If no suitable element is found, we'll return the original element –which in our case doesn't exist, which means we'll return "undefined".

Next up in our start view script, we have the function that is called whenever the start button is tapped as shown in the following code snippet:

```
startView.startGame = function()  
{  
    PKUI.CORE.pushView ( gameView );  
}  
  
</script>
```

Real short, but packs a punch. This displays our game view to the player, which actually starts the game. For devices that support it (as this was being written, iOS and Android), the player also sees a nice animation between this view (start) and the next one (game).

If you want to know more about how the `pushView()` method works, visit <https://github.com/photokandyStudios/YASMF/wiki/PKUI.CORE.pushView>.

Whew! That was a lot of work for a pretty simple view. Thankfully, most of the work is actually done by the framework, so our actual `startView.html` file is pretty small.

What did we do?

We implemented our start view, which is presented to the player when they first launch the app. We properly localized the view's title image based on the player's locale, and we also properly localized HTML content based on the locale.

We defined the various hooks and text for the widgets on the view such as the **Start** button, and attached touch listeners to them to make them function correctly.

We covered a portion of the framework that provides support for pushing views onto the screen as well.

What else do I need to know?

It probably doesn't take much to guess, but there are several complementary functions to the `pushView` method: `popView`, `showView`, and `hideView`.

The `popView` function does the exact opposite of `pushView`, that is, it moves the views right (instead of left) by popping them off the view stack.

The `showView` and `hideView` functions do essentially the same thing, but simpler. They don't do any animation at all. Furthermore, since they don't involve any other view on the stack, they are most useful at the beginning of an app when we have to figure out how to display our very first view with no previous view to animate.

If you want to know more about view management, you might want to visit <https://github.com/photokandyStudios/YASMF/wiki/Understanding-the-View-Stack-and-View-Management> and explore <https://github.com/photokandyStudios/YASMF/wiki/PKUI.CORE>.

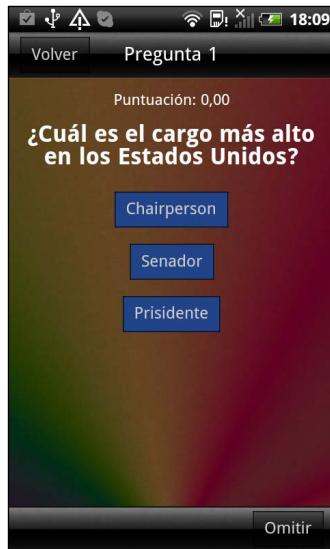
Implementing our game view

To get started, create a file named `gameView.html` under the `www/views` directory. When we're done, we'll have a view that looks like the following screenshot for iOS:

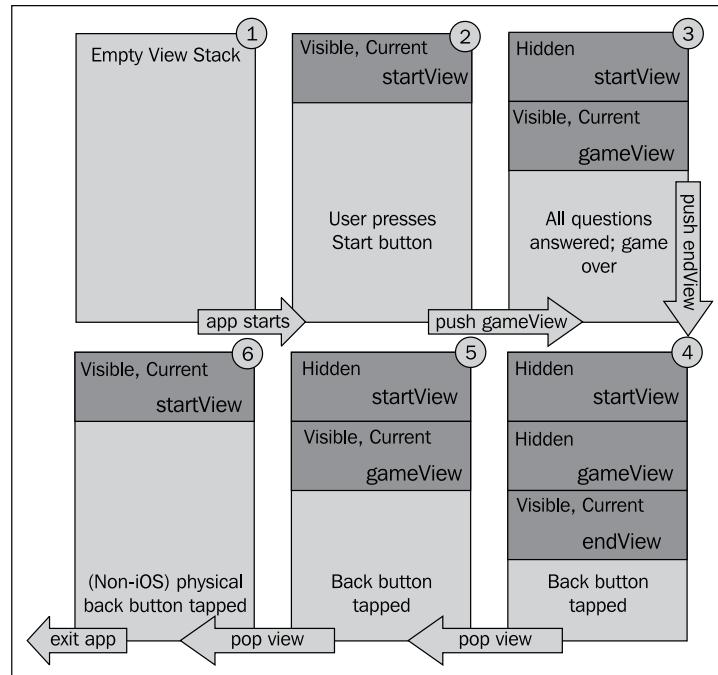


Let's Get Local!

For Android, the view will look like the following screenshot:



Now, before we get too deep into the view itself, let's go over the view stack and how it helps us deal with navigation. The view stack is shown in the following screenshot:



The view stack is really just a stack that maintains the list of previously visible views and the currently visible view. When our app first starts, the stack will be empty as identified in the first step in the preceding screenshot. Then, the `startView` view is pushed onto the stack using the `showView` method, and you have the stack in (2). When the player taps the **Start** button, the `gameView` view is pushed onto the stack, which results in the stack as seen in (3). Then, when the game is over, we'll push the `endView` view on the stack, resulting in (4).

Because we're tracking all these views, including the ones that are no longer visible (especially at the end of the game), it makes it easy to go back to a previous view. For iOS, this is done via the **back** button. For Android, the device often has a physical back button that is used instead. Regardless of how a `back` event is triggered, we need to be able to go backwards in the stack.

Let's say that the user now decides to go back in the stack; we would have the stack in (5). If they decide to go back another step, (6) would result. At this point, iOS would permit no further backtracking, but for Android, another `back` event should exit the user out of the app.

Getting on with it

The game view will be very similar to our start view, except that it is a little more complicated. After all, it plays an entire game. Thankfully, there's really nothing terribly new here, so it should be smooth going.

Let's start with the HTML portion of the view given as follows:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div id="gameView_title"></div>
    <button class="barButton backButton"
      id="gameView_backButton" style="left:10px"></button>
  </div>
  <div class="content avoidNavigationBar avoidToolBar"
    id="gameView_gameArea">
    <div id="gameView_scoreArea" style="height:1em; text-align: center;"></div>
    <div id="gameView_questionArea" style="text-align: center"></div>
  </div>
  <div class="toolBar">
    <button class="barButton" id="gameView_nextButton"
      style="right:10px" ></button>
  </div>
</div>
```

I've highlighted what's new in the earlier code, but there is not much as you can see. First we've defined a `back` button that lives in the navigation bar, and in the content area we've defined two new areas: one for the player's score, and another for the actual question (and answers).

Up next, while similar to the localized content in the start view, we have templates that specify how a question and its answers are displayed; this is given as follows:

```
<div id="gameView_questionTemplate" class="hidden">
  <h2>%QUESTION%</h2>
  <div style="text-align:center;">%ANSWERS%</div>
</div>
```

First, we define the question template, which consists of a second-level heading that will have the question's text, and a div element that will contain all the answers. But what will the answers look like? That's next:

```
<div id="gameView_answerTemplate" class="hidden">
  <button class="barButton answerButton"
    onclick="gameView.selectAnswer(%ANSWER_INDEX%); ">%ANSWER%
  </button><br/>
</div>
```

Each answer will be presented as a button with the answer text inside, and an `onclick` event attached to call the `gameView.selectAnswer()` method with the selected answer.

Of course, as these are templates, they don't appear to the player, and so they are given the `hidden` class. But we'll definitely make use of them in our JavaScript when we construct an actual random question to display to the player. Let's go over the script now:

```
<script>

var gameView = $ge("gameView") || {};
gameView.questionNumber = -1;
gameView.score = 0;
gameView.theCurrentQuestion;
```

By now you should be familiar with our namespacing technique, which comes first in our code. After that, though, we define the properties in our view. The question number, which will act as our counter so that when it reaches ten, we know the game is over; the score; and the current question. The latter isn't obvious, but it will be an actual question object, not an index to the object.

After that, we have the `initializeView` function, which will wire up all the widgets and do the localization of the text, as seen in the following code snippet:

```
gameView.initializeView = function ()
{
  PKUTIL.include ( ["./models/quizQuestions.js",
    "./models/quizQuestion.js"] );
```

```
gameView.viewTitle = $ge("gameView_title");
gameView.viewTitle.innerHTML = __T("APP_TITLE");

gameView.backButton = $ge("gameView_backButton");
gameView.backButton.innerHTML = __T("BACK");
PKUI.CORE.addTouchListener(gameView.backButton, "touchend",
    function () { PKUI.CORE.popView(); });

gameView.nextButton = $ge("gameView_nextButton");
gameView.nextButton.innerHTML = __T("SKIP");

PKUI.CORE.addTouchListener(gameView.nextButton, "touchend",
    gameView.nextQuestion);

gameView.scoreArea = $ge("gameView_scoreArea");
gameView.questionArea = $ge("gameView_questionArea");
}
```

I've highlighted a few areas in the preceding code block. The last ones are more or less the same, as we're storing the `gameView_scoreArea` and `gameView_questionArea` elements into properties for later use, so that's not really anything new. What is new about it is that we aren't loading any content into them yet.

The second highlight is not something you'd really ever add to a production game. You may ask, so why is it here? The idea is that this button lets us skip the current question without a penalty. Why? The answer is testing. I don't want to have to tap through an answer, tap through the alert saying if I got it right or wrong a million times to see if the localization is working for all the questions. Hence, skip was born.

The first highlight though, is more interesting. It's a JavaScript include. "Wait," I hear you saying, "JavaScript doesn't do includes." And you'd be right.

But, it is possible to simulate an include by using `XMLHttpRequest`, which is often referred to as AJAX. With this short include statement, we're asking the browser to load the two referenced JavaScript files (`quizQuestions.js` and `quizQuestion.js`) on our behalf. It's important that this happens too; otherwise, our game would have no questions!

The `PKUTIL.include()` function is defined in `www/framework/utility.js`. We'll worry about the full implementation details a little later in this project, but it would suffice to say, it does what it says. The scripts are loaded and waiting for us when we need to use the questions. (At this point the reader with a gazillion questions is asking this key question, "Does the order matter?", the answer is, "Yes." And you'll see why in a short bit.)

Let's Get Local! —————

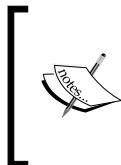
So now that we have the initialization for `gameView` down, let's look at another key method: `viewWillAppear`. It is shown in the following code snippet:

```
gameView.viewWillAppear = function ()  
{  
    gameView.questionNumber = 1;  
    gameView.score = 0;  
    gameView.nextQuestion();  
}
```

The latter part of this code is fairly innocuous. We set the question number to 1, the score to zero, and call the `nextQuestion()` method, which, as it turns out, renders the next question and displays it to the player.

The `viewWillAppear()` function, as you may remember, is called by `PKUI.CORE.pushView()` and `PKUI.CORE.showView()` methods just prior to the actual animation that renders the view onscreen. Therefore, the act of the **Start** button on the start view pushing the game view on the stack will call this function, and start the game.

It also works when we're coming back to the view by popping the end view off the stack. We'll receive a `viewWillAppear` notification, reset the game, and it's as if the user gets a whole new game. It's almost magic!



To those who have done any amount of Objective-C programming for iOS using Apple's frameworks, I'll apologize right now for using the concepts in the framework. It's just that, well, they fit the view model so well! If you prefer Android's methodology, or Microsoft's, feel free to substitute. I just happen to like the framework Apple has built up for their platform.

Of course, we need to actually do something when the back button is pressed, the code for it is as follows:

```
gameView.backButtonPressed = function ()  
{  
    PKUI.CORE.popView();  
}
```

The `popView()` method is literally the reverse of `pushView`. It takes the currently visible view (`gameView`), pops it off the stack, and displays the underlying view, in this case, `startView`. The best thing to do here would be to prompt the player if they really wanted to do this; it will end their game, perhaps prematurely. For now, as an example, we'll leave it at this.

Next, we need to define how a question is displayed on the screen. We do that in `nextQuestion()`, as seen in the following code snippet:

```
gameView.nextQuestion = function ()  
{
```

First, we'll get a random question from the `QQ` namespace:

```
// load the next question into the view  
gameView.theCurrentQuestion = QQ.getRandomQuestion();
```

Next, we get our templates:

```
var theQuestionTemplate =  
    $ge("gameView_questionTemplate").innerHTML;  
var theAnswerTemplate =  
    $ge("gameView_answerTemplate").innerHTML;
```

Now that we have our templates, we'll replace all occurrences of "`%QUESTION%`" with the translated question, as shown in the following code snippet:

```
theQuestionTemplate = theQuestionTemplate.replace(  
    "%QUESTION%",  
    __T(gameView.theCurrentQuestion.getQuestion()) );
```

Generating the answers is a little more tricky. There may be two, three, or more answers for any one question, so we'll ask the question for a list of randomized answers first, and then loop through that list while building up an HTML string, as shown in the following code snippet:

```
var theAnswers =  
    gameView.theCurrentQuestion.getRandomizedAnswers();  
  
var theAnswersHTML = "";  
  
for (var i=0; i<theAnswers.length; i++)  
{
```

For each answer, we'll replace the `%ANSWER%` text with the translated text of the answer, and "`%ANSWER_INDEX%`" with the current index (`i`), as shown in the following screenshot:

```
theAnswersHTML += theAnswerTemplate.replace(  
    "%ANSWER%",  
    __T(gameView.theCurrentQuestion.answerAtIndex(  
        theAnswers[i] ) ).replace ( "%ANSWER_INDEX%",  
        theAnswers[i] );  
}
```

Let's Get Local! —————

Now that we've got the HTML for our answers, we can replace %ANSWERS% in the question template with it as follows:

```
theQuestionTemplate = theQuestionTemplate.replace (
    "%ANSWERS%", theAnswersHTML );
```

At this point, we can display the question to the player:

```
gameView.questionArea.innerHTML = theQuestionTemplate;
```

We also want to update the player's score. We're going to have an artificially absurd scoring system to highlight whether or not our localization is working correctly. Note that the 2 in the following code snippet specifies we want two decimal places in the score.

```
gameView.scoreArea.innerHTML = __T("SCORE_%1",
    [ __N(gameView.score, "2") ]);
```

We'll also update the view's title with the current question number. This time the "0" following code snippet indicates no decimal points:

```
gameView.viewTitle.innerHTML = __T("QUESTION_%1",
    [ __N(gameView.questionNumber, "0") ]);

}
```

All of this is well and good, but it does nothing without the user being able to select an answer, which is where the next function comes in:

```
gameView.selectAnswer = function ( theAnswer )
{
```

First, we'll ask the current question if the answer selected is correct using the following code snippet:

```
if (gameView.theCurrentQuestion.testAnswer ( theAnswer ))
{
```

If it is, we'll tell the user they got it right, and increment their score as follows:

```
    alert (__T("CORRECT"));
    gameView.score += 483.07;
}
else
{
```

But if it is wrong, we'll indicate that it is incorrect, and decrement their score (We're mean, I guess. Not really though-we want to test that negative numbers work too.), using the following code snippet:

```
    alert (_T("INCORRECT"));
    gameView.score -= 192.19;
}
```

Next, we check to see if we've asked the last question in the set as follows:

```
if (gameView.questionNumber >= 10)
{
```

If we have, we'll communicate the score to the end view and push it onto the stack. This ends the game, using the following code snippet:

```
    endView.setScore ( gameView.score );
    PKUI.CORE.pushView ( endView );
}
else
{
```

In this case, we've got more questions to answer, so we load the next question as follows:

```
    gameView.questionNumber++;
    gameView.nextQuestion();
}
}

</script>
```

With that, we're done with the game view. Tell me, that wasn't too difficult, was it?

What did we do?

We implemented the actual game in one view. We also learned how to handle the back button on Android, and back navigation on iOS. We also gained an understanding of how to use HTML blocks that are hidden as templates for dynamic content.

What else do I need to know?

If you remember, I mentioned that we'd talk about that wonderful little `include` function a little more. Let's look at it a bit closer:

```
PKUTIL.include = function ( theScripts, completion )  
{
```

First off, let me clue you into something: we're using recursion here to load the scripts. So, as you'll see in the following code, we're testing the length of the incoming array, and if it is zero, we call the `completion` method passed to us. This allows us—if we like—to have code called after all the scripts are loaded. This code block is as follows:

```
var theNewScripts = theScripts;  
if (theNewScripts.length == 0)  
{  
    if (completion)  
    {  
        completion();  
    }  
    return;  
}
```

In the next section, we'll pop off the next script to load. This also explains that the array must contain the scripts in reverse order of their dependencies. Yes, you could reverse the array yourself and you should, but I wanted to make the point. To pop off the script the following code instruction is used:

```
var theScriptName = theNewScripts.pop();
```

Then we call another previously unknown function, `PKUTIL.load()`. This method takes the script filename, and then calls the `completion` function we've given it. It will call it regardless of success or failure. Notice that it is an incoming parameter to the `completion` function. This function is shown in the following screenshot:

```
PKUTIL.load ( theScriptName, true, function ( success, data )  
{
```

If the script was successfully loaded, we create a `SCRIPT` DOM element and add the data to it. It is important to note that nothing happens with the script until we actually attach it to the DOM. We do this by appending the child to the `BODY`. It is at this point that whatever is in the script will be executed. This conditional `if` block is shown in the following code snippet:

```
if (success)  
{  
    var theScriptElement = document.createElement("script");
```

```
theScriptElement.type = "text/javascript";
theScriptElement.charset = "utf-8";
theScriptElement.text = data;
document.body.appendChild ( theScriptElement ); // add
it as a script tag
}
```

If we fail to load the script, we'll generate a log message on the console. You could make a case that something worse should happen, like a fatal error that stops everything, but this also permits loading libraries that may or may not be there and taking advantage of them if they happen to exist. Perhaps not a feature one would use frequently, but useful at times nonetheless. The conditional `else` block is as follows:

```
else
{
    console.log ("WARNING: Failed to load " + theScriptName );
}
```

And say hello to our little friend, recursion. We call ourselves with the array of script names (minus the one we just popped), with the `completion` function, and sooner or later, we'll end up with no items in the array. Then, the `completion` function will be called as seen in the following code block:

```
PKUTIL.include ( theNewScripts, completion );
}
);
}
}
```

The `PKUTIL.load()` function is another interesting beast, which must work correctly for our `includes` to work. It's defined something like the following (for full implementation details, visit <https://github.com/photokandyStudios/YASMF/blob/master/framework/utility.js#L126>):

```
PKUTIL.load = function ( theFileName, aSync, completion )
{
```

First, we'll check to see if the browser understands `XMLHttpRequest`. If it doesn't, we'll call `completion` with a failure notice and a message describing that we couldn't load anything, as shown in the following code block:

```
if (!window.XMLHttpRequest)
{
    if (completion)
    {
        completion ( PKUTIL.COMPLETION_FAILURE,
```

```
        "This browser does not support  
        XMLHttpRequest." );  
    return;  
}  
}
```

Next we set up the XMLHttpRequest, and assign the onreadystatechange function as follows:

```
var r = new XMLHttpRequest();  
r.onreadystatechange = function()  
{
```

This function can be called many different times during the loading process, so we check for a specific value. In this case, 4 means that the content has been loaded:

```
if (r.readyState == 4)  
{
```

Of course, just because we got data doesn't mean that it is useable data. We need to verify the status of the load, and here we get into a little bit of murky territory. iOS defines success with a zero value, while Android defines it with a 200:

```
if ( r.status==200 || r.status == 0)  
{
```

If we've successfully loaded the data, we'll call the completion function with a success notification, and the data, as follows:

```
if (completion)  
{  
    completion ( PKUTIL.COMPLETION_SUCCESS,  
                r.responseText );  
}
```

But if we've failed to load the data, we call the completion function with a failure notification and the status value of the load, as follows:

```
else  
{  
    if (completion)  
    {  
        completion ( PKUTIL.COMPLETION_FAILURE,  
                    r.status );  
    }
```

}{ }{ }

Keep in mind that we're still just setting up the XMLHttpRequest object and that we've not actually triggered the load yet.

The next step is to specify the path to the file, and here we run into a problem on WP7 versus Android and iOS. On both Android and iOS we can load files relative to the `index.html` file, but on WP7, we have to load them relative to the `/app/www` directory. Subtle to track down, but critically important. Even though we aren't supporting WP7 in this book, the framework does, and so it needs to handle cases like this using the following code snippet:

```
if (device.platform=="WinCE")
{
    r.open ('GET', "/app/www/" + theFileName, aSync);
}
else
{
    r.open ('GET', theFileName, aSync);
}
```

Now that we've set the filename, we fire off the load:

```
r.send ( null );
```



Should you ever decide to support WP7, it is critical that even though the framework supports passing `false` for `aSync`, which should result in a synchronous load, you shouldn't actually ever do so. WP7's browser does funny things when it can't load data asynchronously. For one thing, it loads it asynchronously anyway (not your intended behavior), and for another thing, it has a tendency to think the file simply doesn't exist. So instead of loading scripts, you'll get errors in the console indicating that a 404 error occurred. And you'll scratch your head (I did!) wondering why in the world that could be when the file is right there. Then you'll remember this long note, change the value back to `true`, and things will suddenly start working. (You seriously do not want to know the hours it took me to debug on WP7 to finally figure this out. I want those hours back!)

Implementing the end view

We'll be creating the file name `endView.html` in the `www/views` directory. When we're done, we'll end up with this view for iOS:



The view for Android will be as follows:



Getting on with it

As with our previous views, the first step is to define the HTML representation:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div id="endView_title"></div>
    <button class="barButton backButton" id="endView_backButton"
style="left:10px" ></button>
  </div>
  <div class="content avoidNavigationBar avoidToolBar" id="endView_
gameArea">
    <div id="endView_resultsArea"></div>
  </div>
  <div class="toolBar">
    <button class="barButton" id="endView_tryAgainButton"
style="right:10px" ></button>
  </div>
</div>
```

I've highlighted two areas in this code: `resultsArea` where we'll tell the player how they scored, and the button in the toolbar, which this time is a Try Again? button. It acts just like a back button, though.

Next, we need localized content. In this case, it's both localized content and a template, as shown in the following code snippet:

```
<div id="endView_template_en" class="hidden">
  <h2>Congratulations!</h2>
  <p>You finished that round with a score of %SCORE%!</p>
  <p>Dated: %DATE%</p>
</div>

<div id="endView_template_es" class="hidden">
  <h2>¡Felicitaciones!</h2>
  <p>¡Se terminó la ronda con una puntuación de %SCORE%!</p>
  <p>Fecha: %DATE%</p>
</div>
```

Again, these `div` elements are hidden so that the player can't see them, but we'll take their content, replace `%SCORE%` and `%DATE%`, and then show the resulting content to the player.

Let's look at our script:

```
<script>

    var endView = $ge("endView") || {}; // properly namespace

    endView.score = 0;
```

First, we set the score to zero, mainly for initialization purposes. We'll provide a utility function next that sets the score to any value. As you should remember, this is called when the game is ending in the game view. This initialization is shown in the following code snippet:

```
endView.setScore = function( theScore )
{
    endView.score = theScore;
}
```

As has been typical of all our previous views, we have an `initializeView()` method. What's a little different is that it doesn't localize the content area; that's because we don't know the score at this point. The `initializeView()` function is called well in advance of the game even starting, let alone being finished. This function is given as follows:

```
endView.initializeView = function ()
{

    endView.viewTitle = $ge("endView_title");
    endView.viewTitle.innerHTML = __T("RESULTS");

    endView.backButton = $ge("endView_backButton");
    endView.backButton.innerHTML = __T("BACK");
    PKUI.CORE.addTouchListener(endView.backButton, "touchend",
        function () { PKUI.CORE.popView(); });

    endView.nextButton = $ge("endView_tryAgainButton");
    endView.nextButton.innerHTML = __T("TRY AGAIN?");

    PKUI.CORE.addTouchListener(endView.nextButton, "touchend",
        function () { PKUI.CORE.popView(); });

    endView.questionArea = $ge("endView_resultsArea");
}
```

Notice that both buttons, the back button and the try again button do the same thing, they pop the view. This works because when we pop off the view, gameView will get the viewWillAppears notification, which resets the game.

This view also needs such a notification to set up the content area, since we'll know the score by the time endView appears on screen:

```
endView.viewWillAppear = function ()
{
    var theTemplate = $geLocale("endView_template").innerHTML;

    theTemplate = theTemplate.replace ( "%SCORE%",
        _N(endView.score, "2") );

    theTemplate = theTemplate.replace ( "%DATE%",
        _D(new Date(), "D") );

    endView.questionArea.innerHTML = theTemplate;
}
```

We get the properly localized template and replace %SCORE% with the actual score, and %DATE% with the current date (the D here means long format date). We then show it to the end user. All of this happens just prior to the view animating on screen.

We need to have code that will handle the back button should it be pressed, which is a pop back to the gameView:

```
endView.backButtonPressed = function ()
{
    PKUI.CORE.popView();
}
```

Astonishingly, that's it. There's really no new ground to cover here, no new methods in the framework, no new utility methods, no new localization concepts. The only thing that looks new is the `__D()` function, which, as you can probably guess, is what localizes dates. In fact, there are two more functions that are similar: `__C()`, which localizes currency and `__PCT()`, which localizes percentages. We'll be dealing with these in later apps.

What did we do?

We created the End view. We properly localized a content template, and localized both numbers and dates.

Putting it all together

We've almost got a fully functional app on our hands, but we're missing a couple of critical components: the part that loads it all and starts it off. For this, we'll be creating an `app.js` file and two HTML files under the `www` directory.

Getting on with it

The `index.html` and `index_android.html` files are what kicks everything off by loading the necessary scripts and calling `app.js`. These are typically pretty standard for each app, so they aren't going to change much throughout the rest of the book.

First, `index.html`, which is intended for iOS, is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Chapter 1 App: Quiz Time</title>
    <meta name="apple-mobile-web-app-capable" content="yes" />
    <meta name="viewport" content="width=device-width,
maximum-scale=1.0" />
    <meta name="format-detection" content="telephone=no" />
    <link rel="stylesheet" href=".framework/base.css"
type="text/css" />
    <link rel="stylesheet" href=".style/style.css"
type="text/css" />
    <script type="application/javascript" charset="utf-8"
src=".cordova/cordova-2.2.0-ios.js"></script>
    <script type="application/javascript" charset="utf-8"
src=".framework/utility.js"></script>
    <script type="application/javascript" charset="utf-8"
src=".app.js"></script>
  </head>
  <body>
```

```
<div class="container" id="rootContainer">
</div>
<div id="preventClicks"></div>
</body>
</html>
```

Next, `index_android.html`, which is for Android is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Chapter 1 App: Quiz Time</title>
    <meta name="apple-mobile-web-app-capable" content="yes" />
    <meta name="viewport" content="width=device-width,
maximum-scale=1.0, target-densityDpi=160" />
    <meta name="format-detection" content="telephone=no" />
    <link rel="stylesheet" href=".framework/base.css"
type="text/css" />
    <link rel="stylesheet" href=".style/style.css"
type="text/css" />
    <script type="application/javascript" charset="utf-8"
src=".cordova/cordova-2.2.0-android.js"></script>
    <script type="application/javascript" charset="utf-8"
src=".framework/utility.js"></script>
    <script type="application/javascript" charset="utf-8"
src=".app.js"></script>
  </head>
  <body>
    <div class="container" id="rootContainer">
    </div>
    <div id="preventClicks"></div>
  </body>
</html>
```

The `app.js` file is what actually starts our app. It is also what initializes our localization, sets our current locale, loads various libraries (such as `ui-core.js`), and finally, starts our app. Let's look at the code now:

```
var APP = APP || {};
```

As usual, we set up our namespace, this time as `APP`. Next, we'll attach an event listener to the `deviceready` event; this fires whenever Cordova has finished loading its libraries. We must wait for this event before we can do much of anything, especially anything that relies on Cordova. If we don't, we'll get errors. We'll set our namespace as follows:

```
document.addEventListener("deviceready", onDeviceReady, false);

function onDeviceReady()
```

Let's Get Local! —————

```
{  
    APP.start();  
}
```

All that the preceding function does is call the `APP.start()` function, which is defined as follows:

```
APP.start = function ()  
{  
    PKUTIL.include ( [ "./framework/ui-core.js",  
                      "./framework/device.js",  
                      "./framework/localization.js" ],  
                     function () { APP.initLocalization(); } );  
}
```

You've already seen `PKUTIL.include`, so it isn't anything new to you, but here we're loading three libraries, and including a completion function to call `APP.initLocalization`. Because the include is asynchronous, we cannot continue writing the code after this call that relies on those libraries, or there's a good chance the library wouldn't be loaded in time. Therefore, we call the `initLocalization` function when all three libraries are fully loaded.

The next function, `initLocalization`, initializes our `jQuery/Globalize` by loading its libraries and when it is complete, we load any locales we might need. When those locales are finished loading, then we call `APP.init` and this is where the real work begins. The `APP.init` function is given as follows:

```
APP.initLocalization = function ()  
{  
    PKLOC.initializeGlobalization()  
    function ()  
    {  
        PKLOC.loadLocales ( [ "en-US", "en-AU", "en-GB",  
                             "es-ES", "es-MX", "es-US", "es" ],  
                             function ()  
                             {  
                                 APP.init();  
                             } );  
    };  
};  
}
```

The APP.init() function defines our app's basic translation matrix (you may see translations you've seen before; that's because they originated from here), and we also proceed to load the three views we have created into the document:

```
APP.init = function ()  
{
```

First, we fake our locale by setting it to Spanish language and the country of Spain. If you want the app to determine the locale by querying the system, comment the line out.

```
    PKLOC.currentUserLocale = "es-ES";
```

Next, we have our basic translation matrix, application titles, the translations for correct and incorrect, start, back and skip, and more:

```
    PKLOC.addTranslation ("en", "APP_TITLE", "Quiz Time");  
    PKLOC.addTranslation ("en", "APP_TITLE_IMAGE", "AppTitle-  
    enus.png");  
    PKLOC.addTranslation ("en", "CORRECT", "Correct!");  
    PKLOC.addTranslation ("en", "INCORRECT", "Incorrect.");  
    PKLOC.addTranslation ("en", "START", "Start");  
    PKLOC.addTranslation ("en", "BACK", "Back");  
    PKLOC.addTranslation ("en", "SKIP", "Skip");  
    PKLOC.addTranslation ("en", "QUESTION_%1", "Question %1");  
    PKLOC.addTranslation ("en", "SCORE_%1", "Score: %1");  
    PKLOC.addTranslation ("en", "RESULTS", "Results");  
    PKLOC.addTranslation ("en", "TRY AGAIN?", "Try Again?");  
  
    PKLOC.addTranslation ("es", "APP_TITLE", "Examen  
    Tiempo");  
    PKLOC.addTranslation ("es", "APP_TITLE_IMAGE", "AppTitle-  
    eses.png");  
    PKLOC.addTranslation ("es", "CORRECT", "¡Correcto!");  
    PKLOC.addTranslation ("es", "INCORRECT", "Incorrecto.");  
    PKLOC.addTranslation ("es", "START", "Comenzar");  
    PKLOC.addTranslation ("es", "BACK", "Volver");  
    PKLOC.addTranslation ("es", "SKIP", "Omitir");  
    PKLOC.addTranslation ("es", "QUESTION_%1", "Pregunta %1");  
    PKLOC.addTranslation ("es", "SCORE_%1", "Puntuación: %1");  
    PKLOC.addTranslation ("es", "RESULTS", "Resultados");  
    PKLOC.addTranslation ("es", "TRY AGAIN?", "¿Intentar du  
    nuevo?");
```

Let's Get Local! —

Next, we call a function in `PKUI.CORE` called `initializeApplication`. All this function does is attach a special event handler that tracks the orientation of the device. But by doing so, it also attaches the device, the form factor, and the orientation to the `BODY` element, which is what permits us to target various platforms with CSS. This function is given as follows:

```
PKUI.CORE.initializeApp ( );
```

Next, we load a view, `gameView` in this case (order doesn't really matter here):

```
PKUTIL.loadHTML ( "./views/gameView.html",
    { id : "gameView",
        className: "container",
        attachTo: $ge("rootContainer"),
        aSync: true
    },
    function (success)
    {
        if (success)
        {
            gameView.initializeView();
        }
    });
});
```

We call `PKUTIL.loadHTML` to accomplish this, and if you're thinking it would be a lot like `PKUTIL.include`, you'd be right. We'll look at the definition a little later, but it should suffice to say, we're loading the content inside `gameView.html`, wrapping it with another `div` with an `id` value of `gameView` and a class of `container`, attaching it to the `rootContainer`, and indicating that it can be loaded asynchronously.

Once it finishes loading, we'll call `initializeView()` on it.

We load the end view the same way as follows:

```
PKUTIL.loadHTML ( "./views/endView.html",
    { id : "endView",
        className: "container",
        attachTo: $ge("rootContainer"),
        aSync: true
    },
    function (success)
    {
        if (success)
        {
            endView.initializeView();
        }
    });
});
```

We load the start view almost exactly the same way as all the others. I'll highlight the difference in the following code block:

```
PKUTIL.loadHTML ( "./views/startView.html",
    {
        id : "startView",
        className: "container",
        attachTo: $ge("rootContainer"),
        aSync: true
    },
    function (success)
    {
        if (success)
        {
            startView.initializeView();
            PKUI.CORE.showView (startView);
        }
    });
}
```

The only thing we do differently is to show `startView` after we initialize it. At this point the game is fully loaded and running, and waiting for the player to tap **Start**.

What did we do?

We tied everything together by creating the `app.js` file. We learned how to initialize the jQuery/Globalize library, how to fake a locale, and how to set up our translation matrix. We learned how to load views, and how to show the first one.

What else do I need to know?

Let's look at `PKUTIL.loadHTML` a little closer:

```
PKUTIL.loadHTML = function( theFileName, options, completion )
{
    var aSync = options["aSync"];
```

The first thing we do is pull out the `aSync` option, we need it to call `PKUTIL.load`. Again, the warning about WP7 and loading synchronously still applies. It is best to assume you'll always be using `true` unless you can rule WP7 out of your supported platforms. We use the `aSync` option as follows:

```
PKUTIL.load ( theFileName, aSync, function ( success, data )
{
    if (success)
    {
```

Let's Get Local! —————

At this point, we've successfully loaded the HTML file, as seen in the following code snippet, and now we have to figure out what to do with it:

```
var theId = options["id"];
var theClass = options["className"];
var attachTo = options["attachTo"];
```

First, we extract out the other parameters we need, namely, `id`, `className`, and `attachTo`:

```
var theElement = document.createElement ("DIV");
theElement.setAttribute ("id", theId);
theElement.setAttribute ("class", theClass);
theElement.style.display = "none";
theElement.innerHTML = data;
```

Next, we create a `div` element, and give it `id` and `class`. We also load the data into the element as shown in the following code block:

```
if (attachTo)
{
    attachTo.appendChild (theElement);
}
else
{
    document.body.appendChild (theElement);
}
```

If possible, we'll attach to the element specified in `attachTo`, but if it isn't defined, we'll attach to the `BODY` element. It is at this point that we become a real DOM element in the display hierarchy.

Unfortunately this isn't all. Remember that our HTML files have `SCRIPT` tags in them. For whatever reason, these scripts don't execute automatically when loaded in this fashion. We have to create `SCRIPT` tags for them again, as shown in the following code snippet:

```
var theScriptTags = theElement.getElementsByTagName
("script");
```

First, we get all the `SCRIPT` tags in our newly created element. Then we'll iterate through each one, like this:

```
for (var i=0;i<theScriptTags.length;i++)
{
    try
    {
```

```
// inspired by
http://bytes.com/topic/javascript/answers/513633-
innerhtml-script-tag
var theScriptElement =
    document.createElement("script");
theScriptElement.type = "text/javascript";
theScriptElement.charset = "utf-8";
if (theScriptTags[i].src)
{
    theScriptElement.src = theScriptTags[i].src;
}
else
{
    theScriptElement.text = theScriptTags[i].text;
}
document.body.appendChild (theScriptElement);
```

If this code looks somewhat familiar, it's because `PKUTIL.include` has a variant of it. The important distinction is that it was only concerned about the data of the script; here we have to worry if the script is defined as an external script. That is why we check to see if the `SRC` attribute is defined.

We also have surrounded this in a `try/catch` block, just in case the scripts have errors in them:

```

}
catch ( err )
{
    console.log ( "When loading " + theFileName +
                  ", error: " + err );
}
}
```

We've finished loading the HTML and the scripts, so we call the `completion` function:

```
if (completion)
{
    completion (PKUTIL.COMPLETION_SUCCESS);
}
```

If, for whatever reason, we couldn't load the view, we generate a log message and call the completion function with a failure notification as follows:

```
else
{
    console.log ("WARNING: Failed to load " + theFileName );
    if (completion)
    {
        completion (PKUTIL.COMPLETION_FAILURE);
    }
}
);
}
```

Next, we should review the new localization functions we encountered. The first was `PKLOC.initializeGlobalization()`:

```
PKLOC.initializeGlobalization = function ( completion )
{
    PKUTIL.include ( [ "./framework/globalize.js" ],
                     completion );
}
```

As you can see, all it does is load the `jQuery/Globalize` framework, and then call its completion handler.

The next function is `PKLOC.loadLocales`. It is designed to make it easy to load `jQuery/Globalize` culture files. These files live in the `www/framework/cultures` directory, and you can have and load as many as you like. Just remember that the more you have, the larger your app will be, and the longer it will take to start:

```
PKLOC.loadLocales = function ( theLocales, completion )
{
    for (var i=0; i<theLocales.length; i++)
    {
        theLocales[i] =
            "./framework/cultures/globalize.culture." +
            theLocales[i] + ".js";
    }
    PKUTIL.include ( theLocales, completion );
}
```

Here we take advantage of the fact that `PKUTIL.include` takes an array of script files. The incoming locales (in no real particular order; jQuery/Globalize culture files only depend upon the jQuery/Globalize library being loaded) are already in an array, and so we alter the array to include the full path and name of the culture file. When we're done, we include them, and the `completion` function will be called when they are all loaded.

Game Over..... Wrapping it up

Wow, we've been through a lot together in this first project. We've learned a lot too, including:

- ▶ How to properly localize text
- ▶ How to properly localize numbers
- ▶ How to properly localize dates
- ▶ How to properly localize images
- ▶ How to properly localize HTML
- ▶ How to implement simple HTML templates
- ▶ How to create a new view
- ▶ How to display the view
- ▶ How to push a new view onscreen, and pop a view offscreen
- ▶ How to handle the Android/WP7 back button
- ▶ How to include files within our JavaScript
- ▶ How to determine the user's locale
- ▶ How to initialize jQuery/Globalize and load the locales we might need

There are some resources that you might find interesting. You might want to look through the YASMF documentation to learn more about the framework we're using. Some of these resources are mentioned as follows:

- ▶ Adobe Photoshop at <http://www.adobe.com/PhotoshopFamily>
- ▶ GIMP at <http://www.gimp.org>
- ▶ PhoneGap downloads at <http://www.phonegap.com/download>
- ▶ PhoneGap documentation at <http://docs.phonegap.com>
- ▶ YASMF GitHub at <https://github.com/photokandyStudios/YASMF/>
- ▶ YASMF documentation at <https://github.com/photokandyStudios/YASMF/wiki/>

Let's Get Local!

- ▶ Xcode at <https://developer.apple.com/xcode>
- ▶ Eclipse Classic 4.2.1 at <http://www.eclipse.org/downloads/packages/eclipse-classic-421/junosr1>
- ▶ Android SDK download at <http://developer.android.com/sdk/index.html>

Can you take the HEAT? The Hotshot Challenge

There are quite a few ways that this project could be enhanced. Why don't you try your hand at one or more of them?

- ▶ The game currently supports English and Spanish. Try adding another language.
- ▶ If you play the game for any length of time, you'll find that the same question is often asked again within the same set. Make it so that a question can only be asked once per set.
- ▶ Add categories to the questions, and then allow the user to select which category they'd like to play through.
- ▶ Add difficulty levels to the questions. These could affect the score awarded as well. Allow the user to select the difficulty level they want to play at.
- ▶ Come up with an alternative look and feel for the game and implement it. Perhaps even allow the user to decide which look and feel they want to use.

Project 2

Let's Get Social!

Social networking has changed the way we share information in our world. Where it used to be an e-mail to a friend (or even a letter!), now it's a Twitter or a Facebook post, often for the world to see. What's even more amazing is how relatively young the various social networks are and how quickly they have changed the way we communicate and consume information. Because of this transformation, our apps need to support sharing to social networks, lest our app appears dated.

What do we build?

In this project, we will build an app that illustrates both sides of the social network equation. The first is that of consuming the information from various sources; we'll be using Twitter streams for this. The second is that of sharing information; we'll be using each platform's native sharing capabilities for this, except for iOS where we'll be using a project called ShareKit to implement sharing. (Note that iOS 5 supports Twitter sharing and iOS 6 expands that to Facebook. Sooner or later a plugin is bound to appear that supports these functions, but ShareKit provides more targets.)

What does it do?

Our app, called Socializer, will display the Twitter streams from five pre-set Twitter accounts. The user can then read these streams, and should they find an interesting tweet, they can tap on it to do more with it. For example, they may wish to view a link embedded in the tweet. More importantly, the end user may wish to share the information using their own social network of choice, and the app will offer a Share button to do just that.

To accomplish this, we'll be working with Twitter's JSON API, a natural fit for an app already written largely in JavaScript. The only downside is that Twitter has a pretty low cap for rate-limiting API requests, so we'll also have to build some basic support for when this occurs. (To be honest, this is far more likely to occur to us as a developer than the user because we often reload the app to test a new feature, which incurs new API requests far faster than an end user would typically incur them.)

We'll also introduce the concept of PhoneGap plugins, as sharing functionality is not present in the typical PhoneGap install by default. A plugin is essentially a bridge between some amount of native code (such as Java, Objective C, or C#) and our JavaScript code.

For this project, we'll be using two plugins for each platform we support. One, `ChildBrowser`, is supported across most platforms, which makes it far easier to write code that uses it. The second is based upon the platform's sharing capabilities and what plugins are available for the platform and PhoneGap. Since this varies, we'll have to deal with separate code paths for each platform, but the idea will be the same—to share content.

Why is it great?

This project is a great introduction to handling APIs using JSON, including Twitter's API. While we're using a very small subset of Twitter's API, the lessons learned in this project can be expanded to deal with the rest of the APIs. Furthermore, JSON APIs are frequently used by many web platforms, and learning how to deal with Twitter's API is a great way to learn how to deal with any JSON API.

We'll also be dealing with how to share content. While Android provides a shared mechanism between all apps that provide a nice list of sharing apps, iOS does not. So we'll also have to write code that is platform specific to handle the differences in how each platform (and the corresponding plugin) supports sharing.

We'll also be working with PhoneGap plugins, something many apps will eventually require in one way or another. For example, our app should be able to handle links to external websites; the best way to do this is to have the `ChildBrowser` plugin handle it. This lets the user stay inside our app and easily return to our app when they are done. Without it, we'd be taking the user out of the app and into the default browser.

How are we going to do it?

To do this, we're going to break down the creation of our app into several different parts, as follows:

- ▶ Designing the app – UI/interaction design
- ▶ Designing the app – the data model

- ▶ Implementing the data model
- ▶ Configuring the plugins
- ▶ Implementing the social view
- ▶ Implementing the tweet view

Just like in the previous project, we'll focus on the design of the app before we handle the implementation.

What do I need to get started?

You'll need to go ahead and create your project, just like we did in the last project. You can, to some degree, copy the previous project and replace the necessary files and settings as well. There is one additional iOS setting that should be modified, but it's really a matter of taste (whether or not you like your status bars black or grey):

1. Open Socializer-info.plist.
2. Add Status Bar style and set it to UIStatusBarStyleOpaqueBlack.

We'll also be using the same directory structure as the previous project, with two exceptions: we'll be adding a www/childbrowser directory and a www/plugins directory. Underneath the www/plugins directory, we'll have a directory for each platform: namely, /www/plugins/Android and /www/plugins/iOS. We'll fill these directories later, but go ahead and create them now.

We'll be using the same framework, so be sure to copy the framework files. We won't worry about localizing the content, but even so, we'll use all the localization functions so that it would be easy to do so. We'll also use a script to deal with scrolling in www/framework called scroller.js. You'll need to add this to your index files in order to use it correctly, as shown in the following code snippet:

```
<script type="application/javascript" charset="utf-8" src=".//framework/scroller.js"></script>
```

You'll also need to download the PhoneGap plugins repository available at <http://www.github.com/phonegap/phonegap-plugins>. This will ensure you have all the necessary plugins we'll need as well as any plugins you might be interested in working with on your own.

Finally, for iOS, we'll need to get the ShareKit 2.0 plugin available at <https://github.com/ShareKit/ShareKit>. Due to the way it is distributed, you'll need to install Git and make sure you enable Git for the project. (Alternatively, you can use the /Submodules directory in the download files).

Designing the app – UI/interaction design

Our first task is to design our user interface and the interaction between the various widgets and views. Like in the previous task, we will have three views: the start view, the social view, and the tweet view.

Getting on with it

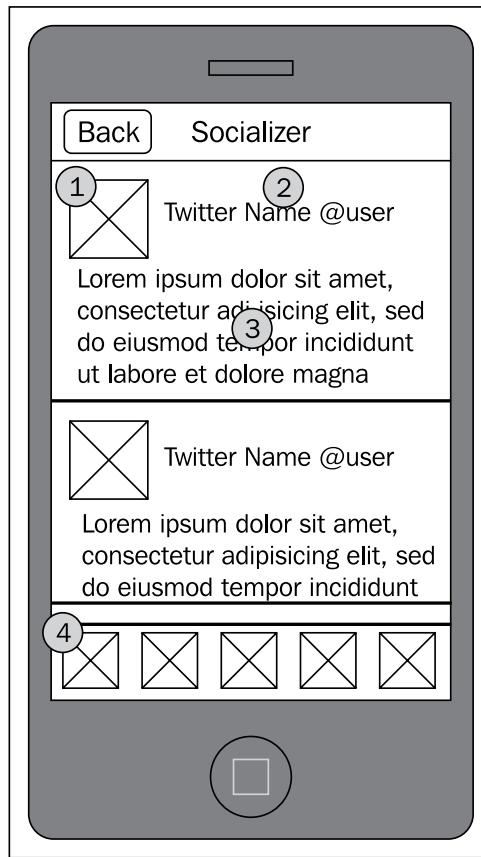
We'll begin with the start view. As in the last project, this will be a very simple view and is entirely optional in this app. All we'll be doing is explaining the app and providing a way to move to the main view.

With that in mind, following is our screenshot for the start view:



In this screenshot, we have a **Start** button (1) that will push the social view on to the view stack. We also have some explanatory text (2).

Our next view is the social view, shown in the following screenshot:



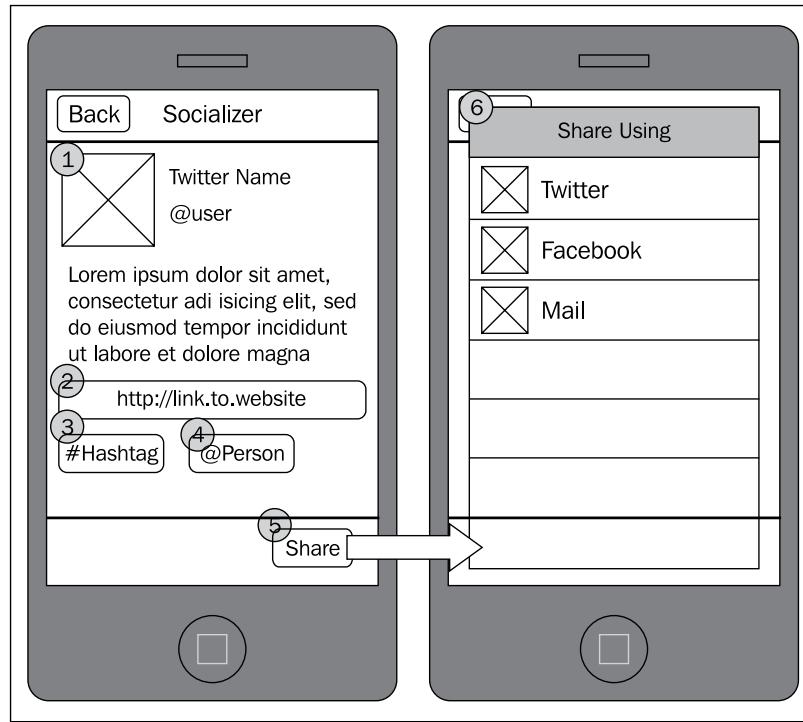
The social view is essentially a list of tweets, one after the other. We'll display several tweets at a time, and, as such, we'll have to deal with scrolling at some point. While you can use various libraries to accomplish this, we'll be using our own minimalist scrolling library.

Each tweet will consist of a profile image (1), the screen name and real name (if available) (2), and the text of the tweet (3). When the user taps a tweet, we'll transition to the tweet view.

At the bottom of the view (4), we have a series of profile images for five different Twitter accounts. The images will be retrieved from Twitter itself; we won't be storing the images ourselves. When an image is tapped, we'll load the respective Twitter stream.

Let's Get Social! —————

Our tweet view looks like the following screenshot:



First, note that our tweet view repeats the tweet (1) that the user tapped on in the stream view. The same information is repeated, but we also list out the various web links (2) that the tweet might have, any hashtags (3), and any user mentions (4). Items (2) to (4) are intended to be tappable: that is, if a user taps on (2), they should be taken to the particular website. If they tap on (3), they should be taken back to the social view with a stream of tweets referencing the hashtag. The same should happen if they tap on (4), except that it would be that particular user's stream.

We also have a **Back** button in our navigation bar to take the user back to the previous view, and a **Share** button (5) in our toolbar. This button, when tapped, should display a list (6) of various social network services. What this list will look like will depend upon the platform the app is on, and what social networks are installed on the device.

Now that we've created our mockup, we need to define some of the resources we'll need. Let's open up our editing program and get busy designing our app.



The preceding screenshot is a pretty good representation of how our final product will look. A lot of it can be accomplished with CSS . The background of the Twitter stream and the navigation bar are the only two components that will be difficult, so we should save those out to our `www/images` directory as `Background.png` and `NavigationBar.png` respectively. Notice that, both have a texture, so make sure to save them in a way that they will tile without visible seams.

What did we do?

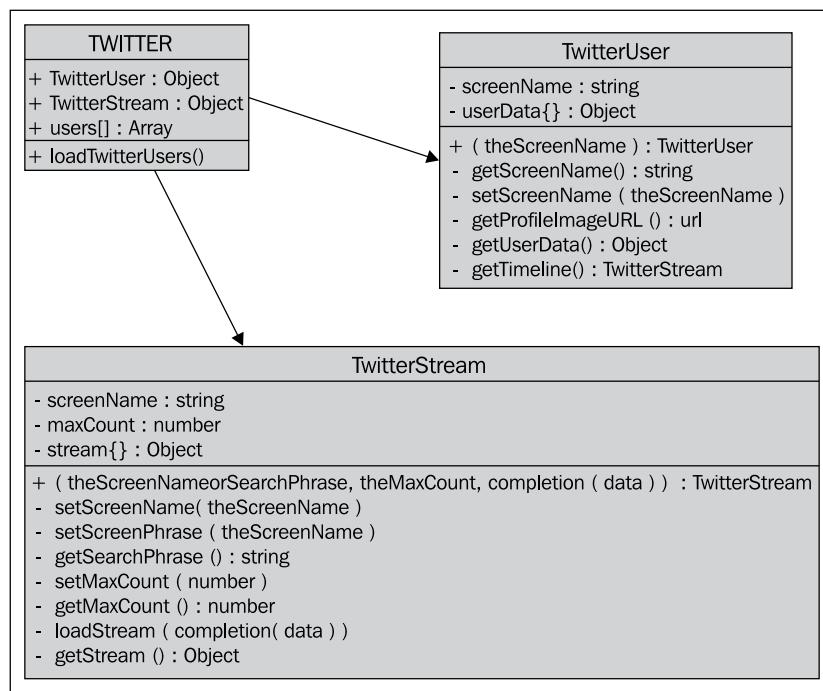
For this task, we've defined how our UI should look, and the various interactions between widgets and views. We also generated a mockup of the app in our graphics editor and created some image resources for our later use.

Designing the app – the data model

In this task, we will design our data model for handling Twitter users and streams. Our model will, to some extent, rely on Twitter's model as well. The results that it returns from its API we will use unmodified. The rest of the model we will define in this task.

Getting on with it

Let's take a look at our data model:



We'll be using **TWITTER** as the namespace and within it, we'll have two objects we'll be using a lot: **TwitterUser** and **TwitterStream**. The idea behind **TwitterUser** is to be an instance of a particular user, which we'll represent by an image on the toolbar in the streams view. The **TwitterStream** object will be a representation of a single stream.

Let's examine **TwitterUser** more closely. The object has two properties: **screenName** and **userData**. The **screenName** property holds the user's Twitter username. The **userData** property will hold the response from Twitter's API. It will have lots of different information about the user, including their profile image URL, their real name, and more.

The constructor will return an initialized `TwitterUser` based upon the supplied screen name. Internally, the constructor just calls the `setScreenName()` method, which will request the user data from Twitter. The `getScreenName()` method simply returns the screen name. The `getProfileImageUrl()` method will return the URL to the user's profile image. The `getUserData()` method will return the data that Twitter returned, and the `getTimeline()` method will create a `TwitterStream` object for the particular user.

The `TwitterStream` object operates on a similar idea: it will house the data returned by Twitter. The `TwitterStream` object also provides us the ability to get a stream for a particular user as well as the ability to return a stream for any search (such as a hashtag).

When constructed, we pass three options: the screen name or the search phrase, the maximum number of tweets to return (up to 200), and a function to call when the stream is fully loaded. It will call the `loadStream()` method to do the actual loading.

We have some methods related to the properties in the object such as `setScreenName()`, `setSearchPhrase()`, `getSearchPhrase()`, `setMaxCount()`, `getMaxCount()`, and `getStream()`.

The `setScreenName()` method does the same thing as setting a `searchPhrase()` method except that it adds an @ character to the name. The `loadStream()` method then can decide which API to call when loading a stream, either by calling the API to return the user's stream, or by calling the search API.

What did we do?

We created and defined our data model for our app. We've defined two objects: namely, `TwitterUser` and `TwitterStream`, and saw how they interact.

Implementing the data model

We'll be creating two files: namely, `twitter.js` and `twitterUsers.js`. Place these in the `www/models` directory.

Getting on with it

Let's start with the `twitter.js` file:

```
var TWITTER = TWITTER || {};
```

As always, we define our namespace, in this case, TWITTER, as seen in the following code snippet:

```
TWITTER._baseURL = "http://api.twitter.com/1/";
TWITTER._searchBase = "http://search.twitter.com/";
```

We define two variables global to the TWITTER namespace: namely, `_baseURL` and `_searchBase`. These two URLs point at Twitter's JSON API; the first is for API requests such as user lookups, user streams, and such, while the latter is only for searching. We define them here for two reasons: to make the URLs a little less nasty in the following code, and if Twitter should ever decide to have a different version of the API (and you want to change it), you can do so here.

Next, we define our first object, `TwitterUser`:

```
TWITTER.TwitterUser = function ( theScreenName, completion )
{
    var self = this;

    self._screenName = "";
    self._userData = {};
}
```

We've defined our two properties: `_screenName` and `_userData`. Unlike in the last project, we're using underscores in front to indicate that these are internal (private) variables that no outside object should access. Instead, an outside object should use the `get`/`set` methods we define next:

```
self.getScreenName = function ()
{
    return self._screenName;
}
```

This one's simple enough, it just returns the private member when asked. But the next one's more complicated:

```
self.setScreenName = function ( theScreenName, completion
)
{
    self._screenName = theScreenName;
```

Like with a normal `set` method, we've assigned `theScreenName` to `_screenName`. But when this happens, we want to load in the user information from Twitter. This is why it is important to have `get`/`set` methods in front of private methods. You might just need to do something important when the value changes or is read.

```
var getUserURL = TWITTER._baseURL +
    "users/lookup.json?screen_name=" +
    encodeURIComponent(theScreenName);
```

Here we've defined our URL that we're going to use to ask Twitter to look up the user in question. For more information about how this particular URL works, see the Twitter documentation at <https://dev.twitter.com/docs/api/1/get/users/lookup>. You can see a full example of what is returned at the bottom of the page.

Now that we have our URL, we're going to use another utility function defined for us in `PKUTIL` (`www/framework/utility.js`) called `loadJSON()`. It uses AJAX to send a request to the preceding URL, and Twitter then sends a response back, in the form of JSON. When it is finished, the function will call the `completion` function we're passing as the second parameter after `getUserURL`. This method can check if the request succeeded or not, and set any private members that are necessary. We'll also call the `completion` function passed to the `setScreenName()` method. These actions are defined in the following code snippet:

```
PKUTIL.loadJSON ( getUserURL, function (
    success, data )
{
    if (success)
    {
        self._userData = data;
    }
    else
    {
        self._userData = { "error": "Twitter error; rate
            limited?" };
    }
}
if (completion)
{
    completion ();
}
```

If `success` is true, then the JSON has been properly returned and parsed into the `data` parameter. We just assign it to the private `_userData` member as seen in the preceding code block.

```
}
```

Finally, regardless of success or failure, we call the `completion` function passed to us. This is important so that we know when we can safely access the `_userData` member (via `getUserData` a little lower down).

```
        }
    );
}

self.getProfileImageURL = function ()
{
    if (self._userData[0])
    {
        return self._userData[0].profile_image_url;
    }
    return "";
}
```

In the preceding code snippet, the method `getProfileImageURL()` is a convenience function that returns the user's profile image URL. This is a link to the avatar being used for Twitter. First we check to see if `_userData[0]` exists, and if so, return `profile_image_url`, a value defined by the Twitter API. If it doesn't, we'll just return an empty string.

```
self.getUserData = function ()
{
    return self._userData;
}
```

Next, `getUserData()` is used to return the `_userData` member. If it has been properly loaded, it will have a lot of values in it, all determined by Twitter. If it has failed to load, it'll have an `error` property in it, and if it hasn't been loaded at all, it'll be empty.

```
self.getTimeline = function ( theMaxCount, completion )
{
    return new TWITTER.TwitterStream ( "@" +
        self._theScreenName, completion, theMaxCount || 25
    );
}
```

`getTimeline()` is also a convenience function used to get the timeline for the Twitter user. `theMaxCount` is the maximum number of tweets to return (up to 200), and `completion` is a function to call when it's all done. We do this by creating a new `TwitterStream` object (defined in the following code snippet) with the Twitter screen name prepended by an @ character.

If `theMaxCount` isn't specified, we use a little `||` trick to indicate the default value, 25 tweets.

```
    self.setScreenName ( theScreenName, completion );
}
```

The last thing we do is actually call the `setScreenName()` method with `theScreenName` and `completion` functions passed in to the constructor. If you remember your JavaScript, this whole function, while we can think of it as defining an object, is also the constructor of that object. In this case, as soon as you create the `TwitterUser` object, we'll fire off a request to Twitter to load in the user's data and set it to `_userData`.

Our next object is the `TwitterStream` object defined as follows:

```
TWITTER.TwitterStream = function (
  theScreenNameOrSearchPhrase, completion, theMaxCount )
{
  var self = this;

  self._searchPhrase = "";
  self._stream = {};
  self._theMaxCount = 25;
```

Here we've defined three properties, namely, `_searchPhrase`, `_stream`, and `_theMaxCount`. The `_searchPhrase` property can be either the screen name of a user or a literal search term, such as a hashtag. The `_stream` property is the actual collection of tweets obtained from Twitter, and the `_theMaxCount` property is the maximum number of tweets to ask for. (Keep in mind that Twitter is free to return less than this amount.)

You may ask why we're storing either a search phrase or a screen name. The reason is that we're attempting to promote some code re-use. It's only logical to assume that a Twitter stream is a Twitter stream, regardless of how it was found, either by asking for a particular user's stream or by searching for a word. Nice assumption, right?

Yeah, totally wrong too. The streams are close, close enough that we can work around the differences, but still, not the same. So even though we're treating them here as one-and-the-same, they really aren't, at least until Twitter decides to change their search API to better match their non-search API.

```
self.setMaxCount = function ( theMaxCount )
{
  self._theMaxCount = theMaxCount;
}
```

```
self.getMaxCount = function ()
{
    return self._theMaxCount;
}
```

Here we have the `get`/`set` methods for `_theMaxCount`. All we do is set and retrieve the value. One thing to note is that this should be called before we actually load a stream; this value is part of the ultimate URL we sent to Twitter.

```
self.setScreenName = function ( theScreenName )
{
    self._searchPhrase = "@" + theScreenName;
}

self.setSearchPhrase = function ( theSearchPhrase )
{
    self._searchPhrase = theSearchPhrase;
}
self.getSearchPhrase = function ()
{
    return self._searchPhrase;
}
```

Notice that we have two `set` methods that act on `_searchPhrase` while we only have one `get` method. What we're doing here is permitting someone to call `setScreenName()` without the @ character. The `_searchPhrase` property will then be set to @ prepended to the screen name. The next `set` method (`setSearchPhrase()`) is intended to be used when setting real search terms (such as a hashtag).

Internally, we'll use that @ at the front to mean something special, but you'll see that in a second.

```
self.getStream = function ()
{
    return self._stream;
}
```

The `getStream()` method just returns our `_stream`, which if we haven't loaded one, will be blank. So let's look at the `loadStream()` method:

```
self.loadStream = function ( completion )
{
    var theStreamURL;
    var forScreenName = false;
```

The `loadStream()` method takes a completion function; we'll call this at the end of the operation no matter what. It lets the rest of our code know when it is safe to access the `_stream` member via `getStream()`.

The other component is the `forScreenName` variable; if true, we'll be asking Twitter for the stream that belongs to the screen name stored in `_searchPhrase`; otherwise, we'll ask Twitter to do an actual search for `_searchPhrase`. This variable is defined in the following code snippet:

```
if (self._searchPhrase.substr(0,1)=="@")
{
    theStreamURL = TWITTER._baseURL +
    "statuses/user_timeline.json?include_entities=
    true&include_rts=true&count=" +
    self._theMaxCount + "&screen_name=" +
    encodeURIComponent(self._searchPhrase);
    forScreenName = true;
}
else
{
    theStreamURL = TWITTER._searchBase +
    "search.json?q=" +
    encodeURIComponent(self._searchPhrase) +
    "&include_entities=true" +
    "&include_rts=true&rpp=" + self._theMaxCount;
    forScreenName = false;
}
```

All we've done so far is define `theStreamURL` to point either at the search API (for a search term) or the non-search API (for a screen name's stream). Next we'll load it with `loadJSON()`, as shown in the following code snippet:

```
PKUTIL.loadJSON ( theStreamURL, function (success,
    data)
{
    if (success)
    {
        if (forScreenName)
        {
            self._stream = data;
        }
        else
        {
            self._stream = data.results;
        }
    }
}
```

Here's another reason why we need to know if we're processing for a screen name or for a search, the JSON we get back is slightly different. When searching, Twitter helpfully includes other information (such as the time it took to execute the search). In our case, we're not interested in anything but the results, hence the two separate code paths.

```
        else
        {
            self._stream = { "error": "Twitter error; rate
                            limited?" };
        }
```

Again, if we have a failure, we're assuming that we are rate-limited.

```
    if (completion)
    {
        completion( self._stream );
    }
```

When done, we call the `completion` method, helpfully passing along the data stream.

```
    }
    );
}
self.setSearchPhrase ( theScreenNameOrSearchPhrase );
self.setMaxCount ( theMaxCount || 25 );
self.loadStream ( completion );
}
```

Just like at the end of the last object, we call some methods at the end of this object too. First we set the incoming search phrase, then set the maximum number of tweets to return (or 25, if it isn't given to us), and then call the `loadStream()` method with the `completion` function. This means that the moment we create a new `TwitterStream` object, it's already working on loading all the tweets we'll be wanting to have access to.

We've taken care of almost all our data model, but we've just a little bit left to do in `twitterUsers.js`:

```
TWITTER.users = Array();
```

First, we create a `users()` array in the `TWITTER` namespace. We're going to use this to store our predefined Twitter users, which will be loaded with the `loadTwitterUsers()` method as follows:

```
TWITTER.loadTwitterUsers = function ( completion )
{
```

```
TWITTER.users.push ( new TWITTER.TwitterUser ( "photoKandy" ,  
function ()  
{ TWITTER.users.push ( new TWITTER.TwitterUser ( "CNN" ,  
function ()  
{ TWITTER.users.push ( new TWITTER.TwitterUser (  
"BBCWorld" , function ()  
{ TWITTER.users.push ( new TWITTER.TwitterUser ( "  
"espn" , function ()  
{ TWITTER.users.push ( new TWITTER.TwitterUser (  
"lemondefr" , completion ) ); }  
) ); }  
) ); }  
) ); }  
} ); }  
}
```

What we've done here essentially is just chained together five requests for five different Twitter accounts. You can store these in an array and ask for them all at once, yes, but our app needs to know when they've all loaded. You could also do this by using recursion through an array of users, but we'll leave it as an example to you, the reader.

What did we do?

We implemented our data model and predefined the five Twitter accounts we want to use. We also went over the `loadJSON()` method in `PKUTIL` which helps with the entire process. We've also been introduced to the Twitter API.

What else do I need to know?

Before we go on, let's take a look at the `loadJSON()` method you've been introduced to. It's been added to this project's `www/framework/utility.js` file, and is defined as follows:

```
PKUTIL.loadJSON = function ( theURL, completion )  
{  
    PKUTIL.load( theURL, true, function ( success, data )  
    {
```

First off, this is a pretty simple function to begin with. What we're really doing is utilizing `PKUTIL.load()` to do the hard work of calling out to the URL and passing us the response, but when the response is received, it's going to be coming back to us in the `data` variable.

```
var theParsedData = { };
```

The `theParsedData` variable will store the actual JSON data, fully parsed.

```
if (success)
{
    try
    {
        theParsedData = JSON.parse ( data );
    }
}
```

If the URL returns something successfully, we try to parse the data. Assuming it is a valid JSON string, it'll be put into `theParsedData`. If it isn't, `JSON.parse()` will throw an exception, as shown in the following code block:

```
}
catch (err)
{
    console.log ("Failed to parse JSON from " + theURL);
    success = COMPLETION_FAILURE;
}
```

Any exceptions will be logged to the console, and we'll end up telling our `completion` function that the request failed, as shown in the following code block:

```
}
if (completion)
{
    completion (success, theParsedData);
}
```

At the end, we call our `completion` function and tell it if the request failed or succeeded, and what the JSON data was (if successfully parsed).

```

);
}
```

Configuring the plugins

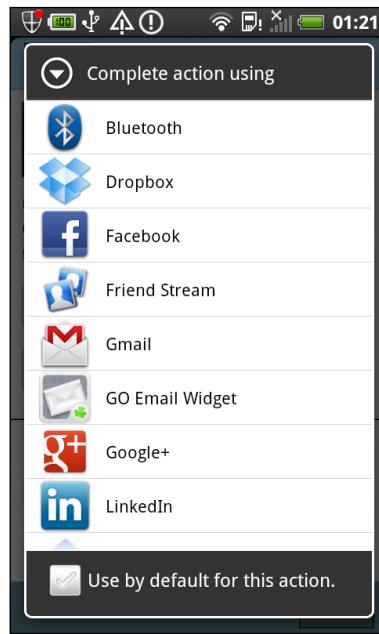
Most PhoneGap plugins aren't terribly hard to install or configure, but they will undoubtedly play a vital role in your app, especially if you need to use a feature that PhoneGap doesn't provide on its own.

In our case, we need two plugins, one to display websites within our app and another to share the content. For the first, we'll be using a plugin called `ChildBrowser` across all the platforms we support, but for the latter, we'll have to use a separate plugin for iOS and Android.

Here's what each sharing plugin will look like when we're done, starting with iOS:



For Android the sharing plugin will look like the following:



Getting ready

If you haven't already, you should download the entire community PhoneGap plugin repository located at <https://github.com/phonegap/phonegap-plugins>. This will provide you with nearly all the content necessary to use the plugins.

If you wish to support sharing on iOS, you also need to download ShareKit 2.0, which is available at <https://github.com/ShareKit/ShareKit>, or use the fork bundled with the code available for this book. It is located outside of the directories for each project in a directory labeled Submodules.

Getting on with it

We're going to split this one up into what we have to do for each platform, as the steps and environments are all quite different.

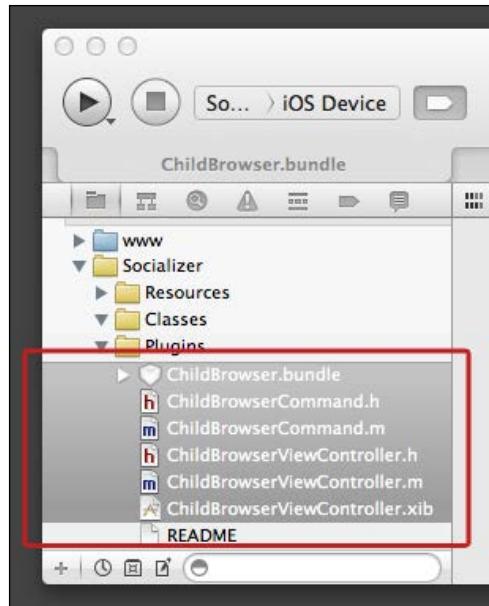
Plugin configuration for iOS

You wouldn't know it, but our first platform is also the hardest. In fact, it'll make the remaining two platforms feel a bit like child's play.

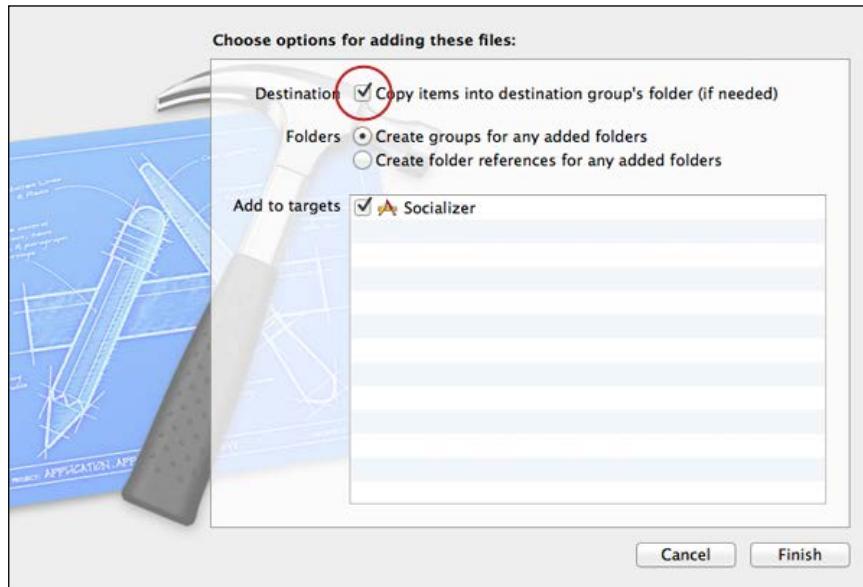
The ChildBrowser plugin itself is easy to install and configure, but ShareKit 2.0 is, well, anything but, especially when used in conjunction with PhoneGap. The problem stems from the fact that when you compile the project with PhoneGap and ShareKit 2.0 together, some symbols are duplicated, and the linker throws out a nasty little error. Long story short, your app doesn't compile. Not good.

Let's look first at the steps necessary for installing the ChildBrowser plugin, as these are far more typical of most plugins:

1. Open the collection of plugins you downloaded and navigate to `ios/ChildBrowser`.
2. Drag `ChildBrowser.bundle`, `ChildBrowserCommand.h`, `ChildBrowserCommand.m`, `ChildBrowserViewController.h`, `ChildBrowserViewController.m`, and `ChildBrowserViewController.xib` into XCode to `Socializer/Plugins`, as shown in the following screenshot:



3. At the prompt, make sure to copy the files (instead of linking to them). This can be done by checking the **Copy items into destination group's folder** entry, as shown in the following screenshot:

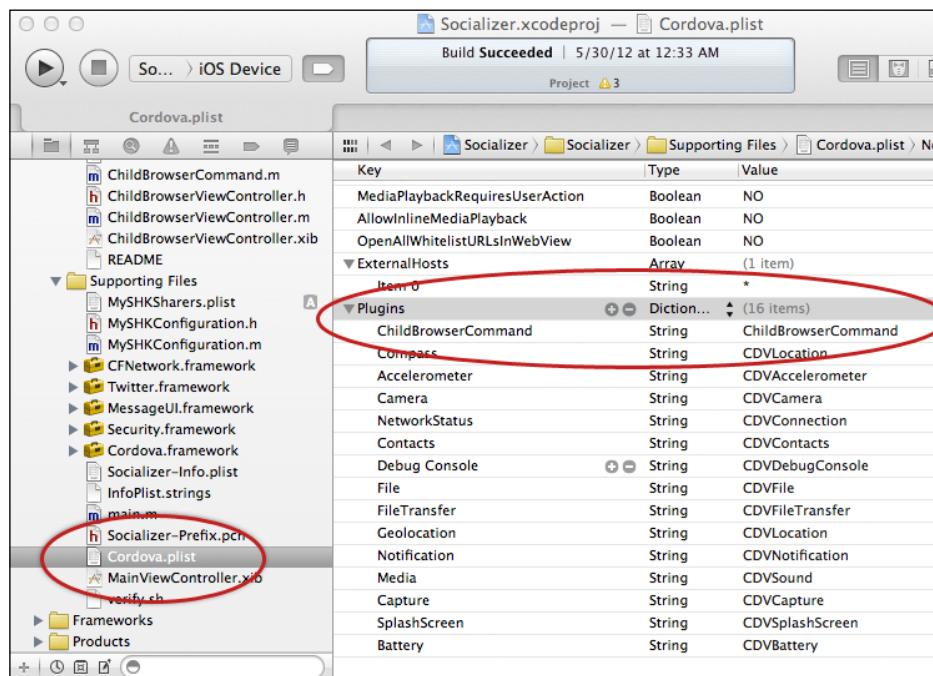


Let's Get Social!

4. Copy ChildBrowser.js to your www/plugins/iOS directory. You can do this in XCode or in Finder.
5. Add the plugin to Cordova.plist in Socializer/Supporting Files in XCode. Find the Plugins row, and add a new entry of the order as seen in the following table:

ChildBrowserCommand	String	ChildBrowserCommand
---------------------	--------	---------------------

This can be better explained with the help of the following screenshot:



There, that was easy, wasn't it? Now comes the hard part, getting ShareKit 2.0 installed and working. For that, we're going to refer to *Appendix B, Installing ShareKit 2.0* as the process is rather long.

When that's done, we need to duplicate the plugin setup that we did for ChildBrowser, except for ShareKit, using the following steps:

1. Navigate to the iOS/ShareKitPlugin directory in your plugin repository.
2. Copy ShareKitPlugin.h, ShareKitPlugin.m, SHKSharer+Phonegap.h, SHKSharer+Phonegap.m to the Plugins folder in your project.

3. Copy ShareKitPlugin.js to your www/plugins/iOS folder.
4. Modify Cordova.plist to add this new plugin to the list.
5. Find the **Plugins** row, and add a new entry, as in the following table:

ShareKitPlugin	String	ShareKitPlugin
----------------	--------	----------------

This can be better explained with the help of the following screenshot:

▼ ExternalHosts	Array	(1 item)
Item 0	String	*
▼ Plugins	Diction...	(16 items)
ChildBrowserCommand	String	ChildBrowserCommand
ShareKitPlugin	String	ShareKitPlugin
Compass	String	CDVLocation
Accelerometer	String	CDVAccelerometer
Camera	String	CDVCamera
NetworkStatus	String	CDVConnection
Contacts	String	CDVContacts

The final step is to update our www/index.html file to include these two plugins for our app. Add the following lines after the line that is loading the cordova-2.2.0-ios.js script:

```
<script type="application/javascript" charset="utf-8">
  src="../plugins/iOS/ShareKitPlugin.js"></script>
<script type="application/javascript" charset="utf-8">
  src="../plugins/iOS/ChildBrowser.js"></script>
```

Whew! We did it, we've got two plugins installed for iOS devices. Now, let's tackle the remaining platform. Don't faint. Android's a lot easier.

Plugin configuration for Android

For Android, we'll be using two plugins: namely, ChildBrowser and Share. Both are located in the repository you should have already downloaded from GitHub. Let's start with installing and configuring ChildBrowser first, using the following steps:

1. Create a new package (**File | New | Package**) under your project's src folder. Name it com.phonegap.plugins.childBrowser.
2. Navigate to Android/ChildBrowser/src/com/phonegap/plugins/childBrowser and drag ChildBrowser.java to the newly created package in Eclipse.

3. Go to `res/xml` in your project and open `plugins.xml` with the text editor (usually this is done by a right-click and then navigating to **Open With | Text Editor**).

4. Add the following line at the bottom of the file, just above the `</plugins>` ending tag:

```
<plugin name="ChildBrowser" value="com.phonegap.plugins.  
childBrowser.ChildBrowser"/>
```

5. Navigate to the `Android/ChildBrowser/www` folder in the repository.

6. Copy `childbrowser.js` to `assets/www/plugins/Android`.

7. Copy the `childbrowser` folder to `assets/www` (copy the folder, not the contents, and you should end up with `assets/www/childbrowser` when done).

For our next plugin, `Share`, use the following steps:

1. Create a package in Eclipse under your project's `src` directory named `com.schaul.plugins.share`.

2. Navigate to `Android/Share` in the plugin repository and copy `Share.java` to the package in Eclipse.

3. Add the following line at the bottom of the `plugins.xml` file:

```
<plugin name="Share" value="com.schaul.plugins.share.Share"/>
```

4. Copy `share.js` to your project's `assets/www/plugins/Android` directory.

5. The last step is to update our `www/index_Android.html` file by adding the following lines just below the portion that is loading the `cordova-2.2.0-android.js` file:

```
<script type="application/javascript"  
charset="utf-8"  
src=".//plugins/Android/childbrowser.js"></script>  
<script type="application/javascript"  
charset="utf-8"  
src=".//plugins/Android/share.js"></script>
```

That's it! Our plugins are correctly installed and configured for Android.

What did we do?

We set up the `ChildBrowser` plugin on both of our supported platforms. We set up `ShareKit 2.0` and the `ShareKitPlugin` for iOS, and the `Share` plugin for Android.

What else do I need to know?

We've not actually dealt with how to use the plugins; we just installed them. We'll be dealing with that as we come to the necessary steps when implementing our project. But there is one important detail to pay attention to: the plugin's `readme` file, if available.

This file will often indicate the installation steps necessary, or any quirks that you might need to watch out for. The proper use of the plugin is also usually detailed. Unfortunately, some plugins don't come with instructions. At that point, the best thing to do is to try installing it in the *normal* fashion (as we've done earlier for `ChildBrowser` and all the other plugins except for `ShareKit`) and see if it works.

The other thing to remember is that PhoneGap is an ongoing project. It means that there are plugins that are out-of-date (and indeed, some have had to be updated by the author for this book) and won't work correctly with the most recent versions of PhoneGap. You'll need to pay attention to the plugins so that you know which version it supports, and if it will need to be modified to work with a newer version of PhoneGap. Modifications usually aren't terribly difficult, but it does involve getting into the native code, so you may want to ask the community for help in the modification. (See the end of the project for links to the community.)

Implementing the social view

While our app has three views, the start view is so similar to the previous project's start view that we won't go into great detail in this project about how it works. You're welcome to take a look at the code in the `www/views/startView.html` file.

The bulk of our code is going to reside in the social view and the tweet view, so that's where our primary focus will be. So let's get started!

Getting ready

Go ahead and create the `socialView.html` file now based on what we have discussed. Then we'll go over the portions you haven't seen before.

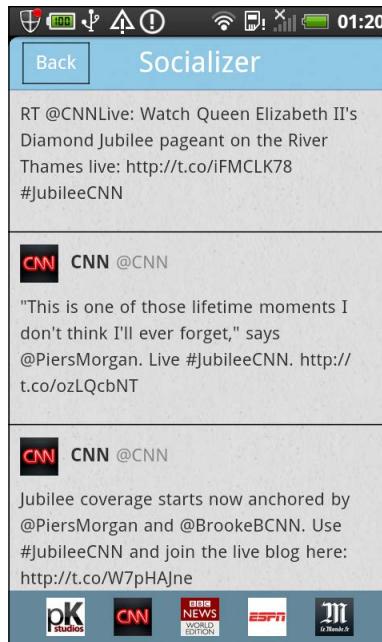
Let's Get Social!

Getting on with it

When we're finished with this task, we should have a view that looks like this for iOS:



The view for Android will be as follows:



As with all our views to this point, we're going to start with the HTML portion that describes the actual view; it is given as follows:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div id="socialView_title"></div>
    <button class="barButton backButton"
      id="socialView_backButton" style="left:10px" ></button>
  </div>
  <div class="content avoidNavigationBar avoidToolBar"
    style="padding:0; overflow: scroll;"
    id="socialView_scroller">
    <div id="socialView_contentArea" style="padding: 0;
      height: auto; position: relative;">
    </div>
  </div>
  <div class="toolBar" id="socialView_toolbar" style="text-align: center">
  </div>
</div>
```

Generally, this looks very much like our previous views, except that there are a couple of critical details. We've added a style to the inner `div` element. This takes away our default `div` styling (from `www/framework/base.css`) and forces the height to fit to the content (instead of to the screen). This means that when we want to scroll, we'll have the whole content to scroll through.

This is the first time we've talked about scrolling in our apps at all, actually, and for good reason: it's often hard on mobile platforms. In a perfect world, we could just rely on `overflow:scroll` to work on all our platforms, but that simply doesn't work. We can rely on native scrolling in iOS 5 and later, but that has its own share of problems (depending on the version of PhoneGap and various other WebKit gotchas), and rules out any lower platform, and of course, it doesn't work on Android at any version. So for iOS and Android, we'll have to use our own implementation for scrolling or use a third-party scrolling library such as iScroll 4. In this case, we're using our own implementation, which we'll cover a little later.

First we need to determine how our toolbar will show its profile images using the following template:

```
<div class="hidden" id="socialView_profileImageIcon">
  <a class="profileImage" style="background-
    image:url(%PROFILE_IMAGE_URL%)"
    href="javascript:socialView.loadStreamFor
    ('@%SCREEN_NAME%');" ></a>
</div>
```

Let's Get Social!

Note that we have a little bit of JavaScript that fires when the user touches the image, this is to load the appropriate stream for that image.

Next, we need to define what the tweets should look like within our view. This is done by using the following code snippet:

```
<div class="hidden" id="socialView_twitterTemplate">
  <div class="twitterItem" onclick="socialView.selectTweet (%INDEX%);">
    
    <div class="twitterName">
      <span class="twitterRealName">%REAL_NAME%</span>
      <span class="twitterScreenName">@%SCREEN_NAME%</span>
    </div>
    <div class="twitterTweet">%TWEET%</div>
  </div>
</div>
```

In this segment of HTML, we've defined what the rest of a tweet should look like. We've given every `div` and `span` a class so that we can target them in our `style.css` file (located in `www/style`). That is mainly to keep the display of the tweet as separate from the content of the tweet as possible and to make it easy to change the look of a tweet whenever we want. Go ahead and take a look at the `style.css` file to get a good idea of how they will work to give our tweets some style.

Next up is our code:

```
var socialView = $ge("socialView") || {};
socialView.firstTime = true;
socialView.currentStream = {};
socialView.lastScrollTop = 0;
socialView.myScroll = {};
```

As always, we give ourselves a namespace, in this case `socialView`. We also declare a few properties: `firstTime`, which will track if this is the first time our view is being displayed or not, and `currentStream`, which will hold the current visible stream from Twitter. The `lastScrollTop` property will record the position the user has scrolled to on our current page so we can restore it when they return from looking an individual tweet, and `myScroll` will hold our actual scroller.

```
socialView.initializeView = function ()
{
  PKUTIL.include ( ["./models/twitterStreams.js",
    "./models/twitterStream.js"], function ()
  {
```

```
// load our toolbar
TWITTER.loadTwitterUsers (
    socialView.initializeToolbar );
}

);

socialView.viewTitle = $ge("socialView_title");
socialView.viewTitle.innerHTML = __T("APP_TITLE");

socialView.backButton = $ge("socialView_backButton");
socialView.backButton.innerHTML = __T("BACK");
PKUI.CORE.addTouchListener(socialView.backButton,
    "touchend", function () { PKUI.CORE.popView(); });

if (device.platform != "WinCE")
{
    socialView.myScroll = new SCROLLER.
        GenericScroller ('socialView_contentArea');
}

}
```

Our initializeView() method isn't terribly different from our previous project. I've highlighted a couple lines, however – note that we load our models and when they are complete, we call TWITTER.loadTwitterUsers(). We pass along a completion function, which we define next so that when Twitter has returned the user data for all five of our Twitter users, we can call it.

We've also defined our scroller. If you want to see the complete code take a look in www/framework/scroller.js, but it should suffice to say, it is a reasonably nice scroller that is simple to use. It doesn't beat native scrolling, but nothing will. You're free to replace it with any library you'd like, but for the purposes of this project, we've gone this route.

```
socialView.initializeToolbar = function ()
{
    var toolbarHtml = "";
    var profileImageTemplate =
        $ge("socialView_profileImageIcon").innerHTML;
    var users = TWITTER.users;

    if (users.error)
```

```
{  
    console.log (streams.error);  
    alert ("Rate limited. Please try again later.");  
}
```

One of the first things we do after obtaining the template's HTML is to check on our `TWITTER.users` array. This array should have been filled with all sorts of user data, but if Twitter has rate-limited us for some reason, it may not be. So we check to see if there is an error condition, and if so, we let the user know. Granted, it's not the best method to let a user know, but for our example app, it suffices.

```
// go through each stream and request the profile image  
for (var i=0; i<users.length; i++)  
{  
    var theTemplate = profileImageTemplate.replace  
        ("%SCREEN_NAME%", users[i].getScreenName())  
        .replace ("%PROFILE_IMAGE_URL%",  
            users[i].getProfileImageURL());  
    toolbarHtml += theTemplate;  
}
```

Next, we iterate through each of the users. There should be five, but you could configure it for a different number and build up an HTML string that we'll put into the toolbar as follows:

```
$ge("socialView_toolbar").innerHTML = toolbarHtml;  
}
```

Our next function, `loadStreamFor()` does the really hard work in this view. It requests a stream from Twitter and then processes it for display. The code snippet for it is as follows:

```
socialView.loadStreamFor = function ( searchPhrase )  
{  
    var aStream = new TWITTER.TwitterStream ( searchPhrase,  
        function ( theStream )  
    {
```

Something to note is that we are now inside the `completion` function, the function that will be called when the Twitter stream is obtained.

```
        var theTweetTemplate =  
            $ge("socialView_twitterTemplate").innerHTML;  
        var theContentArea = $ge("socialView_contentArea");  
        var theStreamHTML = "";
```

```
if (theStream.error)
{
    console.log (theStream.error);
    alert ("Rate limited. Please try again later.");
}
```

Because Twitter may rate-limit us at any time, we check again for any error in the stream in the preceding code snippet.

```
for (var i=0; i<theStream.length; i++)
{
    var theTweet = theStream[i];
    var theTemplate =
        theTweetTemplate.replace("%INDEX%", i)
            .replace ("%PROFILE_IMAGE_URL%",
                      theTweet.profile_image_url ||
                      theTweet.user.profile_image_url)
            .replace ("%REAL_NAME%",
                      theTweet.from_user ||
                      theTweet.user.name)
            .replace ("%SCREEN_NAME%",
                      theTweet.from_user ||
                      theTweet.user.screen_name)
            .replace ("%TWEET%",
                      theTweet.text);
    theStreamHTML += theTemplate;
}
```

Here we're iterating through each item in the stream and building up a large HTML string from the template we defined earlier.

One important part to notice is how we're obtaining the data of the tweet, using `theTweet.from_user || theTweet.user.screen_name` and such. This is to deal with how Twitter returns a slightly different data format when searching for a word or a hashtag versus the data format when returning a user's timeline. Should one be undefined, we'll load the other, and since we can only get one or the other, it's easier than building a lot of if statements to take care of it.

```
theContentArea.innerHTML = theStreamHTML;
socialView.currentStream = theStream;
if (socialView.myScroll.scrollTo)
```

```
{  
    socialView.myScroll.scrollTo ( 0, 0 );  
}
```

Once our stream HTML is built, we assign it to the content area so that the user can see it. We also store the stream into the `currentStream` property so we can reference it later. When that's done, we scroll to the top of the page so that the user can see the most recent tweets.

```
}  
, 100  
);  
}
```

That last `100`? Well, it's actually part of the call to `TwitterStream()`. It's the number of items to return in the stream.

Our next function deals with what should happen when a user taps on a displayed tweet:

```
socialView.selectTweet = function ( theIndex )  
{  
    var theTweet = socialView.currentStream[theIndex];  
    tweetView.setTweet ( theTweet );  
    PKUI.CORE.pushView ( tweetView );  
}
```

This function is pretty simple. All we do is tell the tweet view what tweet was tapped, and then push it on to the view stack.

```
socialView.viewWillAppear = function ()  
{  
    document.addEventListener("backbutton",  
        socialView.backButtonPressed, false );  
    if (socialView.firstTime)  
    {  
        socialView.loadStreamFor ( "@photokandy" );  
        socialView.firstTime = false;  
    }  
    if (socialView.myScroll.scrollTo)  
    {  
        PKUTIL.delay ( 50, function ()  
        {  
            socialView.myScroll.scrollTo ( 0,  
                socialView.lastScrollTop );  
        }  
    }  
}
```

```
        }  
    );  
}  
}
```

This `viewWillAppear()` method is pretty similar to the last project except for the middle and the last portion. In the middle we're checking if this is the first time the view has been displayed. If it is, we want to load a default stream for the user. Remember, up till now we've only loaded a stream when the user taps on a profile image in the toolbar. But we don't want to reload this stream every time our view displays; we could be coming back from the tweet view and the user might want to continue where they left off in the previous stream. In the final portion, we're checking to see if we had a previous scroll position, and if so, we scroll the view to that point. We have to create a delay here, since if we set it too early, the view will be offscreen (and won't scroll), or it will be onscreen, and it'll be noticeable to the user.

The remaining two functions, `viewWillDisappear()` and `backButtonPressed()` present no new functionality, so while you do need them in your code, we won't go over them here.

That's it, not terribly difficult, but it does what we need—display a list of tweets. Once a user taps on the tweet, they'll be taken to the tweet view to do more, and that's what we'll look at in the next task.

What did we do?

In this task we defined the HTML code and templates for our social view. We also used the Twitter-stream data to construct a Twitter stream that the end user can interact with.

Implementing the tweet view

Our tweet view will be where the user interacts with a given tweet. They can open any links within the tweet using the `ChildBrowser` plugin, or they can search any hashtags contained within the tweet (or any mentions, too). The view also gives the user the opportunity to share the tweet to any of their social networks.

Getting ready

Go ahead and create your own `www/tweetView.html` file based on the one we discussed. We'll go over the code that is new, while leaving the rest to you to review.

Let's Get Social!

Getting on with it

For this next task, we should end up with a view that looks like the following on iOS:



For Android, the view will be as follows:



This time, we're not going to display the HTML for defining the layout of our view. You may ask why? This is because you've seen it several times before and can look it up in the code for this project. We're going to start with the templates that will define the content instead:

```
<div class="hidden" id="tweetView_tweetTemplate">
  <div class="twitterItem" onclick="tweetView.selectTweet (%INDEX%);">
    
    <div class="twitterRealName">%REAL_NAME%</div>
    <div class="twitterScreenName">@%SCREEN_NAME%</div>
    <div class="twitterTweet">%TWEET%</div>
    <div class="twitterEntities">%ENTITIES%</div>
  </div>
</div>
```

This code is pretty similar to the template in the previous view with a couple of exceptions: one that we've made the profile image larger, and two, we've added a `div` element that lists all the *entities* in the tweet. Twitter defines an entity as a URL, a hashtag, or a mention of another twitter user. We'll display any of these that are in a tweet so that the user can tap on them to get more information.

```
<div class="hidden" id="tweetView_entityTemplate">
  <DIV class="entity %TYPE%">%ENTITY%</DIV>
</div>
```

Here's our template for any entity. Notice that we've given it the class of `entity`, so that all our entities can have a similar appearance.

Next up, we define what each particular entity looks like, in this case, the URL template.

```
<div class="hidden" id="tweetView_urlEntityTemplate">
  <a href="javascript:PKUTIL.showURL ('%URL%');"
    class="openInNewWindow url" target="_blank">%DISPLAYURL%</a>
  </div>
```

Note the use of `PKUTIL.showURL()` in this template. It is a convenience method we've defined in `PKUTIL` to use `ChildBrowser` to show a webpage. We've done the work of combining how it works on each platform and put it into one function so that it is easy to call. We'll take a look at it a little later.

```
<div class="hidden" id="tweetView_hashEntityTemplate">
  <a href="javascript:socialView.loadStreamFor ('%23%HASHTAG%');"
    PKUI.CORE.popView();" class="hash">%TEXT%</a>
  </div>
```

This template is for a hashtag. The big difference between this and the previous template is that it is actually referring back to our previous view! It does this to tell it to load a stream for the hashtag, and then we call `popView()` to go back to the view. Chances are the view won't have the loaded information from Twitter just yet, but give it a second and it'll reload with the new stream.

Similarly, the code for a mention is as follows:

```
<div class="hidden" id="tweetView_userEntityTemplate">
  <a href="javascript:socialView.loadStreamFor('@%USER%');
    PKUI.CORE.popView();" class="user" >@%TEXT%</a>
</div>
```

So that defines how our tweet looks and works, let's see how the view actually creates the tweet itself:

```
var tweetView = $ge("tweetView") || {};
tweetView.theTweet = {};
tweetView.setTweet = function ( aTweet )
{
  tweetView.theTweet = aTweet;
}
```

Here, we've defined the `setTweet()` method, which stores a given tweet into our `theTweet` property. Remember, this is called from the Twitter stream view when a tweet is tapped to send us the tweet to display.

The next method of interest is `loadTweet()`. We'll skip the `initializeView()` method as it is similar to the previous view. The `loadTweet()` method is given as follows:

```
tweetView.loadTweet = function ()
{
  var theTweet = tweetView.theTweet;

  var theTweetTemplate =
    $ge("tweetView_tweetTemplate").innerHTML;
  var theEntityTemplate =
    $ge("tweetView_entityTemplate").innerHTML;
  var theURLEntityTemplate =
    $ge("tweetView_urlEntityTemplate").innerHTML;
  var theHashEntityTemplate =
    $ge("tweetView_hashEntityTemplate").innerHTML;
  var theUserEntityTemplate =
    $ge("tweetView_userEntityTemplate").innerHTML;
```

First, we obtain the HTML for each template we need—and there are several! These are given as follows:

```
var theContentArea = $ge("tweetView_contentArea");
var theTweetHTML = "";
var theEntitiesHTML = "";

var theURLEntities = theTweet.entities.urls;
for (var i=0;i<theURLEntities.length;i++)
{
    var theURLEntity = theURLEntities[i];
    theEntitiesHTML += theEntityTemplate.replace
        ("%TYPE%", "url")
        .replace ("%ENTITY%",
            theURLEntityTemplate.replace ("%URL%",
                theURLEntity.url )
            .replace ("%DISPLAYURL%",
                theURLEntity.display_url )
        );
}
```

In this code, we've gone through every URL entity that Twitter has sent us and added it to our entity HTML string. We'll repeat that for hashtags and for mentions, but the code is so similar, we won't repeat it here.

```
var theTemplate = theTweetTemplate
    .replace ("%PROFILE_IMAGE_URL%",
        theTweet.profile_image_url ||
        theTweet.user.profile_image_url)
    .replace ("%REAL_NAME%",
        theTweet.from_user ||
        theTweet.user.name)
    .replace ("%SCREEN_NAME%",
        theTweet.from_user ||
        theTweet.user.screen_name)
    .replace ("%TWEET%", theTweet.text)
    .replace ("%ENTITIES%", theEntitiesHTML );
theTweetHTML += theTemplate;
theContentArea.innerHTML = theTweetHTML;
```

Once we've gone through all the entities, we handle the tweet itself. Note that we had to handle the entities first because we handle the substitution earlier. Just like with the previous view, we correctly handle if the tweet is from a search or from a timeline as well.

The next method of interest is the `share()` method, so we'll skip over `viewWillAppear()`, `viewWillHide()`, and `backButtonPressed()`. Suffice to say, the only different thing the `viewWillAppear()` method does than any of the others is call the `loadTweet()` method to display the tweet when our view is shown.

The `share()` method is where we call each platform's plugin for sharing. Each platform has a slightly different syntax, so we have to check which platform we're on and decide which plugin to call based on that. We can do so using the following code snippet:

```
tweetView.share = function ()  
{  
    switch (device.platform)  
    {  
        case "Android": window.plugins.share.show(  
            { subject: 'Share',  
              text: tweetView.theTweet.text,  
            },  
            function() {},  
            function() { alert ('Error sharing.'); }  
        );  
        break;  
  
        default:  
            window.plugins.shareKit.share (  
                tweetView.theTweet.text );  
    }  
}
```

For Android, we're using the `Share` plugin, and this is how we can share with it. Android will then display a list of services that support sharing, including Twitter and Facebook, if the user has them installed. The text we give it will be included in the message, and Android is nice enough to let us send a success and a failure function should we want to do something after the tweet.

Our default method is for iOS, which will display an action sheet listing a few services, probably Twitter and Facebook, and the user can tap the button of the service they want to share to. Once they authenticate to the service, they can then send the message.

What did we do?

We displayed a single tweet and processed the various entities within it. We demonstrated loading an external site in the `ChildBrowser` plugin by using `PKUTIL.showURL()`. We also demonstrated how to use the various sharing plugins.

What else do I need to know?

Let's take a quick look at `PKUTIL.showURL()`, the method used to display a `ChildBrowser` with an external site. It's a pretty simple function, but since it takes three different ways to show `ChildBrowser`, we packaged it up into a function that makes it easy to use.

```
PKUTIL.showURL = function ( theURL )
{
    switch (device.platform)
    {
        case "Android":
            window.plugins.childBrowser.showWebPage( theURL );
            break;
```

For Android, it's simple to call `ChildBrowser`. Typically this is how you call any plugin you want to use in PhoneGap.

```
case "WinCE":
    var options =
    {
        url:theURL,
        geolocationEnabled:false
    };
    Cordova.exec(null, null,"ChildBrowserCommand",
        "showWebPage", options);
    break;
```

WP7 is here too, since the platform supports it, and is a little more difficult. We have to pack the URL into an options array and then send it to the plugin to show.

```
default:
    cordova.exec ("ChildBrowserCommand.showWebPage",
        theURL);
}
```

For iOS, it's very similar to Android's method, except we call it directly instead of using `window.plugins.*`.

Game Over..... Wrapping it up

Well, you've done it. You've successfully written an app that displays information obtained from Twitter and that lets the user share it on their own social network. For some platforms, it wasn't too hard to configure the plugins necessary to do this work, while for iOS, you probably got a bit more familiar with Xcode and header paths and such than you wanted. The rest was easy; adding plugins gets easier the more you do it, and chances are pretty good you'll need at least the `ChildBrowser` plugin in nearly every project you do. Thankfully, it's also an easy install!

Some resources you might find valuable are as follows:

- ▶ ShareKit: <https://github.com/ShareKit/ShareKit>
- ▶ JSON: <http://www.json.org/>
- ▶ Twitter JSON documentation: <https://dev.twitter.com/docs/api/1>
- ▶ Phonegap plugins: <http://www.github.com/phonegap/phonegap-plugins>
- ▶ Phonegap community: <http://groups.google.com/group/phonegap>
- ▶ iScroll 4: <http://cubiq.org/iscroll-4>

Can you take the HEAT? The Hotshot Challenge

As a project, Socializer does what it is set out to do, but there's actually so much more that you could do to it to make it truly useful. Why don't you try one or more of the following challenges:

- ▶ Let the end user select their own initial Twitter accounts, instead of our initial five.
- ▶ Display a loading graphic while the Twitter stream is loading so that the user knows that the app is working on something.
- ▶ Style any links, mentions, or hashtags in the Twitter stream to make them stand out more.
- ▶ Try your hand at working with the API of any other social network you like.
- ▶ Try to add OAuth authentication.

Project 3

Being Productive

PhoneGap can be used for more than simple games and social media apps; it can also be used to create productivity apps that can be very useful. To do that, however, we need to learn about how to store persistent data using PhoneGap's File APIs. In this project, we'll do just that. We'll build a simple note-taking app named Filer that uses the File API to manage the available notes.

What do we build?

At its core, Filer is more about file management than it is about taking notes, but it is absolutely critical that you get file management right. Users don't take it kindly when an app corrupts or loses their data, so you must make sure to manage it correctly. Once that is accomplished, you can move on to making the app more complex. Thankfully, the concepts you learn in this project can be applied to all your future apps.

What does it do?

As the name implies, the app permits the user to *file* away notes for later retrieval. Doing this requires the use of the File APIs provided by PhoneGap. Not only do we need to be able to save and load notes, but we need to manage them as well. This includes removing notes at the user's request, renaming them, and duplicating them as well.

Once a note is created or opened, the app itself becomes very simple, essentially a large TEXTAREA element that will accept any kind of text you want to put in it. We'll also take a look at good ways to save and retrieve the data you enter.

Why is it great?

This app is a great way to learn the File APIs present in PhoneGap to manage files that your App needs in order to save and retrieve data. We'll also consider how to present this to the user in a form they can easily understand.

How are we going to do it?

We'll be going about creating this app much like we have the past apps, using the following pointers:

- ▶ Designing the user interface
- ▶ Designing the data model
- ▶ Implementing the data models
- ▶ Implementing the documents view
- ▶ Implementing the file view

What do I need to get started?

You should be able to create your project and set it up much the same way as the prior apps. Call this project `Filer`.

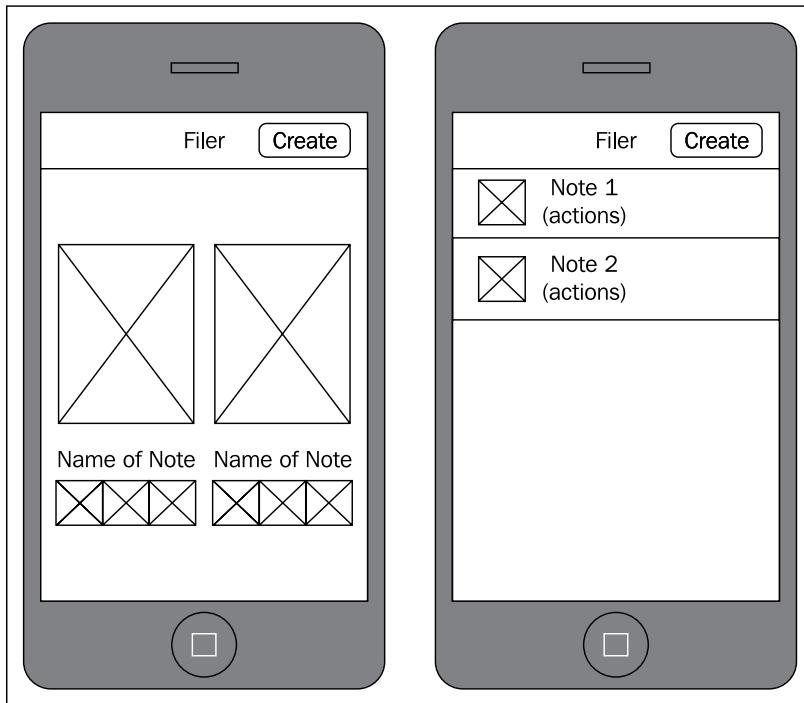
Note that, if you want, you can include the sharing libraries from the second project. We'll not use them directly, but there is a challenge at the end of the project that asks you to add sharing. If you intend to do this, you might as well add everything now.

Designing the user interface

First, get your paper and pencil out or use your favorite image editor. Like in previous projects, we'll design our views using sketches and wireframes first, then flesh them out a bit more to design the graphical assets.

Getting on with it

As in previous projects, the first view is the start view, but since it is the same as all the prior apps, we won't go into detail about it here (refer to the *Designing the UI/interactions* section of *Project 1, Let's Get Local!*). Instead, let's go to the documents view, shown in the following screenshot:



In this view we've actually got two looks; the left is for the iPhone, while the right is for Android. The reason for the two different looks is simply how a lot of apps do things on each platform. You typically see large, horizontal scrolling interfaces on iOS, and on Android you typically see vertical lists representing files.

Let's go over how this view works. The button in the navigation bar, named **Create**, allows the user to create a new note. Below the navigation bar is the list of files that are available. On the first run, of course, this will be empty, but as files are created, they are added here. This view will scroll as needed in order to show the entire list.

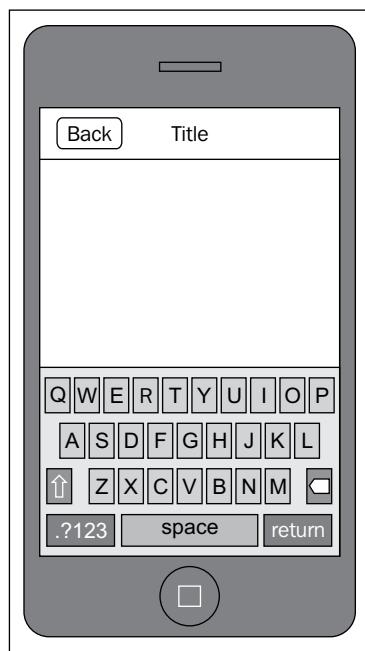
Each item in the list will have the same contents, even though they are arranged and sized differently. The first is the icon that represents the item; many apps will render a version of the content as this icon. To avoid complexity, we won't do that here; we'll use a static image instead. Tapping on the icon will open the note. The next is the icon's label, this shows the name of the file. When pressed, however, it will allow the user to rename the file.

Below the file name are the following three icons:

- ▶ For duplicating (copying) the note
- ▶ For sharing the note
- ▶ For destroying (deleting) the note

At first, none of this is terribly difficult, and really, it isn't. But the way the File APIs are implemented, it does take a bit of work to get right.

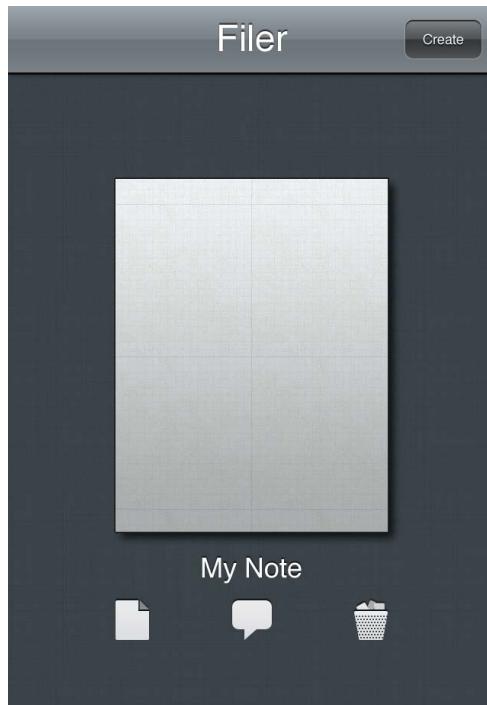
Let's move on to the file view, seen in the following screenshot:



This view is pretty simple: it displays the contents of the note and allows the user to edit it. Notice that there's no **Save** button; the idea is that the notes will save themselves automatically.

When the view first appears, the keyboard won't be visible. This allows the user to see the note fill the screen. Once the note is tapped, however, the keyboard will appear, and the user will be able to change the note to their desire.

Now that we've created the wireframes, let's go into our graphics program and create our resources. Here's what we came up with:



We'll take most of the interface here as images, the icons, the large paper image, and the navigation bar and view background itself as well. For Android, only the icons and the paper image matter; the latter two are for iOS only.

The icons themselves are obtainable from App-Bits for free (<http://app-bits.com/free-icons.html>), and the background texture is from Subtle Patterns, again for free (<http://subtlepatterns.com>).

What did we do?

In this task, we created the desired look and feel, and generated the necessary resources for our app.

Designing the data model

Go ahead and get your paper and pencil out again. We need to design the data model for the app. We'll have two portions: one to manage the list of available documents, and another to manage a single document.

Getting on with it

Here's what our model looks like:

FilerDocuments	FilerDocument
<ul style="list-style-type: none"> - state : string - documents : array - fileSystem : object - fileEntry : function - completion : function - failure : function - loadFileSystem - getFileSystem - dispatchFailure - getFileSystemSuccess - getDocuments - getDocumentsSuccess - getDocumentCount - getDocumentAtIndex - deleteDocumentAtIndex - deleteDocumentAtIndexSuccess - renameDocumentAtIndexTo - renameDocumentAtIndexToSuccess - copyDocumentAtIndexTo - copyDocumentAtIndexToSuccess - createDocumentAtIndex - openDocumentAtIndex - getFileEntry - getAvailableDocuments 	<ul style="list-style-type: none"> - fileEntry : object - fileName : string - state : string - completion : function - failure : function - title : string - text : string - getTitle - setTitle - getText - setText - readFileContents - dispatchFailure - gotFile - finishedReadingFile - saveFileContents - gotFileWriter - serialize

The first model, named **FilerDocuments**, is responsible for managing all the files available to the app, while the one on the right, named **FilerDocument**, is responsible only for a single note. The latter is responsible for loading a note and saving a note, while the former is responsible for reading an entire directory of notes and then managing them via renames, copies, and deletes.

A few notes of interest before we wrap this task up. Notice all methods that end with ... Success. This is simply due to the way the File API is structured; everything is done asynchronously so you have to write each call to it with callbacks to both a success and a failure function. The success function points at the corresponding Success method, while the failure function points at the generic dispatchFailure method. (Failures are pretty generic; we want to log the failure, whereas successes may require additional steps to complete an operation.)

The `fileSystem` and `fileEntry` properties are also related to the File API. The `fileEntry` property is a pointer to a specific file, while the `fileSystem` property is a pointer to a specific directory on the device. (PhoneGap lets you specify if the directory should be a persistent one or a temporary one; we're using persistent.)

On the second model, note the title and text properties as well as the associated get/set methods. This is the actual data of a single note; everything else was simply to manage it.

What did we do?

We created our data model for the document manager and a single note. In the next task, we'll implement both.

Implementing the data models

At this point you should have your project already created. We're going to be creating two models under the `www/models` directory, named `filerDocuments.js` and `filerDocument.js`.

Getting on with it

Let's get started by working on the `Documents` model that manages all the available documents:

```
var DOCS = DOCS || {};  
  
DOCS.Filers = function ( completion, failure )  
{  
    var self = this;  
  
    self.state = "";
```

This is the beginning of our constructor for the `Filers` object. The `completion` and `failure` variables are passed in because at the end of the constructor we will kick off a directory read operation and we want to alert the application when we're finished (or we get an error).

```
    self.state = "";
```

The `state` property will store the current progress of an operation, which should make it easier to debug if an operation fails.

```
    self.completion = completion;
```

The `completion` function here initially receives the `completion` but it also stores the `completion` function used by other functions within the object. This is because an operation may take several steps, each requiring an interim `completion` method. This just happens to be one from the app, not from within our object.

```
    self.documents = [];
```

The `documents` property stores the information received from the filesystem on each file we can read. It's not the actual document.

```
    self.fileSystem = {};
```

The `fileSystem` property points at the persistent storage on the device. Most operations begin by asking for a filesystem, and we can speed it up by saving it the first time we ask. Then other operations can use our cached value.

```
self.failure = failure;
```

Just like `completion`, this is the `failure` function. `dispatchFailure()` will be called first, which then calls this one, if it is non-null.

```
self.fileEntry = {};
```

For some of our operations, we have to store the information about a specific file; we do that using the `fileEntry` property.

```
self.loadFileSystem = function ( completion, failure )
{
    self.completion = completion;
    self.failure = failure;
    self.getFileSystem();
}
```

The `loadFileSystem()` function can be called by the app at any time, but it is usually called when the app suspects that the documents available to us have changed. Say there may be a new one out there, and we want to be sure to display it to the user. Most of the operations in this class will try to re-read the directory after an operation (like renaming a file), but not every operation supports this, and this doesn't stop documents from appearing that we didn't explicitly create (say, from an iTunes import).

```
self.getFileSystem = function()
{
    self.state = "Requesting File System";
    window.requestFileSystem ( LocalFileSystem.PERSISTENT,
        0, self.getFileSystemSuccess, self.dispatchFailure );
}
```

The `getFilesystem()` function does the first thing we have to do when asking to see what files we have available to us: requests the filesystem. In this case, we're asking for the persistent filesystem so that the data is stored permanently.

```
self.dispatchFailure = function ( e )
{
    console.log ("While " + self.State + ", encountered
error: " + JSON.stringify(e));
    if (self.failure)
    {
        self.failure ( e );
    }
}
```

Generally, I like to keep success/failure methods close to the invoking method, but failures can be handled pretty generically (in our case), and so I just have one failure function that all our operations can call. It records a nice log message for us, and then checks to see if the app has registered a failure callback, and if it has, we'll call it too.

```
self.getFileSystemSuccess = function ( fileSystem )
{
    self.state = "Received File System";
    self.fileSystem = fileSystem;
    self.getDocuments ( fileSystem.root );
}
```

When we're in the preceding function, we've got a valid filesystem. We save it for later use, and then we also call `getDocuments()` to start the process of getting every document our app can access.

```
self.getDocuments = function ( directoryEntry )
{
    self.state = "Requesting Reader";
    var directoryReader = directoryEntry.createReader();

    self.state = "Requesting Entries from Reader";
    directoryReader.readEntries (
        self.getDocumentsSuccess, self.dispatchFailure );
}
```

In order to go over every entry in the directory of the filesystem we've got, we have to create a directory reader. We can do this by using the `directoryEntry` function passed to us (which is pointing to the filesystem we requested). Once we have that, we ask it to read all the entries and call `getDocumentsSuccess()` when it is finished.

```
self.getDocumentsSuccess = function ( entries )
{
    var theDocuments = [];
    for (var i=0; i<entries.length; i++)
    {
        // is the entry a file? (we won't iterate subdirs)
        if (entries[i].isFile)
        {
            var theFileName = entries[i].name;
            var theFileType =
                theFileName.substr(theFileName.length-4,4);
            if (theFileType === ".fln")
            {
```

```
        // a file we know we can process
        theDocuments.push ( entries[i] );
    }
}
}
self.documents = theDocuments;
self.state = "";
if (self.completion)
{
    self.completion ( self );
}
}
```

In the preceding function, we read through all the entries given to us. One should never assume that all the entries in a directory are something that our app can handle, so we screen for subdirectories (which we won't be creating, so it won't be anything we can deal with), and then we also check for the file extension. If it is .fln, we assume the file is one of ours and add it to the list. If it has anything else, we ignore it.

Once we're done iterating over the list, we call the `completion` method (if it exists) so that the app can do what it wants with the list.

```
self.getDocumentCount = function ()
{
    return self.documents.length;
}

self.getDocumentAtIndex = function ( idx )
{
    return self.documents[ idx ];
}
```

The prior two methods are pretty self-explanatory. The first returns the number of documents we were able to get from the directory, and the second returns the information obtained for a specific document.

```
self.deleteDocumentAtIndex = function ( idx, completion,
failure )
{
    self.completion = completion;
    self.failure = failure;
    self.state = "Removing a Document";
    self.documents [ idx ].remove (
        self.deleteDocumentAtIndexSuccess,
        self.dispatchFailure);
}
```

This method isn't playing games; it'll physically remove the document at the specified index. Our app will ask the user first if they'd like to remove the document, so that it can't be accidentally called, but this function won't ask anyone if it is okay on its own. So be careful when calling it.

```
self.deleteDocumentAtIndexSuccess = function ()
{
    self.state = "";
    self.getFileSystem();
}
```

After a successful delete, we need to re-read the filesystem so that our documents array is up-to-date. We do this by calling `getFileSystem()`. You may wonder how the `completion` method defined in `deleteDocumentAtIndex` gets called, though. It gets called at the end of `getFileSystem()`. It checks to see if the `completion` property has been set (which we do at the beginning of `deleteDocumentAtIndex`), and if it has, it calls it. This is a pattern a lot of our operations will follow.

```
self.renameDocumentAtIndexTo = function ( idx, newName,
                                         completion, failure )
{
    self.completion = completion;
    self.failure = failure;
    self.state = "Renaming a Document";
    self.documents [ idx ].moveTo (
        self.fileSystem.root, newName,
        self.renameDocumentAtIndexToSuccess,
        self.dispatchFailure );
}

self.renameDocumentAtIndexToSuccess = function ()
{
    self.state = "";
    self.getFileSystem();
}
```

Renaming a document is simply a `moveTo` operation to the same directory. It follows the same pattern of operation as the preceding `delete` operation. Note that there is no check here for the new name of the file for if it isn't already being used by an existing file. If there is a name conflict, the new file will overwrite the old file, not likely something you want to occur. Since the preceding `deletion` method doesn't ask, we won't ask here either, but it is something you should do in the app itself.

```
self.copyDocumentAtIndexTo = function ( idx, newName,
                                         completion, failure )
```

```
{  
    self.completion = completion;  
    self.failure = failure;  
    self.state = "Duplicating a Document";  
    self.documents [ idx ].copyTo ( self.fileSystem.root,  
        newName, self.copyDocumentAtIndexToSuccess,  
        self.dispatchFailure );  
}  
self.copyDocumentAtIndexToSuccess = function ()  
{  
    self.state = "";  
    self.getFileSystem();  
}
```

Copying is again remarkably similar to renaming, as seen in the prior code; the difference is that we use `copyTo` instead of `moveTo`. The operation is also a bit different; if you were to try to copy over an existing document, the attempt fails, unlike moving over an existing document.

```
self.createDocument = function ( theDocumentName,  
    completion, failure )  
{  
    self.completion = completion;  
    self.failure = failure;  
    self.state = "Creating a Document";  
    self.fileSystem.root.getFile ( theDocumentName,  
        {create: true, exclusive: false},  
        function ( theFileEntry )  
        {  
            self.fileEntry = theFileEntry;  
            self.state = "";  
            self.getFileSystem();  
        }, self.dispatchFailure );  
}
```

The `createDocument()` method creates a new file in the directory and after it does so, it re-reads the filesystem. This demonstrates an alternative to using `...Success()` methods. It works just the same, though. Just like renaming, this can be dangerous if a file with the same name already exists, so be sure to check before calling this method.

```
self.openDocumentAtIndex = function ( idx, completion,  
    failure )  
{  
    self.completion = completion;  
    self.failure = failure;
```

```
self.state = "Opening a Document";
self.fileSystem.root.getFile (
    self.documents[idx].name, {create: false,
        exclusive: false},
    function ( theFileEntry )
    {
        self.fileEntry = theFileEntry;
        self.state = "";
        self.getFileSystem();
    }, self.dispatchFailure );
}
```

As seen in the prior code, opening a document is very similar to creating a document, except we don't ask the filesystem to create it if it doesn't exist.

```
self.getFileEntry = function ()
{
    return self.fileEntry;
}
```

Some operations, such as creating and opening a document, also set the `fileEntry` property to the newly opened document. This is handy for use when asking a note to open itself. It can read in contents of the file in this property.

```
self.getFileSystem ();
}
```

As we mentioned prior to looking at the code for our model, we said that we'd initiate a directory read upon creation, and this is what we're doing at the end of the model. That way, when we create an object, it will instantly go to work reading the entries in the directory.

Now, let's look at the code for a single document:

```
var DOC = DOC || {};

DOC.Filer = function ( theFileEntry, completion, failure )
{
    var self = this;

    // file and state
    self.fileEntry = theFileEntry;
    self.fileName = self.fileEntry.name;
    self.completion = completion;
    self.failure = failure;
    self.state = "";
```

```
// file-specific
self.title = "My Filer";
self.text = "";

self.getTitle = function ()
{
    return self.title;
}

self.setTitle = function ( theTitle )
{
    self.title = theTitle;
}

self.getText = function ()

{
    return self.text;
}

self.setText = function ( theText )
{
    self.text = theText;
}
```

The preceding code should be self-explanatory now. Next up, in the following code, we see how to read the contents of a file:

```
self.readFileContents = function()
{
    self.state = "Reading a File";
    self.fileEntry.file ( self.gotFile,
        self.dispatchFailure );
}

self.dispatchFailure = function( e )
{
    console.log ("While " + self.State + ", encountered
        error: " + e.target.error.code);
    if (self.failure)
    {
        self.failure ( e );
    }
}
```

When requesting to read a file, we have to call the `file()` method of the file's corresponding `fileEntry`. If it finds the file, it'll call `gotFile()`, but if it can't read it, for some reason, it'll call `dispatchFailure()`.

```
self.gotFile = function ( theFile )
{
    var reader = new FileReader ();
    reader.onloadend = self.finishedReadingFile;
    reader.onerror = self.dispatchFailure;
    reader.readAsText ( theFile );
}
```

Once we've got the file, we have to create a `FileReader` variable for it. Unlike other API calls, we have to set up some event handlers, but they mean the same thing here as completion and failure. Then we ask the reader to read the file.

```
self.finishedReadingFile = function ( e )
{
    var theFileContents = e.target.result;
```

Once we are here, `e.target.result` has the contents of the entire file. Now we can try to load it in.

```
if (!theFileContents)
{
    theFileContents = '{"title":"New
File", "text":""}';
}
```

If there's nothing in the file, we set up some reasonable defaults. Notice that we're using JSON here. This is because we'll be storing our file in the JSON file format.

Next, we try to parse the contents of the file as JSON. This is where the `try/catch` block comes in. If we can't parse the contents of the file, we'll get an error and we can call the `failure` function. But if we do parse it correctly, we can set our own `title` and `text` to the file's `title` and `text`, and we'll have successfully loaded the file's contents.

```
self.writeFileContents = function ( completion , failure )
{
    self.completion = completion;
    self.failure = failure;
    self.fileEntry.createWriter ( self.getWriter,
        self.dispatchFailure );
}
```

```
self.gotFileWriter = function ( writer )
{
    writer.onerror = self.failure;
    writer.onwriteend = function ( e )
    {
        if (self.completion)
        {
            self.completion();
        }
    };
    writer.write ( JSON.stringify ( self.serialize() ) );
}
```

Saving a file isn't terribly different than loading a file, except that we can create the file writer directly from the `fileEntry` property rather than calling `file()` first. In `gotFileWriter`, though, we have to set similar events before calling `write()` with the file contents. We `stringify` the results of `serialize()` so that it is in a proper JSON format.

```
self.serialize = function ()
{
    return { "title": self.title, "text": self.text };
}
```

Speaking of serialization, here's the method that does it. Not hard, but you may be asking why we didn't just `stringify` `self`. And that's a great question. Turns out you can't `stringify` objects that contain methods, because it will lose those methods; so that's one reason. Another reason is that we really don't need to save the entire object, just the title and the text; so instead of saving a lot of stuff we don't need, we'll just return an object that has exactly what we need.

```
    self.readFileContents () ;
}
```

Like our first model, we ask the document here to load its file contents immediately upon creation.

What did we do?

In this section, we created two data models, one for the list of available documents in a directory, and the second for the actual note itself.

What else do I need to know?

The File API is notoriously hard to get used to, especially for programmers who assume that the order of processing is always the next statement after this one. The File API, however, does things differently, by requiring each operation to have a success and failure callback. Furthermore, there are several operations when reading or saving a file (or when reading a directory), and as such the callback chain can start to get pretty confusing. This is generally why I try to make separate functions instead of inline callbacks, but there are times when inline callbacks make the most sense.

The File API can do more than what we've covered here, so you'd do well to go take a look at it at http://docs.phonegap.com/en/edge/cordova_file_file.md.html#File. Just remember how the callbacks work and you'll be fine, even if your code will feel a bit spaghetti-ish.

Implementing documents view

The documents view will be used to display the list of available documents to the end user. It will also permit the user to create a document, rename a document, copy a document, and delete a document.

Let's take a quick look at the finished product, on iOS first:



For Android, the view will be as follows:



Notice that the screenshot for the iPhone has a totally different look and feel than the Android screenshot. While many apps for the iPhone use the alternative method used for Android, the method of scrolling horizontally through large document representations is more common on the iPhone, and is what we use here. Thankfully, it only takes a small code change and some CSS to render the two disparate looks. Otherwise, they function identically.

Getting on with it

As always, we're going to start with the HTML portion of the view. The boilerplate portion is virtually identical to our previous apps, so we'll start with the template instead:

```
<div id="documentsView_documentTemplate" class="hidden">
    <div class="documentContainer">
        <div class="documentImage">
            
        </div>
        <div class="documentTitle"
            onclick="documentsView.renameDocument (%INDEX%) ">
            <span>%TITLE%</span>
        </div>
```

```

<div class="documentActions">
    
    
    
</div>
</div>
</div>
```

This template defines the HTML for each document we display. It's not terribly complicated. Note that we have `onClick` handlers for each portion of the template that can respond to touch, but beyond that the style is controlled in `style.css`.

Let's take a look at the code that powers this view:

```

var documentsView = $ge("documentsView") || {};
documentsView.lastScrollLeft = 0;
documentsView.myScroll = {};
documentsView.availableDocuments = {};
```

First up, our properties. `lastScrollLeft` is for maintaining our scroll position when we switch between views. `myScroll` will hold our scroller (for iOS and Android), and `availableDocuments` will hold all the documents the filesystem has for our app.

The `initializeView()` method is so similar to our previous projects (refer to the *Implementing the start view* section of *Project 1, Let's Get Local!*), I'll go ahead and skip it and jump to `displayAvailableDocuments()` (which the `initializeView()` method does call).

```

documentsView.displayAvailableDocuments = function ()
{
    documentsView.availableDocuments = new DOCS.Filers (
        documentsView.documentIterator
    ,
        function () // failure function
    {
        var anAlert = new PKUI.MESSAGE.Alert
            (__T("Oops!"),

```

```
    __T("I couldn't read your persistent
        storage!"));
    anAlert.show();
}
);
}

}
```

First, we create a new `DOCS.Filers` object. Remember that this will immediately send a request to the filesystem for all the files it contains that we can use. When it successfully completes that request, it will call `documentsView.documentIterator()`, a method that will go over each item in the list and render the preceding template. If it fails, however, it calls the `failure` function defined earlier and displays an alert message.

This is big; we're no longer using the in-built `alert()` method! Instead, we're creating a new `Alert` object with the title of `Oops!` and `I couldn't read your persistent storage!` Granted, not the best error message in the world, but if this does occur, we're essentially toast anyway. The bigger issue is that this object, which we'll cover in more detail as we progress through this task, provides us with platform-specific non-native alerts. This means we can customize them to our needs; in this case that isn't much, we're displaying an error message, but the `PKUI.MESSAGE` namespace provides options for prompts as well. The `Alert` object also gives us the ability to specify a callback when a button is pressed, very useful if we need to ask a Yes/No question.

The next method, `reloadAvailableDocuments()`, is so similar to the earlier method that I'll also skip it. It's only used when the file view is being popped off the view stack.

```
documentsView.documentIterator = function ( o )
{
    var theHTML = "";
    var theNumberOfDocuments = 0;
    for (var i=0; i<o.getDocumentCount(); i++)
    {
        var theDocumentEntry = o.getDocumentAtIndex ( i );

        theHTML += PKUTIL.instanceOfTemplate (
            $ge("documentsView_documentTemplate"),
            { "title":
                theDocumentEntry.name.substr(0,
                theDocumentEntry.name.length-4),
            "index": i
            }
        );
        theNumberOfDocuments++;
    }
}
```

```

if (PKDEVICE.platform() == "ios")
{
    $ge("documentsView_contentArea").style.width =
        ((theNumberOfDocuments) * 246)) + "px";
}

$ge("documentsView_contentArea").innerHTML = theHTML;
}

```

This is a pretty simple function: all we do is iterate through the documents that are returned from the filesystem and create a new instance of the `documentsView_documentTemplate` template. We're using a new convenience method called `PKUTIL.instanceOfTemplate()` to make this easier. It will take a DOM element and an object containing the properties that should be replaced, in this case, `title` and `index`, along with their corresponding values. (The `substr()` method is used to chop off the file extension.)

This method is doing the same thing we were doing manually before, using `replace()`, but it does it better. If you hadn't noticed, we were cleverly avoiding using the same substitution variable in our templates more than once. This is because `replace()` only replaces one instance at a time. Our convenience method keeps calling `replace()` until all instances are replaced, which means we can now use `%TITLE%` and `%INDEX%` all we want.

The portion of code specific to iOS simply determines the width of the content area for scrolling purposes. For Android, this code isn't executed.

```

documentsView.openDocument = function ( idx )
{
    documentsView.availableDocuments.openDocumentAtIndex
        ( idx,
            function ()
            {
                fileView.setFileEntry ( documentsView.
                    availableDocuments.getFileEntry() );
                PKUI.CORE.pushView ( fileView );
            },
            function ( e ) { console.log ( JSON.stringify(e) )
                }
        );
}

```

Opening a document occurs when a user taps on the document's icon. (For Android, this is something you should think about changing, but for the purposes of this app, we'll keep consistent).

We call `openDocumentAtIndex()` and pass along the completion and failure functions that are called when the document is opened. The `success` method will set the `fileEntry` property of the `fileView` method and then push it on to the screen. This act will trigger loading the contents as well. Failure will log the error to the console, though you probably should add a meaningful error alert as well.

```
documentsView.createNewDocument = function ()
{
    {
        var anAlert = new PKUI.MESSAGE.Prompt
            (__T("Create Document"),
             __T("This will create a new document
                  with the name below:"),
             "text",
             "New Filer " + __D(new Date(),
                  "yyyy-MM-dd-HH-mm-ss"),
             __T("Don't Create<|Create>"),
             function (i)
            {
                if (i==1)
                {
                    documentsView.availableDocuments.
                    createDocument ( "" +
                        anAlert.inputElement.value+".fln",
                        function ()
                    {
                        fileView.setFileEntry (
                            documentsView.
                            availableDocuments.
                            getFileEntry() );
                        PKUI.CORE.pushView ( fileView );
                    },
                    function (e)
                    {
                        var anAlert = new
                            PKUI.MESSAGE.Alert (
                                __T("Oops!"),
                                __T("Couldn't create the
                                      file.") );
                        anAlert.show();
                    }
                );
            }
        );
        anAlert.show();
    }
}
```

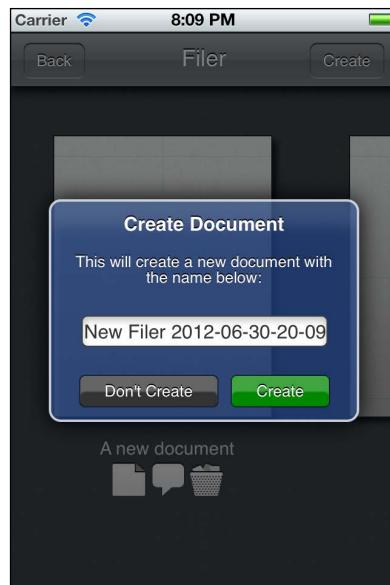
Welcome to the wonderful world of chaining callbacks! We have two steps to create a document. First, we ask the user what they want to name the document (using an admittedly obtuse default). Then we create the document, which means we have to have another success/failure callback. If we fail to create the document, we create another alert to further confuse matters.

The big deal, though, is that our initial request of the user is actually giving them a chance to type something in to our alert message! We've not done this yet in any of our apps, and this is monumental. Furthermore, we have custom buttons—a `Don't Create` button and a `Create` button.

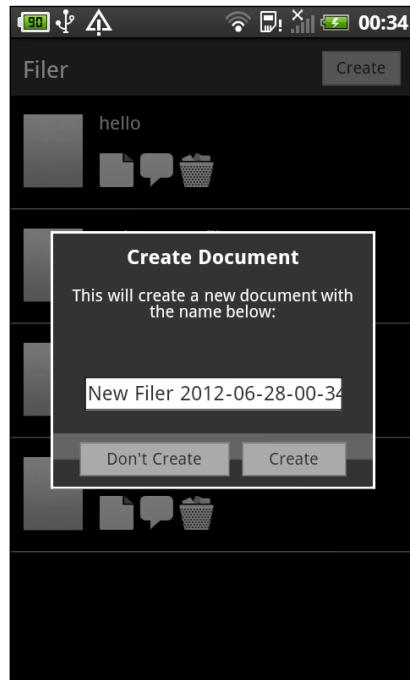
If you're wondering what the `<` and `>` are in the preceding code attached to the buttons – great catch! These are used primarily for iOS, though you could extend these to the other platforms as well. iOS has the concept of a destructive action; these buttons should always be colored red. (Or, if the locale that you are targeting uses a different color, use that color instead.) It also has the concept of **Cancel** button colors (typically a darker gray). To add to it, we decided to color buttons that would go to the next step in the process green.

Each of these gets a special character at the end of the button's name. For example, `Cancel<` would color the button a darker color and use the text of `Cancel` for the button. `Go>` would use `Go` as the text, and color the button green. `Delete*`, on the other hand, would use `Delete` as the text, but color the button red.

Just so you have a good idea of what an alert/prompt will look like on each system, here's an example for iOS:



For Android, the view will be as follows:



Renaming a document is somewhat similar to creating a new document, except that we won't display the document at the end. We will ask the user what the new name should be, and if they choose to continue, we'll try to perform the function. The following code snippet can be used for this action:

```
documentsView.renameDocument = function ( idx )
{
    var theFileName = documentsView.availableDocuments.
        getDocumentAtIndex(idx).name;
    theFileName = theFileName.substr(0,theFileName.length-4);

    var anAlert = new PKUI.MESSAGE.Prompt (
        __T("Rename Document"),
        __T("Rename your document to the
              following:"),
        "text",
        theFileName,
        __T("Cancel<|Rename>"),
        function (i)
```

```
{  
    if (i==1)  
    {  
        var theNewFileName =  
        ""+anAlertInputElement.value+".fln";  
        try {  
            documentsView.availableDocuments.  
            renameDocumentAtIndexTo  
            ( idx, theNewFileName,  
            documentsView.documentIterator,  
            function ( e )  
            {  
                var anAlert = new  
                    PKUI.MESSAGE.Alert  
                    ( __T("Oops!"),  
                    __T("Couldn't rename the  
                    file.") );  
                anAlert.show();  
            }  
        );  
    }  
    catch (e)  
    {  
        var anotherAlert = new  
            PKUI.MESSAGE.Alert ( __T("Oops!"),  
            __T("Couldn't rename the  
            file.") );  
        anotherAlert.show();  
    }  
};  
anAlert.show();  
}
```

If there is a failure of some sort, we'll indicate this by displaying an error, once as a failure function, and second in the `catch` portion of the `try/catch` block.

Note that, as written, we make no check here to see if the new name would conflict with another file. Therefore, if the user renamed one file to the name of another, the previous file would be overwritten. You should add an additional check in your code to make sure that the new file name doesn't already exist.

The `copyDocument()` method is nearly identical, so we'll skip it and move on to the `deleteDocument()` method shown in the following code snippet:

```
documentsView.deleteDocument = function ( idx )
{
    var anAlert = new PKUI.MESSAGE.Confirm (
        __T("Remove Document"),
        __T("This will remove the document. This
            action is unrecoverable."),
        __T("Don't Remove<|Remove*"),
        function (i)
        {
            if (i==1)
            {
                documentsView.availableDocuments.
                    deleteDocumentAtIndex
                    ( idx, documentsView.documentIterator,
                    function (e)
                    {
                        var anAlert = new PKUI.MESSAGE.Alert
                        (__T("Oops!"),
                        __T("Couldn't delete the file."));
                        anAlert.show();
                    }
                );
            }
        }
    );
    anAlert.show();
}
```

Deleting a document is much more simple than copying or renaming one, so we're not delving as deep in a callback chain here. The primary thing I wanted to point out was the use of the * to indicate that the `Remove` button would display as a red button to warn the user that it was a destructive action on iOS. Android silently ignores this flag, though you could modify the framework to display similar colors as well on Android.

The remaining methods are similar to those in the previous views, so we'll go ahead and skip them.

What did we do?

Though there is a lot of room to improve, we've created a pretty good document manager for our app. We've permitted the user to rename their files, delete them, copy them, open them, and create them, all things a good file manager should do. The only thing we didn't do was permit the user to share them, though the intent is there with the pleasant **Share** icon. This was only in the interest of space, and because it is a subject that we've covered before.

What else do I need to know?

There are plenty of things we *haven't* covered in this file manager of ours, and they're big ones and definitely things you need to think about implementing on your own. The code itself would be self-explanatory, so we won't go into great detail, but here are the primary issues:

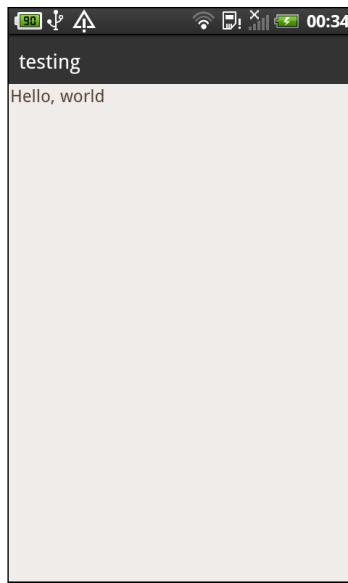
- ▶ Filenames can't contain certain characters: Everyone handles this somewhat differently; you could display an error to the user indicating that they need to pick different characters or you could silently change them to something else (typical for iOS). Either way, you should check for them prior to creating a new file or renaming/copying a document.
- ▶ Create/Rename operations can overwrite existing data: You'd think that since a copy operation will fail if the destination file exists, that rename/create would also. Unfortunately, no. They'll just overwrite the file. You must iterate over the entire directory structure in order to determine if you're about to overwrite an existing file! Users don't like losing data, even if they themselves were the cause.
- ▶ Opening a document for non-iOS users: Android users shouldn't need to know that tapping the icon will open the document; they'll assume the entire area is tappable (minus the icons). So it would be a good idea to give them another icon for renaming a file and allow the filename itself (as well as the document icon) to open the file instead of renaming the file.
- ▶ iOS document images should reflect the contents of the file: This one's harder to implement, granted, but typically the document icon would contain some portion of the actual contents of the file. There are various ways of doing this, from reading the actual contents in and displaying them over the DOM (and clipping them after some portion) to rendering them to an HTML canvas tag and saving the result as a thumbnail. Either way, it's something the user will expect.

Implementing the file view

This view is pretty easy; actually it's essentially a big TEXTAREA element with some code to automatically save the contents every few seconds. Let's take a look at how it will render on each platform, first for iOS:



For Android, the view will be as follows:



All of these look pretty similar and reflect the simplicity of the view.

Getting on with it

The HTML portion of the view is much like prior views, so we'll skip that for now. Just know that there is a TEXTAREA element named `fileView_text` that our code will reference. There is also an `onClick` handler on the title bar to enable changing the title of the note. These are seen in the following code snippet:

```
var fileView = $ge("fileView") || {};  
  
fileView.theFileEntry = {};  
fileView.theFilerDocument = {};  
fileView.theSaveTimer = -1;
```

As always, we have several properties. The first is intended to store information about the file we're currently working on, while the second is the actual document contents. The last property will store a value returned by `setInterval()`, this is used to call our auto-save functionality every few seconds.

```
fileView.setFileEntry = function ( theNewFileEntry )  
{  
    fileView.theFileEntry = theNewFileEntry;  
    fileView.theFilerDocument = {};  
}
```

Here we just provide a way for the `documentView` method to tell us which file to work with.

The next method, `initializeView()`, is similar enough to the other views that we'll skip over it. Next up is `entitleDocument()` shown in the following code snippet:

```
fileView.entitleDocument = function ()  
{  
    var anAlert = new PKUI.MESSAGE.Prompt (  
        __T("Entitle"),  
        __T("What's the title of this  
document?"),  
        "text",  
        fileView.theFilerDocument.getTitle(),  
        __T("Cancel<|Entitle>"),  
        function ( i )  
        {  
            if ( i==1 )  
            {  
                fileView.theFilerDocument.setTitle (
```

```
        anAlert.inputElement.value ) ;
        fileView.viewTitle.innerHTML =
        fileView.theFilerDocument.getTitle() ;
    }
}
);
anAlert.show() ;
}
```

When the title is tapped, we'll display a prompt to the user that enables them to change the title of the note.

```
fileView.loadDocument = function ()
{
    fileView.viewTitle = $ge("fileView_title");
    fileView.viewTitle.innerHTML =
    fileView.theFileEntry.name.substr(0,
    fileView.theFileEntry.name.length-4);
    fileView.theTextElement = $ge("fileView_text");
    fileView.theTextElement.value = "";

    fileView.theFilerDocument = new DOC.Filer (
    fileView.theFileEntry,
    function ()
    {
        fileView.viewTitle.innerHTML =
        fileView.theFilerDocument.getTitle();
        fileView.theTextElement.value =
        fileView.theFilerDocument.getText();
        fileView.theSaveTimer =
        setInterval (
        fileView.saveDocument, 5000 );
    },
    function (e)
    {
        PKUI.CORE.popView();
        var anAlert = new PKUI.MESSAGE.Alert
        (__T("Oops!"),
        __T("Couldn't open the file."));
        anAlert.show();
    }
);
}
```

Loading the contents of a specific document is accomplished by creating a new `DOC.Filer()` object using the contents of our `fileEntry` property. This is assumed to have been set by `documentView` prior to pushing us onto the view stack.

Upon successfully parsing the document, we set the navigation bar's title to the document's title, and the `TEXTAREA` element's contents to the note's text. Then we set up the auto-save at an interval of five seconds.

If, for some reason, we can't open the file, we'll display an error, but we'll also pop ourselves off the view stack. No sense in displaying an editor if we can't even open the file.

```
fileView.saveDocument = function ()
{
    fileView.theFilerDocument.setText (
        fileView.theTextElement.value );
    fileView.theFilerDocument.saveFileContents (
        function ()
    {
        console.log ("Auto save successful.");
    },
    function (e)
    {
        PKUI.CORE.popView();
        var anAlert = new PKUI.MESSAGE.Alert (
            __T("Oops!"),
            __T("Couldn't save to the file.") );
        anAlert.show();
    }
);
}
```

Saving the contents is a simple affair. We copy the text from the `TEXTAREA` element and put in the `Filer` object. Then we ask it to save the contents of the file. If it is successful, we just log a message to the console (something you'd remove in a production app), and if it isn't, we display an error and pop the view. (Whether popping the view is a good idea or not is debatable.)

```
fileView.viewWillAppear = function ()
{
    fileView.loadDocument();
}
```

Our `viewWillAppear()` method is pretty simple: we kick off a load of our note. This means that by setting `fileEntry` and pushing us on the view stack, we'll automatically load the contents of the note.

```
fileView.viewWillHide = function ()
{
    if (fileView.theSaveTimer!==-1)
    {
        clearInterval (fileView.theSaveTimer);
        fileView.theSaveTimer = -1;
    }
    fileView.saveDocument();
    documentsView.reloadAvailableDocuments();
}
```

Our `viewWillHide()` method is a little more complex. Here we disable our auto-save. After all, we don't want to be saving a document that is no longer open. Then we force-save the document. Perhaps the user is navigating back in between an auto-save interval; they wouldn't want to lose any data, right?

After we save the contents, we also force the `documentsView` method to reload the list of documents. This isn't a big deal when we're-editing existing documents, but it is a big deal when we're creating new documents, as we want the file manager to be able to display our newly created note.

What did we do?

We created a simple text editor view that can open file contents and save them back again. It implements a simple auto-save function as well.

What else do I need to know?

Most of the devices that you will be targeting use soft keyboards for input. This means that some portion of the screen will be covered by the onscreen keyboard.

How each device does this differs by platform and type of keyboard. For example, Android permits many different keyboards to be installed, and not every keyboard does things the same way.

Essentially, though, what happens is the available real-estate is moved or resized to permit the onscreen keyboard. This means our user interface also moves along with the keyboard. Whether or not this is done fluidly depends on the platform (and on Android, the keyboard itself, to some extent). iOS does this the best; there's a minimum of fuss involved, and the display scrolls neatly to ensure the text remains on the screen.

Android has a tendency to flicker a bit while they do this, unfortunately, and there's very little we can do to control how the keyboard itself appears.

One interesting option would be to implement the soft keyboard ourselves in pure HTML, CSS, and JavaScript. Technically this can work, but it remains a pretty large hack, and your soft keyboard won't really ever act like the legitimate keyboard on the platform. (And on Android, fans of a particular keyboard configuration will instantly hate it.) You'd also have to take into account the case when a user has connected a Bluetooth keyboard. This typically prevents the soft keyboard from appearing, which means the full real-estate of the screen is used for our display. Since there's no way (short of developing our own plugin) to determine if a hard keyboard is attached, we highly advise against using this option.

Some Android distributions also add an interesting quirk. It appears that `input` and `textarea` elements actually display another editable region above themselves when being edited. On my phone, this was visible by having a portion of the flashing cursor visible just underneath the current editor, almost as if the DOM element was just a mirror of a native input element. It wouldn't have been noticeable except for the fact that they were slightly misaligned. Odd, anyway, and I thought I would mention it.

Game Over..... Wrapping it up

We've accomplished quite a lot in this task and while none of it is particularly glorious, it is absolutely necessary for what's ahead. Our apps must be able to store data permanently and they must also be able to retrieve that same data. Likewise, they need to provide methods for managing that data, including renaming, duplicating, and deleting it.

Can you take the HEAT? The Hotshot Challenge

There are several ways you can improve upon this app:

- ▶ Our app only asks the user if they really want to delete a file, but other operations are equally dangerous. Add in confirmations if the action the user is about to perform would overwrite data (for example, renaming a document to an existing document's name or creating a document with the same name as an existing document).
- ▶ Add functionality to check if the filename a user is supplying is valid. Then, either indicate this to the user or silently change the invalid characters to valid characters.

- ▶ We don't provide subdirectory functionality, but there's no reason why you couldn't. In fact, we explicitly ignore subdirectories in our code, as they add a lot of complexity to the file management system. Why don't you add subdirectory management to the app?
- ▶ Instead of storing notes, perhaps you could store some forms instead. Perhaps simple addresses or reminders—really, just about anything.

Project 4

Let's Take a Trip

Geolocation has become very important in today's world, especially since most phones have the capability to determine your position to an astonishing degree of accuracy. Given that this used to (not so long ago) be the mainstay of expensive GPS gadgets, it is amazing how quickly this ability has become ubiquitous. Because of this, users expect location-aware applications, and not just that, but they expect a nice map that responds to their input right along with it.

What do we build?

Our project is aimed at two concepts, the first being to simply use a (very) small portion of the Google Maps API (as of this writing, 3.9). We will use this API to display a fully functional map that is centered on the user's current location. Secondly, we will use the geolocation features that PhoneGap provides in order to obtain the user's current location. When we're done, we'll have an app that can not only display a map centered around the user's current location, but we'll also have an app that can record their movements and show them on-screen. Instead of a voice recorder, think of it as a location recorder.

What does it do?

We introduced document management in the last project, and we'll further shore up those capabilities in this project. Thankfully, most of the work is done for us, but as you recall, there were a few situations where the user could get into trouble (if they used a name that collided with an existing file). We'll take care of that in this project so that we have a much more robust solution.

We'll also be going over the various methods that we can use to load in the Google Maps APIs, something that turns out to be a little more difficult than one would initially think, and this is complicated by the fact that we have to deal with the possibility of losing (or not having) network access. We won't use the entire Google Maps API, it deserves a book all its own, but we will use some basic features, including markers and lines.

Finally, we'll be using geolocation. Some browsers are kind enough to provide a good implementation of geolocation, and since PhoneGap's implementation follows the W3C standard, it will use the browser solution if it is good enough. This also means that what we're building could even be made to work outside of PhoneGap if one switched to using `localStorage` instead of persistent files.

When we've put it all together, we'll have a pretty interesting app. An app that records our location (while we let it) and that can then display it. This opens up all sorts of possibilities for extension: you could share a path with a friend, you could export to KML for ingestion by other applications, and more.

Why is it great?

Geolocation and interactive maps are something users expect in today's apps. If you display an address, you should at least be able to display a map to go along with it. If you offer searches by location, you should be able to locate where the user is and provide them with relevant results. Geolocation and interactive maps aren't just for turn-by-turn location or helping someone who's become lost; they are invaluable in a lot of other applications as well.

How are we going to do it?

In many ways, this task is easier than those that have preceded it. For one, our framework has become pretty stable (though there are some changes for this project), and we have a good start on document management. What's really left is creating the data model that is capable of storing location information and saving and retrieving it. This is also our first app where we'll dispense with the start view – we'll jump straight into the app from now on.

In order to accomplish this, we'll use the same familiar steps we've used before:

- ▶ Designing our UI and the look and feel
- ▶ Designing our data model
- ▶ Implementing our data model
- ▶ Changing our document manager
- ▶ Implementing our map view

What do I need to get started?

As always, go ahead and create your project. Be sure to follow the same project steps as in previous projects, though you don't need to worry about plugin support; we don't need any native plugins for this app.

Also, take a look at the Google Maps API (<https://developers.google.com/maps/documentation/javascript/3.9/reference>). While there, you might want to sign up for an API key as well. Although you can use the API without one (and we do here), having a key allows for usage metrics, and should your app become popular enough, the ability to pay for your use so that you aren't restricted to the low API caps enforced for non-key users. Be sure to take a look at their documentation as well; there's a lot of it there, enough to fill several chapters alone, but it is well worth perusing. Put it this way: there are things Google Maps does that I never knew it could do, and you might discover the same.

Designing our UI and the look and feel

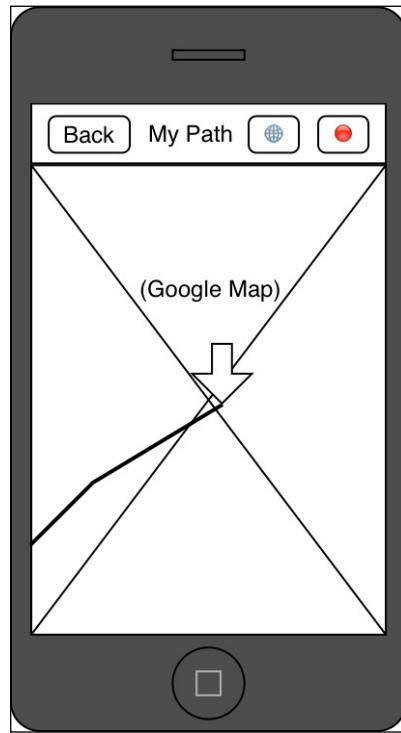
As always, let's not start coding until we have a good idea about how we want our app to look. Thankfully, we really only have to focus on the map view. We've already covered the look and feel of our document manager in the previous project, and it's not changed much here. Furthermore, since a good portion of the view will actually be taken up by the interactive map provided by Google, there's not even a lot we have to do there.

Getting ready

Once again, get your pencil and paper out or your favorite graphics editor; we'll be using them to design our wireframe and then build out any of the assets we might need later on.

Getting on with it

The following screenshot gives a final mockup of our map view:



At first glance, this appears like a pretty simple view—and it is—but don't let that fool you. There's a lot of power underneath!

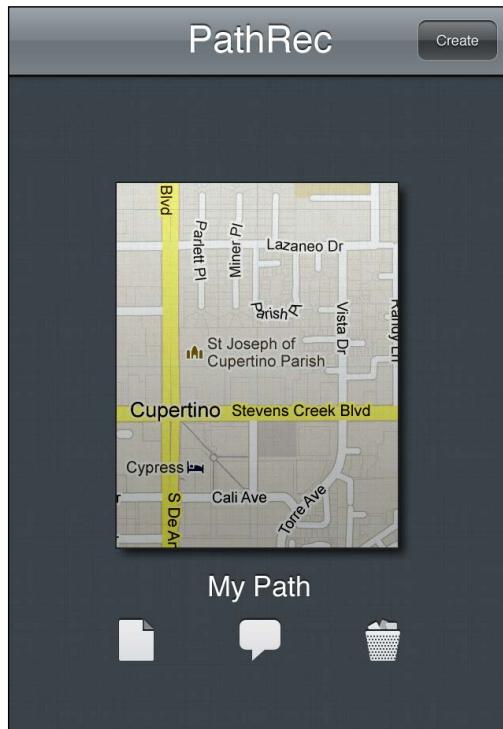
Let's go over the various items:

- ▶ The **Back** button will take the user back to the document management interface. For Android, of course, this button will not be present; the device's physical back button will do the trick.
- ▶ The title on the navigation bar will be the title of the document. The user can tap on it to change the document via a prompt alert from our framework.
- ▶ The button with the *globe* is intended to be the *find me* button. The view will automatically do this when shown, but one of the hallmarks of an interactive map is that you should be able to explore it on your own without constantly being dragged back to your current position. This button is intended to re-center your map after you've done some exploration.

- ▶ The button with the *red dot* is the *record* button. When tapped the view will begin to record your position (as often as it changes) and draw a line following your progress. If tapped again, it will stop tracking your location.
- ▶ Below the navigation bar is the Google Map (here, an X image). This will, of course, be populated from Google. Moving the map with your finger will, of course, pan the map, but more importantly, we will catch this event so that we can *unlock* the map from your current location.
- ▶ The downward pointing arrow is a representation of the typical Google Marker; this will indicate your current position.
- ▶ The line is a representation of some path already recorded; it will indicate every update in your path during the recording.

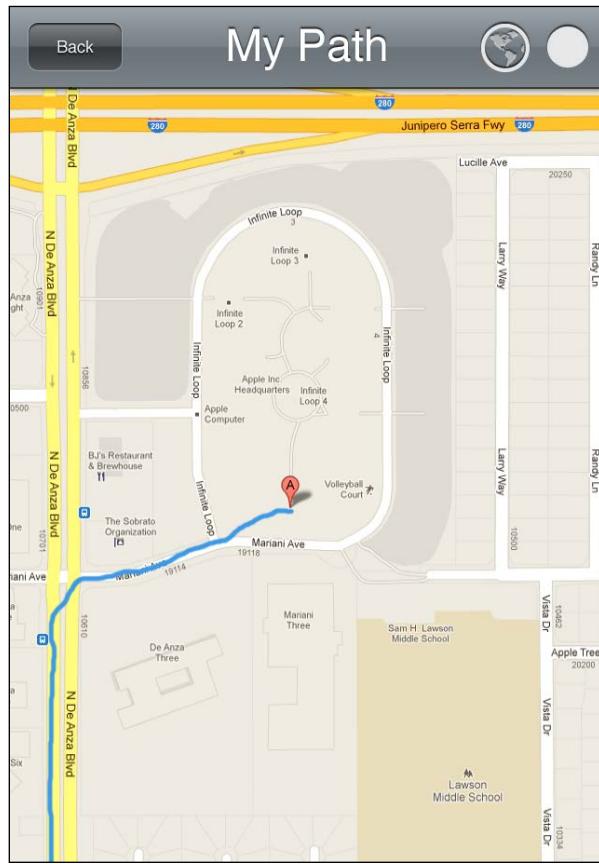
Now that we've defined how everything should work together, let's go into our graphics program and create the graphical assets we'll need.

The documents view will be as follows:



Let's Take a Trip —————

The map view will be as follows:



Generally, we've used the same assets as for our previous project, though our document image has changed to a map. We'll also need to use the images on our buttons, one for the standard find me icon, and two for the various states of the record button—a circle (for record) and a pause icon (not shown). These icons are from App-Bits' free icon set available at <http://app-bits.com/free-icons.html>. You might want to go ahead and download that set as well, if you haven't already.

What did we do?

In this task we've covered how the user interface will be designed and how the various pieces of it will work. We skipped discussing the document manager since it is virtually identical to the prior project, the only thing that has changed is the image we're using and the file extension.

Designing our data model

In this task, we will work on designing our data models. We'll focus on our document and the items that go in it; the document manager model remains unchanged from the previous project.

Getting ready

If you look up the documentation for geolocation, you'll notice that the position information contains quite a bit of information including latitude, longitude, altitude, heading, and speed. Most implementations also return the accuracy of the location and altitude, but we'll be ignoring that for now. Since our map will show the current location, we will assume that the user won't start recording until the current location is correct, and so waiting for the accuracy to settle down is less important. If we were going to begin recording instantly, we would need to wait for the accuracy to narrow to an acceptable limit, and this is where those values become useful.

Go ahead and get out your paper and pencil, and we'll start working on our data model.

Getting on with it

Our data model will look like the following screenshot:

PathRecDocument	PathRecDocumentItem
<ul style="list-style-type: none"> - fileEntry : object - fileName : string - completion : Function - failure : Function - state : string - title : string - nodes : Array of PathRecDocumentItems <ul style="list-style-type: none"> - getTitle - setTitle - getNodes - setNodes - addNode - getNodeAtIndex - getCount - readFileContents - dispatchFailure - gotFile - finishedReadingFile - saveFileContents - gotFileWriter - serialize 	<ul style="list-style-type: none"> - timestamp : date - latitude : number - longitude : number - altitude : number - heading : number - speed : number <ul style="list-style-type: none"> - setPosition - setTimestamp - getLatitude - getLongitude - getAltitude - getHeading - getSpeed - getLatLong - getGoogleLatLng - getGoogleMaker - serialize

Technically, we have three models: the two shown in the preceding screenshot and the `PathRecDocumentCollection` model, which is identical to the document manager model we saw in the last project. Since it is identical, we'll skip that and focus on the two models shown in the preceding screenshot. Let's start with `PathRecDocumentItem`:

- ▶ `timestamp`, `latitude`, `longitude`, `altitude`, `heading`, and `speed` are all properties our item needs to store. We'll collect these at object creation time and store them via `setPosition()` so that our item will be immediately populated. Unlike the `position` object used for geolocation, we won't store the coordinates in a `coords` object, but we will have to deal with that later.
- ▶ `setPosition()` takes either a geolocation position (with a `coords` object) or a serialized `PathRecDocumentItem` object (without a `coords` object). It will update the properties appropriately.
- ▶ The `get...()` property will return the requested property value.
- ▶ The `getLatLong()` property returns the latitude and longitude in the form `lat, long`.
- ▶ The `getGoogleLatLng()` property returns a Google Maps `LatLng` object.
- ▶ The `getGoogleMarker()` property returns a Google Maps `Marker` object.
- ▶ The `serialize()` property returns an object ready for storing in a JSON document.

Keep in mind that the preceding model only stores a single geolocation position; to string a bunch of them together requires the next model, `PathRecDocument`, which includes the following:

- ▶ `fileEntry`, `filename`, `completion`, `failure`, and `state` are all the same as the document in our last project.
- ▶ The `title` property stores the title of the document.
- ▶ `nodes` is an array of the earlier-listed items; this is how we'll store a series of geolocation positions. Put them all together in a line, and we'll have the path that a user took during a recording.
- ▶ The `get/setTitle()` methods returns and sets the title of the document.
- ▶ The `get/setNodes()` methods will return and set the `nodes`; these take arrays.
- ▶ The `addNode()` method will push a node onto the list of notes; this must be a `PathRecDocumentItem`.
- ▶ The `getNodeAtIndex()` method will return a node at the given index.
- ▶ The `getNodeCount()` method will return how many nodes are in the path.
- ▶ The remaining methods are the same as the document model in the previous project.

- ▶ The `serialize()` method will return an object suitable for storing in a file. Unlike in the previous project, this time `serialize()` must iterate through each node, calling its `serialize()` method to build up an array of nodes without all the extra methods in `PathRecDocumentItem`. (After all, there's no reason to store these.) The result will be an object with a `title` property, and a `nodes` array with only the position information contained within; everything else will be stripped out.

What did we do?

In this task, we created our data model, and reused portions of our previous project's data model. After all, why reinvent the wheel, right?

Next up, we need to actually implement this data model. We'll tackle that in the next task.

Implementing our data model

We'll be creating two files, namely, `PathRecDocumentCollection.js` and `PathRecDocument.js` to store our three data models. Since the first is so much like the last project's document manager, we'll skip over most of that code in the project and focus on the latter script.

Getting ready

Go ahead and open up your editor and copy the `PathRecDocument.js` and `PathRecDocumentCollection.js` files from the `www/models` directory to your project so that you can follow along with our discussion of the implementation.

Getting on with it

Before we get started with the real meat, let's take a quick look at some changes in our `PathRecDocumentCollection` model in the `PathRecDocumentCollection.js` file:

```
self.renameDocumentAtIndexTo = function ( idx, newName,
completion, failure )
{
    self.completion = completion;
    self.failure = failure;
    self.state = "Renaming a Document";
    for (var i=0; i<self.documents.length;i++)
    {
        if (self.documents[i].name.toLowerCase().trim() ==
newName.toLowerCase().trim())
```

```
        {
            self.dispatchFailure ( { "error": "The file
                already exists" } );
            return;
        }
    }
    self.documents [ idx ].moveTo ( self.fileSystem.root,
        newName.trim(), self.renameDocumentAtIndexToSuccess,
        self.dispatchFailure);
}
```

You'll notice that our `renameDocumentAtIndexTo` now has an additional few lines of code to ensure we don't overwrite a file that already exists. If there is a file with the same name, we dispatch an error to the `failure` method, and our document manager will happily prevent the user from doing anything dangerous. We've done the same thing to creating a document and copying a document.

With that out of the way, let's go to `PathRecDocumentItem` in `PathRecDocument.js`:

```
var DOC = DOC || {};

DOC.PathRecDocumentItem = function ( position )
{
    var self = this;

    self.timestamp = {};
    self.latitude = 0;
    self.longitude = 0;
    self.altitude = 0;
    self.heading = 0;
    self.speed = 0;
```

As with our model, the preceding defines our properties.

```
self.setPosition = function ( position )
{
    self.timestamp = position.timestamp;

    if (position.coords)
    {
        self.latitude = position.coords.latitude;
        self.longitude = position.coords.longitude;
        self.altitude = position.coords.altitude;
        self.heading = position.coords.heading;
```

```
        self.speed = position.coords.speed;
    }
else
{
    self.latitude = position.latitude;
    self.longitude = position.longitude;
    self.altitude = position.altitude;
    self.heading = position.heading;
    self.speed = position.speed;
}
}
```

The `setPosition()` method will set the properties to the incoming position. If it is a geolocation position (it will have a `coords` object), we use those values, but if it is a serialized `PathRecDocumentItem`, we just use those values instead.

```
self.getLatitude = function ()
{
    return self.latitude;
}
```

Like in all good objects, we provide getters for all the properties. Since they are all so simple, we won't go over each one of them.

```
self.getLatLong = function ()
{
    return self.latitude + "," + self.longitude;
}

self.getGoogleLatLng = function ()
{
    return new google.maps.LatLng( self.latitude,
        self.longitude );
}
```

The previous two methods are really convenience methods. One is to return the latitude and longitude in the form `lat, long`, and another to return a Google Maps `LatLng` object. This object is a critical object in the Google Maps API.

```
self.getGoogleMarker = function ( withMap )
{
    return new google.maps.Marker(
    {
        map:withMap,
```

```
        title:self.getLatitude(),
        draggable:false,
        position:self.getGoogleLatLang()
    }
);
}
```

This is also a convenience method, but it returns a Google Maps Marker instead. This requires a Google Map to already be initialized, but otherwise it will set up a marker with the title of Lat, Long at the same position.

```
self.serialize = function ()
{
    return {
        "timestamp": self.timestamp,
        "latitude": self.latitude,
        "longitude": self.longitude,
        "altitude": self.altitude,
        "heading": self.heading,
        "speed": self.speed
    };
}
```

In order to save the item to a file, it needs to be serialized. Since we don't need to serialize the methods, all we'll do is return an object with the location in it.

```
if (position)
{
    self.setPosition ( position );
}
}
```

Finally, at the end of the constructor, we'll set the position, if one was passed to us. If not, the object will have no position data set at all.

Next, we'll look at the PathRecDocument object. A good majority of it is similar to the document object in the last project, so we'll omit those portions.

```
DOC.PathRecDocument = function ( theFileEntry, completion,
failure )
{
    self.title = "My Path";
    self.nodes = [];
}
```

The only real difference so far is that instead of storing text, we're storing an array of `PathRecDocumentItems`. These will be used to store the coordinates within the path.

```
self.getNodes = function ()
{
    return self.nodes;
}

self.setNodes = function ( theNodes )
{
    self.nodes = theNodes;
}
```

So far, these getters and setters are pretty typical. We can request the list of items (`getNodes`), and give the object a new list (`setNodes`).

```
self.addNode = function ( aNode )
{
    self.nodes.push ( aNode );
}
```

The `addNode()` method will put a new `PathRecDocumentItem` into our node list.

```
self.getNodeAtIndex = function ( idx )
{
    return self.nodes [idx];
}

self.getNodeCount = function ()
{
    return self.nodes.length;
}
```

While we could use the `getNodes()` method to return the entire list, it can also be convenient to work with them individually; hence we work with `getNodeAtIndex` and `getNodeCount`.

```
self.finishedReadingFile = function ( e )
{
    var theFileContents = e.target.result;
    if (!theFileContents)
    {
        theFileContents = '{"title":"New
File","nodes":[]}';
    }
}
```

Most of the code leading up to the actually loading of our document has been omitted here. It's the same as the prior project, but here we start seeing some differences. First, if there's nothing in the file, we assume it to be a blank document, but we need to initialize our document with a default title and an empty list of nodes.

```
try
{
    var data = JSON.parse ( theFileContents );
    self.title = data.title;
    for (var i=0; i<data.nodes.length; i++)
    {
        self.addNode ( new DOC.PathRecDocumentItem (
            data.nodes[i] ) );
    }
}
```

Next, while the title is easy enough to set, we have to iterate through the list of nodes from the file and add them to the document. When we complete, our document will have all the saved nodes in the file.

```
self.serialize = function ()
{
    var serializedNodes = [];
    for (var i=0; i<self.nodes.length; i++)
    {
        serializedNodes.push ( self.nodes[i].serialize() );
    }
    return { "title": self.title, "nodes": serializedNodes
    };
}
```

Saving the file contents is actually the same as in the previous project, but what has changed in the previous code snippet is the `serialize()` method. First, we create an empty array and then iterate through our list of positions. We then serialize each one and add the serialized result to our array. This ensures that the array has only position data without method definitions. Then we return the title and the serialized positions, this is enough to save the document!

What did we do?

In this task, we created the data model for each geolocation position, the document that contains them, and then reused the document manager implementation from the previous project.

Changing our document manager

We've made a few minor changes to the way the document manager works. Nothing huge, but it does merit going over.

Getting ready

Open up `www/views/documentsView.html` in an editor so that you can follow along with the discussion.

Getting on with it

The largest change is how we deal with tapping parts of the document list display based on the platform we're on. If you recall in the last project, Android didn't feel terribly at home because one had to tap the icon to open the document, but if they tapped the name, they would be prompted to rename the document. In our new manager, we've reversed it so that tapping the name will open the document, and tapping the icon will prompt for a rename operation.

The changes are shown in the following document template:

```
<div id="documentsView_documentTemplate" class="hidden">
  <div class="documentContainer">
    <div class="documentTapArea"
        onclick="documentsView.documentContainerTapped(%INDEX%)"></div>
    <div class="documentImage">
      
    </div>
    <div class="documentTitle"
        onclick="documentsView.documentNameTapped(%INDEX%)">
      <span>%TITLE%</span>
    </div>
    <div class="documentActions">
      
      
      
    </div>
  </div>
</div>
```

The key differences have been highlighted. First, we've introduced a new `div` element called `documentTapArea`. This lives behind the entire document detail so that it can be a tap target for Android. It will fire off `documentContainerTapped()` so that we can respond to the event should it happen.

The next difference is for the icon: we fire off `documentIconTapped()` instead of `openDocument`.

The final difference is the title: we fire off `documentNameTapped()` instead of `renameDocument`.

These changes are easily picked out in the code, as seen in the following code snippet:

```
documentsView.documentIconTapped = function ( idx )
{
    if (PKDEVICE.platform() == "ios")
    {
        documentsView.openDocument(idx);
    }
    else
    {
        documentsView.renameDocument(idx);
    }
}

documentsView.documentNameTapped = function ( idx )
{
    if (PKDEVICE.platform() == "ios")
    {
        documentsView.renameDocument(idx);
    }
    else
    {
        documentsView.openDocument(idx);
    }
}

documentsView.documentContainerTapped = function ( idx )
{
    if (PKDEVICE.platform() == "ios")
    {
        return;
    }
    else
    {
        documentsView.openDocument(idx);
    }
}
```

What we do in each of these methods is determine how to react based on the current platform. If we're an iOS device, the icon should open the document (it's the largest tap target), the tap target beneath the document should do nothing, and the title should prompt for a rename. On the other hand, all other devices should open a document if the name or tap target have been tapped, and only prompt for a rename if the icon has been tapped.

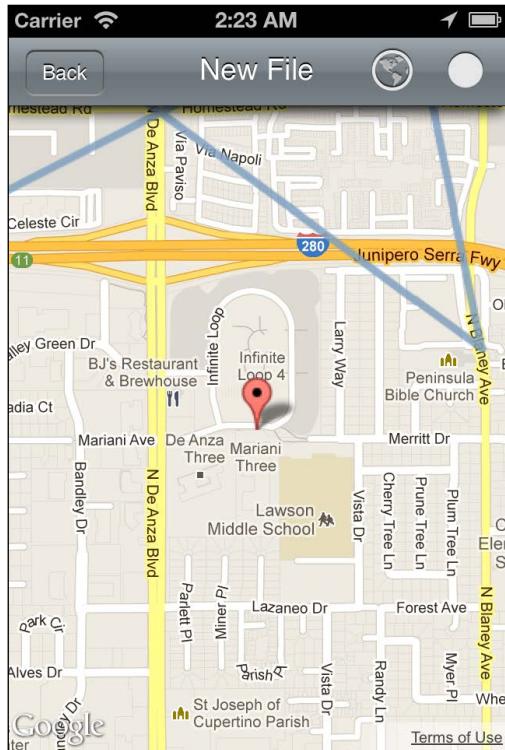
What did we do?

In this task we covered some of the subtle changes made to the document manager to help make it feel more at home on the Android platform.

Implementing our map view

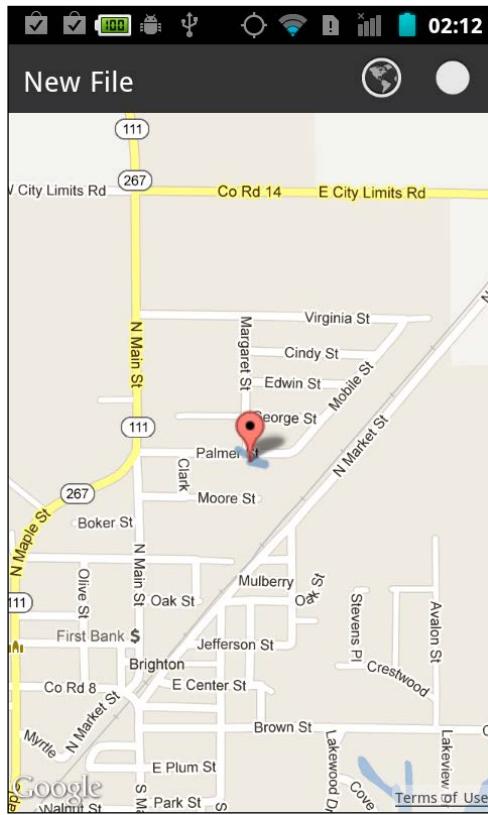
In reality, the appearance of our map view will be fairly simple. Google Maps will take care of the interactive content (and scrolling it, thankfully), so our primary focus will be on the navigation bar and interacting with the Google Maps.

When we're done, we should have something that looks like the following on iOS:



Let's Take a Trip

The view on an Android device should be as follows:



Getting ready

We'll be working from the `mapView.js` file in the `www/views` directory, so go ahead and open it so you can follow along.

Getting on with it

Let's look at our HTML for the view first:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div id="mapView_title"></div>
    <span style="display: block; position: absolute;
      right:10px; top: 6px; width: auto; text-align: right;">
```

```
<span class="iconButton" id="mapView_trackButton"
      style="margin-right: 10px;" ></span>
<span class="iconButton" id="mapViewActionButton"
      ></span>
</span>
<button class="barButton backButton"
      id="mapView_backButton" style="left:10px;" ></button>
</div>
```

First, our navigation bar has a few buttons in it. `trackButton` is intended to re-center the map on the user's location if the map has been moved. `actionButton` is essentially the *record* button, but we've named it *action* because it controls the state of recording, essentially *start* and *stop*.

```
<div class="content avoidNavigationBar" style="padding:0;
overflow:hidden;" id="mapView_scroller">
  <div id="mapView_contentArea">
    <div id="mapView_mapCanvas" style="width:100%;
height:100%;">
      </div>
    </div>
  </div>
</div>
```

The only real difference between our other views and this one at this point in the code is the `div` element named `mapView_mapCanvas`; this will be used to hold our Google Map. We give it a 100% width and height so that it fills the view.

And that's all there is to the view. Now let's look at the code behind it:

```
var mapView = $ge("mapView") || {};  
  
mapView.theFirstTime = true;  
mapView.theFileEntry = {};  
mapView.thePathDocument = {};  
mapView.theSaveTimer = -1;  
  
mapView.map = {};  
mapView.watchID = -1;  
mapView.polyline;  
mapView.currentPositionMarker = {};  
mapView.keepMapCentered = true;  
mapView.lastKnownPosition = {};  
mapView.recordingPath = false;
```

The initial properties should look familiar, they are like those in the previous project. But beyond those, we start with several properties that are going to be critically important to our view. Let's go over them before proceeding:

- ▶ The `map` property will store the reference to the Google Map we'll eventually be creating.
- ▶ The `watchID` property will store the timer ID for the watch we'll create for geolocation. The idea is that PhoneGap will call an `update` function every time the position changes, and gives us this ID to cancel this update functionality when needed. While we're in the view, we can get as many position updates as needed, but when we leave the view, we should cancel the watch.
- ▶ `Polyline` is another Google Maps API object. It will store a series of positions in an array, much like our own `PathRecDocument.nodes` property, but it does so in a Google way. We will have to translate our position items into this object in order to display the path on a map.
- ▶ The `currentPositionMarker` property will store a Google Maps `Marker` object. We'll use this marker to indicate the user's current position. You can get pretty fancy with the marker, including animating it and giving a custom image, but for now we'll use the defaults.
- ▶ The `keepMapCentered` property tracks whether or not we should, as the name implies, keep the map centered on the user's current position. If `true`, we'll pan the map's center to the current position at each update. But if `false`, we won't. We change this value whenever a `drag` is detected on the map. This means the user wants to explore the map around the current position, and if we didn't set this to `false`, the map would snap back to the current position every time the position changed, creating a disconcerting and painful experience.
- ▶ The `lastKnownPosition` property will store our last known position. This will always be kept up-to-date so that we can pan to the last known position whenever `keepMapCentered` is set to `true` after having once been set to `false`.
- ▶ The `recordingPath` property indicates whether or not we are recording the current location. If `true`, we'll store position updates and add them to the document, but if `false`, we won't store any updates. The user can toggle this by tapping on `actionButton`.

```
mapView.actionButtonPressed = function ()  
{  
    mapView.recordingPath = !mapView.recordingPath;  
    if (mapView.recordingPath)  
    {  
        mapView.actionButton.innerHTML = __T("STOP");  
    }  
}
```

```

        else
        {
            mapView.actionButton.innerHTML = __T("RECORD");
        }
        mapView.geolocationUpdate ( mapView.lastKnownPosition );
    }
}

```

The `initializeView()` method itself is similar to our other views, so we'll skip ahead to `ActionButtonPressed()`. All this function will do is toggle whether or not we are recording (and update the button accordingly). It then sends our view a geolocation update with the last known position. This way recording starts immediately and at the current position rather than waiting until a new position is received.

```

mapView.trackButtonPressed = function ()
{
    mapView.keepMapCentered = true;
    mapView.geolocationUpdate ( mapView.lastKnownPosition );
}

```

If the user pans the map, we will turn `keepMapCentered` to `false` so that the map doesn't keep springing back to the current position. This method will set it back to `true` and send a new update so that the map immediately moves to the current position (rather than waiting for a new update). This ensures that the map responds immediately, otherwise the user may not think anything happened.

If we're recording, this does of course incur an additional position in our path, but it will be at a previously known position, so it won't display. One could add code to prevent adding nodes that are exactly like the previous node.

```

mapView.geolocationUpdate = function ( position )
{
    var theLatLng = new google.maps.LatLng (
        position.coords.latitude,
        position.coords.longitude );

    mapView.lastKnownPosition = position;

    if (mapView.keepMapCentered)
    {
        mapView.map.panTo ( theLatLng );
    }

    mapView.currentPositionMarker.setPosition (theLatLng);

    if (mapView.recordingPath)

```

```
{  
    mapView.polyline.getPath().push ( theLatLng );  
    mapView.thePathDocument.addNode ( new  
        DOC.PathRecDocumentItem ( position ) );  
}  
}
```

The `geolocationUpdate()` method is called every time the user's position changes (or at a set frequency, if you wish). The first thing we do is create a new `LatLng` object with the coordinates so that we can update our marker. We also store the position into our `lastKnownPosition` property so that we can re-center the map at any point.

If `keepMapCentered` is `true`, we call the `panTo()` method on the map, which will pan the map smoothly to the new position, if possible. If the new position is a long way out of the current view of the map, Google Maps will just snap to the new position without any animation. Thankfully, we don't have to actually track that on our own; we can be assured the map will be centered on the correct location.

We then update our marker with the correct location so that it is also up-to-date. If the user is playing with the map, this means they can always see the current location (if in view).

Finally, if we're recording, we call the `push()` method for the path in our `polyline`. This has the nice benefit of automatically updating the Google Maps view to include the change in our path. Then we add the node to our own document so that we can keep track of it as well.

```
mapView.geolocationError = function ( error )  
{  
    var anAlert =  
        new PKUI.MESSAGE.Alert (   
            __T("Geolocation Error") ,  
            __T(error.message) );  
    anAlert.show();  
}
```

The geolocation functionality also allows an error function. You could do many different things here as the error object will indicate with some granularity why the function is being called: a timeout (couldn't retrieve the position in a certain period of time), the position itself is unavailable, or our request for the position was denied (which a user has the right to do). In our case, we just display the message in an alert.

```
...  
mapView.loadDocument = function ()  
{  
    mapView.viewTitle = $ge ("mapView_title");
```

```

mapView.viewTitle.innerHTML =
mapView.theFileEntry.name.substr(0,
mapView.theFileEntry.name.length-4);

mapView.thePathDocument = new DOC.PathRecDocument (
mapView.theFileEntry,
function ()
{
    mapView.viewTitle.innerHTML =
    mapView.thePathDocument.getTitle();
    mapView.theSaveTimer = setInterval (
    mapView.saveDocument, 5000 );
    for (var i=0; i<mapView.thePathDocument.
    getNodeCount(); i++)
    {
        var theNode = mapView.thePathDocument.
        getNodeAtIndex ( i );
        mapView.polyline.getPath().push ( new
        google.maps.LatLng ( theNode.latitude,
        theNode.longitude ) );
    }
},
function (e)
{
    PKUI.CORE.popView();
    var anAlert = new PKUI.MESSAGE.Alert (__T("Oops!"),
    __T("Couldn't open the file."));
    anAlert.show();
}
);

mapView.polyline = new google.maps.Polyline ( {
strokeColor: '#80A0C0', strokeOpacity:0.85,
strokeWeight:5 } );
mapView.polyline.setMap ( mapView.map );
}

```

Loading our document is pretty similar to the previous project except for what happens after the document is completely loaded into memory. In order to display our path on the Google Map, we have to push all the loaded positions into the polyline, which is created near the end of the method.

```

...
mapView.viewWillAppear = function ()
{

```

```
document.addEventListener("backbutton",
    mapView.backButtonPressed, false );
mapView.actionButton.innerHTML = __T("RECORD");
mapView.trackButton.innerHTML = __T("CENTER");
}
mapView.viewDidAppear = function ()
{
    if (mapView.theFirstTime)
    {
        mapView.map = new google.maps.Map (
            $ge("MapView_mapCanvas"),
            { disableDefaultUI: true,
                center: new google.maps.LatLng(40,-90),
                zoom: 15,
                mapTypeId: google.maps.MapTypeId.ROADMAP
            }
        );
        google.maps.event.addListener ( mapView.map,
            'dragstart', function () {
                mapView.keepMapCentered = false; } );
        mapView.currentPositionMarker = new
        google.maps.Marker(
            {
                map:mapView.map,
                title: "Current Position"
            }
        );
    }
    mapView.watchID = navigator.geolocation.watchPosition (
        mapView.geolocationUpdate, mapView.geolocationError,
        {
            enableHighAccuracy: true
        }
    );
}

mapView.loadDocument();
}
```

When showing the view for the first time, we will create the Google Map. There's no real reason to create it again later which is why we do this once. The map uses our `MapView_mapCanvas` element, which is set to fill the entire view. Notice that we pass several properties to the constructor:

- ▶ `disableDefaultUI` when `true` will turn off all the UI elements on the Google Map. This means buttons such as switching between satellite and road map types will be gone, as well as street view, panning, and zooming. Interactions with the map still work, but getting the controls out of the way is a good idea in this case, since some of them will cause the app to open a new browser (such as street view).
- ▶ `center` is where the map should be initially centered. The latitude and longitude mean nothing in particular. It will be reset when we get our first geolocation update.
- ▶ `zoom` indicates how far the map should be zoomed. A level of 15 permits users to see their current street, so it is a pretty good level to start at. The user, of course, can change their zoom later to anything they want.
- ▶ `mapTypeId` indicates the type of map, in this case, the road map type.

Once we create a map, we also add a listener to it that is fired whenever the user pans the map on their own. If they do, we set `keepMapCentered` to `false` so that future updates won't re-center the map.

After that, we initialize our current position marker. We have to tell the marker which map it will be displayed on (the one we just created), and also the title of the marker. The title doesn't appear by default, so we've picked something nice and generic.

Near the end of the method, we create a watch for the user's position using `navigator.geolocation.watchPosition()`. The first parameter indicates the `success` method to call, the second the `failure` method, and the third indicates the various options. All we are requesting in this option list is that the GPS return a highly accurate position. This can take several seconds to acquire, especially if the GPS hasn't been used in a while, so the user will see the position update as the GPS narrows things down. This is why we don't start recording immediately upon displaying the view; otherwise, we'd have to implement some sort of tolerance method to figure out when the position was accurate enough. In our case, the user will only start recording once they are satisfied with the current position on the screen, so they do our work for us.

At the end of the method, we kick off loading our document so that the view will be populated as quickly as possible.

Oh, and if you're asking why we are using `viewWillAppear` and `viewDidAppear`, it's because the Google Map doesn't like to be created while the view is in the process of being animated in. Therefore, we let the animation occur, and then load the map when everything has settled down.

```
mapView.viewWillHide = function ()  
{  
    navigator.geolocation.clearWatch ( mapView.watchID );
```

```
mapView.recordingPath = false;
mapView.keepMapCentered = true;
mapView.polyline.setMap(null); // remove from the map.
mapView.polyline = null; // and destroy.

}
```

When the view is about to be dismissed, we clear the watch on the user's position. After all, there's little to no reason to continue receiving updates when the view isn't the current view on the stack. We also turn recording off (just in case the user didn't stop the recording before dismissing the view), and turn map centering back on. We then tell the polyline that it shouldn't be visible on the map anymore and destroy it. If we didn't do this, it would be all too easy for previous paths to show up on the map from previously loaded documents.

What did we do?

In this task, we created an interactive Google Map that displays a recorded path as well as the user's current position. We dealt with polylines (which actually display the path on the map) as well as markers. We also handled recording the path and centering the map.

What else do I need to know?

Here's the sad truth as our application currently stands: when the phone is locked or the app is in the background, no updates occur. This is expected behavior as the phone has no real way to know that we want to continue receiving updates. There are platform-specific methods that you can use to register your app to continue working during a lock or backgrounding, but we'll let you go into those yourself. For now, the app only receives updates when the screen is on and the app is in the foreground.

Game Over..... Wrapping it up

In this project we achieved quite a bit. We created an interactive Google Map that the user can pan and zoom. We created markers and polylines that displayed on the map as well. We also created a way to record a path and then later display it. We also created the document model capable of storing the positional information so that it could be saved and loaded. Finally, we also made some minor changes to the document manager to make it a little more friendly to Android devices.

Some resources you might find useful:

- ▶ Google Maps API (3.9) at <https://developers.google.com/maps/documentation/javascript/3.9/reference>
- ▶ KML at <https://developers.google.com/kml/documentation/>

Can you take the HEAT? The Hotshot Challenge

We've done quite a bit in this project, but there's always room for improvement and extension. Why don't you see if you can complete some of these challenges?

- ▶ Add the ability to share a path with a friend. This could include an image of the path, the file, or some creative representation. Or, you could get into the *native* side of things and look at how to tell the phone that your app can open certain kinds of files.
- ▶ Add the ability to export a path to KML. This is a standard interchange format for geolocation data.
- ▶ Add displays of the additional data we're capturing. We only show a path using latitude and longitude, but we're capturing heading, speed, and altitude as well.
- ▶ Add the ability to edit the path once recorded, including moving, removing, and adding points.
- ▶ Come up with a way to reduce the amount of data recorded. We track every point as the user's position changes, but this is often not necessary. If someone is driving in a straight line, it would make the most sense to discard the intervening points (as well as save memory).
- ▶ This may involve some native coding, but look into and implement what is required to allow the app to capture geolocation data even while in the background and/or the device is locked. (This would save battery life, since the display wouldn't need to be visible.)

Project 5

Talking to Your App

The media capabilities of our mobile devices are frankly amazing, especially when you consider where we were five, ten, fifteen years ago. The first mass-produced MP3 player was the *SaeHan/Eiger MPMan* (http://en.wikipedia.org/wiki/Portable_media_player#SaeHan.2FEiger_MPMan) introduced in 1997. The device had 32 MB of storage, enough for roughly 6 to 7 songs (assuming 1 MB/minute, 5m per song). While it may seem paltry by today's standards, it was a revolution and spawned a new way to listen to music.

Today's devices are now so much more, portable entertainment devices that can play games, video, and all sorts of audio. Being able to play sounds in your app is critical, and there are few apps that could make the case for having no sound whatsoever. Although a bit on the extreme side, perhaps, *TweetBot* is a classic example of an app that is enhanced by the sound it produces via the user's interactions.

Today's devices can also record audio for a variety of reasons, whether it be for a reminder later, recording a speech or meeting, and more. There are a lot of apps that wouldn't require this functionality, but for a certain segment, it's important that you know how to record.

What do we build?

We will build a fairly simple app with one purpose: to store and play back the end user's recordings, whatever they may be. They could be a short memo or a meeting. We'll be using a lot of our existing framework, and there's not a lot visually, but there is a lot going on underneath to support audio playback and recording.

What does it do?

In this project, you will be able to play and record audio. We'll be recording to the `WAV` format for iOS and the `AMR` format for Android. Other platforms support other formats, so if you are targeting a platform other than Android or iOS, be sure to double-check what formats are supported.

You'll also be able to play audio; we'll support both `MP3` and `WAV` on Android and `WAV` on iOS (the primary reason we exclude MP3 for iOS here is a bug that causes MP3 format audios to render with horrible quality and extremely loud volume).

Why is it great?

There's another reason why this project is so great: we're introducing gesture support. That's right: *swipe-to-delete* and *long-press* will feature in this app as well.

How are we going to do it?

We'll be following the same task list as our previous projects:

- ▶ Designing the user interface and the look and feel
- ▶ Designing the data model
- ▶ Implementing the data model
- ▶ Implementing gesture support
- ▶ Implementing the main view

What do I need to get started?

As always, go ahead and create your project following the same steps we've used in previous projects. You might also want to refer to the PhoneGap Media API documentation, as we'll be using it extensively. (refer to http://docs.phonegap.com/en/edge/cordova_media_media.md.html#Media)

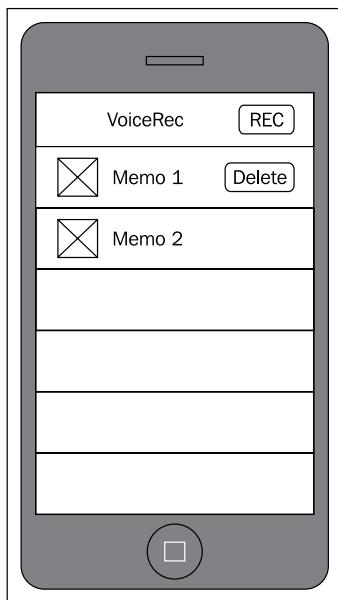
Designing the user interface and the look and feel

This app will be *visually* simpler than any of our previous apps. We only need one view, and the look of that view has already largely been defined by our Android interfaces for the last two projects. That's right; the view is essentially a list of items, nothing fancy.

Getting on with it

There are some things we will change in our list of items. Essentially we will clean up the list by hiding the action icons (delete, share, and so on) and showing them only when we receive a gesture. We will also include *Play* and *Pause* buttons in the list item instead of any particular document image. After all, we don't have album art for recordings the user creates themselves.

Let's look at the mockup:



As you can see, this mockup is pretty similar to the Android file listings we've had in our previous projects. It is substantially different than the document-based list on iOS, but the preceding view is common enough that users will know how to use it.

The icons in the list will not be document icons. Instead we'll use play and pause icons to show the state of the document. If it is currently being played, we'll show the pause icon, and if it is not being played, we'll show the play icon.

The **Delete** button on the right is shown by using the horizontal swipe gesture; these buttons are otherwise invisible.

Where are the rest of our document actions, like rename or copy, you ask? That's a great question. They're still available, but only when a user holds their finger on the item for more than a second. At that point, the long-press swipe is recognized, and a small menu will pop up asking the user what they would like to do.

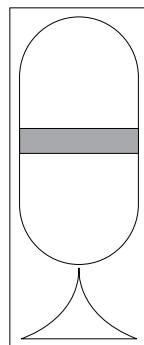
The **Record** button is intended to start a recording session. It will ask for the filename, and once entered, it will display another pop up indicating that it is recording. The user can stop recording by pressing the stop button on the pop up. We'll also display a microphone icon on this pop up to indicate to the user that the app is recording.

Now that we have the mockup finished, let's work on our graphical design in our graphics editor. The result will be as follows:

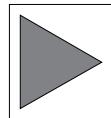


The following are also some icons that we created as part of our mockup. You can find their images in the code files available for this project.

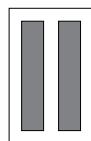
A microphone is shown as follows:



A play button can be seen as follows:



And a pause button will be as follows:



Notice that the background canvas and the navigation bar is the same as our previous project. All we really need for this task are the icons for the play state, the pause state, and the microphone. Everything else can be handled with CSS and HTML.

What did we do?

In this section, we've designed the user interface and defined the feel of the app. We've also designed the icons we'll be needing.

Designing the data model

The data model isn't terribly complex, but it is definitely a bit different from our previous projects. The document collection is fundamentally the same, so we won't cover that model, but the document itself is different. It must load and manage audio resources instead of regular files.

Getting on with it

Our model is defined as follows:

VoiceRecDocument	
- fileEntry	Object
- filename	String
- fileType	String
- completion	Function
- failure	Function
- state	String

VoiceRecDocument	
- title	String
- media	Object
- position	Number
- duration	Number
- playing	Boolean
- recording	Boolean
- paused	Boolean
- positionTimer	Timer ID
- durationTimer	Timer ID
- getFileName()	
- setFileName()	
- initializeMediaObject()	
- isPlaying()	
- isRecording()	
- updatePosition()	
- updateDuration()	
- getPlaybackPosition()	
- setPlaybackPosition()	
- getDuration()	
- startPlayback()	
- pausePlayback()	
- releaseResources()	
- stopPlayback()	
- startRecording()	
- stopRecording()	
- dispatchFailure()	
- dispatchSuccess()	

Let's go over what each of these properties and methods should do:

- ▶ The `fileEntry` property stores the file entry obtained from the File API.
- ▶ The `fileName` property stores the full path to the audio file.
- ▶ The `fileType` property stores the extension of the audio file (WAV, MP3, and so on).
- ▶ The `completion` and `failure` methods point to the `completion` and `failure` functions.

- ▶ The `title` property stores the name of the file (minus the path and extension).
- ▶ The `media` property will store the `Media` object from the `Media` API.
- ▶ The `position` property will indicate the current playback position in seconds.
- ▶ The `duration` property will indicate the current recording duration or length of the file for playback (in seconds).
- ▶ `playing`, `recording`, `paused` are internal state methods used to track what is happening inside the object.
- ▶ `positionTimer` and `durationTimer` are timer IDs used to update the position and duration properties.
- ▶ The `get/setFileName` methods gets/sets the `fileName` property.
- ▶ The `isPlaying/isRecording` methods return the respective property.
- ▶ `updatePosition/updateDuration` are internal methods used to update the position and duration property during playback and recording.
- ▶ `get/setPlaybackPosition` will get or set the current playback position. If setting, this will use the `seekTo()` method of the `Media` API.
- ▶ `start/pause/stopPlayback` will start, pause, or stop playback.
- ▶ `releaseResources` will allow the memory consumed by the media file to be released back to the device so that we don't run out of memory.
- ▶ `start/stopRecording` will start or stop recording.
- ▶ `dispatchFailure/Success` will call the `failure` or `completion` method.

What did we do?

In this task, we defined the model for `VoiceRecDocument` as well as the various interactions that go on internally.

Implementing the data model

Now that we've designed the model, let's go ahead and implement it.

Getting ready

Like in our previous projects, we'll have two data models: the first one to deal with the collection of playable files, and the second one to deal with handling a specific playable file. The first, `VoiceRecDocumentCollection.js` is quite similar to our previous projects, and so we won't go over it in this task. But the `VoiceRecDocument.js` file is very different, so go ahead and open it up (it's in the `www/models` directory), so you can follow along.

Getting on with it

Let's start using the following code snippet:

```
DOC.VoiceRecDocument = function(theFileEntry, completion, failure)
{
    var self = this;
    self.fileEntry = theFileEntry;
    self.fileName = self.fileEntry fullPath;
    self fileType =
PKUTIL.FILE.getFileExtensionPart(self.fileName);
    self.completion = completion;
    self.failure = failure;
    self.state = "";
```

As in our prior projects, the incoming parameters include a file entry obtained from the `File` API. We'll use this to determine the name of the audio file to play as well as its type. We're using some new functions, introduced in this version of the framework, to do work with the various portions that make up a file, namely, the path, the filename, and the file extension. Earlier, we use `PKUTIL.FILE.getFileExtensionPart()` to obtain the type of the file, whether it is an MP3, WAV, or something else.

```
self.title = PKUTIL.FILE.getFileNamePart(self.fileName);
self.media = null;
self.position = 0;
self.duration = 0;
self.playing = false;
self.recording = false;
self.paused = false;
self.positionTimer = -1;
self.durationTimer = -1;
```

Here we define several properties that we will use to keep track of the various states and timers we need to use to properly manage our audio:

- ▶ `title`: This property gives the title of the file, essentially the filename minus the path and extension.
- ▶ `media`: This property gives the `Media` object provided by PhoneGap. This property will be set whenever the program needs to play a sound or record something.
- ▶ `position`: This property gives the approximate position in the sound file for playback. It's approximate because it is updated every few milliseconds with the position. We'll discuss why in a bit.
- ▶ `duration`: This property gives the duration of the sound file (if playing), or the approximate duration of the recording (while, or after, recording).

- ▶ `playing, recording, paused`: These are simply Boolean variables intended to make it easy to determine what we're doing. Are we playing the file, recording a file, and, if we're playing, are we paused?
- ▶ `durationTimer, positionTimer`: Timer IDs are used to track the intervals that get created whenever we load a media file or prepare one for recording. The `durationTimer` property updates the `duration` property, and the `positionTimer` property updates the `position`.

```

self.getFileName = function()
{
    return self.fileName;
}

self.setFileName = function(theFileName)
{
    self.theFileName = theFileName;
    self.fileType =
        PKUTIL.FILE.getFileExtensionPart(self.fileName);
    self.title = PKUTIL.FILE.getFileNamePart(self.fileName);
}

```

The preceding code snippet handles getting and setting the filename. If we set a filename, we have to update the filename, type, and title.

```

self.initializeMediaObject = function()
{

```

This method is a very important method; we'll be calling it at the top of most of our methods that work with playback or recording. This is to ensure that the `media` property is properly initialized. But it is also to ensure a few other details are correctly set up, as follows:

```

if (self.media == null)
{
    if (PKDEVICE.platform() == "android")
    {
        self.fileName = self.fileName.replace ("file://","");
    }
}

```

First, we do these steps if and only if we don't already have a `media` object at hand. If we do, there's no need to initialize it again.

Secondly, we check if we're running on Android. If we are, the `file://` prefix that comes out of the File API will confuse the Media APIs, and so we remove it.

```

self.media = new Media(self.fileName,
self.dispatchSuccess, self.dispatchFailure);

```

Next, we initialize the `media` property with a new `Media` object. This object requires the filename of the audio file, and two functions: one for when various audio functions complete successfully (generally only when playback or recording has stopped), and another for when something goes wrong.

```
    self.positionTimer = setInterval(self.updatePosition,  
        250);  
    self.durationTimer = setInterval(self.updateDuration,  
        250);  
}  
}
```

Finally, we also set up our two timers to update every quarter of a second. These times could be made faster or slower depending upon the granularity of updates you like, but 250 milliseconds seems to be enough.

```
self.isPlaying = function()  
{  
    return self.playing;  
}  
  
self.isRecording = function()  
{  
    return self.recording;  
}
```

Of course, like any good model, we need to provide methods to indicate our state. Hence, `isPlaying` and `isRecording` are used in the preceding code snippet.

```
self.updatePosition = function()  
{  
    if (self.playing)  
    {  
        self.media.getCurrentPosition(function(position)  
        {  
            self.position = position;  
        }, self.dispatchFailure);  
    } else  
    {  
        if (self.recording)  
        {  
            self.position += 0.25;  
        }  
    }  
}
```

```
    } else
    {
        self.position = 0;
    }
}
```

If you recall, this function is called continuously during playback and recording. If playing, we ask the Media API what the current position is, but we have to supply a callback method in order to actually find out what the position is. This should usually be called nearly instantly, but we can't guarantee it, so this is why we have encapsulated obtaining the position somewhat. We'll define a `getPosition()` method later that just looks at the `position` property instead of having to do the callback every time we want to know where we are in the audio file.

```
self.updateDuration = function()
{
    if (self.media.getDuration() > -1)
    {
        self.duration = self.media.getDuration();
        clearInterval(self.durationTimer);
        self.durationTimer = -1;
    } else
    {
        self.duration--;
        if (self.duration < -20)
        {
            self.duration = -1;
            clearInterval(self.durationTimer);
            self.durationTimer = -1;
        }
    }
}
```

Obtaining the duration is even harder than obtaining the current position, mainly because it is quite possible that the Media API is streaming a file from the Internet rather than playing a local file. Therefore, the duration may take some time to obtain.

For as long as the duration timer is running, we'll ask the Media API if it has a duration for the file yet. If it doesn't, it'll return -1. If it does return a value greater than -1, we can stop the timer, since once we get a duration, it isn't likely to change.

There's no need to keep asking for the duration forever, especially if we can't determine the duration, so we use the negative numbers -1 to -20 of our `duration` property as a kind of timeout. We subtract one each time we fail to obtain a valid duration, and if we go below -20, we give up by stopping the timer.

```
self.getPlaybackPosition = function()
{
    return self.position;
}

self.setPlaybackPosition = function(newPosition)
{
    self.position = newPosition;
    self.initializeMediaObject();
    self.media.seekTo(newPosition * 1000);
}
```

Getting the playback position is now simple, we just return our own `position` property. But sometimes we need to change the current playback position. To do this, we use the `seekTo()` method of the `Media` API to adjust the position. For whatever reason, the position used in the `seekTo()` method is in milliseconds, while the position we obtain constantly with our timer is in seconds, hence the multiplication by 1000.

```
self.getDuration = function()
{
    return self.duration;
}

self.startPlayback = function()
{
    self.initializeMediaObject();
    self.media.play();
    self.paused = false;
    self.recording = false;
    self.playing = true;
}

self.pausePlayback = function()
{
    self.initializeMediaObject();
    self.media.pause();
    self.playing = false;
```

```
    self.paused = true;
    self.recording = false;
}
```

Starting playback is actually very simple: once we initialize the object, we just call the `play()` method on it. Playback will start as soon as possible. We also set our state properties to indicate that we are playing.

Once playing, we can also pause easily: we just have to call the `pause()` method. We update our state to reflect that we are paused as well.

```
self.releaseResources = function()
{
    if (self.recording)
    {
        self.stopRecording();
    }
    if (self.positionTimer > -1)
    {
        clearInterval(self.positionTimer);
    }
    if (self.durationTimer > -1)
    {
        clearInterval(self.durationTimer);
    }
    self.durationTimer = -1;
    self.positionTimer = -1;
    self.media.release();
    self.media = null;
}
```

Since media files can consume a lot of memory, whenever they aren't in use, they should be released from memory. When we release the file, we also need to stop the timers, if running).

```
self.stopPlayback = function()
{
    self.initializeMediaObject();
    self.media.stop();
    self.isPlaying = false;
    self.isPaused = false;
    self.isRecording = false;
}
```

Stopping playback is quite simple: just call the `stop()` method instead of the `pause()` method. The difference between the two is that pausing playback allows a subsequent call to the `play()` method to resume immediately where we paused. Calling the `stop()` method will reset our position to zero, so the next `play()` method will start from the beginning.

```
self.startRecording = function()
{
    self.initializeMediaObject();
    self.media.startRecord();
    self.isPlaying = false;
    self.isPaused = false;
    self.isRecording = true;
}

self.stopRecording = function()
{
    self.initializeMediaObject();
    self.media.stopRecord();
    self.isPlaying = false;
    self.isPaused = false;
    self.isRecording = false;
}
```

Recording is similarly easy: we just call `startRecord()` or `stopRecord()`. There is no functionality for providing support for pausing in the middle of recording.

```
self.dispatchFailure = function(e)
{
    console.log("While " + self.State + ", encountered error:
" + e.target.error.code);
    if (self.failure)
    {
        self.failure(e);
    }
}
```

Our failure method is pretty simple. If an error occurs, we'll log it, and then call the `failure` method given when creating this object.

```
self.dispatchSuccess = function()
{
    if (self.completion)
    {
        self.completion();
    }
}
```

The success function is even simpler: we just call the `completion()` method passed in when creating the object.

What did we do?

In this task, we created the data model for a specific audio file as well as the methods for initiating, pausing, and stopping playback, and those for initiating and stopping recording.

What else do I need to know?

The `completion` method is generally called at the end of playback, though it can be called for other reasons as well. In general, one would use this to clean up the media object, but if it is called when not expected, the result would be an abrupt stop of playback.

The other important issue is that each platform supports only certain media files for playback and even different ones for recording. Here's a short list:

Platform	Plays	Records
iOS	WAV	WAV
Android	MP3,WAV, 3GR	3GR

Implementing gesture support

Gestures are a critical component of most mobile platforms these days, and users expect the apps they use to support them. A gesture can be fairly elaborate (say, drawing a shape, or using multiple fingers) or simple (just pressing an item for a certain time), but it is necessary that you get used to the idea.

Getting ready

When working on the device using native code, gesture recognition is typically provided to us nearly for free. That is, the framework provided by the OS does the hard work of recognizing a gesture.

Unfortunately, with PhoneGap, we lose that for free part and have to implement our gestures on our own. That's where `ui-gestures.js` in the `www/framework` directory comes in. Go ahead and open it up so that we can walk through some of what it does.

Getting on with it

Let's take a look at the following code, starting at the top:

```
var GESTURES = GESTURES || {};  
  
GESTURES.consoleLogging = false;  
  
GESTURES.SimpleGesture = function(element)  
{
```

The first thing we do is define a new namespace called GESTURES. Then we create a SimpleGesture class within it. SimpleGesture will be the basis for all single-finger gestures that we support, which includes a long press gesture, a horizontal swipe gesture, and a vertical swipe gesture.

```
var self = this;  
  
self.theElement = {};  
  
self._touchStartX = 0;  
self._touchStartY = 0;  
self._touchX = 0;  
self._touchY = 0;  
self._deltaX = 0;  
self._deltaY = 0;  
self._duration = 0;  
self._timerId = -1;  
self._distance = 0;  
self._event = {};  
self._cleared = false;
```

There's a lot of properties that go into detecting gestures. Essentially, we have to keep track of where a touch first started, and then where that touch ended (`touchStartX`, `touchStartY`, `touchX`, `touchY`). We also need to know how far away that final touch was from when it started (`deltaX`, `deltaY`, `distance`). In order to prevent gestures from being recognized when someone holds their finger on the screen for a long time to slowly scroll through a list, we also track the duration of the touch. If it goes for too long, we refuse to detect a gesture and possibly interrupt the user performing some other operation.

We also keep track of whether or not the gesture has been recognized or cancelled. This is done with the `_cleared` property. If `_cleared` is `true`, the gesture has been recognized or cancelled and must not be recognized again (until the user lifts their finger from the screen).

```

self.attachToElement = function(element)
{
    self.theElement = element;
    self.theElement.addEventListener("touchstart",
        self.touchStart, false);
    self.theElement.addEventListener("touchmove",
        self.touchMove, false);
    self.theElement.addEventListener("touchend",
        self.touchEnd, false);

    self.theElement.addEventListener("mousedown",
        self.mouseDown, false);
    self.theElement.addEventListener("mousemove",
        self.mouseMove, false);
    self.theElement.addEventListener("mouseup", self.mouseUp,
        false);
}

}

```

The first step, however, is to attach all our event listeners to a particular element. We attach six, namely, touchstart, touchmove, touchend, mousedown, mousemove, and mouseup. The first three are for WebKit browsers; the last three are for Windows Phone browsers. (Since gesture support is part of the YASMF framework, it needs to support more than just iOS and Android, hence the support here for WP7.)

```

self.recognizeGesture = function(o)
{
    if (GESTURES.consoleLogging)
    {
        console.log("default recognizer... ");
    }
}
self.attachGestureRecognizer = function(fn)
{
    self.recognizeGesture = fn;
}

```

Part of what makes the SimpleGesture class so flexible is that it allows the recognizeGesture() method in the prior code snippet to be overridden using attachGestureRecognizer(). This also means that as part of the default implementation, we don't recognize any gesture at all yet. It's just a placeholder for the recognition engines later on.

```

self.updateGesture = function()
{
    self._duration += 100;
}

```

```
self._distance = Math.sqrt((self._deltaX * self._deltaX) +
    (self._deltaY * self._deltaY));
if (GESTURES.consoleLogging)
{
    console.log("gesture: start: (" + self._touchStartX +
        "," + self._touchStartY + ") current: (" + self._touchX +
        "," + self._touchY + ") delta: (" + self._deltaX + "," +
        self._deltaY + ") delay: " + self._duration + "ms, " +
        self._distance + "px");
}      if (!self._cleared)
{
    self.recognizeGesture(self);
}
}
```

When a suspected gesture is in progress, we call the `updateGesture()` method every 100 milliseconds. It will then helpfully calculate the distance from the initial touch and call the `recognizeGesture()` method, assuming that a gesture hasn't already been recognized yet.

Try to work out how we obtain the distance; you should recognize it from your geometry lessons.

```
self.clearEvent = function()
{
    if (self._cleared)
    {
        if (self._event.cancelBubble)
        {
            self._event.cancelBubble();
        }
        if (self._event.stopPropagation)
        {
            self._event.stopPropagation();
        }
        if (self._event.preventDefault)
        {
            self._event.preventDefault();
        } else
        {
            self._event.returnValue = false;
        }
    }
}
```

```

if (self._timerId > -1)
{
    clearInterval(self._timerId);
    self._timerId = -1;
}
self._cleared = true;
}

```

When we recognize a gesture or determine that there is no gesture at all, there's really no reason to continue tracking the fingers and such, so the preceding method cancels out all the timers. However, it only prevents the default actions that would otherwise occur (such as clicks) if the gesture itself is physically recognized (not just cancelled). This is because as part of the recognition process, we can call this method once (cancelled) or twice (recognized). The second time through we'll cancel all the default actions. This means that attaching gestures to elements won't prevent the click events from firing as long as gesture isn't recognized.

```

self.eventStart = function()
{
    if (GESTURES.consoleLogging)
    {
        console.log("eventstart");
    }
    self._duration = 0;
    self._deltaX = 0;
    self._deltaY = 0;
    self._cleared = false;
    self._touchStartX = self._touchX;
    self._touchStartY = self._touchY;
    self._timerId = setInterval(self.updateGesture, 100);
}

```

`eventStart()` is a fairly generic function. All it does is clear out some of our properties and then set others to the first touch point. It also starts the timer that calls the `updateGesture` method.

```

self.touchStart = function(event)
{
    if (GESTURES.consoleLogging)
    {
        console.log("touchstart");
    }
    if (event)
    {

```

```
self._touchX = event.touches[0].screenX;
self._touchY = event.touches[0].screenY;
self._event = event;
} else
{
    self._touchX = window.event.screenX;
    self._touchY = window.event.screenY;
    self._event = window.event;
}
self.eventStart();
}

self.mouseDown = function(event)
{
    if (GESTURES.consoleLogging)
    {
        console.log("mousedown");
    }
    if (event)
    {
        self._touchX = event.screenX;
        self._touchY = event.screenY;
        self._event = event;
    } else
    {
        self._touchX = window.event.screenX;
        self._touchY = window.event.screenY;
        self._event = window.event;
    }
    self.eventStart();
}
```

You may wonder why we have such similar handlers for `touchstart` and `mousedown`. This is because this part of the framework can technically live outside of the framework. That means it could recognize mouse events on a desktop computer as well. The other thing, however, to remember is that WP7 thinks touches are mouse events, not touch events, which is why we have to keep track of the difference. Note that we call the `eventStart()` method to do the stuff that is common to each methodology.

```
self.eventMove = function()
{
    if (GESTURES.consoleLogging)
    {
```

```

        console.log("eventmove");
    }
    self._deltaX = self._touchX - self._touchStartX;
    self._deltaY = self._touchY - self._touchStartY;

}

```

When the touch moves, `eventMove` will eventually be called by `touchMove()` or `mouseMove()`. Their code is pretty similar to `touchStart/mouseStart()` so we won't cover their code here. The main point is that as the touch moves around, the deltas are continually updated so that when `updateGesture()` is called, it can accurately determine the distance.

```

self.eventEnd = function()
{
    if (GESTURES.consoleLogging)
    {
        console.log("eventend");
    }
    self.clearEvent();
}

```

When the finger is lifted from the screen, `eventEnd()` will be called from either `touchEnd()` or `mouseUp()`. We call `clearEvent()` to reset all the tracking and timers involved.

```

self.attachToElement(element);
}

```

Finally, at the end of the object creation process, we attach the events to the incoming element. At that point, any gesture applied to the element will be tracked, but not yet recognized. That's what is covered in the next snippet:

```

GESTURES.LongPressGesture = function(element, whatToDo,
delayToRecognition, delayToCancel)
{

```

Detecting a long press is probably the easiest kind of gesture. Essentially a long press is a touch that stays within a certain spot for a certain amount of time. Since humans aren't perfect, we have to allow some tolerance to how much a finger can wiggle during this time. Thus, we can't cancel the gesture the instant we detect some finger movement. That said, we should cancel the gesture if the finger movement goes outside of a specific radius (here we'll use 25 px), because the user may be doing a different gesture altogether (or none at all).

```

var myGesture = new GESTURES.SimpleGesture(element);

```

First, we create a new `SimpleGesture` object. Then we're going to extend it using a poor man's inheritance. (It isn't really object-oriented inheritance, but it is good enough for what we need.)

```
myGesture._delayToRecognition = delayToRecognition || 1000;
myGesture._delayToCancel = delayToCancel || 3000;
myGesture._whatToDo = whatToDo;
```

We then attach the `whatToDo`, `delayToRecognition`, and `delayToCancel` parameters to the new object. If the latter two aren't supplied, we give defaults of 3000 milliseconds and we'll ignore the gesture, and 1000 milliseconds to the recognition of a long press.

`whatToDo` must be a function; we'll call it if the gesture is recognized.

```
myGesture.attachGestureRecognizer(function(o)
{
    if (GESTURES.consoleLogging)
    {
        console.log("longpress recognizer...") ;
    }
    if (o._distance < 25)
    {
        if (o._duration >= o._delayToRecognition && o._duration
            <= o._delayToCancel)
        {
            o.clearEvent();
            o._whatToDo(o);
        }
    }
})
```

Here's where we override the `SimpleGesture` object's *do-nothing* recognizer and attach our own. If the distance between the first position and current position of the touch is less than 25 px, we'll consider looking at the gesture. Then the duration of the gesture must be longer than the `delayToRecognition` (1000 milliseconds default) parameter, but not longer than the `delayToCancel` (3000 milliseconds default) parameter. If we fall in between, we'll clear the event and call `whatToDo()`.

```
else
{
    o.clearEvent();
}
});
```

On the other hand, if the distance is more than 25 px, the person isn't doing a long press. They're doing something else, so we cancel the gesture entirely.

```

        return myGesture;
    }

GESTURES.HorizontalSwipeGesture = function(element, whatToDo,
radiusToRecognition, delayToCancel)
{

```

Horizontal swipes are a little more complicated, but not by too much. First, we need to have a definition of horizontal. Again, the human finger is likely to wobble and wiggle a bit when making the gesture. We also need a minimum length; a movement horizontally of a couple pixels shouldn't be enough to trigger the gesture. Here we define a horizontal swipe as any swipe longer than 50 px and one that doesn't vary along the vertical axis by more than 25 px in either direction (50 px total).

We also do away with the `delayToRecognition` we used for long presses – that's where the length of the swipe comes into play.

```
var myGesture = new GESTURES.SimpleGesture(element);
```

Just like for our long press, we'll create the new gesture from a `SimpleGesture` object.

```

myGesture._radiusToRecognition = radiusToRecognition || 50;
myGesture._delayToCancel = delayToCancel || 3000;
myGesture._whatToDo = whatToDo;
```

Then we attach the various parameters. The `radiusToRecognition` parameter is really the length of the swipe. Anything inside that radius won't be considered at all, but anything outside the radius is long enough to be considered.

```

myGesture.attachGestureRecognizer(function(o)
{
    if (GESTURES.consoleLogging)
    {
        console.log("horizontal recognizer...");
    }
    if (o._distance > o._radiusToRecognition)
    {
        if (o._duration <= o._delayToCancel)
        {
            if (Math.abs(o._deltaY) < 25)
            {
                o.clearEvent();
                o._whatToDo(o);
            }
        }
    }
})
```

```
        }  
    } );  
    return myGesture;  
}
```

The recognizer itself is pretty easy; the distance between the first and current point must be greater than the `radiusToRecognition` parameter, the duration must not be longer than the `delayToCancel` (3000 milliseconds by default) parameter, and the finger must not have gone up or down more than 25 px from that first touch. If all these conditions are met, we've had a horizontal swipe, and we clear the event and call `whatToDo()`.

The vertical swipe is essentially identical except that instead of using `deltaY` in the earlier code, it uses `deltaX`. A vertical swipe is only valid if the finger doesn't vary on the horizontal axis by more than 25 px on either side. Since they are so similar, we won't go over the code.

What did we do?

In this task, we've looked at how to recognize three simple gestures, namely, the long press gesture, the horizontal swipe gesture, and the vertical swipe gesture. In the next task, we'll implement the gesture recognizers in order to provide the various actions we can perform on a document, such as copying, renaming, or deleting them.

Implementing the main view

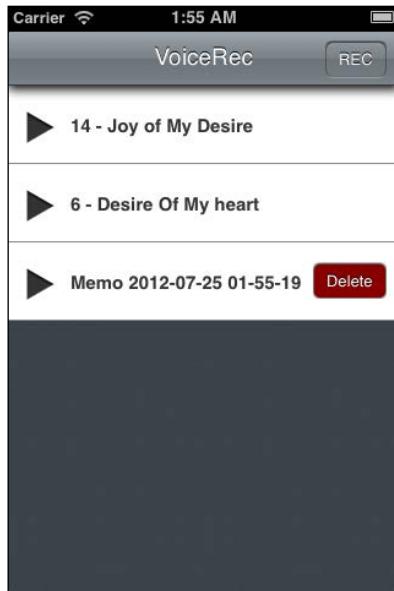
Our main view is pretty simple visually, but it is definitely complex underneath. Let's take a look at how the final result will appear. First, this is how it looks when recording:



Next, this is how it looks after the recording:



If we swipe on the newly recorded item, we get the **Delete** button as shown in the next screenshot:



If we long press on the recorded item, the following screen will appear:



Getting ready

Go ahead and open the `documentsView.js` file in `www/views` so that you can follow along.

Getting on with it

Let's first go over the HTML for our view. The first portion (with class `viewBackground`) is like all the previous projects, so we'll skip that portion and go straight to the template that shows each list item on the screen, seen in the following code block:

```
<div id="documentsView_documentTemplate" class="hidden">
  <div class="documentContainer"
    id="documentsView_item%INDEX%">
    <div class="documentTapArea"
      id="documentsView_item%INDEX%_area"
      onclick="documentsView.documentContainerTapped(%INDEX%);">
    </div>
    <div class="documentImage">
      
    </div>
```

```

<div class="documentTitle"
    onclick="documentsView.documentContainerTapped(%INDEX%) ; ">
    <span>%TITLE%</span>
</div>
<div class="documentActions"
    id="documentsView_actions%INDEX%">
    <button class="destructive barButton"
        id="documentsView_deleteButton%INDEX%"
        onclick="documentsView.deleteDocument(%INDEX%) ; return
            false;">
        %DELETE%
    </button>
</div>
</div>
</div>

```

This template is similar to, though not exactly like, the template we used in prior projects. What we've done is to change the documentImage class of the div element into the state of the document. If it shows a *play* icon, this particular item can be tapped to start playback. If it shows *pause*, it can be tapped to pause playback.

The **Delete** button is contained within the documentActions class of the div element (which is hidden by default). When tapped, it'll call deleteDocument() with the index of the item tapped.

All the other elements, such as the title, call documentContainerTapped(). This allows tapping on everything except the **Delete** button when visible to start or pause playback. If we didn't have these onclick handlers everywhere, some elements would not respond to touch like we would expect.

Now let's go over how the view works:

```

var documentsView = $ge("documentsView") || { };
documentsView.firstTime = true;
documentsView.lastScrollLeft = 0;
documentsView.lastScrollTop = 0;
documentsView.myScroll = { };
documentsView.availableDocuments = { };

```

So far, pretty much the same as previous projects.

```

documentsView.currentDocument = null;
documentsView.documentCurrentlyPlaying = -1;

```

Here, however, we begin to diverge. We need to store both the current item that is playing and an index to that item. If we didn't, and we tapped two or more items, we'd end up with all of those items playing at once. Instead, we need to stop the previous item and start the tapped item so that only one audio file is playing at one time.

We'll skip over initializing the view, since it's like all the other projects. Instead, we'll look at the `documentContainerTapped()` method in the following code snippet:

```
documentsView.documentContainerTapped = function(idx)
{
    var theElement = $ge("documentsView_item" + idx + "_img");

    if (documentsView.documentCurrentlyPlaying == idx)
    {
        if (documentsView.currentDocument.isPlaying())
        {
            documentsView.currentDocument.pausePlayback();
            theElement.setAttribute("src",
                "./images/playButton.png");
        } else
        {
            documentsView.currentDocument.startPlayback();
            theElement.setAttribute("src",
                "./images/pauseButton.png");
        }
    }
}
```

First, we check to see if the index (`idx`) is the same as the currently playing file. If it is, we need to pause (or resume) it. We don't want to release any resources or stop the file – otherwise when the item is tapped again, playback will start all over. We also set the image of the particular item to either the *play* or *pause* icon as appropriate.

```
else
{
    if (documentsView.documentCurrentlyPlaying > -1)
    {
        var theOldElement = $ge("documentsView_item" +
            documentsView.documentCurrentlyPlaying + "_img");
        documentsView.currentDocument.releaseResources();
        documentsView.currentDocument = null;
        documentsView.documentCurrentlyPlaying = -1;
        theOldElement.setAttribute("src",
            "./images/playButton.png");
    }
}
```

If the index (`idx`) is different, we first check to see if anything's currently playing (or paused). If it is, we release those resources so that we aren't keeping on to them when we're going to start playing a different item.

```
documentsView.currentDocument = new
DOC.VoiceRecDocument(documentsView.availableDocuments.
getDocumentAtIndex(idx), documentsView.mediaSuccess,
null);
documentsView.currentDocument.startPlayback();
documentsView.documentCurrentlyPlaying = idx;
theElement.setAttribute("src",
"./images/pauseButton.png");
}
}
```

Next we create a new `VoiceRecDocument` function and start the playback.

```
documentsView.mediaSuccess = function()
{
    var theElement = $ge("documentsView_item" +
documentsView.documentCurrentlyPlaying + "_img");
    documentsView.currentDocument.releaseResources();
    documentsView.currentDocument = null;
    documentsView.documentCurrentlyPlaying = -1;
    theElement.setAttribute("src", "./images/playButton.png");
}
```

The `mediaSuccess()` method, which is passed when creating the `VoiceRecDocument` function earlier, is generally called whenever the audio file is forcibly stopped (not paused) or stops on its own. Since we don't provide our own stop method visually, we can safely assume that the file has stopped playing on its own. When that is the case, we release the resources so that the file isn't taking up any memory when it isn't being played.

Next, we're going to skip to the `documentIterator()` method:

```
documentsView.documentIterator = function(o)
{
    var theHTML = "";
    var theNumberOfDocuments = 0;
    for (var i = 0; i < o.getDocumentCount(); i++)
    {
        var theDocumentEntry = o.getDocumentAtIndex(i);

        theHTML += PKUTIL.instanceOfTemplate($ge
("documentsView_documentTemplate"),
```

```
{  
    "title" : PKUTIL.FILE.getFileNamePart (  
        theDocumentEntry.name ),  
    "index" : i,  
    "delete" : __T("DELETE")  
});  
theNumberOfDocuments++;  
}  
$ge("documentsView_contentArea").innerHTML = theHTML;
```

This first portion is fairly self-explanatory. We assign the title of the item to the file name (minus the path and extension), we assign the indexes, and then also fill in the word delete whenever we come across it.

```
PKUTIL.delay(100, function()  
{
```

Next, we delay for a short time to make sure the DOM has had time to process all the new items before we go on to working with them.

```
for (var i = 0; i < theNumberOfDocuments; i++)  
{  
    var theElement = $ge("documentsView_item" + i + "");
```

Each element we created needs to have two gestures applied: a long press gesture and a horizontal swipe gesture. So first we look up the element using the preceding code snippet.

```
var theLPGesture = new  
GESTURES.LongPressGesture(theElement, function(o)  
{  
    documentsView.longPressReceived(o.data);  
});  
theLPGesture.data = i;
```

Next, we create the long press gesture and attach it to the element. When the gesture is recognized, we'll call longpressReceived() with o.data. This data is what we set in the next line; it'll be the index of the item that has been long pressed.

```
var theHSGesture = new  
GESTURES.HorizontalSwipeGesture(theElement,  
function(o)  
{  
    documentsView.horizontalSwipeReceived(o.data);  
});  
theHSGesture.data = i;  
}  
});  
}
```

Assigning a horizontal swipe gesture is much the same, except we call `horizontalSwipeReceived`.

```
documentsView.longPressReceived = function(idx)
{
    var anAlert = new PKUI.MESSAGE.Confirm(__T("Select
Action"), __T("Select an action to perform:"),
"Copy|Rename|Cancel<", function(i)
{
    PKUTIL.delay(100, function()
    {
        if (i == 0)
        {
            documentsView.copyDocument(idx);
        }
        if (i == 1)
        {
            documentsView.renameDocument(idx);
        }
    });
});
anAlert.show();
}
```

When a long press is received, we'll call the preceding method. We'll create a confirmation pop up with three possible actions: `Copy`, `Rename`, and `Cancel`. We've modified the framework a little to support more than two buttons on a pop up, so don't worry that they'll be crowding anything out.

If the user taps **Copy**, we'll call the `copyDocument()` method, and if they tap **Rename**, we'll call `renameDocument()`.

```
documentsView.horizontalSwipeReceived = function(idx)
{
    var theActionContainer = $ge("documentsView_actions" +
idx);
    if (theActionContainer.style.display == "block")
    {
        theActionContainer.style.opacity = "0";
        PKUTIL.delay(400, function()
        {
            theActionContainer.style.display = "none";
        });
    } else
```

```
{  
    theActionContainer.style.display = "block";  
    PKUTIL.delay(50, function()  
    {  
        theActionContainer.style.opacity = "1";  
    });  
}  
  
}
```

When a horizontal swipe is received, however, we do something different: we want to either display or hide the **Delete** button for that item. We can see if it is visible by checking the `style.display` property. We've taken the route of setting the `opacity` and `display` to show or hide it. This may not exactly match the native methodology (iOS slides this button in, for example), but it works well enough.

```
documentsView.startRecording = function (theFileEntry)  
{
```

Recording is probably the most complicated and difficult thing to get right. After all, each platform has different recording types that they support, but they also have their own quirks (such as whether or not the file must already exist or not).

```
documentsView.currentDocument = new  
DOC.VoiceRecDocument(theFileEntry, null, null);
```

First, we create the new document with the desired filename; we ask the user this in the `createDocument()` method.

```
var anAlert = new PKUI.MESSAGE.Confirm(  
    __T("Recording..."),  
    "<img src='./images/microphone.png' width=54  
height=123>",  
    __T("STOP_"), function(i)  
{  
    documentsView.currentDocument.stopRecording();  
    documentsView.currentDocument.releaseResources();  
    documentsView.currentDocument = null;  
    documentsView.documentCurrentlyPlaying = -1;  
    documentsView.reloadAvailableDocuments();  
});  
anAlert.show();
```

Next, we display a simple alert that has the microphone image in it and a **Stop** button. (That `_` at the end tells the alert to let the button fill the entire alert's width so that it is easier to tap.) When the user taps the **Stop** button, we'll stop recording and release all the resources. We also have to reload the documents so that the user can tap on it to play it back if they want.

```
    documentsView.currentDocument.startRecording() ;  
}
```

Here we take advantage of one important fact of our pop-up system—they don't block our script execution. That means, we can be showing the alert and then continue to do other work, in this case, recording.

```
documentsView.createNewDocument = function()  
{
```

That was the easy part; getting ready to record is the hard part. We do this in `createNewDocument()`, which is called when the **REC** button is tapped.

```
var anAlert = new PKUI.MESSAGE.Prompt(__T("Create  
Document"), __T("This will create a new document with the  
name below:"), "text", "Memo " + __D(new Date(), "yyyy-MM-  
dd HH-mm-ss"), __T("Don't Create<|Create>"), function(i)  
{
```

Like always, we ask the user to give us a new name for the document. Here we use Memo and the date.

```
if (i === 1)  
{  
    var fileType = ".wav";  
    if (device.platform() == "android")  
    {  
        fileType = ".3gr";  
    }  
    documentsView.availableDocuments.createDocument("") +  
    anAlert.inputElement.value + fileType, function()
```

First, we figure out what type of file we can record to based on the platform, and then pass that to `createDocument()`. Then, we define what should happen when `createDocument()` succeeds:

```
{  
    if (documentsView.documentCurrentlyPlaying > -1)  
    {  
        documentsView.mediaSuccess();  
    }
```

If we have an existing audio file playing, we stop it. We wouldn't want it to interfere with recording, after all.

```
var theFileEntry =
  documentsView.availableDocuments.getFileEntry();
if (PKDEVICE.platform() == "ios")
{
  console.log(4);
  theFileEntry.createWriter(function(writer)
  {
    console.log(5);
    writer.onwriteend = function(e)
    {
      documentsView.startRecording(theFileEntry);
    };

    writer.write("It doesn't matter what goes
here.");
    console.log(12);
  }, function(err)
  {
    console.log(6);
    var anAlert = new
      PKUI.MESSAGE.Alert(__T("Oops!"), __T("Couldn't
create the file."));
    anAlert.show();
  });
}
```

Then, for iOS, we create a new file with some junk text. For some reason, the Media API requires the file to exist prior to the recording, or it will fail to record. The other platforms don't have this restriction.

```
else
{
  if (PKDEVICE.platform() == "android")
  {
    theFileEntry.remove( function () {
      documentsView.startRecording(theFileEntry); } ,
    null );
  }
}
```

That said, the file gets created anyway, and can confuse Android. So we delete the file entirely before recording.

```
        else
        {
            documentsView.startRecording(theFileEntry);
        }
    },
    function(e)
{
    var anAlert = new PKUI.MESSAGE.Alert(__T("Oops!"),
__T("Couldn't create the file."));
    anAlert.show();
});

anAlert.show();
}

documentsView.renameDocument = function(idx)
{
    var theFile =
documentsView.availableDocuments.getDocumentAtIndex
(idx).name;
    var theFileName = PKUTIL.FILE.getFileNamePart(theFile);
    var theFileExt =
PKUTIL.FILE.getFileExtensionPart(theFile);
```

Previously, we only had to worry about one type of file in the list. But when renaming a file, or copying a file, we need to be sensitive to the type of file we are working with, because we need to duplicate the file extension on the new name. We do this by first getting the file extension in the preceding code snippet.

```
var anAlert = new PKUI.MESSAGE.Prompt(__T("Rename
Document"), __T("Rename your document to the following:"),  

"text", theFileName, __T("Cancel<|Rename>"), function(i)
{
    if (i == 1)
    {
        if (documentsView.documentCurrentlyPlaying > -1)
        {
            documentsView.currentDocument.releaseResources();
        }
    }
});
```

Then, if we're renaming (or deleting) a file (we don't do this for a copy), we stop any playback of the file we're renaming (or deleting) prior to actually doing the operation.

```
var theNewFileName = "" + anAlertInputElement.value +  
PKUTIL.FILE.extensionSeparator + theFileExt;
```

Next we construct the new filename with the same extension as the old file. From here on out, the code is identical to the prior projects, so we won't go over the rest of it. The important thing here is that you must preserve the file extension of any file when renaming or copying it.

What did we do?

We created a view that can list the available audio files, manage their playback, and also record new files. We also created a long press gesture and a horizontal swipe gesture for each item to implement the various file management operations necessary.

Game Over..... Wrapping it up

We've created a simple media recorder and playback app. It can manage all the files it creates (and even play a few it didn't). It also supports, for the first time, simple gestures, which are key to simplifying the complexity of the user interface.

Can you take the HEAT? The Hotshot Challenge

There are, of course, many ways you could enhance this project, as follows:

- ▶ Pop up a simple media player view over the document view while the item is playing. Give the user a way to go backward and forward in the audio file.
- ▶ Create a separate view while recording that displays the duration of the recording in a prominent fashion.
- ▶ If you want to get really creative, add this project to the `Filer` project. Allow a recording to be created as a document is written on the device, and then when that document is later viewed, play the recording so that the user can follow along.

Project 6

Say Cheese!

There was a time, not so long ago, when cameras on mobile phones didn't exist. And then there was a time when those cameras were so bad, they were really only useful for snapping *reminder shots*, for example, snapping a picture of a store item that you wanted to look up more information about later. But now so many mobile devices have excellent cameras that can take fantastic pictures. In this project, we'll look at how to use these cameras in our own app.

What do we build?

In this Project, we're going to build an app called `Imgm`. It won't be a complete app; all it will do is take pictures and let you view them. But there are a lot of things you could add on to it, such as sharing functionality or creative filters that could change the image after it was taken.

What this app will do, however, is show the process for taking a picture via the device's camera and also show the process for accessing the user's photo album so that images can be imported to the app. Like with our previous apps, we'll have full file management, so images can be deleted, copied, and renamed at will. We'll accomplish this a little differently though; we'll be implementing something seen in a lot of photo apps – the image grid.

What does it do?

Ultimately, the app will let you take a picture or import one from your photo library, and then view it. While it sounds simple enough, the app has full image management, which means we'll go over how to delete, copy, and rename images as well. In most photo apps, there is usually the option to delete images in batches (say you took several blurry photos and don't want them anymore), and deleting images in batches with the `File API` can be a bit tricky.

Most photo apps also use what is called the image grid, which is a grid of small image thumbnails that let the user see several images at once on the screen and scroll through them all. This grid isn't terribly hard to implement; it's really just a series of thumbnails that wrap using good-old HTML. Where this idea falls apart is that there's no functionality in PhoneGap to create a thumbnail from an image; which means that the images displayed as a thumbnail are really the full-sized image, they're just scaled down to a small thumbnail. This scaling introduces a lot of performance problems, and as a workaround, we'll be using the HTML5 Canvas tag to actually get some of that performance back.

Why is it great?

We'll be covering a lot of technologies in this project, all of which are critical if you're going to make a performant photo app. We'll look at how to take a picture and how to import one from the user's photo library – both of which are critically important in a large number of apps. Social apps use this kind of functionality quite often, but there are other apps that do so as well.

As always, we'll be covering file management, which is never that easy. In this project, we'll be introduced to the idea of a wrapper for the File API, which should make dealing with files just a little bit easier. Ultimately, though, we still have to get a bit tricky when dealing with files, so be prepared to twist your brain inside out.

We'll be working with something that's gaining a lot of traction in the mobile web world: the **HTML5 Canvas**. It's been used to great effect in desktop websites, but only now have our mobile devices started to become fast enough to use Canvas in interesting ways.

How are we going to do it?

As in all our previous tasks, we're going to approach this using our tried-and-true approach:

- ▶ Designing the user interface and the look and feel
- ▶ Designing the data model
- ▶ Implementing the documents view
- ▶ Implementing the image view

What do I need to get started?

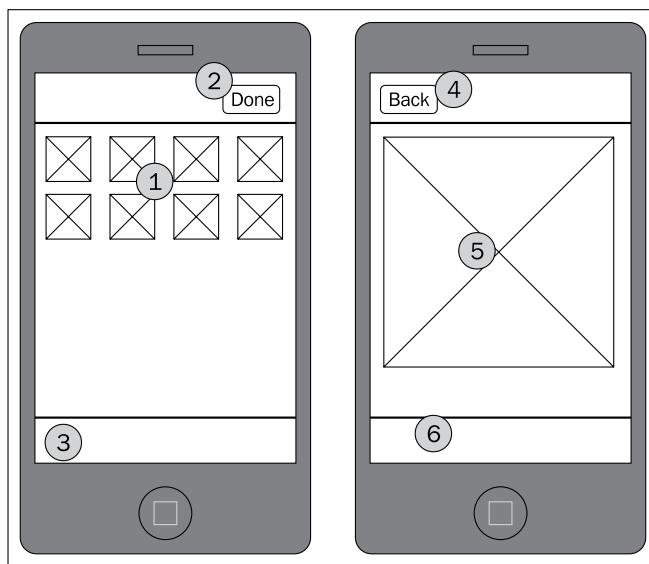
In order to get started, you'll need to create the project as you've done in each project so far. You should also take a look at the resources directory in the project files for this project. We've got several icon files that you might find interesting.

Designing the user interface and the look and feel

Conceptually, this is a pretty simple app from a user interface perspective. If you've seen a photo app on a phone, chances are you already know where we're headed. Even so, let's design a mockup, and then flesh that out a bit to come up with the assets we'll need for our look and feel.

Getting on with it

Let's examine the mockups for this project:



The left-most screen is simply a grid (1) of all the images that the user has added to the app. These images might be taken with the camera, or they might have been imported in various ways.

In the navigation bar, we'll have a fancy title – photo apps, for some reason, seem to call for something less utilitarian than some of our apps have been lately. We'll also be including an **Edit** button that can also change state to a **Done** button (2). This button indicates the current state; if a user taps **Edit**, the images will become selectable for batch operations such as a *delete* operation. In this mode, the button switches to **Done**. If the user taps the button again, they will exit selection mode, and the button will indicate **Edit** once again. Any selections the user made will be discarded.

Say Cheese! —

It's been a little while since we had a toolbar (3), too. We'll show various icons here depending upon the current editing state. If we're not in selection mode (the navigation bar button shows **Edit**), we'll show a camera and a film roll. These two icons will allow interaction with the camera and photo album, respectively. If we are in the selection mode (the navigation bar button shows **Done**), and have at least one image selected, we'll show a trashcan icon and a person icon indicating *delete* and *share*, respectively.

If we aren't in a selection mode, the user can tap an image to see it enlarged. At this point, we'll move to the right-most screen seen in the preceding screenshot. The navigation bar will have a **Back** button (4). The image will be displayed in the content area (5). If the image is larger than the screen, it will be able to scroll. In the toolbar (6), we'll provide a way to delete the image, or to share it.

Now that we've gone over the mockups, let's flesh things out a bit more. Here's the final result after doing some work in our favorite graphics editor:



While the basic elements of all our apps are present, it's clear that they've been spruced up quite a bit. The navigation bar has been given a fresh coat of paint and a playful font for the title of the app. The toolbar has also been given a nice upgrade to a darker color.

Each image will be given a simple white border and a drop shadow to help it stand out from the background. These effects are easily achieved using CSS, so we won't need images for this particular part.

For the rest of the app though, it's a different story. We'll have four icons, namely, a camera, a film roll, a trashcan, and a person. These icons will always reside in the toolbar and can be tapped to perform an action. The navigation bar and toolbar itself will also need to be a graphical asset. These would be quite difficult to render using CSS alone (especially since there is some subtle *noise* in both to give them some texture). Finally, the title itself needs to be an image asset since the font we've used may not be available on the device. About the only thing other than the photos themselves on this workup that we can do without images is the **Edit** button on the navigation bar. We'll be using the same CSS we've been using to accomplish the display of the button.

The second screen of this app, the image view, isn't mocked up here – all it would be is a large image in the middle with the rest of the design unchanged. The two icons would be the trashcan and the person in the toolbar, but that's about all that's different.

Now that we've determined what needs to be a graphical asset, it's time to splice them out of our mockup. You can see the final result in the `www/images` directory of our project.



We made the title its own image separate from the navigation bar. This is because the navigation bar is free to tile and we wouldn't want the title to tile along with it!



What did we do?

In this task we've mocked up the user interface and detailed how everything works together. We've also fleshed out the mockup so that we were able to generate the image assets we'll need for the implementation of the app.

Designing the data model

For the first time, our data model is going to be remarkably simple. There's really not a lot to keep track of here. In a way, our data model is exactly replicated by what is in persistent storage—the images themselves.

Getting on with it

Just like in prior tasks, we do have a document collection model that reads all the images in persistent storage and lets our document view interact with them. There has been almost no change in this particular model (save for the name), so we won't cover it here.

What we will cover is not quite a data model, but still important. When a user taps the **Edit** button, we want them to be able to select multiple pictures for a batch operation (such as delete). To do this, we need to keep track of which images are selected, and which ones aren't.

Say Cheese!

The model itself is so simple that it doesn't actually have its own code file. It's just an array combined with a single property that indicates if we are in selection mode or not. This is what it looks like:

- ▶ `inSelectionMode`
- ▶ `selectedItems []`

That's it. Dreadfully simple, yes, but it is important to understand how this works to provide a selection mechanism.

When the app begins, we won't be in the selection mode, so `inSelectionMode` will be `false`. If the user taps **Edit**, we change this to `true`, and change the color of the borders around all the images to a light yellow color. The color itself really doesn't matter; it's done simply to show that the device responded to the tap and that all the images are currently not selected (we'll use red to indicate selected images). This also means the `selectedItems` array will also be blank.

There are two ways we could approach how to keep track of which images are selected. We could set up an array that had as many items as images on the screen. This would work, but chances are pretty good that most of these images would remain unselected throughout the selection operation. Unless the user is intending to delete all the images (which, while possible, is something that doesn't happen very often), there's really no need to waste all that space.

Instead, we'll keep a track of these selections using a `sparse array`; each item in the array will instead point to the image that is selected. That means, if we select three images, the array only needs to be three items long. Any image not contained within the array can be considered to be unselected, and those that are in the array are considered selected.

This does present a few difficulties in how to manage this array, though. Thankfully it's not too hard to get one's head around.

Let's imagine that the user taps on image 3 while in selection mode. First we'll change the border color to something striking (red, in our example) to indicate that we *heard* the user. Then we'll use the `push` method of the array to add the image to the selection. At this point our array contains exactly one item with the value of 3. The user goes on to select a few more items, and say we finally end up with an array of 3, 1, 9. This means images 1, 3, and 9 are selected. (Notice that the order simply doesn't matter.)

Now, let's imagine that the user taps image 1 again. It's already selected, so we should unselect it. To do this, we need to remove the second item of the array so that we're left with an array of 3, 9. JavaScript makes it really easy to do this, though the names of the methods might not be immediately obvious.

First, we'll use `indexOf` to find where `1` is in the array. Once we've found the location, we'll use `splice()` to tell JavaScript to remove that particular item. The `splice()` method can be used for really cool array operations, but it also does item removal particularly well.

With these three methods on the array, we can track the selection state of any image. If we can't find the image in the array, we know it isn't selected. If we do find the image in the array, we know it is selected. And that's really all we need to know.

When the user is done with their selection, they have the option to do something with it. This is where things can get a little dicey. Let's say they want to delete several images at once. The `File` API is a bit painful, as we've seen from previous tasks, and now we have to figure out a way to call it several times in a row. In other programming languages with synchronous file operations, we'd use a simple `for` loop, but we don't have that luxury with the `File` API provided by PhoneGap, since it is an asynchronous API.

The other thing the user can do is end their selection, which they can do by tapping on **Done**. When this occurs, we will change all the image borders back to white to indicate that we've heard the user, and to also indicate that any selected images are now unselected.

What did we do?

In this task, we've examined a simple data model that keeps track of selections in an array. We've discussed how we'll use `push()`, `indexOf()`, and `splice()` to maintain this array, and how all this will appear to the user.

Since this model is so simple, we've not bothered to give it a separate file; we'll make it a part of the document view that we'll implement next.

Implementing the document view

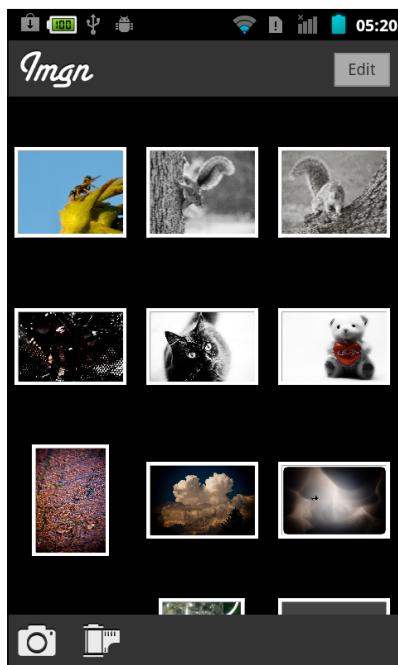
While there are portions of our document view that are the same as or similar to the previous projects, there is also quite a bit that is different. The view has to handle taking pictures, importing them, and then dealing with the user selecting several pictures at once in advance of a batch operation. This means there is quite a bit going on, even if the underlying model is pretty simple.

Say Cheese! —

Here's how the view will look, first for iOS:



For Android, the view will be as follows:



Getting ready

Our view is located in `www/views/documentsView.html` in the files for this project if you want to follow along.

Getting on with it

As always, let's start with the HTML portion of our view:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div style="padding-top:7px"
      id="documentsView_title"></div>
    <button class="barButton" id="documentsView_editButton"
      style="right:10px" ></button>
  </div>
  <div class="content avoidNavigationBar avoidToolBar"
    style="padding:0; overflow: scroll;"
    id="documentsView_scroller">
    <div id="documentsView_contentArea" style="padding: 0;
      height: auto; position: relative;"></div>
  </div>
  <div class="toolBar">
    <span class="icon" id="documentsView_cameraButton"></span>
    <span class="icon" id="documentsView_importButton"></span>
    <span class="icon" id="documentsView_deleteButton"></span>
    <span class="icon" id="documentsView_shareButton"></span>
  </div>
</div>
```

Most of this code is similar to the document views in previous projects. The `div` element `documentsView_title` has an in-line style to bring the title image down a bit; otherwise, it would align to the top of the navigation bar. The rest of the changes rest in the `div` classed `toolBar` where we have four icons defined, namely the photo (`photo_64.png`) icon, the film (`film_64.png`) icon, the trash (`trash_64.png`) icon, and the person (`man_64.png`) icon. As these icons aren't going to change based on localization, it is safe to put them here rather than define the content in `initializeView()`.

Say Cheese! —

Next, let's look at our template that we will use for each image:

```
<div id="documentsView_documentTemplate" class="hidden">
  <div class="documentContainer"
    id="documentsView_item%INDEX%">
    <div class="documentImage">
      <canvas width=84 height=84
        id="documentsView_item%INDEX%_canvas"
        onclick="documentsView.documentContainerTapped
          (%INDEX%);"></canvas>
    </div>
  </div>
</div>
```

This is probably the simplest template we've had in quite some time. All that is contained within is a `canvas` tag with a unique `id` value and a `click` handler. Don't let the simplicity fool you that we use a `canvas` tag rather than an `img` tag means that we have to have code that draws images to the `canvas` tag later on. The gains, however, that using `canvas` brings is worth this extra code.



We have given the `canvas` tag a specific `width` and `height`; this gives it a defined shape until we can later override it with the actual image width and height. This is simply to make a smoother transition from an unloaded image to a loaded image.



With the HTML out of the way, let's take a look at the code:

```
var documentsView = $ge("documentsView") || {};
documentsView.firstTime = true;
documentsView.lastScrollLeft = 0;
documentsView.lastScrollTop = 0;
documentsView.myScroll = {};
documentsView.availableDocuments = {};
documentsView.inSelectionMode = false;
documentsView.selectedItems = [];
documentsView.globalAlert = null;
```

Most of these properties just used are the ones we're familiar with from previous projects. However, there are three that are important to this particular project, mentioned as follows:

- ▶ `inSelectionMode`: This indicates if the user has placed us into the selection mode. In the selection mode, the user can select multiple images for a batch operation (such as a delete operation), whereas outside of the selection mode, tapping the image results in viewing it larger.

- ▶ `selectedItems`: This is an array, as discussed in the previous task, it contains the selected images. This is a sparse array, as it only contains those images actually selected and not those that are unselected.
- ▶ `globalAlert`: This is a placeholder for an alert. We'll use this when doing a batch delete. If the user happens to delete several images at once, it might take a few seconds, and so we want to be able to display an alert over the action going on underneath.

After this, we have the initialization code for the view given as follows:

```
documentsView.initializeView = function()
{
    PKUTIL.include(["./models/ImageDocumentCollection.js",
    "./models/VoiceRecDocument.js"], function()
    {
        documentsView.displayAvailableDocuments();
    });

    documentsView.viewTitle = $ge("documentsView_title");
    documentsView.viewTitle.innerHTML = __T("APP_TITLE_IMG");

    documentsView.editButton =
    $ge("documentsView_editButton");
    documentsView.editButton.innerHTML = __T("EDIT");
    PKUI.CORE.addTouchListener(documentsView.editButton,
    "touchend",
    function(e)
    {
        documentsView.toggleSelection();
    }
);
}
```

So far, not much is different than in our previous projects. In the preceding code, we've added text and code to the **Edit** button – if tapped, we'll call `toggleSelection()`, which will switch the selection modes.

Next, we'll define the handlers for each icon on the toolbar using the following code snippet:

```
documentsView.cameraButton =
$ge("documentsView_cameraButton");
PKUI.CORE.addTouchListener(documentsView.cameraButton,
"touchend",
function(e)
{
```

Say Cheese! —

```
documentsView.takePicture() ;
}
);

documentsView.importButton =
$ge("documentsView_importButton");
PKUI.CORE.addTouchListener(documentsView.importButton,
"touchend",
function(e)
{
    documentsView.importPicture();
}
);
;

documentsView.deleteButton =
$ge("documentsView_deleteButton");
documentsView.deleteButton.style.display="none";
PKUI.CORE.addTouchListener(documentsView.deleteButton,
"touchend",
function(e)
{
    documentsView.confirmDeletePictures();
}
);
;

documentsView.shareButton =
$ge("documentsView_shareButton");
documentsView.shareButton.style.display="none";
```



We've not attached a touch listener to the **Share** button for now. Refer to *Project 2, Let's Get Social!* if you want to implement sharing.

Something important to note is that the last two icons are set to a `display="none"`, which means they won't show up on the screen. This is because they only apply to selected images and there's no need to display them if they can't do anything. When we change selection modes and at least one image is selected, we'll re-display them.

With the view initialized, let's look at `toggleSelection()`. This is the handler for when the user taps the **Edit** button:

```
documentsView.toggleSelection = function ()
{
    var i;
```

```
var anElement;
documentsView.inSelectionMode =
!documentsView.inSelectionMode;
```

The first thing we need to do (given the name of the method) is to switch selection modes. We'll take advantage of how Booleans work to simply switch the value: if it is `false` coming in, we'll switch it to `true`, and vice versa.

```
if (documentsView.inSelectionMode)
{
    documentsView.editButton.innerHTML=__T("DONE");
    documentsView.selectedItems=[];
    for (i=0; i<documentsView.availableDocuments.
getDocumentCount(); i++)
    {
        anElement = $ge("documentsView_item"+i+"_canvas");
        anElement.style.border = "3px solid #FF8";
    }

    documentsView.cameraButton.style.display="none";
    documentsView.importButton.style.display="none";
    documentsView.deleteButton.style.display="none";
    documentsView.shareButton.style.display="none";
}
```

As seen in the preceding code, if we're now in the selection mode, we'll alter the **Edit** button to display **Done** so that the user knows how to end the selection mode. Next we empty the `selectedItems` array so that any previous selections are wiped out. Since we're using the image borders to indicate selection state, we need to iterate through each image and set its border to the unselected state (light yellow). Finally, we hide all the icons in the toolbar as none of them are immediately applicable.

```
else
{
    documentsView.editButton.innerHTML=__T("EDIT");
    for (i=0; i<documentsView.availableDocuments.
getDocumentCount(); i++)
    {
        anElement = $ge("documentsView_item"+i+"_canvas");
        anElement.style.border = "3px solid #FFF";
    }

    documentsView.cameraButton.style.display="inline";
```

```
documentsView.importButton.style.display="inline";
documentsView.deleteButton.style.display="none";
documentsView.shareButton.style.display="none";

}
}
```

On the other hand, if we're ending a selection, we need to switch the **Done** button back to **Edit** and then turn all the image borders back to white. We also have to re-enable the first two icons (camera and import) since they now apply to our current state. The last two icons are hidden (since they may have been visible just prior to ending the selection).

Switching the selection mode, of course, isn't sufficient to actually implement selection, so let's look at `documentContainerTapped()`, which is called every time an image is tapped:

```
documentsView.documentContainerTapped = function(idx)
{
    var theElement = $ge("documentsView_item" + idx +
    "_canvas");
    if (documentsView.inSelectionMode)
    {
```

Tapping on an image means different things based on whether or not we're in the selection mode. If we are in the selection mode, tapping on it should either select the image (if not previously selected), unselect the image (if previously selected), and then update the toolbar; this is done using the following code snippet:

```
if ( documentsView.selectedItems.indexOf (idx) > -1 )
{
    theElement.style.border = "3px solid #FF8";
    documentsView.selectedItems.splice (
        documentsView.selectedItems.indexOf( idx), 1);
}
```

To determine if the image is selected, we use `indexOf()`. If the image is in the array, we know the image is currently selected; therefore, tapping on it should deselect it and we change the border color to light yellow and remove the image from the `selectedItems` array using the `splice()` method.

```
else
{
    theElement.style.border = "3px solid #800";
    documentsView.selectedItems.push(idx);
}
```

If the image isn't found in the `selectedItems` array, we know we need to select it, so we change the border color (red) and add it to the array via the `push()` method.

```
if (documentsView.selectedItems.length>0)
{
    documentsView.deleteButton.style.display="inline";
    documentsView.shareButton.style.display="inline";
}
else
{
    documentsView.deleteButton.style.display="none";
    documentsView.shareButton.style.display="none";
}
}
```

Regardless of whether or not we selected or deselected the image, we need to handle the toolbar. If we've selected at least one image, we'll display the *delete* and *share* icons. If the selection ever becomes empty, we'll hide them again.

```
else
{
    PKUI.CORE.pushView (imageView);
    PKUTIL.delay(500, function()
    {
        imageView.setImage ( documentsView.availableDocuments.
            getDocumentAtIndex(idx).fullPath, idx );
    } );
}
}
```

If we're not in the selection mode, tapping on the image should move to the image view so that we can see it at full size. We push the view first, then after 500 ms we actually tell the view what image to display. This may seem odd at first (normally we'd do this in the opposite order), but it is intended to smooth the transition to the new view. Images obtained from the camera can be quite large, and loading that image takes some time. If it is loading at the same time as the transition, the transition will stutter, making the app feel slower than it really is. So instead, we wait until the transition will be over, and then tell the view to load the image.

Say Cheese! —

With selection out of the way, let's look at how we take and import pictures. Essentially these actions are the same thing, only the source of the image is different. Of course, to the user, they are very different. One involves taking a picture (framing the picture, waiting for the right moment, pressing the shutter release, and so on) while the other only involves the user searching for an image already on their camera. But to the app, they are technically the same thing with different image sources. Because of this, we'll have three methods, the first two to determine what the image source is, and the final one to do the actual work of taking or importing the image.

```
documentsView.takePicture = function()
{
    documentsView.doPicture ( Camera.PictureSourceType.CAMERA
    );
}
```

First up, taking a picture. We'll call `doPicture()` with the source of `CAMERA`, which indicates that the picture is to be obtained from the device camera. The user interface will vary based on the platform and device, but we don't need to worry about that as PhoneGap provides the interface for us.

```
documentsView.importPicture = function()
{
    documentsView.doPicture (
    Camera.PictureSourceType.PHOTOLIBRARY );
}
```

To import a picture, we call `doPicture()` with the source of `PHOTOLIBRARY`. This indicates that the image should come from the user's own library of images. Again, the user interface will vary based on the platform and device, but our app doesn't have to worry about that as PhoneGap will handle all the details for us.

```
documentsView.doPicture = function( source )
{
    navigator.camera.getPicture (
        function (uri)
    {
        PKFILE.moveFileTo ( uri, "doc://" +
        PKUTIL.getUnixTime() + ".jpg",
        function ()
        {
            documentsView.reloadAvailableDocuments();
        },
        function (evt)
        {
```

```
        console.log (JSON.stringify(evt));
        var anAlert = new
        PKUI.MESSAGE.Alert(__T("Oops!"), __T("Failed to
        save the image."));
        anAlert.show();
    } )
},
function (msg)
{
    var anAlert = new PKUI.MESSAGE.Alert(__T("Oops!"),
    msg);
    anAlert.show();
},
{ quality: 50,
destinationType: Camera.DestinationType.FILE_URI,
sourceType: source,
encodingType: Camera.EncodingType.JPEG,
mediaType: Camera.MediaType.PICTURE,
correctOrientation: true,
saveToPhotoAlbum: false
}
);
}
```

There are several layers of callbacks going on in this function, each of which relies on the previous step being executed correctly. Let's look at the outer layer for now.

To take an image, or import one, we call `navigator.camera.getPicture()` with three parameters: the `success` function, the `failure` function, and the options. The options in our function are at the very end of the method. Here's what each one means:

- ▶ `quality`: This is the compression used for the image. We use 50 because it is a good tradeoff between quality and file size. Furthermore, some devices have problems handling images from the camera with a quality higher than 50. (Typically, on devices that exhibit problems with a quality larger than 50, the app would crash. Not a good end-user experience.)
- ▶ `destinationType`: It determines the destination. There are two options here: we can either request a base64-encoded string which represents the image data, or we can request the file location where the image was saved. We're requesting the file location instead of base64, simply due to ease of handling and memory concerns. (Base64 is at least double the file size of the image.)

Say Cheese!

- ▶ `sourceType`: It determines where the image should come from. If set to `CAMERA`, it will get the image from the camera. If `PHOTOLIBRARY`, it will get it from the user's library. Note that we take the incoming parameter here, which is what `takePicture()` and `importPicture()` send to us.
- ▶ `encodingType`: This is the image format, generally either JPEG or PNG. PNGs are great for images with a lot of pixel repetition (such as diagrams), and are lossless. For photography, however, PNGs would be too large. Instead, we'll use JPEGs. Though lossy, they won't be so large as to be unwieldy.
- ▶ `mediaType`: The camera can often be used to take video instead of a still image. In this case, all we want is a still image, so we send `PICTURE`. This also limits the available formats when importing an image. Without this, a user could import a video instead, which we can't handle.
- ▶ `correctOrientation`: This parameter can be used to correct the orientation used to take an image. For example, if the phone was rotated and we didn't correct the orientation, the image might appear sideways or upside-down. With this enabled, we get the image right-side up.
- ▶ `saveToPhotoAlbum`: This can be either `true` or `false`. If `true`, the image taken by the camera will be saved to the photo album and our app. If `false`, only our app receives the image. While we use `false` here, it is really a matter of choice. Does it make sense for the image to be saved to the album as well as your app? The answer to that question depends on your app and its target audience.

If we move into the `success` function for `navigator.camera.getPicture()`, we see this line:

```
PKFILE.moveTo ( uri, "doc://" +
    PKUTIL.getUnixTime() + ".jpg",
```

The file given to us by the camera may be in a temporary spot (especially on iOS), so we first move the file to a more permanent location. We're using the `PKFILE FILE` API wrapper to do this, which we'll discuss later on in this task. Using `doc://` here ensures that the file is written to persistent storage, and using the Unix Time (which is milliseconds since 1st January, 1970) ensures a nearly unique filename. (It's frankly impossible for the user to take two images in quick enough succession to end up on the same millisecond.)

When the move is complete, we call another `success` function, which simply reloads the available documents, which in our case redisplays the image grid with the new image in it.

Along the way we also have `failure` functions where we have the `alerts`. These are important if, for some reason, the camera fails to take the picture, or an import goes badly.

Next up, let's handle deleting multiple pictures at once (deleting a single picture is done the same way we've deleted documents in past projects):

```
documentsView.confirmDeletePictures = function ()  
{  
    var anAlert = new PKUI.MESSAGE.Confirm(__T("Delete  
Image(s)"), __T("This will delete the selected image(s).  
This action is unrecoverable."), __T("Don't  
Delete<|Delete*"), function(i)  
    {  
        if (i == 1)  
        {  
            PKUTIL.delay ( 100,  
            documentsView.deleteSelectedPictures ) ;  
        }  
    }) ;  
    anAlert.show() ;  
}
```

First we ask the user if they really want to delete the selected images as the action is unrecoverable. If they do, we call `deleteSelectedPictures()` after a short delay. This delay is to give enough time for the first alert to go away before `deleteSelectedPictures()` puts up its own alert.

```
documentsView.deleteSelectedPictures = function ()  
{  
    if (documentsView.selectedItems.length > 0)  
    {  
        var currentIndex = documentsView.selectedItems.pop() ;  
        if (documentsView.globalAlert == null)  
        {  
            documentsView.globalAlert = new  
            PKUI.MESSAGE.Alert(__T("Please Wait"), __T("Deleting  
Selected Images...")) ;  
            documentsView.globalAlert.show() ;  
        }  
        PKUTIL.delay (100, function () {  
            PKFILE.removeFile (  
            documentsView.availableDocuments.  
            getDocumentAtIndex(currentIndex).fullPath,  
            documentsView.deleteSelectedPictures,  
            function (e)  
            {  
                documentsView.globalAlert.hide() ;  
            }  
        }) ;  
    }  
}
```

```
var anAlert = new PKUI.MESSAGE.Alert (__T("Oops!") ,
__T("Failed to remove file."));
anAlert.show();
documentsView.reloadAvailableDocuments();
}
);
}
);
}
else
{
    if (documentsView.globalAlert)
    {
        documentsView.toggleSelection();
        documentsView.reloadAvailableDocuments();
        documentsView.globalAlert.hide();
        PKUTIL.delay (750, function() {
            documentsView.globalAlert = null; } );
    }
    else
    {
        console.log ("ASSERT: We shouldn't be able to delete
anything without having prior selected something.");
    }
}
}
```

This method requires us to switch our brain inside-out a bit, since there's something missing we might normally expect—a `for` loop. Because the FILE APIs are all asynchronous, we can't loop around them, we need to be able to ensure all the API requests are finished before we tell the user we're done.

So instead, we use something akin to recursion. It's not true recursion, since the function calls aren't nested within each other, but it is close enough to be a bit painful on our neurons.

We start by checking the length of the `selectedItems` array. If it has any images in it, we know we need to delete one. If it has nothing in it, we know we've finished the job and can clean everything up.

If we need to remove an image, we call `PKFILE.removeFile` with the full path to the image. We also pass along our current method to the `success` function. This means that once the image is successfully deleted, we'll be called again to repeat the process until we've deleted all images in the `selectedImages` array. The process will only stop when a failure occurs or when the `selectedImages` array is empty.

When the `selectedImages` array is empty, we need to clean things up. We clear the alert we created when starting the deletion process, and we reload all our available documents.

With all this talk about reloading the document listing, perhaps we should go into that a bit:

```
documentsView.documentIterator = function(o)
{
```

The first portion of this function is identical to previous projects, so we'll skip ahead to the interesting bits in the following code snippet:

```
PKUTIL.delay(100, function()
{
    for (var i = 0; i < theNumberOfDocuments; i++)
    {
        var theDocumentEntry = o.getDocumentAtIndex(i);
        var theElement = $ge("documentsView_item" + i + "");
        var theLPGesture = new
GESTURES.LongPressGesture(theElement, function(o)
{
    documentsView.longPressReceived(o.data);
});
theLPGesture.data = i;
```

Just like in our last project, we have a long press gesture attached to each image. When a long press is received, we'll display a menu allowing the user to delete, copy, or rename the image.

The more interesting bit is this next part, where we actually render the image's thumbnail onto each canvas:

```
var img = new Image();
img.i = i;
```

First, for each available document, we create a new `Image` object. We assign an `index` to a property of the object as well, because we'll need it later when the image is finished loading.

```
img.onload = function ()
{
```

We then attach a method to the `onload` event of the image. This is where we'll render the image to the canvas.

```
var newWidth = 84;
var newHeight = (this.height / this.width) * 84;
if (newHeight > 84)
{
    newHeight = 84;
    newWidth = (this.width / this.height) * 84;
}
```

Say Cheese! —

The first thing we do after the image is loaded is to determine the size of the thumbnail. Since we want to maintain the aspect ratio, we have to know the image's width and height, something we can do by using `this.height` and `this.width`. First we assume the image will fit into a width of 84 pixels, and determine the height using the ratio of the height to the width. But on some images this might result in them being taller than 84 pixels, and so we redo the calculation for these images based on a height of 84 pixels.

```
var newLeft = 42 - (newWidth/2);  
var newTop = 42 - (newHeight/2);
```

Once the thumbnail's width and height are determined, we can figure out the top and left so that the image is nicely centered within its 84 x 84 container.

```
var theCanvas = $ge("documentsView_item" + this.i +  
    "_canvas");  
theCanvas.setAttribute ("width", newWidth *  
    window.devicePixelRatio);  
theCanvas.setAttribute ("height", newHeight *  
    window.devicePixelRatio);  
theCanvas.style.width = ""+newWidth+"px";  
theCanvas.style.height = ""+newHeight+"px";  
theCanvas.style.left = ""+newLeft +"px";  
theCanvas.style.top = ""+newTop+"px";
```

Next, we obtain the canvas and set its width and height to the newly calculated values. Notice that we set both the CSS and the HTML width and height; this is important because the size of the canvas may actually be different on displays with different pixel ratios. (On a retina display the canvas is actually twice as large, but the CSS compresses it back to the original size visually.)

```
var theCanvasCtx = theCanvas.getContext ("2d");  
theCanvasCtx.save();  
theCanvasCtx.scale (window.devicePixelRatio,  
    window.devicePixelRatio);  
theCanvasCtx.imageSmoothingEnabled = false;
```

Next, we obtain a context from the canvas. We then scale the canvas to the pixel ratio so that we can continue to use point-based pixels on devices where each point is more than one pixel. Then we turn off image smoothing; this is to help speed up the next line:

```
theCanvasCtx.drawImage (this, 0, 0, newWidth,  
    newHeight);
```

This line physically draws the image to the canvas at the size we calculated earlier. Once done, we will have a nice thumbnail instead of an empty canvas.

This operation isn't free, however; it takes time to draw the image to a smaller scale, but we only have to do this once (whenever we load the available documents). If we had used `IMG` tags for all our thumbnails, this would have had to be done every time something changed on the view, which would be so slow as to be unusable.

```
    theCanvasCtx.restore();
}
img.src = theDocumentEntry.fullPath;
}
);
}
```

Finally, we set the image's source. This triggers loading of the image, which will trigger the image's `onload()` method when finished. This operation isn't free either, but again, it only happens whenever we load the list of documents.

The rest of the view's code is very similar, if not identical, to prior projects, so we won't go over that code again.

What did we do?

We've touched on quite a bit of stuff in this task. We created code that can take and import pictures, we worked with the HTML5 `canvas` tags, and we started working with a `FILE API` wrapper that makes working with files just a bit easier.

What else do I need to know?

We mentioned that we would discuss the `FILE API` wrapper a bit more, and here's as good a place as any.

Think of `PKFILE` as a convenience wrapper that makes working with files somewhat easier. It doesn't take away the asynchronous nature, but it does encapsulate some of the operations that a file operation typically has to do, getting the file system especially. It also gives us the ability to add in some shortcuts for referencing persistent and temporary storage.

Let's cover that last part first. Any filename that contains one of the following automatically gets translated to the system-specific value:

- ▶ `doc://` is translated to `/path/to/app/persistent/storage/`.
- ▶ `tmp://` is translated to `/path/to/app/temporary/storage/`.
- ▶ `file://localhost` is replaced with `" "`; the `FILE APIs` can't handle a path starting with this.

Say Cheese! —

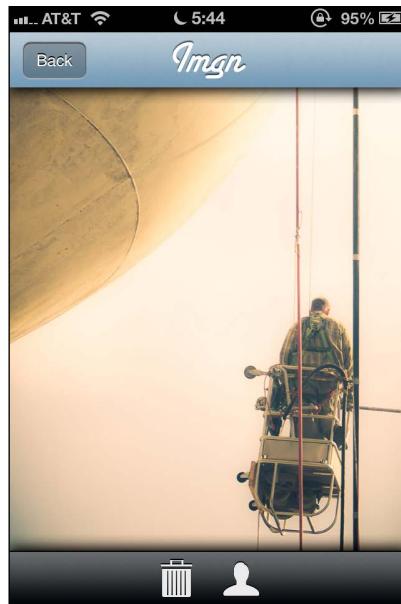
Because `PKFILE` will translate these values in any path or filename for every method, we no longer have to worry about obtaining the file systems for ourselves. This cuts out at least one callback chain and simplifies how we refer to files within our own app's storage. For example, we can refer to `doc://photo.jpg` instead of something like `/var/something/somethingelse/app/Documents/photo.jpg`.

Each method in `PKFILE` takes some combination of filenames, and `success` and `failure` parameters. The `failure` functions are always passed an object that indicates the reason for failure. The `success` functions aren't passed any parameters.

We won't go over the code for the wrapper, mainly because it isn't really anything you haven't seen before. If you want to take a look, it is located in `framework/fileutil.js` in the files available for this project.

Implementing the image view

The image view itself is very simple: all it does is display a single image along with two icons in the toolbar (**Delete** and **Share**). Here's how our view will look, first for iOS:



For Android, the view is as follows:



Getting ready

If you want to follow along, the code is in [www/views/imageView.js](#).

Getting on with it

Typically we'd start with the HTML for the view, but this is very similar to the previous project. Instead we'll start with the template used to display the image:

```
<div id="imageView_documentTemplate" class="hidden">
  
</div>
```

This is probably the simplest template we've ever had. It is literally just an image with a specified width. The height will be inferred from the aspect ratio of the image.

Like the template, the code is going to very simple as well:

```
var imageView = $ge("imageView") || {};
imageView.imagePath = "";
imageView.imageIndex = -1;
```

Say Cheese! —

```
imageView.setImage = function ( imagePath, imageIndex )
{
    imageView.imagePath = imagePath;
    imageView.imageIndex = imageIndex;

    $ge("imageView_contentArea").innerHTML =
        PKUTIL.instanceOfTemplate($ge
        ("imageView_documentTemplate"),
        { "src" : imageView.imagePath });
}
```

We store two items: the path to the image and the index of the image. We also provide a method called `setImage` that others can use to tell us which image to load. Once called, we replace the content area with the image data.

Our view also supports deleting the picture being viewed. Here's how that is handled:

```
imageView.confirmDeletePictures = function ()
{
    var anAlert = new PKUI.MESSAGE.Confirm(__T("Delete
Image"), __T("This will delete the selected image.
This action is unrecoverable."), __T("Don't
Delete<|Delete*"), function(i)
{
    if (i == 1)
    {
        PKUTIL.delay ( 100, imageView.deleteSelectedPicture );
    }
});
anAlert.show();
}

imageView.deleteSelectedPicture = function ()
{
    PKFILE.removeFile ( imageView.imagePath,
    function ()
    {
        PKUTIL.delay(100, function()
        { PKUI.CORE.popView(); } );
        documentsView.reloadAvailableDocuments();

    },
}
```

```
function (e)
{
    var anAlert = new PKUI.MESSAGE.Alert (__T("Oops!"),
__T("Failed to remove file."));
anAlert.show();
documentsView.reloadAvailableDocuments();
}
);
}
```

We first ask the user if they are sure, and if so, we will remove the file via `PKFILE.removeFile()`. Once removed, we pop ourselves off the view stack because it doesn't make a lot of sense to be viewing an image that is now deleted. We also tell the `documentsView` to reload its documents, seeing as we've modified the file system.

```
imageView.viewDidHide = function ()
{
    $ge("imageView_contentArea").innerHTML = "";
}
```

There's one last method to cover, and that's the `viewDidHide()` method seen in the preceding code snippet. All we do here is clear out the content area so that when the image view is not being displayed, no image is sitting there hidden and taking up the memory. It also means the next time the view is displayed, there isn't an odd transition where the last loaded image is visible for a couple moments before the new one is loaded. (Remember, we wait a few milliseconds before loading in a new picture to ensure a smooth transition.)

What did we do?

In this task, we implemented the image view, handled loading the image, and also handled deleting the image.

Game Over..... Wrapping it up

Well, that wasn't terribly hard, was it? We have been able to take pictures using the in-built camera, and we've also been able to import images directly from the user's photo library. We've worked with displaying thumbnails via the HTML5 canvas tag to improve performance as well. Finally, we have worked a little with a FILE API wrapper to help make our code simpler to write.

Say Cheese! —

Can you take the HEAT? The Hotshot Challenge

There's a lot that could be done to improve the project as it stands now. Why don't you try a few?

- ▶ Add sharing capabilities to the app. Be careful to deal with batch sharing multiple images, too.
- ▶ Add filters to the images so that they can be converted to black and white, sepia, and more. If you want to save the files, you'll need to write a native plugin to save the data returned from the canvas.
- ▶ Speed things up a bit by generating a thumbnail from each image if it doesn't already have one. You could store the thumbnail in local storage if you wanted, it wouldn't be very big, or use the File API to store it.

Project 7

Let's Go to the Movies!

It didn't take long after the introduction of cameras on phones to ask the question, "What about video?" Initially video recording was hampered by the limited space on phones as well as the hardware. It is one thing to snap a few JPEG files that might be a few hundred kilobytes each, but it is entirely another thing to take a video longer than a few seconds that doesn't add up to a pretty large file size. Furthermore, video has to be encoded and compressed, something that can be done in software, but is much better done in hardware.

When video recording did become practical, it changed the way we saw the world. Suddenly it was possible to have footage of news happening anywhere there was a cell phone – practically everywhere. Video recording became common place, so chances are good that there are very few handsets today that don't support it. In this project, we'll take a look at recording and playing video in our apps.

What do we build?

In all honesty, we're going to be building our *Project 6, Say Cheese!*. Remember that one? Instead of working with still pictures, we'll be working with videos instead.

While the user interface and a good percentage of the code will be the same, video does present some interesting challenges. For example, how does one get a thumbnail of a video? We all know how a thumbnail should look, but how does one actually get it? Or, frankly, how do we do more than simply display the video? How do we enable playback?

Unfortunately (or fortunately, depending on your viewpoint), we'll need to delve into some native code in order to accomplish these tasks. The tasks themselves aren't complicated, but PhoneGap doesn't provide support for thumbnails of videos, and the Android platform doesn't provide good support for the HTML5 VIDEO tag, so we'll have to use another plugin for playing video as well.

What does it do?

Our app, called Mem'ry (we've got to be *hip* by leaving vowels out, right?) will allow the user to record video from within the app. Any recorded videos can be played back. Furthermore, we'll use the same document management we've used before to permit management of these files—deletion, duplication, and renaming.

Why is it great?

Let's face it, recording video isn't something that a lot of apps have to deal with. But playing it back? That's a much larger percentage of apps. Therefore, it is very important that you know how to play videos for the end user, but should you ever need to record video as well, this app will give you the tools necessary for that as well.

We'll also delve into some areas that PhoneGap doesn't really provide any support for. It's not so much PhoneGap's fault, really, it's more that videos aren't simply a collection of images. They're compressed and encoded, and there's no obvious way to get a thumbnail of a video and display it as an image in HTML, which is what we've been using so far when working with PhoneGap. A thumbnail of an image, of course, is easy: just scale the image down. A thumbnail of a video? Not so easy; you've got to construct it from the compressed and encoded file, and HTML doesn't know how to do that. Fortunately, the SDKs provided on the most popular platforms make it easy, but we'll have to do some native coding to get there.

How are we going to do it?

We'll be following these steps:

- ▶ Preparing for the video thumbnail plugin
- ▶ Implementing the video thumbnail plugin for iOS
- ▶ Implementing the video thumbnail plugin for Android
- ▶ Integrating with the video thumbnail plugin
- ▶ Implementing recording and importing of video
- ▶ Implementing video playback

What do I need to get started?

First, make sure to get the files for this project from the download available for this book. We won't be listing the code in its entirety, since it is so similar to that in *Project 6, Say Cheese!*, so in order for you to follow along you should either have a quick reference to your previous project or the one from our files.

Second, be sure to download the Android Video Player plugin from <https://github.com/phonegap/phonegap-plugins/tree/master/Android/VideoPlayer>. In this project, we used the plugin for PhoneGap version 2.x and higher.

Preparing for the video thumbnail plugin

Our app is practically identical with regards to the prior project's user interface and its interactions, so we won't go through the entire design process. Instead, we'll start right off with implementing the video thumbnail plugin.

Getting ready

We'll be getting our hands dirty with native code for the first time in this book. While we've used plugins before (the ChildBrowser plugin in *Project 2, Let's Get Social!*), we've never created our own yet. Since we support more than one platform, we'll also have to write the plugin more than once.

Getting on with it

This task is essentially composed of three steps:

- ▶ Configuring the project to use the plugins
- ▶ Creating the JavaScript interface
- ▶ Creating the native code

The first two tasks are quite simple, but the last one, well, we'll cross that bridge in a few pages.

Configuring the project to use the plugins

The first thing we need to do is to configure the projects we're building to use the new plugin. The steps are different for each platform, but at a *high level* they are essentially the same thing. We're telling PhoneGap about the plugin and that it is available for use. The steps are as follows for the different platforms:

► iOS

1. Using Xcode, navigate to the `Cordova.plist` file.
2. Expand the **Plugins** section.
3. Click on the **+** sign that appears when hovering over a plugin. This will insert a new row.
4. Add `PKVideoThumbnail` as the key, and `PKVideoThumbnail` as the value.
5. Save the file.
6. Go to the `index.html` file and add the `PKVideoThumbnail.js` script file given as follows:

```
<script type="application/javascript"
charset="utf-8"
src="../plugins/iOS/PKVideoThumbnail.js">
</script>
```

► Android

1. Using Eclipse, navigate to the `/res/xml/config.xml` file and open it. If you aren't presented with the editable XML, switch to the XML text editor view.
2. Find the **Plugin** section of the XML file.
3. Add the following line:
`<plugin name="PKVideoThumbnail" value="com.kerrishotts.PKVideoThumbnail.PKVideoThumbnail"/>`
4. Save the file.
5. Go to the `index_android.html` file and add the `PKVideoThumbnail.js` script file given as follows:

```
<script type="application/javascript"
charset="utf-8"
src="../plugins/Android/PKVideoThumbnail.js">
</script>
```

All this does is tell PhoneGap that we're going to have a plugin available with the name of `PKVideoThumbnail`. Without this, the app wouldn't work correctly, since it wouldn't know how to contact the plugin.

Creating the JavaScript interface

While it is technically possible to call a plugin without having any corresponding .js file, it is a fact that it is often easier to create an interface so that calling the plugin is just a little bit easier. The interface files will be nearly identical, but the iOS version will be just enough different that it wouldn't work under the Android platform and vice versa.

Under the `www/plugins/iOS` directory, create a file called `PKVideoThumbnail.js` with the following contents:

```
var PKVideoThumbnail = PKVideoThumbnail || {};
PKVideoThumbnail.createThumbnail = function
( source, target, success, failure )
{
    cordova.exec(success, failure,
    "PKVideoThumbnail",
    "createThumbnail",
    ["file://localhost" + source, target]);
}
```

What the preceding code does is simply create an easy-to-use wrapper call that lets us call `PKVideoThumbnail.createThumbnail()` whenever we need to get a video's thumbnail instead of using `cordova.exec(some_function,error_function,"PKVideoThumbnail","createThumbnail",[])`.

For some plugins that offer a lot of functionality, the JavaScript interface essentially acts as a go-between. It translates arguments to what the plugin can understand, and then handles the return results when returning back to JavaScript. In our case, the wrapper is pretty small.

For the Android version, create a file named `PKVideoThumbnail.js` under `www/plugins/Android` with these contents:

```
var PKVideoThumbnail = PKVideoThumbnail || {};
PKVideoThumbnail.createThumbnail = function
( source, target, success, failure )
{
    cordova.exec(success, failure,
    "PKVideoThumbnail",
    "createThumbnail",
    [source, target]);
}
```

Could you spot the difference between the two files?

I'll give you a clue: look at the penultimate line, just before `source`. The iOS version adds `file://localhost`, while the Android version doesn't. Small detail, but without it, the iOS version of the app wouldn't work.

What did we do?

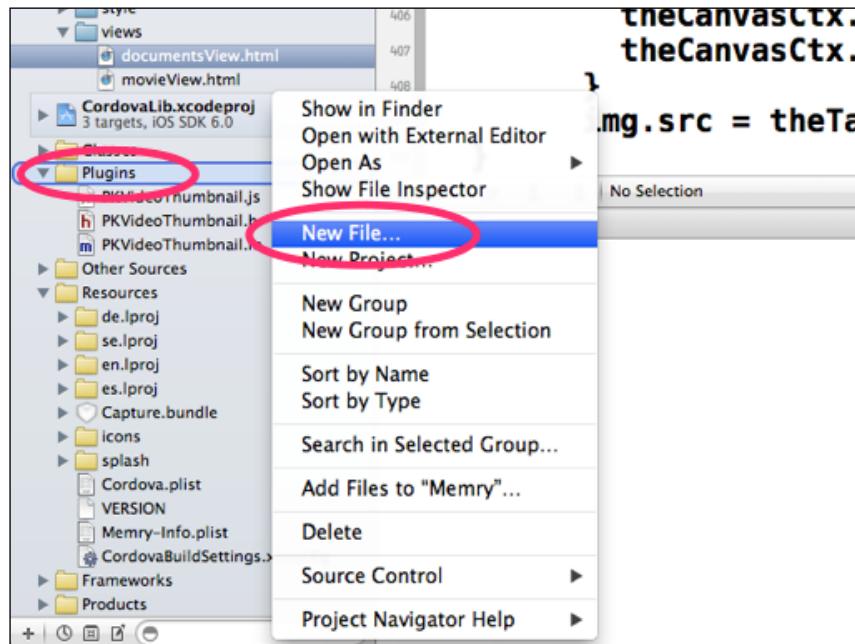
In this task, we modified the project settings so that the project knows about the plugin we're about to create. We also created the JavaScript interface for both iOS and Android that will permit us to communicate with the native code.

Implementing the video thumbnail plugin for iOS

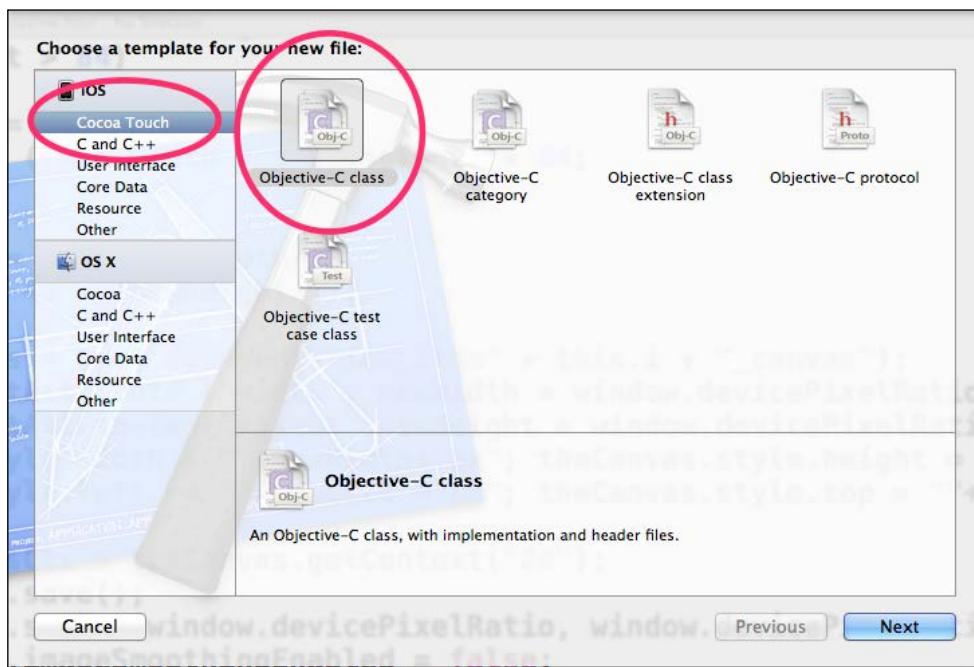
The iOS version of the video thumbnail will use a *hidden* video player to construct the thumbnail. Technically, we could use another library, but the video player is so convenient and fast when it comes to building images from videos. We hide it to ensure that the user will never actually see what's going on.

Getting ready

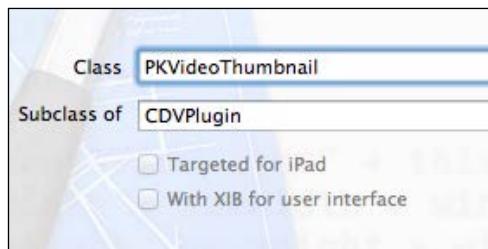
To start, let's create a new **Objective-C class** within Xcode. The easiest way is to right-click on the **Plugins** folder and select **New File...**, as seen in the following screenshot:



It's important that you use the **Plugins** folder that is a sibling to the **Other Sources** and **Resources** directories; not the one within the **www** directory.



Next, ensure the **Cocoa Touch** category is selected and then select the **Objective-C class** icon. Click on **Next**.



Then give the class a name, in this case `PKVideoThumbnail`, and ensure that it is a subclass of `CDVPlugin`. Click on **Next** again, and then you'll be prompted to verify where you want to save the files. It should be set to the **Plugins** folder; if not, be sure to navigate there before completing the task.

Getting on with it

Now that we have the .h and .m files, we need to fill them in. Let's start with the .h file first. The following is the *interface* or *specification* for the class:

```
#import <Cordova/CDVPlugin.h>

@interface PKVideoThumbnail : CDVPlugin

#if CORDOVA_VERSION_MIN_REQUIRED <= __CORDOVA_2_0_0
- (void) createThumbnail:(NSMutableArray*)arguments
withDict:(NSMutableDictionary*)options;
#else
- (void) createThumbnail:(CDVInvokedUrlCommand*)command;
#endif
@end
```

What we're doing in the preceding code is defining a method called `createThumbnail` that will be executed whenever we call `PKVideoThumbnail.createThumbnail()` from JavaScript.

Take note of the `#if...#else...#endif` construct; PhoneGap changed their method signature between versions 2.0 and 2.1, and so this handles both variations.

So far, though, we've not written any substantive code yet; we've been *declaring* or *defining* thus far. Let's change that by working on the .m file as shown in the following code snippet:

```
#import "PKVideoThumbnail.h"
#import <Cordova/CDVPluginResult.h>
#import <MediaPlayer/MediaPlayer.h>
```

First, we import several libraries that we need in order to construct our plugin. We also import our .h file as well; otherwise, the compiler would complain.

```
@implementation PKVideoThumbnail

BOOL extractVideoThumbnail ( NSString *theSourceVideoName,
                           NSString *theTargetImageName )
{
    UIImage *thumbnail;
```

Next, we define a method called `extractVideoThumbnail` that takes two parameters, namely, the path to the video and the location and name we should use when creating the image. Our method will return YES if we're successful, and NO if not. (This is the Objective-C Boolean equivalent to TRUE and FALSE.)

We also define a `thumbnail` of the type `UIImage`. The asterisk (*) indicates that this is a pointer – something very important in C-based languages. Essentially you'll use one whenever declaring a variable or parameter that's an object. When using numbers, you wouldn't use an asterisk, but in this case, we're only declaring one variable to start.

```
// BASED ON http://stackoverflow.com/a/6432050 //
MPMoviePlayerController *mp = [[MPMoviePlayerController
alloc]
initWithContentURL: [NSURL
URLWithString:theSourceVideoName] ];
mp.shouldAutoplay = NO;
mp.initialPlaybackTime = 1;
mp.currentPlaybackTime = 1;
```

The next thing we do is declare and create `MPMoviePlayerController`. It's an object, so it gets the asterisk as well. We'll name it `mp` for short.

We pass in the path to the video, which needs to be prefixed with `file://localhost`; remember that we do this in `PKVideoThumbnail.js`.

Then we set the playback time to 1 second in, and indicate that it shouldn't auto play. We just want a single image from the video, so we don't want to actually play it for the user yet.

```
thumbnail = [mp thumbnailImageAtTime:1
timeOption:MPMovieTimeOptionNearestKeyFrame];
[mp stop];
[mp release];
```

Next we ask for the image nearest the 1 second point in the movie. Since compression and encoding in movies uses key frames, we may not get the image at the exact 1 second mark, but it should be pretty close.

```
return [UIImageJPEGRepresentation ( thumbnail, 1.0)
writeToFile:theTargetImageName atomically:YES];
```

Finally, we save the thumbnail out to the desired file – our JavaScript will generally use the name of the movie and add a `.jpg` extension. The return value of the operation will either be YES or NO. If it is NO, the thumbnail wasn't successfully written.

```
}
```

```
#if CORDOVA_VERSION_MIN_REQUIRED <= __CORDOVA_2_0_0
```

Next up, we need to define the plugin handler for PhoneGap version 2.0 or lower:

```
- (void) createThumbnail:(NSMutableArray*) arguments  
withDict:(NSMutableDictionary*) options  
{  
    NSString* callbackId = [arguments objectAtIndex:0];  
    CDVPluginResult* pluginResult = nil;  
    NSString* javaScript = nil;
```

These three variables are always defined in plugins. They are critical to the functioning of the plugin. The first is a unique ID that PhoneGap uses to track calls between JavaScript and native code. The second is the result of our plugin's activities; we can use it to pass data back to JavaScript. The last one is the resulting JavaScript of the return code; this is used to call the success or failure routine.

```
@try {  
    NSString* theSourceVideoName = [arguments  
        objectAtIndex:1];  
    NSString* theTargetImageName = [arguments  
        objectAtIndex:2];
```

Next, we obtain the two parameters that should have been passed to the plugin.

```
if ( extractVideoThumbnail(theSourceVideoName,  
theTargetImageName) )  
{  
    pluginResult = [CDVPluginResult  
    resultWithStatus:CDVCommandStatus_OK  
    messageAsString:theTargetImageName] ;  
    javaScript = [pluginResult  
    toSuccessCallbackString:callbackId] ;  
}  
else  
{  
    pluginResult = [CDVPluginResult  
    resultWithStatus:CDVCommandStatus_ERROR  
    messageAsString:theTargetImageName] ;  
    javaScript = [pluginResult  
    toErrorCallbackString:callbackId] ;  
}
```

We call the `extractVideoThumbnail` method with the two parameters. As we said before, if it returns YES, then it worked, and so our plugin result will be an OK. If it returns NO, we'll return an error result instead.

```

} @catch (NSError* exception) {
    pluginResult = [CDVPluginResult
        resultWithStatus:CDVCommandStatus_JSON_EXCEPTION
        messageAsString:[exception reason]];
    javaScript = [pluginResult
        toErrorCallbackString:callbackId];
}

[self writeJavascript:javaScript];
}

```

The `@catch` block here is also important; it catches any errors that occur within the `@try` block. This might occur if something really went wrong or the wrong number (or type) of parameters were sent.

```

#else
- (void) createThumbnail: (CDVInvokedUrlCommand*) command
{
    CDVPluginResult* pluginResult = nil;
    NSString* javaScript = nil;

    @try {
        NSString* theSourceVideoName = [command.arguments
            objectAtIndex:0];
        NSString* theTargetImageName = [command.arguments
            objectAtIndex:1];

        if ( extractVideoThumbnail(theSourceVideoName,
            theTargetImageName) )
        {
            pluginResult = [CDVPluginResult
                resultWithStatus:CDVCommandStatus_OK
                messageAsString:theTargetImageName];
            javaScript = [pluginResult
                toSuccessCallbackString:command.callbackId];
        }
        else
        {
            pluginResult = [CDVPluginResult
                resultWithStatus:CDVCommandStatus_ERROR
                messageAsString:theTargetImageName];
            javaScript = [pluginResult
                toErrorCallbackString:command.callbackId];
        }
    }
}

```

Let's Go to the Movies! —

```
    } @catch (NSError* exception) {
        pluginResult = [CDVPluginResult
            resultWithStatus:CDVCommandStatus_JSON_EXCEPTION
            messageAsString:[exception reason]];
        javaScript = [pluginResult
            toErrorCallbackString:command.callbackId];
    }

    [self writeJavascript:javaScript];
}
#endif

@end
```

Finally, we essentially repeat ourselves, but using the PhoneGap 2.1 version of the plugin interface. Look closely, it does the same thing, but there are some subtle differences.

What did we do?

That's it! Now when we call `PKVideoThumbnail.createThumbnail()`, we'll be able to extract a thumbnail from any video we take or import. Cool, isn't it?

What else do I need to know?

Okay, so `extractVideoThumbnail()` isn't quite standard Objective-C style. Typically, one would write it:

```
BOOL extractThumbnailToFile: (NSString *)
    theTargetImageName fromVideoNamed: (NSString
    *)theSourceVideoName
```

And we would have called it like:

```
if (extractThumbnailToFile:theTargetImageName
    fromVideoNamed:theSourceVideoName) ...
```

But our way does the same thing and is a little less wordy. However, when working with Objective-C methods, it is important to recognize the differences in defining method signatures. If you're going to write a lot of Objective-C code, it's best to get used to the latter, but in a pinch, the former works too.

One last thing: what happens if we can't extract a thumbnail from the video? You'll notice that there's nothing in the code that appears to handle this possibility. Chances are that `thumbnail` will be `NULL` and that the attempt to write the thumbnail to storage will either return `NO` or raise an exception. Either way, we're covered with our code later on where we return `ERROR` if `NO` is returned.

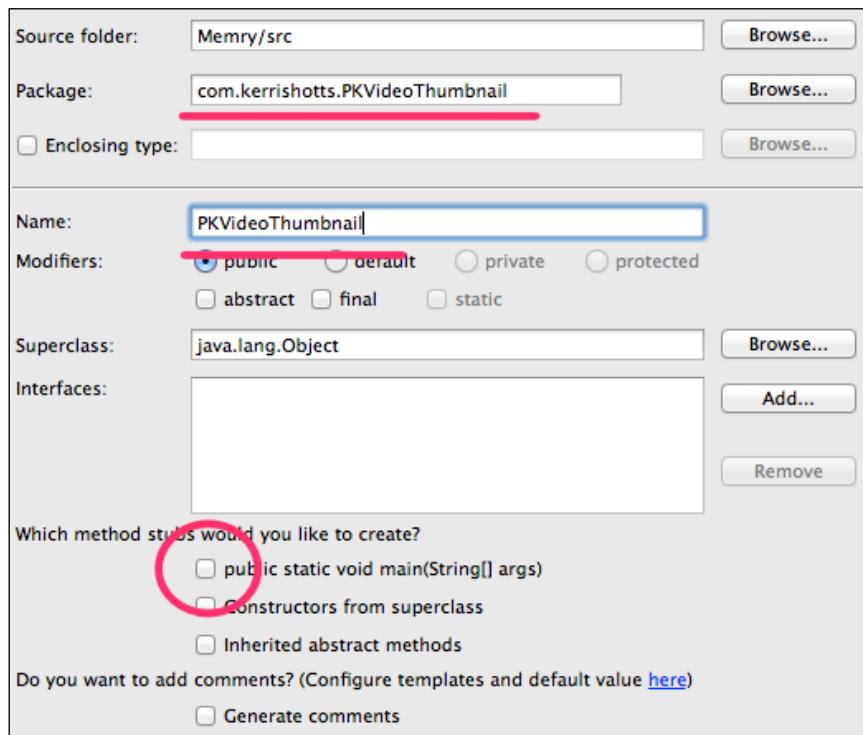
Implementing the video thumbnail plugin for Android

The Android version of the plugin is very similar to the iOS version, although it doesn't have to worry about the version of PhoneGap in use, so it is a little shorter. Ultimately, though, the steps are the same: grab a frame from a video, save it to storage, and return to JavaScript.

Getting ready

First off, create a new class by opening the **File** menu, selecting **New**, and then selecting **Class**.

Next, set the **Package** to `com.kerrishotts.PKVideoThumbnail`, the **Name** field of the class to `PKVideoThumbnail`, and then uncheck the first item under **Which method stubs would you like to create?** – we won't need any sample code. This is shown in the following screenshot:



Next, open the resulting file `PKVideoThumbnail.java` file, and we'll start writing the Android version.

Getting on with it

Unlike iOS, we only need one file, and it ends up being a bit shorter too, given as follows:

```
package com.kerrishotts.PKVideoThumbnail;

import org.apache.cordova.api.Plugin;
import org.apache.cordova.api.PluginResult;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

import android.graphics.Bitmap;
import android.graphics.Bitmap.CompressFormat;
import android.media.*;
import android.provider.MediaStore;

import java.io.*;
```

First, as in iOS, we import the libraries we'll need for our plugin to work.

```
public class PKVideoThumbnail extends Plugin {

    public PluginResult execute(String action, JSONArray args,
        String callbackId) {
```

Unlike iOS, we define a single method called `execute`. If our plugin had multiple actions, we'd need to handle each one within this `execute` method. In iOS, this is done for us.

```
    try {
        if (action.equals("createThumbnail")) {
```

Since we need to check if the incoming action is `createThumbnail`, we have the preceding code. Technically, we could avoid this for a plugin only performing one action, but it would be nonstandard to do so.

```
        String sourceVideo = args.getString(0);
        String targetImage = args.getString(1);
```

In the prior code, we define our two parameters that we pass in from JavaScript. Note the lack of any asterisks like we had in the iOS version. Nice not to have to worry about those pesky things, right?

```
        Bitmap thumbnail =
ThumbnailUtils.createVideoThumbnail (
sourceVideo, MediaStore.Images.Thumbnails.
MINI_KIND);
```

Creating a thumbnail from a video is really quite simple; there's already a simple routine pre-written for us. We just pass the path to the video, and ask for a specific size (in our case, MINI_KIND).

```
FileOutputStream theOutputStream;
try
{
    File theOutputFile = new File
        (targetImage.substring(7));
    if (!theOutputFile.exists())
    {
        if (!theOutputFile.createNewFile())
        {
            return new
                PluginResult(PluginResult.Status.ERROR,
                    "Could not save thumbnail.");
        }
    }
    if (theOutputFile.canWrite())
    {
        theOutputStream = new
            FileOutputStream (theOutputFile);
        if (theOutputStream != null)
        {
            thumbnail.compress
                (CompressFormat.JPEG, 75,
                    theOutputStream);
        }
        else
        {
            return new PluginResult
                (PluginResult.Status.ERROR, "Could not
                    save thumbnail; target not
                    writeable.");
        }
    }
}
catch (IOException e)
{
    e.printStackTrace();
    return new PluginResult(PluginResult.Status.
        IO_EXCEPTION, "I/O exception saving
        thumbnail.");
}
```

Actually saving the thumbnail to storage is a bit more involved. We need to check to see if we should create the file first (by checking if it doesn't exist first), and then if we can write to the file. Once we do that, we can use the `thumbnail.compress()` method to do the actual work of saving the file. All the other stuff is there to handle errors and such, and Java requires that you handle them. If not, the code will fail to compile.

```
        return new PluginResult
        (PluginResult.Status.OK, targetImage );
```

At this point, if we're executing this bit of code, the thumbnail has been created successfully, and so we return `OK`. If we aren't here, we've returned `ERROR` or `IO_EXCEPTION` or even something else, as listed in the following code snippet:

```
    } else {
        return new PluginResult(PluginResult.
        Status.INVALID_ACTION);
    }
} catch (JSONException e) {
    return new PluginResult(PluginResult.
    Status.JSON_EXCEPTION);
}
}
```

What did we do?

That's all! We've extracted a thumbnail from a video file and saved it to a JPEG file for our JavaScript to work with.

What else do I need to know?

Ok, so if you actually take a close look at what gets saved out of this code, you'll notice it really isn't very much of a thumbnail. In fact it's a image that's the same size as the video resolution. That's not too big a deal, though, since we'll shrink it down in our JavaScript, but I wanted to let you know.

Integrating with the video thumbnail plugin

Next, we need to actually make the changes to our code so that we can display the video thumbnails. Open up the `documentsView.html` file under `www/views` so that you can follow along.

Getting ready

The theory behind displaying the thumbnails is much the same as displaying the thumbnails in our last project; that is, we're still using the `canvas` tag to speed up the feel of our app, and we're still generating thumbnails from JPEG files. The difference is that we have to generate those JPEG files from the video file.

Getting on with it

Let's start by taking a look at the `documentIterator()` method in the following code snippet:

```
documentsView.documentIterator = function(o)
{
    var theHTML = "";
    var theNumberOfDocuments = 0;
    documentsView.documentToIndex = {};
    for (var i = 0; i < o.getDocumentCount(); i++)
    {
        var theDocumentEntry = o.getDocumentAtIndex(i);

        theHTML += PKUTIL.instanceOfTemplate($ge
            ("documentsView_documentTemplate"),
            { "src" : theDocumentEntry.fullPath,
              "index" : i
            });

        documentsView.documentToIndex[
            PKUTIL.FILE.getFileNamePart
            (theDocumentEntry.fullPath) ] = i;
        theNumberOfDocuments++;
    }
    $ge("documentsView_contentArea").innerHTML = theHTML;
```

Let's Go to the Movies!

When compared to our previous project, the content is pretty similar so far. The only difference is the highlighted line. We have defined a variable earlier in the file named `documentToIndex`, which is an object. We're using it as an associative array, however, so that we can later map the file back to its index. For example, if file `1239548.mov` is the third item in our document list, we'd store 3 in the space for `1239548.mov`.

Next, as before, we wait for 100 milliseconds before attaching the long press handler and such as seen in the following code snippet:

```
PKUTIL.delay(100, function()
{
    for (var i = 0; i < theNumberOfDocuments; i++)
    {
        var theDocumentEntry = o.getDocumentAtIndex(i);
        var theElement = $ge("documentsView_item" + i + "");
        var theLPGesture = new
GESTURES.LongPressGesture(theElement, function(o)
{
    documentsView.longPressReceived(o.data);
});
theLPGesture.data = i;

PKVideoThumbnail.createThumbnail (
theDocumentEntry.fullPath,
PKUTIL.FILE.getPathPart ( theDocumentEntry.fullPath
) + PKUTIL.FILE.getFileNamePart (
theDocumentEntry.fullPath ) + ".jpg",
documentsView.renderVideoThumbnail,
function ( theError )
{
    console.log ( JSON.stringify ( theError ) );
}
)
```

This last portion is where we ask our new plugin to do the work of extracting the video thumbnail. We know the video's path in persistent storage, so we can pass that part along. We can also construct the filename for the JPEG as well (what we're doing is taking everything but the video's extension and substituting `.jpg` instead). Then when the video thumbnail has been generated successfully, `renderVideoThumbnail()` will be called. If an error occurs, we'll log it to the console.

```
}
```

252

```
});
```

```
documentsView.renderVideoThumbnail = function (
theTargetImage )
{
```

```
var img = new Image();
var i = documentsView.documentToIndex[
    PKUTIL.FILE.getFileNamePart ( theTargetImage ) ];
```

The `renderVideoThumbnail()` method as a whole is very similar to the remainder of the code inside the last project's `documentIterator()`. We've broken it out to make it a little easier to read, but otherwise it does the same thing. The only difference between the two is that we have to figure out the index – which image are we talking about. If you remember the variable we defined earlier, `documentToIndex[]`, we can figure out the index of the image from the filename, which is what we do in the preceding code snippet. From there on the code is identical, and we won't list the rest here.

What did we do?

In this task, we modified the `documentIterator()` method to work with our new plugin. We've asked for a thumbnail from a video, and we've managed to display it back to the end user when needed.

Implementing recording and importing of video

We've done the first part of our app, which was displaying thumbnails from videos, but we've got to actually record them before we can get anything into our app. In this task, we'll do exactly that—record a new video.

Getting ready

We'll be working in the `documentsView.html` file in the `www/views` directory, if you want to follow along.

Getting on with it

You might think that we'd use the camera code from our previous project, and you'd be partially correct. For iOS, we can indeed use virtually the exact same code to import new videos, but for any platform to record video, we have to use a new API—the `CAPTURE` API.

Let's take a look at the code for `takeMovie()`:

```
documentsView.takeMovie = function()
{
    navigator.device.capture.captureVideo(
```

Let's Go to the Movies!

The CAPTURE API provides more methods than simply capturing video; you can capture audio as well (which is similar to using the MEDIA API in *Project 5, Talking to Your App*). In our case, we use the `captureVideo()` method. It takes three parameters: the `success` function, the `failure` function, and any options we want to pass along. In our case, the only option is that we will limit the user to one video at a time. Technically, the API will allow more than one video in a session, but for our purposes, one at a time simplifies things.

```
function (mediaFiles)
{
    var uri = mediaFiles[0].fullPath;
    var fileExt = PKUTIL.FILE.
        getFileExtensionPart ( uri );
    PKFILE.moveTo ( uri,
        "doc://" + PKUTIL.getUnixTime() + "." + fileExt,
        function ()
        {
            documentsView.reloadAvailableDocuments();
        },
        function (evt)
        {
            console.log (JSON.stringify(evt));
            var anAlert = new PKUI.MESSAGE.Alert
                (_T("Oops!"), _T("Failed to save the video."));
            anAlert.show();
        }
    );
}
```

The preceding code shows the `success` method, which will be called with a list of files. In our case, it will only have one filename, which we get by using the zeroth index of the list. From that point forward, it's nearly identical to the way we copy a file from temporary storage to permanent storage in *Project 6, Say Cheese!*. The only difference is that we don't assume that the file extension will be `.jpg`. Videos on different platforms can often have very different extensions.

```
function (error)
{
    var msg = 'An error occurred during capture: ' +
        error.code;
    var anAlert = new PKUI.MESSAGE.
        Alert(_T("Oops!"), msg);
    anAlert.show();
},
```

Next up is the `failure` function, where all we'll worry about is letting the user know the error code, but if you wanted, you could give a much better error message based on the code.

```
{limit: 1});  
}
```

Finally, the third parameter is that set of options. In this case, we only want one video at a time, but there are other options one could pass, such as the video encoding type. Different platforms support these other options differently, so we won't dwell on them here, but they are available in the PhoneGap API documentation should you need them. (http://docs.phonegap.com/en/edge/cordova_media_capture_capture.md.html#Capture)

For importing video, we can use nearly the same code from the previous project, but we'll put it all in `importMovie()`:

```
documentsView.importMovie = function()  
{  
    navigator.camera.getPicture ( function (uri)  
    {  
        var fileExt = PKUTIL.FILE.getFileExtensionPart ( uri );  
        PKFILE.moveFileTo ( uri, "doc://" + PKUTIL.getUnixTime()  
        + "." + fileExt, function ()  
        {  
            documentsView.reloadAvailableDocuments ();  
        },  
        function (evt)  
        {  
            console.log (JSON.stringify(evt));  
            var anAlert = new PKUI.MESSAGE.  
            Alert (__T("Oops!"), __T("Failed to save  
            the video."));  
            anAlert.show();  
        } )  
    },  
    function (msg)  
    {  
        var anAlert = new PKUI.MESSAGE.Alert  
        (__T("Oops!"), msg);  
        anAlert.show();  
    },  
    { destinationType: Camera.DestinationType.FILE_URI,  
    sourceType: Camera.PictureSourceType.PHOTOLIBRARY,  
    mediaType: Camera.MediaType.VIDEO,  
    saveToPhotoAlbum: false  
    }  
};  
}
```

Let's Go to the Movies! —

Most of this code is identical to the code in the previous project for `doPicture()`. The only real differences are the handling of the file extension and the `mediaType` option. Notice we pass `Camera.MediaType.VIDEO` instead. This ensures we will only get videos in return.

One catch: this doesn't seem to work well on Android. We haven't disabled it in the app (just in case you have better luck), but you may wish to disable the import functionality in your Android apps. On iOS, however, it works quite well.

What did we do?

In this task, we implemented the code to record video and import video.

Implementing video playback

Playing videos is a very important feature, especially if we're recording it, right? But even for apps that don't support video recording, video playback can be essential. Consider an e-learning type of application; reading about the subject can work well for a lot of people, but actually seeing the subject in action can help even more. Video would make an excellent platform for this type of learning.

Getting ready

We'll be working in `documentsView.html` and `movieView.html` in the `www/views` directory if you want to follow along.

Getting on with it

First off, there's a catch. Playing video happens to be really easy on iOS devices. Playing video on other devices, well, not so much.

For iOS, we'll use the movie view, which is akin to the image view in our previous project. Most of the code is duplicated, so we'll just talk about the changes here.

The template portion of the movie view looks like the following:

```
<div id="movieView_documentTemplate" class="hidden">
    <video src="%SRC%" controls autoplay autobuffer
        style="width:100%; height: 100%;" />
</div>
```

Whereas, in the previous project we used an `IMG` tag to display the image, we are now using a `VIDEO` tag to display the video. This is a feature of HTML 5 that iOS supports very nicely, and as such, it makes it very easy for us to support video playback.

The tag will have %SRC% replaced with the video filename when the documents view calls `setMovie()`, which looks like the following:

```
movieView.setMovie = function ( moviePath, movieIndex )  
{  
    movieView.moviePath = moviePath;  
    movieView.movieIndex = movieIndex;  
  
    $ge("movieView_contentArea").innerHTML =  
        PKUTIL.instanceOfTemplate($ge  
        ("movieView_documentTemplate"),  
        { "src" : movieView.moviePath,  
          "thumb": PKUTIL.FILE.getPathPart ( moviePath ) +  
                  PKUTIL.FILE.getFileNamePart ( moviePath ) +  
                  ".jpg" } );  
}
```

It's essentially the same as `setImage()` in the prior project, though we have introduced a `thumb` portion, which you could use if you wanted to display a small thumbnail that the user had to click on prior to playing the movie.

Once the movie view is pushed (by the documents view), and `setMovie()` is called, the `VIDEO` tag in the preceding code will cause the video to start playing instantly on iOS devices. Chances are good that the video will also fill the entire screen, something common on all mobile platforms. Tablets will generally permit inline video, but smaller form factors usually attempt to play video in full screen mode.

So, as we already said, iOS is easy: HTML 5 video is properly supported, and we can display it simply with a minimum of fuss.

Oh, if the other platforms were only as nice. Android, for example, claims to support the `VIDEO` tag, but its implementation is so horribly broken that chances are slim you'll find a device on which it will actually work. Yes, the controls will display, but that's about it.

What do we do instead? We use another plugin, this time written by Simon MacDonald who decided to help the PhoneGap community out by providing a simple video player plugin. (<http://simonmacdonald.blogspot.com/2011/11/video-player-plugin-for-phonegap.html>)

First, you need to install the plugin into your Java project using the following steps:

1. Copy the `src` directory from the `Android/VideoPlayer` directory and add it to your project. Make sure the contents are included; deep inside this directory is a file called `VideoPlayer.java` that you need in your project.
2. Copy the `video.js` file in the `Android/VideoPlayer/www` directory in the plugin package to `www/plugins/Android` in your project.
3. Add the following line to your `index_android.html` file:

```
<script type="application/javascript"
charset="utf-8" src=".//plugins/Android/video.js">
</script>
```

4. Add the following line to your `/res/xml/config.xml` file:

```
<plugin name="VideoPlayer" value="com.
phonegap.plugins.video.VideoPlayer"/>
```

5. Next, we'll alter our code in `documentsView.html` to play video using this player if we're an Android device. We'll be doing this in the `documentContainerTapped()` method:

```
documentsView.documentContainerTapped = function(idx)
{
    var theElement = $ge("documentsView_item"
+ idx + "_canvas");
    if (documentsView.inSelectionMode)
    {
        ... this code is identical to the previous chapter ...
    }
    else
    {
        if ( PKDEVICE.platform() != "android" )
        {
            PKUI.CORE.pushView (movieView);
            PKUTIL.delay(500, function()
            {
                movieView.setMovie (
                    documentsView.availableDocuments.
                    getDocumentAtIndex(idx).fullPath, idx );
            } );
        }
    }
}
```

```
        else
        {
            window.plugins.videoPlayer.play(
                documentsView.availableDocuments.
                getDocumentAtIndex(idx).fullPath );
        }
    }
}
```

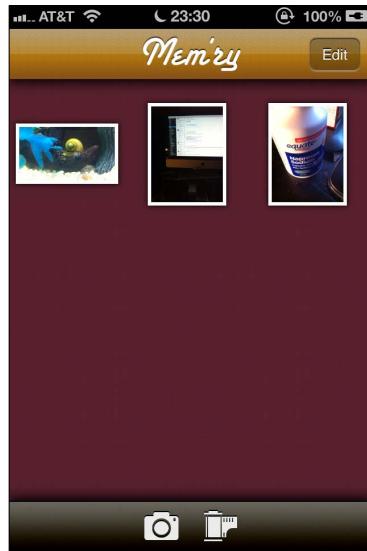
If we're any platform but Android, we'll attempt to use the movie view to play the video, but if we are on Android, we'll use the highlighted code in the preceding code snippet, which asks the video player plugin to play the desired video. When asked, the video will be played immediately, and in full screen. This does mean that Android devices won't ever display the movie view, but the actions available from within the view (*delete* and *share*) are also available from the documents view, so this isn't a big loss.

What did we do?

In this task we used the HTML 5 `VIDEO` tag to play video on devices that support it, and we also learned how to use the video player plugin created by Simon MacDonald to play video on Android devices.

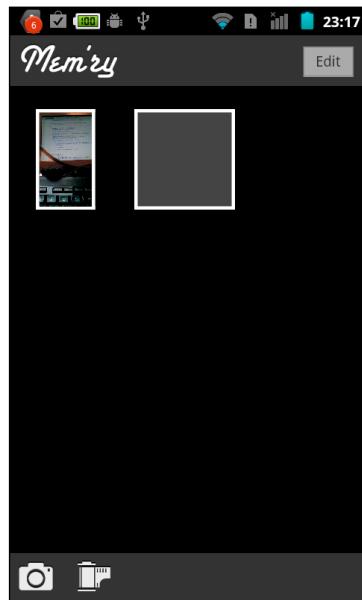
Game Over..... Wrapping it up

Let's see what we finally ended up with; first for iOS the view will be as follows:



Let's Go to the Movies! —

For Android the view will be as follows:



If you look at our final app, it's very similar to our previous project's app. We've changed some of the graphics a little bit, yes, but visually it's nearly identical, and from the code's perspective, it's nearly identical. We've swapped out the bits that dealt solely with images and replaced them with bits that work with video instead. We've worked with HTML 5 VIDEO and also written our own plugins using native code. You should now be able to take what you've learned and apply it to your own apps in order to record and play video files.

Can you take the HEAT? The Hotshot Challenge

There are several ways that you could further improve this app. How about challenging yourself to a few?

- ▶ Add the ability to upload the video files to a social network via the **Share** button.
- ▶ Allow the user to record more than one video at once, and process each one accordingly.
- ▶ Change the whole app around into an e-learning style app where the videos aren't recorded by the user, but baked into the app. Then allow the user to watch your videos so that they can learn about a particular subject.

Project 8

Playing Around

Smartphones have been no strangers to fun little games that helped pass the time. From the seemingly eternally-existing Solitaire to Snake!, Tetris, or Pop-the-Bubble variants, we've found ways to pass the time with our mobile devices. Even if you nearly always write productive applications, sooner or later, the *bug* to write a game is likely to bite.

What do we build?

In this project, we're going to put together a game called *Cave Runner*. Okay, it won't win any prizes based on the originality of the game (or the title), nor will it win *Best Game of the Year*. But it's amusing, and has a lot of potential to expand in various ways, and so serves as a good base, especially for the quick and diverting category that many games try to fit into.

What does it do?

To accomplish this, we're going to be relying heavily upon the HTML5 Canvas, which is quite literally the only way we're going to achieve anything even approaching 60 fps (the target for most games). Even so, only recent and powerful devices are going to meet this target, and so we also will need to sludge around in the mathematics around how to create a game that isn't reliant upon its frame rate. If the game's timing relied solely on the frame rate, 30 fps would feel as if we're sludging through mud, that is, the game would feel like it was progressing in slow motion. Instead, we have to act like we're running at 60 fps, even if we can't display that many frames, so that we avoid this effect.

While controlling a game character on a console, portable game machine, or PC is pretty obvious (keyboard, mouse, D-pad, Joystick, and so on), how does one control a game character on a mobile device which probably has none of those features? There are two answers: use the multi-touch screen, which can be used to simulate a Joystick or D-pad, or use the device's built-in accelerometer. We'll talk about using both in this task.

Which brings us to the last big thing; it's not so hard to implement, but it is absolutely critical to have in place—**persistent settings**. If we're going to provide two methods of control, we need a way to save which method the user prefers. While we have used the `File API` in previous projects to store persistent content, we're going to use `localStorage` this time. After all, we're only storing a simple flag, not a lot of user-generated content.

Why is it great?

Hopefully, you'll have a bit of fun with the game as it stands now, but even as a simple game, it introduces you to the concepts you'll need to create complex games further down the road. We'll work on keeping the game going at the same speed, regardless of frame rate. We'll talk about persistent settings using `localStorage`. We'll also work out how to control the game using the touch screen and the accelerometer. All of these things combine to create a good game, and you should have a good base from which to build on for any future endeavors.

How are we going to do it?

We're going to approach this much like we have the prior projects:

- ▶ Designing the game
- ▶ Implementing the options view
- ▶ Generating levels
- ▶ Drawing to the canvas
- ▶ Keeping up
- ▶ Performing updates
- ▶ Handling touch-based input
- ▶ Handling the accelerometer

What do I need to get started?

Go ahead and create your project, or use the project in the code package for this book as a start. You'll want to use the images in `www/images`. If you want some insight on how we designed the graphic assets, feel free to look in the `/resources` directory in the code package for this project.

In general, we'll be talking more about the code that is already written than spelling the code out verbatim. As such, it would be a good idea to have the project downloaded so that you have the code as a reference. Go ahead and compile it for your device, too, and play with it, to get a better feel for what we'll be talking about.

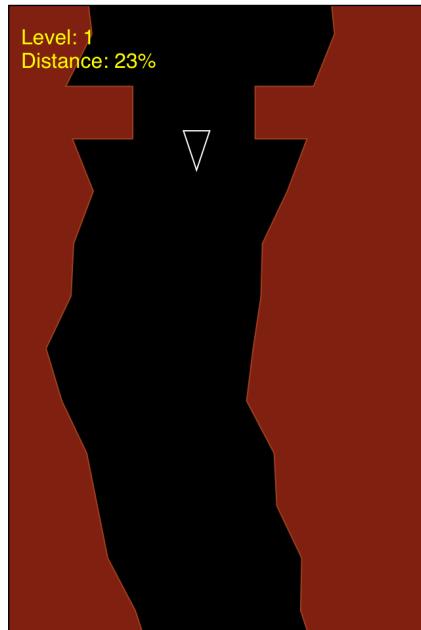
Designing the game

Where in previous projects we'd be developing the user interface and the interactions between the various widgets and views, we'll be designing how our game looks and acts instead. While similar, there's often a lot more that has to go into designing games (graphic assets, level design, character design, animation, and so on). Unfortunately, we can't go over everything given the length of the project, but we can give you a good start.

Getting on with it

The primary theme of the game may already be evident from the title, Cave Runner. These kinds of games have been around since the first computers, even if the graphical quality was a tad bit coarse. In short, we're going to develop a game that has a series of levels through which the player (who controls a ship) has to navigate safely in order to advance. Each level will be more difficult than the previous, and in our particular version, as long as the player can keep up, there's no end to the levels. In all practicality, there will be a point where the player can't navigate safely through a given level, and so the game always ends with a crash. Think of it as an endurance run where we already know the outcome – it's the journey that's the point.

The level consists of a cave-like structure with walls on both sides of the screen. These walls are irregular and random and together form a safe path for the ship. If the ship touches the edges, the game is over.



Playing Around

To make things a little more difficult, there are obstacles that get in the ship's way. In the first levels, they don't appear very often, but as the levels get harder, the obstacles appear more often. The obstacle looks like a wall with an opening cut out, and the ship must pass within the opening in order to be safe.

Our levels will be randomly generated according to certain parameters in order to create an ever-changing landscape. Even though our levels are random, you could just easily create static levels and load them in wherever necessary, something which we suggest at the end of the project.

Our ship will be very simple: a triangle. Yes, one can get a lot more complicated with animation and such, but for the simple visual style of our game, it works well for our needs.

In order to move the ship, the player has two options: touch or swipe the screen in order to control the ship, or tilt the device. The ship will move according to the direction of the swipe or the tilt, that is, tilting or swiping left will move the ship left, and vice versa. Since we're calling our character a *ship*, we're intentionally introducing some fuzzy mechanics to the movement. In other words, the ship doesn't respond instantly, nor does it stop instantly. Think of the ship as if it has thrusters on it.

Of course, we could have decided that the position of the ship was directly related to the position of the finger on the screen or the degree of the tilt, and for some games this would be appropriate. It is always important to recognize that you should tailor your control mechanism to your game and use what makes sense.

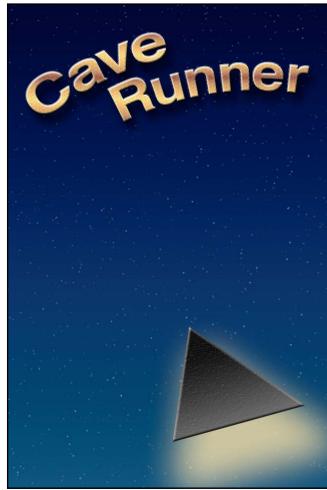
Our game itself will be contained within one view—the game view. Outside of the game will live the start view and an options view. The start view contains two buttons, namely, **Play** and **Options**. Tapping on **Play** will switch to the game view, while the **Options** button will switch to the options view.

The options view gives two iconic representations of controlling movement: one for tilting the device and one for sliding a finger across the screen. Tapping either of these elements will select that method as the control method. An additional **Back** button lets the user get back to the start view.

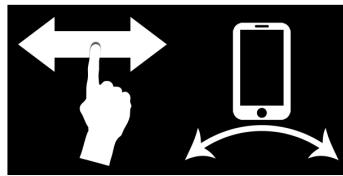
Inside the game view we have several items that need to be displayed. Of course, the level and the ship are required, but games often display other information as well. In our case, we'll display the current level and the distance travelled through the level. Should we need to display a message (such as ***Crash!*** or **Level Complete**) we'll show it in the middle of the screen along with two buttons: one to restart or continue, depending on the situation, and one to go back to the game view.

And that's it, really; it's not a complicated game, and yet it can provide a base for more complicated endeavors in the future.

Let's have a quick look at our graphical assets before we wrap this task up. Our splash screen looks like this:



Our control icons will look like this:



Our buttons that we use throughout the game won't require any graphical assets. We'll just use a rounded rectangle with a border and shadow, which we can accomplish with CSS.

What did we do?

In this task we designed our game mechanics and assets. We've figured out the views we need as well.

What else do I need to know?

By no means is game design this simple. We can quickly go through this particular game partly because it is both simple and the mechanism is also well known. Even slightly more complex games will require a good deal of time to design, and it is best to do so before even writing a single line of code. Figure out your visual style, your sound style, the mechanics of the game, control mechanisms, levels, and animations. All of this will take time and lots of paper.

Implementing the options view

The start view is a simple view that we're not going to dwell on; feel free to look at the code in `www/views/startView.html`. In this task we'll focus on the options view located in `www/views/optionsView.html`. It's only moderately more complex than the start view, so some of the code is very similar.

When done, we'll have something that looks like the following screenshot:



Getting on with it

Let's take a look at the HTML for the view first:

```
<div class="viewBackground">
    <div id="optionsView_contentArea" style="padding: 0;
        height: auto; position: relative;"></div>
</div>
```

The first portion is very simple; the actual content is in a template that we'll process for localization:

```
<div id="optionsView_actions" class="hidden">
    <div id="optionsView_changeControls">
        <div id="optionsView_tilt"
            ontouchend="optionsView.selectTilt();">
```

```
%TILT%
</div>
<div id="optionsView_slide"
ontouchend="optionsView.selectSlide();">
    %SLIDE%
</div>
</div>
<div id="optionsView_backButton"
ontouchend="PKUI.CORE.popView();" >%BACK%</div>
</div>
```

In this part of the code, we've defined two methods of control. Touching either the `optionsView_tilt` or `optionsView_slide` icon will call the method to select that control method. The **Back** button will pop the view and return to the start view.

Note that we've not defined any styles here; the styling lives in `www/styles/style.css`. Here's the styling we're using for our view:

```
#startView_contentArea,
#optionsView_contentArea
{
    background-image: url('..//images/splash.png') !important;
    height: 100% !important;
}
```

We're using a background image for our start and options view (but not the game view), which we define in the preceding code. This puts a nice image behind our controls, so we need to be careful where we place the buttons so as to not overwrite any critical text or graphical elements.

Each button is styled as follows:

```
...
#optionsView_backButton,
...
{
    position: absolute;
    left: 50px;
    width: 200px;
    background-color: #8BF;
    height: 2em;
    font-size: 24pt;
    line-height: 1.75em;
    text-align: center;
```

Playing Around

```
color: white;
border: 4px solid white;
border-radius: 1em;
text-shadow: 0 1px 1px #000;
box-shadow: 0px 10px 20px #000, 1px 1px 1px #000 inset;
-webkit-transform: rotate(-12deg);
}
```

This gives us a nice rounded button with bright colors as seen in the screenshot heading this section. It's also slightly rotated off-kilter, something games can usually get away with. Doing this in a productivity application wouldn't be all that good an idea.

```
#optionsView_backButton
{
    top: 300px;
    left: 32px;
    -webkit-transform: rotate(12deg);
}
```

For each button, we have to specify the location of the button, and if we want, we also override the rotation to have buttons in different rotations on the screen.

```
#optionsView_changeControls
{
    position: absolute;
    top: 120px;
    left: 32px;
    width: 256px;
}
```

For our control selection, we first indicate where the icons will live, in the preceding code snippet, and then specify each one's properties as follows:

```
#optionsView_tilt,
#optionsView_slide
{
    width: 128px;
    text-align: center;
    font-family: "Bradley Hand", sans-serif;
    font-size: 24pt;
    color: #FFFF80;
    text-shadow: 0px -1px 1px #000;
    height: 160px;
    float: left;
    color: #FFFFFF;
}
```

The font, in this case, gives us a handwritten feel. Note that we provide a fallback in case the device doesn't support this font, which is likely on Android devices. All recent iOS devices provide this font automatically.

```
#optionsView_tilt.selected,  
#optionsView_slide.selected  
{  
    color: #FFFF80;  
    background-color: rgba(255,255,255,0.25);  
    border-radius: 25px;  
}
```

When selected, we change the color of the text and make the selected item appear to be highlighted.

Let's take a look at the code; it's really pretty simple:

```
var optionsView = $ge("optionsView") || {};  
  
optionsView.initializeView = function()  
{  
  
    $ge("optionsView_contentArea").innerHTML =  
  
        PKUTIL.instanceOfTemplate ( $ge("optionsView_actions") ,  
        { "tilt": __T("TILT") ,  
        "slide": __T("SLIDE") ,  
        "back": __T("BACK")  
        }  
    );  
  
    optionsView.displayControlSetting();  
  
}
```

As always, our `initializeView` method is used to set the view up and perform the necessary localization. It also calls `displayControlSetting`, which will highlight the appropriate control method:

```
optionsView.displayControlSetting = function ()  
{  
    $ge("optionsView_slide").className="";  
    $ge("optionsView_tilt").className="";  
    if (localStorage.controlSetting)
```

```
{  
    // use the saved setting  
    if (localStorage.controlSetting == "slide")  
    {  
        $ge("optionsView_slide").className="selected";  
    }  
    else  
    {  
        $ge("optionsView_tilt").className="selected";  
    }  
}  
else  
{  
    // default to the slide control option  
    $ge("optionsView_slide").className="selected";  
}
```

Notice that we're using `localStorage` here; it's almost so subtle you could miss it. First we check for the existence of our desired property (`controlSetting`). If it doesn't exist, we default to the `slide` control method. If it does exist, we'll use whichever value is stored in the property.

While `localStorage` isn't guaranteed to be 100 percent persistent (iOS has the option to delete it should the device's space get low), it's good enough for these kinds of settings.

When the user taps one of the control methods, we set `localStorage.controlSetting` as follows:

```
optionsView.selectTilt = function ()  
{  
    localStorage.controlSetting = "tilt";  
    optionsView.displayControlSetting();  
}  
  
optionsView.selectSlide = function ()  
{  
    localStorage.controlSetting = "slide";  
    optionsView.displayControlSetting();  
}
```

From this point on, the code is like we've used before, so we won't reprint it here.

What did we do?

In this task, we created our options view. We created the ability for the user to select either a *tilt* or *swipe* control method for their character, and we have used `localStorage` to both save and read back the user's preference.

What else do I need to know?

What if the user sees this screen and decides that *swipe* is the control method they want? This means we never set a property within `localStorage`. This means that when the game starts, how will it know which control method to use?

Simple, we'll do a check there too. If there's nothing in `localStorage`, we'll assume the user wants to use the *swipe* method. The key here is to be consistent; if the game decided to use the *tilt* method instead, but displayed the *swipe* as the default in our options view, the player would obviously be confused as to which option means what.

Generating levels

It's hard to imagine a game without at least one level, and that level needs to have some sort of content in it. In this task, we'll examine how to generate content for the levels in our game.

Getting ready

Open the `gameView.html` file in `www/views`. We'll be using this file quite a bit, so it would be a good idea to have it open for reference.

Getting on with it

There are a few ways one can generate a level. One can use random content, pseudo-random content, or static content. The first is pretty easy: just use random numbers for everything. Unfortunately, this doesn't usually result in terribly nice levels, and there's little guarantee of winnability or difficulty.

The third method is also pretty easy: use static content. This means that you've determined the entire level ahead of time, and stored it in a file. When the game requests the level, it can be read back. This means it is the same every time, which can be good (or bad), depending on the game, but it also means that you have a clear way to ensure both winnability and difficulty. For puzzle games, this is nearly always the method one uses.

Playing Around

Our method is to be pseudo-random. We'll be using plenty of random numbers; we don't want perfectly straight cave walls or easily-guessed paths. But we also want to build in some level of increase in difficulty over time, as well as restrict the levels to a few parameters to help ensure (though not guarantee) winnability. It is possible to guarantee a level can be winnable with enough code, but we won't go quite that far in this game.

Let's go through the code used to generate a level using the following code snippet:

```
function generateLevel ( lvl )
{
    points = new Array();
    points[0] = new Array();
    points[1] = new Array();
    points[2] = new Array();
    points[3] = new Array();
```

First we initialize our `points []` array, and then four arrays within it. The first two arrays contain the left and right points that make up the cave wall. The last two contain the edges of any obstacle opening, or `-1` if there is no obstacle in place.

```
...
var lastLeft= (cWidth/5) ;
var lastRight=(cWidth/5) ;
```

Next, we start to define some variables that we'll be using to control our level. These two variables store the last points generated for the cave, but we need to have something to start from as well. `cWidth` is defined earlier in the file as being the width of the screen, so you can see how this would generate an open area in the middle of the screen at the beginning of our level. This is important, since we don't want to surprise the player with an immediate obstacle they can't avoid.

```
var bias = 0;
```

`bias` controls the direction our cave walls will tend towards. They'll still be randomly generated, but we introduce `bias` whenever the walls hit the edge of the screen so that there is always some movement.

```
var rndWidth = Math.floor(cWidth/ 10) + (lvl*10);
```

`rndWidth` controls how much our cave wall can vary over a particular distance. In this case, it is controlled by the width of the screen and our current level. This means the cave gets harder to navigate as we progress through our levels.

```
var channelWidth = Math.floor(cWidth / 2.25) - (lvl*16);
```

`channelWidth`, on the other hand, limits just how close the cave walls can get. It's also based on the width of the screen and the level. You'll notice that at some high level, the channel will be too small to permit passage of the ship. At this point, the game can be considered over, or one could also build in a way to prevent this value from ever getting too small.

```
var wallChance = 0.75 - (lvl/25);
if (wallChance < 0.15) { wallChance = 0.15; };
```

An obstacle, or wall, is only generated every so often; we don't want an obstacle to appear at every point. So we generate some sort of chance that is also based on the level. Easier levels will have fewer obstacles, while harder levels will have several.

```
var wallEvery = Math.floor(30 - (lvl/2));
if (wallEvery < 10) { wallEvery = 10; };
```

`wallEvery` also factors in how often obstacles appear, but in a different way. It controls how many points must be between an obstacle before it can have a chance to be generated. In this case, we'll start off at 29 points, but will steadily lower it as the levels increase. This means obstacles will not only appear more often, but closer together.

```
for (var i=0; i< Math.floor(300 + ( 125 * (lvl/2) )
); i++)
{
```

Next, we want to create a cave several hundred points long. The first level will start out with 366 points, and will only increase from there.

```
var newLeft = lastLeft + ( bias * (7+lvl) ) + ( (
rndWidth/2) - Math.floor( Math.random()*
(rndWidth+1) ) );
var newRight = lastRight + ( bias * (7+lvl) ) + ( (
rndWidth/2) - Math.floor( Math.random()*
(rndWidth+1) ) );
```

For each point, we determine the left and right side of the cave. We base this on the previous point, add in `bias` (increased with the level), and then add a random number within our allowed width, and we have a cavern wall that will vary by a random amount, but not by so much (at least in the first levels) that the level will be impossible.

```
if ( newLeft < 10 ) { newLeft = 10; bias = 1; }
if ( newLeft > (cWidth/1.5) ) { newLeft = cWidth/1.5;
bias = -1; }
if ( cWidth - newRight < newLeft + channelWidth )
{
    newRight = cWidth - ( newLeft + (channelWidth) );
}
```

Playing Around

```
if ( cWidth - newRight > newLeft + (channelWidth*1.5))
{
    newRight = newRight + (Math.random() * rndWidth);
}
if ( newRight < 10 ) { newRight = 10; }
if ( newRight > (cWidth-10)) { newRight = cWidth-10; }
```

Of course, without a few restrictions on the sides, it would be possible for the cave to wander off the screen, which does the player no good if they can't see it. So, we keep the cave on the screen. For the first two restrictions, we also affect the bias; this will tend to give the cave a zig-zag pattern overall.

```
points[0].push ( newLeft );
points[1].push ( newRight );

lastLeft = newLeft;
lastRight = newRight;
```

Finally, we add the points to the array and store them for future reference (the next iteration in the loop).

```
if ( (i % wallEvery) == 0 && ( i > 30 ) )
{
```

Next, we determine if it is time to put an obstacle in the way. First, we only check every so often (`wallEvery`), and we also restrict any obstacle from appearing within the first 30 points of the cave.

```
if (Math.random() >wallChance)
{
```

Next we decide if a wall will appear at this point; this makes obstacles pretty rare early on, but they add up in later levels.

```
var openingWidth = channelWidth/1.35;
var caveWidth = ((cWidth-newRight) - newLeft)
- openingWidth;
var wallOpening = Math.floor ( Math.random() *
caveWidth );
points[2].push ( newLeft + wallOpening );
points[3].push ( newLeft + wallOpening +
openingWidth );
```

For the obstacle, we create an opening that is a smaller opening than the width of the cave; this means the wall juts out from the cave by some degree. We then determine some random value within the range of the cave's opening and that's where the opening will go.

```
        }
        else
            // no wall
            points[2].push( -1 );
            points[3].push( -1 );
        }
    }
else
    // no wall
    points[2].push( -1 );
    points[3].push( -1 );
}
}
```

If there is no obstacle, we push `-1`; this way we can know if there's an obstacle (or not) at any given point.

And that's it! This will create a long, winding cave that's more treacherous the higher the level.

What did we do?

In this task, we generated the levels for our game, based on pseudo-random generation with some specific rules in place to create levels increasing in difficulty over the course of the game.

What else do I need to know?

This isn't the only way to generate a pseudo-random level, of course. There's all sorts of ways that are beyond the scope of this book, and there are things you can do to ensure that a level always stays winnable too. Level generation is a subject all on its own (with many, many smart people working in the field), so it won't take long to get your mind blown with some of the level-generation techniques out there. See http://en.wikipedia.org/wiki/Procedural_generation for examples of games that generate their levels procedurally as well as some links to articles describing various ways to generate levels procedurally. Keep in mind that this is highly specific to the kind of game you're developing.

Drawing to the canvas

Of course, it does no good to generate a level if we don't display it to the player. That's what we'll be doing in this task.

Getting on with it

First, we set the canvas up, also taking care to deal with retina screens in the process.

```
var c = $ge("gameView_canvas");
var ctx = c.getContext("2d");

c.setAttribute ("width", cWidth * window.devicePixelRatio);
c.setAttribute ("height",cHeight * window.devicePixelRatio);
c.setAttribute ("left", (screen.width/2) - (cWidth/2));
c.style.width = ""+cWidth+"px";
c.style.height = ""+cHeight+"px";
```

We store these as global variables, because it is important not to have to do DOM walking for every frame of content—that only slows us down—something that is a bad thing when we're trying to render a frame within 16 milliseconds (the maximum amount of time we can take if we're targeting 60 fps).

```
function doAnim(timestamp)
{
...
ctx.save();
ctx.scale (window.devicePixelRatio, window.devicePixelRatio);
ctx.fillStyle = "#802010";
ctx.strokeStyle = "#A04020";
ctx.clearRect ( 0, 0, cWidth, cHeight);
```

Next, we set up a few properties and then clear the canvas. It is critical to clear the canvas for every frame; otherwise you'll end up leaving ghosts behind of the previous frame.

Then we draw both sides of the cave using a loop:

```
for (var i=0; i<2; i++)
{
    var pts;
    var cLeft = -10;
    if (i==0) { pts = points[0]; }
    if (i==1) { pts = points[1]; cLeft = cWidth+10; }
```

Based on which part of the cave we're drawing, we assign either the 0 or 1 index of `points []` to another variable, `pts`. This lets us avoid having to double index the points array such as `points [i] [x]` and use `pts [x]` instead. We also define the left (or right) side of the wall that is off screen; this comes in handy for drawing the cave, since we need to fill the cave walls to make them solid. This means we are essentially drawing a big rectangle with one portion of it very rough; the rough side being the cave wall.

```
ctx.beginPath();
ctx.moveTo ( cLeft, -pieceWidth );
```

We begin the path and then move to the leftmost, topmost portion of the canvas; in fact, quite a way off of it. This ensures the player never sees the edge of the big rectangle we're drawing.

```
for (var j = Math.floor ( currentTop / pieceWidth )-1;
      j < Math.floor ( currentTop / pieceWidth ) + (
      (cHeight+(2*pieceWidth)) / pieceWidth );
      j++)
{
```

Next, we loop through each point in the array, but only over the ones necessary. If the player is halfway through the level, there's no point in drawing the previous points, nor is there any point in drawing parts of the cave that are beyond the screen's height.

```
var p = pts[j];
var y = (j * pieceWidth) - currentTop;
```

In order to keep the motion of the cave smooth, we multiply the current piece by the piece width and subtract our current position in the level. Technically, this would result in a jumpy view at the first and last drawn index, but we're drawing a couple of points beyond both sides, so any jumpiness is kept off screen.

```
if (i==1) { p = cWidth - p; }
ctx.lineTo ( p, y );
```

Next, we draw a line to the given point. If we're working on the right side, the point is to the left of the screen's right edge, which is the right side of the cave.

```
if ( points[2][j] > -1 )
{
  ctx.lineTo ( points[i+2][j], y );
  ctx.lineTo ( points[i+2][j], y+pieceWidth );
}
```

Playing Around

If we have an obstacle to display, we also draw a line to the point and then a vertical line one pieceWidth high. This will cause the obstacle to appear like a wall with an opening in it.

```
        }
        ctx.lineTo ( cLeft, ((cWidth+2)*pieceWidth) );
        ctx.closePath();
        ctx.fill();
        ctx.stroke();
    }
```

Finally, we draw a line off screen again, and then proceed to fill and stroke the path. The player will only see the rough edges of the cave wall.

```
ctx.strokeStyle = "#FFFFFF";
ctx.beginPath();
ctx.moveTo ( shipPositionX-10, shipPositionY-5 );
ctx.lineTo ( shipPositionX+10, shipPositionY-5 );
ctx.lineTo ( shipPositionX , shipPositionY+25 );
ctx.lineTo ( shipPositionX-10, shipPositionY-5 );
ctx.closePath();
ctx.stroke();
```

After we draw the cave walls, we need to draw the ship. In this case, we draw a simple triangle.

```
ctx.fillStyle = "#FFFF00";
ctx.font = "16px Helvetica";
ctx.fillText ( "Level: " + currentLevel, 10, 30 );
ctx.fillText ( "Distance: " + Math.floor((currentTop / (points[0].length*pieceWidth))*100) + "%", 10, 48 );
```

Generally most games need to display some text (such as a score), and so we do something similar too. We show the current level and the distance travelled through the level.

```
if (amTouching)
{
    ctx.fillStyle = "rgba(255,255,255,0.25)";
    ctx.beginPath();
    ctx.arc ( lastTouchX, 400, 50, 0, 2*Math.PI, false );
    ctx.closePath();
    ctx.fill();
}
ctx.restore();

...
}
```

Finally, if the user's control method is swiping, we display a translucent circle where the user is touching (assuming it is at the bottom of the screen) so that the user knows that the touch has been registered.

Keep in mind that when drawing a frame, we need to be quick, and thankfully the operations that we just saw depend on the hardware; we can achieve a frame within a few milliseconds. If we stay under 17 milliseconds, we can achieve nearly 60 fps, though on older hardware, this is more like 20-30 milliseconds. So we need to do something else, which we'll cover next, that is, drop frames so that the game doesn't feel too sluggish.

What did we do?

In this task, we drew the level on the canvas, displayed the ship, and put various text on the canvas as well. We also kept it quick, something we could only do by using the `canvas` tag.

What else do I need to know?

Of course, the preceding code only draws one frame; we need to call it multiple times in order to achieve fluid scrolling. There are two ways to do this: we can use `setTimeout` or use `requestAnimationFrame`. The latter is preferred, as it has better resolution than `setTimeout`, but it isn't yet supported in all mobile browsers. For our purposes, we use `setTimeout`. It calls in `doUpdate`, which we'll go over later, rather than `doAnim`.

Keeping up

Even if we can't hit 60 fps, we need to act like we are hitting 60 fps anyway. If we don't, any slowdowns will cause the game to feel as if it was in slow-mo. Instead, we need to *drop frames* and move things along as if we were getting 60 fps. The display will not be as smooth, but the gameplay won't get that slow-mo feeling until we drop down to a really low frame rate.

Getting on with it

If you were paying close attention to our code in the last task, you'll notice that we skipped a few lines. These were the lines crucial to keeping the game progressing as if it were hitting 60 fps:

```
var startTime;
function doAnim(timestamp)
{
    if (!timestamp) {
        timestamp = (new Date()).getTime();
    }
}
```

```
var diff = timestamp - startTime;  
...  
doUpdate ( 60/(1000/diff) );  
startTime = timestamp;  
}
```

All we're doing is measuring the time between frames and then applying some math to it. We then pass this in to `doUpdate`, which we'll discuss later. This number equates to the number of frames that should have passed in a given period of time. If we're at 60 fps, we'll always end up with this number being 1; but if we're at 30 fps, the number will be 2; and at 15 fps it will be 4, and so on. Since the number can be fractional, we can be very fine-grained when updating our game.

What did we do?

So, that was short, but it's not a terribly complicated concept. But it is very important; without it, if the device slowed down for any reason, the game would slow down too, making it feel as if it was in slow-motion. Instead, we'll drop the frames in order to keep the speed up. It won't be as smooth, but gameplay should always trump getting every frame in.

Performing updates

Of course, drawing the same frame over and over won't do any good. We need to update the game too. This includes responding to user input, checking for collisions, and moving us along the cave.

Getting on with it

We'll be doing all our updating in... guess what! `doUpdate`:

```
function doUpdate ( f )  
{
```

If you recall from the last task, `f` is the incoming frame multiplier. We want this to be 1, but it might be 2 if we're getting 30 fps, or 4 if we're getting 15. We'll use this at various points to multiply any updates so that everything moves as if we were getting 60 fps, even when we aren't.

```
var gameOver = false;  
var levelOver = false;
```

```
var pixels = ctx.getImageData(Math.floor(shipPositionX * window.devicePixelRatio),
    Math.floor(shipPositionY * window.devicePixelRatio),
    1,1).data;
```

There are the various methods of doing collision detection. We could determine it mathematically, but that gets more than a bit painful. Instead, we'll use the canvas' own data and do pixel-based collision detection instead.

The preceding line requests very little data from the canvas. In fact, we're only checking the center point of the ship for a collision. Part of this is so that we can be lenient on the user, part of it is laziness, but the other part is that pixel-based collision detection using the canvas is horribly slow. In fact, just checking for one pixel cuts our fps nearly in half.

```
if ( pixels[0] != 0 )
{
    $ge("gameView_nextButton").innerHTML =
        __T("START_OVER");
    showMessage (__T('CRASHED'));
    gameOver = true;
    currentLevel = 0;
}
```

If the pixel data in the data returned from the canvas is not zero (black), then we know we've impacted on something; what we've hit doesn't matter. We mark the game as over, and tell the user about it.

```
if (f > 0 && f != Infinity)
{
```

Sometimes `f` comes in as `0` or `Infinity`. This is most often at the start of a game, when we have to pass in a time difference, but there's not really been any. In this case, the division in `doAnim` will return `infinity`. We don't want to do anything in either case, so we make sure that we only operate if `f` is a reasonable value.

```
if (controlMethod == 0)
{
    if (buttonDown != 0)
    {
        if (Math.abs(shipAcceleration)<1)
            { shipAcceleration = buttonDown; }
            shipAcceleration = Math.min ( 10,
            shipAcceleration + ( buttonDown *
            deviceFactor ) );
            //shipAcceleration = buttonDown * 3;
    }
}
```

```
        else
    {
        shipAcceleration = shipAcceleration / 1.5;
        if (Math.abs(shipAcceleration)<0.25)
            { shipAcceleration = 0; }
        }
        shipPositionX += (shipAcceleration*f);
    }
```

buttonDown reflects the user's input if they are using touch controls. If they are sliding left, buttonDown will be negative. If they are going to the right, buttonDown will be positive. If they aren't doing anything, it will be zero. If they are using tilt controls, we'll calculate this value differently, but we'll show that in the next section.

If we're non-zero, we build up some acceleration, as if a thruster was on the ship. Since thrusters can't react instantly, the ship takes a little bit to react. This has the effect of making the game a little harder; one has to take into account the reaction time of the ship.

If we're zero, we reduce the acceleration, as if the ship was coasting to a stop. Once we reach a certain threshold, we stop the ship entirely, but until that point, there is some movement. Again, this adds some difficulty, as it must be considered when moving the ship.

One important variable in the preceding code snippet is deviceFactor. This is subjective; when the ship's movement felt right on an Android device, it felt too slow on an iOS device, and so this variable compensates a bit for that difference. Never be afraid to tweak the movement mechanics on different devices so that it feels the same, even if it isn't technically the same.

Note that we multiply the acceleration by f; this keeps the ship's movement working as if it were happening in a game with 60 fps.

```
var speed = ((4+currentLevel) * (f));
currentTop+= speed;
}
```

Next, we calculate the vertical distance through the cave, which is done by adding a number to currentTop. We adjust it slowly based on the current level as well, so higher levels will get faster and faster. Again, we multiply by f to keep things feeling smooth.

```
if ( Math.floor (currentTop/pieceWidth) >
points[0].length )
{
    $ge("gameView_nextButton").innerHTML = __T("CONTINUE");
    showMessage (__T('NEXT_LEVEL'));
    levelOver = true;
}
```

If the player has managed to navigate the entire level, we need to stop the game and tell the user that they made it. When they continue, they'll pick up at the next higher level.

```
if (!gameOver && !levelOver)
{
    timer = setTimeout ( doAnim,17);
}
}
```

Here's our `setTimeout` that keeps everything going. Note that we do this unless the game is over or the level is over. If one wanted to add a pause feature, one would also avoid setting a timer at this point.

The `17` here is intended to get us as close to 60 fps as possible. It doesn't work out to that in reality, as browsers don't have good resolution, so the next frame could arrive in 12 milliseconds or in 30. WebKit, thankfully, has something on the order of 4 milliseconds resolution, so it isn't likely to be far off of `17`, and so we can get up to 56 fps, assuming a modern device.

What did we do?

In this task, we handled the updating of the ship's position, the position in the cave, and whether or not we crashed, or completed the level.

What else do I need to know?

Yeah, pixel-based collision detection is a pretty lazy way out. In nearly every circumstance, math-based collision detection is the better (and faster) way to do it. There's lots of good stuff out there about how to do good collision detection, and it falls out of the scope of this project. You might start with http://en.wikipedia.org/wiki/Collision_detection for more information.

It wouldn't be quite so bad in our game if it wasn't for the fact that even requesting a single pixel from the canvas data drops our frame rate by nearly half. I suspect this has to do with having to transfer the data off of the GPU and back to the CPU for processing, but that's just a guess. Even so, it's a bit painful, and if I'd had more caffeine when writing the collision detection routines, I'd have gone the math route.

Handling touch-based input

Since we don't have a physical keyboard or joystick or D-pad, we have to emulate one on the screen. We could do this with two buttons on the screen for our game: one to go left, and one to go right. In fact, the game has it built in, just hidden. Another way is to allow for the differences in how a swipe might occur: a slow movement when our ship is not in danger, or a sudden movement when we need to avoid an obstacle in a hurry. Another method would be to simply link up the touch position on the screen to the ship; essentially our finger would have to trace the path through the cave. For smaller devices this might be fine, but for larger devices, it is better to go with some other method.

Getting on with it

In our game, our touch input can handle swipes (where the finger isn't always on the screen) to move the ship in short bursts, or it can handle long drags where the finger is always on the screen and slight movements generate movement.

To put it simply: if we move left, the ship should move left, and vice versa. However, if we need to get out of an obstacle's way in a hurry, we shouldn't have to move a long distance. We should be able to move a short distance in a quick burst, and so we also measure the distance between movements so we can tune the ship's movement to how fast our finger is moving. If the finger is moving slowly, the ship moves slowly. If it moves quickly, the ship moves really quickly.

None of this is hard; in fact, the hardest part isn't making it work, it's making it work well. It's hard to make a control method feel totally natural, and I won't claim to have mastered it here. It takes lots of testing to get a control mechanism just right.

Let's look at our code:

```
// check to see if our control method has changed
if (!localStorage.controlSetting)
{
    localStorage.controlSetting = "slide";
}
controlMethod = ( (localStorage.controlSetting) == "slide"
? 0 : 1 );
```

Before the game starts, we check the control method the user has selected – remember the options view. If the control method is `slide`, we'll attach events to an overlay DIV:

```
if (controlMethod == 0)
{
    $ge("gameView_overlay").addEventListener ( "touchstart",
    canvasTouchStart );
```

```
$ge("gameView_overlay").addEventListener ( "touchmove",
    canvasTouchMove  );
$ge("gameView_overlay").addEventListener ( "touchend",
    canvasTouchEnd  );
}
```

This overlay literally covers the entire canvas. You may wonder why we have to use an overlay—it turns out that the canvas itself isn't always so hot at handling touch events!

```
function canvasTouchStart (evt)
{
    lastTouchX = evt.touches[0].pageX;
    amTouching = true;
}
```

When a finger touches the screen, we record the initial touch, and tell the game that a finger is touching the screen. If you remember `doUpdate`, this last part tells the game to draw a translucent circle at the x position of the touch to give the user feedback.

```
function canvasTouchMove (evt)
{
    if (touchTimer>-1) { clearTimeout(touchTimer); touchTimer = -1; }
    var curTouchX = evt.touches[0].pageX;
    var deltaX = curTouchX-lastTouchX;
    if (Math.abs(deltaX)> 1)
    {
        buttonDown = ( (deltaX) / Math.abs(deltaX) ) / (
        8/Math.min(Math.abs(deltaX) ,8));
        lastTouchX = curTouchX;
    }
    else
    {
        buttonDown = 0;
    }
    // if player stays in same spot, clear the button...
    touchTimer = setTimeout ( function() { buttonDown = 0; }, 25 );
}
```

If a movement is received, we need to calculate the distance between the last x position and the new x position. We then give `buttonDown` a negative or positive value based on the direction of the movement. We also divide it if the movement was slow; if it was a fast movement (over five pixels), we'd have values of `-1` for left and `+1` for right, but a slow movement might return `-0.2` and `+0.2`.

If the finger hasn't moved by much (it needs to have moved by more than one pixel to register movement to the ship), then we indicate that `buttonDown` is 0, so that the ship will coast to a stop. We also set up a timer to fire in a few short ms to turn `buttonDown` to zero as well. This is because we won't receive a `touchMove` event if the finger stays absolutely still, so we need a way to catch this. If a movement is received before the timer expires, we cancel the timer so that the value never becomes zero as long as there is adequate movement.

```
function canvasTouchEnd (evt)
{
    buttonDown = 0;
    amTouching = false;
}
```

When the finger is lifted, we will instantly allow the ship to coast to a stop, and stop displaying the translucent circle.

This isn't the only way to do movement, and I urge you to experiment with different ways of processing touch-based input.

What did we do?

In this task, we dealt with touch-based input in order to allow the user to move our game's ship.

Handling the accelerometer

In order to respond to the tilt of the device, we need to use the device's accelerometer. These aren't the easiest things to deal with, and our implementation is a bit naïve. Unfortunately, it doesn't take long until you start getting into the math that's more than a bit complicated, and so lies outside the scope of this project.

Getting on with it

Accelerometer-based input is hard – really hard. So hard, in fact, that the game doesn't have a particularly good implementation of it. You are encouraged to experiment with a lot of devices and algorithms to come up with a good control scheme.

To turn on the accelerometer check, we first have to set up a watch for it:

```
tiltWatch = navigator.accelerometer.watchAcceleration (
    updateAccelerometer,
    accelerometerError,
    { frequency: 40 } );
```

This sets up a 40 milliseconds watch—not really quite as fast as I'd like, but workable. Every 40 milliseconds, the `updateAccelerometer` method will be called. If an error occurs, then `accelerometerError` will be called.

Watching the accelerometer takes some effort, so when done with it, it is always a good idea to clear it:

```
navigator.accelerometer.clearWatch (tiltWatch);
tiltWatch = -1;
```

So what do we get with the accelerometer? We get an object containing four values: a timestamp, an x value, a y value, and a z value. These values indicate the acceleration in a given direction. If the device is lying flat on a table, the x and y values will be zero, while the z value will be equal to the force of gravity. Generally we can assume that the x value corresponds to left/right tilt (assuming the device is upright), which is all we need for our game.

```
function updateAccelerometer ( a )
{
    if (amCalibrated)
    {
        var p = previousAccelerometer;
        var avgX = (p.x * 0.7) + (a.x * 0.3);
        previousAccelerometer = a;
        previousAccelerometer.x = avgX;
    }
}
```

When we receive an update, we apply a weighted average of the new input and the previous input, giving more weight to the previous input. It turns out that accelerometer-based input is really, really noisy, and so we only give the new input small importance. This has the side-effect, unfortunately, of making movement feel a little sluggish. The fractional values make a big difference in how well the ship responds, but at the trade-off of a ship that feels jittery. Feel free to experiment with these numbers, but they need to add up to one.

Note the `amCalibrated` variable; you could also compare the data to a calibrated value instead. The calibrated value is often obtained just before the level begins, when `amCalibrated` would be set to `false`. The next accelerometer update would calibrate the device:

```
else
{
    calibratedAccelerometer = a;
    previousAccelerometer = a;
    amCalibrated = true;
}
}
```

A better way to calibrate the accelerometer would be to take a series of inputs and average them out. For that matter, that is the same way to smooth out the accelerometer itself; but those algorithms are beyond the scope of this book.

Now, let's go back to `doUpdate` and look at the code specific for the tilt-based input:

```
if (f > 0 && f != Infinity)
{
    if (controlMethod == 0)
    {
        ...
    }
    else
    {
        // calculate the position based on the
        // accelerometer data
        if (amCalibrated)
        {
            shipPositionX = (window.innerWidth / 2) -
                (previousAccelerometer.x * 32);
            if ( shipPositionX < 0 )
            {
                shipPositionX = 0;
            }
            if ( shipPositionX > (window.innerWidth) )
            {
                shipPositionX = window.innerWidth;
            }
        }
    }
}
```

Here we make the assumption that with no tilt, the ship should be in the middle of the screen (which is `window.innerWidth / 2`). We then subtract the value of the `x` value from the last accelerometer sample and multiply it by 32. This number is really quite arbitrary, feel free to experiment here. Personally, 32 felt about right—not requiring the device to tilt all the way over, but also requiring sufficient tilt to make movement feel substantial.

The remainder of the lines ensure that the ship can't go off the edges of the screen.

What did we do?

In this task, we dealt with handling accelerometer-based input.

What else do I need to know?

Working with the accelerometer is no simple task. While our code is pretty simple, there are a lot of complicated algorithms out there to both reduce the noise but at the same time keep the movement from feeling sluggish—something I can't say we've really accomplished here.

Another alternative is to use the device's gyroscope. These values aren't based on acceleration, but on the position of the device itself, and though noisy, they aren't as noisy as the accelerometer values. As such, they can be easier to work with. The problem is that only iOS exposes these to a browser. On Android, one would need to write a plugin to work with this type of sensor. Furthermore, not every device has a gyroscope, so you would need to provide a fallback to the accelerometer just in case.

Game Over..... Wrapping it up

Whew! That was a lot of work, but the result is pretty fun. See how far you can get before it gets too hard. Let me tell you, it doesn't take me very long to crash and burn.

The final results are displayed in the following pages. I've not included Android screenshots since they appear virtually identical.

The starting screen will be as follows:



The options view will be as follows:



And an unfortunate crash is seen in the following screenshot:



Can you take the HEAT? The Hotshot Challenge

There are a lot of ways this game could be enhanced. Why don't you try a few?

- ▶ The game currently lacks a pause option; why don't you add one?
- ▶ Our game is naïve with regards to multitasking. Upon resumption, it will happily extrapolate where we should be in the cave after what might be a very long time. A better method would be to pause the game when it is in the background.
- ▶ We're using pixel-based collision detection. Why don't you try to use math-based detection instead?
- ▶ Try various control schemes until you find some you like.
- ▶ Use gyroscope values if available.
- ▶ Add powerups or other objects in the map that could affect the player for good or bad.
- ▶ Make static levels for the game that could be loaded on demand.
- ▶ Add logic to make sure any level is winnable.

Project 9

Blending In

CSS, HTML, and JavaScript can take us a long way to building an app that feels 99 percent native. That is, it almost feels native, it looks nearly native, and it acts mostly native. But if you take a closer look, you can see the small differences that tell any user with a lot of experience with their device that things really aren't native. To help overcome this, we can *blend in* to our environment using plugins that use real, true-blue native components.

What do we build?

We're going to revisit the *Socializer* app from *Project 2, Lets Get Social!* It's a good fit for blending in, and we can easily add our native components on to it without worrying too much about changing a lot of the app itself. There are some minor changes, of course, but in general, it works much the same, and even gains a little functionality in the process!

What does it do?

Although, we will add a couple of features to the app, the primary goal here is to use native components. For this, we'll be taking advantage of several great plugins in the PhoneGap Plugins repository (<http://github.com/phonegap/phonegap-plugins/tree/master/ios>). As the URL implies, we'll be doing this for iOS only. Unfortunately there aren't a lot of native plugins in this repository for Android. You can search around a bit and find some here and there (and some really great ones), but for now, we'll focus on iOS.

In fact, we'll be using quite a few native components: the navigation bar, the tab bar, the ActionSheet, the message box, the picker, and the e-mail composer. Oh, and let's not forget ChildBrowser either!

Why is it great?

This project will give us a much better understanding of interacting with multiple plugins, and we will also get much closer to a native look and feel.

How are we going to do it?

Here are the steps we'll be following:

- ▶ Installing the plugins
- ▶ Adding the navigation bar
- ▶ Adding the tab bar
- ▶ Adding the ActionSheet
- ▶ Adding the message box
- ▶ Adding the picker
- ▶ Adding the e-mail composer

What do I need to get started?

Make sure you've downloaded the code for this project so that you can follow along.

Installing the plugins

We've already had to deal with plugins a little bit, from ChildBrowser to even our own plugins. But this time, we'll be adding a lot of them. To install the plugins, go ahead and open up (or create) your Xcode project.

Getting on with it

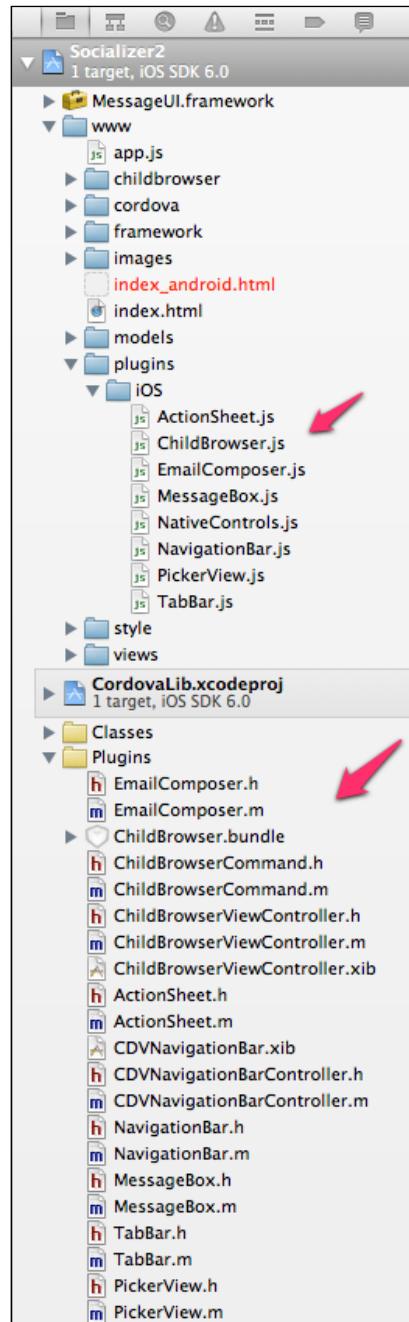
When you've downloaded the PhoneGap plugin repository, you should be able to extract it and navigate to the `ios` folder inside. Go ahead and install the ChildBrowser plugin, as described in *Project 2, Let's Get Social!* in the *Configuring the plugins* section. Once done, you need to go into each of the following directories and install each plugin:

- ▶ ActionSheet
 - Copy the `ActionSheet.h` and `ActionSheet.m` files into Xcode into the `Plugins` directory.
 - Copy the `ActionSheet.js` file to the `www/plugins/ios` directory using Finder.

- ▶ EmailComposer
 - Copy the `EmailComposer.h` and `EmailComposer.m` files into Xcode into the Plugins directory.
 - Copy the `EmailComposer.js` file to the `www/plugins/iOS` directory using Finder.
- ▶ MessageBox
 - Copy the `MessageBox.h` and `MessageBox.m` files into Xcode into the Plugins directory.
 - Copy the `MessageBox.js` file to the `www/plugins/iOS` directory using Finder.
- ▶ NavigationBar
 - Copy the `CDVNavigationBar.xib`, `CDVNavigationBarController.h`, `CDVNavigationBarController.m`, `NavigationBar.h`, and `NavigationBar.m` files into Xcode into the Plugins directory.
 - Copy the `NavigationBar.js` file to the `www/plugins/iOS` directory using Finder.
- ▶ PickerView
 - Copy the `PickerView.h` and `PickerView.m` files into Xcode into the Plugins directory.
 - Copy the `PickerView.js` file to the `www/plugins/iOS` directory using Finder.
- ▶ TabBar
 - Copy the `TabBar.h` and `TabBar.m` files into Xcode into the Plugins directory.
 - Copy the `TabBar.js` file to the `www/plugins/iOS` directory using Finder.

Blending In —

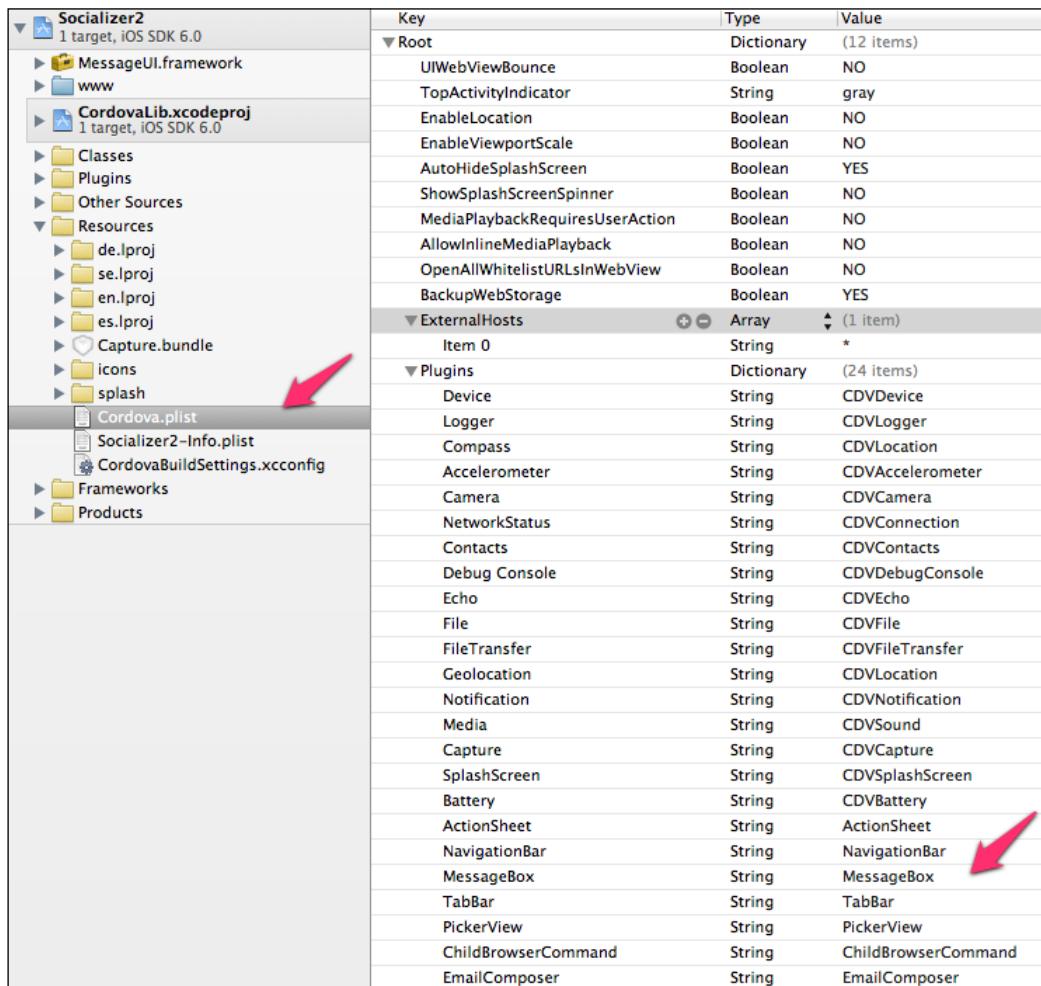
When done, you should have something that looks like this in Xcode:



Next, navigate to `Cordova.plist` in the `Resources` directory, and add the following key/value pairs to the `Plugins` section:

- ▶ `ActionSheet`, `String`, `ActionSheet`
- ▶ `NavigationBar`, `String`, `NavigationBar`
- ▶ `MessageBox`, `String`, `MessageBox`
- ▶ `TabBar`, `String`, `TabBar`
- ▶ `PickerView`, `String`, `PickerView`
- ▶ `EmailComposer`, `String`, `EmailComposer`

The result should look like the following screenshot:



	Key	Type	Value
↳ Root	Dictionary	(12 items)	
UIWebViewBounce	Boolean	NO	
TopActivityIndicator	String	gray	
EnableLocation	Boolean	NO	
EnableViewportScale	Boolean	NO	
AutoHideSplashScreen	Boolean	YES	
ShowSplashScreenSpinner	Boolean	NO	
MediaPlaybackRequiresUserAction	Boolean	NO	
AllowInlineMediaPlayback	Boolean	NO	
OpenAllWhitelistURLsInWebView	Boolean	NO	
BackupWebStorage	Boolean	YES	
ExternalHosts	Array	(1 item)	
Item 0	String	*	
Plugins	Dictionary	(24 items)	
Device	String	CDVDevice	
Logger	String	CDVLogger	
Compass	String	CDVLocation	
Accelerometer	String	CDVAccelerometer	
Camera	String	CDVCamera	
NetworkStatus	String	CDVConnection	
Contacts	String	CDVContacts	
Debug Console	String	CDVDebugConsole	
Echo	String	CDVEcho	
File	String	CDVFile	
FileTransfer	String	CDVFileTransfer	
Geolocation	String	CDVLocation	
Notification	String	CDVNotification	
Media	String	CDVSound	
Capture	String	CDVCapture	
SplashScreen	String	CDVSplashScreen	
Battery	String	CDVBattery	
ActionSheet	String	ActionSheet	
NavigationBar	String	NavigationBar	
MessageBox	String	MessageBox	
TabBar	String	TabBar	
PickerView	String	PickerView	
ChildBrowserCommand	String	ChildBrowserCommand	
EmailComposer	String	EmailComposer	

Next, add the script tags to our index.html file:

```
<script type="application/javascript" charset="utf-8"
src="./plugins/iOS/ChildBrowser.js"></script>
<script type="application/javascript" charset="utf-8"
src="./plugins/iOS/NavigationBar.js"></script>
<script type="application/javascript" charset="utf-8"
src="./plugins/iOS/TabBar.js"></script>
<script type="application/javascript" charset="utf-8"
src="./plugins/iOS/MessageBox.js"></script>
<script type="application/javascript" charset="utf-8"
src="./plugins/iOS/ActionSheet.js"></script>
<script type="application/javascript" charset="utf-8"
src="./plugins/iOS/PickerView.js"></script>
<script type="application/javascript" charset="utf-8"
src="./plugins/iOS/EmailComposer.js"></script>
```

What did we do?

In this section, you added all the plugins our project will use to Xcode.

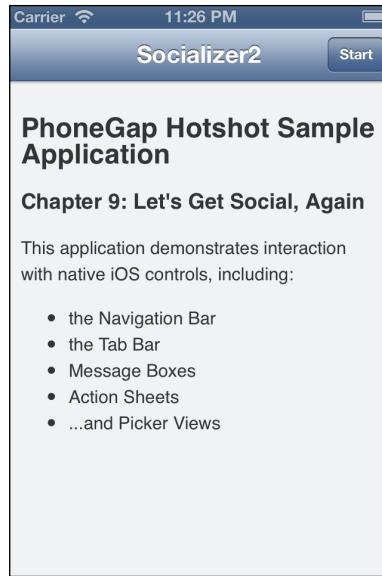
What else do I need to know?

There are a couple files that may give you problems when compiling—ActionSheet.h and PickerView.h. In particular, the error references an inability to find CDVPlugin.h. Just change the code at the top until it looks like the following:

```
/*#ifdef CORDOVA_FRAMEWORK
#import <CORDOVA/CDVPlugin.h>
/*#else
/*#import "CDVPlugin.h"
/*#endif
```

Adding the navigation bar

We're already pretty familiar with the concept of the navigation bar. We've been using one at the top of our HTML views for several projects now. This time, however, we're going to take that out and replace it with a native navigation bar. Here's how it will look:



Getting ready

This will require minor surgery on all three views—`startView.html`, `socialView.html`, and `tweetView.html` in the `www/views` directory. Go ahead and open those files so you can follow along. You might also want to open the versions from *Project 2, Let's Get Social!* as well, so you can see what has changed.

Getting on with it

First, our HTML views are going to change, since we need to remove our own navigation bar.

For `startView.html`:

```
<div class="viewBackground">
  <div class="content" style="padding:0; overflow: scroll; -webkit-overflow-scrolling: touch;" id="startView_scroller">
    <div class="content" id="startView_welcome">
      </div>
    </div>
  </div>
```

For `socialView.html`:

```
<div class="viewBackground">
  <div class="content" style="padding:0; overflow: scroll; -webkit-overflow-scrolling: touch;" id="socialView_scroller">
```

Blending In —

```
<div id="socialView_contentArea" style="padding: 0;  
height: auto; position: relative;">  
  </div>  
</div>  
</div>
```

For tweetView.html:

```
<div class="viewBackground">  
  <div class="content " style="padding:0; overflow:scroll; -  
  webkit-overflow-scrolling: touch;" id="tweetView_scroller">  
    <div id="tweetView_contentArea" style="padding: 0; height:  
    auto; position: relative;">  
      </div>  
    </div>  
  </div>
```

If you're comparing each view to that of *Project 2, Let's Get Social!* the navigation bar and tool bars have been removed in each view. We've also added native iOS scrolling to each view using `-webkit-overflow-scrolling: touch`.

By doing this, we've also removed several buttons: buttons that we'll need to replace in the navigation bar. But we will also need to remove the old references to them, typically in each view's `initializeView()` method:

For startView.html:

```
startView.initializeView = function() {  
  var theWelcomeContent = $geLocale("startView_welcome");  
  $ge("startView_welcome").innerHTML =  
    theWelcomeContent.innerHTML;  
}
```

For socialView.html:

```
socialView.initializeView = function() {  
  PKUTIL.include(["./models/twitterStreams.js",  
  "./models/twitterStream.js"], function() {  
    TWITTER.loadTwitterUsers(socialView.initializeToolbar);  
  });  
}
```

For tweetView.html:

(the method is removed)

You may be wondering, then, if we aren't going to initialize our buttons or the navigation bar in `initializeView()`, when and where are we going to do it?

First, the initialization part actually needs to happen in `app.js`. The navigation bar is going to persist across all of our views, and we can only have one of them. Therefore, we need to initialize it at the very start of our app. So in `app.js` in the `APP.init()` method (after `PKUI.CORE.initializeApp`), we add the following code snippet:

```
window.addEventListener("resize", function() {
    plugins.navigationBar.resize();
    , false);
    plugins.navigationBar.init();
    plugins.navigationBar.create();
    plugins.navigationBar.hideLeftButton();
    plugins.navigationBar.hideRightButton();
    plugins.navigationBar.setTitle(__T("APP_TITLE"));
    plugins.navigationBar.show();
```

This initializes and creates the navigation bar, hides its two buttons, and then sets its title. Finally, we display it on the screen.

Once we've done that, we can do some work in our views to tweak the navigation bar to our needs in each view.

In `startView.html`:

```
startView.viewDidAppear = function () {
    plugins.navigationBar.hideLeftButton();
    plugins.navigationBar.hideRightButton();
    plugins.navigationBar.setTitle(__T("APP_TITLE"));
    plugins.navigationBar.setupRightButton( __T("START"),
    null, startView.startApp);
    plugins.navigationBar.showRightButton();
}
```

First, we hide the buttons (just in case there are buttons visible we don't want seen), and then we set the title to the app's title—`Socializer2`. Then, we give the right button a title of `Start`, and link it to the `startApp()` method. At the end, we also show the right button.

```
startView.viewWillHide = function ()
{
    plugins.navigationBar.hideLeftButton();
    plugins.navigationBar.hideRightButton();
    plugins.navigationBar.setTitle("");
}
```

Blending In —————

When the view is about to hide, we essentially clear the entire navigation bar so that the buttons don't hang around for the next view.

In `socialView.html`:

```
socialView.viewDidAppear = function() {
    plugins.navigationBar.hideLeftButton();
    plugins.navigationBar.hideRightButton();
    plugins.navigationBar.setTitle(socialView.currentTitle);
    plugins.navigationBar.setupLeftButton( __T("BACK"), null,
    socialView.backButtonPressed);
    plugins.navigationBar.setupRightButton(__T("#"), null,
    socialView.changeReturnCount);
    plugins.navigationBar.showLeftButton();
    plugins.navigationBar.showRightButton();

    ...
}
```

In this case, we set the title of the view to the currently selected Twitter account, which by default will be the first one. We've added `currentTitle` to `socialView`, and are setting it in `loadStreamFor()` so that we can keep track of it.

We also add a Back button, on the left, and then we add a # button on the right. The Back button is just like every back button we've done in the past, so it will return us back to the starting view. The # button—this one's going to be interesting, but we'll save it for later.

```
socialView.viewWillHide = function() {
    plugins.navigationBar.hideLeftButton();
    plugins.navigationBar.hideRightButton();
    plugins.navigationBar.setTitle("");
    ...
}
```

Again, like good citizens, we clean up after ourselves!

In `tweetView.html`:

```
tweetView.viewDidAppear = function ()
{
    plugins.navigationBar.hideLeftButton();
    plugins.navigationBar.hideRightButton();
    plugins.navigationBar.setupLeftButton( __T("BACK"), null,
    tweetView.backButtonPressed);
    plugins.navigationBar.showLeftButton();
```

```
plugins.navigationBar.setupRightButton( __T("SHARE") , null ,  
tweetView.share ) ;  
plugins.navigationBar.showRightButton() ;  
}
```

Okay, this is a little different. Notice anything missing? That's right, we aren't setting the title. That's because we'll actually set it in `loadTweet()`:

```
{ ...  
    plugins.navigationBar.setTitle(theTweet.text) ;  
... }
```

Because we load the tweet immediately upon loading the tweet view, this has the effect of setting the navigation bar's title to the tweet's text.

But what if the tweet's text is too long? In fact, the chances of it being too long are really good. The native navigation bar will happily truncate it and append a "..." at the end, so we never have to worry if it will actually end up overflowing its bounds.

And of course, we'll clean up after ourselves, but there's no sense in printing the same code again.

What did we do?

In this task, we added the navigation bar and modified its title and interacted with the buttons on the navigation bar.

What else do I need to know?

The iOS-native navigation bar can do a lot of cool things, but the plugin doesn't expose all of those cool features (such as changing the tint color). So, for the time being, we're left with a rather plain-looking navigation bar—certainly not the color we used in *Project 2, Let's Get Social!* You can get around this by using some native Objective-C code, but the plugin itself doesn't offer us any other option but the black gloss navigation bar.

For a long time, iOS navigation bars were limited to two buttons—one on the left, and one on the right. And for the iPhone and iPod Touch, this is still a pretty good idea. For the iPad, one can add quite a few buttons to the bar without getting in the way of any text. Again, however, the plugin doesn't expose this functionality, so we can only have one button on the left and one button on the right.

Usually the button on the left is a **Back** button. This is typically given a left-pointing arrow, but the only way to do this is to create an image and pass it to the plugin. For our purposes, we decided to use a regular button without the left-pointing arrow. The plugin does have instructions on how to create the image and add it to the project if you want it, though.

Adding the tab bar

In the social view and tweet view we've had something similar to an iOS-native tab bar—something we called a toolbar. This is where the **Share** button in the tweet view lived. Unfortunately there's no native plugin for an actual toolbar, so we moved the **Share** button to the navigation bar.

In the social view, however, we're using the toolbar just like a real tab bar—namely a method to switch the contents of the view. We have five icons that represent Twitter accounts, and pressing any one of them will load recent posts from that stream. This works perfectly for a tab bar. Here's how it will look:



Getting ready

There's only one hitch. While we were able to display full-color, pretty avatars in our HTML tab bar, the iOS-native tab bar doesn't support that. In fact, it requires the images to be masks. iOS will then create the non-selected and selected images it uses on the tab bar from that mask. Essentially whatever is *white* (realistically, whatever has a value other than transparent) is painted on to the tab bar, and whatever is *transparent* isn't painted on to the tab bar.

This means we can't use the images we get from Twitter. We'd end up with five rectangle-shaped icons with no similarity to the original avatars. So we need to create our own versions in Photoshop or your preferred editor.

For tab bars, the best size to aim it is 30 x 30 for non-retina displays and 60 x 60 for retina displays. Then each image is saved as `tab#.png` and `tab#@2x.png`. The `@2x` version is for the retina displays. You can see each one in our code download in the `www/images` directory. All we did was to take the avatar from Twitter and color it white and then strip out all the background stuff and make it transparent.

Getting on with it

We'll be doing most of our work in our social view this time, since it's really the only view that did anything close to what a real tab bar does. Even so, we do need to add some code to `app.js` again (in the same spot as the previous task):

```
window.addEventListener("resize", function() {
    plugins.navigationBar.resize();
    plugins.tabBar.resize(); }
, false);

plugins.navigationBar.init();
plugins.tabBar.init();

plugins.navigationBar.create();
plugins.tabBar.create();

plugins.navigationBar.hideLeftButton();
plugins.navigationBar.hideRightButton();

plugins.navigationBar.setTitle(__T("APP_TITLE"));
plugins.navigationBar.show();
```

Notice the two commands to initialize and create the tab bar. This has to be done here as the tab bar must always follow the initialization and creation of the navigation bar; otherwise the size of the web view (sandwiched in between) will be incorrectly set.

Now, let's go to our social view:

```
socialView.initializeToolbar = function() {
var users = TWITTER.users;

if (users.error) {
    console.log(streams.error);
```

```
        alert("Rate limited. Please try again later.");
        return;
    }

    for (var i = 0; i < users.length; i++) {
        plugins.tabBar.createItem("tab" + i,
            users[i].getScreenName(),
            "/www/images/tab"+i+".png",
            {
                onSelect: function(tabName) {
                    var i = tabName.substr(3,1);
                    socialView.loadStreamFor('@' +
                        users[i].getScreenName());
                }
            });
    }
}
```

First, we create a tab bar item for each Twitter account. We give it the name of `tab#`—so `tab0`, `tab1`, and so on. We give the Twitter account name as the text of the tab bar, and then we use the images we created earlier as the icons for each tab bar instead of using the avatars Twitter gives us. Notice that we don't ever specify `@2x`; iOS just knows to use it when on a retina display. (Magic! It's also worth noting that the same thing happens when writing native code; rarely does one have to worry about appending `@2x` programmatically.)

We also add an `onSelect` handler to each tab item. We'll take the last character of the name of the tab item, which will be a number from 0 to 4, and then load the stream for that index. This means tapping on the first tab bar (named `tab0`) will load the stream for the first Twitter account.

```
...
socialView.viewDidAppear = function() {
    plugins.navigationBar.hideLeftButton();
    plugins.navigationBar.hideRightButton();
    plugins.navigationBar.setTitle(socialView.currentTitle);
    plugins.navigationBar.setupLeftButton(__T("BACK"), null,
        socialView.backButtonPressed);
    plugins.navigationBar.setupRightButton(__T("#"), null,
        socialView.changeReturnCount);
    plugins.navigationBar.showLeftButton();
    plugins.navigationBar.showRightButton();

    plugins.tabBar.show();
```

```
plugins.tabBar.showItems  
  ("tab0", "tab1", "tab2", "tab3", "tab4") ;  
  ...  
}
```

Now we've added the code to show the tab bar. We've also added the code to show each tab bar item. Since we know we'll always have five items, we just hardcode these values for now, but it would be equally possible to create lots of tab bar items and only show a few at a time.

```
socialView.viewWillHide = function() {  
  plugins.navigationBar.hideLeftButton();  
  plugins.navigationBar.hideRightButton();  
  plugins.navigationBar.setTitle("");  
  plugins.tabBar.hide();  
  ...  
}
```

Finally, we clean up after ourselves and hide the tab bar whenever the view hides. This keeps the tab bar from being visible on any other view.

What did we do?

In this task, we created the tab bar and then assigned tab bar items to it. We also assigned callback functions to each tab bar item.

What else do I need to know?

Apple is pretty strict when it comes to how a tab bar works. For one, on anything other than an iPad, it should never have more than five icons. (The reasons are pretty obvious: there's not a lot of space!)

So, what to do if you need more than five tabs? The accepted method would be to show four of those icons, add a *more* icon (consisting of three dots as the image), and then when the user taps that tab, show the remaining tabs in a table list. You can see this behavior in the *Music* app on the iPhone and iPod Touch, and if we were writing native code, we'd get this behavior nearly for free.

But we aren't writing native code, and so we have to do this manually. This means, if you want more than five tabs, you'll need to manually create the *more* tab and display the list of remaining tabs in a list on your own. One more catch: on an iPad, you should display all the tabs—displaying a *more* button is valid only for the iPhone and iPod Touch.

Finally, Apple highly suggests that one use no more than seven tabs on an iPad, but this is not as rigorously enforced.

Adding the ActionSheet

ActionSheets are great ways to present a few limited choices to the user, and so far we've been doing the equivalent by using pop-up message boxes with several buttons in them. For this project, we're going to display an ActionSheet when the user taps the **Share** button in the tweet view. Here's how it will look:



Getting on with it

In `TweetView.html`:

```
tweetView.share = function() {
    var actionSheet = window.plugins.actionSheet;
    actionSheet.create(__T('Share'), ['Twitter', 'Facebook',
    __T('Email'), __T('Cancel')], 
    function(buttonValue, buttonIndex) {
        if (buttonIndex==0)
        {
            PKUTIL.showURL("https://twitter.com/intent/tweet?text=" +
            encodeURIComponent(tweetView.theTweet.text) +
            "%20(via%20" + encodeURIComponent("@" +
            (tweetView.theTweet.from_user || 
            tweetView.theTweet.user.screen_name)) + ")");
        }
    }, {cancelButtonIndex: 3});
```

First off, we create the ActionSheet using `actionSheet.create()`. We give the sheet a title (`Share`), and then specify the buttons that can appear (`Twitter`, `Facebook`, `Email`, and `Cancel`). We then specify the handler for the ActionSheet, which will share the tweet to Twitter if the `Twitter` button is tapped. It won't do anything yet for the other buttons. Finally, we indicate that the `Cancel` button is the last button. This is so that iOS will know to color the `Cancel` button a different color to make it obvious that it is different. (Remember that indexes are zero-based.)

When our callback is invoked, we get two values: `buttonIndex` and `buttonValue`. While `buttonValue` could be useful, chances are good this could be anything, especially when we consider localization. Instead it's better to use `buttonIndex`. The first button will be `buttonIndex` zero, and so on.

What did we do?

In this task, we added an ActionSheet when the user taps on the `Share` button for a specific tweet.

What else do I need to know?

The ActionSheet is a pretty nifty thing, and it works really well on an iPhone or iPod Touch. On the iPad, it will appear in the center of the screen as a popover, not the best user interface, but something we can deal with. There are native methods you can use to position it correctly and add an arrow, but Apple seems to take apps using centered ActionSheets too, so we're not worrying about that now.

Something else that we need to know is that it's a good idea to keep the number of items to a small number, or the list will start to get really long. While the ActionSheet is supposed to condense these into a scrollable list view when this happens, there are bugs as to when it happens. This means that you can get a really long list that has some buttons cut off or missing entirely. (This is usually most visible on the iPad.)

Long story-short, keep the number of items to a reasonable value. Five or less is probably a good idea on a small device, and ten or less on an iPad.

Adding the message box

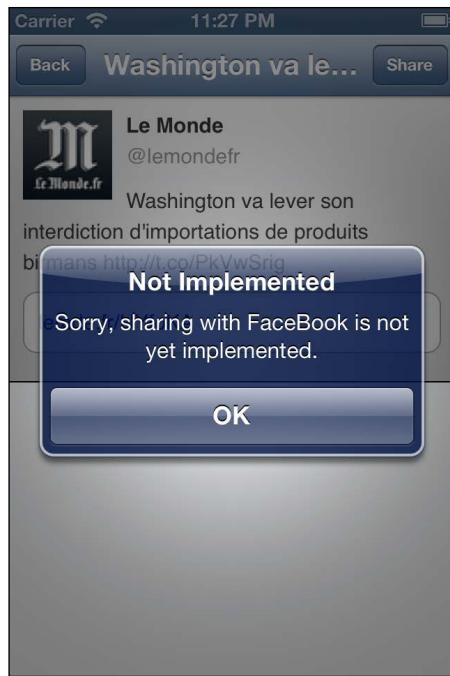
So far we've done pretty good with our own message box. It's not quite like an iOS-native message box, but it's pretty close. In this case, though, we want to go all the way.

Getting on with it

Working with the Message Box plugin is really easy. Let's go back to the `share()` method in our Tweet View:

```
tweetView.share = function() {
    var actionSheet = window.plugins.actionSheet;
    actionSheet.create('Share', ['Twitter', 'Facebook', 'Email',
    'Cancel'],
    function(buttonValue, buttonIndex) {
        if (buttonIndex==0)
        {
            PKUTIL.showURL("https://twitter.com/intent/tweet?text="
            + encodeURIComponent(tweetView.theTweet.text) +
            "%20(via%20" + encodeURIComponent("@" +
            (tweetView.theTweet.from_user ||
            tweetView.theTweet.user.screen_name)) + ")");
        }
        if (buttonIndex==1)
        {
            var messageBox = window.plugins.messageBox;
            messageBox.alert('Not Implemented', 'Sorry, sharing with
            FaceBook is not yet implemented.',
            function(button) {
                console.log('button ' + button + ' pressed');
            });
        }
    }, {cancelButtonIndex: 3});
```

I've highlighted the preceding code that displays a native message box. In this case, we use it for the second button on our ActionSheet: FaceBook. Since we've not implemented it yet, we display a nice notice to the user. Here's what it looks like:



Just like the ActionSheet and tab bar, we can respond (if we like) to any pressed buttons. In this case, we just log it to the console, but there are confirmation message boxes that one could use to make different things happen depending on what button was pressed.

What did we do?

We displayed a native message box and handled the callback when a button is pressed.

What else do I need to know?

Never, ever do *this in a real application*. I don't mean never use a message box; no, I mean never tell a user a feature isn't implemented in the first place. In fact, never show them that the feature was intended to be implemented but wasn't. Users don't appreciate it very much, and Apple will certainly reject the app if you leave it in.

The message box plugin describes other uses, including confirmation boxes, input boxes, and password input boxes. Read the plugin's readme for more information about how to use these additional features.

Adding the picker

Pickers are all over the place in iOS, and we really don't have a good analogue for them in our own framework yet. These things look like the following screenshot:



They're great at showing several choices at once and letting the user select one. They are often used to pick calendar dates, times, or even just a specific number out of a large range. They are good for this because they allow the user to scroll over a large range quickly.

In our example, we're just going to give a few options, but we could have had two hundred items in here with no real loss of functionality (but really, the user wouldn't need all of those in our case).

Getting on with it

We're going to go back to the social view for a moment and revisit something we said we'd come back to. Remember that # button? Yeah, now's the time we handle it.

Essentially, we're going to give the user the option to pick how many tweets they want to be loaded at once. Cool, right? This could be done in an ActionSheet, but the number of items is more than would really fit well, and so we'll use a picker instead.

```
socialView.numberOfTweets = 100;  
...  
socialView.loadStreamFor = function(searchPhrase) {  
...  
    var aStream = new TWITTER.TwitterStream(searchPhrase,  
        function(theStream) {  
...  
    }, socialView.numberOfTweets);  
}
```

First off, we've added a new property to the view: the number of tweets to load; and we've defaulted this to 100. Then we modified `loadStreamFor()` to use this value instead of the hard coded value in *Project 2, Let's Get Social!*

```
socialView.changeReturnCount = function ()  
{  
    var pickerView = window.plugins.pickerView;  
    var slots = [  
        {name: 'count', value: socialView.numberOfTweets, data: [  
            {value: 25, text: __T('Twenty-Five')},  
            {value: 50, text: __T('Fifty')},  
            {value: 75, text: __T('Seventy-Five')},  
            {value:100, text: __T('One Hundred')},  
            {value:125, text: __T('One Hundred Twenty-Five')},  
            {value:150, text: __T('One Hundred Fifty')},  
            {value:175, text: __T('One Hundred Seventy-Five')},  
            {value:200, text: __T('Two Hundred')}  
        ]}  
    ];
```

Next, we create the values that we want to display in our picker. The `name` isn't displayed anywhere; it's just used when we get told what the user selected. The first `value` is the currently selected value. This means the picker will show the user which item is currently selected. The remaining items define the value and the display text for each item. So the user will see Twenty-Five, but we'll get back 25.

```
pickerView.create(__T('Number of Tweets'), slots,  
function(selectedValues, buttonIndex) {  
    if (buttonIndex == 1)  
    {  
        socialView.numberOfTweets = selectedValues["count"];  
        socialView.loadStreamFor(socialView.currentTitle);  
    }  
, {style: 'black-opaque', doneButtonLabel: __T('OK'),  
cancelButtonLabel: __T('Cancel')});  
}
```

Next, we actually create the picker. We give it the title of `Number of Tweets`, so the user knows what they are setting. We pass in the values to display (slots), and then pass in a callback handler. This handler will set the value to the selected value and reload the Twitter stream, but only if the user taps on **OK**. Finally, we set the style (black), and pass in the names of the two buttons.

Pickers aren't limited to single-value lists. You can get multiple columns going, which is useful when setting a date, for example. You could have a Year column, a Month column, and a Date column. See the Plugin's readme file for more information on how to do this.

What did we do?

We created a picker, filled it with data, and reacted to the user picking a specific result.

Adding the e-mail composer

Sharing via e-mail is built in to nearly every app, and yet it's difficult to actually accomplish without using a plugin. Since this app is intended to share things easily, let's add sharing via e-mail.

Getting on with it

Back in our tweet view, we have the following:

```
tweetView.share = function() {
    var actionSheet = window.plugins.actionSheet;
    actionSheet.create('Share', ['Twitter', 'Facebook', 'Email',
    'Cancel'],
    function(buttonValue, buttonIndex) {
        if (buttonIndex==0)
        {
            PKUTIL.showURL("https://twitter.com/intent/tweet?text=" +
            encodeURIComponent(tweetView.theTweet.text) +
            "%20(via%20" + encodeURIComponent("@" +
            (tweetView.theTweet.from_user ||
            tweetView.theTweet.user.screen_name)) + ")");
        }
        if (buttonIndex==1)
        {
            var messageBox = window.plugins.messageBox;
            messageBox.alert('Not Implemented', 'Sorry, sharing with
            FaceBook is not yet implemented.',
            function(button) {
                console.log('button ' + button + ' pressed');
            });
        }
    });
}
```

```

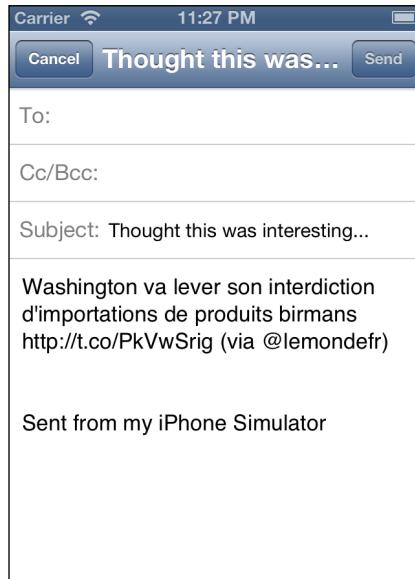
        }
        if (buttonIndex==2)
        {
            cordova.exec(null, null, "EmailComposer",
            "showEmailComposer",
            [{"body": tweetView.theTweet.text + " (via @" +
            (tweetView.theTweet.from_user || tweetView.theTweet.user.screen_name) + ")",
            "subject": "Thought this was interesting..."}]);
        },
        {cancelButtonIndex: 3});
    }
}

```

Again, we're in the `share()` method, and I've highlighted the code that's changed. Here we're responding to the **Email** button, which is index 2. We call the plugin using `cordova.exec()`, giving it the plugin name and the method we want to use (`showEmailComposer`). Then we pass the body and subject to the plugin. The body will be the tweet, plus who it was from, and the subject will be *"Thought this was interesting..."*. We could pass in more information, such as who the message should be sent to, but in our case, we don't know that information, so we don't send it on to the plugin.

Once done, we let go of control entirely, though it is possible to determine if the user actually did share via e-mail or not. In our case, we don't really care if they did, just that we offer the option to do so.

Here's what it looks like:



What did we do?

In this task, we created an e-mail composer with the subject and body set to data we specified.

Game Over..... Wrapping it up

We've accomplished quite a bit—we used seven plugins in total, and our app looks and feels pretty native now. It's not perfect; for example, navigation bars have a cool animation on native apps and in our app, it *blanks out* during view changes, but other than that, things look and feel pretty close to what a user would expect.

Can you take the HEAT? The Hotshot Challenge

Of course, there are always things that can be added and changed. Why don't you try a few of these challenges:

- ▶ Add Facebook sharing to the app.
- ▶ Store the number of tweets to load so that it is a persistent setting.
- ▶ Change up the way the navigation bar is cleared at the end of each view; try to achieve a more *native* feel. To truly achieve a native feel, you may need to delve into native code.
- ▶ Want a really complicated challenge? Download the avatars from Twitter and then figure out how to mask them programmatically. Then save them to the user's temporary storage and use them as the icons on the tab bar.

Project 10

Scaling Up

So far we've really only covered how to create apps for smaller devices such as phones. But there are a lot of other mobile devices that aren't phone-shaped, namely tablets (and the so called phablets, which are often 7 inches instead of 10). Though there aren't as many users who have tablets as phones, it is still an incredibly important market.

Sometimes it is possible to simply display the same user interface on a larger device. This is often seen in games, where the graphics and control areas are often just scaled to the device's screen size. Other times it is possible to largely use the same user interface, but small tweaks are required in order to make it function well on a larger screen. And there are other times when there is simply no choice; the user interface must be re-thought entirely for the larger screen.

What do we build?

In this project, we'll revisit an app that we created in *Project 3, Being Productive*. The app isn't terribly complex, but it is flexible enough to support various ways of scaling up, which is what we'll do. We're going to create several different versions of Filer, each with different concepts of scaling to a tablet-sized screen.

What does it do?

One of the main problems with developing for a larger screen when one has been solely developing for a small screen is "What to do with all that space?" When one is constrained to developing for a 320 x 80 or 600 x 800 dimension, it can be a sudden shock to realize that one has a lot more pixels to fill. Often, these larger displays are sized at 1024 x 768, 1280 x 768, 1280 x 800, or higher. In fact, the iPad 3 has a display that's technically 2048 x 1536, which when you think about it, is pretty astounding. Thankfully, the iPad 3 scales that back to 1024 x 768 for us.

What we'll do in this project is rethink the user interface for Filer to account for the larger real estate. We won't focus so much on the actual functionality—we've done that work in *Project 3, Being Productive*, but we will deal with how to handle larger screens.

We'll focus on two typical scenarios: **scale-it-up**, where we simply scale the interface to fit the new screen size, and **split view** (otherwise referred to as **master-detail**), where we will add a sidebar to the interface (something you see quite often on the iPad, for example, the **Settings** app).

Why is it great?

Sometimes an app just screams for more space—and note-taking apps aren't any exception. A bigger screen means that there is a larger on-screen keyboard, and a bigger screen means that there is more space for important content—such as text. At other times, we can transition an app to the larger screen by using split-view layouts that allow us to efficiently flatten the app's hierarchy. We'll explore all these options with our three versions of Filer.

How are we going to do it?

We'll approach the three designs as follows:

- ▶ Designing the scaled-up UI
- ▶ Implementing the scaled-up UI
- ▶ Designing the split-view UI
- ▶ Implementing the split-view UI

What do I need to get started?

For this particular task, we'll be working with the files for this project, so if you want to follow along, go ahead and download them. There are two directories named `1` and `2`, which are versions of the app in this project. The first is what we'll focus on next, while the second is what we'll work on later.

Designing the scaled-up UI

A lot of apps can simply "scale up" to fit the larger screen, and our framework, thankfully, does a lot of the "scaling" part for us. While this works well for games, we do need to do a bit more work to make Filer fit the big screen well.

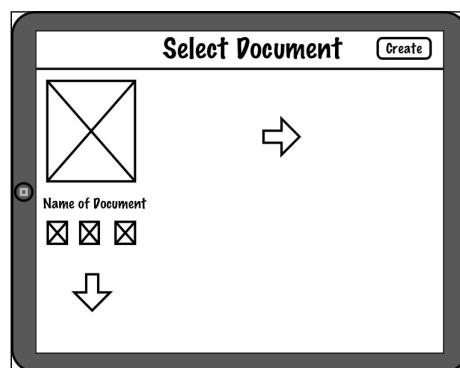
Getting on with it

If you remember the Filer app from *Project 3, Being Productive*, there were three views: a **start view**, a **documents view**, and the **document view**. We'll be scrapping the first view—there'd be nothing to do with it to make it work on a larger screen anyway. Instead, we'll focus on the last two views—and, in all honesty, for this task, we're only really going to make a lot of the changes to the first of them.

Let's take a look at the screenshot from the documents view for the Filer app from *Project 3, Being Productive*:



For our tablet-sized app, we'll display this list of documents horizontally and vertically, rather than just horizontally. On an iPad, this will show about three icons across when in portrait orientation and four icons across when in landscape. This means our mockup looks like this:



For the purposes of this project, we'll keep the **Create** button on the navigation bar, but there is a challenge at the end of the project to turn this into a larger feature within the document list. For example, some apps might have a blank document image with a "plus" icon in it to symbolize creating a new document. Others might use a dashed rectangle to indicate the same thing. For the larger display, something like this is definitely appropriate; while on a small screen, it'd be seen as a waste of space.

Beyond making this change to the document list, that's really all we're going to be doing to the app. The rest of it will work as is on the larger screen, thanks to the fact that our framework is designed to fill the screen.

What did we do?

In this task we went over the app from *Project 3, Being Productive*, and created a new mockup for the user interface for the larger screen.

What else do I need to know?

Scaling up an iPhone app is pretty easy if you've already planned for the future of scaling it up. That is, if you've planned everything down to the pixel and built for a 320 x 480 screen, you're going to have to change all those pixels around on a larger screen. When dealing with simple productivity apps such as Filer, building a layout that can scale to a larger screen isn't terribly difficult, but get into more complex layouts and graphics, and it starts to become a challenge.

In some ways, highly graphical games have it both the hardest and the easiest. A game is probably going to keep the same user interface when scaling to a new screen—with perhaps a few minor tweaks to button placement or size. The graphics, however, are going to be the same, visually. Underneath the hood, though, those graphics may be rendered at vastly different resolutions. A certain graphic might work fine on a small screen, but get that up to a larger screen, and it will either seem too large or too small. To avoid forcing the browser to scale *everything* (which always slows things down and results in some blurriness), it is better to re-render the graphics for the target screen, because you never know what kind of screens will be out in the future. It is, for this reason, always better to create your graphics in a vector format—this way you can always create a new rendition when a new size is needed.

One of the hardest things to deal with properly are full-screen images. These might be in-game backgrounds, menu backgrounds, splashes, and so on, and you want them to look as nice as possible. In our sample game (*Project 8, Playing Around*), we didn't focus on this a great deal, but if you had a device that had a substantially different aspect ratio than I had, you probably noticed some letter-boxing when the full-screen assets were shown. This is one way to approach it without having to do a lot of work—the other would be to scale and crop the image, potentially blurring it a bit, and losing portions of the image. The only other realistic option is to create an image specifically for each supported resolution.

For the best visual appearance, you should always render your images at the device's native resolution. For a Retina iPad, this would be 2048 x 1536 for a full-screen image. This, of course, is different for just about every Android device, and there's no terribly easy way to deal with it. You can replace the graphics via JavaScript based on the size of the screen, or you can use media queries to target specific graphic elements. You should note that though the framework we use does make a distinction between phone-sized devices and tablet-sized devices as well as non-retina and retina displays, it does nothing about all the different resolutions available on Android. Your best option would be to use CSS media queries (for more information, see https://developer.mozilla.org/en-US/docs/CSS/Media_queries).

In other ways, non-game apps can be terribly painful to scale up. You might be dealing with, for example, a lot of content that is formatted in a reasonably complex manner. It looks great given one screen size, but on another, things may break in odd places, especially when just scaling it. Sometimes the fact that we're working in HTML and CSS will save us—it's meant for dealing with complicated layout, but just as many times as not, it'll cause the look and feel to go awry in a way that you hadn't envisioned.

This is when creating code and layout *specifically* for the tablet-size screen may be necessary. You can do things in your JavaScript, HTML, and CSS code to handle these sizes—you could put a DIV tag classed with `tablet` in your HTML, and have a CSS rule that hides it on anything but a tablet. Likewise, you could hide the phone UI elements if you're trying to create a universal app that can run on both phone-sized screens and tablet-sized screens. Or, if you're positioning certain things with JavaScript, you can always look at the type of device you're running on to get a good idea about what to do—and worst case, look at the width or height of the screen. Again, using media queries can often help when dealing with multiple resolutions.

Implementing the scaled-up UI

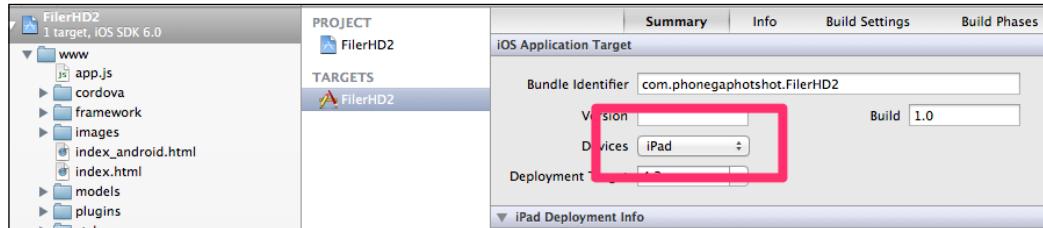
Now that we've designed the UI for Filer HD mark I, it's time to implement it. The number of changes that we've made to our code are astoundingly small, so get ready to keep your eyes open—blink, and you might miss it!

Getting ready

Although we're focusing primarily on the iOS platform for this app, the concepts apply equally to any platform-based tablet. With that said, to render an iOS app specific to the iPad, there are a couple of settings that need to be set in the project itself, outside of code:

In the project's settings in Xcode, change the **Devices** setting to **iPad** from **Universal**.

You should see the project setting like so:



Aside from this change (and setting our typical settings for the project in `Cordova.plist` for iOS), we'll copy the files from *Project 3, Being Productive*. If you want to follow along, navigate to `/1/www` in the code files of this project.

Getting on with it

There's really surprisingly little that we have to do to make the app fit to the larger screen. As we've said before, our framework does do a good portion of the work—it always attempts to ensure that the content fills the screen, and our HTML in our views helps too—where possible, we want to use percentages not pixels. (This is not to say that using pixels is not good; for example, we use them liberally when dealing with button placement on the navigation bar.)

Just as a reminder, here's the documents view's HTML:

```
<div class="viewBackground">
  <div class="navigationBar">
    <div id="documentsView_title"></div>
    <button class="barButton" id="documentsView_createButton"
           style="right:10px" ></button>
  </div>
  <div class="content avoidNavigationBar" style="padding:0; overflow:
  scroll;" id="documentsView_scroller">
    <div id="documentsView_contentArea" style="padding: 0; height:
    auto; position: relative;">
      </div>
    </div>
  </div>
</div>

<div id="documentsView_documentTemplate" class="hidden">
  <div class="documentContainer">
    <div class="documentImage">
      
    </div>
```

```
<div class="documentTitle" onclick="documentsView.  
renameDocument(%INDEX%)">  
    <span>%TITLE%</span>  
</div>  
    <div class="documentActions">  
          
          
          
    </div>  
</div>  
</div>
```

Nothing here has to change, really. Next up, here's some of our styling:

```
.documentContainer  
{  
    width: 240px;  
    height: auto;  
    padding: 40px;  
    padding-left: 20px;  
    padding-right: 0px;  
    display: inline-block;  
    text-align: center;  
}  
.documentContainer .documentImage img  
{  
    width: 190px;  
    height: 242px;  
}  
  
.landscape .documentContainer  
{  
    padding: 10px;  
}  
  
.landscape .documentContainer .documentImage img  
{  
    width: 125px;  
    height: 160px;  
}
```

So far, no change for the iPad devices. We did strip out the code that turned this into a list for Android devices, and so all tablet devices will get nice, big document icons.

The one place where we made some change was in the actual code, so let's take a look there:

```
documentsView.documentIterator = function ( o )
{
    var theHTML = "";
    var theNumberOfDocuments = 0;
    for (var i=0; i<o.getDocumentCount(); i++)
    {
        var theDocumentEntry = o.getDocumentAtIndex ( i );

        theHTML += PKUTIL.instanceOfTemplate ( $ge("documentsView_
documentTemplate"),
            { "title": theDocumentEntry.name.substr(0, theDocumentEntry.name.length-4),
              "index": i
            }
        );
        theNumberOfDocuments++;
    }
// code deleted
$ge("documentsView_contentArea").innerHTML = theHTML;
}
```

See the comment about deleted code?

That's right—we *deleted* something! If you remember, on the iPhone we wanted the document listing to scroll *horizontally*, so there was some code that set the width of the document list's container to *the number of items multiplied by the width*. This would make DIV larger, and it therefore allowed the content to scroll horizontally. For this app, we want the width to be of one-screen width, and then we want the browser to *wrap* our documents just like words on a page. So the next document to the right of the screen actually goes below the left-most document and starts a new row.

Guess what—that's *it*. Our buttons didn't need to change position, and the text editor in our document view is already coded to fill the entire screen, so we didn't need to do anything there. The only thing we *had* to take out was code that was written specifically for a phone-sized device.



So what does it look like now?

A screenshot of a tablet application displaying a full-screen text document. The top status bar shows 'Carrier' with signal strength, '12:34 PM', '100%', and a battery icon. The title 'Lorem Ipsum' is centered above the text. The text is a standard placeholder text, known as 'Lorem Ipsum', used in the publishing industry. It consists of multiple paragraphs of Latin text, which is mostly gibberish but follows a structured layout.

What did we do?

In this task we converted what was a phone app into a tablet app.

What else do I need to know?

We accomplished this by actually removing code that was constraining the app to that of a phone, but not all apps will be so simple to scale. Sometimes you may need to reposition buttons, content, and various elements in order to fit better on the larger screen. If you want the app to remain universal (that is, it will work on both a phone-sized device and a tablet-sized device), your work becomes harder, because you have to keep both layouts around. Thankfully, CSS, HTML, and JavaScript come to the rescue and help us out by letting us target certain classes and IDs or media queries with CSS and writing JavaScript code specific to a particular layout. In our case, we could have kept the phone-sized specific code by checking for the size of the device—the framework happily takes care of the rest of the sizing itself to the different sizes.

Sometimes you can get by with the way your framework handles the different viewport sizes, but there are often times when this falls flat—the UI is either too sparse on a large screen or too crammed on a small screen. If this occurs, it would be better to build your UI specifically for each device size rather than relying on the framework to get things right—because sometimes it doesn't.

Designing the split-view UI

The split-view layout is by far one of the most popular methods for scaling your app to the tablet. It also has the side benefit of flattening the application's information hierarchy, which is just the technical way of saying that it takes less "taps" to get somewhere in the app.

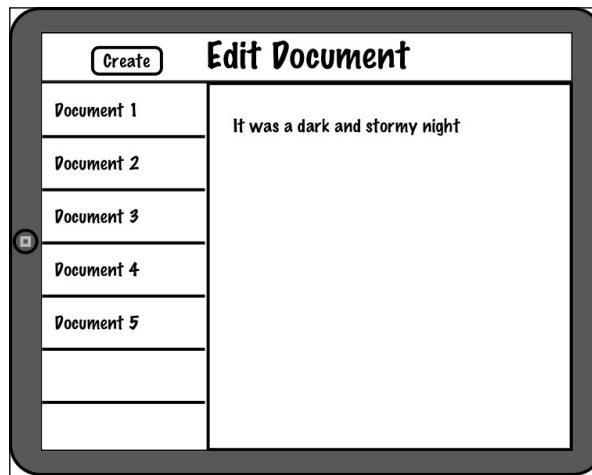
Most tablet platforms implement this view in similar ways—in landscape mode it's always there on the left (but sometimes on the right), and in portrait mode it's usually hidden offscreen, ready and waiting for when the user taps a button to call it out. Other times the view is always visible in portrait mode, but this depends on the type of app and whether or not the loss of screen space is worth having the split view always visible.

Getting on with it

The split view is really just two views put together. That's the easy way to think about it—one view is on the left in a smaller sidebar while the second view is on the right. The left view is technically called the **master view**, while the right view is called the **detail view**. Technically, this pattern is the **master-detail** pattern, and it is most obvious when working with data records where the record selection occurs in the master view (on the left), and the detail of the record shows in the detail view (on the right).

In our app, we're going to make the documents list the master view so that the document itself can be the focus of the user. This means that a specific document will become the detail. In this arrangement, however, we need to switch from the pretty grid listing of documents in the previous version of the app and go back to a simple list.

Here's what our design looks like now:



The only thing not quite "true" in this sketch is the **Create** button on the top. In reality there's going to be a title next to it as well, just that there wasn't enough space in the mockup to put it there. So it won't look as out-of-place as it does in the previous screenshot.

What isn't obvious is what happens when the device rotates to portrait orientation. The sidebar will actually disappear—leaving only the text document visible. We'll have a button on the left titled **Documents** that will bring the sidebar back, and once the sidebar is displayed, a **Close** button that will dismiss it. In a challenge at the end of this project, you'll be asked to implement gestures to open and close the sidebar.

What did we do?

In this task, we designed the user interface for a split-view layout.

What else do I need to know?

It really depends on your app if the master view needs to go away when oriented in portrait. Some apps can spare the loss of screen-width, other apps can't. When dealing with content (such as graphics, text, and so on.), it's probably a good idea to get rid of it when in portrait mode. If you're dealing with settings, properties, or the like, then you can keep it there with little impact on usability.

Implementing the split-view UI

Now it's time to implement the changes necessary to make our app a split-view app. From this point forward, we'll be working in `/2/www` if you want to follow along. We started from the code in the last task and then modified it to suit the new user interface.

Getting on with it

Unlike the last time, we're going to be making several modifications. Not a *lot* of code, mind you, but we'll be touching several different files and making tweaks to support the change. The documents view gets only a few minor modifications, though we will change the display to a list instead of a grid, and the file view gets several modifications. For one, it has to handle what to do when there is no document loaded (which will happen at the beginning of the app). Second, it has to change the way it handles autosaving content (since there is no longer any dismissal of the view).

But first, we need to get the layout set up to show two views side-by-side. We'll start in `index.html`:

```
<body>
  <div class="container" id="rootContainer">
    <div class="container leftSplit" id="leftSplitContainer">
      </div>
    <div class="container rightSplit" id="rightSplitContainer">
      </div>
    </div>
  </body>
```

Notice that we've added two elements to the `rootContainer` element. The first is `leftSplitContainer` and the second is `rightSplitContainer`. The position of these elements should be apparent from the ID values.

This alone isn't going to get us there, though. We need to style these appropriately—and for this, we've made a change in the framework's base CSS. Look at `2/www/framework/base.css`:

```
.landscape .container.leftSplit {
  width:319px;
}

.landscape .container.rightSplit {
  left:320px;
}
```

```
.portrait .container.leftSplit
{
    display: none;
    width: 319px;
    z-index:2;
    box-shadow: rgb(0, 0, 0) 0px 0px 8px;
}

.portrait .container.rightSplit
{
    left:0;
}
```

What we've done is indicated that while the device is in landscape orientation, the two splits should be side-by-side. The left-hand will be 319 pixels wide, and the right-hand view will start at pixel 320. Whatever's left on the screen will determine the width of the right-hand view. *The size of this sidebar is not set in stone—if your app needs a smaller sidebar, go ahead and aim lower—likewise, if it needs a larger one, set it larger. It is best, however, to not exceed half the width of the screen. If you feel the need, it is time to decide if you've got your views on the correct side.*

In portrait mode, we *hide* the sidebar. That said, this sidebar can appear again when the user wants, so we also make sure that it is indexed above all of the other content. We also give it a shadow so that there is a visual distinction for the user between the sidebar and the content underneath.

Changing the styles alone, however, isn't enough. Let's take a look at what we've changed in `app.js`:

```
// load our document view
PKUTIL.loadHTML ( "./views/documentsView.html",
    { id : "documentsView",
        className: "container",
        attachTo: $ge("leftSplitContainer"),
        aSync: true
    },
    function (success)
    {
        if (success)
        {
            documentsView.initializeView();
            PKUI.CORE.showView ( documentsView );
        }
    });
});
```

```
// load our fileView
PKUTIL.loadHTML ( "./views/fileView.html",
    { id : "fileView",
        className: "container",
        attachTo: $ge("rightSplitContainer"),
        aSync: true
    },
    function (success)
    {
        if (success)
        {
            fileView.initializeView();
            PKUI.CORE.showView ( fileView );
        }
    });
}

window.addEventListener('orientationchange', APP.updateSidebar,
false);
}
```

First, our loading code has changed a little. Note that instead of attaching to `rootContainer`, we attach to `leftSplitContainer` and `rightSplitContainer`. This first step is critical to ensuring that each view's content ends up in the right place on the screen. Note also that we show each view. This is new too—previously we would only have shown one view, but since we are combining two views on the screen, we need them both to be visible.

There's one last new feature at the bottom of the previous code—a new event listener. We'll cover the following code, but essentially we ask the browser to notify us of any change in orientation. While the CSS and HTML go a long way to making sure our layout is correct when the orientation changes, they don't get us *all* the way, and so we need some code to figure out the rest.

```
APP.toggleSidebar = function ()
{
    $ge("leftSplitContainer").style.display =
        ( ($ge("leftSplitContainer").style.display == "block") ? "none" :
    "block" );
}
```

This code is called by both the documents and file views, and its sole purpose is to toggle the appearance of the sidebar. If it is visible, this function will hide it, and vice versa.

```
APP.updateSidebar = function ()
{
    if (PKDEVICE.isPortrait())
```

```
{  
    $ge("leftSplitContainer").style.display = "none";  
}  
else  
{  
    $ge("leftSplitContainer").style.display = "block";  
}  
}
```

This code gets called at every change in orientation. If we change to portrait, we'll hide the sidebar, and if we change to landscape, we show it.

Next, we need to make some minor changes to the documents view:

```
<div class="viewBackground">  
    <div class="navigationBar">  
        <div id="documentsView_title"></div>  
        <button class="barButton" id="documentsView_closeButton"  
            style="left:10px" ></button>  
        <button class="barButton" id="documentsView_createButton"  
            style="right:10px" ></button>  
    </div>  
    <div class="content avoidNavigationBar" style="padding:0; overflow:  
        scroll;" id="documentsView_scroller">  
        <div id="documentsView_contentArea" style="padding: 0; height:  
            auto; position: relative;">  
            </div>  
        </div>  
    </div>
```

The main difference is that we've added a **Close** button to the view—this button will allow the user to dismiss the sidebar when they've previously elected to display it in portrait mode. We'll add some styles later on to prevent it from being visible while in landscape mode. We also changed the click handlers for certain areas, since this is now a list instead of a grid.

To support the new button, we have added some code to `documentsView.initializeView`:

```
documentsView.initializeView = function ()  
{  
    PKUTIL.include ( ['./models/FilerDocuments.js', './models/  
FilerDocument.js'], function ()  
    {  
        // display the list of available documents
```

```
        documentsView.displayAvailableDocuments() ;
    }
);

documentsView.viewTitle = $ge("documentsView_title");
documentsView.viewTitle.innerHTML = __T("APP_TITLE");

documentsView.closeButton = $ge("documentsView_closeButton");
documentsView.closeButton.innerHTML = __T("CLOSE");
PKUI.CORE.addTouchListener(documentsView.closeButton, "touchend",
function () { APP.toggleSidebar(); });

documentsView.createButton = $ge("documentsView_createButton");
documentsView.createButton.innerHTML = __T("CREATE");
PKUI.CORE.addTouchListener(documentsView.createButton, "touchend",
function () { documentsView.createNewDocument(); });
```

All this does is call `APP.toggleSidebar` whenever the **Close** button is tapped. Since the sidebar will be visible when we call this function, this means it will dismiss the sidebar by hiding it.

The only other change? We removed calls to `PKUI.CORE.pushView`. These would normally have pushed the file view on the stack, but since it's already visible, we don't need to push anything. So we just remove those lines.

We did make changes to the styling, however (in `style.css`):

```
.documentContainer
{
    padding: 10px;
    background-color: #FFFFFF;
    width: 100%;
    height: 90px;
    color: #000;
    text-align: left;
    border-bottom: 1px solid #C0C2C4;
}
.documentContainer .documentImage img
{
    width: 60px;
    height: 70px;
}
```

```
.documentContainer .documentImage
{
    width: 70px;
    height: 80px;
    float: left;
}

.documentContainer .documentTitle
{
    height: 2em;
}
```

The previous code will make each document item a nice little list item with the icon on the left, the title on the right, and the action icons below. You could take what you've learned from previous projects and add gesture support as well.

The file view itself gets quite a bit of change:

```
<div class="viewBackground">
<div class="navigationBar">
    <button class="barButton" id="fileView_documentsButton"
style="left:10px" ></button>
    <div id="fileView_title"></div>
</div>
<div class="content avoidNavigationBar" style="padding:0; "
id="fileView_scroller">
    <div id="fileView_contentArea">
        <textarea id="fileView_text"></textarea>
    </div>
</div>
</div>
```

The first thing different is that we've added a documents button on the navigation bar in place of the **Back** button. This button will show the sidebar when in portrait mode. When in landscape, we'll have a special style that makes this button go away.

Next, let's look at the code:

```
fileView.initializeView = function ()
{
    fileView.viewTitle = $ge("fileView_title");
    fileView.viewTitle.innerHTML = __T("Select or Create a Document");
    PKUI.CORE.addTouchListener(fileView.viewTitle, "touchend",
function () { fileView.entitleDocument(); } );
```

```
fileView.documentsButton = $ge("fileView_documentsButton");
fileView.documentsButton.innerHTML = __T("DOCUMENTS");
PKUI.CORE.addTouchListener(fileView.documentsButton, "touchend",
function () {APP.toggleSidebar();} )

$ge("fileView_text").style.display = "none";
}
```

This code is pretty similar—instead of adding a listener to a **Back** button, we add one to the **Documents** button. The only other thing we do differently is hide the TEXTAREA control—if no document is loaded, there's no reason to show it.

```
fileView.hasLoadedDocument = false;
...
fileView.setFileEntry = function ( theNewFileEntry )
{
    if (fileView.hasLoadedDocument)
    {
        // we're potentially loading a NEW document -- save the old
        one.
        if (fileView.theSaveTimer!==-1)
        {
            // clear the interval so we don't save again.
            clearInterval (fileView.theSaveTimer);
            fileView.theSaveTimer = -1;
        }
        fileView.saveDocument();      // force the save so we have the
        up-to-date contents
        documentsView.reloadAvailableDocuments(); // reload our
        directory structure (just in case)
    }

    // now load the correct document
    fileView.theFileEntry = theNewFileEntry;
    fileView.theFilerDocument = {};
    fileView.hasLoadedDocument = true;
    fileView.loadDocument();
}
```

There's nothing new in this code—just a lot of movement of code. Some of this code used to live in `viewWillHide`—to handle saving just before the view disappeared. But this view will never disappear now, so how do we know we should save the document? Turns out that the only time we'll ever know is when a new document is selected. So before we load *that* document, we save the one we've currently got loaded. We can tell if we have one loaded by `fileView.hasLoadedDocument`—we'll set it to `true` once we get a document loaded.

At the end of the function, `fileView.loadDocument()` used to live in `viewWillAppear`. Since we may not have a document selected (especially at the beginning of the app), we aren't trying to load a nonexistent document there, and so we move it here.

Speaking of loading a document, consider the following code:

```
fileView.loadDocument = function ()
{
    // load our document.
    fileView.viewTitle = $ge("fileView_title");
    fileView.viewTitle.innerHTML = fileView.theFileEntry.name.
substr(0,
        fileView.theFileEntry.name.
length-4);
    fileView.theTextElement = $ge("fileView_text");
    fileView.theTextElement.value = "";
    $ge("fileView_text").style.display = "block";
    ...
}
```

The highlighted code in `loadDocument` is what makes sure our `TEXTAREA` element to become visible so that the user can edit it. One thing our code doesn't do is hide it again if there's an error—you should definitely handle an error appropriately in your own code by hiding the `TEXTAREA` element and ensuring that you don't autosave, and so on.

The only other change necessary was the removal of any code handling a back-button event—this leaves `viewWillAppear` and `viewWillHide` completely empty! Note that for Android, we still need a `backButtonPressed` function, but we don't do anything in it. (You could save the document and quit the app, though, if you wanted.)

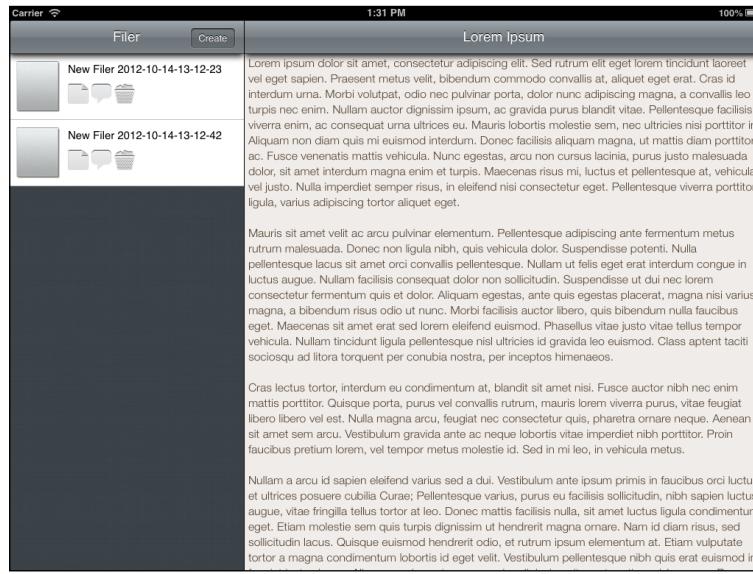
We're *almost* done—one last thing to do before we quit, and that's to make the styles hide our buttons appropriately. In `style.css`, we have:

```
.landscape #documentsView_closeButton,
.landscape #fileView_documentsButton
{
    display: none;
}

.portrait #documentsView_closeButton,
.portrait #fileView_documentsButton
{
    display: inline-block;
}
```

Scaling Up

And that's it! Here's what things look like in landscape mode:



And in portrait mode (with the sidebar visible), it looks like this:



Notice in the image on the right that the sidebar is visible in the portrait orientation—this is because we clicked the **Documents** button to bring it up. Now if we clicked **Close**, the sidebar would disappear.

What did we do?

We converted the Filer app to a split-view application ready for the tablet-sized screen.

What else do I need to know?

If your app includes a couple of views (like this one), then conversion is going to be pretty simple. But if your app needs to have navigation occur in both views, then you're going to run into a hitch as our current framework *doesn't* actually support it.

The view stack as implemented by our current framework assumes only one view on the screen at one time. But in a split-view app, you can have multiple views on the screen. Although the framework plans to support this in the near future, it's not currently available, and so you would need to handle navigation between views on your own. In short, don't try to use view pushing or popping—otherwise funny things will happen.

If you want to follow the framework's progress beyond this book, please visit the framework's Github page: <https://github.com/photokandyStudios/YASMF>.

Game Over..... Wrapping it up

Well, we did it. We've converted an app into two different forms ready for the tablet-sized screen. Although these are the most popular ways of doing it, that doesn't mean that another method might not work better for your app, or a combination of several. It all depends on your content and how the app itself works. You would do well to consult your platform's *Human Interface Guidelines* (links in Appendix A, Quick Design Pattern Reference) too. Only after a thorough examination of your content, layout, graphics, and so on, can you determine what approach would be best—and even then, don't be afraid to experiment and try something else.

Can you take the HEAT? The Hotshot Challenge

This project only covered two methods of scaling to a tablet interface. There are, of course, myriad ways of improving what we've shown here, or using other design patterns to scale to a tablet interface. Why don't you try a few?

- ▶ The split-view pattern has the left split view (or master view) disappearing when in portrait mode. When the **Documents** button is tapped, it appears immediately. Why don't you add some animation instead, to make this less jarring? (Don't forget to animate it when **Close** is tapped too.)
- ▶ Continuing the theme, dismiss the left split view automatically when you select (or create) a document when in portrait mode.
- ▶ Most apps today will allow a gesture to open and close the sidebar (usually a horizontal swipe). Add this to the app.
- ▶ Lastly, when the app opens in portrait mode, there's no real indication of what to do (short of tapping **Documents**)—make the sidebar appear automatically.
- ▶ Instead of using **Create** buttons to create a document, use a "create" item in the document list instead. This can be similar in shape to the existing document items, or not—your call!
- ▶ Take the split view a step further and add a twist! If you've seen the Facebook iOS app, you know that the sidebar actually lives below the main content. The main content can then be slid to the right, which exposes the sidebar underneath. Try and implement this style of app.
- ▶ Try putting the sidebar of the split view in a different position. Putting it on the right is the easiest, but a harder challenge would be the top or the bottom of the screen.

A

Quick Design Pattern Reference

While it is important that your app be unique and stand out from the crowd, it is also important to realize that some things are already pretty standardized, and that there is no need to re-invent the wheel. For example, when building a login screen, you know that you need to ask for a username or an e-mail and a password; that's because this has become a pretty standard design pattern, and users are familiar with it from their various experiences with websites and other apps.

The patterns presented in this appendix are quick sketches. There's no code attached, though where applicable, we've pointed out the projects in which the patterns were used. The appearance of each sketch is based on iOS, but there are parallels for each pattern in most mobile platforms. If you want some really great patterns along with apps that use them, you might want to consider Theresa Neil's book, *Mobile Design Pattern Gallery: UI Patterns for Mobile Applications* published by O'Reilly. She also has a website that has some great information at <http://www.mobiledesignpatterngallery.com>. Some other useful sites that contain lots of real examples of various patterns include:

- ▶ <http://www.mobile-patterns.com/>
- ▶ <http://inspired-ui.com/>
- ▶ <http://pttrns.com/>
- ▶ <http://www.mobiletuxedo.com/category/ui-patterns/>

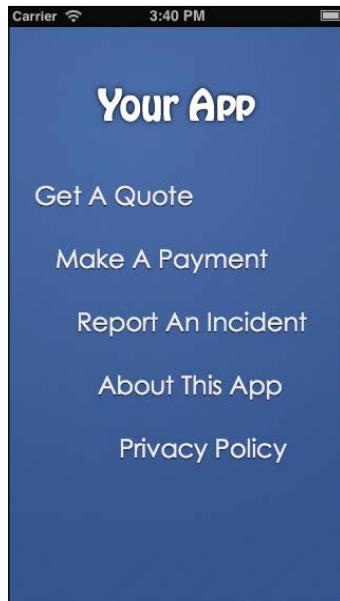
Don't forget to follow the **Human interface guidelines (HIG)** for your platform. On some platforms (such as Apple's iOS and Microsoft's Windows Phone 7 and 8), failure to follow the HIG is bounds for rejection of your app entirely. No matter what, the HIG is designed to ensure that all apps have some degree of consistency and user friendliness. The HIG isn't there to pound you into submission; the guidelines are there for legitimately good reasons.

See these guides for more information:

- ▶ Apple iOS HIG: <http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html>
- ▶ Android's UI guidelines: http://developer.android.com/guide/practices/ui_guidelines/index.html
- ▶ Windows Phone guidelines: [http://msdn.microsoft.com/en-us/library/windowsphone/design/hh202915\(v=vs.92\).aspx](http://msdn.microsoft.com/en-us/library/windowsphone/design/hh202915(v=vs.92).aspx)

The navigation list

The navigation list is a simple navigation pattern for your app. If you have a few items that you want your user to perform, you can use this pattern to present a menu of choices to them. This pattern works well when each topic in your app is different from every other topic. In the following example, getting a quote, making a payment, and reporting an incident would all be very different workflows.



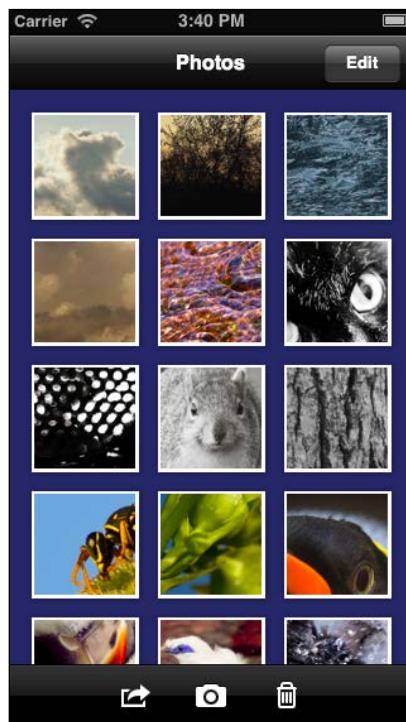
You can take the opportunity to style the screen to a large degree, but be careful *not* to obscure the list items themselves. By all means, include icons and such, but always make sure the text is nice and readable.

If the user needs to log in for any particular item, be sure to keep them logged for any of the other items, if they are used within the same session. (That is, if I make a payment, I shouldn't then have to log in again to report an incident.)

In general, the number of items in this list should be kept to a minimum. If you have to scroll, you might want to rethink your application's hierarchy.

The grid

The grid is a very recognizable navigation pattern for users, and works really well for images and videos (think back to our *Img'n* and *Mem'ry* apps in *Project 6, Say Cheese!* and *Project 7, Let's Go to the Movies!* respectively).



Be sure to display the thumbnails at a large enough scale so that details can be seen. If you want to display a caption, it should appear below the image.

Tapping the thumbnail will generally show the thumbnail at a larger scale. However, if you're using this navigation pattern for, say, sections of your app, each thumbnail would be an icon, and tapping that icon would take you to the appropriate part of the app. In this case, be certain that your icons are all distinct and recognizable.

Using this pattern works best for images and videos, and less so for sections of your app. Although apps have used this pattern in the past (such as an older version of Facebook), most have trended towards using some other mechanism.

When dealing with images and videos, *long pressing* on the thumbnail will typically result in some pop-up list of actions, such as deleting, and moving or the app will allow the user to re-order the list (think of the wiggly home screen on iOS devices).

Carousel 1

There are many different uses for carousels. In the following example, we're displaying a series of documents in a horizontal carousel. Each image below the primary image is an *action*. The user can tap them to do something particular with that document. Also, the *name* of the document (below the primary image) is typically tappable, and will allow the user to rename the document. If you recall, we used this pattern in *Project 3, Being Productive* and *Project 10, Scaling Up*.



Sometimes the action buttons may not be inside the carousel itself; they may be on a toolbar or navigation bar instead. Use whatever fits best for your user.

If more than one document is available, it is a good idea to make sure that part of one of the other documents is visible on the sides of the screen; this helps the user clue-in that the carousel can be scrolled.

The large images should generally be representations of the actual document, typically, a representation of the first page or sheet of a document.

Carousel 2

This carousel is a little different than the last one. It is more typically used to display large images one-by-one or to display tours of an app in a friendly interface. It is also often used to display different sets of information or options in a limited space.



The circles below the main content are optional; if you're viewing an image, users often know to swipe left or right to see the previous or next image. If you're viewing a tour of the app, or different sets of information, however, it is better to display the circles (or the platform's equivalent) so that the user has a good idea of how many pages of content there are.

When displaying the circles, it's best to keep the number of pages to no more than seven or eight. Go beyond this number, and the circles themselves can be distracting and detract from the content.

The login screen

Most apps have them and although they are sometimes the bane of a user's existence (primarily because the password is hard to type on a small screen), they're also critically important.



It's best, if possible, to use an e-mail for the user's unique name. This is a piece of information they already know and have at hand. If that's not possible, however, replace **Email** with *Username*.

Always have the **Password** field obscure the characters. This is typically done with dots that replace each character as they are typed. The last character can be displayed, but should only be shown for a short time. Most platforms will give this to you for free as long as you specify that the input field is a **Password** field.

Sometimes, it's acceptable to have an option to show the password in its entirety. This is typically offered as a checkbox or toggle below the **Password** field. It's not used very often, and only in circumstances where the password itself may be very complicated. One good example is when entering Wi-Fi information; some devices allow you to see the password as you type it without obscuring the letters since Wi-Fi passwords are painfully complex.

Make the **Login** or **Sign in** button obvious. It should beg to be tapped. Give it a different color, larger text—anything to draw the user's attention to it.

Don't forget to give the user a way to reset or retrieve their password (and their username, if you don't use their e-mail). If your app doesn't provide this mechanism, you'll be left with some very upset users.

If it makes sense for your app, you might also want to consider a **Remember Me** option. This is often used in apps where security isn't quite as important as, say, a bank application. If you don't want to have the app remember the user forever, it's often acceptable to remember the user for a couple of weeks or a month. If you do add this feature, be sure to warn the user about the dangers of using this feature on shared devices and on networks they don't trust.

One final note: use SSL. That is, the login process should be over a secure connection.

The sign-up form

The corollary to the login form is the sign-up form. If at all possible, include this form if a user needs to sign in to use your app and provide an option on the log-in form to display this form. Some platforms restrict your ability to display forms like these (especially if payment outside of the platform's app store is possible), but if at all possible, include it for the sake of your user. Nothing's worse than finding a log-in screen, but no way to create a brand new account.



Keep it short and sweet. Don't ask for the user's life story and if you must ask for lots and lots of fields, use a multi-part form. It is best to keep the number of questions to less than seven or eight. The fewer things you ask, the more likely users will sign up.

When asking for e-mail or passwords, it is a good idea to confirm those inputs with a second field. It's all too easy to type an e-mail or password incorrectly; and if no confirmation is in place, the user might continue on having no clue what they did wrong. Then when they can't log in, they'll blame your app.

If you have to ask for personal information, don't forget to explain how you handle the user's private data. Link to your privacy policy. And don't forget to use SSL for the sign-up form.

Make your **Sign Up** button such that it just begs to be tapped on. Give it a pretty color, a large font; it needs to be noticed. Also, if at all possible, provide a **No Thanks** button that allows the user to use your app *without an account*. For some apps, there's a lot of content that can be browsed without a sign in, and some users will use this as a way to *demo* your service. That is, they'll use it to see if they want to create an account or not. Don't put your content behind a login wall if you don't have to.

The table

Tables are everywhere. We've used them in most of our apps, Projects 2 to 7, and also Projects 9 to 10. These can be styled nearly to the point where they're not recognizable as a table, but essentially any repeatable content is a table. So a list of tweets, list of e-mails, or a contact list is a table.

Header 1
Item #1
Item #2
Item #3
Item #4
Item #5
Header 2
Item #1
Item #2
Item #3
Item #4
Item #5

Tables should provide information in a reasonably succinct form. Important text should stand out, while supplementary information should be in a lighter color or smaller font. If an image is provided, be sure that it is large enough to be useful, and that the text wraps nicely around it. Think carefully about the position of the image; the position might convey important information (for example, in a messaging app images on the left might refer to messages being sent to you, while your image on the right might refer to messages that you sent out).

For those platforms that support disclosure icons (arrows, checkmarks, and so on), be sure to put them in the correct place for your platform. If tapping them does something other than the action that would occur by tapping the row, be sure to give enough of a tappable area for the user to target.

Some apps have added lots of gestures to table rows. TweetBot is a good example where swiping one direction will do one action, while swiping another direction will do another. Generally this is pretty novel, and users aren't going to do a lot of swiping on table rows, except if it looks like the content is deletable. In this case, they might swipe right-to-left to attempt to delete a row. (If the row is in fact something that can be deleted, then the appearance of a **Delete** button would be appropriate.) An example of this is shown in the following screenshot:



The list of choices

These appear in a lot of forms, but they all boil down to the same thing: the app wants you to pick something from a list. The list can be single selection; that is, only one item can be selected, or it can allow multiple selections (which is usually represented with a radio button, check mark, or other similar icon).



The appearance can vary drastically, from a *Select* list on iOS, to an iOS *ActionSheet*, to an Android menu. Pick whatever works best for the situation and the platform.

Don't use these to pick numbers, unless there are very few. No one wants to scroll through a hundred rows numbered 1-100 just to pick 97. Use an input field for that. But if the possible values are 25, 50, 75, and 100, then this is okay (though it might be wise to spell the numbers out).

Doing things in bulk

There are many different ways to do actions in bulk, but the following example is a very common pattern. We used something similar in *Project 7, Let's Go to the Movies!* and *Project 8, Playing Around*.



In the preceding pattern, tapping on an item will mark that item as selected. This can be by placing a checkmark by it, changing the border color (which is what we used), or by highlighting it using some other method. As long as it is obvious what items are selected and what items aren't, there are a lot of possibilities here.

Once the selection is made, then the actions at the bottom of this pattern come into play. The user might delete the items, or they might want to do something else. If you decide to use icons and not words, be sure to use icons that users already understand, for example, a trashcan works well in place of *delete*.

Do try to keep the number of actions possible to the absolute minimum, especially when dealing with the limited real estate of a mobile phone.

Searching

If your app displays a lot of data, it's almost inevitable that you'll need to provide a mechanism for searching through that data.



The preceding pattern is one of the most common on most platforms. The user can type in the **Search** field and the results will appear in a table below the **Search** field. The **Cancel** button is not always present. Sometimes, it's just a small icon, but this depends on the app and the platform.

Whether or not the search occurs while the user is typing or it requires them to tap the **Search** button in their onscreen keyboard is up to you. If you can quickly search the dataset, it might be wise to display search results as the user types. This way they can quickly see the result set being whittled down to what they want.

If, however, it takes a long time to search, then it is better to wait until the user types what they want and then tells you when to search. Then you can take your time (don't forget to display some sort of notice that your app is thinking), and then display the results when the search is complete.

Another common pattern when searching is the need to scope the search, as in the following pattern:



Here, the buttons below the search bar are segmented buttons, something typical of iOS. There are equivalents for most platforms. If one of them is tapped, it is highlighted, and the search only occurs within the displayed scope.

In this example, if the user tapped **To**, then the search would only occur for the **To** portion of each item.

When it comes to searching, there are also often requests to sort and filter the data as well. You can usually accommodate these by using a toolbar with the sort and filter buttons on it, and then display a menu listing the various options. For example, tapping a **Sort** button could display a menu of first name, last name, account number, and so on. There are various patterns that one could use to implement more complex sorting and filtering, but if possible, keep it simple.

Some things to keep in mind

Often, the details are what matter. Here are some tips to help make sure you have not only a great-looking app, but a great-feeling app as well:

- ▶ Avoid novel or undiscoverable interactions. Or, if you do want to use such interactions, try to also provide a second discoverable method for achieving the same thing. One could argue that the slide menu a la Facebook could be considered novel, but should the gesture not be discovered, the button that triggers the menu is highly visible and likely to be tapped.
- ▶ Respect your user's expectations of what UI elements and gestures do. At first this seems obvious; you wouldn't send an e-mail by clicking on a trashcan. That said, there are plenty of ways you can do things that the user didn't expect, even in a subtle manner. Your user's expectations are highly specific according to the platform, so following your platform's HIG will help out significantly in this area.
- ▶ Do blend in. By this I mean that your app should look like it belongs on your user's device. This means that your app should respect the HIG for the platform. It also means your app should give the appearance of being a native app. Failing to do so may cause your users to feel like your app is a second-class citizen on the device. By giving the appearance of being native, your app should also do its best to feel native – that is it should have speed and response close to that of a native app. (It may not always be easy, or even possible, on some platforms using PhoneGap. In this case, you should try to get as close as you can without impacting your project and its timeline.)
- ▶ Be a perfectionist. By this I mean that you should make sure that all your objects align nicely, your textures blend seamlessly, images are scaled correctly (especially according to aspect ratio), and so on. This requires painstaking attention to detail, but in the end your app will look and feel better for it.
- ▶ Be responsive. Whenever possible, avoid freezing the user interface. If you must freeze the UI, then put up an indicator to the user so they know their inputs will be ignored. When it comes to being responsive, this isn't simply being responsive to a tap on a button, but also with regards to scrolling performance. If an app scrolls in a jerky fashion, the app will feel slow.
- ▶ Sip your data. While your app may often be used on a Wi-Fi connection, don't forget about your users who may have to deal with a cellular connection instead. Not only might their connection itself be slower, but they are usually under pretty onerous data caps. Cache extensively. Avoid downloading what you've already downloaded. If it can be compressed, by all means, compress it. Alternatively, give your user an *out* – if your app is going to use a lot of data no matter what, you might want to let them disable the portions of your app that use a lot of data when on a cellular connection.

Summary

These are only a few of the myriad design patterns available. When possible, do some research and see if there is a pattern for what you're trying to do in your app. Yes, sometimes, your app will require that you do something totally unique, but more often than not you'll find a successful pattern that many apps already use. Your app will be more usable, and your user will thank you for that. You'll also field fewer support calls about how to use your app.

B

Installing ShareKit 2.0

It's astonishing, quite frankly, that there's no system-defined framework on iOS that enables easy sharing across many different services. iOS 5 does provide the Twitter framework, and iOS 6 expands on it by providing the Facebook framework, but there's no system-defined framework that you can use to easily share to a wider variety of services. Maybe iOS 7 will address that need but don't hold your breath.

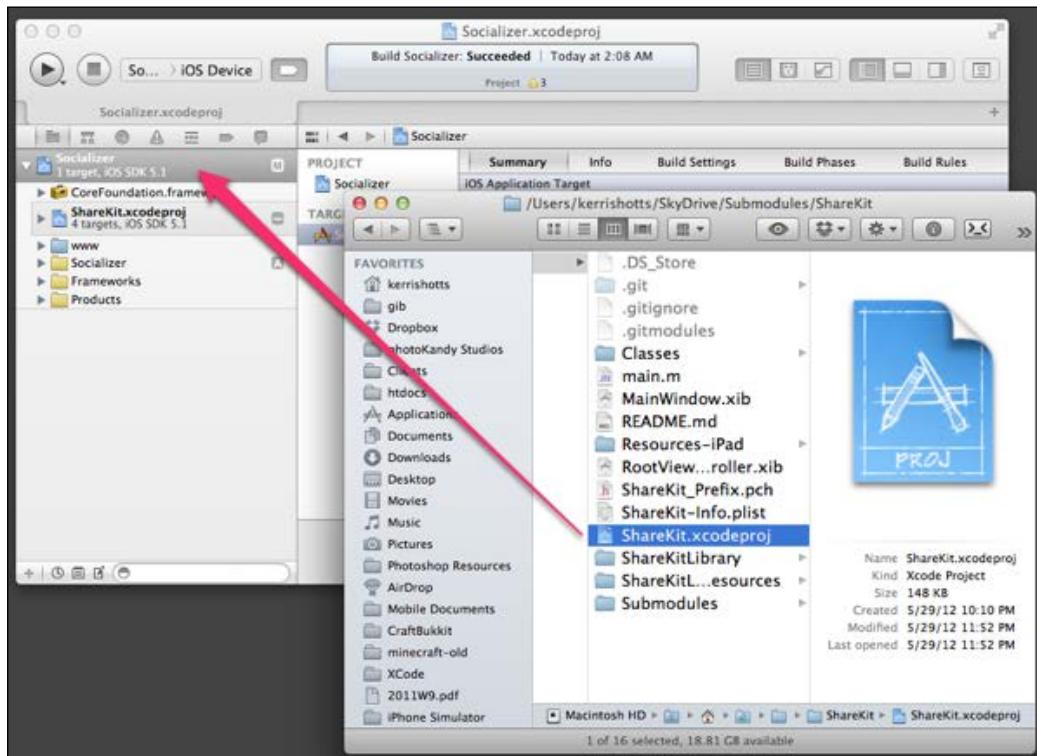
Thankfully, there is an open source framework called ShareKit that helps to address this. It gives us an easy way to share to any number of services (assuming one has API keys for each), and it's not hard to call from PhoneGap. Unfortunately, installing it is also a royal pain.

We've listed the steps that worked for us as follows. Some of these steps come from ShareKit's Wiki at <https://github.com/ShareKit/ShareKit/wiki/Installing-sharekit>, so it would be a good idea to refer to that document as well.

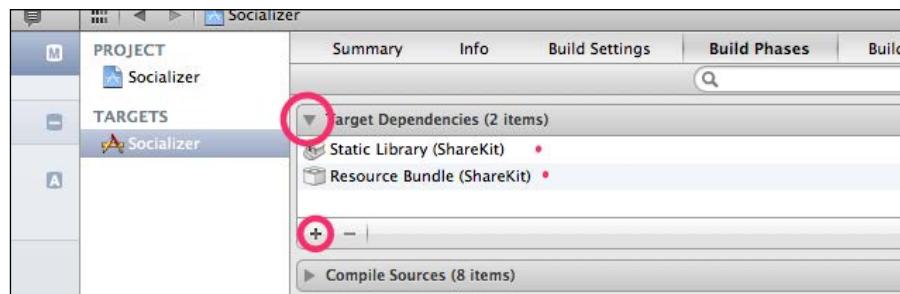
1. You need to get the ShareKit files. ShareKit recommends using Git to do this, or you can use our copy – it is your option. Frankly, it will be easier to use the copy in the `Submodules` directory in our code package. It won't be the latest version of ShareKit, but it has the required source code changes to work.

Installing ShareKit 2.0

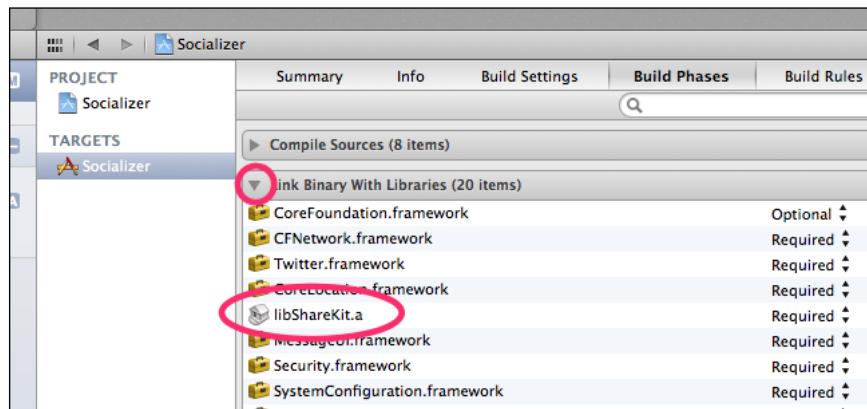
2. Add ShareKit to the project using the following steps:
 - i. Navigate to Submodules/ShareKit and drag ShareKit.xcodeproj to your project as shown in the following screenshot:



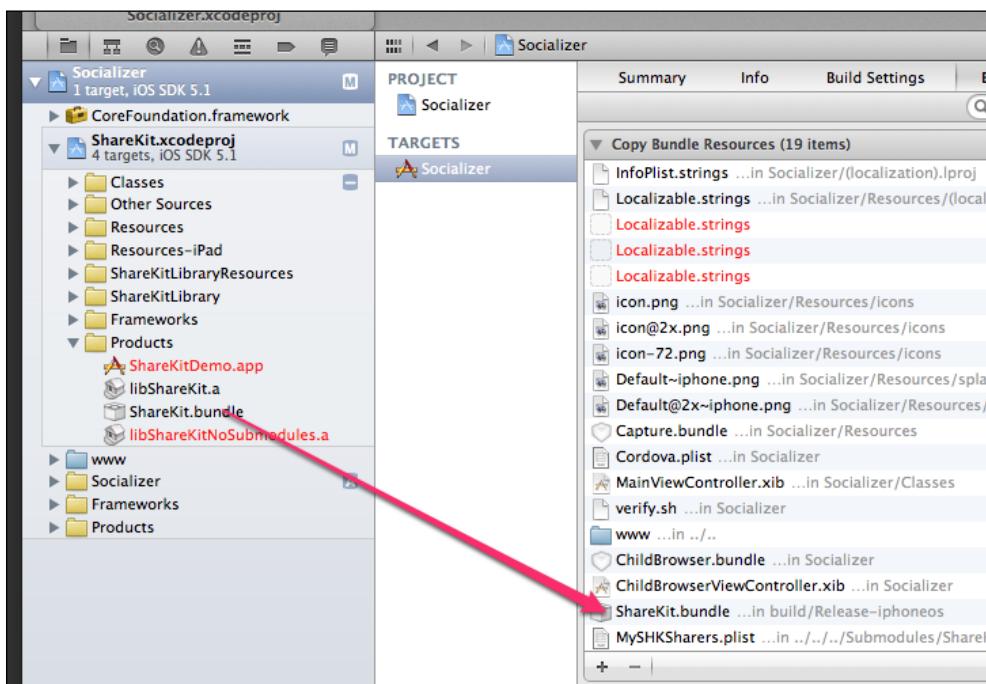
3. We need to make various adjustments to permit ShareKit to compile:
 - i. Go to your project's settings, click on **Build Phases**, and expand **Target Dependencies**. Add the **Static Library (ShareKit)** and **Resource Bundle (ShareKit)** dependencies as shown in the following screenshot:



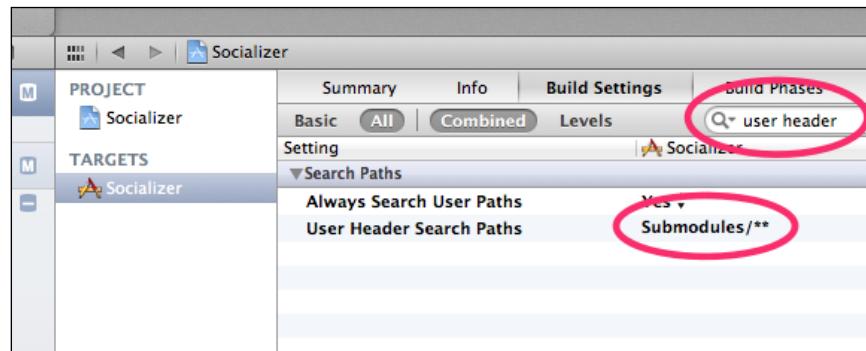
- ii. Expand **Link Binary With Libraries** and add **libShareKit.a** as shown in the following screenshot:



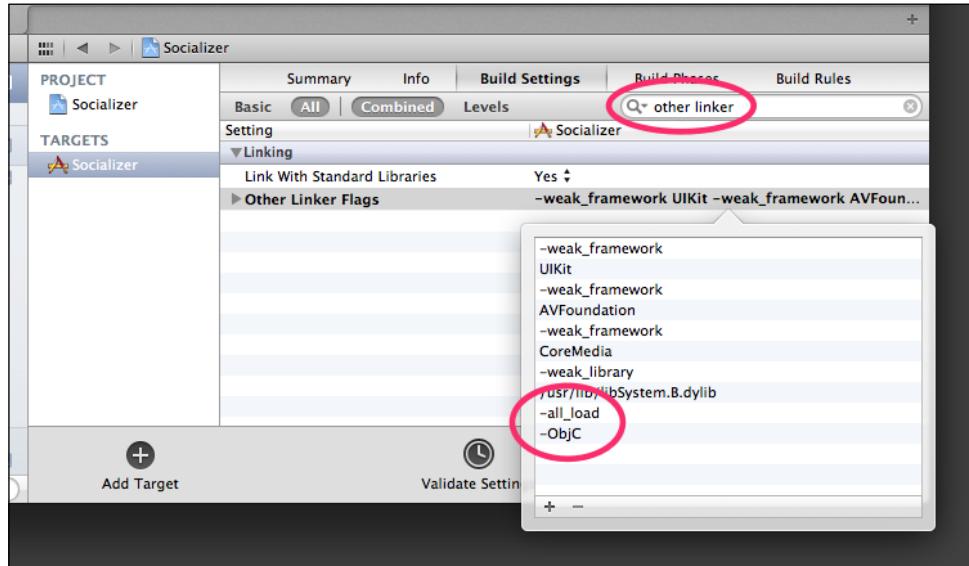
- iii. Expand **Copy Bundle Resources** and copy Sharekit's resource bundle (located in the ShareKit subprocess under the `Products` folder and named `ShareKit.bundle`) to the list as shown in the following screenshot:



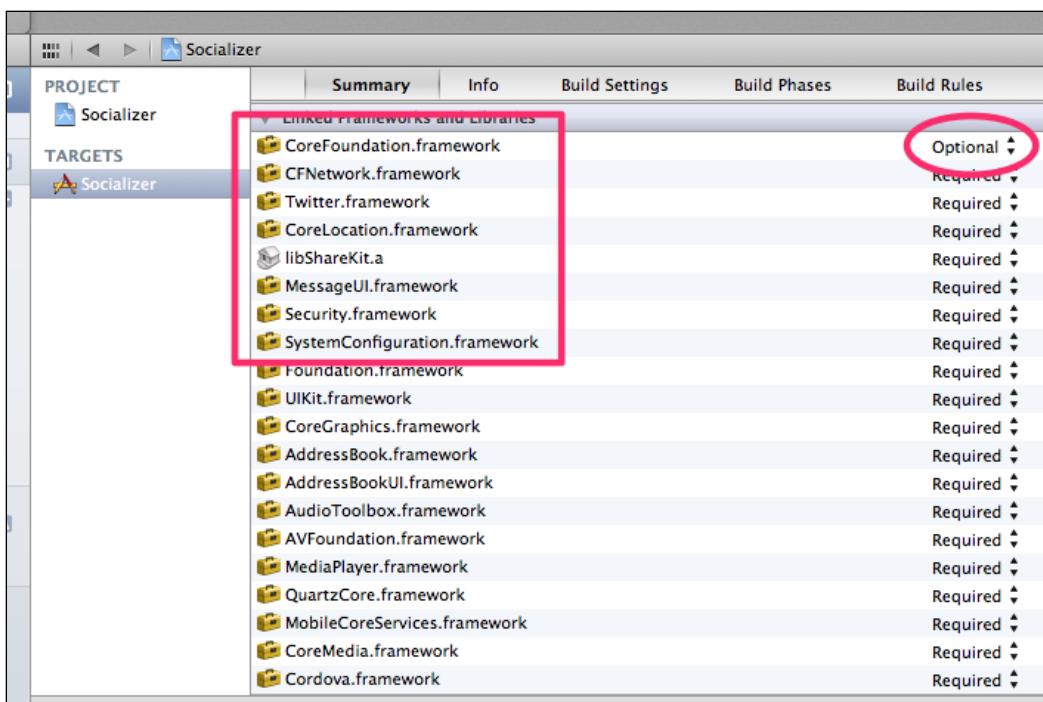
4. Next we need to adjust the header search paths and linker flags:
 - i. Switch to **Build Settings**.
 - ii. Search for user header.
 - iii. Double-click the row named **User Header Search Paths**.
 - iv. Add **Submodules** and check the checkbox next to the new entry.
It should look like **Submodules/**** as seen in the following screenshot:



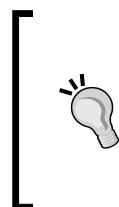
- v. Now search for other linker.
- vi. Double-click the row named **Other Linker Flags**.
- vii. Add `-all_load` and, on a separate line, `-ObjC`, if not already present as shown in the following screenshot:



5. We need to add the various Apple Frameworks this plugin requires:
 - i. Go back to the **Summary** tab.
 - ii. Scroll down to **Linked Libraries and Frameworks**.
 - iii. Add `SystemConfiguration.framework`, `Security.framework`, `MessageUI.framework`, `CFNetwork.framework` (required for Flickr sharing), `CoreLocation.framework` (Required for FourSquare sharing), `Twitter.Framework` and `CoreFoundation.framework`.
 - iv. Make `CoreFoundation.framework` **Optional** as shown in the following screenshot:



At this point you should be able to build the project with no errors.



You may receive an error regarding a failure to find a protocol definition if you aren't using our copy of ShareKit. This occurs on the Flickr code; to get rid of the error, just remove `OFFlickrAPIRequestDelegate`, on the line causing the error. Since we don't use Flickr in the examples, this won't cause a problem. (It might, however, should you decide to use Flickr in the future.)

We need to obtain API keys for the services we intend on using. In our project, we used Twitter, Facebook, and ReadItLater (now Pocket). You're free to use whatever services you would like but you must use your own API keys to do so. (Our code package does not provide you with API keys.) Visit <https://github.com/ShareKit/ShareKit/wiki/3rd-party-api-links> for some good information about how to obtain these keys. Please refer the following steps:

1. We have to configure ShareKit 2.0 with these keys so create a new file in XCode and set it to subclass DefaultSHKConfigurator. We used MySHKConfiguration for the name, but you can use whatever you like. Be sure to save this new file to your project and not the subproject.
2. Look at ShareKit | Classes | ShareKit | Sharers | Configuration | DefaultSHKConfigurator.m for the methods you can override. Each of these is labeled with comments indicating the code specific to each social networking service.
3. Copy the code you need to override and paste it into your own configurator file.
4. Alter each method to return the appropriate API key, secret, or other string.

Here's an example:

```
- (NSString*)appName {
    return @"Socializer";
}

- (NSString*)appURL {
    return @"http://www.example.com";
}

- (NSString*)facebookAppId {
    return @"122308337901327";
}

- (NSString*)facebookLocalAppId {
    return @"";
}

- (NSArray*)facebookListOfPermissions {
    return [NSArray arrayWithObjects:@"publish_stream", @"offline_access", nil];
}

- (NSString*)readItLaterKey {
    return @"apikey";
}
```

```
// Twitter - http://dev.twitter.com/apps/newhttp://dev.twitter.  
com/apps/new  
- (NSNumber*)forcePreIOS5TwitterAccess {  
    return [NSNumber numberWithBool:true];  
}  
  
- (NSString*)twitterConsumerKey {  
    return @"apikey";  
}  
  
- (NSString*)twitterSecret {  
    return @"apikey";  
}  
// You need to set this if using OAuth, see note above (xAuth  
users can skip it)  
- (NSString*)twitterCallbackUrl {  
    return @"http://www.example.com/callback";  
}  
// To use xAuth, set to 1  
- (NSNumber*)twitterUseXAuth {  
    return [NSNumber numberWithInt:0];  
}  
- (NSString*)twitterUsername {  
    return @"";  
}
```

5. Navigate to your project's /Classes/AppDelegate.m file and add the following at the end of the didFinishLaunchingWithOptions: method, just prior to the return statement (assuming your file is named MySHKConfigurator):

```
DefaultSHKConfigurator *configurator = [[MySHKConfigurator alloc]  
init];  
[SHKConfiguration sharedInstanceWithConfigurator:configurator];
```

6. Also add the following code to the import section:

```
#import "SHK.h"  
#import "SHKConfiguration.h"  
#import "MySHKConfiguration.h"  
#import "SHKFacebook.h"
```

7. Next, we need to enable offline sharing (optional):

- Add [SHK flushOfflineQueue]; just below our newly added code in the last step.

8. If you're supporting Facebook, you need to support Single Sign On (SSO):

- ❑ Add the following code at the end of the `handleOpenURL:` method in `AppDelegate.m` (replacing the `return YES;` statement):

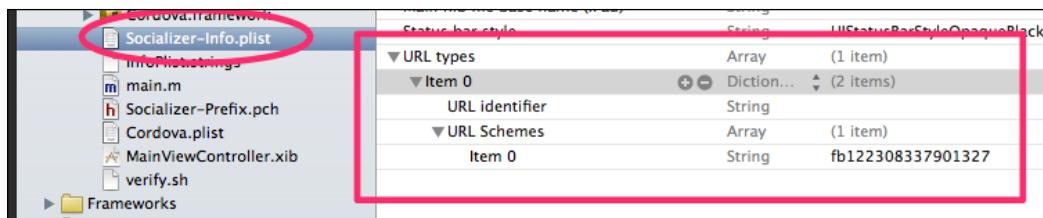
```
    return [self handleOpenURL:url];
}

- (BOOL)handleOpenURL:(NSURL*)url
{
    NSString* scheme = [url scheme];
    NSString* prefix = [NSString stringWithFormat:@"fb%@", SHKCONFIG.facebookAppId];
    if ([scheme hasPrefix:prefix])
        return [SHKFacebook handleOpenURL:url];
    return YES;
}

- (BOOL)application:(UIApplication *)application
openURL:(NSURL *)url sourceApplication:(NSString *)
sourceApplication annotation:(id)annotation
{
    return [self handleOpenURL:url];
}
```

- ❑ Add a custom URL scheme to your project's `info.plist` file:

- ❑ Add **URL Types**
- ❑ Add **Item 0**
- ❑ Add **URL Schemes**
- ❑ Add **Item 0** with a value of "fb" and your app ID from Facebook



To determine how your `AppDelegate.m` file should look when complete, verify your file against ours in the downloadable code package.

We're almost done; all we need now is to indicate which social networks we don't support:

1. COPY the `SHKSharers.plist` file from the ShareKit subproject into your project. Be sure to name it something different, like `MySHKSharers.plist`.
 2. Remove the rows corresponding to the networks you don't support.
- Here's an example of ours:

Key	Type	Value
▼ actions	Array	(7 items)
Item 0	String	SHKPhotoAlbum
Item 1	String	SHKCopy
Item 2	String	SHKMail
Item 3	String	SHKPrint
Item 4	String	SHKSafari
Item 5	String	SHKTextMessage
Item 6	String	SHKLogout
▼ services	Array	(3 items)
Item 0	String	SHKTwitter
Item 1	String	SHKFacebook
Item 2	String	SHKReadItLater

3. Add the following code to the end of your configurator file:

```
- (NSString*)sharersPlistName {
    return @"MySHKSharers.plist";
}
```

At this point, you should have the ShareKit framework integrated into your project. You'll want to refer to *Project 2, Let's Get Social!* to finish integrating it with the ShareKit PhoneGap plugin.

I wish I could give you more to watch out for regarding build errors. Unfortunately, every time I do this, I seem to come across new errors I hadn't seen the time before. (The Flickr error previously in this document is a good example!) It usually takes a cross-your-fingers-and-hope methodology to fix these kinds of errors (such as my removal of the protocol causing the error), but removing the error itself might not always work. If you have problems, it would be a good idea to ask the PhoneGap community or the ShareKit community if they have any suggestions.

Index

Symbols

\$getLocale() function 39
@catch block 245
@try block 245

A

accelerometer, Cave Runner game
handling 286-289
actionButton 161
actionButtonPressed() method 163
ActionSheet 294
ActionSheets, Socializer app
adding 308, 309
addAnswer method 27
addNode() method 155
addTranslation method 28
amCalibrated variable 287
Android
used, for implementing video
thumbnail plugin 247-250
Android UI's guidelines
URL 340
answerAtIndex function 24, 26
App-Bits
URL 113
APP.init function 60
Apple 307
Apple iOS HIG
URL 340
apps
tips, for enhancement 353
APP.start() function 60
attachGestureRecognizer() method 187

B

backButtonPressed function 101, 335
bias variable 272

C

canvas, Cave Runner game
setting up 276-279
Canvas tag 208
CAPTURE API 254
captureVideo() method 254
Carousel 1 342, 343
Carousel 2 343, 344
carousels 342
Cave Runner game
accelerometer, handling 286-289
building 261
canvas, setting up 276-279
designing 263-265
features 262
implementing 262
levels, generating 271-275
options view, implementing 266-271
prerequisites 262
slow-mo feeling, applying 279, 280
touch-based input, handling 284-286
updates, performing 280-283
working 261, 262
center property 167
channelWidth variable 273
check mark 349
ChildBrowser plugin
about 70, 237, 294
configuring, for Android 91
configuring, for iOS 88-90

completion method 50, 115, 118, 176
completion variable 115
copyDocument() method 134, 201
correctOrientation property 224
createDocument() method 120, 202
createNewDocument() method 203
CSS 293
currentPositionMarker property 162
cWidth variable 272

D

data model, Filer
designing 113, 114
Documents model, implementing 115-124
FilerDocuments 114
implementing 115

data model, Img app
designing 211-213

data model, My Path
designing 149, 150
document manager model 150
implementing 151-156
PathRecDocumentCollection model 150
PathRecDocumentItem 150

data model properties, VoiceRec
duration 178
durationTimer 179
media 178
paused 179
playing 179
position 178
positionTimer 179
recording 179
title 178

data model, Quiz Time!
data formatting 29
designing 20-22
implementing 23-33
localization efforts 29
translation effort 29

data model, Socializer
designing 76, 77
implementing 77-84

data model, VoiceRec
designing 175, 176
implementing 177-184

deleteDocumentAtIndex method 119
deleteDocument() method 134, 197
deleteSelectedPictures() method 225
design pattern reference
actions, in bulk 350
Carousel 1 342, 343
Carousel 2 343, 344
grid 341, 342
list of choices 349
login screen 344, 345
navigation list 340, 341
searching mechanism 351, 352
sign up form 346
table 347, 348

destinationType property 223
detail view 326
directoryEntry function 117
disableDefaultUI property 167
dispatchFailure() function 114, 116, 123, 177
displayAvailableDocuments() 127
DIV tag 321
documentActions class 197
documentContainerTapped() method 158, 198, 220, 258
documentIconTapped() method 158
documentIterator() method 199, 251, 253
document manager, My Path
changing 157-159

documents property 115
documents view 319
documentsView.documentIterator() 128
documentsView_documentTemplate
template 129

documents view, Filer
implementing 125-134
issues 135

documentsView method 140
documentTapArea 158
document view 319
document view, Img app
implementing 213-230

documentView method 137
doPicture() method 222
duration property 177, 178
durationTimer method 177

E

EmailComposer 295
e-mail composer, Socializer app
 adding 314-316
encodingType property 224
end view, Quiz Time!
 implementing 54-63
entitleDocument() method 137
eventStart() method 189
execute method 248
extractVideoThumbnail method 242

F

failure function 114, 176, 223
failure variable 115
File API
 about 125, 207
 using 109
fileEntry property 114, 116, 176
file() method 123
fileName property 176
Filer
 building 109
 data model, designing 113
 data model, implementing 115
 documents view, implementing 125-134
 features 110
 file view, implementing 136-140
 implementing 110
 improvements 141
 prerequisites 110
 user interface, designing 110, 111
 working 109
 wrapping up 141
.FileReader variable 123
fileSystem property 114, 116
fileType property 176
file view, Filer
 implementing 136-140
fileView method 130
fileView_text 137
for loop 213

G

gameView.html file 271
gameView_questionArea element 45
game view, Quiz Time!
 implementing 41-49
gameView_scoreArea element 45
gameView.selectAnswer() method 44
geolocation 143
geolocationUpdate() method 164
gestures 185
gesture support, VoiceRec
 implementing 185-194
getCorrectAnswer function 26
getDocuments() method 117
getDocumentsSuccess() method 117
getFileName method 177
getFileSystem() function 116
getMaxCount() method 77
getNodes() method 155
getPlaybackPosition method 177
getProfileImageUrl() method 77, 80
getScreenName() method 77
getSearchPhrase() method 77
getStream() method 77, 82
getTimeline() function 80
getUserData() method 77, 80
globalAlert 217
Google Maps API
 URL 145
gotFile() method 123
gotFileWriter 124
grid 341, 342

H

hashtag 74
hideView function 41
HTML 293
Human interface guidelines (HIG) 340

I

image view, Imgtn app
 implementing 230-233

Imgn app
building 207
data model, designing 211-213
document view, implementing 213-230
features 208
image view, implementing 230-233
implementing 208
prerequisites 208
user interface, designing 209-211
working 207, 208
importPicture() method 224
include function 50
indexOf() method 213, 220
initializeView() method 44, 56, 97, 127, 137, 163, 215, 269, 300
inSelectionMode 212, 216
installation, ShareKit 2.0 355-363
iOS
used, for implementing video thumbnail plugin 240-246
isRecording method 177
iUI 9

J

JavaScript 293
jQuery/Globalize framework 66
jQuery Mobile 9
jQuery Touch 9

K

keepMapCentered property 162, 163

L

lastKnownPosition property 162-164
lastScrollTop property 96
LatLng object 164
levels, Cave Runner game
generating 271-275
loadFileSystem() function 116
loadJSON() method 79, 83, 85
loadStreamFor() method 302
loadStream() method 77, 83
loadTweet() method 104, 106, 303
loadTwitterUsers() method 84

localStorage
using 144
login screen 344, 345
lookupTranslation function 31

M

main view, VoiceRec
implementing 194-205
map property 162
mapTypeId property 167
MapView_mapCanvas element
about 166
center property 167
disableDefaultUI property 167
mapTypeId property 167
zoom property 167
map view, My Path
implementing 159-168
master-detail 318
master-detail pattern 326
master view 326
media property 177, 178
mediaSuccess() method 199
mediaType property 224
Mem'ry app
building 235, 236
features 236
implementing 236
preparing, for video thumbnail plugin 237-240
prerequisites 237
video, importing 253-256
video playback, implementing 256-259
video recording, implementing 253-256
video thumbnail plugin, implementing
for Android 247-250
video thumbnail plugin, implementing
for iOS 240-246
video thumbnails, displaying 251-253
working 236
MessageBox 295
message box, Socializer app
adding 310, 311
My Path
building 143
data model, designing 149-151
data model, implementing 151-156

document manager, changing 157-159
features 144
implementing 144
improvements 169
map view, implementing 159-167
prerequisites 145
UI, designing 145-148
working 144
wrapping up 168

N

NavigationBar 295
navigationBar class 35
navigation bar, Socializer app
 adding 298-303
navigation list 340, 341
navigator.geolocation.watchPosition() 167
nextQuestion() method 46, 47

O

OFFlickrAPIRequestDelegate 359
onclick event 44
onClick handler 137
onload() method 229
onreadystatechange function 52
openDocumentAtIndex() method 130
options view, Cave Runner game
 implementing 266-271

P

panTo() method 164
PathRecDocument data model, My Path
 addNode() method 150
 completion property 150
 failure property 150
 fileEntry property 150
 filename property 150
 getNodeAtIndex() method 150
 getNodeCount() method 150
 getNodes() method 150
 getTitle() method 150
 nodes 150
 serialize() method 151
 setNodes() method 150
 setTitle() method 150

state property 150
title property 150
PathRecDocumentItem data model, My Path
 altitude property 150
 getGoogleLatLang() property 150
 getGoogleMarker() property 150
 getLatLong() property 150
 get() property 150
 heading property 150
 latitude property 150
 longitude property 150
 serialize() property 150
 setPosition() 150
 speed property 150
 timestamp property 150

patterns
 sites, for example 339

paused method 177

PhoneGap

 Cave Runner game, building 261
 Filer, building 109
 Imgn app, building 207
 Mem'ry app, building 235, 236
 My Path, building 143
 Quiz Time!, building 9
 Socializer, building 69
 VoiceRec, building 171

PhoneGap 2.2.0

 URL 11

PhoneGap (Cordova) 2.2.0 12

pickers, Socializer app
 adding 312-314

PickerView 295

PKLOC.addTranslation() function 28, 37

PKLOC.localizedText array 28

PKUI.CORE.pushView() method 46

PKUI.CORE.showView() method 46

PKUTIL.include() function 45

PKUTIL.instanceOfTemplate() 129

PKUTIL.load() method 50

playing method 177

plugins, Socializer

 configuring 86-90

 configuring, for Android 91, 92

 configuring, for iOS 88

 installing 294-298

points[] array 272
Polyline 162
popView() method 41, 46
position property 177, 178
positionTimer method 177
prerequisites, Quiz Time!
 code snippet 12
 Eclipse for Android development 11
 jQuery/Globalize repository 14
 PhoneGap 2.2.0 11
 project, for various platforms 11
 Xcode for iOS development 11
 YASMF framework 11
push() method 212, 221
pushView() method 40

Q

quality property 223
Quiz Time!
 about 9
 building 9
 collection of questions, designing 21
 data model, designing 20
 data model, implementing 23
 end view, implementing 54
 enhancements 68
 features 10
 game view, implementing 41
 implementing 11, 58, 59
 prerequisites 11
 question model, designing 20
 start view, implementing 34
 UI/interactions, designing 14-19
 working 10
 wrapping up 67

R

radio button 349
radiusToRecognition parameter 193
recognizeGesture() method 187, 188
recording method 177
recordingPath property 162
releaseResources 177
reloadAvailableDocuments() method 128
renameDocument() method 201

renderVideoThumbnail() method 252, 253
rndWidth variable 272
rootContainer element 328

S

saveToPhotoAlbum property 224
scaled-up UI
 designing 318-321
 implementing 321-326
scale-it-up 318
scaling up project
 about 317
 building 317
 features 318
 implementing 318
 prerequisites 318
 working 317, 318
SCRIPT tags 64
searching 351, 352
selectedItems array 212, 217, 221
Sencha Touch 9
serialize() method 124, 156
setCorrectAnswer method 27
setFileName method 177
setImage() method 232, 257
setInterval() method 137
setMaxCount() method 77
setPlaybackPosition method 177
setPosition() method 153
setScreenName() method 77
setSearchPhrase() method 77
setTweet() method 104
ShareKit 69
ShareKit 2.0
 downloading 88
 installing 355-363
ShareKit plugin
 configuring, for iOS 90
share() method 106, 310, 315
Share plugin
 configuring, for Android 92
showView function 41
sign up form 346
SimpleGesture class 186
smartphones 261

Socializer
about 293
ActionSheets, adding 308, 309
building 69
challenges 108
data model, designing 76
data model, implementing 77-84
e-mail composer, adding 314-316
features 70, 294
implementing 70, 294
message box, adding 310, 311
navigation bar, adding 298-303
pickers, adding 312-314
plugins, configuring 86, 88
plugins, installing 294-298
prerequisites 71, 294
social view, implementing 93-101
tab bar, adding 304-307
tweet view, implementing 101-106
user interface, designing 72-75
working 69, 70, 293
wrapping up 108
social networking 69
socialView.html 300, 302
social view, Socializer
implementing 93-101
sourceType property 224
sparse array 212
splice() method 213, 220
split view 318
split-view UI
designing 326, 327
implementing 328-337
startApp() method 301
startRecording 177
start view 319
startView.html 300, 301
start view, Quiz Time!
implementing 34-40
state property 115
stopRecording 177
Submodules directory 355
substituteVariables function 31
substr() method 129
Subtle Patterns
URL 113
success/failure methods 117
success function 223, 226
Success method 114 177

T

TabBar 295
tab bar, Socializer app
adding 304-307
table 347, 348
takePicture() method 224
TEXTAREA element 109
theParsedData variable 86
theTweet property 104
title property 177, 178
toggleSelection() method 217, 218
touch-based input, Cave Runner game
handling 284-286
trackButton 161
tweetView.html 300, 302
tweet view, Socializer
implementing 101-106
TwitterStream object 76
TwitterUser object 76

U

UI, My Path
designing 145-147
documents view 147
map view 148
simple view 146
UI, Quiz Time!
designing 14-20
UI, Socializer
designing 72-75
updateAccelerometer method 287
updateDuration method 177
updateGesture() method 188, 189
updatePosition method 177
updates, Cave Runner game
performing 280-283
user interface, Filer
designing 110
documents view 110, 111
documents view, working 111
file view 112

icons 113
images 113
large paper image 113
navigation bar 113
resources, creating 112
start view 110

user interface, Img'n app
designing 209-211
mockup 209

user interface, VoiceRec
Delete button 173
designing 172-175
mockup 173
Play and Pause buttons 173
Record button 174

V

video
about 235
importing 253-256

video playback
implementing 256-259

video recording
about 235
implementing 253-256

VIDEO tag 236

video thumbnail plugin
implementing, for Android 247-246
preparing for 237-240

video thumbnails
displaying 251-253

viewBackground class 35

viewDidAppear() method 167

viewDidHide() method 233

viewWillAppear() function 46, 101, 106, 140, 167

viewWillHide() method 101, 140

VoiceRec
building 171
data model, designing 175
data model, implementing 177-184
enhancements 206
features 172
gesture support, implementing 185
implementing 172
main view, implementing 194
prerequisites 172
user interface, designing 172
working 172
wrapping up 206

VoiceRecDocument function 199

W

wallEvery variable 273

watchID property 162

Windows Phone guidelines
URL 340

X

XMLHttpRequest 52

Y

YASMF (Yet Another Simple Mobile Framework)
about 9
URL 9

Z

zoom property 167



Thank you for buying PhoneGap 2.x Mobile Application Development HOTSHOT

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

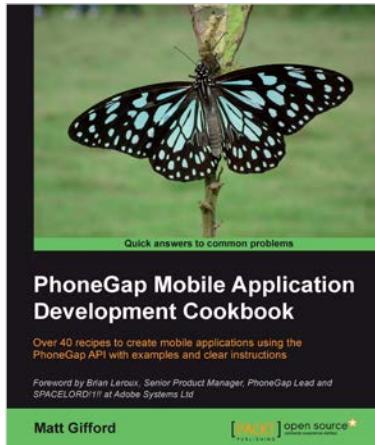
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

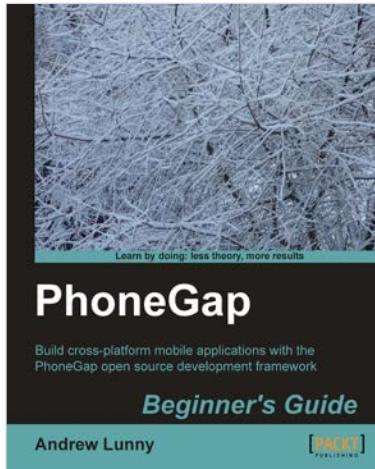


PhoneGap Mobile Application Development Cookbook

ISBN: 978-1-84951-858-1 Paperback: 320 pages

Over 40 recipes to create mobile applications using the PhoneGap API with examples and clear instructions

1. Use the PhoneGap API to create native mobile applications that work on a wide range of mobile devices
2. Discover the native device features and functions you can access and include within your applications
3. Packed with clear and concise examples to show you how to easily build native mobile applications



PhoneGap Beginner's Guide

ISBN: 978-1-84951-536-8 Paperback: 328 pages

Build cross-platform mobile applications with the PhoneGap open source development framework

1. Learn how to use the PhoneGap mobile application framework
2. Develop cross-platform code for iOS, Android, BlackBerry, and more
3. Write robust and extensible JavaScript code
4. Master new HTML5 and CSS3 APIs

Please check www.PacktPub.com for information on our titles



WordPress Mobile Applications with PhoneGap

ISBN: 978-1-84951-986-1 Paperback: 96 pages

A straightforward, example-based guide to leveraging your web development skills to build mobile applications using WordPress, jQuery, jQuery Mobile, and PhoneGap

1. Discover how we can leverage on Wordpress as a content management system and serve content to mobile apps by exposing its API
2. Learn how to build geolocation mobile applications using Wordpress and PhoneGap
3. Step-by-step instructions on how you can make use of jQuery and jQuery mobile to provide an interface between Wordpress and your PhoneGap app



Android 3.0 Application Development Cookbook

ISBN: 978-1-84951-294-7 Paperback: 272 pages

Over 70 working recipes covering every aspect of Android development

1. Written for Android 3.0 but also applicable to lower versions
2. Quickly develop applications that take advantage of the very latest mobile technologies, including web apps, sensors, and touch screens
3. Part of Packt's Cookbook series: Discover tips and tricks for varied and imaginative uses of the latest Android features

Please check www.PacktPub.com for information on our titles