



Quick answers to common problems

Appcelerator Titanium Business Application Development Cookbook

Over 40 hands-on recipes to quickly and efficiently create
business grade Titanium Enterprise apps

Benjamin Bahrenburg

[PACKT]
PUBLISHING

www.allitebooks.com

Appcelerator Titanium Business Application Development Cookbook

Over 40 hands-on recipes to quickly and efficiently
create business grade Titanium Enterprise apps

Benjamin Bahrenburg



BIRMINGHAM - MUMBAI

Appcelerator Titanium Business Application Development Cookbook

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: June 2013

Production Reference: 1180613

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

ISBN 978-1-84969-534-3

www.packtpub.com

Cover Image by Will Kewley (william.kewley@kbbs.ie)

Credits

Author

Benjamin Bahrenburg

Project Coordinator

Hardik Patel

Reviewers

David Bankier

Imraan Jhetam

Proofreader

Katherine Tarr

Acquisition Editor

Joanne Fitzpatrick

Indexer

Hemangini Bari

Lead Technical Editor

Ritika Dewani

Graphics

Abhinash Sahu

Technical Editors

Prasad Dalvi

Pushpak Poddar

Production Coordinators

Melwyn Dsa

Zahid Shaikh

Cover Work

Melwyn Dsa

Zahid Shaikh

About the Author

Benjamin Bahrenburg is a developer, blogger, speaker, and consultant. Ben specializes in building enterprise solutions using mobile technologies, geolocation services, and domain-specific languages. Over the last decade, he has provided mobility solutions for numerous Fortune 100 organizations.

Ben is passionate about cross-platform development, particularly the use of Titanium mobile for mobile app development. He was an early adopter of the Titanium mobile SDK and has built apps since the earliest previews of Appcelerator's mobile platform. Ben is an active member of the mobile development community and holds a Titanium certification in addition to being part of the Appcelerator Titan evangelist group. A strong advocate and contributor to the Titanium module ecosystem, Ben has published numerous open source modules used in thousands of published apps.

An active blogger at bencoding.com, he frequently posts tutorials on mobile development and enterprise coding topics.

I would like to thank my family, friends, and co-workers for their enthusiasm and encouragement. Without their support, this book would not have been possible. I would also like to thank the Titanium community for providing a thriving environment to learn, create, and develop within.

About the Reviewers

David Bankier leads YY Digital, a company that builds integrated mobile, tablet, and server-side applications for enterprise. Previously, David worked in the telecommunications industry with a strong focus on VoIP technologies. He has used Titanium for mobile development since 2010 and uses a mix of Node, Scala, and Java on the backend. David is also the creator of TiShadow, the popular open source project for rapid Titanium development. As a Titan, David's blog <http://www.yydigital.com/blog> dives deep into Titanium.

Imraan Jhetam is a medical doctor and entrepreneur living in England with equal loves for both medical law and technology. He earned his Medical Degree from the University of Natal in 1983, his MBA from the University of Witwatersrand and a Masters of Law Degree from Cardiff University.

Imraan has been fascinated by computers since his youth and taught himself the basics of programming during his university years. He has been writing programs since the mid 1970's in various languages and for different platforms and has fond memories of his first Apple IIe with its then impressive 64 KB RAM.

When he is not busy seeing patients or writing medico-legal reports, he spends his time developing applications and developed Snappa, a social-sharing game that is the better way to draw something for friends. This was written using the incredible Titanium Studio tools and Appcelerator Cloud Services and is now in the Apple and Android App Stores. He was also third prize winner at the first Codestrong Hackathon with two e-payment apps PayBill and PayPad, that also included social media, geolocation, photos, and bar-codes, and which were developed in a restricted and short time using Appcelerator Titanium Studio.

You can contact Imraan via www.snappa.mobi or via Twitter @The__i.

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
Chapter 1: Patterns and Platform Tools	7
Introduction	7
Defining an app namespace	10
CommonJS in practice	10
Using platform indicators	19
Global logging using Ti.App.Listener	27
Chapter 2: Cross-platform UI	33
Introduction	33
Cross-platform HUD progress indicator	34
In-app notifications	38
Screen Break Menu	44
Metro Style Tab Control	49
Slideout Menu	53
Chapter 3: Using Databases and Managing Files	63
Introduction	63
Accessing your database's Ti.Filesystem	64
DbTableChecker SQLite table existence checking	69
Recursively handling files using Dossier	74
Tuning your SQLite database for maximum performance	77
Data access using DbLazyProvider	83
NoSQL using MongloDb	87
Chapter 4: Interacting with Web Services	93
Introduction	93
Consuming RSS feeds	94
Creating a business location map using Yahoo Local	101
Using Google Analytics in your app	106

Making SOAP service calls using SUDS.js	111
Using the LinkedIn Contacts API	117
Chapter 5: Charting and Documents	127
Introduction	127
Opening PDF documents	128
Using an iPad for document signatures	134
Creating PDF documents from images or screenshots	140
Generating PDF documents using jsPDF	144
Creating a scheduling chart using RGraph	148
Displaying information with Google gauges	154
Chapter 6: Using Location Services	159
Introduction	159
Native geolocation using basicGeo	160
Using the Ti.GeoProviders framework for geolocation	168
Multitenant geolocation	176
Calculating distance between addresses	181
Background geolocation management	188
Chapter 7: Threads, Queues, and Message Passing	197
Introduction	197
Queuing multiple downloads	198
Launching one app from another	206
Cross-platform URL schemes	214
Opening your Android app with BOOT_COMPLETED	224
iOS Multithreading using Web Workers	233
Chapter 8: Basic Security Approaches	239
Introduction	239
Implementing iOS data protection in Titanium	240
AES encryption using JavaScript	246
Basic authentication using Ti.Network.HTTPClient	252
Implementing a cross-platform passcode screen	257
Working with protected ZIP files on iOS	262
Chapter 9: App Security Using Encryption and Other Techniques	269
Introduction	269
Using secure properties	270
Object and string encryption	277
Working with encrypted files	281
Handling protected PDFs on iOS	286
Android lock screen monitor	292

Appendix: Titanium Resources	297
Getting started with Titanium	297
Getting started with Android	298
Getting started with iOS	298
Titanium testing resources	298
Modules and open source	299
Titanium community links	300
Index	303

Preface

The demand for mobile apps by business enterprises has never been greater. While meeting this demand is becoming increasingly business critical, the complexities of development are amplified by the explosion in devices and platforms. Mobile JavaScript frameworks have been available since the start of the smartphone revolution and is the perfect platform to meet these challenges. Recognizing the cross-platform nature of these frameworks, enterprises have begun to rely on them for building cost-effective yet compelling cross-platform solutions.

With an innovative architecture, Appcelerator's Titanium Mobile stands out for its rapid development speed, large number of APIs, and focus on providing a native experience. With a thriving community and a growing number of enterprise partners, Titanium has quickly become one of the largest cross-platform mobile frameworks.

Appcelerator Titanium Business Application Development Cookbook shows how you can create native mobile apps using your existing JavaScript skills. Through a series of examples and supporting code, all aspects needed for Enterprise Titanium development are covered. You will learn to use a single JavaScript codebase to develop cross-platform enterprise mobile apps for iOS and Android.

Beginning with a discussion on design patterns, the practical recipes in this cookbook progress through different topics required for Enterprise mobile development. Each recipe within this cookbook is a self-contained lesson. Feel free to pick and choose which recipes are of interest and use them as a reference on your Titanium projects. Recipes in this book demonstrate how to work with data on the device, create charts and graphs, and interact with various web services. Later recipes discuss application security and native module development, which provide additional resources to accelerate your next Titanium mobile development project.

What this book covers

Chapter 1, Patterns and Platform Tools, covers MVC constructs, CommonJS, lazy loading, and other mobile development patterns to help improve code maintenance and reduce your memory footprint.

Chapter 2, Cross-platform UI, demonstrates how to provide a native experience across iOS and Android from a common codebase while creating a branded experience.

Chapter 3, Using Databases and Managing Files, covers approaches on using NoSQL, SQLite, and the filesystem to manage device data.

Chapter 4, Interacting with Web Services, demonstrates how to interact with a variety of different web technologies including RSS, REST, YQL, and SOAP.

Chapter 5, Charting and Documents, covers approaches on how to visualize data through the use of charts, document displays, magazine, and gallery layouts.

Chapter 6, Using Location Services, discusses how to effectively use geolocation in a globally reliable manner using both on-demand and background techniques.

Chapter 7, Threads, Queues, and Message Passing, covers how to leverage queues, intents, and threading to separate application responsibility and providing message passing functionality.

Chapter 8, Basic Security Approaches, provides recipes on best practices for login management, storing credentials, and use of third party JavaScript encryption libraries.

Chapter 9, App Security Using Encryption and Other Techniques, covers the secure handling of application assets, configurations, and data required in Enterprise app development.

Appendix, Titanium Resources, provides additional links and details on how to get started.

What you need for this book

- ▶ Windows, Mac OS X
- ▶ Android SDK 2.3 or higher
- ▶ iOS SDK 5.1 or higher
- ▶ Titanium SDK 3 or higher

Who this book is for

This book is for readers with some knowledge of JavaScript who are looking to rapidly build mobile enterprise apps. Experienced mobile developers looking to expand their skills and develop cross-platform solutions will find these practical recipes a helpful resource in getting started on their next Titanium project.

Conventions

In this book, you will 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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"This namespace contains common APIs such as `Ti.UI.View` and `Ti.UI.TableView`."

A block of code is set as follows:



```
//Create our application namespace
var my = {
  ui:{
    mod : require('dateWin')
  },
  tools:{},
  controllers:{}
};
```



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

```
//Add a global event to listen for log messages
Ti.App.addEventListener('app:log', function(e) {
  //Provide Log Message to CommonJS Logging Component
  my.logger.add(e);
});

//Open our sample window
my.ui.mainWindow.open();
```

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: " This recipe walks through using the screen break interaction pattern to create additional space for an **Add Notes** field."

 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/support>, 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.

1

Patterns and Platform Tools

In this chapter we will cover:

- ▶ Defining an app namespace
- ▶ CommonJS in practice
- ▶ Using platform indicators
- ▶ Global logging using `Ti.App.Listener`

Introduction

For many, building a Titanium app will be their first experience with a large, complete JavaScript project. Whether you are designing a small expense tracking system or a complete CRM tool, implementing proper design patterns in Titanium will improve the performance and maintainability of your app.

The cross-platform nature and underlying architecture of Titanium influences how many common design patterns can be implemented. In this chapter, we will demonstrate how to apply patterns to increase speed of development while implementing best practices for multiple device support.

Introducing Titanium

Appcelerator Titanium Mobile is a platform for building cross-platform native mobile applications using modern web technologies, such as JavaScript, CSS, and HTML. Titanium Mobile is an open source project developed by Appcelerator Inc and licensed under the OSI-approved Apache Public License (Version 2).

The Titanium Mobile project is one of the most active on Github with a large number of commits each day. The Github repository is the focal point for many in the community including module developers, app developers needing night builds, and individual contributors to the Titanium project.

The Titanium ecosystem is one of the largest in the industry with a community of more than 450,000 worldwide developers running apps on 10 percent of the world's devices. Appcelerator boasts one of the largest mobile marketplaces providing third-party components for Titanium Mobile platform.

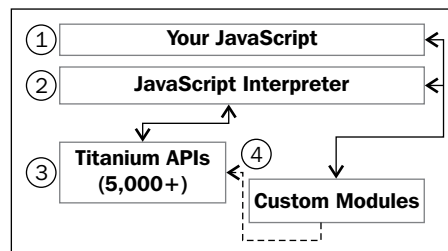
Architecture of Titanium

Titanium is a module-based mobile development platform consisting of JavaScript and native platform code (Java, Objective-C, and C++). The architectural goal of Titanium is to provide a cross-platform JavaScript runtime and API for mobile development; this differs from other frameworks' approaches of building "native-wrapped" web applications.

Titanium uses a JavaScript interpreter to create a bridge between your app's JavaScript code and the underlying native platform. This approach allows Titanium to expose a large number of APIs, and native UI widgets without sacrificing performance. Titanium's UI controls are truly native and not visually emulated through CSS. Thus, when you create a `Ti.UI.Switch`, it is actually using the native `UISwitch` control on iOS.

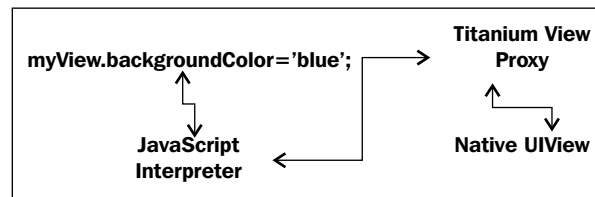
Each Titanium application is organized into layered architecture consisting of the following major components:

- ▶ **Your JavaScript code:** At compile time, this will be encoded and inserted into Java or Objective-C files
- ▶ **Titanium's JavaScript interpreter:** On Android V8 or JavaScriptCore for iOS
- ▶ **The Titanium API:** This is specific for a targeted platform created in Java, Objective-C, or C++
- ▶ **Native Custom Modules:** A large variety of open source and commercial modules are available



At runtime, the Titanium SDK embedded within your app creates a native code JavaScript execution context. This execution content is then used to evaluate your app's JavaScript code. As your JavaScript is executed, it will create proxy objects to access native APIs such as buttons and views. These proxies are special objects that exist both in the JavaScript and native contexts acting as a bridge between the two environments.

For example, if we have a `Ti.UI.View` object and we update the `backgroundColor` to `blue`, the property is changed in JavaScript and then the proxy then updates the correct property in the underlying native layer as shown in the following diagram:



Building a Cross-platform

Titanium provides a high-level cross-platform API, however it is not a write once, run anywhere framework. When building cross-platform apps, it is recommended to adopt a "write once, adapt everywhere" philosophy. With Titanium you can add platform-specific code to handle each platform's different UI requirements, while keeping your business logic 100 percent cross-platform compatible.

Building best of breed cross-platform applications, Titanium provides tools to:

- ▶ Identify the platform and model at runtime
- ▶ Ability to handle platform-specific resources at build time
- ▶ Apply platform and device-specific styling

In addition to platform tooling, the Titanium API is designed to assist with cross-platform development. Each major component such as `Maps`, `Contacts`, and `FileSystem` are separated into its own component namespaces under the top-level namespace called `Ti` or `Titanium`. These component namespaces then have their own child namespaces to segment platform-specific behavior.

An example of this segmentation is the `Ti.UI` namespace, which contains all UI components. This namespace contains common APIs such as `Ti.UI.View` and `Ti.UI.TableView`. Additionally, the `Ti.UI` namespace has platform-specific child namespaces such as `Ti.UI.iPad` containing controls such as `Ti.UI.iPad.Popover`. The same design applies to non-visual APIs such as `Ti.Android`, a namespace which contains Android-specific behavior.

Defining an app namespace

Using namespaces is important in Titanium app development, as it helps organize your code while not polluting the global namespace. The practice of creating variables and methods without being associated with a namespace or other scoping condition is called **polluting the global namespace**. Since these functions and objects are scoped globally, they will not be eligible for collection until the global namespace loses scope during application shutdown. This can often result in memory leaks or other unwanted side effects.

How to do it...

The following example shows how to create a namespace for our app in our `app.js` called `my` with three subnamespaces called `ui`, `tools`, and `controllers`.

```
var my = {ui:{},tools:{},controllers:{}}
```

As we build our recipes, we will continue to add functionality to the preceding namespaces.

CommonJS in practice

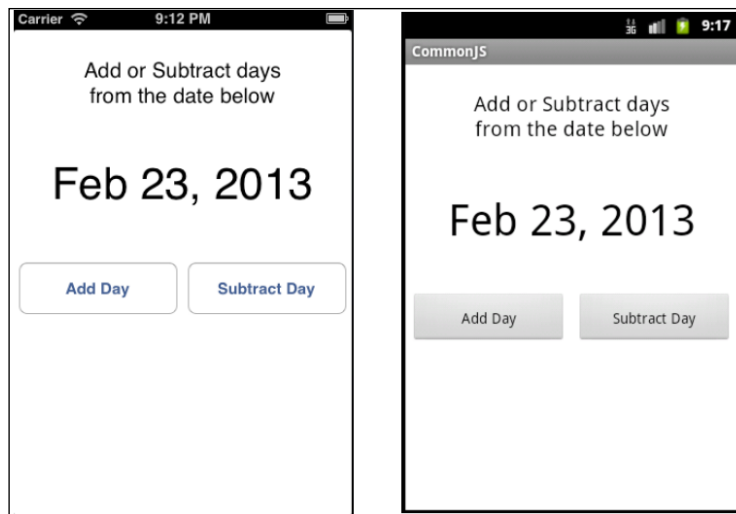
Organizing your application code using CommonJS modules is a best practice in Titanium development. CommonJS is a popular specification for creating reusable JavaScript modules and has been adopted by several major platforms and frameworks such as Node.js and MongoDB.

CommonJS modules help solve JavaScript scope problems, placing each module within its own namespace and execution context. Variables and functions are locally scoped to the module, unless explicitly exported for use by other modules.

In addition to assisting with JavaScript scope concerns, CommonJS provides a pattern to expose a public stable interface to program against. The information-hiding design pattern allows module developers to update the internals of the module without breaking the public contract or interface. The ability to maintain a stable public interface in JavaScript is the key part of writing maintainable code that will be shared across apps and teams.

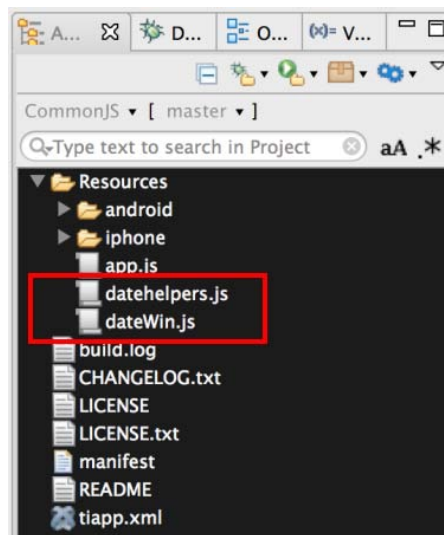
Titanium has implemented CommonJS in a similar fashion to Node.js in that you use the `require` method to return a JavaScript object, with properties, functions, and other data assigned to it, which form the public interface to the module.

The following screenshots illustrate the example app used to demonstrate the CommonJS high-level concepts that will be used throughout the book.



Getting ready

Adding the CommonJS modules used in this recipe is straightforward and consists of copying the `datehelpers.js` and `dateWin.js` files into the root of our Titanium project as shown in the following screenshot:



How to do it...

The following recipe illustrates how to use CommonJS to create both `UI` and `Tools` modules. In the following example, a simple app is created, which allows the user to increase or decrease the date by a day.

Creating the project's `app.js`

In our `app.js` we create our application namespace. These namespace variables will be used to reference our CommonJS modules later in the example.

```
//Create our application namespace
var my = {
  ui:{
    mod : require('dateWin')
  },
  tools:{},
  controllers:{}
};
```



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.

`Ti.UI.Window` is then created using the `my.ui.mod` already added to our app namespace. The `open` method is then called on our `win` object to open our example app's main window.

```
my.ui.win = my.ui.mod.createWindow();

my.ui.win.open();
```

Building the `datehelpers` module

In the `Resources` folder of our project, we have a CommonJS module `datehelpers.js`. This module has the following code:

1. The `helpers` method is created within the `datehelpers` module. This function is private by default until it is exposed using the `exports` object.

```
var helpers = function(){
  var createAt = new Date();
```

- The `createdOn` method is added to the `helpers` function. This function returns the `createAt` variable. This function is used to provide a timestamp value to demonstrate how the module can be initialized several times. Each time a new session is created for the module, the `createAt` variable will display the newly initialized timestamp.

```
this.createdOn = function() {
  return createAt;
};
```

- The `addDays` method is added to the `helpers` function. This method increases the provided date value by the number of days provided in the `n` argument.

```
this.addDays = function(value,n) {
  var tempValue = new Date(value.getTime());
  tempValue.setDate(tempValue.getDate()+n);
  return tempValue;
}
};
```

The `module.exports` is the object returned as the result of a `require` call. By assigning the `helpers` function to `module.exports`, we are able to make the private `helpers` function publically accessible.

```
module.exports = helpers;
```

The dateWin module

Also included in the `Resources` folder of our project is a CommonJS module `dateWin.js`. The following code section discusses the contents of this module.

- Use the `require` keyword to import the `datehelpers` CommonJS module. This is imported in the `mod` module level variable for later usage.

```
var mod = require('datehelpers');
```

- The `createWindow` function returns `Ti.UI.Window` allowing the user to work with the recipe.

```
exports.fetchWindow=function() {
```

- Next a new instance of the `dateHelper` module is created.

```
var dateHelper = new mod();
```

- The next step in building the `createWindow` function is to set the `currentDateTime` module property to a default of the current date/time.

```
var currentDateTime = new Date();
```


5. The `Ti.UI.Window` object, which will be returned by the `createWindow` function, is then created. This will be used to attach all of our UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff'
});
```

6. The `dateDisplayLabel` is used to show the result of the `datehelper` module as the user increases or decreases the date value.

```
var dateDisplayLabel = Ti.UI.createLabel({
  text:String.formatDate
  (exports.currentDateTime,"medium"),
  top:120, height:50, width:Ti.UI.FILL,
  textAlign:'center', color:'#000', font:{fontSize:42}
});
win.add(dateDisplayLabel);
```

7. The `addButton` is used later in this recipe to call the `datehelper` module and add days to the current module value.

```
var addButton = Ti.UI.createButton({
  title:"Add Day", top:220, left:5, width:150, height:50
});
win.add(addButton);
```

8. The `subtractButton` is used later in this recipe to call the `datehelper` module and reduce the date of the current module value.

```
var subtractButton = Ti.UI.createButton({
  title:"Subtract Day", top:220, right:5,
  width:150, height:50
});
win.add(subtractButton);
```

9. The following code in the `addButton` and `subtractButton` click handlers shows how the `AddDays` method is called to increment the `currentDateTime` property of the module.

```
addButton.addEventListener('click',function(e){
```

10. The following line demonstrates how to increase the `currentDateTime` value by a single day:

```
exports.currentDateTime =
dateHelper.addDays(currentDateTime,1);
```

11. Update the `dateDisplayLabel` with the new module value.

```
dateDisplayLabel.text = String.formatDate(
    exports.currentDateTime, "medium");
});
subtractButton.addEventListener('click', function(e) {
```

12. The following code snippet demonstrates how to reduce the `currentDateTime` by a day:

```
exports.currentDateTime =
    _dateHelper.addDays(currentDateTime, -1);
```

13. Update the `dateDisplayLabel` with the new module value.

```
dateDisplayLabel.text = String.formatDate(
    currentDateTime, "medium");
});
return win;
};
```

How it works...

Creating a module is easy. You simply add a JavaScript file to your project and write your application code. By default, any variables, objects, or functions are private unless you have added them to the `module` or `exports` objects. The `module` and `exports` objects are special JavaScript objects created by Titanium and returned by the `require` method.

Require

To use a CommonJS module you must use the globally available `require` function. This function has a single parameter through which you provide the path to your JavaScript module. The following line demonstrates how to load a CommonJS module called `datehelpers.js` located in the root of your Titanium project.

```
var myModule = require('datehelpers');
```



When providing the `require` function with an absolute path, Titanium will start from the `Resources` folder of your project.

Titanium has optimized the module loading process so that when a module is first loaded using the `require` method, it is then cached to avoid the need to re-evaluate the module's JavaScript. This approach significantly improves the load performance for modules which share common dependencies. It is helpful to keep in mind the JavaScript is not re-evaluated if you have the need to manage/alter the module on load.

Properties

Adding a property to your CommonJS module is easy. You simply attach a variable to the `exports` object.

The following snippet demonstrates how to create a basic CommonJS property with a default value.

```
exports.myProperty = "I am a property";
```

More complex object properties can also be created, for example, as follows:

```
exports.myObjectProperty = {  
  foo:1,  
  bar:'hello'  
};
```

You can also use a private variable to set the initial property's value. This often makes for more readable and streamlined code.

Create a local variable reflecting your business need.

```
var _myObjectProperty = {  
  foo:1,  
  bar:'hello'  
};
```

You can then assign the local variable to a property attached to the `exports` object, for example, as follows:

```
exports.myObjectProperty = _myObjectProperty
```



Remember these are just properties on the `exports` JavaScript object and can be changed by any caller.

Functions

Creating public functions is easy in CommonJS, you simply add a function to the `exports` object. Create an `AddDays` method as part of the `exports` object. This function accepts a date in the `value` parameter and an integer as the `n` value.

```
exports.AddDays = function(value,n) {
```

Create a new variable to avoid the provided value from mutating.

```
  var workingValue = new Date(value.getTime());
```

Increase the date by using the `n` value provided. This could be either a positive or negative value. Negative values will decrease the date value provided.

```
workingValue.setDate(workingValue.getDate()+n);
```

Return the new adjusted value.

```
return workingValue;
};
```

You can also assign an `exports` method to a privately scoped function. This can be helpful in managing large or complex functions.

Create a locally scoped function named `addDays`.

```
function addDays(value,n) {
  var workingValue = new Date(value.getTime());
  workingValue.setDate(workingValue.getDate()+n);
  return workingValue;
};
```

The `addDays` function is then assigned to `exports.AddDays` exposing it to callers outside of the module.

```
exports.AddDays = addDays;
```



Remember these are just methods on the `exports` JavaScript object and can be changed by any caller.

Instance object using `module.exports`

Titanium provides the ability to create a module instance object using `module.exports`. This allows you to create a new instance of the function or object attached to `module.exports`. This is helpful in describing one particular object and the instance methods represent actions that this particular object can take.

This pattern encourages developers to think more modularly and to follow the single responsibility principle as only one object or function can be assigned to the `module.exports`.

The following code snippets demonstrate how to create and call a module using this pattern:

1. Using Titanium Studio, create the `employee` (`employee.js`) module file.
2. Next create the `employee` function.

```
var employee = function(name,employeeId,title) {
  this.name = name;
  this.employeeId = employeeId;
  this.title = title;
};
```

```
    this.isVIP = function(level){
        return (title.toUpperCase() === 'CEO');
    }
};
```

3. Then assign the `employee` function to `module.exports`. This will make the `employee` function publicly available to call using `require`.

```
module.exports = employee;
```

4. Using `require`, a reference to the module is created and assigned to the `employee` variable.

```
var employee = require('employee');
```

5. Next the `bob` and `chris` objects are created using new instances of the `employee` object created earlier.

```
var bob = new employee('Bob Smith',1234,'manager');
var chris = new employee('Chris Jones',001,'CEO');
```

6. Finally, the properties and functions on the `bob` and `chris` objects are called to demonstrate each object's instance information.

```
Ti.API.info('Is ' + bob.name + ' a VIP? ' + bob.isVIP());
Ti.API.info('Is ' + chris.name + ' a VIP? ' + chris.isVIP());
```

CommonJS global scope anti-pattern

The CommonJS implementation across Android and iOS is largely the same with one major scoping exception. If you are using a version of the Titanium framework below version 3.1, Android scopes all variable access to the module itself, while iOS allows the module to access objects outside of the module already loaded in the execution context. This should be considered an anti-pattern as it breaks many of the encapsulation principles the CommonJS specification was designed to prevent.

In Titanium Version 3.1, the decision has been made to deprecate global scope access in both iOS and Android in favor of the new `Alloy.Globals` object. You can read more about the `Alloy.Globals` object at <http://docs.appcelerator.com/titanium/latest/#!/api/Alloy>.

The following recipe demonstrates this common anti-pattern and highlights the CommonJS implementation differences in this area between iOS and Android.

```
//Create our application namespace
var my = {
    tools: require('scope_test'),
    session:{
        foo: "Session value in context"
    }
};
```

The `testScope` method is called on the `tools` module. This demonstrates how CommonJS module scope anti-pattern works.

```
my.tools.testScope();
```

The `tools` module containing the `testScope` method is part of this recipe's code and can be found in the `scope.js` file root of our project. This module contains the following code:

```
exports.testScope = function() {  
  Ti.API.log  
    ("Test Module Scope - Foo = " + my.session.foo);  
  return my.session.foo;  
};
```

The scope-access anti-pattern is shown when calling the `my.tools.testScope()` method. In iOS, `my.tools.testScope()` returns "Session Value in context", because it has access to `my.session.foo` from the current execution context. In Android, prior to Titanium SDK Version 3.1, undefined object used to be returned as the module did not have access to the `my.session.foo` object. In the Titanium SDK Version 3.1 and greater, Android now returns "Session Value in context" as it has access to the `my.session.foo` object.

Access to global scope has been deprecated on both platforms starting with Titanium SDK Version 3.1 and will be removed in a future version of the SDK. If you have previously implemented this anti-pattern, corrective action is recommended as the deprecation of this feature will cause breaking changes within your app.

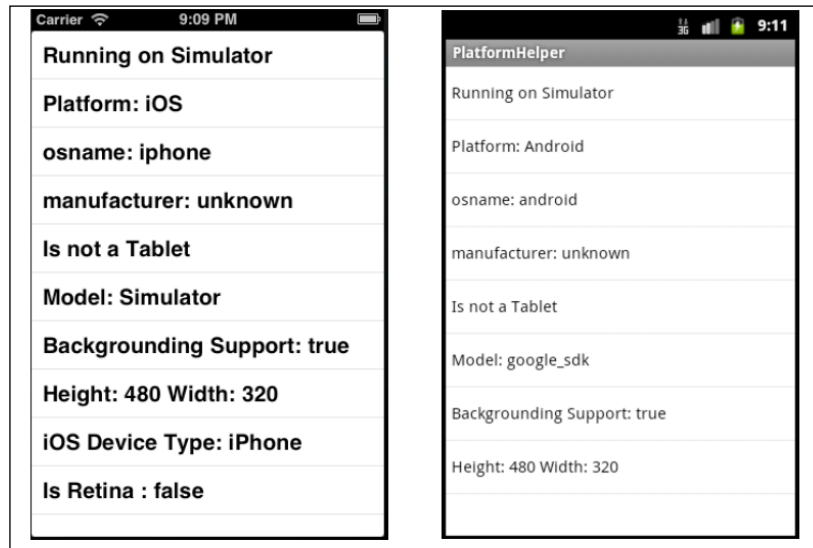
See also

- ▶ To learn more about the CommonJS specification, please visit the wiki on CommonJS.org at <http://wiki.commonjs.org/wiki/CommonJS>
- ▶ For more details and examples on Titanium's implementation of the CommonJS specification, please review their guides at http://docs.appcelerator.com/titanium/latest/#!/guide/CommonJS_Modules_in_Titanium

Using platform indicators

Handling different devices, platforms, and models is often the biggest challenge with cross-platform development. Titanium provides the `Ti.Platform` namespace to help you make decisions in your code on how to handle runtime execution.

In the following recipes, we will walk through how to create a `PlatformHelpers` CommonJS module containing convenience methods to solve many of your platform-related queries. The following screenshots demonstrate this recipe while running on both the iPhone and Android platforms.

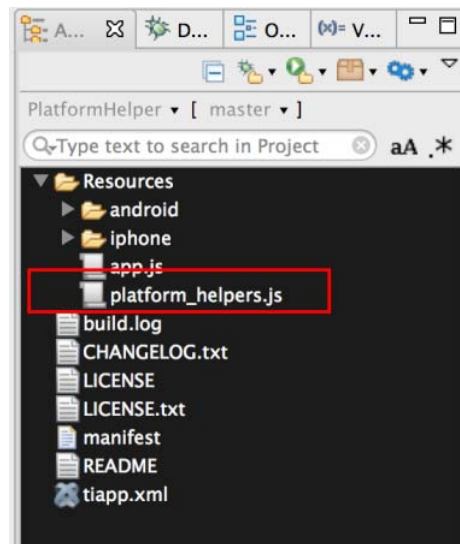


Getting ready

The `Ti.Platform` namespace is helpful for providing platform and device-specific details, but often you need a high level of detail such as when you have a tablet running your app and if so if it is an iPad Mini.

Adding recipe components to your project

Adding the `PlatformHelpers` module to your project is easy. Simply copy the `platform_helpers.js` file into your project as shown in the following screenshot:



How to do it...

The following `app.js` demonstrates the `PlatformHelpers` module described earlier in this chapter. This sample app presents a window with your device details. Give it a try on all of your Android and iOS devices.

1. To create our same app, first we declare our app namespace and import our `PlatformHelper` module.

```
//Create our application namespace
var my = {
  platformHelpers : require('platform_helpers')
};
```

```
(function(){
```

2. Next we create the window in which we will present the device information.

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff'
});
```


3. Next we create an empty array to store our device details.

```
var deviceData = [];
```

An entry is added to `deviceData` if your device is running on a simulator.

The `isSimulator` property is used to show the simulator message, only if the recipe is currently running on a platform simulator or emulator.

```
if(my.platformHelpers.isSimulator){
  deviceData.push({
    title:'Running on Simulator', child:false
  });
}
```

4. Next the platform, operating system's name, and manufacturer are added to `deviceData`.

```
deviceData.push({
  title:'Platform: ' +
    (my.platformHelpers.isAndroid ? 'Android' : 'iOS'),
  child:false
});
```

```
deviceData.push({
  title:'osname: ' + my.platformHelpers.osname,
  child:false
});
```

```
deviceData.push({
  title:'manufacturer: ' +
    my.platformHelpers.manufacturer, child:false
});
```

5. The following statement adds a flag to `deviceData` indicating if the current device is a tablet form factor:

```
deviceData.push({
  title:(my.platformHelpers.isTablet ?
    'Is a Tablet' : 'Is not a Tablet'), child:false
});
```

6. Next we add the device model and specify if it supports background tasks in the `deviceData` array.

```
deviceData.push({
  title:'Model: ' + my.platformHelpers.deviceModel,
  child:false
});
```

```
deviceData.push({
  title:'Backgrounding Support: ' +
  my.platformHelpers.supportsBackground,
  child:false
});
```

7. Screen dimensions are the most commonly used properties. The following snippet adds the height and width of the screen to the `deviceData` array:

```
deviceData.push({
  title:'Height: ' + my.platformHelpers.deviceHeight +
  ' Width: ' + my.platformHelpers.deviceWidth,
  child:false
});
```

8. If your app is currently running on an iOS device, the following snippet adds the device type and specifies if it is retina enabled, to the `deviceData` array.

```
if(my.platformHelpers.isIOS){
  deviceData.push({
    title:'iOS Device Type: ' +
    (my.platformHelpers.iPad ?
    (my.platformHelpers.iPadMiniNonRetina ?
    'iPad Mini' : 'Is an iPad') : 'iPhone'),
    child:false
  });

  deviceData.push({
    title:'Is Retina : ' + my.platformHelpers.isRetina,
    child:false
  });
}
```

9. The `PlatformHelper` module provides a great deal of data. To best display this, we will be using a `Ti.UI.TableView` with the `deviceData` array that we have built in an earlier code snippet.

```
var tableView = Ti.UI.createTableView({top:0,
data:deviceData});

win.add(tableView);
win.open();
})();
```

How it works...

The device platform lookups will frequently be accessed across your app. To avoid performance issues by repeatedly crossing the JavaScript Bridge, we import the values we will use and assign them to properties in our CommonJS `PlatformHelpers` module.

```
exports.osname = Ti.Platform.osname;
exports.manufacturer = Ti.Platform.manufacturer;
exports.deviceModel = Ti.Platform.model;
exports.deviceHeight = Ti.Platform.displayCaps.platformHeight;
exports.deviceWidth = Ti.Platform.displayCaps.platformWidth;
exports.densitylevel = Ti.Platform.displayCaps.density;
exports.deviceDPI = Ti.Platform.displayCaps.dpi;
```

It is often helpful to have a Boolean indicator for the platform with which you are working. The following snippet shows how to create `isAndroid` and `isIOS` properties to accomplish this:

```
exports.isAndroid = exports.osname === 'android';
exports.isIOS = (exports.osname === 'ipad' ||
exports.osname === 'iphone');
```

Simulator check

Depending on your platform, several features may not work in that platform's simulator or emulator. By using the `isSimulator` property, detect which environment you are in and branch your code accordingly.

```
exports.isSimulator = (function(){
  return (Ti.Platform.model.toUpperCase() === 'GOOGLE_SDK' ||
  Ti.Platform.model.toUpperCase() === 'SIMULATOR' ||
  Ti.Platform.model.toUpperCase() === 'X86_64')
})();
```

Background capabilities

Mobile apps are often required to perform tasks in the background. This recipe demonstrates how to perform a version-based capability check to determine if the device your application is running on supports backgrounding/multitasking.

```
exports.supportsBackground = (function(){
```

Now perform the following steps:

1. First check if the user is running on Android. If so, return `true` as most Android ROMs support background processing by default.

```
  if(exports.osname === 'android'){
    return true;
  }
```

- Next confirm the recipe is running on an iOS device.

```
if(exports.osname === 'iphone'){
```

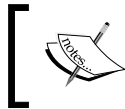
- The version is checked to ensure that the device is running iOS 4 or greater. This confirms the operating system supports background processing.

```
var osVersion = Ti.Platform.version.split(".");
//If no iOS 4, then false
if(parseInt(osVersion[0],10) < 4){
  return false;
}
```

- Finally the recipe confirms the device version is greater than the iPhone 3GS. This confirms the hardware supports background processing.

```
var model = exports.deviceModel.toLowerCase()
.replace("iphone", "").trim();
var phoneVersion = Ti.Platform.version.split(".");
if(parseInt(phoneVersion[0],10) < 3){
  return false;
}
}
//Assume modern device return true
return true;

}) ();
```



Depending on the platform, this feature may be turned off by the user. A secondary capacity check is also recommended.

Detecting tablets

Universal app development is a common requirement on both Android and iOS. The following helper provides a Boolean indicator if your app is running on a tablet form factor:

```
//Determine if running on a tablet
exports.isTablet = (function() {
```

Check if the device is either an iPad or an Android device with at least one dimension of 700 pixels or greater.

```
var tabletCheck = exports.osname === 'ipad' ||
(exports.osname === 'android' &&
(!Math.min(
exports.deviceHeight,
exports.deviceWidth
```

```
    ) < 700));  
    return tabletCheck;  
  }) ();
```



For Android, this checks if there is a height or width greater than 700 pixels. You may wish to change this based on your targeted devices. For example, if you are targeting some of the larger screen Android phones, you would need to update the default 700 pixels/points to reflect them having a large screen yet still being considered a phone form factor.

A 4-inch iPhone

With the introduction of the iPhone 5, we need to be aware of two different iPhone screen sizes. The following snippet shows how to create a property indicating if your app is running on an iPhone with this new screen size.

```
exports.newiPhoneSize = (function() {
```

First verify the recipe is running on an iPhone device.

```
  if(exports.osname !== 'iphone') {  
    return false;  
  }
```

Next check the size to see if there is any dimension greater than 480 points.

```
    return (Math.max(  
      exports.deviceHeight,  
      exports.deviceWidth  
    ) > 480);  
  });
```



Please note, the preceding `newiPhoneSize` method will only return `true`, if the app has been completed to support a 4-inch display. Otherwise, it will return `false` as your app is running in the letterbox mode.

iPad

Titanium allows you to create universal apps on iOS. Using the following property, you can branch your code to show different functionalities to your iPad users:

```
exports.iPad = exports.osname === 'ipad';
```

iPad Mini

The iPad Mini was designed with the same resolution as the first and second generation iPads. Although designed to run iPad apps without modification, the smaller screen size often requires UI adjustments for smaller touch targets. The following code demonstrates how to determine if your app is running on an iPad Mini:

```
exports.iPadMiniNonRetina= (function() {
```

Now perform the following steps:

1. First check if the recipe is running on a nonretina iPad.

```
    if((exports.osname !== 'ipad') ||
       (exports.osname === 'ipad' &&
        exports.densitylevel==='high')){
        return false;
    }
}
```

2. Next verify if the nonretina iPad is not an iPad 1 or iPad 2. If not either of these modules, assume the recipe is running on an iPad Mini.

```
    var modelToCompare = exports.deviceModel.toLowerCase();
    return !(
        (modelToCompare==="ipad1,1") ||
        (modelToCompare==="ipad2,1") ||
        (modelToCompare==="ipad2,2") ||
        (modelToCompare==="ipad2,3") ||
        (modelToCompare==="ipad2,4")
    );
}) ();
```

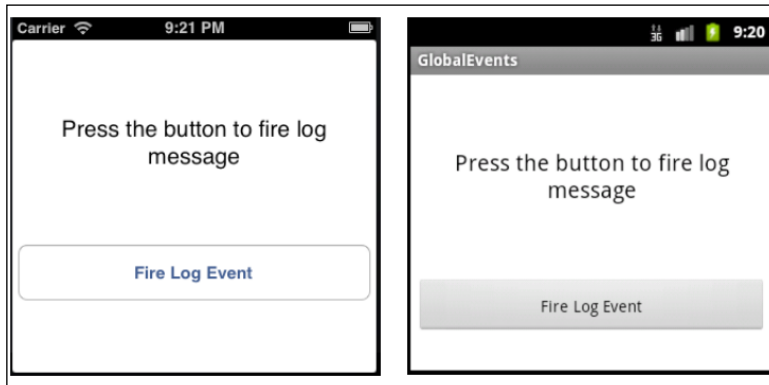


Apple currently does not provide a platform indicator for the iPad Mini. This check uses model numbers and might not be future proof.

Global logging using Ti.App.Listener

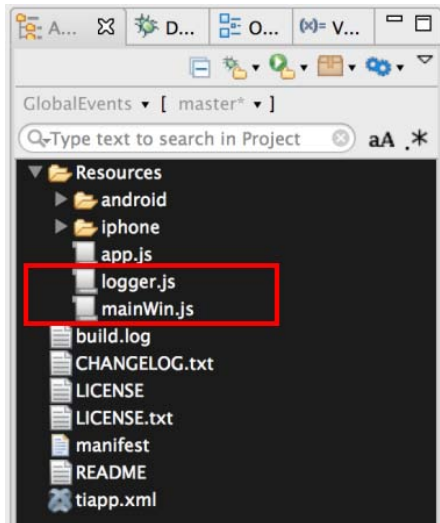
There is built-in ability of Titanium to fire and listen to application wide events. This powerful feature can be a perfect fit for creating loosely-coupled components. This pattern can provide a decoupled logging approach, which is helpful for specific situations such as analytics, logging, and process monitoring modules.

This recipe details how `Ti.App.Listener` can be used to create a decoupled re-useable application component. The following screenshot demonstrates this recipe running on both the iPhone and Android platforms:



Getting ready

Adding the CommonJS modules used in this recipe is straightforward and consists of copying the `logger.js` and `mainWin.js` files into the root of our Titanium project as shown in the following screenshot:



How to do it...

Application-level events allow you to implement a publish/subscribe pattern globally in your app, even across execution contexts. By simply adding a listener such as the following, you can receive messages fired from anywhere in your app:

```
Ti.App.addEventListener('app:myEvent', myFunctionToHandleThisEvent);
```

Firing a custom event in Titanium is easy. You simply use the `Ti.App.fireEvent` and provide the event name and an optional parameter payload. This example shows how to call the `app:myEvent` listener we defined earlier.

```
Ti.App.fireEvent('app:myEvent', "My parameter objects");
```



It is recommended that you name your events using a descriptive convention. For example, `app:myEvent` describes that this is an application event and is defined in `my app.js` file.

Designing global logging using events

The following recipes show how to use application-level custom events to implement logging across the different contexts of our app.

Defining our app.js

In the `app.js`, we define our application namespace and import the logger and UI CommonJS modules.

```
//Create our application namespace
var my = {
  ui: {
    mainWindow : require('mainWin').createWindow()
  },
  logger : require('logger')
};
```

Next in our `app.js`, the `setup` method is called on the `logger` module. This ensures our database logging tables are created.

```
//Run setup to create our logging db
my.logger.setup();
```

The application-level listener is then defined in our `app.js`. This listener will wait for the `app:log` event to be fired and then call the logger's `add` method with the payload provided.

```
//Add a global event to listen for log messages
Ti.App.addEventListener('app:log', function(e) {
  //Provide Log Message to CommonJS Logging Component
```



```
    my.logger.add(e);
  });

  //Open our sample window
  my.ui.mainWindow.open();
```

The logging module

The following code snippet demonstrates how to create the basic CommonJS module, `logger.js`, used in our global listener recipe. This module has two methods; `setup`, which creates our database objects, and `add`, which the listener calls to log our messages.

Create a constant with the logging database name.

```
var LOG_DB_NAME = "my_log_db";
```

Create the `setup` module-level method. This is used to create the `LOG_HISTORY` table, which will later be used to record all log statements.

```
exports.setup=function(){
  var createSQL = 'CREATE TABLE IF NOT EXISTS LOG_HISTORY ';
  createSQL += '(LOG_NAME TEXT, LOG_MESSAGE TEXT, ';
  createSQL += 'LOG_DATE DATE)';
  //Install the db if needed
  var db = Ti.Database.open(LOG_DB_NAME);
  //Create our logging table if needed
  db.execute(createSQL);
  //Close the db
  db.close();
};
```

Create the `add` module-level method. This method is used to record the information to be added to the log table.

```
exports.add=function(logInfo){
  var insertSQL ="INSERT INTO LOG_HISTORY ";
  insertSQL += " (LOG_NAME,LOG_MESSAGE,LOG_DATE) ";
  insertSQL += " VALUES(?,?,?) ";
  var db = Ti.Database.open(LOG_DB_NAME);
  //Create our logging table if needed
  db.execute(insertSQL,logInfo.name,logInfo.message,
  new Date());
  //Close the db
  db.close();
};
```

Bringing it all together

The following sections show the contents of our `mainWin.js` module. This module returns a window with a single button that when pressed fires our logging event.

Window and module variable setup

The `fetchWindow` function returns a simple window demonstrating how to fire an application-level event. To demonstrate, a new message is sent each time, and we add a module level variable named `_press_Count`, which increments with each click.

Create the module-level variable `_press_Count` to record the number of times `addLogButton` has been pressed.

```
var _press_Count = 0;
```

The `fetchWindow` method returns a `Ti.UI.Window` containing the UI elements for this recipe.

```
exports.fetchWindow=function() {
```

A `Ti.UI.Window` is created for all of our UI elements to be attached.

```
  var win = Ti.UI.createWindow({
    backgroundColor: '#fff'
  });
```

The `addLogButton` `Ti.UI.Button` is created. This button will be used to trigger the `Ti.App.fireEvent` used to demonstrate global listeners in this recipe.

```
  var addLogButton = Ti.UI.createButton({
    title:"Fire Log Event", top:180,
    left:5,right:5, height:50
  });
```

Firing the application-level event

On clicking on the `addLogButton` button, the `_press_Count` variable is increased by one and a logging object is created. The following code demonstrates how the `Ti.App.fireEvent` method is called for the `app:log` event and a JavaScript object containing our logging details is provided. The listener we defined in our `app.js` will then receive this message and log our results.

```
  addLogButton.addEventListener('click', function(e) {
```

The first action taken after the user taps the `addLogButton` is to increase the `_pressCount` by one.

```
    _press_Count ++;
```

Next the `logObject` containing information to be submitted for logging is created.

```
var logObject = {
  name: 'test log',
  message: 'example message ' +
    _press_Count
};
```

Finally, the `Ti.App.fireEvent` is called for the `app:log` event. The `logObject` is also provided as a parameter when calling this application global event.

```
Ti.App.fireEvent('app:log', logObject);
});
win.add(addLogButton);

//Return window
return win;
};
```



App logging is just one of the many examples of how you can use application-level events to decouple your app's components. This pattern will be used several times over the course of this book when decoupled component interaction is required.

There's more...

Since application-level listeners are globally scoped, you need to take caution where and when they are defined. A general rule of thumb is to define application-level listeners in an area of your app that is always available and never reloaded, such as your `app.js`, CommonJS file which is only loaded once, or another bootstrapping method.

If you must define them within a window or other module that is loaded repeatedly, make sure you remove the listener when the object is no longer needed. To remove the event listener, you need to call the `Ti.App.removeEventListener` method with the same arguments used on creation. The following snippet shows how to remove the `app:myEvent` listener created earlier:

```
Ti.App.removeEventListener('app:myEvent',
myFunctionToHandleThisEvent);
```



If you do not remove a listener, all associated JavaScript objects will not be eligible for garbage collection, often resulting in a memory leak.

2

Cross-platform UI

In this chapter, we will cover:

- ▶ Cross-platform HUD progress indicator
- ▶ In-app notifications
- ▶ Screen Break Menu
- ▶ Metro Style Tab Control
- ▶ Slideout Menu

Introduction

Titanium is not a write-once-and-run-anywhere application framework, but more of a write-once-and-modify-for-each-platform one. Nowhere is this more important than in the user interface of your app. Great user experiences involve making the user feel familiar and at the same time accentuate the device the app is running upon. Titanium does this by embracing both common and platform-specific UI widgets.

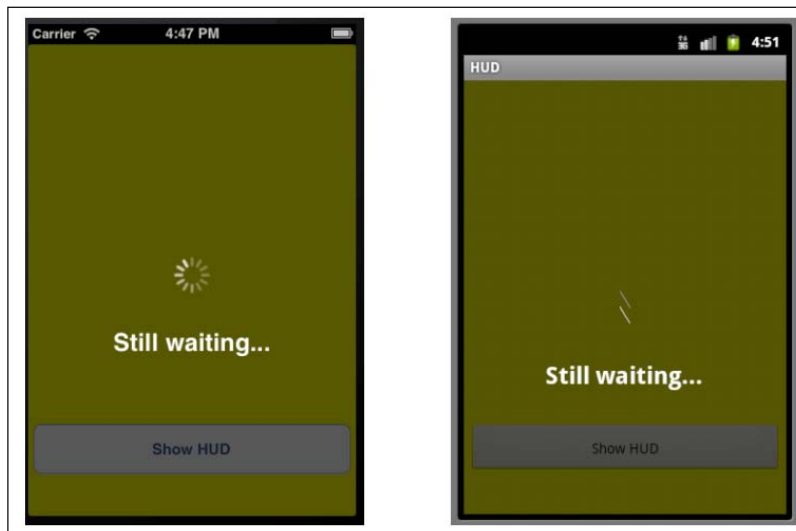
The recipes in this chapter will demonstrate how to create helpful UI widgets to create a unique cross-platform experience for your Enterprise users. Each recipe is designed to be easily branded and used within your existing Titanium Enterprise apps.

Cross-platform HUD progress indicator

The **Heads Up Display (HUD)** interaction pattern, named after the UIKit `UIProgressHUD` component, is an effective way to provide progress information to your users. This is especially true for Enterprise mobile apps, as they typically are deeply integrated with backend systems. So the app does not appear sluggish or unresponsive while making these calls. It is recommended to use the `Waiting` or `Progress` indicators to provide feedback to the users. This recipe demonstrates how to use an HUD module to present a `Waiting` indicator to inform your user about the progress of long-running actions in your app.

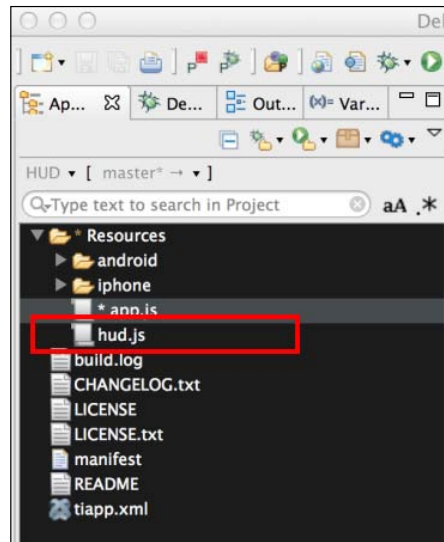
Getting ready

This chainable CommonJS HUD module, `hud.js`, provides a native iOS and Android progress indicator experience. This is a simple recipe demonstrating how to use the HUD module. You can use this example to easily incorporate the module into your Titanium project.



Adding the HUD module to your project

Adding the HUD module to your project is easy. Simply copy the `hud.js` file into the `Resources` folder of your project, as shown in the following screenshot:



Once you've added the `hud.js` file to your project, you need to use `require` to import the module into your code:

```
//Create our application namespace
var my = {
  hud : require('hud'),
  isAndroid : Ti.Platform.osname === 'android'
};
```

Creating a sample window

To demonstrate the HUD module, we create a basic window with a single button. To do this, first we create a `Ti.UI.Window` and attach a button:

```
var win = Ti.UI.createWindow({
  backgroundColor:'yellow'
});

var hudButton = Ti.UI.createButton({
  title:'Show HUD', height:50, right:5, left:5, bottom: 40
});
win.add(hudButton);
```

Adding HUD listeners

Next in our example, we add a series of listeners. The following code block demonstrates how to add a listener, to be fired when the HUD window is closed:

```
my.hud.addEventListener('close', function(e) {
  Ti.API.info('HUD Window Closed');
});
```

1. You can also listen to an `open` event, which will fire when the HUD window is opened:

```
my.hud.addEventListener('open', function(e) {
  Ti.API.info('HUD Window Open');
});
```

2. The `dblclick` event is fired when the user double taps on the HUD window:

```
my.hud.addEventListener(dblclick, function(e) {
  Ti.API.info('HUD Window double clicked');
});
```

3. The `hudTextChanged` event is fired whenever `updateMessage` is called to update the HUD window message:

```
my.hud.addEventListener('hudTextChanged', function(e) {
  Ti.API.info('Text Changed from ' + e.oldValue +
    ' to ' + e.newValue);
});
```

4. If you are using the **Close Timer** functionality, it is helpful to know if your window is being closed by the timer or by some other method. To help determine this, you can subscribe to the `timerClose` event, as shown in the following snippet:

```
my.hud.addEventListener('timerClose', function(e) {
  Ti.API.info('HUD Window Closed by Timer');
});
```

Creating a HUD indicator

We can now use the button created earlier, to demonstrate how to show the HUD window. To the click handler of the button we've added logic to load the HUD module with the message text, `Please Wait . . .`. Then we call the `show` method, which displays the HUD window to the user.

```
hudButton.addEventListener('click', function(e) {
  win.navBarHidden=true;
  my.hud.load('Please Wait...').show();
});
```



On Android, you might wish to hide the navigation bar so that the HUD window can take up the entire screen. If you don't hide the navigation bar, the HUD module will set the window title to the same text as provided for the HUD display message.

Updating the HUD message

By calling the `updateMessage` function, you can update the HUD window text at any time. The following line demonstrates how to update the text from `Please Wait...` to `Still waiting...`

```
my.hud.updateMessage('Still waiting...');
```

Closing the HUD window

There are two ways to close the HUD module window. The first and most common way is to call the `hide` function, as shown here:

```
my.hud.hide();
```

The second way to close the HUD window is to use the `addCloseTimer` function on the HUD module before the `show` function is called, as highlighted in the following snippet. In this example, the HUD window will close after 5,000 milliseconds.

```
my.hud.load('Please Wait...').addCloseTimer(5000).show();
```



The `addCloseTimer` function is helpful in dealing with timeout situations. For example, when making network calls you might wish to abandon the process after five seconds so as not to keep the user waiting too long.

How it works...

The HUD module provides a series of functions to create, update, and maintain the HUD progress window. The following list provides a description of each method along with their associated functionalities:

- ▶ The `load` function: This is called to build the `Ti.UI.Window`, `Ti.UI.ActivityIndicator`, and `Ti.UI.Label`, that will later be used to display your progress indicator. Please note you will need to first call the `load` function, followed by the `show` function, to display the HUD window.
- ▶ The `show` function: This displays the HUD window. Please note you first need to call the `load` method before calling `show`. If this is done out of sequence, the HUD window will not appear.

- ▶ The `updateMessage` function: To update the HUD label, you call this function, providing a string with the updated text to be displayed.
- ▶ The `hide` function: Calling this will close the HUD window and remove all of the HUD objects created.
- ▶ The `addCloseTimer` function: You use this function to set a timer that will close the HUD window after the provided duration (in milliseconds).
- ▶ The `removeCloseTimer` function: You use this function to close the timer that you set using `addCloseTimer`.
- ▶ The `addEventListener` function: The HUD module supports several events as detailed in the next section. You can subscribe to these events using this function.
- ▶ The `removeEventListener` function: Any events you have subscribed to, can be removed by calling this function. To remove an event, you need to provide the same name and callback arguments as you used while calling `addEventListener`.



All functions in the HUD module are chainable, similar to jQuery functions.

In-app notifications

There is often a requirement to alert your users that something has happened. Titanium allows you to use the `alert` function, just as you would do on the web. But, often this modal alert can be limiting and outside of your design requirements.

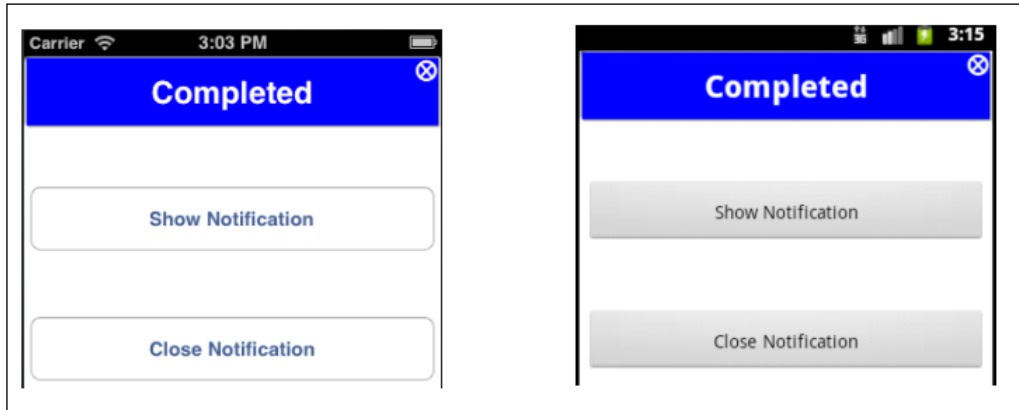
This recipe walks through how to use the `NotifyUI` module to create cross-platform, branded notification windows.

Getting ready

`NotifyUI` is a CommonJS module consisting of a single JavaScript file, `notify.js`, and a few image assets used for styling. You can control the styling of the module through configuration or by updating core image files, making it easy to fit your existing branding requirements.

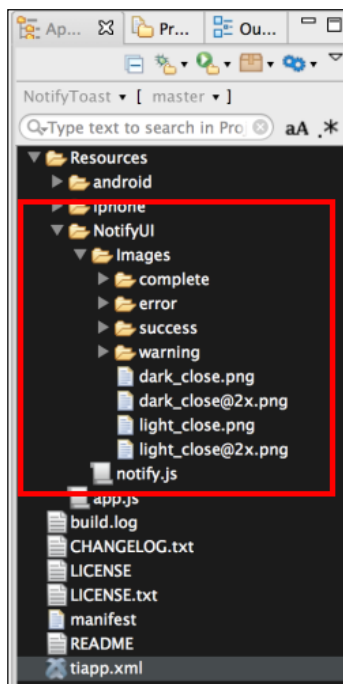
This recipe is a simple demonstration on how to use the `NotifyUI` module. You can use this example to quickly incorporate the module into your Titanium project.

The following screenshots show this recipe in action:



Adding the NotifyUI module to your project

Adding the `NotifyUI` module to your project is easy. Simply copy the `NotifyUI` folder into the `Resources` folder of your Titanium project, as highlighted in the following screenshot. This will install the CommonJS module and all supporting images.





The `NotifyUI` folder must be copied to the `Resources` folder of your Titanium project in order for this recipe to work as designed.

How to do it...

Once you've added the `NotifyUI` folder to your project, you need to use `require` to import the module into your code:

```
//Create our application namespace
var my = {
  notify = require('./NotifyUI/notify'),
  isAndroid : Ti.Platform.osname === 'android'
};
```

Creating a sample window

To demonstrate the `NotifyUI` module, we create a basic window with two buttons:

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff'
});

var showNotifyButton = Ti.UI.createButton({
  title:'Show Notification', top: 100,
  height:50, right:5, left:5
});
win.add(showNotifyButton);

var closeNotifyButton = Ti.UI.createButton({
  title:'Close Notification', top: 200,
  height:50, right:5, left:5
});
win.add(closeNotifyButton);
```

Adding NotifyUI listeners

Next in our example, we add a series of listeners. The following code block demonstrates how to add a listener, to be fired when the `NotifyUI` notification window is closed:

```
my.notify.addEventListener('close', function(e) {
  Ti.API.info(Notify Window Closed);
});
```

1. You can also listen to `open` event, which will fire when the notification window is opened.

```
my.notify.addEventListener('open', function(e) {
  Ti.API.info('Notify Window Open');
});
```
2. The `dblclick` event is fired when the user double taps on the notification window.

```
my.notify.addEventListener('dblclick', function(e) {
  Ti.API.info('Notify Window double clicked');
});
```
3. The `textChanged` event is fired from the `NotifyUI` module whenever `updateMessage` is called to update the notification window message.

```
my.notify.addEventListener('textChanged', function(e) {
  Ti.API.info('Text Changed from ' + e.oldValue +
    ' to ' + e.newValue);
});
```

If you are using the Close Timer functionality, it is helpful to know if your window is being closed by the timer or by some other method. To help determine this, you can subscribe to the `timerClose` event, as shown in the following snippet:

```
my.notify.addEventListener('timerClose', function(e) {
  Ti.API.info('Notify Window Closed by Timer');
});
```

Showing a message window

We can now use the button created earlier, to demonstrate how to show the notification window. To the click handler of the button we've added logic to load the module with the message text, `Hello World`, and the style, `complete`. Then we call the `show` method, which displays the notification window to the user.

```
showNotifyButton.addEventListener('click', function(e) {
  win.navBarHidden = true;
  my.notify.load({
    style:'complete',
    message:'Hello World'
  }).show();
});
```

Updating a message

By calling the `updateMessage` function, you can update the notification text any time. The following demonstrates how to update the text from `Hello World` to `I'm a new message`:

```
my.notify.updateMessage("I'm a new message");
```

Closing a message window

There are three ways to close the notification window. The first and most common way is to call the `hide` function, as shown in the following snippet:

```
my.notify.hide();
```

The second way to close the notification window is to use the `addCloseTimer` function before the `show` function is called, as shown in the following code block. In this example, the window will automatically close after 5,000 milliseconds (5 seconds).

```
my.notify.load({
  style: 'complete',
  message: 'Hello World'
}).addCloseTimer(5000).show();
```


The third and the final way to close the notification window is to have the user double tap on the message. This gesture will trigger the module to internally handle the close actions.

How it works...

The `NotifyUI` module provides a series of functions to create, update, and maintain the notification window. The following list provides a description of each method, along with their associated functionalities:

- ▶ The `load` function: This is called to build the `Ti.UI.Window`, `Ti.UI.Label`, and associated notification style, that will later be used to display your notification. Please note you will need to first call the `load` function, followed by the `show` function, to display the notification window.
- ▶ The `show` function: This displays the notification window. Please note you first need to call the `load` method before calling `show`. If this is done out of sequence, the notification window will not appear. These sequenced operations provide the flexibility to separate the load and show operations. This can be important if you have several notification events in succession.
- ▶ The `updateMessage` function: To update the notification label, you call this function, providing it a string with the updated text to be displayed.
- ▶ The `hide` function: Calling this will close the notification window and remove all the `NotifyUI` objects created.

- ▶ The `addCloseTimer` function: You use this function to set a timer that will close the notification window after the provided duration (in milliseconds).
- ▶ The `removeCloseTimer` function: Using this you can close the timer that you set using `addCloseTimer`.
- ▶ The `addEventListener` function: The `NotifyUI` module supports several events as detailed in the next section. You can subscribe to these events using this function.
- ▶ The `removeEventListener` function: Any events you have subscribed to can be removed by calling this function. To remove an event, you need to provide the same arguments as you used while calling `addEventListener`.

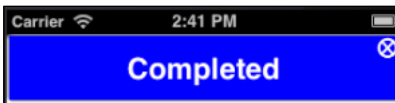
 A majority of the `NotifyUI` functions are chainable, providing a useable pattern similar to the one provided by jQuery.

Built-in message styles

The `NotifyUI` module comes with the following built-in styles:

- ▶ **Complete:** This displays a window with a blue background and the default text, `Completed`

```
my.notify.load({style:'complete'}).show();
```



- ▶ **Error:** This displays a window with a red background and the default text, `Error`

```
my.notify.load({style:'error'}).show();
```



- ▶ **Warning:** This displays a window with an orange background and the default text, `Warning`

```
my.notify.load({style:'warning'}).show();
```



- ▶ **Success:** This displays a window with a green background and the default text, `Success`

```
my.notify.load({style: 'success'}).show();
```



All default message texts and text colors can be updated by providing the `message` and `messageColor` object properties to the `load` function call.

Screen Break Menu

Enterprise apps often contain a number of features. Providing a compelling way to display these features and submenus can be challenging. One unique navigation option is to implement a Screen Break Menu, similar to how iOS folders work. This navigation technique allows you to present additional information only when requested by the user.

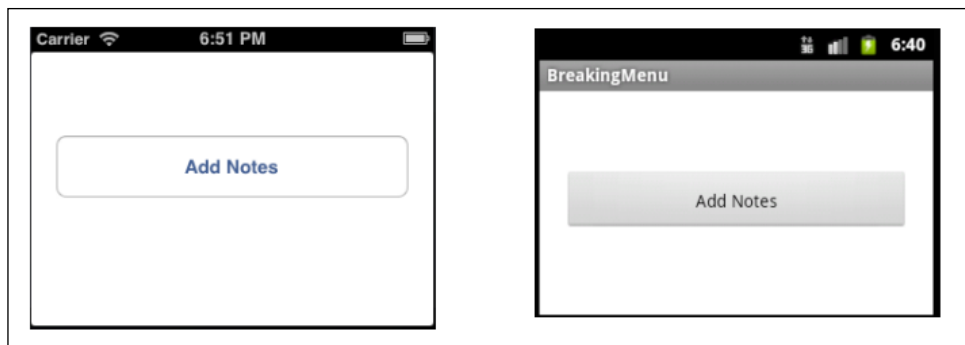
In this recipe, we will demonstrate how you can implement a cross-platform version of the Screen Break animation and link this effect to a menu view.

Getting ready

This recipe walks through using the screen break interaction pattern to create additional space for an **Add Notes** field. You can easily tweak this recipe for more complex menu needs, such as implementing advanced options or an interactive help menu.

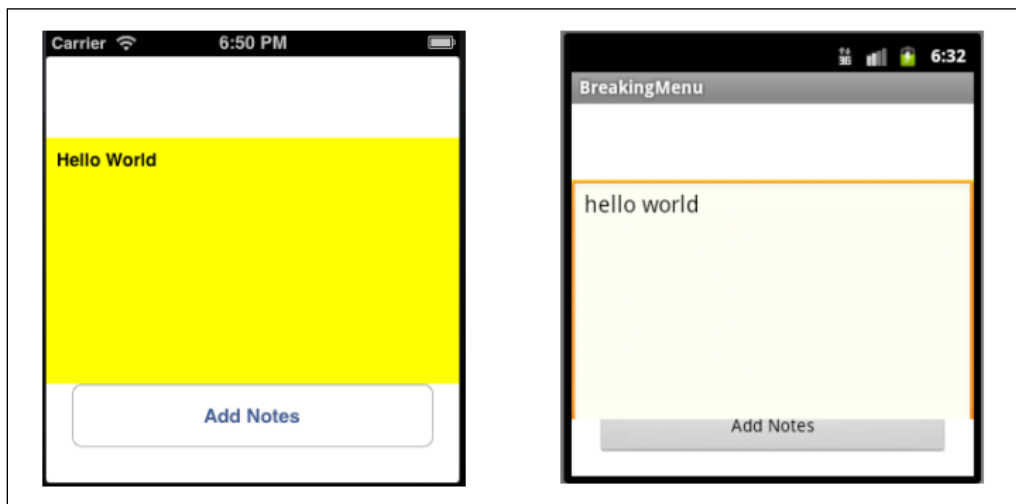
Menu when closed

When the menu is closed, all space is usable for other controls. The following image shows the Screen Break Menu in its closed state.



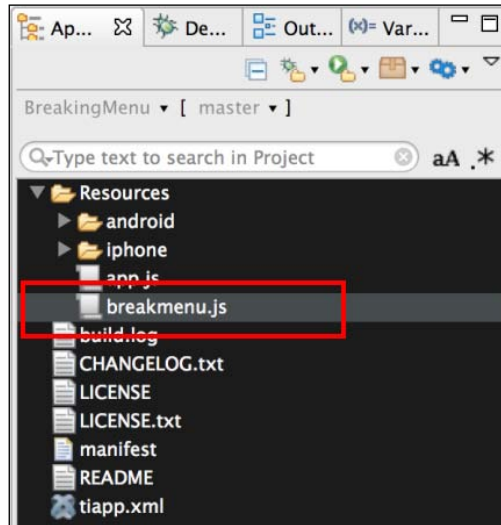
The menu, when opened

When the menu is opened, the screen is split and the bottom section of the `Ti.UI.Window` is animated, exposing a menu area. The following example shows the Screen Break Menu, when opened, and displaying a `Ti.UI.TextArea` for taking notes:



Adding the Screen Break Menu to your project

The Screen Break Menu is a CommonJS module consisting of a single JavaScript file. To install it, simply copy the `breakmenu.js` file into your project, as shown in the following screenshot:



How to do it...

Once you've added the `breakmenu.js` file to your project, you need to use `require` to import the module into your code, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  breakingMenu : require('breakmenu')
};
```

Creating the sample window

To demonstrate the Screen Break Menu, we create a basic window with a single button. When this button is clicked, the Screen Break Menu is activated.

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff'
});

//Create button
var button = Ti.UI.createButton({
  title:'Add Notes', left:20, right:20, height:50, top:150
});
win.add(button);
```

Adding Screen Break Menu listeners

Next in our example, we add a series of listeners. The following code block demonstrates how to add a listener, to be fired when the Screen Break Menu starts, to show the menu.

```
my.breakingMenu.addEventListener('breaking',function(e) {
  Ti.API.info("Screen is breaking");
});
```

1. The broken event is fired when the menu is fully displayed.

```
my.breakingMenu.addEventListener('broken',function(e) {
  Ti.API.info("Screen is broken, menu is visible");
});
```

2. The closing event is fired when the menu starts to close. This will take a second to fully close.

```
my.breakingMenu.addEventListener('closing',function(e) {
  Ti.API.info('Menu starting to close');
});
```

3. Only when the menu is fully hidden, the closed event is fired.

```
my.breakingMenu.addEventListener('closed',function(e) {
  Ti.API.info('Menu is closed');
});
```

Creating a notes menu object

Displaying a `Ti.UI.View` is at the core of the Screen Break Menu. In this example, we create a notes menu object containing `Ti.UI.View` and `Ti.UI.TextArea` controls. These controls are used to present the user with an area for entering notes, when the Screen Break Menu is visible.

```
var notesMenu = {
  value : '', //Our value for the notes screen
  view : function(){
    //Create a view container
    var vwMenu = Ti.UI.createView({
      backgroundColor:'yellow',top:150,
      height:200, width: Ti.UI.FILL
    });
    //Create a textArea for the user to type
    var txtField = Ti.UI.createTextArea({
      width: Ti.UI.FILL,height:Ti.UI.FILL,
      hintText:"Add a note", value : notesMenu.value,
      backgroundColor:'transparent',
      font:{fontSize:14,fontWeight:'bold'}
    });
```

```
vwMenu.add(txtField);
//When text is added, update the menu value
txtField.addEventListener('change',function(e){
    notesMenu.value = txtField.value;
});
//Return the view so we can later bind for display
return vwMenu;
}
};
```

Showing the menu

When the sample's button is pressed, the `breakScreen` function is called to show the Screen Break Menu.

The `breakScreen` function takes the following parameters:

- ▶ The first parameter is the `Ti.UI.Window` that you wish to split in order to reveal the menu.
- ▶ The second parameter is the `Ti.UI.View` that you wish to use as your menu. Please note this can only be a `Ti.UI.View` object type.
- ▶ The final parameter is the `settings` object. This object contains the following properties:
 - `breakStartPoint`: The position measured from the top of the screen indicating where the break should start.
 - `bottomViewHeight`: This property decides the height of the bottom-half section. This value determines the size of the bottom half of the screen that is animated down the screen.
 - `slideFinishPoint`: The position from the top of the screen indicating where the bottom view should slide to.



You will need to adjust these `settings` properties to meet your screen layout requirements.

The following code block implements all the three properties:

```
button.addEventListener('click',function(e){
    //Options object
    var settings = {};
    //Point of the screen where the screen will separate
    settings.breakStartPoint = 150;
    //Size of the bottom half of screen, ie the sliding part
    settings.bottomViewHeight = 218,
```

```
//The bottom point you want the screen to slide to.  
//Measured from top of screen  
settings.slideFinishPoint = 340;  
//Call function to split the screen & show our menu  
my.breakingMenu.breakScreen(win,notesMenu.view(), settings);  
});
```

Metro Style Tab Control

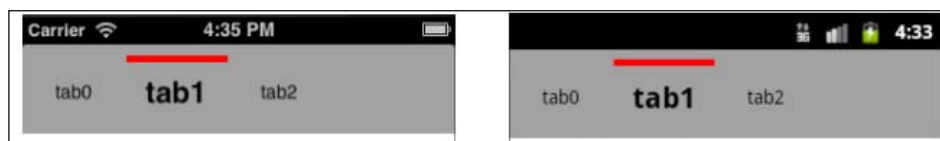
A tabbed interface is an effective way to organize your Enterprise Titanium app. With their rising prevalence in smart phones and well designed, compelling UI/UX consumer applications, the expectation level for UI-rich Enterprise application is increasing among Enterprise users. Your average Enterprise user is now starting to expect a richer experience than provided in a traditional tabbed interface.

This recipe demonstrates how to create a unique cross-platform experience using the **Metro Style Tab Control** for your Enterprise app. This control allows you to use the tabbed interaction pattern in a compelling new way while still meeting your branding or organizational styling requirements.

Getting ready

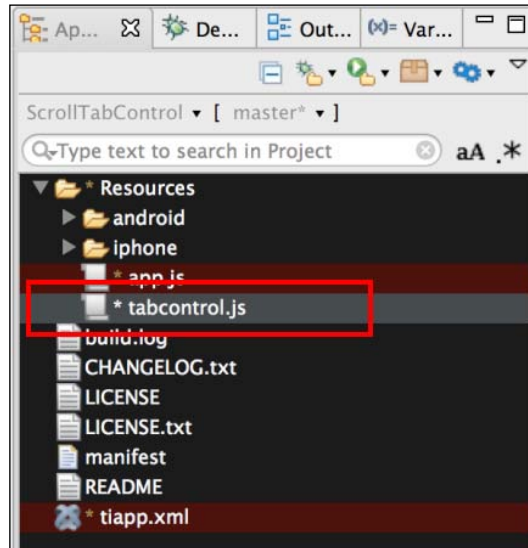
This recipe walks through the process of creating a Metro Style Tab Control to organize your app's navigation. The Tab Control is designed to be easily configured to meet your corporate branding and display requirements.

The next example shows a simple 3-tab navigator at the top of our cross-platform app. The following sections will walk you through the process of creating this sample, and demonstrate how to configure the control to meet your specific needs.



Adding the Tab Control to your project

Metro Tab Control is a CommonJS module, consisting of a single JavaScript file. To install, simply copy the `tabcontrol.js` file into your project, as shown in the following screenshot:



How to do it...

Once you've added the `tabcontrol.js` file to your project, you need to use `require` to import the module into your code, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  tabControl : require('tabcontrol')
};
```

Creating a sample window

To demonstrate the Metro Style Tab Control, create a `Ti.UI.Window` to attach this custom control with three tabs.

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff',title:"Tab Control Example",
  fullscreen:false, navBarHidden:true
});
```

Creating the settings object

When creating the Tab Control, you need a `tabSettings` object and an array of tabs. The `tabSettings` object contains all the properties needed to create the Tab Control. The tabs are then added to the control. When creating the `tabSettings` object, you can use any valid `Ti.UI.ScrollView` properties as the Tab Control provides a wrapper around the Titanium native `Ti.UI.ScrollView`.

The `tabSettings` object also has a Tab Control-specific object called `selectedCue`. The `selectedCue` object contains the `selectedBackgroundColor` and `backgroundColor` properties. The `selectedBackgroundColor` property is the color used to provide a visual cue when the tab is selected. The `backgroundColor` property is the default background color used for the visual cue area.

```
var tabSettings = {
  top:0, //Position the control a top of win
  selectedCue : {
    selectedBackgroundColor:'red',
    backgroundColor:'transparent',
  }
};
```

Defining tabs

You define all of the tabs in the Tab Control at the time of creation. For this example, we will first create an empty array, and then push each Tab Definition into the array.

```
var tabs = []; //Create tab collection
```

A **Tab Definition** has two main parts, the `Tab` and the `Label` sections. In the `Tab` section, you can use any parameter that is valid, when creating a `Ti.UI.View` object. Any additional parameters will be considered custom properties of the `Tab` object and will be available once the Tab Control has been added to the `Ti.UI.Window`.

In the `Label` section, you can use any parameter that is valid, when creating a `Ti.UI.Label` object. The `Label` section also has a tab-specific property called `selectedFont`. The `selectedFont` object has the same properties as the `Ti.UI.Label` font object, but is only applied when the tab has been selected. The next sample is used to demonstrate an increase in the font size and weight.



Make sure you include a `width` property in your tab definition or `TabStrip` will not layout correctly.

```
var tab0 = {
  Tab : {
    left:1, width:75, height:50
  },
};
```

```
Label :{
  text:'tab0',
  selectedFont:{
    fontSize:22, fontWeight:'bold'
  },
  font:{
    fontSize:14
  }
}
};
tabs.push(tab0); //Add tab to collection
```

To complete this recipe, add `tab1` and `tab2` using the same pattern as demonstrated earlier.

Adding Tab Control to the window

The next step in the recipe is to call the `createTabStrip` method, providing our settings object, and array of tabs. This method returns an instance of the custom tab control that is then added to the example's `Ti.UI.Window`, as shown in the following snippet:

```
//Create our tab control
var tabControl = my.tabControl.createTabStrip(tabSettings,tabs);
//Add our tab control to the window
win.add(tabControl);
```

Adding tab listeners

The Metro Tab Control is derived from the `Ti.UI.ScrollView`. It, therefore, inherits its base listeners such as `pinch`, `scroll`, and `click`. Additional events are also available for you to subscribe to.

1. When a user taps on a tab, the `tabClick` event is fired, providing the index of the tab tapped.

```
tabControl.addEventListener('tabClick',function(e) {
  Ti.API.info("Tab at Index: " + e.index + " was clicked");
});
```
2. When a tab is selected, the `indexChanged` event is fired, providing the current and prior index values.

```
tabControl.addEventListener('indexChanged',function(e) {
  Ti.API.info("Tab Index changed from " + e.oldIndex +
    " to " + e.index);
});
```

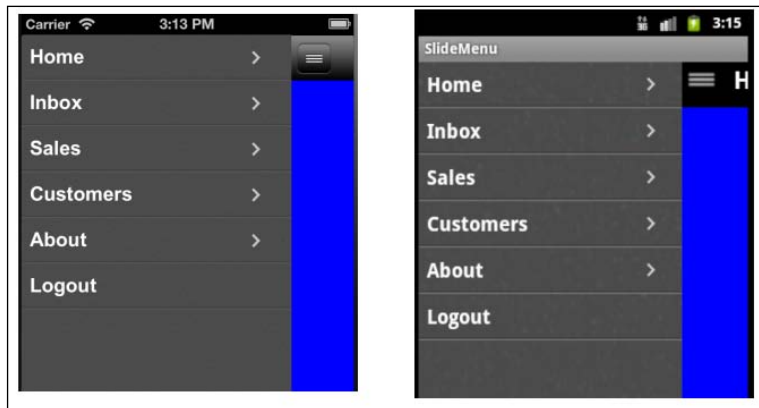
Slideout Menu

Using a Slideout menu for navigation has become a popular way to display navigation options. This navigation pattern was made popular by Facebook and Path apps. The strength of this navigation pattern is that it can effectively present a large number of menu options. Most Enterprise apps can benefit from this kind of navigation as it increases feature discoverability.

Getting ready

In this recipe, we will demonstrate how to create a sample app using the Slideout menu to access four application views, and a logout button. The `Slideout Menu` module provides a chainable CommonJS module to easily manage your menu and visual assets that can be easily branded.

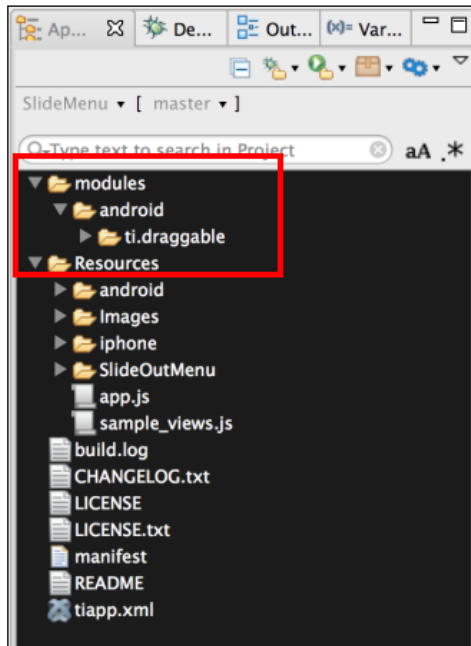
The following screenshots show the Slideout menu when open in our sample cross-platform app. The following steps will walk you through creating this sample, and demonstrate how to configure the control to meet your needs.



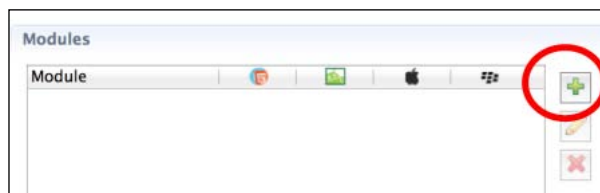
Installing the `Ti.Draggable` module

When running on Android, the Slideout menu uses a popular native module called `Ti.Draggable`. This module enables the user to easily slide back the menu cover.

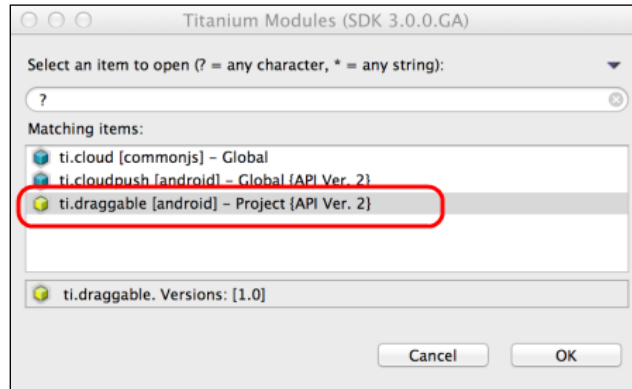
To install this module, you can easily copy the `modules` directory from the recipe's source, or download the `Ti.Draggable` project from <https://github.com/pec1985/TiDraggable>. Once you have the unzipped module, you will want to copy the contents into your project's `modules` folder, as highlighted in the following screenshot:



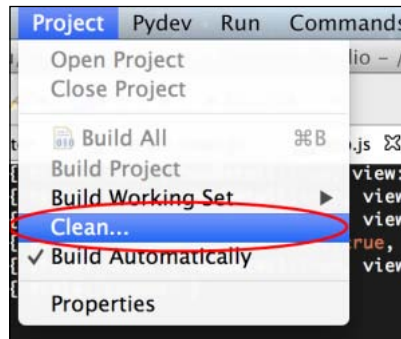
After you have copied the files into the `modules` directory, you will need to add the module to your project. This is easily done in Titanium Studio by clicking on the `tiapp.xml` file. Once the project configuration screen has loaded, click on the green plus button (shown circled in the following screenshot).



You will then be presented with a list of all the modules you have installed. Select the **ti.draggable [android]** module, as shown circled in the following screenshot. After selecting this module, click on **OK** to add the reference module to your project.

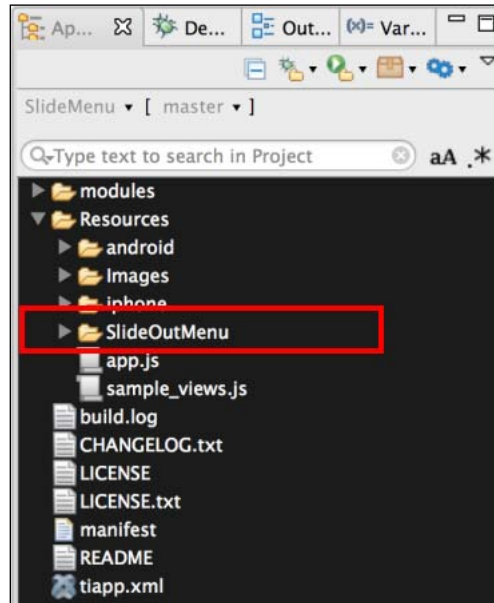


Once you have added the module, make sure you save your project before running it. After saving, you will need to clean your project. You do this by selecting **Clean...** under the **Project** toolbar option in Titanium Studio, as demonstrated in the following screenshot:



Adding the Slideout Menu module to your project

Adding the `Slideout Menu` module to your project is easy. Simply copy the `SlideOutMenu` folder into the `Resources` folder of your Titanium project, as shown in the following screenshot. This will install the CommonJS module and all supporting images.



The `SlideOutMenu` folder must be located in the root folder of your Titanium project in order for this recipe to work as designed.

How to do it...

Once you've added the `SlideOutMenu` folder to your project, you need to use `require` to import the module into your code.

```
//Create our application namespace
var my = {
  views:{},
  menu : require('./SlideOutMenu/MASlidingMenu')
};
```

Defining our content views

For this sample, we have created a CommonJS module that works as a view factory. In your app, these views would be the content you wish to have displayed on the screen.

```
my.sampleView = require('sample_views');
```

The following snippet builds a `views` object that contains our placeholder views created by the sample view factory.

```
my.views = {
  home : my.sampleView.placeholderView({
    title:'Home',backgroundColor:'blue'}),
  inbox : my.sampleView.placeholderView({
    title:'Home',backgroundColor:'green'}),
  sales : my.sampleView.placeholderView({
    title:'Home',backgroundColor:'yellow'}),
  customers : my.sampleView.placeholderView({
    title:'Home',backgroundColor:'blue'}),
  about : my.sampleView.placeholderView({
    title:'About',backgroundColor:'purple'})
};
```

Building our menu items

The next step in our recipe is to define our menu items. When open, the menu will display an array of menu items, as shown by `menuData` in the following snippet.

Each menu item can contain the following properties:

- ▶ **Title:** This property is used to display the menu text of the item.
- ▶ **Detail Cue:** The property `hasDetail` determines if the more arrow visual cue should be displayed. This property is optional, and by default `false`.
- ▶ **View Pointer:** The optional property `view` is used to hold a reference to the view that should be displayed when the menu item is pressed.

```
var menuData = [
  { title:'Home', hasDetail:true, view: my.views.home },
  { title:'Inbox', hasDetail:true, view: my.views.inbox },
  { title:'Sales', hasDetail:true, view: my.views.sales },
  { title:'Customers', hasDetail:true,
view: my.views.customers },
  { title:'About', hasDetail:true, view: my.views.about },
  { title:'Logout' }
];
```

Opening the app window

To open the sample window, you first need to call the `addMenuItems` function and provide the `menuData` array that we created earlier. This will create the menu objects within the module. Next, you need to pipe the result of the `addMenuItems` function into the `open` method. This will open your main application window and display the first view in your menu.

```
my.menu.addMenuItems(menuData).open();
```

Adding menu listeners

The Slideout menu provides listeners for `buttonclick`, `switch`, `open`, `close`, and `sliding`. The following details each of these events as used in the sample.

1. When a menu item is tapped, the `buttonclick` event is fired, providing the menu item index.

```
my.menu.addEventListener('buttonclick', function(e) {
  if (e.index === 2) {
    alert('You clicked on Logout');
  }
});
```

2. When a display view is switched for another view, the `switch` event is fired. This event can be used to help load display-view content.

```
my.menu.addEventListener('switch', function(e) {
  Ti.API.info('menuRow = ' + JSON.stringify(e.menuRow));
  Ti.API.info('index = ' + e.index);
  Ti.API.info('view = ' + JSON.stringify(e.view));
});
```

3. As the user slides the display view to expose the menu, the `sliding` event is fired. This provides the view distance.

```
my.menu.addEventListener('sliding', function(e) {
  Ti.API.info('distance = ' + e.distance);
});
```

4. When the menu is fully displayed, the `open` event is fired. This event is helpful for tracking visual state.

```
my.menu.addEventListener('menuOpened', function(e) {
  Ti.API.info('Menu is open');
});
```

5. When the menu is fully closed, the `menuClosed` event is fired. This event is helpful for tracking visual state.

```
my.menu.addEventListener('menuClosed ', function(e) {
  Ti.API.info('Menu is closed');
});
```

Adding custom application listeners

Our example app also uses the following custom application level events. These events are fired by our sample views to close and toggle the menu.

```
Ti.App.addEventListener('app:toggle_menu',function(e){
  Ti.API.info('App Listener called to toggle menu');
  my.menu.toggle();
});
Ti.App.addEventListener('app:close_menu',function(e){
  Ti.API.info('App Listener called to close menu');
  my.menu.hide();
});
```

How it works...

The `Slideout Menu` module provides several functions to help you build your app's navigation system.

Creating the menu

The next snippet demonstrates how to use the `addMenuItems` helper function to create a menu using a list of menu items. See this recipe's *How to do it...* section for an example.

```
my.menu.addMenuItems(menuData);
```

Opening the menu container

The following line demonstrates how to open the menu container and display the first menu item. See this recipe's *How to do it...* section for an example.

```
my.menu.open();
```



The `addMenuItems` function needs to be called before the menu container can be opened.

Showing the menu

The following line demonstrates how to fully display the menu:

```
my.menu.expose();
```

Toggling the menu

When designing your navigator, you will want to implement a button to allow your users to open and close the menu. The `toggle` method, shown in the following snippet, performs this operation for you:

```
my.menu.toggle();
```

Closing the menu

The following line demonstrates how you can hide the menu:

```
my.menu.hide();
```

Determining menu state

You can use the following line to determine if the menu is now fully in view:

```
Ti.API.info("Menu is open " + my.menu.isOpen());
```

Accessing the current view

You will often need to access the currently displayed view to perform input validation or other operations. The following line demonstrates how you can easily access the current view:

```
var current = my.menu.currentView();
```

Closing the menu container

The following line demonstrates how to close the menu container and all associated display views:

```
my.menu.dispose();
```

Using the sample's Global Events

Each of our sample views can fire two helpful custom-application-level events. The first of these events is `app:toggle_menu`, which is called when you tap on the menu icon. This will either show or hide the menu, depending on its current status.

```
toMenu.addEventListener('click', function(){
  Ti.App.fireEvent('app:toggle_menu');
});
```

Often, a user wants to start working with the screen before the menu has fully closed. To facilitate this, each display view will call the `app:close_menu` event on a double tap. This will automatically close the view and allow the user to continue working with the current screen.

```
view.addEventListener('dblclick',function(e){
  Ti.App.fireEvent('app:close_menu');
});
```

See also

- ▶ This recipe is inspired by and contains forks of several great open source projects. The `MASlidingMenu.js` CommonJS module used in this recipe is the fork of the `MASlidingMenu` project.
 - ❑ `MASlidingMenu`: Available at <https://github.com/appersonlabs/MASlidingMenu>. Author: Matt Apperson.
 - ❑ `TiDraggable`: Available at <https://github.com/pec1985/TiDraggable>. Author: Pedro Enrique.
 - ❑ `CoverSliderExample`: Available at <https://github.com/atsusy/CoverSliderExample>. Author: Atsushi Kataoka.

3

Using Databases and Managing Files

In this chapter, we will cover:

- ▶ Accessing your database's Ti.Filesystem
- ▶ DbTableChecker SQLite table existence checking
- ▶ Recursively handling files using Dossier
- ▶ Tuning your SQLite database for maximum performance
- ▶ Data access using DbLazyProvider
- ▶ NoSQL using MongloDb

Introduction

Titanium has several components designed to assist in handling your data needs. Additionally, the Titanium open source community has provided a wealth of alternative data-handling options.

The recipes in this chapter demonstrate a variety of helpful approaches to work with data while on the device. With topics ranging from file-system management to SQLite tuning, there will be an approach to assist your Enterprise data needs, no matter what your preferred approach is.

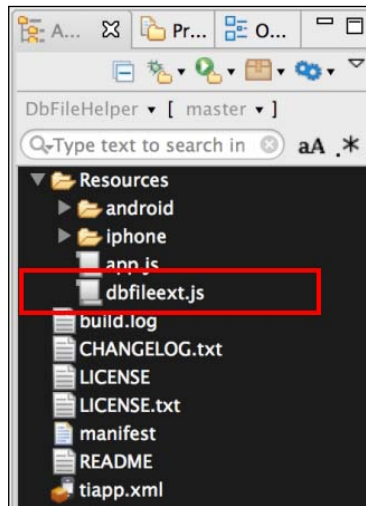
Accessing your database's Ti.Filesystem

Titanium includes a powerful database API that provides easy access to the underlying platform's SQLite implementation. Having access to `Ti.Filesystem` for your app's SQLite database allows you the flexibility to handle updates, migrations, and installations more effectively.

The `DbFileExt` module provides easy access to your SQLite `Ti.Filesystem` object. The module also includes other convenient methods to help work with database files.

Getting ready

Adding the `DbFileExt` module to your project is easy. Simply copy the `dbfileext.js` file into your project, as shown in the following screenshot:



How to do it...


Once you've added the `dbfileext.js` file to your project, you need to use `require` to import the module into your code:

```
//Create our application namespace
var my = {
  isAndroid : Ti.Platform.osname === 'android',
  dbfileext : require('dbfileext')
};
```

Finding our database `Ti.Filesystem.File`

The `dbFile` function provides easy access to the `Ti.Filesystem.File` object for your database. Simply provide the database name you wish to access, and the associated `Ti.Filesystem.File` object will be returned:

```
Var myDbFile = my.dbfileext.dbFile('testdb');
```

 If an invalid database name is provided to the `dbFile` function, a new `Ti.Filesystem.File` object will be returned. This allows you to add a database or file later. It is recommended to use the `exists` method when any operations are performed.

Determining the database directory

Determining your application's database folder can be challenging, as it differs according to the platform. Leveraging the `dbDirectory` function assists with this challenge, providing you specify the appropriate directory path based on the device your app is running on:

```
my.dbfileext.dbDirectory();
```

File exist check


You will often need to check if a database has been installed, before opening. The `DbFileExt` module makes this straightforward with the `dbExists` function. Simply paste in the database name and a Boolean is returned, identifying if the database has been installed:

```
my.dbfileext.dbExists('testdb');
```

RemoteBackup – iOS-specific attribute

With iOS 5, Apple included the ability to back up files to the iCloud service. You can enable or disable this feature on your SQLite database file through the use of the `dbRemoteBackup` function. When you provide the database name and a Boolean, you are indicating if you would like to have remote backup enabled:

```
my.dbfileext.dbRemoteBackup('testdb', true);
```

 If this function is called on the Android platform, no action will be performed, as this relates to iOS-specific functionality.

Renaming a database file

The `DbFileExt` module allows you to rename your database files. This can be helpful when versioning or keeping backups of your database. To rename a database, provide the current database name and the new name as parameters to the `dbRename` method, as the following statement shows:

```
my.dbfileext.dbRename('current name', 'new name');
```

Listing all databases

When dealing with large apps, or apps that require database versioning, you will often need a list of all the databases installed within your app's sandbox. The `dbList` function provides the name and native path for each SQLite database within your app's database folder. You can then use this array to remove any unneeded database files:

```
var installedDatabase = my.dbfileext.dbList();
```

Removing a database file

The `dbRemove` function provides an easy and safe way to remove any unwanted database files. Provide the database name you wish to delete and the `DbFileExt` module will remove the file from your device:

```
my.dbfileext.dbRemove('newtest');
```

How it works...

The next series of tests demonstrate how the `DbFileExt` module works within a windowless sample `app.js`.

Setting up our test

First, we use the `Ti.Database.open` function to create a sample database. The following highlighted code demonstrates how to create a database named `testdb`:

```
//Create our application namespace
var my = {
  isAndroid : Ti.Platform.osname === 'android',
  dbfileext : require('dbfileext')
};

var testDb = Ti.Database.open('testdb');
```

Next, we build a `Ti.Filesystem` object with a known reference to our `testdb` database. This will later be used to verify if the `DbFileExt` module is returning the correct values.

```
var testFileReference = (function(){
  if(my.isAndroid){
```

```

    return Ti.Filesystem.getFile(
  }else{
    return testDb.file;
  }
}) ();

```



The iOS `Ti.Database` object has a `file` property that can be used in your tests. On Android, you need to create the file object using the proper directory and file names.

Finding our database `Ti.Filesystem`

Next, we compare the `Ti.Filesystem.File` object returned by the `DbFileExt` module to see if it matches our test file. These should always match no matter what the platform.

```

Ti.API.info("Does the module return the same as our test?");
Ti.API.info((testFileReference.nativePath ===
  my.dbfileext.dbFile('testdb').nativePath) ?
  "Test Pass" : "Test Failed");

```

Determining database directory

The directory where your SQLite file is installed differs by platform. For example, it is in the `data` directory on Android and the `Private Documents` folder on iOS. The next example demonstrates how to write the path to your devices database directory to Titanium Studio's console:

```

Ti.API.info("Your database directory is " +
  my.dbfileext.dbDirectory());

```

File exist check

The most common use of the `DbFileExt` module is to check if a database has already been installed. The next test compares the `dbExists` result in `DbFileExt` with those generated by our test `Ti.Filesystem.File` object:

```

Ti.API.info("Does the exists test work?");
Ti.API.info((testFileReference.exists() ===
  my.dbfileext.dbExists('testdb'))
  ? "Both Exist" : "Test Failed");

```

Renaming a database file

You will encounter the need to rename databases for a variety of reasons, the most common being archiving or versioning. The following highlighted code demonstrates how to rename the `testdb` database to `oldtest`. The `dbExists` method is then called on both the new and old names to demonstrate that the rename function has worked properly.

```
my.dbfileext.dbRename('testdb','oldtestdb');
Ti.API.info("Does the test db exist? " +
  my.dbfileext.dbExists('testdb'));
Ti.API.info("How about the oldtest one? Does that exist? " +
  my.dbfileext.dbExists('oldtest'));
```

Listing all databases

It is often helpful to have a list of all the databases that are available within your app. This can be particularly helpful with large apps or apps containing third-party components.

To demonstrate this functionality, first we create several databases in our sample app:

```
Ti.Database.open('test1');
Ti.Database.open('test2');
Ti.Database.open('test3');
```

The `dbList` function of the `DbFileExt` module is used to return all the databases installed into our app:

```
var installedDb = my.dbfileext.dbList();
```

The following snippet writes the list of databases provided by the `installedDb` variable to Titanium Studio's console. This will list all of the databases used in this sample, along with any that might also be installed within your Titanium project.

```
installedDb.forEach(function(db) {
  Ti.API.info("Db Name = " + db.dbName);
  Ti.API.info("nativePath = " + db.nativePath);
});
```

Removing a database file

The final test in this recipe demonstrates how to use the `dbRemove` function to delete an installed database. The following highlighted snippet shows how to delete the `oldtest` database that we used earlier in the sample:

```
my.dbfileext.dbRemove('oldtest');
Ti.API.info("db still exists? " +
  my.dbfileext.dbExists('oldtest'));
```

DbTableChecker SQLite table existence checking

SQLite and the `Ti.Database` API provide many powerful features. This recipe demonstrates how to use the existing `Ti.Database` API and SQL statements to check if a table has already been created.

The ability to check if database objects exist is critical for Enterprise apps, for versioning and data migration purposes. A typical usage of this process would be to check if a table already exists during schema migration. For example, a specific data migration might be performed if the client table in your app already exists while creating the table for the first time.

Getting ready

This recipe relies on the Titanium framework's `Ti.Database` API, requiring no dependencies. In the next section, we create a simple `app.js` file, demonstrating how to perform the table-exists check.

How to do it...

The following steps demonstrate how to check if a table exists within an SQLite database.

Creating our module

In Titanium Studio, create a module called `dbtablechecker.js`. This module should have the following code snippet:

```
exports.tableExists = function (dbName, tableName) {
    var conn = Ti.Database.open(dbName);
    var selectSQL = 'SELECT name FROM sqlite_master '
        selectSQL += ' WHERE type="table" AND name=?';

    var getReader = conn.execute(selectSQL, tableName);
    var doesExist = (getReader.getRowCount() > 0);

    //Clean-up
    getReader.close();
    conn.close();
    getReader = null;
    conn = null;

    return doesExist;
};
```


Namespace and app setup

Next, in our application's `app.js` file, we create our application namespace and use the `require` method to import the `CommonJS` module into our application:

```
//Create our application namespace
var my = {
  dbTableChecker : require('dbtablechecker')
};
```

Creating our window

To help demonstrate this recipe, a window with three buttons is created. These buttons will allow you to create, test whether the table exists, and remove the `myTable` sample table.

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff',layout:'vertical'
});
```

Testing if a table exists

The first button in the recipe demonstrates how to call the `tableExists` function created earlier in the `CommonJS` module of `dbTableChecker`:

```
var tableExistsBtn = Ti.UI.createButton({
  title:'Does Table Exist?', height:50, right:5,
  left:5, top: 20
});
win.add(tableExistsBtn);
```

When the button is pressed, the `tableExists` function is called within our `CommonJS` module to determine if the specified table exists. The next highlighted code snippet demonstrates checking if a table named `myTable` exists within the `myDatabase` `SQLite` database.

```
tableExistsBtn.addEventListener('click',function(e){
  //Check if our table exists in our database
  var doesExist = my.dbTableChecker.
    tableExists('myDatabase','myTable');
  //Alert the user if the table exists or not
  alert('Table "myTable" ' + (doesExist ? ' exists' :
    "does not exist"));
});
```

Creating a table

The next button in this recipe is used to create a sample table, which is to be used in our tests. When exploring this recipe, this button will allow you to drop and re-create a sample table several times.

```
var makeTestBtn = Ti.UI.createButton({
  title:'Create Test Table', height:50, right:5, left:5, top: 20
});
win.add(makeTestBtn);
```

The following highlighted code demonstrates calling the `my.testers.makeTable` function to create a table named `myTable` in the `myDatabase` database:

```
makeTestBtn.addEventListener('click',function(e){
  //Create a sample table
  my.testers.makeTable('myDatabase','myTable');

  //Alert the user a test table has been created
  alert('Table "myTable" was created.');
```



See this recipe's *How it works...* section for more information on `my.testers`.

Removing a table

The last button in this recipe is used to drop the sample table. When exploring this recipe, this button will allow you to drop and re-create a sample table several times:

```
var removeTestBtn = Ti.UI.createButton({
  title:'Remove Test Table', height:50, right:5, left:5, top: 20
});
win.add(removeTestBtn);
```

The highlighted line in the following code snippet demonstrates how to call the `my.testers.removeTable` function to drop a table named `myTable` in our database:

```
removeTestBtn.addEventListener('click',function(e){
  //Create a sample table
  my.testers.removeTable('myDatabase','myTable');
  //Alert the user a test table has been removed
  alert('Table "myTable" was removed.');
```



See this recipe's *How it works...* section for more information on `my.testers`.

How it works...

This recipe uses the SQLite data dictionary and several helper methods to support testing. The functionality of these methods and how they are composed, is discussed here.

Testing helpers

This recipe uses two helper functions, `makeTable` and `dropTable`, to manage our sample table. These methods allow for the `tableExists` method to be tested repeatedly without conflict.

```
my.testers = {
```

1. The `makeTable` function uses `dbName` to open a database connection. Once the database has been opened, a table is created (if it doesn't exist) using the provided `tableName` parameter:

```
makeTable : function(dbName,tableName) {
    var conn = Ti.Database.open(dbName);
    var createSql = 'CREATE TABLE IF NOT EXISTS '
    createSql += tableName ;
    createSql += '(id INTEGER PRIMARY KEY AUTOINCREMENT, '
    createSql += ' my_column TEXT)';

    conn.execute(createSql);

    //Clean-up
    conn.close();
    conn = null;
},
```

2. The `removeTable` function uses `dbName` to open a database connection. Once the database has been opened, the table name provided in the `tableName` parameter is dropped, if the table exists:

```
removeTable : function(dbName,tableName) {
    var conn = Ti.Database.open(dbName);
    var dropSql = 'DROP TABLE IF EXISTS ' + tableName;

    conn.execute(dropSql);
```

```

        Conn.close();
        conn = null;
    }
};

```

The tableExists method

The `dbTableCheck` module has a single method named `tableExists` which returns a Boolean result if the provided table name exists within the database. This check is performed by querying the data dictionary of the SQLite database.

```

exports.tableExists = function (dbName, tableName) {
    var conn = Ti.Database.open(dbName);

```

1. The following SQL statements will query the SQLite data dictionary table, `sqlite_master`, for a list of tables with a specified table name. The `?` character is a parameter that is replaced by the `execute` method.

```

        var selectSQL = 'SELECT name FROM sqlite_master;
        selectSQL += ' WHERE type="table" AND name=?';

```

2. The data dictionary query is then executed using the `tableName` variable as its parameter. A `Ti.Database.DbResultSet` is returned and allocated to the `getReader` variable:

```

        var getReader = conn.execute(selectSQL, tableName);

```

3. Next, the `getRowCount` method is used to determine if any rows are returned. This is converted into a Boolean statement that will later be returned by this method:

```

        var doesExist = (getReader.getRowCount() > 0);

```

4. The `DbResultSet` and database is then closed to reduce the number of active objects:

```

        getReader.close();
        conn.close();
        getReader = null;
        conn = null;

```

5. The Boolean result determined earlier is returned by the following method:

```

        return doesExist;
    };
};

```

Recursively handling files using Dossier

When creating your Titanium Enterprise app, you will often find the need to copy the contents of a directory to another place. Two of the most common examples of this would be: implementing a caching approach and performing lazy, loaded installations. For example, Dossier can be used to create an initial content cache, by copying files from your app's `Resources` directory into a working directory under `Ti.Filesystem.applicationDataDirectory`. This would allow for the user to see the initial content while data is being refreshed in the background.

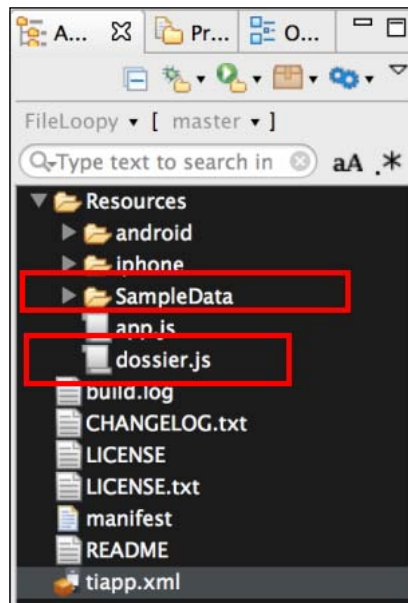
The `Dossier` CommonJS module provides a cross-platform API for handling these types of folder operations. The next section demonstrates how to install and use the `Dossier` module within your Titanium Enterprise app.

Getting ready

The `Dossier` CommonJS module is installed by including the `dossier.js` file into your project.

Adding the Dossier module into your project

Adding the `dossier` module to your project is easy. Simply copy the `dossier.js` file and the `SampleData` folder into the `Resources` folder of your Titanium project, as highlighted in the following screenshot. This will install all the files needed for this recipe.



How to do it...

Once you've added the `dossier.js` file to your project, you need to use `require` in order to import the module into your code:

```
//Create our application namespace
var my = {
  dossier : require('dossier');
};
```

Creating sample directories

To demonstrate the copy and move features of Dossier, a `Ti.Filesystem` object is created for both our source and destination directories. In the next snippet, the `sourceDir` variable contains a directory reference to the `SampleData` folder we copied as part of the recipe setup, and `targetDir` references a new folder named `NewSampleData`, to be created in your device's data directory.

```
var sourceDir =
  Ti.Filesystem.getFile(Ti.Filesystem.resourcesDirectory,
    'SampleData');

var targetDir = Ti.Filesystem.getFile(
  Ti.Filesystem.applicationDataDirectory
  + '/NewSampleData');
```

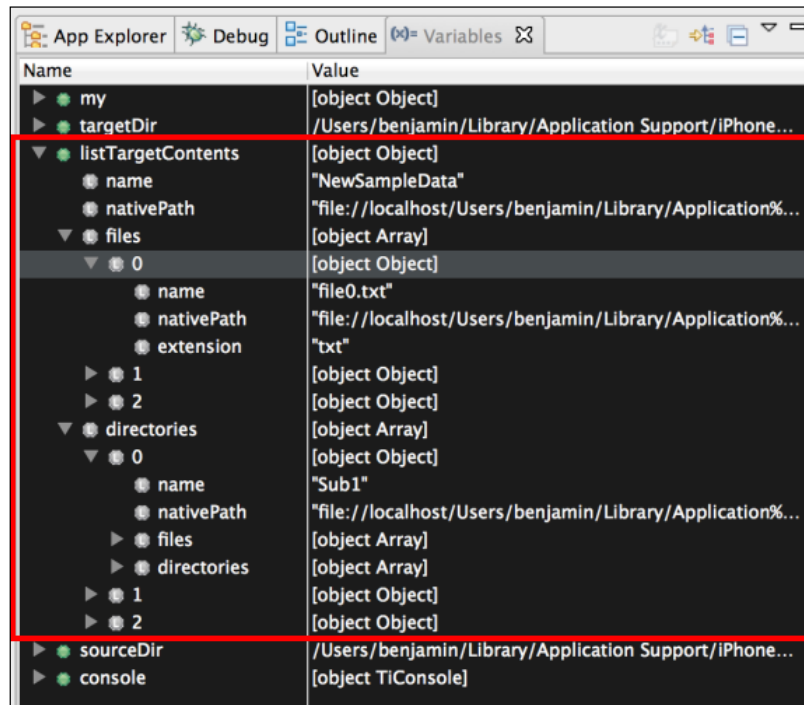
Recursive listing of directory contents

When dealing with dynamic or downloadable content, you will often need to list all of the content of a specific directory. The `listContents` method allows you to recursively query the content of a directory, listing all files and subdirectories within.

The following code snippet demonstrates how to query our recipe's `sourceDir` for a list of all files and folders:

```
var listTargetContents =
  my.dossier.listContents(sourceDir.nativePath);
```

The `listContents` method returns a file explorer dictionary object, listing all of the files and subdirectories in a hierarchal format, as highlighted in the following screenshot:




Recursively copying directory contents

Many Enterprise apps are content driven. To improve your first-time installation experience, you might wish to bundle introductory content within your app and copy this bundled content into local cache on startup.

Using `dossier` module's `copy` method, you can copy the entire content of one folder to another. The following code demonstrates how to copy all the content of our source directory to our new target directory:

```
my.dossier.copy(sourceDir.nativePath, targetDir.nativePath);
```

 During the copy process, any existing content in the target folder will be removed.

Recursively moving directory contents

The `move` method creates a new copy of all the content of your source directory to your target folder. Once the copying process has been successfully completed, the source directory is removed.

```
my.dossier.move(sourceDir.nativePath, targetDir.nativePath);
```



During the move process, any existing content in the target folder will be removed and replaced with the content of our source folder. Additionally, after all the files have been moved, the source content will be removed if possible. In cases where the source directory is read-only, the contents will be retained.

See also

- ▶ The `Dossier` module used in this recipe is an open source project available on Github. If you are interested in learning more or contributing, please visit the project at <https://github.com/benbahrenburg/Dossier>.

Tuning your SQLite database for maximum performance

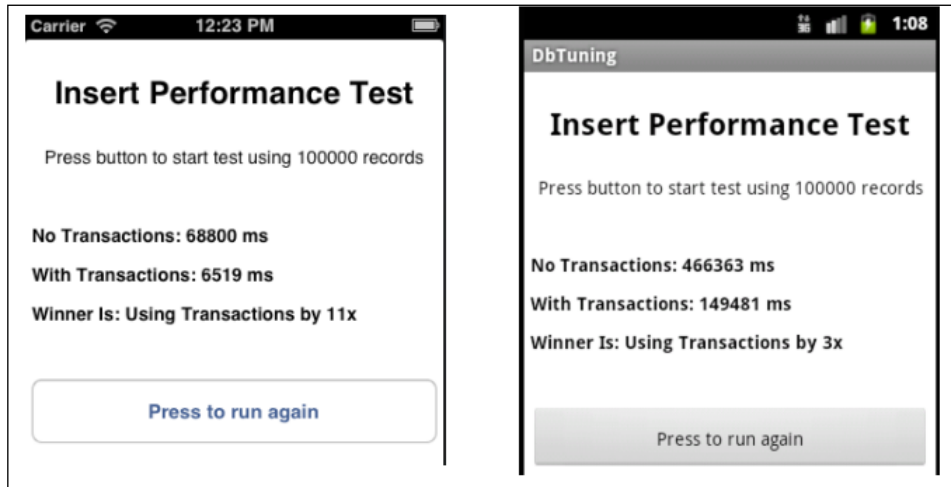
Data access is a common bottleneck in Titanium Enterprise development. Through proper use of SQLite transactions, you can experience up to a 10x improvement in bulk insert operations.

SQLite transactions provide reliable units of work that allow for data recovery and keep the database consistent. By default, each time an insert, update, or delete operation is performed on an SQLite database, an implicit transaction is created before and after your statement is executed. This helps keep your database in a consistent state. For batch operations however, this introduces an additional level of overhead and can drastically reduce your app's performance.

This recipe demonstrates how to use SQLite transactions to improve app performance when conducting batch SQL actions and working with large datasets.

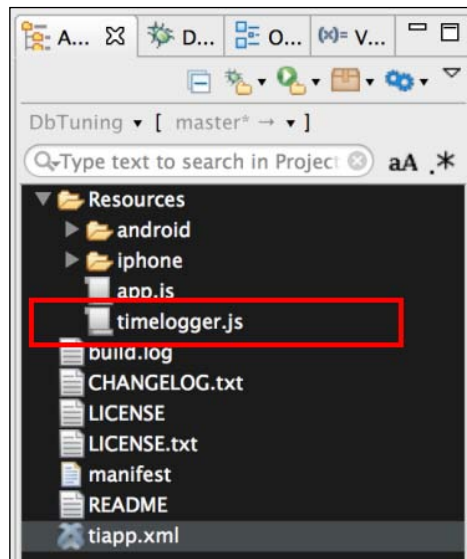
Getting ready

This recipe will run a performance comparison using 100,000 rows. Included in this recipe is the following basic UI that allows you to benchmark your different devices.



Adding the TimeLogger module

This recipe uses `TimeLogger` to record the duration of each performance test. To add the `TimeLogger` module to your project, copy the `timelogger.js` file into your project, as shown in the following screenshot:



How to do it...

The first step in this recipe is to create the application namespace and import the timer module, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  timer : require('timelogger')
};
```

Creating our testing Interface

The next step is to create a `Ti.UI.Window` for our recipe. This will be used to provide a launching point for our performance tests.

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff',layout:'vertical'
});
```

1. Now, a label is created to track the results of our No Transactions performance test:

```
var noTransactionLabel = Ti.UI.createLabel({
  text: "No Transactions: NA",
  height:20, right:5, left:5, top: 40,
  textAlign:'left',
  color:'#000', font:{fontWeight:'bold',fontSize:14}
});
win.add(noTransactionLabel);
```

2. Then, a label is created to track the results of our With Transactions performance test:

```
var withTransactionLabel = Ti.UI.createLabel({
  text: "With Transactions: NA",
  height:20, right:5, left:5,
  top: 10, textAlign:'left',
  color:'#000', font:{fontWeight:'bold',fontSize:14}
});
win.add(withTransactionLabel);
```

3. The final UI element we create is a button that, when pressed, will launch our tests:

```
var runTestButton = Ti.UI.createButton({
  title:'Start Performance Test',
  height:50, right:5, left:5, top: 40
});
win.add(runTestButton);
```

Benchmarking

When the `runTestButton` button is pressed, a benchmark 100,000 record insert is performed, both with and without the use of transactional scoping. The screen is then updated with the elapsed milliseconds for each test.

1. The following code demonstrates how each test is run when the button's click event is triggered:

```
runTestButton.addEventListener('click', function(e) {
```

2. First, a test is run without explicitly creating a transaction. This test returns the number of milliseconds elapsed while running, and stores the result in the `noTransactions` variable, as the following snippet demonstrates:

```
var noTransactions = performanceTest.run(false);  
noTransactionLabel.text = "No Transactions: " +  
    noTransactions + ' ms';
```

3. Next, the same test is run using `BEGIN` and `COMMIT` statements to create an explicit transaction. This test returns the milliseconds elapsed during the test, and stores the result in the `withTransactions` variable, as shown in the following snippet:

```
var withTransactions = performanceTest.run(true);  
withTransactionLabel.text = "With Transactions: " +  
    withTransactions + ' ms';  
});
```

How it works...

This recipe uses the helper objects discussed in this section, to perform the benchmarking operations.

Database setup

`dbTestHelpers` is the first helper object used during benchmarking. This object contains all the code needed to set up, create, and manage the database used in our tests:

```
var dbTestHelpers = {
```

1. The `maxIterations` property controls the number of iterations in our test:
`maxIterations : 100001,`
2. The `createOrResetDb` method is used to return a known and consistent database to test against:

```
    createOrResetDb : function() {  
        return Ti.Database.open("perf_test");  
    },
```

3. The `resetTestTable` method is then called to drop and re-create our test table. This allows us to run our tests several times, while maintaining a consistent sample size:

```
resetTestTable : function(db){
    var dropSql = 'DROP TABLE IF EXISTS TEST_INSERT';
    var createSql = 'CREATE TABLE IF NOT EXISTS ' +
        'TEST_INSERT ' +
        '(TEST_ID INTEGER, TEST_NAME TEXT, ' +
        'TEST_DATE DATE)';
    db.execute(dropSql);
    db.execute(createSql);
},
```

4. The `createSQLStatement` method returns the SQL that will be used to perform our insert operations:

```
createSQLStatement : function(){
    var sql = 'INSERT INTO TEST_INSERT ' +
        '(TEST_ID, TEST_NAME, TEST_DATE) ' +
        'VALUES(?,?,?)';
    return sql;
},
```

5. The `createDummyObject` function creates a unique object, to be inserted into each row:

```
createDummyObject : function(iterator){
    var dummy = {
        id:iterator, name : 'test record ' + iterator,
        date : new Date()
    };

    return dummy;
}
};
```

Performing the tests

The `performanceTest` object runs and times the recipe database's inserts. The `run` method starts our benchmark and provides an indicator if the test is to use transactions or not:

```
var performanceTest = {
    run : function(useTransaction){
```

1. The first step in our test is to create a database connection and to reset our table. This is done by calling the `dbTestHelper` method, discussed earlier:

```
var db = dbTestHelpers.createOrResetDb();
dbTestHelpers.resetTestTable(db);
```

2. After our database has been set up, the next step is to create our `insert` statement and `timer` objects, as demonstrated here:

```
var dummyObject = null;
var insertSQL = dbTestHelpers.createSQLStatement();
var insertTimer = new my.timer("Insert Timer");
```

3. If the `useTransaction` flag has been set, we then explicitly begin a transaction:

```
if(useTransaction){
    db.execute('BEGIN;');
}
```

4. In the next step of this recipe, a loop is created, to insert records into the test table, a specific number of times. By default, this test will insert 100,000 records, and time the total duration.

```
for (var iLoop = 0;
    iLoop < dbTestHelpers.maxIterations; iLoop++){
    dummyObject = dbTestHelpers.createDummyObject(iLoop);
    db.execute(insertSQL, dummyObject.id,
        dummyObject.name, dummyObject.date);
}
```

5. If the `useTransaction` flag has been set, we then explicitly "commit" the transaction:

```
if(useTransaction){
    db.execute('COMMIT;');
}
```

6. The final step of this method is to retrieve our execution duration from our `timer` object. This value (in milliseconds) is then returned for later comparison operations:

```
var totalInsertTime = insertTimer.getResults().msElapsed;
db.close();
//Return total ms elapsed
return totalInsertTime;
}
};
```

See also

- ▶ To learn more about SQLite transactions and how they are implemented, please read the official documentation at http://www.sqlite.org/lang_transaction.html

Data access using DbLazyProvider

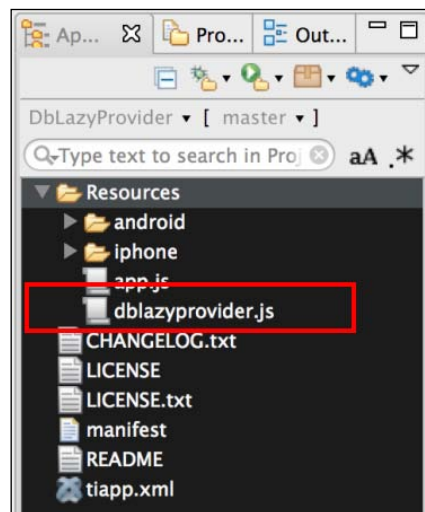
Data access is a common challenge when building any Enterprise app. The `DbLazyProvider` module provides a lightweight wrapper around the `Ti.Database` API. This module provides helpers for commonly required operations such as managing transactions and lazy-loading connections.

Lazy loading is a common efficiency pattern that defers the initialization of an object until needed. By lazy-loading app-database connections, any memory usage or IO operation associated with creating a database connection is deferred until needed.

The following section demonstrates how to use the `DbLazyProvider` module to implement a lazy-loading pattern in your app, while maintaining control over your database transactions.

Getting ready

Adding the `DbLazyProvider` module to your project is easy. Simply copy the `dblazyprovider.js` file into your project, as shown in the following screenshot:



How to do it...

The first step in this recipe is to create the application namespace and import the `DbLazyProvider` module, as demonstrated here:

```
//Create our application namespace
var my = {
  dbProvider : require('dblazyprovider')
};
```

Next, a sample table named `MY_TEST` is created. If the table already exists, any existing records will be purged so we can start a new test.

```
var dbSetup = new my.dbProvider("myDb");
var createSql = 'CREATE TABLE IF NOT EXISTS ';
createSql += 'MY_TEST (TEST_ID INTEGER, ';
createSql += 'TEST_NAME TEXT)';
dbSetup.connect().execute(createSql);
dbSetup.connect().execute('DELETE FROM MY_TEST');
dbSetup.close();
```

Creating our testing Interface

Now, we create a `Ti.UI.Window` for our recipe. This will be used to provide a launching point for our tests.

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff',layout:'vertical'
});
```

Lazy-loading using transactions

The `insertDemoBtn` button is used to trigger our recipe, showing how to use transactions with `DbLazyProvider`:

```
var insertDemoBtn = Ti.UI.createButton({
  title:'Run Inserts', height:50, right:5, left:5, top: 20;
});
win.add(insertDemoBtn);
```

1. Clicking on the `insertDemoBtn` button will insert 1,000 rows into our table:

```
insertDemoBtn.addEventListener('click',function(e){
  var maxIteration = 1000;
  var sql = 'INSERT INTO MY_TEST ';
  sql+='(TEST_ID, TEST_NAME) VALUES(?,?)';
  var db = new my.dbProvider("myDb");
  for (var iLoop = 0; iLoop < maxIteration; iLoop++){
```

- The following code shows using the `connect` method with a transaction parameter of `true` to obtain the `Ti.Database` object that is used to execute the SQL statement. This will automatically create a transaction on the first insert for you.

```
db.connect(true).execute(sql,iLoop,'test ' + iLoop);
```

- When the `close` method is called, any transactions that were created will be committed, and the database connection will be closed:

```
db.close();
alert('Test done ' + maxIteration + ' rows inserted');
});
```

Creating a select statement using lazy-loading

Not all SQL statements benefit from transactions. The following snippet demonstrates how to perform a select statement without using transactions:

```
var selectDemoBtn = Ti.UI.createButton({
    title:'Run Select', height:50, right:5, left:5,
    top: 20
});
win.add(selectDemoBtn);
```

- Click on the `selectDemoBtn` button to create a new `DbLazyProvider` object, and run a select statement:

```
selectDemoBtn.addEventListener('click',function(e){
    var db = new my.dbProvider("myDb");
    var selectSQL = 'SELECT * FROM MY_TEST ';
```

- The highlighted code shows how to use the `connect` method without providing any parameters. This will, by default, avoid using transactions.

```
var getReader = db.connect().execute(selectSQL);
var rowCount = getReader.getRowCount();
```

- On using the `close` method, the database connection will be closed and all objects will be set to null:

```
db.close();
alert('Rows available ' + rowCount);
});
```

How it works...

The `DbLazyProvider` module is a lightweight, yet powerful, wrapper over `Ti.Database`.

Creating a new DbLazyProvider object

To create a new `DbLazyProvider` object, simply use the `require` method to import the module and create a new object using the reference in the following highlighted snippet. This will create a new object wrapper for the database name provided.

```
//Create our application namespace
var my = {
  dbProvider : require('dblazyprovider')
};
//Create a new dbLazy object
var db = new my.dbProvider("myDb");
```

Getting a connection object

The most frequently used method of `DbLazyProvider` is `connect`. This method will create a new `Ti.Database` connection if needed, and then return the database object:

```
db.connect().execute('Your SQL goes here');
```

If you wish to have your SQL statement start a transaction, you simply pass in a Boolean parameter of `true` when calling the `connect` method, as shown here:

```
db.connect(true).execute('Your SQL goes here');
```

The transaction created will be used until either the `close` or `commit` method is called on your `DbLazyProvider` object.

Beginning a transaction

By default, transactions are handled automatically for you, using the `connect` and `close` methods. You can also explicitly create a transaction at any time using the `beginTransaction` method, as shown here:

```
db.beginTransaction();
```

Ending a transaction

By default, transactions are handled automatically for you, using the `connect` and `close` methods. You can also explicitly commit or finish a transaction at any time using the `commit` method, as shown here:

```
db.commit();
```

Opening a database connection

The module, by default, will wait until a database connection is needed, before opening the `Ti.Database` object. If a connection is needed in advance, you can call the `open` method at any time:

```
db.open();
```



You can also pass in a new database name to switch your database reference.

Closing a database connection

Using the `close` method on your `DbLazyProvider` objects is important, as it both commits any pending transactions and closes your database connection. The `close` method should be called after each transaction grouping or when the database connection is no longer needed.

```
db.close();
```

NoSQL using MongloDb

NoSQL databases are often a perfect fit for managing your application's data, as it allows you to work with objects instead of tables. In addition to the benefit of working with objects, using NoSQL on mobile reduces complexity by removing the need for schema management, data migration, and other common maintenance issues associated with maintaining a relational data model.

MongloDb is a pure JavaScript implementation of the popular MongoDB NoSQL solution. The `MongloDb` module allows you to query and persist objects using the familiar MongoDB syntax in your Titanium app. This recipe demonstrates how to leverage the `MongloDb` module within your new or existing Titanium project.

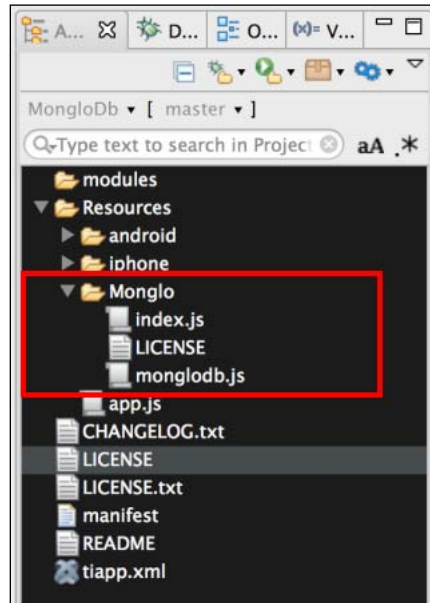
Getting ready

Installing MongloDb for Titanium is a straightforward process. You can either copy the `Monglo` folder from the recipe's source code, or create the bundle yourself.

Creating the bundle is a three-step process:

1. Using Titanium Studio, create a folder called `Monglo` in the `Resources` folder of your project.
2. Download the latest `monglobd.js` file from `monglobd.com` into the `Monglo` folder created in the previous step.
3. Download into the `Monglo` folder the latest `index.js` file from the Titanium Store project available on `monglobd.com`.

Whether you copied the bundle from the recipe source or created your own, the `Monglo` folder in your Titanium project should resemble the highlighted part of the following screenshot:



How to do it...

Once you have installed `MongloDb`, you will need to use `require` to import the module into your code:

```
//Create our application namespace
var my = {
  monglo : require('./Monglo/monglodb').Monglo
};
```

Initializing your database

After the module has been imported, a new instance of the database needs to be initialized. The following code demonstrates how to create a new instance of `Monglo` with the name `myDb`:

```
my.db = my.monglo('myDb');
```

Adding the Titanium storage provider

MongloDb has the ability to support a wide range of storage providers. This recipe implements the Titanium Store provider to persist data. Associating a storage provider with MongloDb is a two-step process. First, we require the Titanium storage provider, as shown here:

```
var tistore = require('./Monglo/index');
```

After the storage provider has been created, the provider is passed into the `use` method, as demonstrated in the following statement. Once the `use` method is called, MongloDb will perform all persistence operations using the Titanium Storage provider.

```
my.db.use('store', tistore);
```

Initializing our collection

Once the storage provider has been associated, durable collections can be created using the `collection` method. The following line demonstrates how to create a document collection named `foo`:

```
my.db.someCollection = my.db.collection('foo');
```

When initializing a named collection, any documents previously persisted by this collection will automatically be reloaded. In the previous example, any documents previously saved in the `foo` document collection will be reloaded when the collection is initialized.

Using events

MongloDb provides events for monitoring a majority of the actions performed. The following snippet demonstrates how to add an event for each supported listener:

```
my.db.someCollection.on('insert', function(){
  Ti.API.info("Document Inserted") ;
});
my.db.someCollection.on('update', function(){
  Ti.API.info("Document Updated");
});
my.db.someCollection.on('remove', function(){
  Ti.API.info("Document Removed");
});
my.db.someCollection.on('find', function(){
  Ti.API.info("Find Used");
});
my.db.someCollection.on('createCollection', function(){
  Ti.API.info("Collection Created");
});
my.db.someCollection.on('removeCollection', function(){
  Ti.API.info("Collection Removed");
});
```

Inserting documents

The next step is to insert three new documents. The next snippet demonstrates how to insert a new document into `someCollection`. The `insert` method has two parameters. The first parameter is a document object to be stored. The second parameter is a callback that lists the errors and provides document information. This is shown in the following highlighted snippet:

```
my.db.someCollection.insert({text: "record 1",
    batchId:'sample_test'}, function ( error, doc )
```

The `error` and `doc` objects are returned as part of the callback function. The `error` object contains any issues encountered during the insert action, and the `doc` object contains a copy of the `Mongo` document created.

```
    Ti.API.info('Error: ' + JSON.stringify(error));
    Ti.API.info('doc: ' + JSON.stringify(doc));
  });
  //Create second record
  my.db.someCollection.insert({text: "record 2",
    batchId:'sample_test'}, function ( error, doc ){ });
  //Create third record
  my.db.someCollection.insert({text: "record 3",
    batchId:'sample_test'}, function ( error, doc ){ });
```

Using find to query

With three records now created, we can use the `find` function to search for all the documents that have the batch ID `sample_test`.

```
my.db.someCollection.find({batchId:'sample_test'},
    function ( error, cursor ){
```

The `find` method returns both an `error` and `cursor` object. The cursor's `forEach` iterator provides a convenient way to inspect each document that is returned. The following snippet demonstrates how to print each document within the `cursor` to the Titanium Studio console as a JSON string:

```
    cursor.forEach(function(doc) {
        Ti.API.info('doc: ' + JSON.stringify(doc));
    });
  });
```

Updating documents

Similar to a traditional database, `MongoDb` provides the ability to change a document using the `update` method. The `update` method has three parameters. The first parameter is used to find the object you wish to update. The next example shows updating any object with the text property equal to `record 1`. The second parameter is the update statement. The example updates each matching object's text property to `updated record 1`. The final parameter is a callback which returns an `error` and `doc` object.

```

my.db.someCollection.update({text: "record 1"},
  {$set: {text: 'updated record 1'}},
  function ( error, doc ) {
    Ti.API.info('Error: ' + JSON.stringify(error));
    Ti.API.info('doc: ' + JSON.stringify(doc));
  });

```

This demonstrated how to update the text property on the first document inserted in this recipe.

Using `findOne` to query for a single document

The `findOne` method provides the means to query your collection for a specific document. The following snippet demonstrates how to query `someCollection` for the document we just updated. The resulting single-matching document is then printed to the Titanium Studio console as a JSON string.

```

my.db.someCollection.findOne({text: 'updated record 1'},
  function ( error, doc ) {
    Ti.API.info('Error: ' + JSON.stringify(error));
    Ti.API.info('doc: ' + JSON.stringify(doc));
  });

```

Removing documents

The final section of this recipe demonstrates how to remove documents from a collection. The following snippet demonstrates how to remove all documents that have the batch ID `sample_test`.

```

my.db.someCollection.remove({batchId: 'sample_test'},
  function (error) {
    Ti.API.info('Error: ' + JSON.stringify(error));
  });

```



The previous code snippet removes all records created in this recipe, allowing you to run the sample several times without creating unwanted records.

See also

All the NoSQL examples shown in this recipe use the MongoDB open source project. Please see the following to learn more about this project:

- ▶ Project site: <http://monglodb.com/>
- ▶ Github project: <https://github.com/Monglo/MongloDB>
- ▶ Titanium Store: <https://github.com/Monglo/MongloDB-Titanium-Store>
- ▶ Google Groups: <https://groups.google.com/forum/#!forum/monglodb>

4

Interacting with Web Services

In this chapter, we will cover:

- ▶ Consuming RSS feeds
- ▶ Creating a business location map using Yahoo Local
- ▶ Using Google Analytics in your app
- ▶ Making SOAP service calls using SUDS.js
- ▶ Using the LinkedIn Contacts API

Introduction

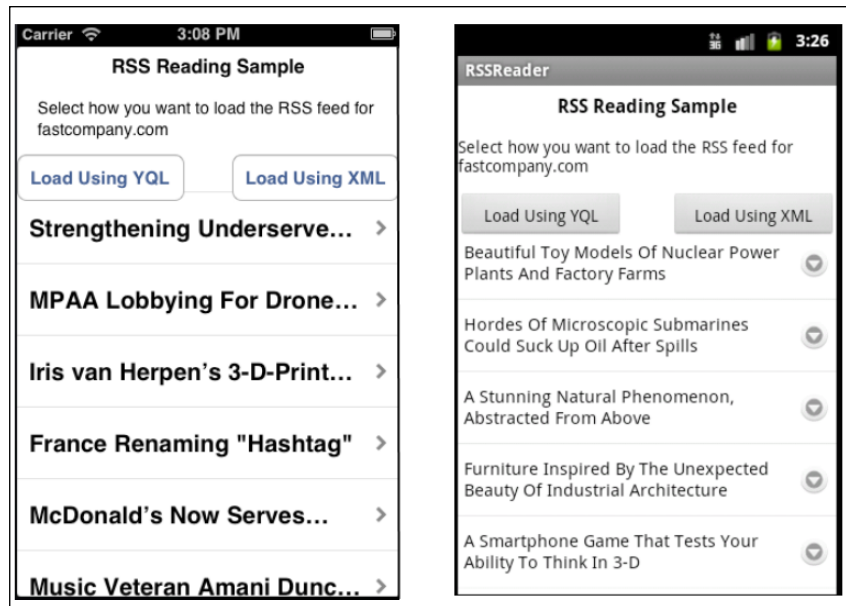
Mobile devices have been one of the driving forces behind the growth in variety and number of web service offerings in both Enterprise and Consumer spaces. As the ultimate disconnected client, mobile apps have revived the interest in **Service-Oriented Architecture (SOA)** as organizations look to extend their existing systems and functionalities to their mobile customers. As a result, you will rarely find an Enterprise app that does not use internal or third-party remote services.

This chapter demonstrates how to use SOAP and REST calls to interface with popular third-party platforms while building your Enterprise apps.

Consuming RSS feeds

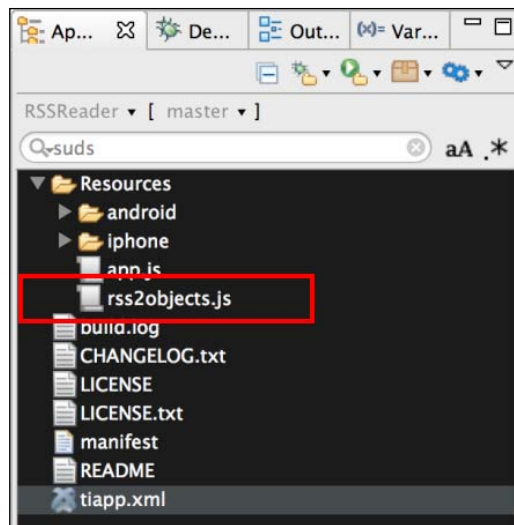
The use of RSS and ATOM feeds is common among Enterprise apps that update content periodically. You can use the techniques demonstrated in this recipe to publish content from your organization or a third party. Many companies use this approach to provide their employees with current news and updates regarding their organization or industry.

In this recipe, an RSS feed from `fastcompany.com` is consumed and displayed in a `Ti.UI.TableView` for the user to review and launch their browser to read the details.



Getting ready

This recipe uses the `rss2objects` CommonJS module. This module and other code assets can be downloaded from the source code provided by the book. Installing this module into your project is straightforward. Simply copy the `rss2objects.js` file into your project, as shown in the following screenshot:



How to do it...

Once you've added the `rss2objects.js` file to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code, as demonstrated in the following code block:

```
//Create our application namespace
var my = {
  reader : require('rss2objects'),
  isAndroid : ( Ti.Platform.osname === 'android'),
  rssTestUrl : "http://www.fastcompany.com/rss.xml"
};
```

Creating a UI for the sample app

The demonstration app for this recipe provides two `Ti.UI.Button` controls to illustrate different techniques for fetching RSS results, and a `Ti.UI.TableView` to display the articles.

1. First we create our `Ti.UI.Window` to attach all UI elements onto:

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff'
});
```

2. Next, a button is created to demonstrate how to use YQL to load an RSS feed:

```
var yqlButton = Ti.UI.createButton({
  title:"Load Using YQL",left:0 , width: 140,
  height:40, top:90
});
win.add(yqlButton);
```

3. A button is then added to demonstrate how to load an RSS feed using XML Parsing:

```
var xmlButton = Ti.UI.createButton({
  title:"Load Using XML",right:0 , width: 140,
  height:40, top:90
});
win.add(xmlButton);
```

4. Finally, a `Ti.TableView` is added to display the RSS articles:

```
var tableView = Ti.UI.createTableView({
  top:120, bottom: 0, width:Ti.UI.FILL
});
win.add(tableView);
```

Reading an RSS feed with YQL

Using Yahoo's YQL platform is a convenient way to read an RSS feed. YQL has two additional benefits over conventional XML parsing. The first benefit is YQL normalizes the feed output for you. The second benefit is Yahoo will convert the RSS stream to JSON. The downside to YQL is it will reduce download speeds due to the need to proxy the feed through Yahoo's servers.

This snippet demonstrates how to call the `yqlQuery` method of the `rss2Objects` module. The `yqlQuery` method takes two arguments: the first is the URL of the RSS feed you wish to read, and the second is a callback used to provide the feed item results.

```
yqlButton.addEventListener('click',function(e){
  my.reader.yqlQuery(my.rssTestUrl,rssResults);
});
```

Reading an RSS feed with XML parsing

The `rss2Objects` module also provides the ability to directly parse the RSS feed with XML. This option provides the best download performance for larger feeds.

To use the XML feed directly, simply call the `query` method on the `rss2Objects` module. The `query` method takes three arguments:

- ▶ The URL for the RSS feed you wish to read.
- ▶ An optional dot query syntax that allows you to provide the path of the nodes you want returned. In the next example, `channel.item` is provided. This tells the `query` method to only return nodes from the `channel item` element list. If you want all objects returned, simply pass a null value as an argument.
- ▶ The third argument is the callback method where the module will send the results once processing has been completed.

```
xmlButton.addEventListener('click',function(e){
  var queryPath = "channel.item";
  my.reader.query(my.rssTestUrl,queryPath,rssResults);
});
```

Displaying the articles

Both YQL and XML parsing demonstrations use the callback method shown next. This function takes the object dictionary provided by the `rss2objects` module and creates `TableView` rows for display:

```
function rssResults(e){
```

1. The parameter `e` in this function provides the result returned from the `rss2Objects` module. The first step is to check if the feed was returned successfully. This is done by checking the `e.success` property, as the following code block shows:

```
    if(!e.success){
        alert("Sorry we encountered an error");
        tableView.setData([]);
        return;
    };
    var items = e.rssDetails.items;
    var itemCount = items.length;
    var rows = [];
```

2. Next, all of the RSS items are looped through and a `Ti.UI.TableViewRow` definition is built for each item:

```
    for (var iLoop=0;iLoop<itemCount ;iLoop++){
        rows.push({
            title: items[iLoop].title,
            article : items[iLoop],
            height: 60,
            hasChild:true,
            color:'#000'
        });
    }
```

3. Finally, the article rows built in the previous step are added to the `Ti.TableView` for display:

```
        tableView.setData(rows);
    };
```

How it works...

The `rss2objects` module provides a straightforward API for reading RSS feeds. The two primary methods provided by the `rss2object.js` module are detailed in the following sections.

Using the yqlQuery function

The `yqlQuery` method uses Titanium's built-in YQL provider to parse the provided RSS feed URL into a dictionary of objects:

```
exports.yqlQuery = function(url, callback) {
```

1. The `onComplete` method is used as a callback to the `Ti.Yahoo.yql` function. This method will process the YQL results into a standard format for consumption.

```
function onComplete(e) {
```

2. The argument `e` is a dictionary (with the results of the RSS feed query) provided by YQL.

```
var results = {
    success: false,
    rssDetails : {
        url: url, full : null, items: null
    }
};
```

3. The `e.success` property is checked to determine if the YQL statement generated an error. On Android, you must check the `e.data` property for `null` instead of using the `e.success` property.

```
//Determine if we have an error
results.success = !((_isAndroid && e.data ==
    null) || (!_isAndroid && (e.data == null ||
    e.success == false)));
```

4. If successful, the `e.data.items` is added to the `results.rssDetails.items` array that will later be the callback method:

```
if(results.success) {
    results.rssDetails.items = [];
    var itemCount = e.data.item.length;
    for (var iLoop=0; iLoop<itemCount ; iLoop++) {
        results.rssDetails.items.push({
            //Add each field in our items
        });
    }
}
```

5. The results of our YQL-parsed RSS feed are provided to the initially used `callback` method, when calling the `yqlQuery` module method:

```
callback(results);
};
```

6. The following snippet demonstrates a YQL select query that will return a title, link, description, and other listed columns from the RSS feed (of the URL argument provided):

```
var yQuery = 'SELECT title, link, description, '
  yQuery += ' pubDate, guid, author, comments, '
  yQuery += ' enclosure, source FROM rss WHERE url = '
  yQuery += '"' + url + '"';
```

7. The next code block demonstrates using the `Ti.Yahoo.yql` method to run the query and send the results to the provided `onComplete` callback function:

```
Ti.Yahoo.yql(yQuery, onComplete);
};
```

Using the query function

The query method uses Titanium's `HTTPClient` and `XML` modules to read the provided RSS URL and to confirm the provided XML into a dictionary of objects:

```
exports.query = function(url, pathQuery, callback) {
  var workers = {
    results : {
      success: false,
      rssDetails : {
        url: url, full : null, items: null
      }
    }
  },
  ;
```

1. The `whenComplete` function is invoked when the `query` method has completed processing or resulted in an error. The `whenComplete` method is used to wrap the callback argument and provide the query results.

```
  whenComplete : function() {
    callback(workers.results);
  }
};
```

2. Action taken by this part of the recipe is to create an HTTP client:

```
var xhr = Ti.Network.createHTTPClient();
```

3. An `onload` callback is created to retrieve the HTTP response provided by the RSS feed:

```
xhr.onload = function(e) {
```

4. When the request is returned, the `responseXML` property is used to gather the XML results from the RSS feed:

```
var xml = this.responseXML;
```

5. The XML results are then converted to objects using our helper method:

```
var objResults = helpers.parseToObjects (xml);
```

6. If the converted XML object is `null`, use the callback method to notify the user that the module failed to read the provided RSS feed:

```
if(objResults ==null){  
    workers.whenComplete();  
    return;  
}
```

7. If the XML was successfully able to be converted into objects, populate the query results with the full XML results and create a flag indicating the query was successfully executed.

```
workers.results.rssDetails.full = objResults;  
workers.results.success = true;  
workers.results.rssDetails.items = objResults.item;
```

8. If a `pathQuery` string was provided, run the query and update the `results` object with the output from the `queryByPath` method.


```
if(pathQuery!=null){  
    workers.results.rssDetails.items =  
        helpers.queryByPath(wobjResults,pathQuery);  
}  
};
```

9. Provide an `onerror` callback to handle an error generated during the `HTTPClient` request.

```
xhr.onerror = function(e) {  
    Ti.API.info(JSON.stringify(e));  
    workers.whenComplete();  
};
```

10. Open the HTTP client and send a `GET` request to the RSS URL provided.

```
xhr.open('GET', url);  
xhr.send();  
};
```

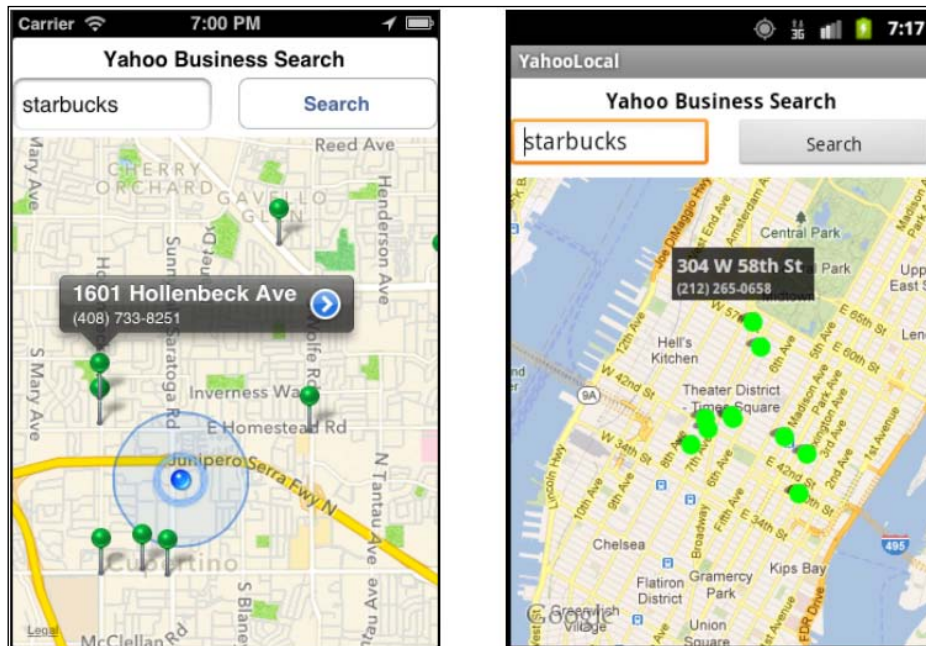
 The order in which the functions are called is important. The `open` method must be called before calling the `send` method, as shown in the previous code section.

See also

- ▶ To learn more about YQL, visit the Yahoo developer site at <http://developer.yahoo.com/yql>
- ▶ For YQL usage guidelines and rate information, visit http://developer.yahoo.com/yql/guide/usage_info_limits.html

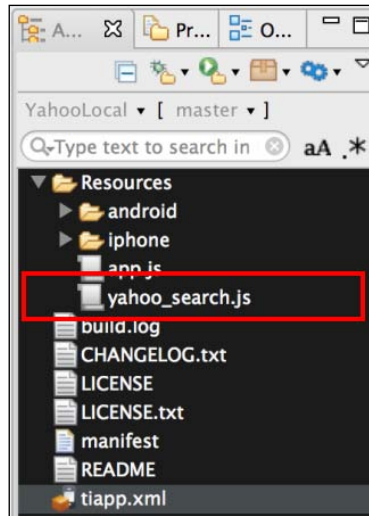
Creating a business location map using Yahoo Local

Providing a store, client, or other location-listing feature is a common requirement for many Enterprise apps. In this recipe, we will show how to use the Yahoo Local Search API and `Ti.Map.View` to provide a store locator. For demonstration purposes, the recipe uses the Yahoo API to provide location search results for the popular American coffee chain, Starbucks. The search results for each location are then displayed on a `Ti.Map.View`, as shown in the following screenshots:



Getting ready

This recipe uses the Yahoo Search CommonJS module. This module and other code assets can be downloaded from the source code provided by the book. Installing this module into your project is straightforward. Simply add the `yahoo_search.js` file into your project, as shown in the following screenshot:



How to do it...

Once you've added the `yahoo_search.js` file to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code, as shown in the following code block:

```
//Create our application namespace
var my = {
  search : require('yahoo_search'),
  isAndroid : ( Ti.Platform.osname === 'android')
};
```

Adding your API key

The next step in our recipe is to add your Yahoo developer API key. The following snippet shows how to register your API key with the module:

```
my.search.addAPIKey('YOUR_KEY_GOES_HERE');
```

Creating a UI for the sample app

This recipe uses a simple UI containing `Ti.UI.TextField`, `Ti.Map.View`, and a `Ti.UI.Button` to search. Here we demonstrate the creation of these UI elements.

1. First, we create a `Ti.UI.Window` to attach our visual elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff'
});
```

2. Next, we attach a text field to allow the user to enter a business for which to search.

```
var txtSubject = Ti.UI.createTextField({
  left:0, width:150, height:40,top:30,
  hintText:'Enter Business Name',
  borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtSubject);
```

3. Next, a button is added to allow the user to search for the business name entered in the text field.

```
var searchButton = Ti.UI.createButton({
  title:'Search', right:0, width:150, height:40,top:30
});
win.add(searchButton);
```

4. Finally, a map view is attached to display our search results.

```
var mapView = Ti.Map.createView({
  mapType: Ti.Map.STANDARD_TYPE, width:Ti.UI.FILL,
  userLocation:true, top:75,bottom:0
});
win.add(mapView);
```

Updating the map

The `updateMap` function is the callback method to the search module. `updateMap` provides the search results and then it transforms them to display to the user.

```
function updateMap(e) {
```

1. The search results are provided as the `e` parameter to the method. The first step in the process is to check if the search was successful, by checking the `e.success` property.

```
  if(!e.success) {
    alert("Sorry we encountered an error");
    return;
  }
```

2. After verifying that the search was successful, a loop is used to create map annotations for each `e.item` provided in the results.

```
var points = [], itemCount = e.items.length,
    rightBtn = Ti.UI.iPhone.SystemButton.DISCLOSURE;
for (var iLoop=0;iLoop<itemCount;iLoop++){
    points.push(Ti.Map.createAnnotation({
        latitude : e.items[iLoop].Latitude,
        longitude : e.items[iLoop].Longitude,
        title : e.items[iLoop].Address,
        subtitle : e.items[iLoop].Phone,
        pinColor : Ti.Map.ANNOTATION_GREEN,
        ClickUrl : items[iLoop].ClickUrl,
        animate:true, rightButton: rightBtn
    }));
}

var currentRegion = { latitude:e.latitude,
    longitude:e.longitude, latitudeDelta:0.04,
    longitudeDelta:0.04};
```

3. The new region object is created using the search coordinates and applied to set the viewing point of the map. This allows the user to see all of the pins added.

```
mapView.setLocation(currentRegion);
mapView.regionFit = true;
```

4. Finally, the points array containing all of our annotations is added to the `mapView`.

```
mapView.annotations = points;
};
```

Searching

When the `searchButton` button is pressed, the following snippet is used to perform a location search using the device's coordinates.

```
searchButton.addEventListener('click',function(e){
```

1. First, any existing map pins are removed
`mapView.removeAllAnnotations();`
2. If the text field's keyboard is open, the `blur` method is called to close it.
`txtSubject.blur();`
3. To avoid an empty search, the text field is checked to make sure it contains a value.

```
if((txtSubject.value+'').length ===0){
    alert('Enter a business to search');
```

```

    return;
}

```

- The module's `currentLocationQuery` method is then called, providing the business name entered in the text field and the `updateMap` function to be used as a callback.

```

my.search.currentLocationQuery(
  txtSubject.value, updateMap);
});

```

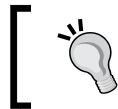
How it works...

The Yahoo Search CommonJS module (`yahoo_search.js`) provides the following public functions, detailed in the following sections.

Using `addAPIKey` to your Yahoo service key

The Yahoo Local Search API requires a developer key. Before using any of the query methods, you must first use `addAPIKey` to associate your developer key with the module.

```
my.search.addAPIKey('YOUR_KEY_GOES_HERE');
```



You can obtain a Yahoo API key by visiting `developer.yahoo.com` and creating a project within their developer portal.

Using the `geoQuery` method

The `geoQuery` function performs a Yahoo Local search using the latitude, longitude, and topic provided. The next example demonstrates how to search for Starbucks locations near Times Square in New York City. When the search has been completed, the results are provided to the `callback` function.

```
my.search.geoQuery(40.75773, -73.985708, 'starbucks', callback);
```

Using the `currentLocationQuery` method

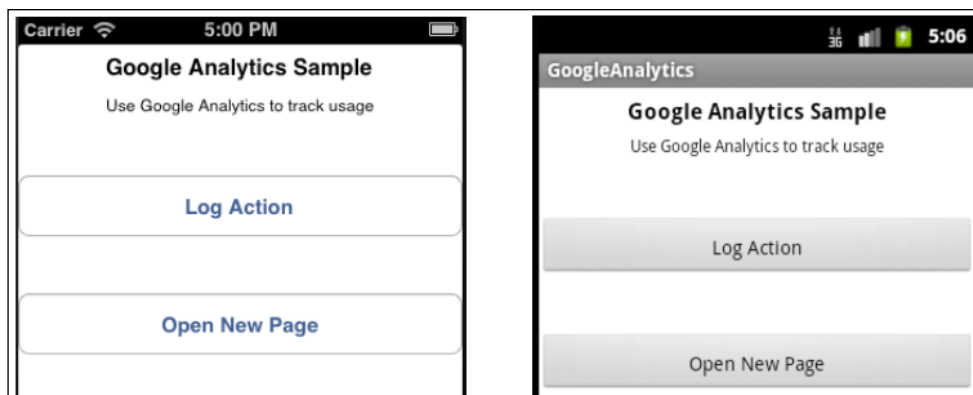
The `currentLocationQuery` method uses your device's location services to determine your current latitude and longitude. It then provides the `geoQuery` function with the required search details. The next code line demonstrates how to search for Starbucks outlets near your current position. Once the search has been completed, the results are provided to the `callback` function.

```
my.search.currentLocationQuery('starbucks', callback);
```

Using Google Analytics in your app

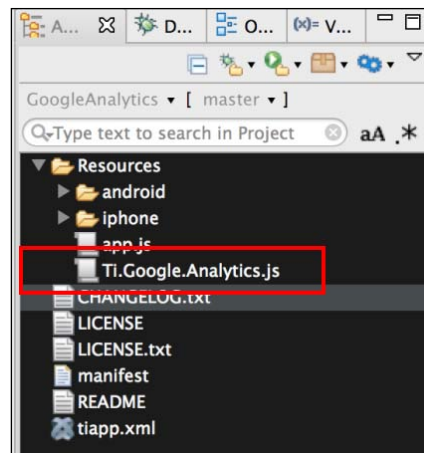
Google Analytics is a popular service used to measure and record website traffic and activities. Most likely, your organization uses Google Analytics, or a similar service, to gather analytics about visitors to your site. Using a Titanium module, you can use the same Google Analytics platform within your app. This approach allows you to view both your mobile and web traffic in one easy-to-use dashboard.

In this recipe, we will demonstrate how to submit both "page view" and "action" events to your Google Analytics dashboard.



Getting ready

This recipe uses the `Ti.Google.Analytics` CommonJS module. This module and other code assets can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing this module into your project is straightforward. Simply copy the `Ti.Google.Analytics.js` file into your project, as shown in the following screenshot:



How to do it...

Once you've added the `Ti.Google.Analytics.js` file to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code, as shown in the following code block:

```
//Create our application namespace
var my = {
  gAnalyticsModule : require('Ti.Google.Analytics'),
  isAndroid : (Ti.Platform.osname === 'android'),
  session: {}
};
```

Creating an instance of the module

The next step in the recipe requires you to have a Google Analytics key. To obtain a key, please register your app at www.google.com/analytics. Once you have a key, you will need to create a new instance of the `Analytics` module and provide your analytics key, as shown here:

```
my.session.ga = new my.gaModule('YOUR_KEY_HERE');
```

Helper functions

Helper functions help to provide more meaningful information about the user's device. These functions are used throughout the recipe whenever a `PageView` action is fired.

1. The following snippet demonstrates how the `isTablet` property is attached to the application namespace on starting an app. This property is used by other functions to indicate if the app is running on a phone or tablet.

```
my.isTablet = (function() {
  var tabletCheck = Ti.Platform.osname === 'ipad' ||
    (Ti.Platform.osname === 'android' &&
      !(Math.min(
        Ti.Platform.displayCaps.platformHeight,
        Ti.Platform.displayCaps.platformWidth
      ) < 700));

  return tabletCheck;
})();
```

2. The `basePage` property works in a way similar to website routing, and sets the first part of the URL (that is sent to Google) as a device indicator. In Google Analytics, this will allow you to better segment usage patterns by device.

```
my.basePage = (function(){
  if(my.isAndroid){
    return ((my.isTablet)?
      'AndroidTablet' : 'AndroidPhone');
  }else{
    //Return iPhone or iPad
    return Ti.Platform.model;
  }
})();
```

Start recording events

The next step is to call the `start` method. This will enable the module to start collecting analytics requests. The `start` method takes an integer value with the number of seconds on how often you wish the module to send queued analytics to Google. This is handled internally by the module using an interval timer.

```
my.session.ga.start(10);
```

Creating our sample UI

This section of the recipe outlines the sample UI used to trigger and submit Google Analytics requests.

1. First, a `Ti.UI.Window` is created to anchor all UI controls.

```
var win = Ti.UI.createWindow({
  backgroundColor:'#fff', layout:'vertical'
});
```

2. After creating our window, two `Ti.UI.Button` controls are added. These buttons will be used later in the recipe to demonstrate how to create a `trackEvent` or `Pageview` event.

```
var button1 = Ti.UI.createButton({
  title: 'Log Action', top:40, height:45,
  width:Ti.UI.FILL
});
win.add(button1);
```

```
var button2 = Ti.UI.createButton({
  title: 'Open New Page', top:40, height:45,
  width:Ti.UI.FILL
});
win.add(button2);
```

Recording an action

The `trackEvent` function allows you to publish granular event tracking to Google Analytics. This method requires the following parameters that will be used to publish actions to your Google Analytics dashboard:

- ▶ **Category:** Typically, the object that was interacted with (for example, a button)
- ▶ **Action:** The type of interaction (for example, a click)
- ▶ **Label:** Useful for categorizing events (for example, nav buttons)
- ▶ **Value:** Values must be non-negative. Useful to pass counts (for example, four times)

The next snippet demonstrates how to call the `trackEvent` method when `button1` is pressed. On firing of the button's click event, the `trackEvent` method is called with a category of `button`, an action of `click`, a label of `recipe_button`, and a value of `trigger_event`.

```
button1.addEventListener('click', function(e) {
  my.session.ga.trackEvent('button', 'click', 'win_button',
    'trigger_event');
});
```


The Pageview function on opening a window

The `trackPageview` function is used to mimic the page traffic or views displayed in your Google Analytics dashboard. Using the `basePage` properties created earlier, you can create device-specific window tracking by using the convention shown in the following code block:

```
win.addEventListener('open', function(e) {
  my.session.ga.trackPageview(my.basePage + "/Main");
});
win.open();
```

The Pageview function on a child window

The following section of the recipe demonstrates how to use the `trackPageview` and `trackEvent` methods to record when and how a child window or view is opened. When the user presses `button2` and the click event is fired, the Google Analytics module is used to log each step of the navigation process.

```
button2.addEventListener('click', function(e) {
```

1. Use the `trackEvent` method to record that the button to open a child window has been pressed:

```
  my.session.ga.trackEvent('button', 'click',
    'win_button', 'open_win2');
```

You need to provide the following parameters to the `trackEvent` method:

- **Category:** A category value of `button` is provided in this sample
 - **Action:** An action value of `click` is provided in this sample
 - **Label:** A label value of `win_button` is provided in this sample
 - **Value:** A value of `open_win2` is provided in this sample
2. Create a new window to demonstrate the `trackPageview` functionality on a child window.

```
  var win2 = Ti.UI.createWindow({
    backgroundColor:'yellow', layout:'vertical'
  });
```

3. On loading of the `win2` window, the `trackPageview` method is called, recording that the individual has viewed the page. The `my.basePage` is used to create a route showing which type of device accessed the `/win2` window.

```
  win2.addEventListener('open', function(e) {
    my.session.ga.trackPageview(my.basePage + "/win2");
  });
  win2.open({modal:true});
});
```

See also

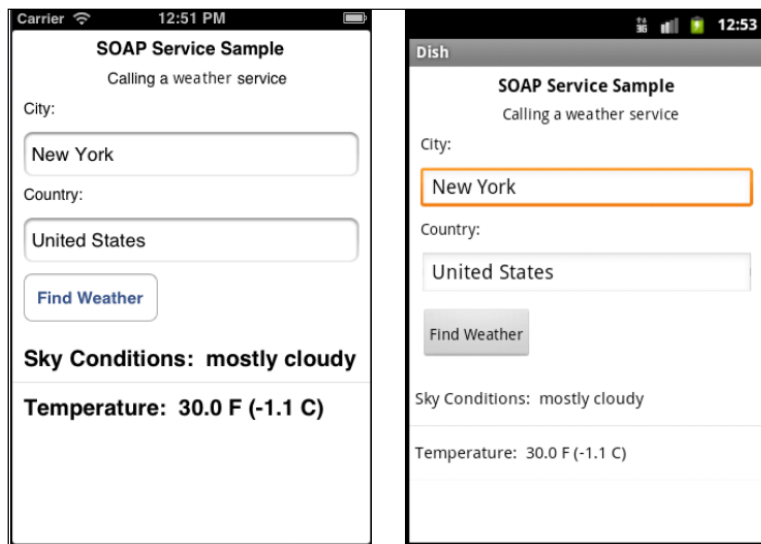
- ▶ This recipe uses a modified version of Roger Chapman's Titanium Mobile Google Analytics module. To learn more about this module, please visit <http://github.com/rogchap/Titanium-Google-Analytics>.
- ▶ For more information on detecting different device characteristics, see the *Using platform indicators* recipe in *Chapter 1, Patterns and Platform Tools*.

Making SOAP service calls using SUDS.js

In many Enterprise market segments, SOAP services remain the dominant web service protocol. Since SOAP is generally implemented over HTTP, most network clients including Titanium's `Ti.Network` can interact effectively with this protocol.

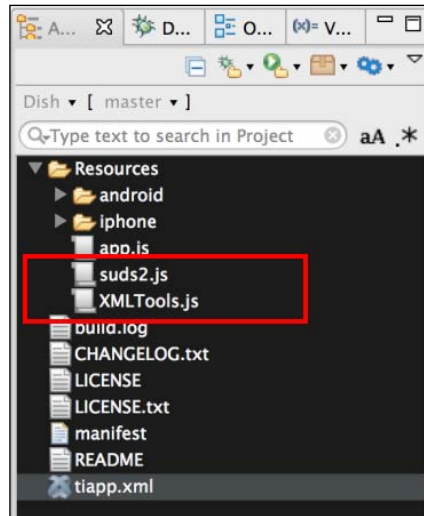
Even with Titanium's `Ti.Network` module working with SOAP, envelopes and XML results can be challenging, and often requires creating a SOAP envelope and a huge amount of XML manipulation. This recipe demonstrates how a few open-source modules can increase your productivity when interacting with SOAP services and their XML results.

To help illustrate how to interact with SOAP services, the recipe uses the `www.webserviceX.NET` weather SOAP service to return weather results for a city entered in the **City:** field, as shown in the following screenshots:



Getting ready

This recipe uses the `SUDS` and `XMLTools` CommonJS modules. These modules and other code assets can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing these modules is straightforward and simply requires copying the `suds2.js` and `XMLTools.js` files into your Titanium project, as highlighted in the following screenshot:



How to do it...

Once you've added the `suds2.js` and `XMLTools.js` modules to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code, as shown in the following snippet:

```
//Create our application namespace
var my = {
  suds : require('suds2'),
  xmlTools : require("XMLTools"),
  isAndroid : (Ti.Platform.osname === 'android')
};
```

SOAP helper methods

In addition to the two CommonJS modules imported earlier, this recipe uses the `soapHelper` object to handle formatting and configuration activities.

```
var soapHelper = {
```

1. The configuration object contains all the configuration details that `suds.js` needs to return an XML result:

```
config : {
  endpoint: "http://www.webserviceX.com/globalweather.asmx";
  targetNamespace: 'http://www.webserviceX.NET/',
  includeNS : false,
  addTargetSchema : true
},
```

2. The `resultParser` object is used to format the returned XML result into JavaScript objects:

```
resultParser : {
```

3. The `removeHeader` object is used to remove the XML header node. Android requires the header to be removed before the `parseString` function will correctly create an XML document object.

```
removeHeader : function(text){
  return text.replace(
    '<?xml version="1.0" encoding="utf-16"?>', '');
},
```

4. The `xmlToObject` function converts a `Ti.XML.Document` object into JavaScript objects.

```
xmlToObject : function(doc){
```

5. The first step is to get a `Ti.XML.Nodelist` for the tag `GetWeatherResponse`.

```
var results = doc.documentElement.getElementsByTagName(
  'GetWeatherResponse');
```

6. Android and iOS handle the conversion process differently. Use the `my.isAndroid` property to branch the conversion logic.

```
if(my.isAndroid){
```

7. The weather service result contains a nested XML document. The following example demonstrates how to read the embedded XML document from the `GetWeatherResponse` node into a new `Ti.XML.Document`. The `removeHeader` function is used to fix the `textContent` value, to be compliant with Android's XML Document format.

```
var docCurrent =Ti.XML.parseString(
  soapHelper.resultParser.removeHeader(
    results.item(0).textContent));
```

- Next, the `Ti.XML.Document` object is provided to the `XMLTools` module's constructor and then converted into JavaScript objects using the `toObject` method, as demonstrated in the following snippet:

```
return new my.xmlTools(docCurrent).toObject();
}else{
```

- On iOS, we use the `getChildNodes` function to obtain the weather child node:

```
var weather =results.item(0).getChildNodes()
.item(0).getChildNodes();
```

- The XML string from the weather node is then loaded into the `XMLTools` module's constructor, and then converted into JavaScript objects using the `toObject` method, as demonstrated in the following code block:

```
var docCurrentFromString = Ti.XML.parseString(
    soapHelper.resultParser.removeHeader
    (weather.item(0).textContent));
return new
    my.xmlTools(docCurrentFromString).toObject();
}
}
}
};
```

Creating the UI

This section of the recipe outlines the sample UI used to call and display results from the weather SOAP service.

- A new `Ti.UI.Window` is created for all UI elements to be attached.

```
var win = Ti.UI.createWindow({
    backgroundColor:'#fff', layout:'vertical'
});
```

- The text field `txtCity` is to allow the user to enter the city whose weather they wish to be displayed.

```
var txtCity = Ti.UI.createTextField({
    value:'New York', hintText:'Enter city',
    top:10, height:40, left:10,
    right:10, textAlign:'left',
    borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtCity);
```

3. The text field `txtCountry` is to allow the user to input the name of the country to which the city belongs.

```
var txtCountry = Ti.UI.createTextField({
    value:'United States', hintText:'Enter country',
    top:10, height:40, left:10, right:10,
    textAlign:'left', borderStyle:
    Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtCountry);
```

4. The `goButton` is a `Ti.UI.Button` used to call the weather SOAP service.

```
var goButton = Ti.UI.createButton({
    title:'Find Weather', left:10, top:10
});
win.add(goButton);
```

5. The `tableView` is a `Ti.UI.TableView` used to display the results of the SOAP service.

```
var tableView = Ti.UI.createTableView({
    visible : false, top:10, height:100, width:Ti.UI.FILL
});
win.add(tableView);
```

The `uiHelpers` object

The `uiHelpers` object is used to update the UI objects with different states of the app as well as load the `tableView` with the SOAP service result.

```
var uiHelpers = {
```

The `updateUI` is used to format the object results from the SOAP service for display.

```
updateUI : function(weather){
    var data = [];
    tableView.visible = true;
    data.push({title: "Sky Conditions: " +
        weather.SkyConditions, color:'#000'});
    data.push({title: "Temperature: " +
        weather.Temperature, color:'#000'});
    data.push({title: "Time: " + weather.Time});
    tableView.setData(data);
};
```

The `resetUI` method is used to hide the `tableView` from the user when SUDS is calling the web service. This method is also used to hide the `tableView` when the a SUDS call results in an error.

```
resetUI :function(){
    tableView.visible = false;
};
};
```

Calling the SOAP service

The click event on the `goButton` is used to perform the weather SOAP service call.

```
goButton.addEventListener('click', function(e) {
```

1. The `resetUI` method is first called to hide the `tableView` while the service is being called.

```
    uiHelpers.resetUI();
```

2. A new instance of `sudsClient` is created with the configuration information defined earlier in the `soapHelper.config` object.

```
    var sudsClient = new my.suds(soapHelper.config);
```

3. The `invoke` method is then called on `sudsClient`. The first argument provided is the SOAP method that `suds` should call.

```
    sudsClient.invoke('GetWeather',
```

4. The second argument provided to `sudsClient` is the name of the city and country that the user has requested, to retrieve weather information.

```
    {
        CityName: txtCity.value,
        CountryName: txtCountry.value
    },
```

5. The final argument of the `invoke` method is the callback method SUDS. This callback method will be provided will to provide a `Ti.XML.Document` with the service's results. The following example demonstrates using an inline function as a callback method:

```
    function(xml) {
```

6. The inline callback method will receive a `Ti.XML.Document` once the service has completed. Once received, the result is parsed into JavaScript objects using the `resultParser` object, as detailed earlier in the recipe.

```
        var weather = soapHelper.resultParser.xml2Object(xml);
```

- The `Status` property is changed on the parsed object to determine if the weather objects have successfully been created.

```
if (weather.Status === 'Success') {
```

- If the service results have successfully been converted into objects, they are provided to the `updateUI` method, to be displayed to the user.

```
    uiHelpers.updateUI(weather);  
  } else {
```

- If an error occurred in calling the service or processing the results, we alert the user and then hide the `tableView` display object.

```
    alert("Sorry a problem happened");  
    uiHelpers.resetUI();  
  }  
});  
});
```

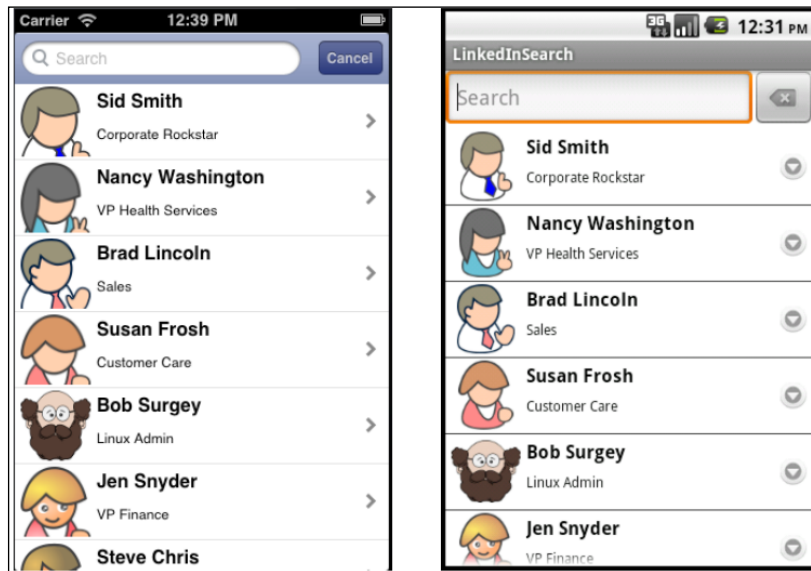
See also

- ▶ Please review the following links to learn more about the open source projects used in this recipe:
 - **SUDS.js**: For additional information on SUDS.js, please visit <http://github.com/kwhinnery/Suds>.
 - **SUDS2.js**: For more information about the version of SUDS.js used in this recipe, please visit <http://github.com/benbahrenburg/Suds2>.
 - **XMLTools**: This recipe uses the XMLTools module created by David Bankier. For more information about this module, please visit <http://github.com/dbankier/XMLTools-For-Appcelerator-Titanium>.

Using the LinkedIn Contacts API

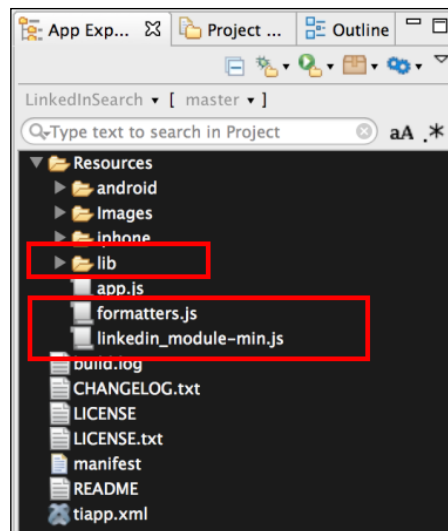
LinkedIn is a popular social-networking site for professionals. The LinkedIn API provides a rich set of integration services for use within your app. For Enterprise-focused apps, LinkedIn features such as Messaging and Contacts can be critical. Common examples of this would be providing a sales agent with access to their contacts within your app.

This recipe demonstrates how to integrate the LinkedIn Contacts API into your Titanium app, in a searchable fashion.



Getting ready

This recipe uses several modules, including the innovative LinkedIn module, `linkedin_module-min.js`. This module and other code assets can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Setting up the dependencies for this recipe is straightforward. First, copy the `lib` folder to the `Resources` folder of your Titanium project, and then, the `copy formatters.js` and `linkedin_module-min.js` files into the `Resources` folder, as shown highlighted in the following screenshot:



How to do it...

After you have set up the recipe's dependencies, the next step is to create your application namespaces in the `app.js` file and use `require` to import the module into your code, as shown in the following snippet:

```
//Create our application namespace
var my = {
  isAndroid : (Ti.Platform.osname === 'android'),
  linkedMod: require('linkedin_module-min'),
  formatters : require('formatters')
};
```

Adding your API key and secret key

The first step in using the LinkedIn module is to create a LinkedIn application at <https://www.linkedin.com/secure/developer>. Once you have registered your application, LinkedIn will provide you with the API and authentication keys needed to interact with their APIs. All of the APIs used in this recipe need OAuth 1.0a authentication to connect. The LinkedIn module will handle this for you by using the `init` method to register your secret and API keys, as shown in the following example:

```
my.linkedMod.init('YOUR_SECRET_KEY', 'YOUR_API_KEY');
```



You must use the `init` method to set your secret and API keys, before using any of the module's functionality.



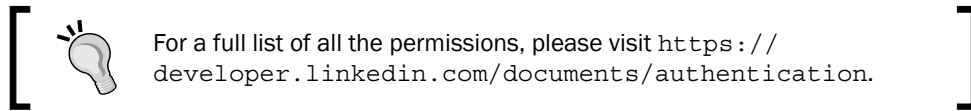
Adding permissions

By default, if no permissions are specified, your app will only have rights to read basic "digital business card" information about the current user. Since this recipe needs access to the user's contacts, we must call the `addPermission` method to request for the `r_network` privilege, as shown in the following snippet:

```
my.linkedMod.addPermission('r_network');
```

To add multiple permissions, simply call the `addPermission` method several times. The following snippet shows how to add the full profile access right to the app:

```
my.linkedMod.addPermission('r_fullprofile');
```



Creating the UI

This section of the recipe outlines how to create the UI that is used to display and search LinkedIn contacts.

1. First, a `Ti.UI.Window` is created. This will be used to attach all of our visual elements and will also trigger a call to the LinkedIn API when the open event is fired.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff'
});
```

2. A `Ti.UI.SearchBar` is then created. The search-bar control provides a search box that filters the contents of `Ti.UI.TableView`. In this recipe, we use the search-bar control to filter the user's LinkedIn contacts by last name.

```
var search = Ti.UI.createSearchBar({
  barColor: '#385292', showCancel: true, hintText: 'Search'
});
```

3. The final UI component added to `Ti.UI.Window` is `Ti.UI.TableView` that will be used to display the user's LinkedIn contacts.

```
var tableView = Ti.UI.createTableView({
  height: Ti.UI.FILL, width: Ti.UI.FILL,
  search: search, filterAttribute: 'lastName'
});
win.add(tableView);
```

Loading your LinkedIn contacts

The next snippet demonstrates how the LinkedIn module can be used to load a list of your contacts as the `Ti.UI.Window` window is loaded.

```
win.addEventListener('open', function(e) {
```

1. Use the `uiHelper.createWaitMsg` method to display a loading message to the user.

```
    uiHelpers.createWaitMsg('Loading please wait...');
```

2. Use the module's `getConnections` method to query the user's contacts from the LinkedIn API. In the next example, an inline function is used as a callback returning the user's contacts as the `_e` variable.

```
    my.linkedMod.getConnections(function(_e) {
```

3. The `uiHelpers.displayContacts` method is used to format and apply returned contacts for display.

```
        uiHelpers.displayContacts(_e);
    });
});
```

```
win.open();
```

Using the uiHelpers object to format results

The `uiHelpers` object is used to format the results of the LinkedIn Contacts API for display.

```
var uiHelpers = {
```

1. The `createWaitMsg` function is used to display a waiting message in the `tableView` while the LinkedIn API is being called.

```
    createWaitMsg : function(msg) {
        tableView.setData([{title:msg}]);
    },
```

2. The `displayContacts` method is the primary method used to convert and display the API results.

```
    displayContacts : function(apiResults) {
```

3. Update the `tableView` to alert the user that we are now loading their contacts.

```
        uiHelpers.createWaitMsg('Loading contacts
        please wait...');
```

4. The `convertToObject` method is called to convert the LinkedIn XML results into more manageable JavaScript objects.

```
var resultAsObjects = my.resultParser
    .convertToObject(apiResults);
```

5. If the conversion result is null, display an error message to the user.

```
if(resultAsObjects == null){
    alert('Sorry we ran into an error');
    return;
}
```

6. Using the `formatter.createContactTableRows` function, format the JavaScript objects into the `Ti.UI.TableViewRow` layout shown earlier in this recipe's screenshots. The `tableView` is then updated with the formatted rows.

```
tableView.setData(
    my.formatters.createContactTableRows(
        resultAsObjects)
    );
};
```

Parsing the LinkedIn API results

The `resultParser` object is used to parse the XML provided by the LinkedIn API, into more manageable JavaScript objects.

```
my.resultParser = {
```

1. The `getText` function is used to return a specific key from a provided `Ti.XML.Element`. If no key is found, a null value is returned.

```
getText : function(item, key) {
    var searchItem =
        item.getElementsByTagName(key).item(0);
    return ((searchItem == null)? null :
        searchItem.textContent);
},
```

2. The `getQueryParams` function is used to return an object with all query string parameters for a provided URL.

```
getQueryParams : function(url){
    var params = {};
    url = url.substring(url.indexOf('?')+1).split('&');
    var pair, d = decodeURIComponent;
    for (var i = url.length - 1; i >= 0; i--) {
        pair = url[i].split('=');
        params[d(pair[0])] = d(pair[1]);
    }
}
```

```

    }
    return params;
  },

```

3. The `formatUrl` function returns a URL to the contact's profile. If no URL can be determined, a link to `linkedin.com` is provided.

```

formatUrl : function(findKey) {
  return ((findKey.hasOwnProperty("key")) ?
    ("http://www.linkedin.com/profile/view?id=" +
    findKey.key) : "http://www.linkedin.com");
},

```

4. The `getProfileUrl` function is used to return a URL to the contact's profile. Since the LinkedIn API does not provide this information, a URL is generated by parsing the `site-standard-profile-request` node for key details. The construction of this URL is demonstrated in the following snippet:

```

getProfileUrl : function(item) {
  var searchItem = item.
    getElementsByTagName("site-standard-profile-
    request").item(0);
  if(searchItem == null){
    return null;
  }
  if(searchItem.hasChildNodes()){
    var findKey = my.resultParser.getQueryParams(
      searchItem.getElementsByTagName('url')
      .item(0).textContent);
    return my.resultParser.formatUrl(findKey);
  }else{
    return null;
  }
},

```

5. The `isPublic` function is used to determine if a contact's information is public. If the profile is not public, we will not add it to the displayed contact list.

```

isPublic : function(item) {
  return !((my.resultParser.getProfileUrl(item)==null));
},

```

6. The `convertToObjects` is the primary method responsible for converting the LinkedIn contacts' XML into JavaScript objects.

```

convertToObjects : function(xmlString) {
  var contacts = [];

```

7. First, the string of XML provided by the API is loaded into a `Ti.XML.Document`.

```
var doc = Ti.XML.parseString(xmlString);
```

8. Retrieve all of the XML nodes under the `person` tag.

```
var items = doc.documentElement.  
  getElementsByTagName("person");
```

9. Start looping through each of the nodes under the `person` tag.

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

10. Check that the profile is public.

```
  if(my.resultParser.isPublic(items.item(i))){
```

11. If the profile is public, create a JavaScript object with properties from the `Ti.XML.Node`.

```
    contacts.push({  
      id : my.resultParser.getText  
        (items.item(i), 'id'), headline :  
        my.resultParser.getText(items.item(i),  
        'headline'),  
      firstName : my.resultParser.  
        getText(items.item(i), 'first-name'),  
      lastName : my.resultParser  
        .getText(items.item(i), 'last-name'),  
      pictureUrl : my.resultParser  
        .getText(items.item(i), 'picture-url'),  
      profileUrl : my.resultParser  
        .getProfileUrl(items.item(i))  
    });  
  }  
}
```

12. For individuals with a large number of contacts, this conversion process can be memory intensive. To help reduce the size of variables in memory, we set all temporary objects to null before returning the converted results.

```
  doc = null, xmlString = null;  
  return contacts;  
}  
};
```

See also

- ▶ This recipe uses the `clearlyinnovative.linkedin` module created by Aaron Saunders, of `clearlyinnovative.com`. For additional documentation, samples, and guidance with the module, please visit <http://www.clearlyinnovative.com/blog/post/12521419647/titanium-appcelerator-quickie-linkedin-api-integration>. To download the source of the module, please visit the project on Github, at <http://github.com/aaronksaunders/clearlyinnovative.linkedin>.

5

Charting and Documents

In this chapter, we will cover:

- ▶ Opening PDF documents
- ▶ Using an iPad for document signatures
- ▶ Creating PDF documents from images or screenshots
- ▶ Generating PDF documents using jsPDF
- ▶ Creating a scheduling chart using RGraph
- ▶ Displaying information with Google gauges

Introduction

Through the convenience of a mobile device, your customers and employees have the ability to access data, documents, and related information, wherever and whenever needed. This chapter provides recipes to assist with the development of at-a-glance and portable information.

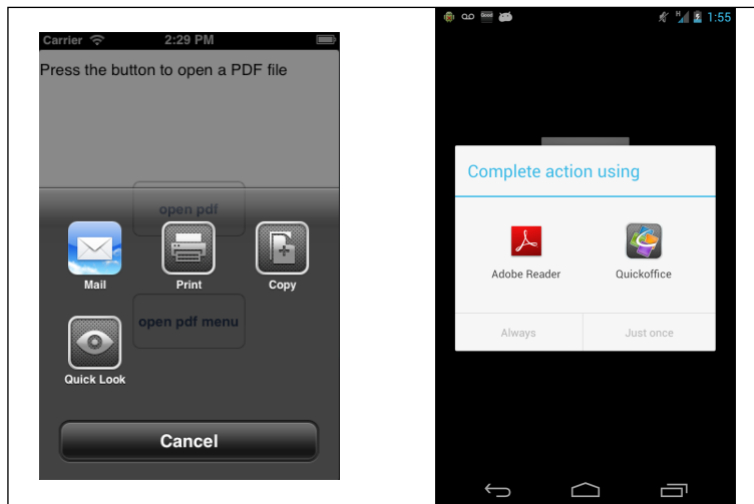
Depending on your industry, an innovative mobile-document strategy can provide a key competitive advantage. Forms, contracts, recipes, and invoices are all examples of areas a mobile document strategy can reduce costs while better engaging your users. In this chapter, we will cover several strategies to deal with documents in a cross-platform fashion.

The creation of dashboards using charts and graphs to paint a compelling picture is a common management-team request, and is a powerful way to harness mobile technology to provide business value. An example of this value proposition is mobility, enabling your marketing, sales, or operations teams with current market data in an at-a-glance format, allowing the management team to make faster decisions. In this chapter, we demonstrate how to use Titanium to create powerful, interactive charting experiences.

Opening PDF documents

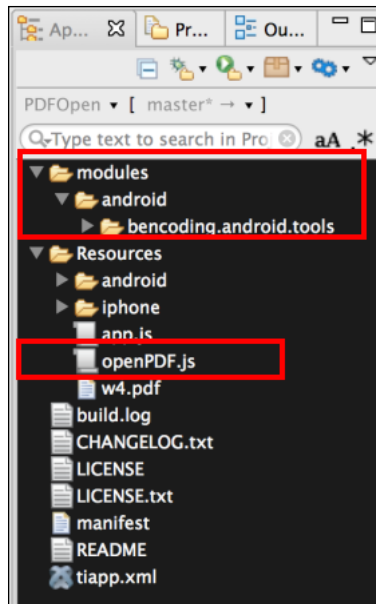
The ability to open, view, and exchange PDF documents is a common requirement in Enterprise app development. This recipe discusses how to use the `openPDF` `CommonJS` module to view PDF documents in a cross-platform manner.

The following screenshots illustrate this recipe running on both an iPhone and an Android device:

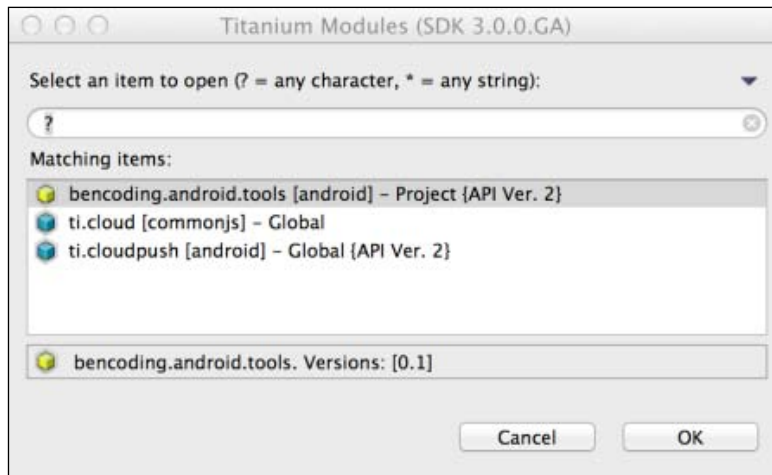


Getting ready

The recipe uses both a `CommonJS` and an Android native module. These can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing these in your project is straightforward. Simply copy the `openPDF.js` file and the `encoding.android.tools` folder into your Titanium project, as shown highlighted in the following screenshot:



After copying the module folder, you will need to click on your `tiapp.xml` file in Titanium Studio and add a reference to the `bencoding.android.tools` module, as shown in the following screenshot:



How to do it...

Once you have added `openPDF.js` and the native android module to your project, you need to create your application namespaces and use `require` to import the module into your code, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  pdfOpener : require('openPDF'),
  isAndroid : (Ti.Platform.osname === 'android')
};
```

Creating a link to the PDF file

The next step in this recipe is to create a variable with the path to our PDF file. For example purposes, we are using a PDF file called `w4.pdf` located in our project's `resources` directory. The following snippet demonstrates how to create this path variable:

```
my.pdfFile = Ti.Filesystem.resourcesDirectory + 'w4.pdf';
```

Creating our example UI

This recipe provides a basic UI to help illustrate how to use the `openPDF` module. This straightforward screen contains a button that opens a dialog providing different options the user can select to open a PDF file. If this recipe is running on an iOS device, the user will have a second button demonstrating how to open the PDF file inside the app.

1. We first create a `Ti.UI.Window` window to attach all UI elements onto:

```
var win = Ti.UI.createWindow({
  exitOnClose: true, title:"Open PDF Example",
  backgroundColor:'#fff'
});
```

2. Next, we add a label instructing the user which button to press:

```
var infoLabel = Ti.UI.createLabel({
  text:'Press the button to open a PDF file',
  top:10, left:10,
  right:10, height:40, textAlign:'center'
});
win.add(infoLabel);
```

3. Next, the button that will launch our first example is added to the `Ti.UI.Window` window:

```
var goButton = Ti.UI.createButton({
    title: 'open pdf', height: '60dp',
    width: '150dp', top: '140dp'
});
win.add(goButton);
```

Opening a PDF file on the click of a button

The first example of how to use the `openPDF` module is shown in the following `goButton` click-event snippet:

```
goButton.addEventListener('click', function(e) {
```

1. First, the `isSupported` method is called to determine if the device supports PDF files. This method will return a `true` if supported, or a `false` if no support is available.

```
    if(my.pdfOpener.isSupported()){
```

2. Next, the file path to our PDF is provided to the `open` method. This will then present the user with an `options` dialog on how they wish to view the file:

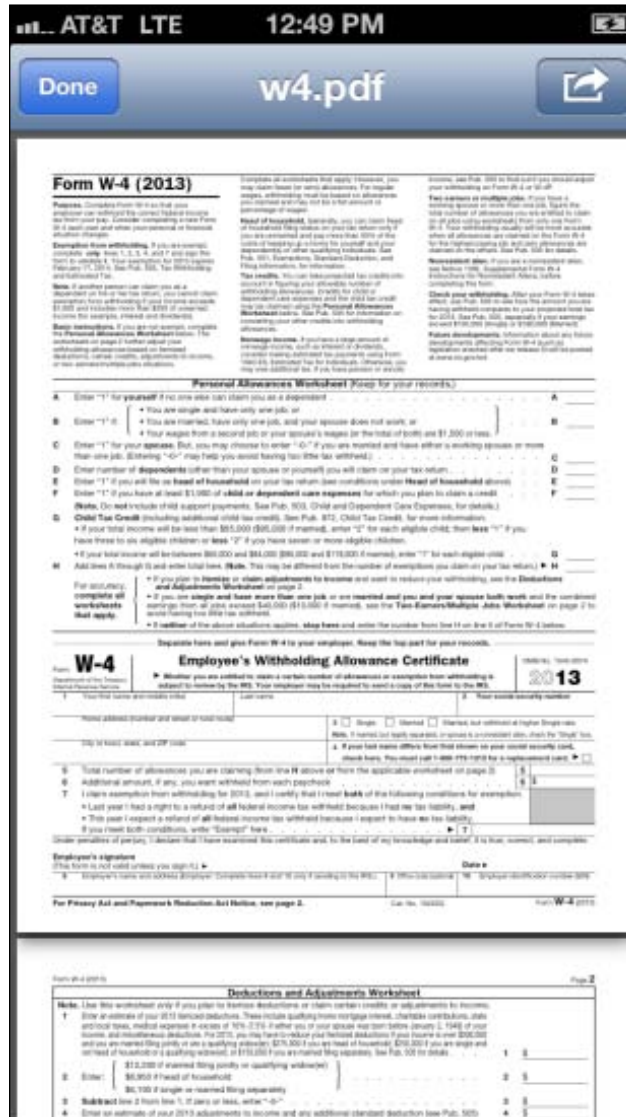
```
        my.pdfOpener.open(my.pdfFile);
    }else{
```

3. If the `isSupported` method results a `false`, you can then call the `isSupportedDetail` method for more information on why. As the following snippet shows, you can use the `reason` property returned by the `isSupportedDetail` method to alert your user to the specific issue. You can also use the `code` property to create your own customized messages.

```
        alert(my.pdfOpener.isSupportedDetail().reason);
    }
});
```

Opening a PDF file within your iOS app

The iOS platform has built-in support for working with PDF files. The `openPDF` module leverages this platform to provide the ability to create a PDF dialog viewer within your app, as shown in the following screenshot:



1. In this section, a check is performed to ensure that the recipe is running on an iOS device. If so, the button `goButton2` is added to our `Ti.UI.Window` window:

```
if(!my.isAndroid){
    var goButton2 = Ti.UI.createButton({
        title: 'open pdf menu', height: 60,
        width: 150d, bottom: 140
    });
    win.add(goButton2);
```

2. The `goButton2` click event demonstrates how to open a PDF file in a dialog window:

```
goButton2.addEventListener('click', function(e) {
```

3. Next, the `isSupported` method is called to verify whether the device can read a PDF file:

```
    if(!my.pdfOpener.isSupported()){
```

4. If the device does not support the displaying of PDF files, the `isSupportDetail` method is called to return the reason why PDF files cannot be displayed:

```
        alert(my.pdfOpener.isSupportedDetail().reason);
        return;
    }
```

5. Finally, the `openDialog` method is called to display the PDF file. The `openDialog` method requires two arguments. The first is a path to the PDF file that is to be displayed. The second is a configuration object containing a UI object that specifies the view relative to where the viewer should be displayed. In the following example, we provide the `infoLabel` label created earlier as our `view` object:

```
    my.pdfOpener.openDialog(my.pdfFile,
        {view:infoLabel, animated:true}
    );
    });
}
```



Additional elements can be added to the configuration object, such as the `animated` property to determine if an animation should be applied while opening and closing a PDF file.

To close the PDF dialog programmatically, use the `closeDialog` method, as shown in the following snippet:

```
my.pdfOpener.closeDialog();
```

File clean-up on closing a window

The `openPDF` module creates cache objects and temporary files to assist in the display process. It is important to call the `dispose` method when PDF operations are no longer needed:

```
win.addEventListener('close', function(e) {  
    my.pdfOpener.dispose();  
});
```



If you are using the `openDialog` option on iOS, calling `dispose` will close the PDF dialog as well.

See also

To learn more about the libraries and frameworks used in this recipe, please visit the following links:

- ▶ **iOS DocumentViewer:** The `openPDF` module uses the `Ti.UI.iOS.DocumentViewer` object to work with PDF files on the iOS platform. For additional details, please review Titanium's online documentation at <http://docs.appcelerator.com>.
- ▶ **Working with Android Intents:** The `openPDF` module uses the `Ti.Android.intent` object to launch a PDF viewer on the user's device. For additional information about working with `Ti.Android.intent`, visit this Appcelerator blog entry at <http://developer.appcelerator.com/blog/2011/09/sharing-project-assets-with-android-intents.html>.
- ▶ **Android Intent Supported:** The `openPDF` library uses the `bencoding.android.tools` module's project to check if there is an available intent to open a PDF document on user's device. To learn more about this project, please visit <https://github.com/benbahrenburg/benCoding.Android.Tools>.

Using an iPad for document signatures

Tablets and other touch devices provide a naturally immersive environment for working with documents. Through the use of several open source modules and our example PDF, this recipe illustrates how to create a document-signing app for your organization.

The following screenshot demonstrates this recipe running on an iPad:

The screenshot shows an iPad interface for a 'Signature Recipe'. At the top, it says 'iPad Signature Recipe' and 'This recipe demonstrates how to collect a signature for a document'. Below this is a section for 'Form W-4 (2013)'. The text includes instructions on how to complete the form, exemptions, and tax credits. A large box contains a handwritten signature 'John Smith'. Below the signature box are two buttons: 'Remove Signature' and 'Save Signature'.

Form W-4 (2013)

Purpose. Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay. Consider completing a new Form W-4 each year and when your personal or financial situation changes.

Exemption from withholding. If you are exempt, complete **only** lines 1, 2, 3, 4, and 7 and sign the form to validate it. Your exemption for 2013 expires February 17, 2014. See Pub. 505, Tax Withholding and Estimated Tax.

Note. If another person can claim you as a dependent on his or her tax return, you cannot claim exemption from withholding if your income exceeds \$1,000 and includes more than \$350 of unearned income (for example, interest and dividends).

Basic instructions. If you are not exempt, complete the **Personal Allowances Worksheet** below. The worksheets on page 2 further adjust your withholding allowances based on itemized deductions, certain credits, adjustments to income, or two-earners/multiple jobs situations.

Complete all worksheets that apply. However, you may claim fewer (or zero) allowances. For regular wages, withholding must be based on allowances you claimed and may not be a flat amount or percentage of wages.

Head of household. Generally, you can claim head of household filing status on your tax return only if you are unmarried and pay more than 50% of the costs of keeping up a home for yourself and your dependent(s) or other qualifying individuals. See Pub. 501, Exemptions, Standard Deduction, and Filing Information, for information.

Tax credits. You can take projected tax credits into account in figuring your allowable number of withholding allowances. Credits for child or dependent care expenses and the child tax credit may be claimed using the **Personal Allowances Worksheet** below. See Pub. 505 for information on converting your other credits into withholding allowances.

Nonwage income. If you have a large amount of nonwage income, such as interest or dividends, consider making estimated tax payments using Form 1040-ES, Estimated Tax for Individuals. Otherwise, you may owe additional tax. If you have pension or annuity income, see Pub. 505 to find out if you should adjust your withholding on Form W-4 or W-4P.

Two earners or multiple jobs. If you have a working spouse or more than one job, figure the total number of allowances you are entitled to claim on all jobs using worksheets from only one Form W-4. Your withholding usually will be most accurate when all allowances are claimed on the Form W-4 for the highest paying job and zero allowances are claimed on the others. See Pub. 505 for details.

Nonresident alien. If you are a nonresident alien, see Notice 1392, Supplemental Form W-4 Instructions for Nonresident Aliens, before completing this form.

Check your withholding. After your Form W-4 takes effect, use Pub. 505 to see how the amount you are having withheld compares to your projected total tax for 2013. See Pub. 505, especially if your earnings exceed \$130,000 (Single) or \$160,000 (Married).

Future developments. Information about any future developments affecting Form W-4 (such as legislation enacted after we release it) will be posted at www.irs.gov/w4.

Use your finger to add a signature in the box below

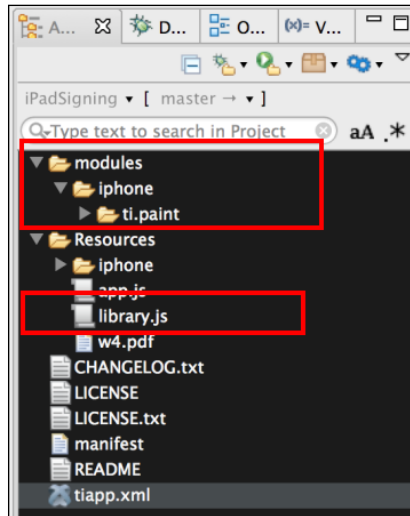
Remove Signature Save Signature



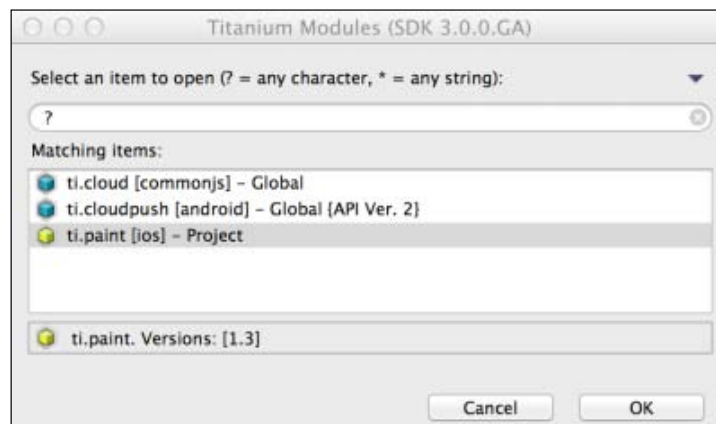
This recipe is designed to run only on an iPad.

Getting ready

This recipe uses both `CommonJS` and native modules. These modules can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. The installation process is straightforward and requires only copying a few folders and files. Simply copy the `library.js` file and the `modules` folder into your Titanium Project, as highlighted in the following screenshot:



After copying the `modules` folder, you will need to click on your `tiapp.xml` file in Titanium Studio and add a reference to `ti.paint` module, as shown in the following screenshot:



How to do it...

Once you have added `library.js` and the native modules to your project, you need to create your application namespaces and use `require` to import the module into your `app.js` file, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  paint : require('ti.paint'),
  library : require('library')
};
```

Creating a window for the recipe

The following snippet demonstrates how to create a `Ti.UI.Window` window to hold all the controls used in this recipe:

```
var win = Ti.UI.createWindow({
  title:"iPad Signature Recipe", backgroundColor:'#fff'
});
```

Adding a document view

The first UI element added to `Ti.UI.Window` is used to display the recipe's document. On iOS, the `Ti.UI.WebView` object has built-in support for displaying PDF files. This sample demonstrates how to load a single PDF document from the `Resources` folder of your Titanium project.

```
var docView = Ti.UI.createWebView({
  top:60, bottom:250, left:10, right:20, url:'w4.pdf'
});
win.add(docView);
```

Adding a signature view

The next UI element added to the `Ti.UI.Window` window is `PaintView`. This `UIView` is provided by the `Ti.Paint` module added during the setup process. This view allows the user to draw on the screen using touch. The following snippet demonstrates how to set up this view to allow the user to draw using a black hue with a stroke size of 10 points:

```
var vwSign = my.paint.createPaintView({
  bottom:60, left:10, right:10, height:140,
  strokeColor:'#000', strokeAlpha:255, strokeWidth:10,
  eraseMode:false, borderColor:'#999'
});
win.add(vwSign);
```

Adding buttons

The next step is to create buttons to save or clear a signature. The `bClear` button is added to the lower-left section of the screen. This button will provide the ability for the user to clear the signature area. This button can also be used to remove a saved signature.

```
var bClear = Ti.UI.createButton({
  title:'Remove Signature', left:10, bottom:10, width:150,
  height:45
});
win.add(bClear);
```

When the click event is fired on the `bClear` button, the signature is cleared and any saved signature files are removed:

```
bClear.addEventListener('click',function(e){
```

1. Call the `clear` method on `vwSign`. This will remove all the contents of `PaintView`.

```
vwSign.clear();
```

2. Next, the `deleteSignature` method is called on the library `CommonJS` module. This removes the stored signature image.

```
my.library.deleteSignature();
```

3. Then, `vwSign` is touch-enabled, and the user is alerted that their signature has been removed. After this is completed, the user will be able to create a new document signature.

```
vwSign.touchEnabled = true;
alert("Signature has been reset.");
});
```

4. Next, the `bSave` button is added to the lower-right side of the iPad screen. This button allows the user to save a copy of their signature for later use.

```
var bSave = Ti.UI.createButton({
  title:'Save Signature', right:10, bottom:10,
  width:150, height:45
});
win.add(bSave);
```

5. When the click event on the `bSave` button is fired, an image of the user's signature is saved to a library folder on the device.

```
bSave.addEventListener('click',function(e){
```

Perform the following steps to complete the saving process:

1. Convert the contents of the signature view into an image blob. This blob is then provided to the library `CommonJS` module's `saveSignature` method to be persisted to the device's documents folder.

```
my.library.saveSignature(vwSign.toImage());
```

2. Once the signature image has been saved to disk, the signature surface is made read-only so no additional updates can be performed:

```
vwSign.touchEnabled = false;
alert("You have successfully signed this document.");
});
```

Reloading a saved signature

The last section of this recipe describes how, on opening a window, to check if the document has been signed and loads the stored signature for display.

```
win.addEventListener('open', function(e) {
```

First the `isSigned` method is called on the `library` module. This method will check if the signature file is present within the device's `library` folder.

```
if(my.library.isSigned()){
```

The `signatureUrl` method is used to provide the signature's image path to the view for display:

```
vwSign.image = my.library.signatureUrl();
```

Since the document has already been signed, the signature view is then disabled so no additional updates can be made without clearing the existing signature first.

```
vwSign.touchEnabled = false;
}
});
```

```
win.open({modal:true});
```

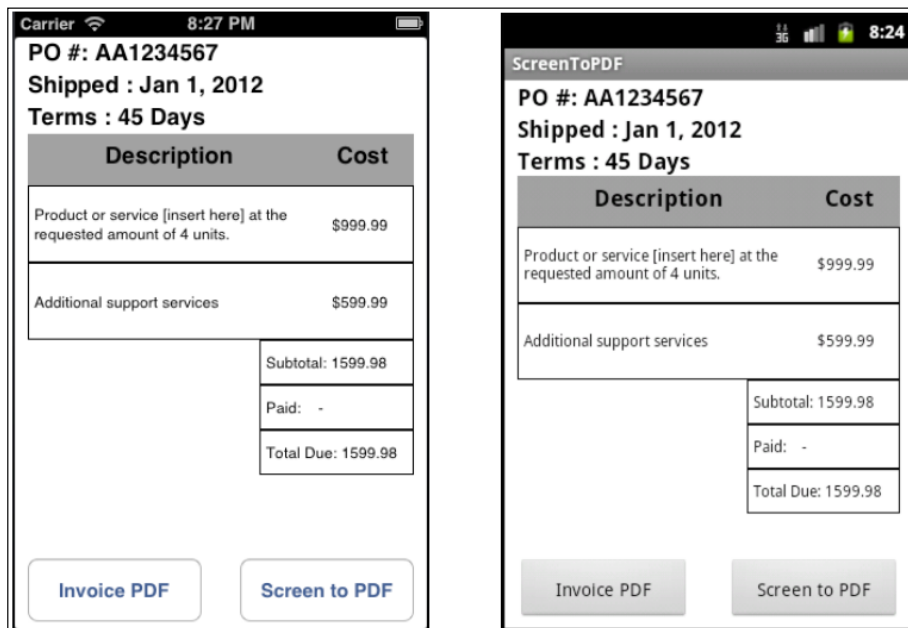
See also

- ▶ This recipe uses the `Ti.Paint` open source module. The `Ti.Paint` module was used in this recipe to provide the user with the ability to sign a document. To learn more about this project, please visit https://github.com/appcelerator/titanium_modules/tree/master/paint/mobile/ios.

Creating PDF documents from images or screenshots

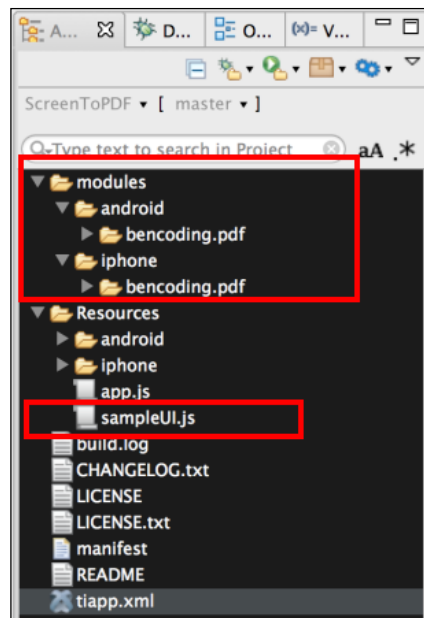
Mobile devices are wonderful consumers of PDF documents. But, what about creating them? Many scenarios, such as issuing recipes or invoices, require a PDF file to be created by the device. One easy approach for doing this is to convert an image into a PDF file. This approach works well with Titanium's robust image creation and maintenance functionality.

This recipe demonstrates how to convert a `Ti.UI.View` into a PDF document. Also demonstrated is how to use Titanium's `Ti.Media.takeScreenshot` API to convert a fullscreen image of your app into a PDF file. This could be helpful for consumers looking to "print" their screen.

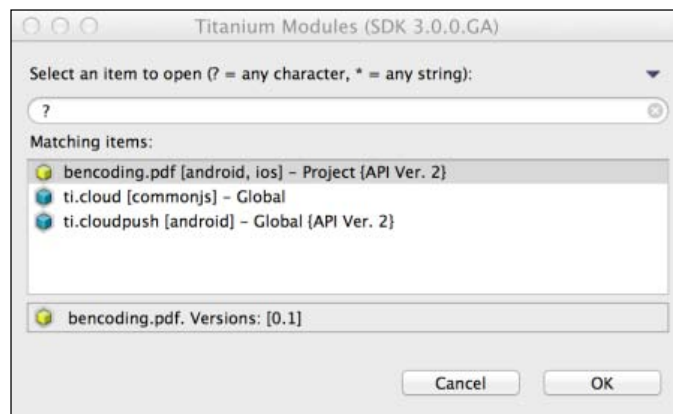


Getting ready

This recipe leverages the `bencoding.pdf` native module. This module and supporting source code can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. The installation process for this recipe is straightforward and only requires copying the two `bencoding.pdf` folders into their appropriate `modules` folder, as highlighted in the screenshot at the end of this section. If your Titanium project does not currently have a `modules` folder, you can simply copy the complete `modules` folder from this recipe into your project. In addition to the two native modules, you will need to copy the `sampleUI.js` file, shown the following screenshot:



After copying the `modules` folder, you will need to click on your `tiapp.xml` file in Titanium Studio, and add a reference to the `bencoding.pdf` modules, as shown in the following screenshot. Please note only one module entry will appear for `bencoding.pdf`, but both iOS and Android will be added to your `tiapp.xml` file, once selected.



How to do it...

Once you have added the `sampleUI.js` file and native modules to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  PDF : require('bencoding.pdf'),
  sampleUI : require('sampleUI'),
  isAndroid : Ti.Platform.osname === 'android'
};
```

Creating a UI for the recipe

Now, we create a `Ti.UI.Window` window to host the recipe's UI elements:

```
var win = Ti.UI.createWindow({
  title:'PDF Screenshot', backgroundColor:'#fff'
});
```

Once `Ti.UI.Window` has been created, the contents of the sample invoice is provided by our `sampleUI` module. The `sampleUI` module uses several `Ti.UI.View` and `Ti.UI.Label` controls to generate a sample invoice layout.

```
var vwContent = my.sampleUI.fetchInvoiceUI();
win.add(vwContent);
```

Creating a PDF from a view

Next, the `makeImageButton` button is added to the recipe's `Ti.UI.Window`. This button will later be used to create a PDF sample invoice.

```
var makeImageButton = Ti.UI.createButton({
  title:'Invoice PDF', bottom:5, left:10,
  width:135, height:50
});
win.add(makeImageButton);
```

When the click event on the `makeImageButton` button is fired, a PDF is generated with the contents from the `Ti.UI.View` containing our sample invoice.

```
makeImageButton.addEventListener('click',function(e){
```

1. The first step in generating a PDF is to create an image from the `Ti.UI.View` containing the invoice layout. The following snippet demonstrates how to do this on both iOS and Android:

```
var image = ((my.isAndroid)?
    vwContent.toImage().media : vwContent.toImage());
```

2. The image blob for the invoice `Ti.UI.View` is then provided to the `convertImageToPDF` method in the `PDF Converters` module. This method converts the provided image into a PDF `Ti.File` blob. In addition to the image on iOS, you can provide a resolution to be used in the conversion process. The following sample uses a resolution of 100:

```
var pdfBlob = my.PDF.createConverters()
    .convertImageToPDF(image,100);
```

3. The invoice PDF `Ti.File` blob can then be saved using the standard `Ti.FileSystem` method, as the following snippet shows:

```
var pdfFile = Ti.FileSystem.getFile(
    Ti.FileSystem.applicationDataDirectory, 'invoice.pdf'
);
pdfFile.write(pdfBlob);
});
```

Creating a PDF document from a screenshot

A similar technique can be used to create a PDF document containing a screenshot of the app. The first step in doing this is to add a button named `ssButton` to the recipe's `Ti.UI.Window`:

```
var ssButton = Ti.UI.createButton({
    title:'Screen to PDF', bottom:5,
    right:10, width:135, height:50
});
win.add(ssButton);
```

When we click on the `ssButton` button, a screenshot will be taken and converted into a PDF file:

```
ssButton.addEventListener('click',function(e) {
```

The first step in this process is to use the `Ti.Media.takeScreenshot` API to take a screenshot of the app:

```
Ti.Media.takeScreenshot(function(event) {
```

The screenshot API returns an image blob of the device's screen:

```
var image = event.media;
```

The screenshot image blob is then provided to the `convertImageToPDF` method in the `PDF Converters` module. This method converts the provided image into a `PDF Ti.File` blob. In addition to the image on iOS, you can provide a resolution to be used in the conversion process. This sample uses a resolution of 96 points:

```
var pdfBlob = my.PDF.createConverters()  
    .convertImageToPDF(image, 96);
```

The screenshot `PDF Ti.File` blob can then be saved using the standard `Ti.FileSystem` methods:

```
var pdfFile = Ti.FileSystem.getFile(  
    Ti.FileSystem.applicationDataDirectory, 'screenshot.pdf'  
);  
pdfFile.write(pdfBlob);  
});  
});
```

See also

- ▶ To learn more about the modules used in this recipe, please visit <https://github.com/benbahrenburg/benCoding.PDF>.

Generating PDF documents using jsPDF

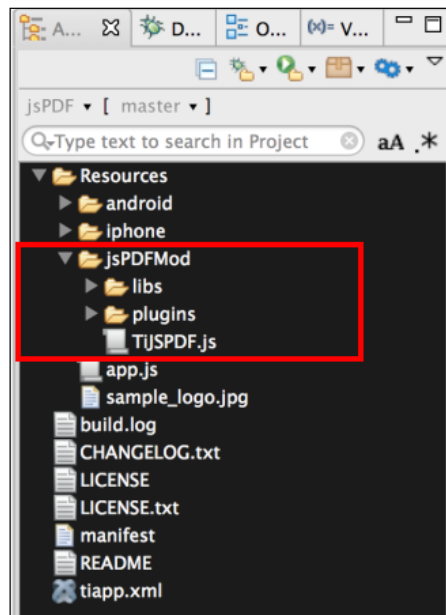
Creating formatted PDF documents on a mobile device can be difficult. Adding cross-platform to the equation only compounds this challenge. The jsPDF JavaScript library and the associated jsPDF Titanium module provide a robust and pure JavaScript cross-platform approach.

This recipe demonstrates how to create formatted PDF documents, similar to the following screenshot, using a powerful JavaScript API:

YOUR Logo	
Your Receipt	
1. Cargo Container	\$2399.99
2. Storage (5 days)	\$1299.99
3. Loading Fee	\$699.99
4. Transport Insurance	\$399.99
Tax 8%	\$383.99
Total:	\$5183.95

Getting ready

This recipe uses the `jsPDF` module for Titanium to create PDF files. This module and supporting source code can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. The installation process for this recipe is straightforward and only requires copying the `jsPDFMod` folder into the `Resources` folder of your Titanium project, as shown in the following screenshot:



How to do it...

Once you have added the `jsPDFMod` folder to your project, you will need to create your application namespaces in the `app.js` file, and use `require` to import the module into your code, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  jsPDF : require('./jsPDFMod/TiJSPDF'),
  isAndroid : Ti.Platform.osname === 'android'
};
```

Creating a UI for the recipe

This recipe contains a basic UI that allows the user to generate a recipe PDF and open an e-mail dialog.

1. Create a `Ti.UI.Window` window to hold all UI controls:

```
var win = Ti.UI.createWindow({
  backgroundColor: '#eee', title: 'jsPDF
  Sample', layout: 'vertical'
});
```

2. Next, the `goButton` button is added to the recipe's `Ti.UI.Window`:

```
var goButton = Ti.UI.createButton({
  height: '50dp', title: 'Email a receipt',
  left: '20dp', right: '20dp', top: '40dp'
});
win.add(goButton);
```

3. The `goButton` button's click event launches the PDF creation process by calling the `generatePDFExample` method:

```
goButton.addEventListener('click', function (e) {
  generatePDFExample();
});
```

Creating a PDF document using jsPDF

The following snippet describes how to use `generatePDFExample` to create a PDF document using the `jsPDF` module:

```
function generatePDFExample(){
```

The next line demonstrates how to create a new instance of the `jsPDF` module. This creates a new virtual PDF document.

```
var doc = new my.jsPDF();
```

The following snippet demonstrates how to add document properties to the PDF document object:

```
doc.setProperties({
  title: 'Title', subject: 'This is the subject',
  author: 'add author', keywords: 'one, two,
  three', creator: 'your organization'
});
```

The following snippet demonstrates how to embed an image into a PDF document. It is important that the images contain a full native path as shown here, otherwise the document will generate errors on saving:

```
var imgSample1 = Ti.Filesystem.resourcesDirectory +
  'sample_logo.jpg';
doc.addImage(imgSample1, 'JPEG', 10, 20, 35, 17);
```

We then create a header using Helvetica bold and a font size of 32:

```
doc.setFont("helvetica");
doc.setFontType("bold");
doc.setFontSize(32);
doc.text(20, 50, 'Your Receipt');
```

The following code snippet creates the itemized recipe section with normal font and a size of 18:

```
doc.setFontType("normal");
doc.setFontSize(18);
doc.text(20, 70, '1. Cargo Container $2399.99');
doc.text(20, 80, '2. Storage (5 days) $1299.99');
doc.text(20, 90, '3. Loading Fee $699.99');
doc.text(20, 100, '4. Transport Insurance $399.99');
```

A `Ti.File` object is created to the location where we wish to save `recipe.pdf`:

```
var myFile = Ti.Filesystem.getFile(
  Ti.Filesystem.applicationDataDirectory, 'recipe.pdf');
```

The `Ti.File` object is then passed into the `jsPDF` module's `save` method. The `save` method will generate a PDF file with the attributes created earlier:

```
doc.save(myFile);
```

The `Ti.File` reference for the saved PDF file is then provided to `Ti.UI.EmailDialog` as an attachment and opened, so the user can compose an e-mail:

```
var emailDialog = Ti.UI.createEmailDialog();
emailDialog.addAttachment(myFile);
emailDialog.open();
};
```

See also

- ▶ To learn more about the jsPDF project created by James Hall, please visit the Github repository at <https://github.com/MrRio/jsPDF>.
- ▶ This recipe uses the a Titanium module for jsPDF. Please refer to the following links for additional documentation, samples, and guidance with the module:

Author: Malcolm Hollingsworth

Repository: <https://github.com/Core-13>

Sponsoring organization: Core13; website: core13.co.uk

Creating a scheduling chart using RGraph

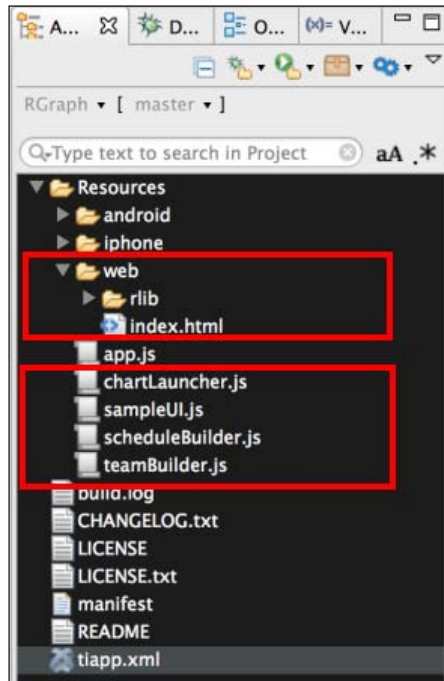
A picture can be worth a thousand words, and in today's fast-paced business environments, using graphics to display information can be a competitive advantage. This recipe demonstrates how you can use the popular RGraph JavaScript library to create a chart reflecting project scheduling. The following screenshots illustrate what this recipe will look like when completed:



Getting ready

This recipe uses RGraph and several non-native components to display the chart. These components can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe.

The first step in setting up this recipe is to copy the `web` folder (highlighted in the following screenshot) into the `Resources` folder of your project folder. After copying the `web` folder, you will need to then copy the `teamBuilder.js`, `chartLauncher.js`, `sampleUI.js`, and `scheduleBuilder.js` files into the `Resources` folder of your Titanium project, as shown in the following screenshot:



How to do it...

After adding the `web` folder and CommonJS modules, as discussed earlier, you will need to create your application namespaces in the `app.js` file and use `require` to import the module into your code, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  schedule : require('scheduleBuilder'),
  team : require('teamBuilder'),
  chartDisplay : require('chartLauncher'),
  sampleUI : require('sampleUI')
};
```

Creating a UI for the recipe

The next step in this recipe is to create the main `Ti.UI.Window` window. This `Ti.UI.Window` will be used to launch the recipe's functionality.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'RGraph Sample'
});
```

After the `Ti.UI.Window` window is created, the `fetchDemoView` method is called on our `sampleUI` module. This displays the recipe's instructions to the user:

```
win.add(my.sampleUI.fetchDemoView());
```

Creating schedules and assigning tasks

This section of the recipe demonstrates how to create schedules, and assign tasks to individuals on your project.

1. The first step is to create a schedule for a team member. This is done by creating a new schedule object and providing the individual's name. The following line demonstrates how to create a schedule for Richard:

```
var richard = new my.schedule('Richard');
```

2. Next, the `createTask` method is used to create a task for the individual.

```
richard.addTask({
  taskName: 'Website', startDay: 15,
  duration: 28, percentageComplete: 67,
  comment: 'Work on website'
});
richard.addTask({
  taskName: 'Fix Web Servers', startDay: 40,
  duration: 15, percentageComplete: 50,
  comment: 'Work with vendor'
});
```

The `createTask` method has the following parameters:

- **Task name:** A string containing the name of the task
 - **Start on day:** This is the position (day) the task is to start
 - **Duration:** The number of days the task will take
 - **Percentage complete:** The percentage completion of the task
 - **Comment:** The comment for the task
3. Finally, the team member's schedule is added to the team:

```
my.team.add(richard);
```

The next snippet demonstrates how to perform the steps discussed earlier, for another team member:

```
var rachel = new my.schedule('Rachel');
rachel.addTask({
  taskName: 'Mock-up', startDay: 0,
  duration: 28, percentageComplete: 50,
  comment: 'Create initial product mock-ups'
});
rachel.addTask({
  taskName: 'Mobile Images', startDay: 40,
  duration: 25, percentageComplete: 74,
  comment: 'Create mobile assets'
});
my.team.add(rachel);
```



You can add additional individuals by following the pattern detailed here.

Launching the example

Perform the following steps to launch the example we created in the previous section:

1. The first step is to add the `goButton` button to the `Ti.UI.Window` window to allow the user to launch the chart sample:

```
var goButton = Ti.UI.createButton({
  title: 'View schedule', bottom: '40dp', left: '25dp',
  right: '25dp', width: Ti.UI.FILL, height: '60dp'
});
win.add(goButton);
```

2. The click event of the `goButton` button launches the chart example by using the `openDialog` method on the `chartLauncher` module. The `openDialog` method uses the `my.team` object created previously to generate a Gantt chart containing a team schedule.

```
goButton.addEventListener('click', function(e) {
```

Calling the `openDialog` method is demonstrated in the following snippet:

```
my.chartDisplay.openDialog(my.team);
});
```

How it works...

The `chartLauncher` module (`chartLauncher.js`) is used to display the team's scheduled tasks. The following line demonstrates how this module uses `RGraph` to create a Gantt chart with the results. First, the `openDialog` method is added to the `exports` object:

```
exports.openDialog = function(team) {
```

The next step in this part of the recipe is to create a window to display to the user:

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff',
  title: 'Sample Chart', barColor: '#000'
});
```

After `Ti.UI.Window` is created, a `Ti.UI.WebView` object is attached. The `RGraph` chart information is contained in the `index.html` file that `Ti.UI.WebView` displays:

```
var webView = Ti.UI.createWebView({
  height:Ti.UI.FILL, width:Ti.UI.FILL, url:'web/index.html'
});
win.add(webView)
```

On the `Ti.UI.Window` open event, the team's schedule information is passed into the `index.html` file containing the `RGraph` methods:

```
win.addEventListener('open', function(e) {
```

The `createSchedules` and `createComments` methods are called on the team object. These format the schedule and comment information so that `RGraph` can display the details.

```
var schedules = JSON.stringify(team.createSchedule());
var comments = JSON.stringify(team.createComments());
```

A load event listener is added to `Ti.UI.WebView` after the `Ti.UI.Window` window is opened. This allows for the contents of `Ti.UI.WebView` to be fully loaded before the information is injected.

```
webView.addEventListener('load', function(x) {
```

The `evalJS` method is used to pass the formatted team information to the `addGraph` method currently being displayed in `Ti.UI.WebView`. You can view the contents of the `addGraph` method by opening the `index.html` file supported with this recipe.

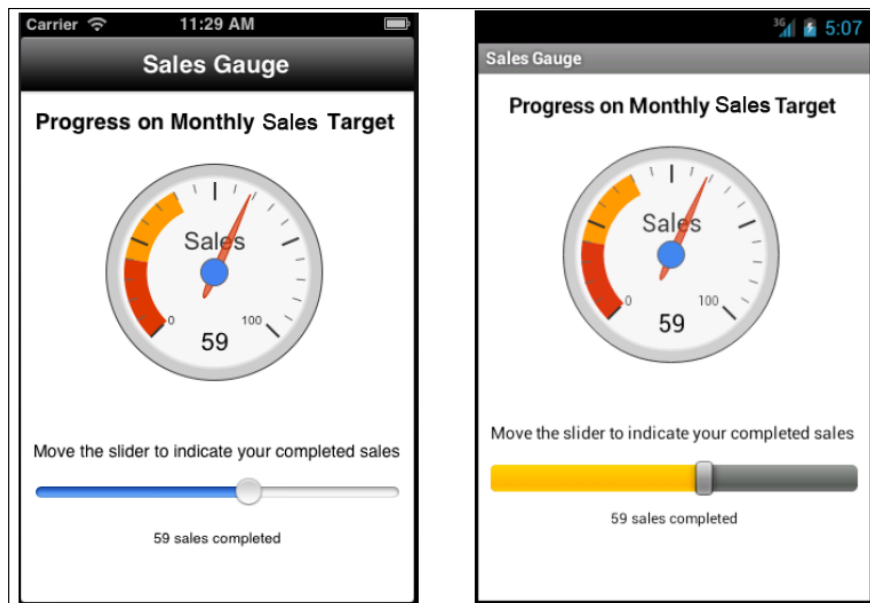
```
webView.evalJS('addGraph(' + schedules + ', ' +
  comments + ')');
});
win.open({modal:true});
};
```


See also

- ▶ This recipe uses the powerful RGraph HTML5/JavaScript canvas library to create the Gantt chart displayed. I encourage you to visit their website, <https://rgraph.net>, to learn more about RGraph and the different charting options provided.

Displaying information with Google gauges

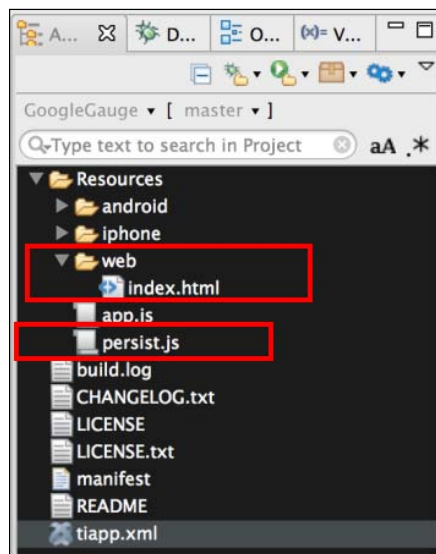
Gauges are a powerful way to display velocity, status, or simple measurements. This recipe demonstrates how to use the Google gauge chart control to display progress towards meeting monthly sales targets. The following screenshots show this completed recipe running on both iOS and Android:



 When running this recipe on Android, please note Android 4.0 or higher is required.

Getting ready

This recipe uses Google Charts to display the gauge. When adding this recipe to your app, you will need to design for a network dependency as Google Charts requires a connection to render chart information. These components can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. The first step in setting up this recipe is to copy the `web` folder highlighted in the following screenshot into the `Resources` folder of your project folder. After copying the `web` folder, you will need to copy the `persist.js` file into the `Resources` folder of your Titanium project as shown in the following screenshot:



How to do it...

After adding the `web` folder and `persist.js` CommonJS module as discussed in the previous section, you will need to create your application namespaces and use `require` to import the module into your code, as the following snippet demonstrates:

```
//Create our application namespace
var my = {
  persist : require('persist'),
  session:{},
  isAndroid : Ti.Platform.osname === 'android'
};
```

Loading saved sales information

After our namespace has been created, the first step is to load our saved sales information. If no prior sales data has been saved, a default of 10 is provided.

```
my.session.sales = my.persist.load();
```

Creating a UI for the recipe

Now we create the UI to display our sales data.

1. First, a `Ti.UI.Window` window is created to host all of our UI elements:

```
var win = Ti.UI.createWindow({
    backgroundColor: '#fff', title: 'Sales Gauge',
    barColor: '#000', layout: 'vertical', fullscreen: false
});
```

2. Next, a `Ti.UI.WebView` is created to display the Google gauge. This `Ti.UI.WebView` is used to display a local html page that hosts the Google Chart control. The following snippet demonstrates how to load a local html page (`index.html`) from the project's web folder:

```
var webView = Ti.UI.createWebView({
    top:10, height:200, width:200, url:'web/index.html'
});
win.add(webView)
```

3. Finally, a `Ti.UI.Slider` is added to the recipe's `Ti.UI.Window`. The user can use this slider to adjust their monthly sales.

```
var slider = Ti.UI.createSlider({
    min:0, max:100, value:my.session.sales, left:10,
    right:10, height:'30dp', top:10
});
win.add(slider);
```

Adjusting sales

As the user adjusts the `Ti.UI.Slider`, the following methods are used to update the Google gauge with the new sales numbers. The `updateSales` method updates the contents of `Ti.UI.WebView` with the new sales number:

```
function updateSales(sales){
```

The passed-in `sales` value is placed into the session's `my.session.sales` property for later use:

```
my.session.sales = sales;
```

The `evalJS` method is called on `Ti.UI.WebView` to provide the new sales information to the `updateGauge` method contained within the example `index.html` file. This method is used to update the Google gauge. For more details, please see the contents of this recipe's `index.html` file.

```
webView.evalJS('updateGauge(' + my.session.sales + ')');
```

After updating `Ti.UI.WebView`, the new sales value is saved into `Ti.App.Properties` for later use:

```
my.persist.save(my.session.sales);
};
```

The change event on the recipe's `Ti.UI.Slider` method adjusts the Google gauge as the user moves the slider:

```
slider.addEventListener('change', function(e) {
```

Each time the change event is fired, the new `Ti.UI.Slider` value is provided to the `updateSales` method, to be reflected by the Google gauge hosted in the recipe's `Ti.Ui.WebView`.

```
updateSales(Math.round(e.value));
});
```

Reloading saved sales

Each time `Ti.Ui.Window` is loaded, the following code is used to display the saved sales values and initialize the Google gauge. The next snippet demonstrates how the recipe reloads the sales information on opening `Ti.UI.Window`:

```
win.addEventListener('open', function(e) {
```

First, a network connection is performed. Since the recipe uses Google Charts, a network connection is required to display this information:

```
if(!Ti.Network.online){
    alert('Network connection required.');
```

```
    return;
```

```
}
```

Finally, a load event listener is added to `Ti.UI.WebView`. Once loaded, the `updateSales` function is called to initialize any previously saved sales information:

```
webView.addEventListener('load', function(x) {
    assist.updateSales(my.session.sales);
});
```


See also

- ▶ This recipe uses Google Charts to provide the gauges displayed. To learn more about Google's charting tools, please visit their developer site, <https://google-developers.appspot.com/chart/>.

6

Using Location Services

In this chapter we will cover:

- ▶ Native geolocation using basicGeo
- ▶ Using the Ti.GeoProviders framework for geolocation
- ▶ Multitenant geolocation
- ▶ Calculating distance between addresses
- ▶ Background geolocation management

Introduction

Until the proliferation of mobiles, it was often challenging to tell the location of your users, making it difficult to provide location-based services. Now, almost every app developer has access to real-time geolocation information directly from the user's device.

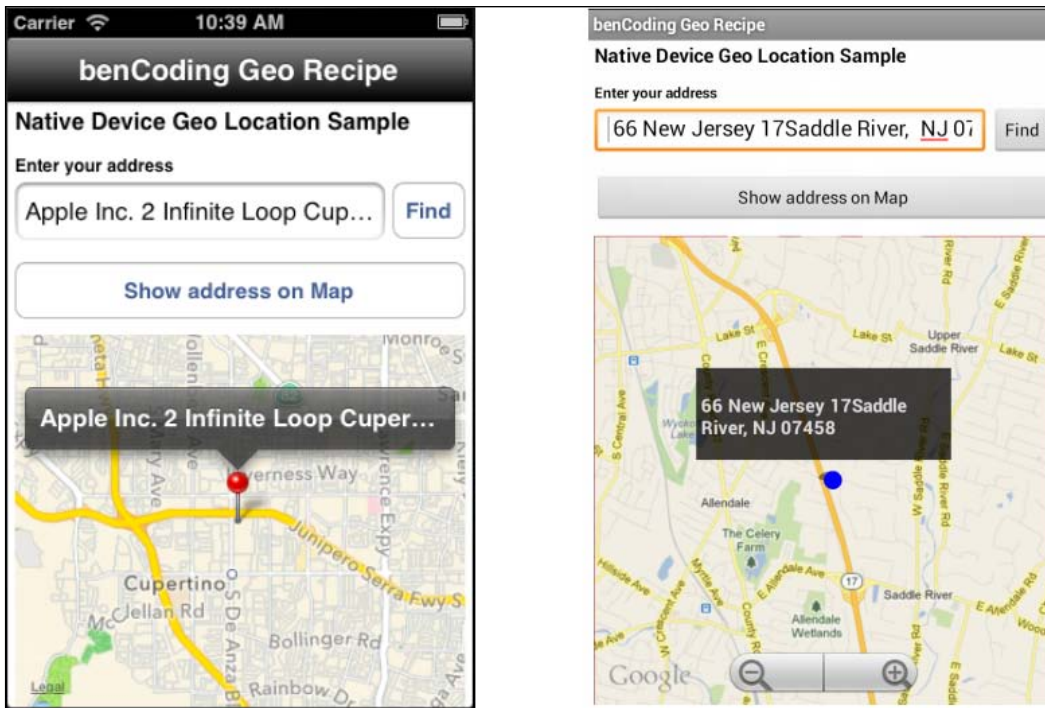
With the globally mobile nature of today's employees, providing location-aware apps to your enterprise is just as important, if not more so as in the consumer market. Geolocation is playing a large role in enterprise organizations across a variety of domains including, fleet management, shipment tracking, sales routing, or simply providing real-time relevant information on resources available to mobile employees.

Through a series of location service examples, this chapter outlines a variety of different approaches you can leverage in your Enterprise Titanium app.

Native geolocation using basicGeo

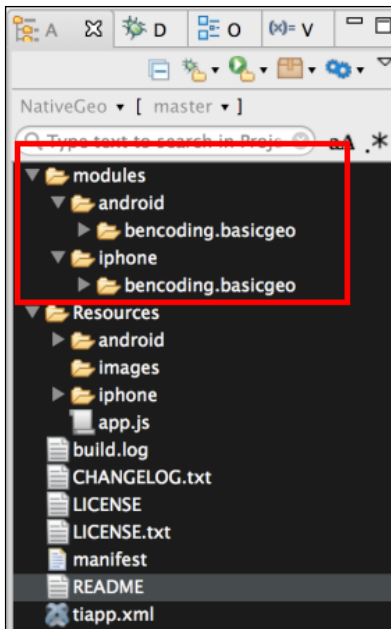
Android and iOS both provide powerful geolocation platform APIs to find the geographical latitude and longitude of a given address. You can use the `basicGeo` module to access these platform APIs in your Titanium module.

This recipe discusses how to leverage this module in your Titanium app to perform forward and reverse geolocation operations. The following screenshots illustrate this recipe running on both an iPhone and an Android device:

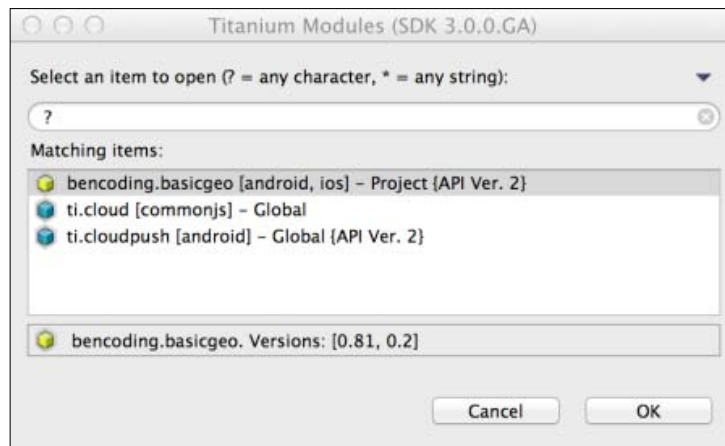


Getting ready

This recipe uses the `benCode.basicGeo` native module. This module can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing these in your project is straightforward. Simply copy the `modules` folder into your project as highlighted in the following screenshot:



After copying the `modules` folder, you will need to click on your `tiapp.xml` file in Titanium Studio and add a reference to the `bencoding.basicgeo` module as shown in the following screenshot:



How to do it...

Once you have added the `bencoding.basicgeo` module to your project, you next need to create your application namespaces in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  basicGeo : require('bencoding.basicgeo'),
  isAndroid : Ti.Platform.osname === 'android'
};
```

Adding availability helpers

With an ever-increasing number of devices each having different capabilities, it is a common requirement to provide progressive enhancements based on the ability of sensors. The availability feature in the `basicGeo` module provides a series of properties that can be used to determine the capabilities of the devices. The following snippet shows how to create the `Availability` proxy, for later use in this recipe:

```
my.available = my.basicGeo.createAvailability();
```

Adding the location services purpose

To use location services on iOS, Apple requires a purpose or reason for the app to access the GPS. On first request, this message will be used in the message presented to the user for approval to use their device's location services. The following snippet demonstrates how to add this purpose to the `basicGeo` module.

```
my.basicGeo.purpose = 'Demo of basicGeo';
```

Building the recipe UI

This following code snippet describes how to create the UI shown in this recipe's earlier screenshots:

1. The first step is to create the `Ti.UI.Window` to which all visual elements will be attached.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'benCoding Geo Recipe',
  barColor: '#000', fullscreen: false
});
```

2. Next a `Ti.UI.TextField` is added to the recipe's `Ti.UI.Window`. The contents of this `Ti.UI.TextField` will be used during the forward geolocation lookup detailed later in this recipe.

```
var txtAddress = Ti.UI.createTextField({
```

```

    hintText:'enter address', height:40,
    left:5, right:60, top:55
  });
  win.add(txtAddress);

```

3. The next step in this recipe is to add a `Ti.Map.View` to the recipe's `Ti.UI.Window`. This UI component will be used to display the address entered in the `txtAddress` `Ti.UI.TextField`.

```

var mapView = Ti.Map.createView({
  top:160, bottom:0, left:5, right:5,
  userLocation:false
});
win.add(mapView);

```

Working with place objects

When performing geolocation lookups, the `basicGeo` module returns a collection of places that match the address provided. The `placeHelpers` object provides convenient functions for working with the results returned from the `basicGeo` module.

```
var placeHelpers = {
```

1. The `address` function provides a formatted address from a `basicGeo` place object. This is used by the "find current location" feature in this recipe.

```

  address :function(place){
    if(my.isAndroid){

```

2. The Android reverse geolocation API provides an `address` field already formatted.

```

      return place.address;
    }else{

```

3. In iOS, the address information is provided as an array of lines. The following method converts these address lines into a formatted string:

```

      var lines = place.addressDictionary
      .FormattedAddressLines;
      var iLength = lines.length, address = '';
      for (iLoop=0;iLoop < iLength;iLoop++){
        if(address.length>0){
          address += ' ' + lines[iLoop];
        }else{
          address = lines[iLoop];
        }
      }
      return address;
    }
  },

```

4. The `addToMap` methods add the place information provided by the `basicGeo` reverse geolocation to the `Ti.Map.View` created earlier in this recipe.

```
addToMap: function(place) {
    var lat = place.latitude,
        lng = place.longitude,
        title = placeHelpers.address(place);
    var pin = Ti.Map.createAnnotation({
        latitude:lat,longitude:lng,
        title:title
    });
    mapView.addAnnotation(pin);
}
```

5. A region is created using the latitude and longitude information from the `basicGeo` module. The `setLocation` method is then called to zoom the `Ti.Map.View` to the pin's coordinates.

```
var region = {latitude:lat,
              longitude:lng,animate:true,
              latitudeDelta:0.04, longitudeDelta:0.04};
mapView.setLocation(region);
}
};
```

Finding current location

Perform the following steps to find the current location:

1. The `findButton` demonstrates how to perform a reverse geolocation lookup using the device's current coordinates. These coordinates are then used to find the current address of the user.

```
var findButton = Ti.UI.createButton({
    right:5, top:55, height:40, width:50, title:'Find'
});
win.add(findButton);
```

2. On the click event of the `findButton`, the `getCurrentPlace` method is called on the `basicGeo` module's `currentGeoLocation` proxy.

```
findButton.addEventListener('click',function(e) {
```

3. The `resultsCallback` method is created to handle the results returned by the `getCurrentPlace` method. The result from the `getCurrentPlace` method is provided to the `e` argument.

```
function resultsCallback(e) {
```

- The `e.success` property provides a flag to determine if the geolocation operation has encountered an error.

```
if(!e.success){
    alert('Sorry we encountered an error.');
```

```
return;
}
```

- The `e.placeCount` property provides the number of place objects returned. Generally, this is a number between 0 and 12 depending on the accuracy. If no places are returned, alert the user that the address was not found.

```
if(e.placeCount === 0){
    alert('Unable to find your address.');
```

```
return;
}
```

- The first place in the `places` collection is provided to the `placeHelpers.address` method. This method provides a formatted address string that is then presented to the user in the `Ti.UI.TextField txtAddress`.

```
txtAddress.value =
    placeHelpers.address(e.places[0]);
};
```

- A new instance of the `CurrentGeoLocation` proxy is created. This proxy contains methods to perform geolocation operations using the device's current coordinates.

```
var currentGeo = my.basicGeo.createCurrentGeolocation();
```

- If the recipe is running on an Android device, the `setCache` method can be called. This enables the `basicGeo` module to use the last best location cached by the device. This provides faster lookup speeds, but can result in less accurate location information.

```
if(my.isAndroid){
    currentGeo.setCache(true);
}
```

- The final step in returning the device's current address is to call the `getCurrentPlace` method. This method performs a reverse geolocation lookup using the device's coordinates and provides a collection of places to the `provide` callback method. The following snippet demonstrates how to call this method using the `resultsCallback` as the callback argument.

```
currentGeo.getCurrentPlace(resultsCallback);
});
```


Forward location lookup

1. The `searchTextAddressButton Ti.UI.Button` performs a forward geolocation lookup using the native device API.

```
var addressOnMapButton = Ti.UI.createButton({
    right:5, left:5, height:40,
    title:'Show address on Map',top:110
});
win.add(searchTextAddressButton);
```

2. The `searchTextAddressButton Ti.UI.Button` on click event performs a forward geolocation lookup using the address entered in the `txtAddress Ti.UI.TextField`.

```
searchTextAddressButton.addEventListener('click',function(e) {
```

3. The first step in the forward geolocation lookup is to verify that the user entered an address in the `txtAddress Ti.UI.TextField`.

```
    if(txtAddress.value.length==0){
        alert('Please enter an address to display');
    }
```

4. The `forwardGeoCallback` method is created to handle the results returned by the `forwardGeocoder` method. The result from the `forwardGeocoder` method is provided to the `e` argument.

```
function forwardGeoCallback(e) {
    if(!e.success) {
        alert('Sorry we encountered an error. ');
        return;
    }
    if(e.placeCount === 0) {
        alert('Unable to find address entered. ');
        return;
    }
}
```

5. The `addToMap` method is creating a call using the first place in the `places` collection. This method will create a pin on the `Ti.Map.View` with the place object details.

```
    placeHelpers.addToMap(e.places[0]);
};
```

6. The next step in performing a forward geolocation lookup, is to call the `forwardGeocoder` method and provide a callback method as shown in the following line:

```
var geoCoder = my.basicGeo.createGeocoder();
```

- In the sample, the `txtAddress.value` and `forwardGeoCallback` are provided to the `forwardGeocoder` method. The results of the forward geolocation lookup will be provided to the `forwardGeoCallback` function as discussed earlier in this recipe.

```
geocoder.forwardGeocoder(txtAddress.value,
    forwardGeoCallback);
});
```

Device capability check

Follow these steps to perform a device capability check:

- This recipe requires that the device supports reverse and forward geolocation operations. On load of the main recipe's `Ti.UI.Window`, the `Availability` object created earlier in this recipe is used to alert the user if his/her device can support running this recipe.

```
win.addEventListener('open', function(e) {
```

- The `reverseGeoSupported` property is checked to determine if the device running the recipe can support running the recipe.

```
    if(!my.available.reverseGeoSupported) {
```

- If the device does not support reverse geolocation, the user is alerted of the possible issues.

```
        if(my.isAndroid) {
            alert("Configuration isn't supported.");
        }else{
            alert('iOS 5 or greater is required');
        }
    }
});
win.open({modal:true});
```



This recipe requires Android 4.0 or greater when running in the emulator due to an Android emulator defect.



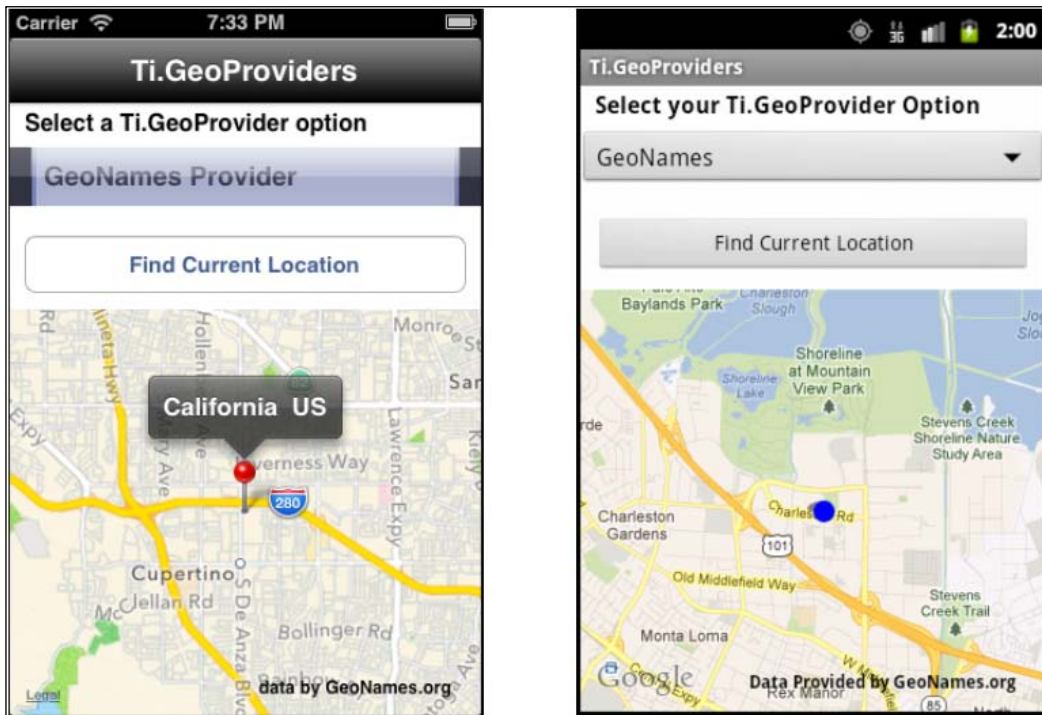
See also

- ▶ This recipe uses the `basicGeo` module to perform native geolocation. For licensing, source code, and to learn more about this project, please visit <https://github.com/benbahrenburg/benCoding.BasicGeo>.

Using the Ti.GeoProviders framework for geolocation

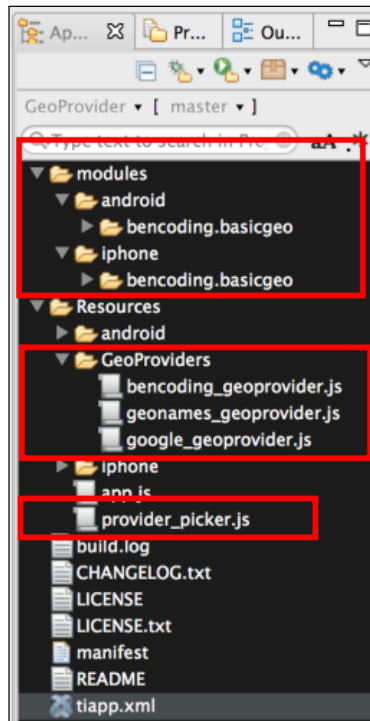
The `Ti.GeoProviders` framework provides a multiprovider approach to reverse geolocation. With a variety of providers, the `Ti.GeoProviders` framework provides a common API for handling GeoNames.org, Google, and `basicGeo` geolocation operations.

The following recipe demonstrates how to use the `Ti.GeoProviders` framework and its associated providers. The following screenshots illustrate this recipe running on both an iPhone and an Android device:

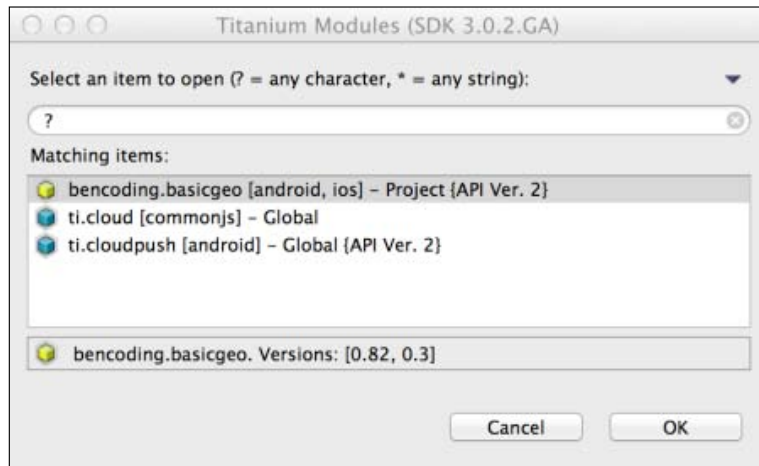


Getting ready

This recipe uses both CommonJS and native modules. These can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Simply copy the `Ti.GeoProviders` folder into the `Resources` folder of your project and then copy the `modules` folder into your project as shown in the following screenshot. Finally, copy the `provider_picker.js` file into the `Resources` folder of your Titanium project as also highlighted in the following screenshot:



After copying the file and folder mentioned here, you will need to click on your **tiapp.xml** file in Titanium Studio and add a reference to the `bencoding.basicgeo` module as shown in the following screenshot:



How to do it...

Once you have added native and CommonJS modules to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  picker : require('provider_picker'),
  currentProvider:null,
  providers:[
    require('./GeoProviders/geonames_geoprovider'),
    require('./GeoProviders/bencoding_geoprovider'),
    require('./GeoProviders/google_geoprovider')
  ]
};
```

Adding your API key

Many of the Geo Providers require an API key. The `Ti.GeoProvider` includes the `addKey` method to allow you to associate your API key before making a service call. The following snippet demonstrates how to add the API key `demo` to your service calls.

```
my.currentProvider.addKey('demo');
```

Adding your purpose

To use location services in iOS requires a purpose or reason for the app to access the GPS. On the first request, this message will be presented to the user. The following code demonstrates how to add this purpose to the `Ti.GeoProviders` using the `addPurpose` method:

```
my.currentProvider.addPurpose('Demo of Geo Provider');
```



Android does not require the purpose be provided; in this case the purpose defined is not used when accessing the GPS.

Building the recipe UI

The following code snippet describes how to create the UI shown in this recipe's earlier screenshots:

1. The first step is to create the `Ti.UI.Window` to which all visual elements will be attached.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Ti.GeoProviders',
  barColor: '#000', fullscreen: false
});
```

2. Next a `Ti.Map.View` is added to the `Ti.UI.Window`. This is used to plot the location information provided by the `GeoProvider`.

```
var mapView = Ti.Map.createView({
  top:140, bottom:0, width:Ti.UI.FILL,
  userLocation:false
});
win.add(mapView);
```

3. A `picker` control is then added to the `Ti.UI.Window`. This control contains a list of providers and a callback method to switch between them. When the user updates the selected picker, the `lookup.updateProvider` method is called to switch the active `Ti.GeoProvider`. See the *Lookup functions* section in this recipe for more details.

```
var picker = my.picker.createPicker({
  top:30, height:40},lookup.updateProvider
});
win.add(picker);
```

4. The `findButton` button is the final UI component added to the recipe's `Ti.UI.Window`. This `Ti.UI.Button` is used to run the recipe's reverse geolocation lookup.

```
var findButton = Ti.UI.createButton({
  title:'Find Current Location',
  left:10, right:10,top:30,height:40
});
win.add(findButton);
```

Running the reverse geolocation

For running the reverse geolocation, perform the following steps:

1. When the user presses the `findButton`, the click event is fired and the following snippet is run:

```
findButton.addEventListener('click',function(e){
```

2. The first step in this section of the recipe is to check that a network connection is available. A network connection is needed to contact the `Ti.GeoProvider` web service to perform the reverse geolocation lookup.

```
  if(!Ti.Network.online){
```

3. If no network connection is available, the user is alerted to this requirement and the lookup process is stopped.

```
    alert("You must be online to run this recipe");
    return;
  }
```

4. The final step in this section of the recipe is to call the `getCurrentAddress` method of `Ti.GeoProviders`. The method will use the `Ti.Geolocation` API to obtain the device's coordinates, and then use the specific logic of `Ti.GeoProviders` to return an address object to the `onSuccess` callback method provided. If an error occurs during the geolocation process the `onError` callback method will be called and provide the error details. The following snippet demonstrates how to call the `getCurrentAddress` method:

```

my.currentProvider.getCurrentAddress(
lookup.onSuccess,lookup.onError);
});

```

Lookup functions

Now perform the following steps:

1. The `lookup` object is used in this recipe to display the results returned by the `GeoProvider`.

```
var lookup = {
```

2. Using the `picker` control discussed earlier, the user can change the recipe's `Ti.GeoProvider`. When the provider is changed, the `updateProvider` method is called with the new provider details to be loaded.

```
updateProvider : function(providerKey){
```

3. Based on the `providerKey` given, the `updateProvider` method will switch the `my.currentProvider` object reference. Provider-specific details such as API key details will also be handled as part of this method. The `geo names provider snippet` demonstrates how provider switching is performed in this recipe.

```
my.currentProvider = my.providers[providerKey];
```

```

if(my.currentProvider.providerName == 'geonames'){
my.currentProvider.provider.addKey('demo');
}

```

4. Cross-provider methods such as `addPurpose` are performed at the end of the `updateProvider` method as they are leveraged by all `Ti.GeoProviders`.

```

my.currentProvider.addPurpose('Geo Demo');
},

```

5. The `addToMap` method is used to create a map pin using the latitude, longitude, and address information provided by the `GeoNames Ti.GeoProvider`.

```

addToMap: function(lat,lng,title){
var pin = Ti.Map.createAnnotation({
latitude:lat,longitude:lng,
title:title
});
mapView.addAnnotation(pin);

```


6. A region is created using the latitude and longitude information from the `Ti.GeoProvider`. The `setLocation` method is then called to zoom the `Ti.Map.View` to the pin's coordinates.

```
var region = {latitude:lat,
              longitude:lng,animate:true,
              latitudeDelta:0.04,
              longitudeDelta:0.04};
mapView.setLocation(region);
},
```

7. The `onSuccess` method is used to handle the successful return from the `Ti.GeoProviders`. This method is used to orchestrate all user interactions after the successful return from the `Ti.GeoProviders`.

```
onSuccess : function(e){
  if(!e.found){
    alert("Unable to find your location");
    return;
  }
}
```

8. The `generateAddress` method is used to create a value for the `title` variable. The `title` variable is then used in the creation of the map pin. As `Ti.GeoProviders` can contain different formats, the `generateAddress` function is used to create a formatted address to be used for display purposes.

```
var title = my.currentProvider.generateAddress(e);
lookup.addToMap(e.latitude,e.longitude,title);
},
```

9. The `onError` method is used by the `Ti.GeoProviders` to return error information if an issue occurred during the reverse geolocation process. All error details are accessible in the `e` argument.

```
onError: function(e) {  
    alert("Error Details:" + JSON.stringify(e));  
}  
};
```

See also

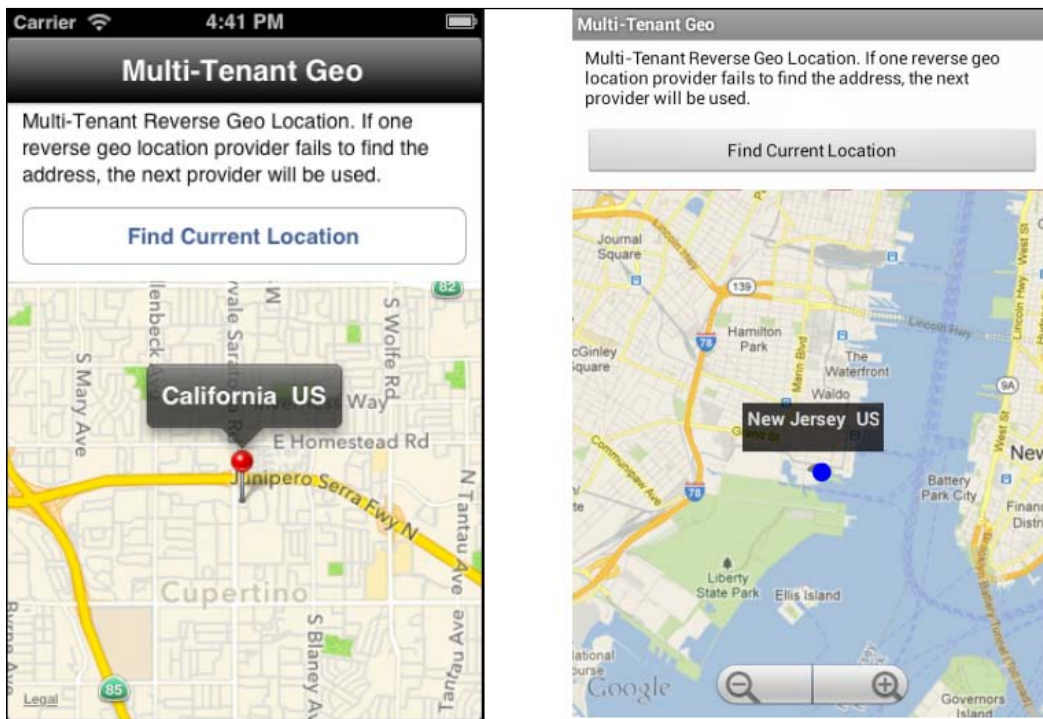
To learn more about the `basicGeo` Titanium module used in this recipe, you can review the following links:

- ▶ **Ti.GeoProvider Framework:** For licensing, source code, and to learn more about this project please visit <https://github.com/benbahrenburg/Ti.GeoProviders>.
- ▶ **basicGeo Module:** For licensing, source code, and to learn more about this project please visit <https://github.com/benbahrenburg/benCoding.BasicGeo>.
- ▶ **GeoNames GeoProvider:** The GeoNames provider uses the `GeoNames.org` web service. For licensing, usage, rates, and documentation please visit <http://www.geonames.org/>.
- ▶ **Google GeoProvider:** The Google provider uses the Google Geocoding API. For licensing, usage, rates, and documentation please visit <https://developers.google.com/maps/documentation/geocoding/>.

Multitenant geolocation

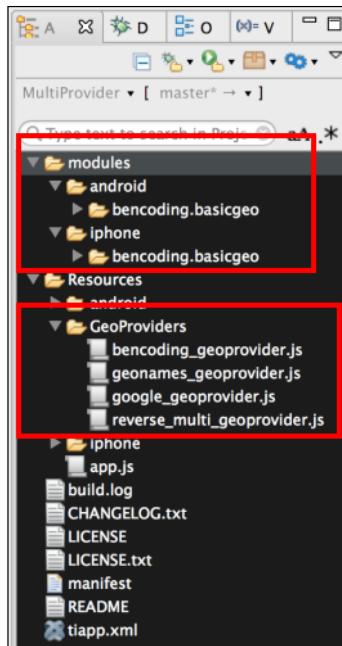
As discussed earlier in this chapter, the `Ti.GeoProviders` framework provides a multiprovider approach to reverse geolocation. The multitenant component includes the ability for the `Ti.GeoProviders` framework to fail over, if a provider is unable to find a suitable location. This multitenant approach helps to ensure your geolocation functionality works for your globally mobile employees.

The following recipe demonstrates how to use the multitenant `Ti.GeoProviders` framework to perform reverse location lookups using a failover approach. The following screenshots illustrate this recipe running on both an iPhone and an Android device:

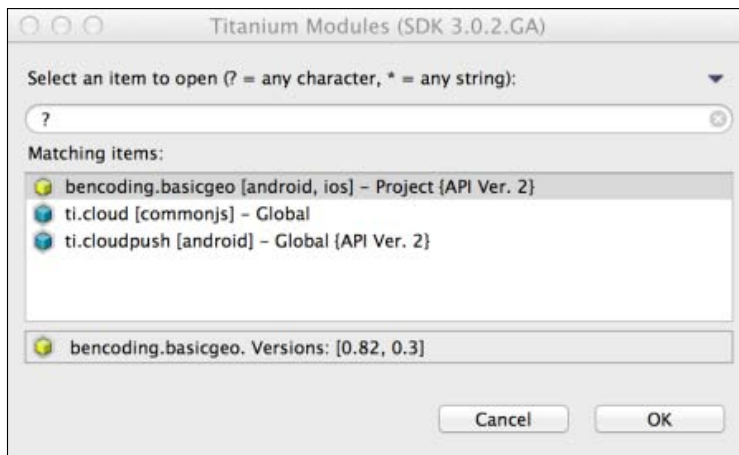


Getting ready

This recipe uses both CommonJS and native modules. These can be downloaded from the source code provided by the book, or individually through the links provided in the See also section at the end of this recipe. Simply copy the `GeoProviders` folder into the `Resources` folder of your project and then copy the `modules` folder into your project as highlighted in the following screenshot:



After copying the mentioned folders, you will need to click on your `tiapp.xml` file in Titanium Studio and add a reference to the `bencoding.basicgeo` module as shown in the following screenshot:



How to do it...

Once you have added native and CommonJS modules to your project, you next need to create your application namespaces in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  multiProvider :
    require('./GeoProviders/reverse_multi_geoprovider')
};
```

Adding providers

The following code snippet demonstrates how to add different GeoProviders to the `multiProvider` module by calling the `addProvider` method. It is advised to add the GeoProviders that you feel will best meet your requirements first since the `multiProvider` will execute them in the order they are added.

```
my.multiProvider.addProvider({
  key : 'demo',
  providerString: 'GeoProviders/geonames_geoprovider'
});

my.multiProvider.addProvider({
  providerString: 'GeoProviders/bencoding_geoprovider'
});

my.multiProvider.addProvider({
  providerString: 'GeoProviders/google_geoprovider'
});
```

Adding your purpose

Using location services on iOS requires a purpose or reason for the app to access the GPS. On the first request, this message will be presented to the user. The following code demonstrates how to add this purpose to the multitenant provider using the `addPurpose` method:

```
my.multiProvider.addPurpose('Demo of Geo Provider');
```



Android does not require the purpose be provided; in this case the purpose defined is not used when accessing the GPS.

Building the recipe UI

The following code snippet describes how to create the UI shown in this recipe's earlier screenshots. The first step is to create the `Ti.UI.Window` to which all visual elements will be attached.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Multi-Tenant Geo',
  barColor: '#000', fullscreen: false
});
```

Next, a `Ti.Map.View` is added to the `Ti.UI.Window`; this will be used to display a map pin with the device's current location and address details.

```
var mapView = Ti.Map.createView({
  top: 120, bottom: 0, width: Ti.UI.FILL,
  userLocation: false
});
win.add(mapView);
```

Lookup helper methods

Now perform the following steps:

1. The lookup object is designed to help format the results of the multitenant reverse geolocation component and present the address information in a graphical way to the recipe user.

```
var lookup = {
```

2. The `addToMap` method creates a map pin and adds the information to `Ti.Map.View`.

```
  addToMap: function(lat, lng, title) {
    var pin = Ti.Map.createAnnotation({
      latitude: lat, longitude: lng,
      title: title
    });
    mapView.addAnnotation(pin);
```

3. A region is created using the map pin coordinates and the `setLocation` function of `Ti.Map.View` is then called. This will zoom the map to the coordinates of the recently added pin.

```
  var region = {latitude: lat, longitude: lng,
    latitudeDelta: 0.04,
    longitudeDelta: 0.04};
  mapView.setLocation(region);
},
```

4. The `onSuccess` method is provided as the success callback when the `getCurrentAddress` method is called. The result of the `getCurrentAddress` method is provided to the `e` parameter.

```
onSuccess : function(e) {
  if(!e.found) {
    alert("Unable to find your location");
    return;
  }
}
```

5. The `getProvider` method is called to create a reference to the provider used to return the location results. This allows for the provider-specific `generateAddress` method to be used.

```
var provider = my.multiProvider.getProvider(
  e.provider.name);
var title = provider.generateAddress(e);
lookup.addToMap(e.latitude,e.longitude,title);
},
```

6. The `onError` method is provided as the error callback when the `getCurrentAddress` method is called. Error details are provided to the `e` parameter.

```
onError: function(e) {
  alert("Error finding your location");
}
};
```

Performing a multitenant reverse geolocation lookup

Now perform the following steps:

1. The final section of this recipe is to perform the multitenant lookup using the `getCurrentAddress` method of the `multiProvider` module.

```
var findButton = Ti.UI.createButton({
  title:'Find Current Location',
  left:10, right:10,top:70,height:40
});
win.add(findButton);
```

2. The `multiProvider` lookup is performed on the click event of `findButton`.

```
findButton.addEventListener('click',function(e) {
```

3. As a network connection is required for the reverse geolocation, the first step in the reverse geolocation process is to validate the network connection.

```
if(!Ti.Network.online) {
```

```
    alert("You must be online to run this recipe");  
    return;  
}
```

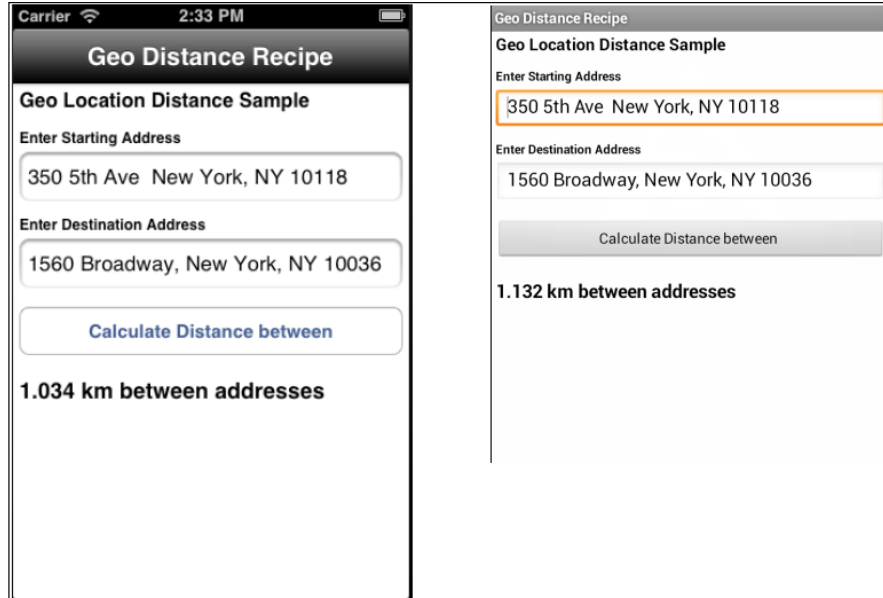
- Next, the `getCurrentAddress` method is called, and a success and error callback method is provided. The following code snippet demonstrates calling this method with the `lookup.onSuccess` and `lookup.onError` callback methods discussed earlier in this recipe.

```
my.multiProvider.getCurrentAddress(  
    lookup.onSuccess,lookup.onError);  
});
```

Calculating distance between addresses

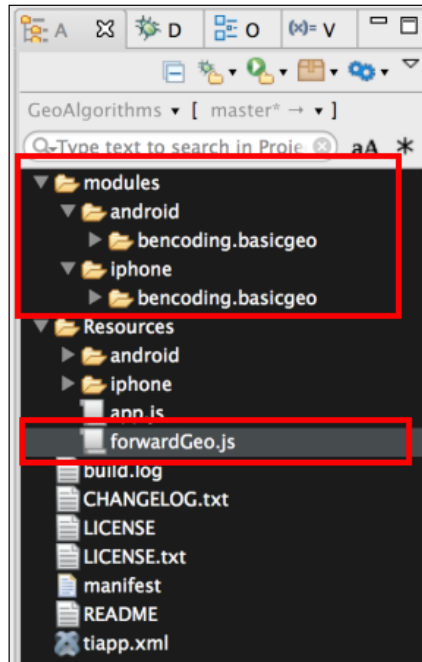
The use of geolocation services in enterprise apps is common. One of the most common geolocation requirements is to calculate the distance between two points. This is helpful in planning routes, determining mileage, forecasting delivery schedules, and more.

The following recipe demonstrates how to calculate the distance between two addresses. This distance measurement is done using a direct distance, not a routing calculation such as those used for walking or driving. The following screenshots illustrate this recipe running on both an iPhone and an Android device.

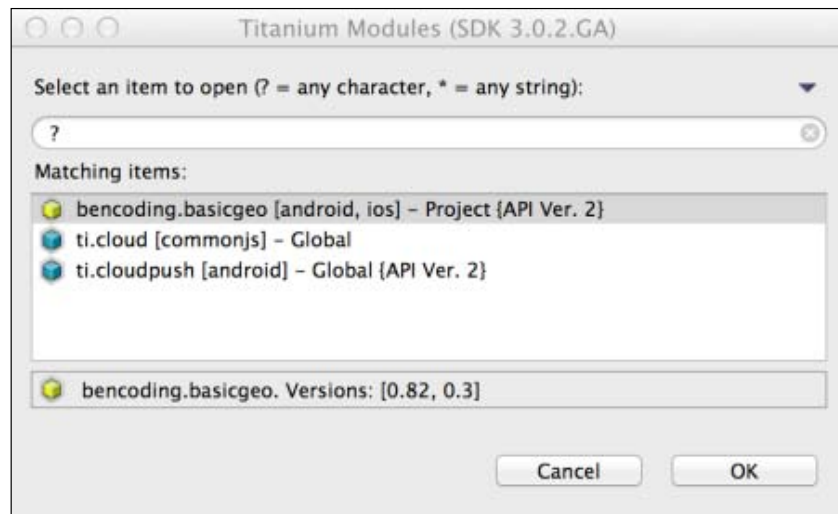


Getting ready

This recipe uses both CommonJS and native modules. These can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Simply copy the `forwardGeo.js` file into the `Resources` folder of your project and then copy the `modules` folder into your project as shown in the following screenshot:



After copying the file and folder mentioned here, you will need to click on your **tiapp.xml** file in Titanium Studio and add a reference to the `bencoding.basicgeo` module as shown in the following screenshot:



How to do it...

Once you have added native and CommonJS modules to your project, you next need to create your application namespaces in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  forward : require('forwardGeo')
};
```

Adding address information

The following `startAddress` and `endAddress` objects are added to the app namespace. These objects will be used to create the address information and coordinate state for this recipe.

```
my.startAddress = {
  needRefresh:true,lat:40.748433, lng:-73.985656,
  address:'350 5th Ave New York, NY 10118'
};

my.endAddress = {
  needRefresh:false, lat:40.75773, lng:-73.985708,
  address:'1560 Broadway, New York, NY 10036'
};
```

Building the recipe UI

The following code snippet describes how to create the UI shown in this recipe's earlier screenshots:

1. The first step is to create the `Ti.UI.Window` to which all visual elements will be attached.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Geo Distance Recipe',
  barColor:'#000',fullscreen:false
});
```

2. Next the `txtStartAddress` `Ti.UI.TextField` is created to allow the user to enter a starting address.

```
var txtStartAddress = Ti.UI.createTextField({
  hintText:'enter starting address',
  value: my.startAddress.address,
  height:40, left:5, right:5, top:55,
  borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtStartAddress);
```

3. Next the `txtEndAddress` `Ti.UI.TextField` is created to allow the user to enter a destination address.

```
var txtEndAddress = Ti.UI.createTextField({
  hintText:'enter destination address',
  value: my.endAddress.address,
  height:40, left:5, right:5, top:125,
  borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtEndAddress);
```

- The `findButton` `Ti.UI.Button` is then added to the `Ti.UI.Window` later in this recipe. This button will be used to perform the distance calculation.

```
var findButton = Ti.UI.createButton({
    title:'Calculate Distance between', height:40,
    left:5, right:5, top:180
});
win.add(findButton);
```

Distance and address methods

This recipe uses the `geo` object to perform distance and address lookup operations.

```
var geo = {
```

- The `distanceInUnits` uses the Haversine formula and computes the direct distance in kilometers or meters between two sets of coordinates.

```
distanceInUnits: function(lat1, lng1, lat2, lng2){
    var rOfEarth = 6371;
    var dLat = (lat2-lat1)*Math.PI/180;
    var dLon = (lng2-lng1)*Math.PI/180;
    var a = Math.sin(dLat/2) * Math.sin(dLat/2) +
    Math.cos(lat1*Math.PI/180) *
    Math.cos(lat2*Math.PI/180) *
    Math.sin(dLon/2) * Math.sin(dLon/2);
    var c = 2 * Math.asin(Math.sqrt(a));
    var distance = rOfEarth * c;
```

- If the distance is less than 1 kilometer, the unit returned is converted to meters.

```
return {
    distance: ((distance < 1) ?
    (distance * 1000) : distance),
    unit: ((distance < 1) ? 'm' : 'km')
};
},
```

- The `findLocations` method is used to obtain the coordinates for the addresses provided.

```
findLocations : function(callback){
```

- The `onFinish` function is the callback method provided to the `forwardGeo` function. The `e` parameter provides the starting and ending address coordinates.

```
function onFinish(e){
    if(!e.success){
        alert(e.message);
        return;
    }
}
```

The `forwardGeo` `e.start` and `e.end` results are assigned to the `my.startAddress` and `my.endAddress` properties. The callback method is then executed, so the distance calculation can be performed.

```
my.startAddress = e.start;
my.endAddress = e.end;
callback();
};
```

5. The `forwardGeo` method is called to obtain the coordinates for the `my.startAddress` and `my.endAddress` objects. As discussed earlier, the geolocation results are provided to the `onFinish` callback method as the following code snippet demonstrates:

```
my.forward.forwardGeo(my.startAddress,
my.endAddress,onFinish);
}
};
```

Finding the distance between the two addresses

When the user presses the `findButton` and the click event is fired, the recipe will perform a distance calculation between the two addresses entered.

```
findButton.addEventListener('click',function(e){
```

1. The first step in this process is to determine if forward geolocation is supported. The coordinates for each address is required for the distance calculation. A forward geolocation lookup is performed to obtain this information.

```
if(!my.forward.isSupported()){
alert('Forward Geocoding is not supported');
return;
}
```

- The `findDistance` method is used to make the distance calculation method call and format the provided results.

```
function findDistance(){
```

- The first step in this section of the recipe is to call the `distanceInUnits` method using the latitude and longitude information for each address.

```
var result = geo.distanceInUnits(  
my.startAddress.lat,my.startAddress.lng,  
my.endAddress.lat,my.endAddress.lng);
```

- Next the distance calculation results need to be formatted. If the results are in kilometres, the `distance` variable is rounded to the first three decimal places. If it is in meters, the full value will be displayed.

```
if(result.unit=='km'){  
    result.distance =  
    result.distance.toFixed(3);  
}  
distanceLabel.text = result.distance + " " +  
result.unit + " between addresses";  
};
```

- If either of the address information objects needs to be refreshed, the `geo.findLocation` method is called. The `findDistance` method is provided as the callback method to the `findLocations` function so that the distance calculation can be performed after the coordinates are obtained.

```
if(my.startAddress.needRefresh ||  
my.endAddress.needRefresh){  
    geo.findLocations(findDistance);  
}else{
```

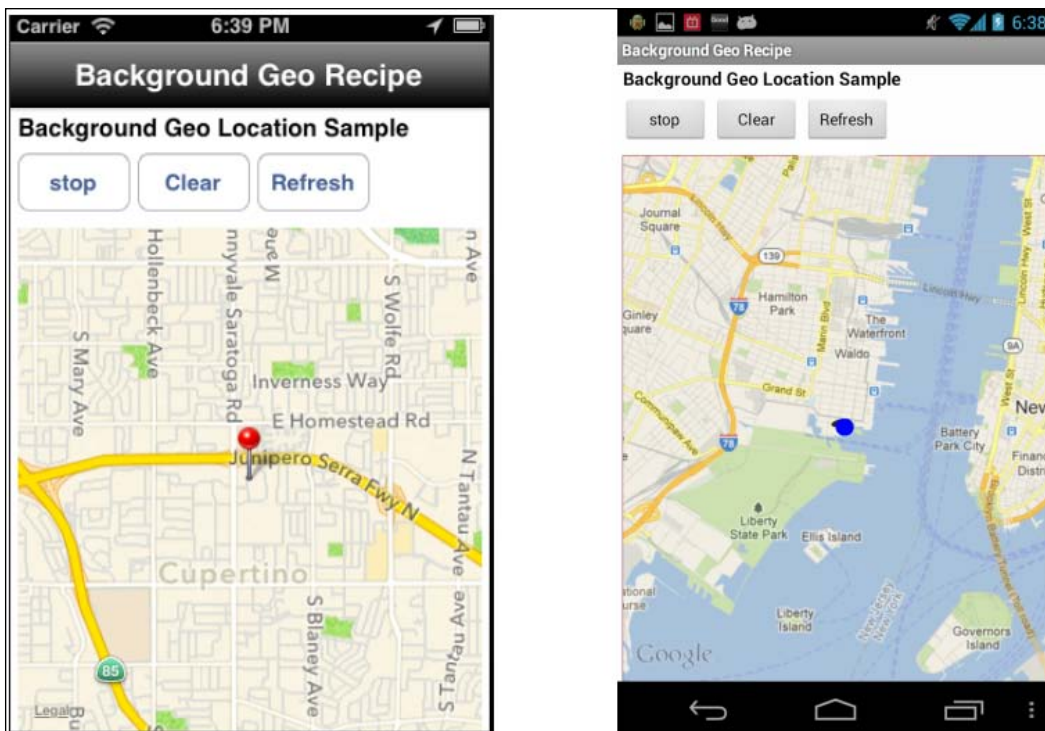
- If the address information objects do not need to be refreshed, the `findDistance` method is called directly to perform the distance calculation.

```
    findDistance();  
}  
});
```

Background geolocation management

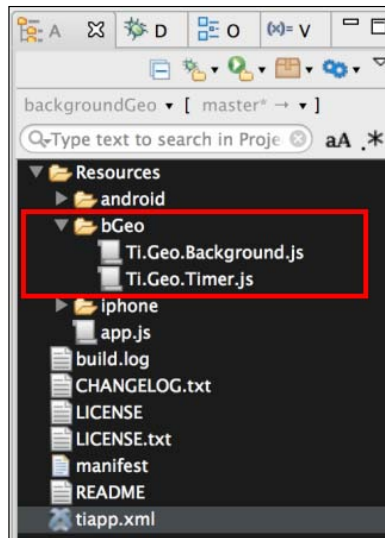
Background geolocation is an important feature of many enterprise applications. The ability to monitor the device's whereabouts while in the background is a powerful feature that can be used for a wide range of activities from personal security to mileage tracking.

The following recipe demonstrates how to use the `Ti.Geolocation` framework to enable background geolocation monitoring. The following screenshots illustrate this recipe running on both an iPhone and an Android device.



Getting ready

This recipe uses a series of CommonJS modules. These modules can be downloaded from the source code provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Simply copy the `bGeo` folder into the `Resources` folder of your project as shown in the following screenshot:



Updating your tiapp.xml file

This recipe requires a few updates to your project's `tiapp.xml` file. The first update is to provide background geolocation support for iOS devices. The following highlighted `UIBackgroundModes` section illustrates the entry required by this recipe:

```
<ios>
  <min-ios-ver>5.0</min-ios-ver>
  <plist>
    <dict>
      <key>UIBackgroundModes</key>
      <array>
        <string>location</string>
      </array>
      <key>NSLocationUsageDescription</key>
      <string>Demo Geo App</string>
    </dict>
  </plist>
</ios>
```


This recipe uses an Android service as a keep alive. The following highlighted section is required for the recipe to create an internal service:

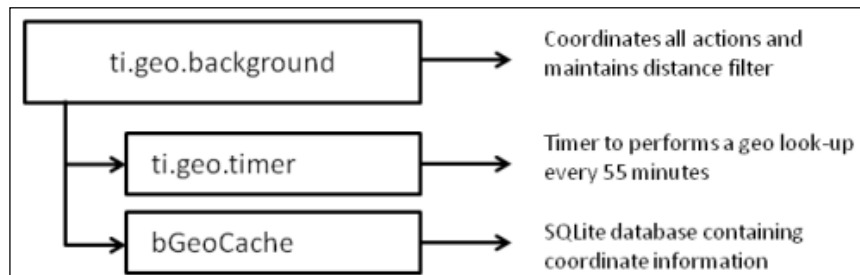
```
<android xmlns:android=
"http://schemas.android.com/apk/res/android">
  <services>
    <service url="bGeo/Ti.Geo.Timer.js" type="interval"/>
  </services>
</android>
```

How to do it...

The `Ti.Geo.Background` CommonJS module utilizes both location manager distance filtering and a keep-alive geo timer. This ensures that a location is recorded both when the device travels a specific distance, or a specified period of time elapses. The `Ti.Geo.Background` module manages all geolocation operations and maintains a distance filter so that coordinates are recorded when the device moves pass a specific threshold distance.

The `Ti.Geo.Timer` performs two activities. First provides a keep-alive loop required to keep an iOS application active, and second, on a scheduled interval the service records the device's current coordinates. This ensures that coordinates are recorded even if the individual hasn't moved.

The following diagram illustrates the interaction between the different `Ti.Geo.Background` components:



Namespace and app setup

Once you have added native and CommonJS modules to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the modules into your code as the following code snippet demonstrates:

```
var my = {
  bGeo : require('bGeo/Ti.Geo.Background'),
  isAndroid : Ti.Platform.osname === 'android',
  session:{started:false}
};
```

Background location options

The `Ti.Geo.Background` module provides a series of optional configuration parameters that allow you to tailor the module to your needs. The first configuration parameter is `purpose`. This is used by iOS when presenting the location services access prompt to your users.

```
my.bGeo.purpose = "Demo Background Recipe";
```

1. The `distanceFilter` is a value in meters on how often you wish to have the location manager fire an alert that the location has changed. The following sample is set to 100 and will fire a location-changed event every time the user travels more than 100 meters.

```
my.bGeo.distanceFilter = 100;
```

2. The `trackSignificantLocationChange` setting indicates that the significant location-change tracking should be used on iOS. This method of geolocation reduces battery impact by only firing events when a network change such as a cell tower switch is performed.

```
my.bGeo.trackSignificantLocationChange = true;
```

3. The `minAge` configuration is a threshold of the minimum frequency, in minutes; you wish to receive location updates. In the following example, you will not receive updates any more frequently than every 3 minutes, no matter the distance the individual is moving.

```
my.bGeo.minAge = 3;
```

4. The `maxAge` configuration is a threshold of the maximum amount of time in minutes you wish to go without receiving an update. This is also the threshold used by the `Ti.Geo.Timer` to perform a coordinate lookup.

```
my.bGeo.maxAge = 30;
```

Building the recipe's UI

The following code snippet describes how to create the UI shown in this recipe's earlier screenshots:

1. The first step is to create the `Ti.UI.Window` upon which all visual elements will be attached.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Background Geo Recipe',
  barColor: '#000', fullscreen: false
});
```

2. The next step is to add a `Ti.Map.View` to the recipe's `Ti.UI.Window`. This will be used to display the coordinates collected while `Ti.Geo.Background` is running.

```
var mapView = Ti.Map.createView({
  top:80, bottom:0, left:5, right:5, userLocation:false
});
win.add(mapView);
```

3. The `startStopButton` is then added to the `Ti.UI.Window`. This button will be used to start and stop the `Ti.Geo.Background` process.

```
var startStopButton = Ti.UI.createButton({
  title:(my.bGeo.active()) ? 'stop' : 'start',
  top:30, height:40, left:5, width:75
});
win.add(startStopButton);
```

4. The `clearButton` is then added to the `Ti.UI.Window`. This button will be used to remove all coordinate information recorded by the `Ti.Geo.Background` process.

```
var clearButton = Ti.UI.createButton({
  title:'Clear', top:30, height:40, left:85, width:75
});
win.add(clearButton);
```

5. The `refreshButton` is then added to the `Ti.UI.Window`. This button will be used to refresh the `Ti.Map.View` with the coordinates recorded by the `Ti.Geo.Background` process.

```
var refreshButton = Ti.UI.createButton({
  title:'Refresh', top:30, height:40, left:165, width:75
});

win.add(refreshButton);
```

The recipe's assistant methods

This recipe implements an `assistant` object, which contains helper functions to work with and display coordinates collected by the `Ti.Geo.Background` module.

```
var assistant = {
```

1. The `addToMap` is the first method in the `assistant` object. This method adds a map pin for each of the coordinate points collected by the `Ti.Geo.Background` module.

```
  addToMap : function(e) {
    var pin = Ti.Map.createAnnotation({
      latitude:e.latitude,
      longitude:e.longitude
    });
```

```

    mapView.addAnnotation(pin);
    var region = {latitude:e.latitude,
    longitude:e.longitude,
    latitudeDelta:0.04, longitudeDelta:0.04};
    mapView.setLocation(region);
  },

```

2. The next method in the assistant object is the `locationChangeCallback` method. This method is provided as the callback method for the `Ti.Geo.Background` module's change event. The change coordinates are provided to the method's `e` parameter. The `locationChangeCallback` method then calls the `addToMap` method to display the newly gathered coordinate information.

```

    locationChangeCallback : function(e) {
        assistant.addToMap(e);
    },

```

3. The final method in the assistant object is the `locationErrorCallback` method. This method is provided as the callback method for the `Ti.Geo.Background` module's error event. Error information is provided to the method's `e` parameter.

```

    locationErrorCallback : function(e) {
        alert('Error due to ' + e.message);
    }
};

```

Geolocation events

The `Ti.Geo.Background` module has several events. The events used in this recipe are detailed in this section. The `change` event is the primary method used in this recipe. This event is fired whenever a location change is generated. The following example demonstrates how to subscribe to this change event providing the callback method `assistant.locationChangeCallback`:

```

my.bGeo.addEventListener('change',
assistant.locationChangeCallback);

```

The `error` event is also used in this recipe to provide an alert to the user when an error has occurred in the module. The following example demonstrates how to subscribe to the error event providing the callback method `assistant.locationErrorCallback`:

```

my.bGeo.addEventListener('error',
assistant.locationErrorCallback);

```

Background button events

This recipe uses a series of buttons to demonstrate how to call the `Ti.Geo.Background` module's methods. The `startStopButton` click event demonstrates how to start and stop the `Ti.Geo.Background` process.

```
startStopButton.addEventListener('click', function(e) {
```

1. If the module is already active, the recipe will toggle the status to `off` and stop the module for recording coordinates.

```
    if(my.bGeo.active() {  
        my.bGeo.stop();  
    }else{
```

2. If the module is off, the recipe will toggle the status to `on` and start the module for recording coordinates.

```
        my.bGeo.start();  
    }
```

3. The `startStopButton` title is updated to refresh the current status of the module.

```
        startStopButton.title=(my.bGeo.active() ?  
        'stop' : 'start');  
    });
```

4. The `click` event of `refreshButton` is used to reload the recorded coordinates to display in the `Ti.Map.View`.

```
refreshButton.addEventListener('click', function(e) {
```

5. First all map annotations are removed.

```
    mapView.removeAllAnnotations();
```

6. The `readCache` method is then called to return an array containing all of the recorded coordinates.

```
    var results = my.bGeo.readCache();
```

7. The snippet then loops through the coordinate array using the `assistant.addToMap` method to create map pins on the recipe's `Ti.Map.View`.

```
    for (iLoop=0;iLoop < results.length;iLoop++){  
        assistant.addToMap(results[iLoop]);  
    }  
});
```

- The `click` event of `clearButton` is used to remove all recorded coordinate information and clear the `Ti.Map.View` of all annotations as the following code snippet demonstrates:

```
clearButton.addEventListener('click',function(e){
    my.bGeo.clearCache();
    mapView.removeAllAnnotations();
});
```

iOS app-level events

The iOS platform does not allow for background services to run while the app is in the foreground. The following block of code demonstrates how to handle this iOS specific scenario:

- First check to ensure the recipe is running on an iOS device.

```
if(!my.isAndroid){
```

- Next an application-level listener is created on the `resumed` event. This will be fired when the app is placed in the foreground.

```
Ti.App.addEventListener('resumed',function(e){
```

- When the app is moved into the foreground, the recipe checks if the `Ti.Geo.Background` module is active. If active, the module's `paused` method must be called to disable the `Ti.App.iOS.BackgroundService`, while leaving the location manager active.


```
    if(my.bGeo.active()){
        my.bGeo.paused();
    }
});
```

- The next step in managing background services on iOS is to add an application-level event listener to the `paused` event. The `paused` event will be fired when the app is placed in the background.

```
Ti.App.addEventListener('paused',function(e){
```

- The next snippet demonstrates how to restart the background service, if the `Ti.Geo.Background` module is active.

```
    if(my.bGeo.active()){
        my.bGeo.restart();
    }
});
```

 The restart method must be called when the app is paused, if you wish to continue collecting coordinates in the background.

6. The following code snippet demonstrates how to stop the `Ti.Geo.Background` module when the application is closed. This also provides a clean shutdown to the background processes and avoids iOS terminating them after approximately 10 minutes.

```
Ti.App.addEventListener('close', function(e) {  
    my.bGeo.stop();  
});  
}
```

See also

- ▶ The `Ti.Geo.Background` module was used in this recipe to provide cross-platform background location services. For licensing, source code, and to learn more about this project please visit <https://github.com/benbahrenburg/Ti.Geo.Background>.

7

Threads, Queues, and Message Passing

In this chapter we will cover:

- ▶ Queuing multiple downloads
- ▶ Launching one app from another
- ▶ Cross-platform URL schemes
- ▶ Opening your Android app on `BOOT_COMPLETED`
- ▶ iOS multithreading using Web Workers

Introduction

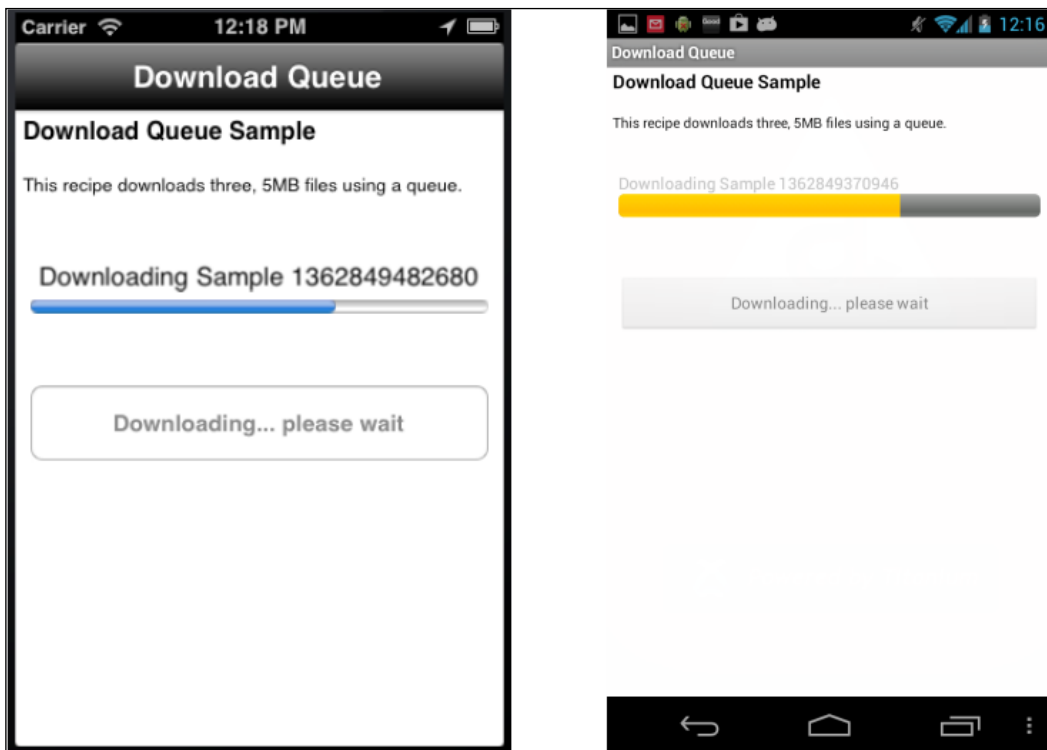
Size, complexity, and volume are common issues encountered in developing and maintaining enterprise apps. Titanium provides support for several common patterns used to separate operational responsibility patterns.

This chapter discusses how to use message passing, queuing, and multithreaded capabilities in Titanium. These step-by-step recipes can then be used to incorporate these design principles into your existing Titanium enterprise application.

Queuing multiple downloads

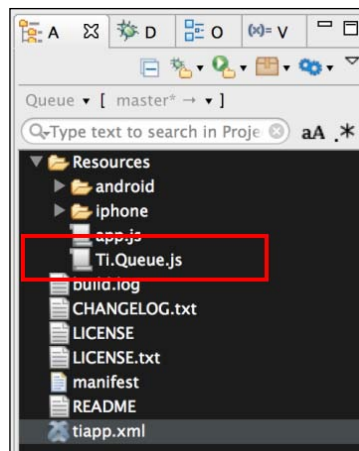
It is commonplace for enterprise apps to require the need to download documents and content from your organization's file servers. Use of queuing is helpful in this scenario to ensure order and delivery while avoiding pitfalls often associated with spawning multiple asynchronous requests. This recipe uses the `Ti.Queue` CommonJS module to create a persistent, named queue to perform file downloads.

To demonstrate how to implement a persistent queue, this recipe will download to your device 5 MB sample files from Github. The following screenshots illustrate what this recipe looks like while running on both the iPhone and Android devices.



Getting ready

This recipe uses the `Ti.Queue` CommonJS module. This module and other code assets can be downloaded from the source provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing these in your project is straightforward. Simply copy the `Ti.Queue.js` file into your project as shown in the following screenshot:



Network connection

This recipe requires a network connection to download files from Github. Please make sure the device or simulator has proper network connectivity.

How to do it...

Once you have added the `Ti.Queue` module to your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  qMod : require('Ti.Queue'),
  jobInfo:{total:0,completed:0}
};
```

Creating the recipe's UI

The following steps outline how to create the UI used in this recipe:

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Download Queue',
  barColor:'#000',layout:'vertical',fullscreen:false
});
```

- Next, a `Ti.UI.ProgressBar` is added to the `Ti.UI.Window`. This will be used to display the status of the queue.

```
var progress = Ti.UI.createProgressBar({
  top:30, height:50,min:0,max:3, value:0, left:10,
  right:10, message:'Tap button to start download'
});
win.add(progress);
```

- The `show` method is called immediately on our `Ti.UI.ProgressBar` so that it will be displayed on the `Ti.UI.Window` correctly.

```
progress.show();
```

- Next, a `Ti.UI.Button` is added to the `Ti.UI.Window`. This control will be used to trigger the queue download logic.

```
var downloadButton = Ti.UI.createButton({
  title:'Run Download Queue', top:40,
  left:10, right:10, height:50,
});
win.add(downloadButton);
```

Creating a queue

The next step in the recipe is to create a named queue as demonstrated in the following code snippet. By default, any named queue is persistent and will store all jobs between sessions.

```
//Create a new version of the queue
var queue = new my.qMod("demo");
```



If a name is not provided when creating a new queue, the queue will not save jobs between app restarts.

Adding jobs to the queue

After the queue has been created, the next step is to create a series of jobs for the queue to manage. The following code block creates three download jobs for the queue to manage.

- The first job is created to download a ZIP file from Github. A timestamp is used to provide a unique name.

```
var sample1Name = new Date().getTime();
queue.enqueue({
  title:'Sample ' + sample1Name,
  url:"https://github.com/benbahrenburg/
Ti.Queue/blob/master/5MB.zip",
  downloadPath:Ti.Filesystem.applicationDataDirectory +
```

```

        sample1Name + '.zip',
        attempts:0
    });

```

2. Again, a second job is created to download a ZIP file from Github. A timestamp is used to provide a unique name.

```

var sample2Name = new Date().getTime();
queue.enqueue({
    title:'Sample ' + sample2Name,
    url:"https://github.com/benbahrenburg/
    Ti.Queue/blob/master/5MB.zip",
    downloadPath:Ti.Filesystem.applicationDataDirectory +
    sample2Name + '.zip',
    attempts:0
});

```

3. Again, a third job is created to download a ZIP file from Github. A timestamp is used to provide a unique name.

```

var sample3Name = new Date().getTime();
queue.enqueue({
    title:'Sample ' + sample3Name,
    url:"https://github.com/benbahrenburg/
    Ti.Queue/blob/master/5MB.zip",
    downloadPath:Ti.Filesystem.applicationDataDirectory +
    sample3Name + '.zip',
    attempts:0
});

```

Recipe's assistant functions

The `assist` object is used to manage the download process. The following is a discussion on how the `assist` object works and can be leveraged within your app:

```

var assist = {

```

1. The `progressSetup` method is used to restart the progress bar when a new download system has been initialized.

```

    progressSetup : function(min,max) {
        progress.min = min;
        progress.max = max;
        progress.value = 0;
        progress.message = 'Starting download';
        downloadButton.title = "Downloading... please wait";
    },

```

2. The `updateProgress` method is used to update the progress bar information and alert the user to the status of the overall download job.

```
updateProgress : function(value,text){
    progress.value = value;
    progress.message = text;
},
```

3. The `whenFinished` method is called after all `Ti.Queue` jobs have finished or generated errors. This method is used to update UI to alert the user about the queued job which has been processed.

```
whenFinish : function(){
    downloadButton.text = "Tap button to start download";
    alert('Finished Download');
    downloadButton.enabled = true;
},
```

4. The `next` method is used to process the next job in the `Ti.Queue`.

```
next : function(){
```

5. This step in the `next` method is to check if there are any jobs available. This is done by calling the `getLength` method on the queue to retrieve the number of jobs currently stored in the queue.

```
    if(queue.getLength() == 0){
```

6. If there are no jobs remaining in the queue, the `whenFinish` method is called and the download process is exited.

```
        assist.updateProgress(my.jobInfo.total,
        'Download Completed');
        assist.whenFinish();
        return;
    }
```

7. If a job is available in the queue, the next step is to update the `progressValue` to show the current processing status as shown in the following code snippet.

```
    var progressValue =
    (my.jobInfo.total - queue.getLength());
```

8. The next step is to call the `peek` method on the queue. This allows us to fetch the next item in the queue without popping it from the queue itself. This is used to write the information to the Titanium Studio console for debugging purposes.

```
var pkItem = queue.peek();
Ti.API.info('Peek value: ' + JSON.stringify(pkItem));
```

9. The next step is to call the `dequeue` method on the queue. This function pops the next item from the queue and returns the item. In the following code snippet, the `dequeue` method is called to provide the next queue job to the `item` variable.

```
var item = queue.dequeue();
```

10. The `updateProgress` method is then called to alert the user to the download progress.

```
assist.updateProgress(progressValue, 'Downloading ' +
item.title);
```

11. Next the `download` method is called to start the download process. To create a recursive function, the `next` method is provided as a callback argument to the `download` method.

```
assist.download(item, assist.next);
```

```
},
```

12. The `download` method contains all logic needed to download the queued job from Github.

```
download :function(item, callback) {
```

13. The first step in downloading the file from Github is to create `Ti.Network.HTTPClient`.

```
var done = false;
var xhr = Ti.Network.createHTTPClient();
xhr.setTimeout(10000);
```

14. The `onload` callback is fired when the `Ti.Network.HTTPClient` receives a successful response.

```
xhr.onload = function(e) {
  if (this.readyState == 4 && !done) {
    done=true;
```

15. If the response does not provide a 200 status code, an error is generated.

```
if(this.status!==200){
  throw('Invalid http reply status code:' +
this.status);
return;
}
```

16. If the proper status code is returned, a `Ti.FileSystem` object is then created and the `responseData` is saved to the provided output path.

```
var saveToFile =
Ti.FileSystem.getFile(item.downloadPath);
if(saveToFile.exists()){
  saveToFile.deleteFile();
}
saveToFile.write(this.responseData);
saveToFile = null;
```

17. After the `responseData` has been persisted to the filesystem, the callback method is triggered. This allows for the recipe to recursively loop through the queue.

```
callback();
}
};
```

18. The `onerror` callback is triggered when the `Ti.Network.HTTPClient` receives an error.

```
xhr.onerror = function(e){
```

19. When an error is received, the provided job is requeued for another attempt, by calling the `requeue` method on the `assist` object.

```
assist.requeue(item,callback)
};
```

20. The download progress is started by calling the `open` and `send` methods on the `Ti.Network.HTTPClient` using the queue item's `url` details as demonstrated in the following code snippet:

```
xhr.open('GET', item.url);
xhr.send();
},
```

21. The `requeue` method is used to add an item back into the queue. The recipe is designed to provide three attempts at downloading before the job is considered a permanent error.

```
requeue :function(item,callback){
  Ti.API.info('requeue is called on download fail. ');
  Ti.API.info('Allowed to retry 3 times');
}
```

22. The item's `attempt` property is checked to determine if the job has been attempted more than three times.

```
if(item.attempts > 3){
```

23. If the job has been attempted more than three times, the item is not readded to the queue.

```
Ti.API.info('Max removed from queue.')
}else{
```

24. If the job has errored, less than three times, the `attempts` property is incremented and the item is added to the queue again.

```
    item.attempts++;
    queue.enqueue(item);
}
```

25. Finally the `callback` method is fired, moving the download process to the next step in its lifecycle.

```
    callback();
}
};
```

Start downloading

When the `downloadButton` is pressed the recipe begins to process the queued jobs.

```
downloadButton.addEventListener('click', function(e) {
```

1. The first step in the download process is to check if the recipe has a network connection. If the network is not available, the recipe will alert the user to exit the process.

```
    if(!Ti.Network.online){
        alert('This recipe requires a network');
        return;
    }
```

2. The next step in the download process is to disable the `downloadButton`. This avoids the user from processing the button again, once the job has started.

```
downloadButton.enabled = false;
```

3. The `my.jobInfo` object is then updated with the current count and status information. This will be used to track the overall download status.

```
my.jobInfo.total = queue.getLength();
my.jobInfo.completed = 0;
```

4. Next the `processSetup` method is called to initialize the `Ti.UI.ProgressBar` with the correct `min` and `max` values.

```
assist.progressSetup(my.jobInfo.completed,
my.jobInfo.total);
```


5. Finally, the `next` method is called on the `assist` object. This begins the download process and creates a recursive loop that will run until the queue is empty.

```
assist.next();  
  
});
```

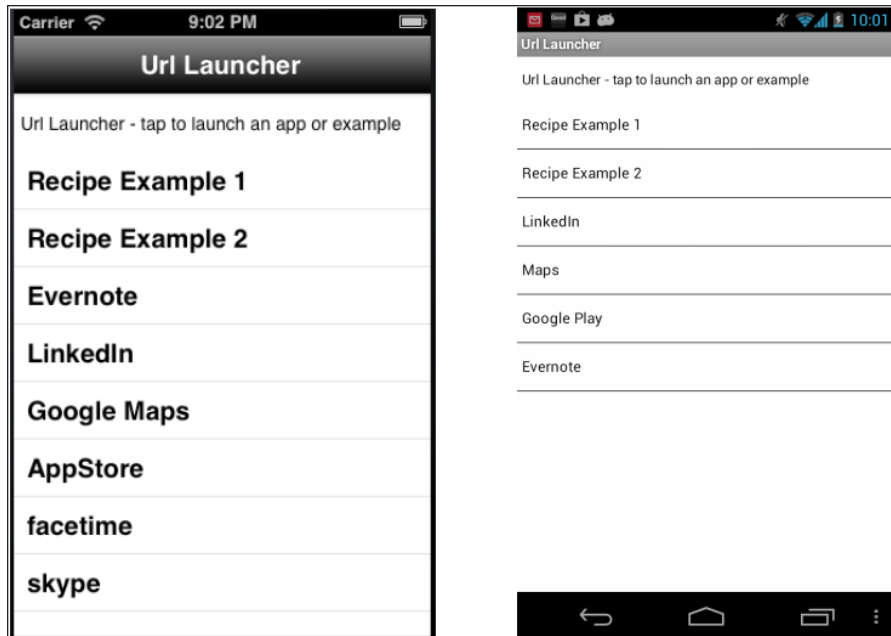
See also

- ▶ To learn more about the `Ti.Queue` module used in this recipe, please visit <https://github.com/benbahrenburg/Ti.Queue>.

Launching one app from another

A majority of mobile enterprise apps are designed around a specific task such as time reporting. Through the use of URL schemes or Android intent filters you can open and communicate between different apps on the user's device. For example, you can provide an option for your organization's time-reporting app to open the expense app when an employee is traveling and needs to record additional information.

This recipe demonstrates how to launch different apps using the native platform's integration pattern. The following screenshot illustrates this recipe running on both an iOS and Android device:

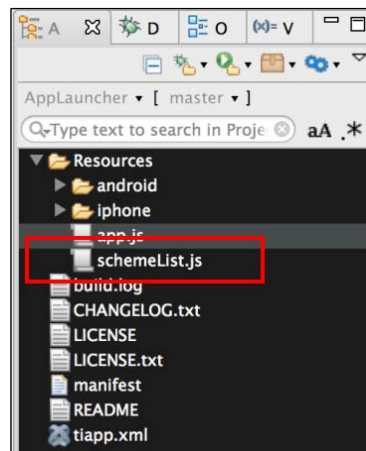




This recipe must be run on a device to fully experience all features. A device is required as the simulator or emulator do not allow apps from the different app stores to be run.

Getting ready

This recipe uses the `schemaList` CommonJS module. This module and other code assets can be downloaded from the source provided by the book. Installing these in your project is straightforward. Simply copy the `schemaList.js` file into your project as shown in the following screenshot:



Network connection

This recipe requires that the following apps are installed on your device:

- ▶ LinkedIn
- ▶ Evernote
- ▶ Google Maps (on iOS and Android)
- ▶ The URL Scheme example recipe discussed later in this chapter

How to do it...

After adding the `schemaList` module, you will need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace  
var my = {
```

```
schemes : require('schemeList'),
isAndroid : Ti.Platform.osname === 'android'
};
```

iOS updating tiapp.xml

Adding the ability to allow other iOS apps to launch your apps is straightforward. The following code snippet demonstrates how to add a custom scheme named, `bencoding-linkLauncher` to this recipe. This allows any iOS app to launch this sample in a similar fashion as discussed later in this recipe.

```
<ios>
  <plist>
    <dict>
```

The `CFBundleURLTypes` node is an array of dictionaries describing the URL schemes supported by the bundle. Under this node, you will list all of the URL and scheme names your app will respond to.

```
  <key>CFBundleURLTypes </key>
  <array>
    <dict>
```

The `CFBundleURLName` is the key that contains the abstract name for this URL type. This is the main way to refer to a particular type. To ensure uniqueness, it is recommended that you use a Java-package style identifier.

```
    <key>CFBundleURLName </key>
    <string>bencoding-linkLauncher</string>
```

The `CFBundleURLSchemes` is the key that contains an array of strings, each of which identifies a URL scheme handled by this type.

```
    <key>CFBundleURLSchemes</key>
    <array>
      <string>bencoding-linkLauncher</string>
    </array>
  </dict>
</array>
</dict>
</plist>
</ios>
```

Creating an app launch list

The `schemeList` CommonJS module, added to the `my.scheme` property provides a list of iOS and Android scheme examples. The following sections describe how schemes are created on both platforms.

iOS scheme list

On iOS, a URL scheme works just like a URL on the web. The only difference is the app's registered URL scheme is used as the protocol. The following steps provide detailed examples of the specific URL formats available.

1. The `getiOSList` provides an array of objects designed to be bound to a `Ti.UI.TableView` and later used to launch other apps.

```
exports.getiOSList = function(){
  var apps = [];
```

2. The most basic type of URL scheme is simply the `CFBundleURLName` of the app. This is shown in the following examples for Evernote and LinkedIn.

```
apps.push({
  title:'Evernote',
  url:'evernote://',
  errorMessage:'Check Evernote is installed'
});
```

```
apps.push({
  title:'LinkedIn',
  url:'linkedin://',
  errorMessage:'Check LinkedIn is installed'
});
```

3. More complex URL schemes can be used, such as the following example showing a URL scheme including the about route.

```
apps.push({
  title:'Recipe Example 1',
  url:'bencoding-linkrecipe://about',
  errorMessage:'Check Link Receiver Recipe is installed'
});
```

4. URL schemes can also include query parameters similar to a web page, as demonstrated in the following code snippet:

```
apps.push({
  title:'Recipe Example 2',
  url:
    'bencodinglinkrecipe: //' +
    'login?user=chris&token=12345',
  errorMessage:'Check Link Receiver Recipe is installed'
});

return apps;
};
```

Android scheme list

Android has the ability to launch applications in a variety of ways. This recipe focuses on how to use a URL scheme similar to iOS and intents for app integration.

The `getAndroidList` provides an array of objects designed to be bound to a `Ti.UI` `TableView` and later used to launch other apps.

```
exports.getAndroidList = function(){
  var apps = [];
```

1. Similar to iOS apps, you can use a route base URL to launch a third-party app. The following code snippet shows how to construct a URL with an about route.

```
apps.push({
  title:'Recipe Example 1',
  type:'url',
  color:'#000',
  url:'bencoding-linkrecipe://com.bencoding/about',
  errorMessage:'Check Link Receiver Recipe is installed'
});
```

2. Just as with web pages, you can build more complex URLs with query string parameters. The following snippet demonstrates how to call a login screen while passing query string information.


```
apps.push({
  title:'Recipe Example 2',
  type:'url',
  color:'#000',
  url:
    'bencoding-linkrecipe://' +
    'login?user=chris&token=12345',
  errorMessage:'Check Link Receiver Recipe is installed'
});
```

3. A more preferred way to launch apps on Android is to use intents. The following example shows how to create an intent to launch the LinkedIn app.

```
var linkedInIntent = Ti.Android.createIntent({
  action: Ti.Android.ACTION_SEND,
  packageName : 'com.linkedin.android',
  className:
    'com.linkedin.android.home.UpdateStatusActivity'
});
```


- Once the LinkedIn intent has been created, it is added to the app's array with a type of intent. This will later be used to launch the LinkedIn app.

```
apps.push({
    title: 'LinkedIn',
    type: 'intent',
    color: '#000',
    intentObject : linkedInIntent
});
```

 The LinkedIn app must be installed on the user's device in order to have the intent launch the app. If the app is not installed, an exception will be generated.

- A more preferred way to launch apps on Android is to use intents. The following example shows how to create an intent to launch the Evernote app.

```
var everNoteIntent = Ti.Android.createIntent({
    action: "com.evernote.action.CREATE_NEW_NOTE"
});
```

 The Evernote app must be installed on the user's device in order to have the intent launch the app. If the app is not installed, an exception will be generated.

- Once the Evernote intent has been created, it is added to the app's array with a type of intent. This will later be used to launch the Evernote app.

```
apps.push({
    title: 'Evernote',
    type: 'intent',
    color: '#000',
    intentObject : everNoteIntent
});

return apps;
};
```

The recipe's UI

This section of the recipe is the UI used to launch the third-party apps.

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'App Launcher',
  barColor: '#000', layout: 'vertical', fullscreen: false
});
```

2. Next the `schemeList` CommonJS module is called to return a list of the applications to launch.

```
var schemaData = ((my.isAndroid) ?
my.schemes.getAndroidList() : my.schemes.getiOSList())
```

3. The `schemaData` object is then formatted and bound to a `Ti.UI.TableView` for display.

```
var tableView = Ti.UI.createTableView({
  data : my.schemes.format(schemaData),
  top:10, bottom:0, width:Ti.UI.FILL
});
win.add(tableView);
```

The final section of this recipe demonstrates how to launch third-party apps using the list displayed in the `Ti.UI.TableView`.

1. A `click` event is added to the `Ti.UI.TableView`. This event will be fired when the user taps on a row in the `Ti.UI.TableView`.

```
tableView.addEventListener('click', function(e) {
```

2. Once the event is fired, the first step is to check the platform on which the recipe is running.

```
  if(my.isAndroid){

    try{
```

3. If running under Android, a check must be performed to determine if the launch type is an intent or URL.

```
    if(e.rowData.type == "intent"){
```

4. If the launch type is an intent, `Ti.Android.currentActivity.startActivity` is called using the provided intent as shown in the following snippet. This will launch the third-party app if it is installed on the user's device.

```
Ti.Android.currentActivity.startActivity(
    e.rowData.intentObject);
}else{
```

5. If the launch type is a URL, the `Ti.Platform.openURL` method is used to open the app using the provided `url` property as shown in the following snippet. This will launch the third-party app if it is installed on the user's device.

```
Ti.Platform.openURL(e.rowData.url);
}
}catch(err){
    alert(e.rowData.errorMessage);
}

}else{
```



If the app is not installed, an exception will be generated.

6. The iOS platform launches third-party apps using a URL scheme similar to how web pages are loaded using the `Ti.Platform.openURL` method. The following snippet demonstrates how to launch a third-party app using the `url` property provided by the `schemeList` CommonJS module:

```
if(Ti.Platform.canOpenURL(e.rowData.url)){
    Ti.Platform.openURL(e.rowData.url);
}else{
    alert(e.rowData.errorMessage);
}
}
});
win.open({modal:true});
```

See also

- ▶ To learn more about using Android intents in Titanium, visit *Forging Titanium Episode 9* at <http://developer.appcelerator.com/blog/2011/10/forging-titanium-episode-9-android-intent-cookbook.html>.

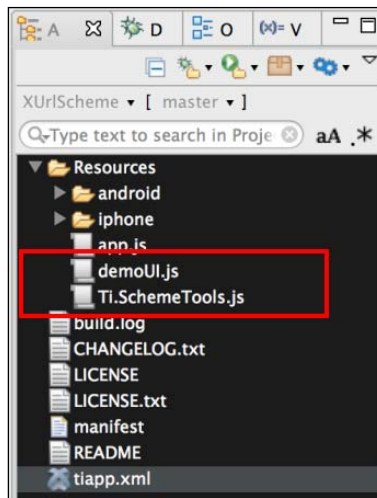
Cross-platform URL schemes

URL schemes on iOS and intent filters on Android provide an open integration point for you to expose functionality in your apps to others. This is particularly helpful if you are building a suite of enterprise apps such as a separate mileage and expense app and want to allow for integration points between them.

This recipe demonstrates how to create a cross-platform URL scheme within your Titanium Enterprise app. We will illustrate how to use this open integration point to access functionality within your Titanium app.

Getting ready

This recipe uses the `demoUI` and `Ti.SchemeTools` CommonJS libraries to help manage and demonstrate how to create cross-platform URL schemes. This module and other code assets can be downloaded from the source provided by the book. To install these modules, simply copy them to the `Resources` folder of your Titanium project as demonstrated in the following screenshot:



AppLauncher requirement

Another requirement of this recipe is the `AppLauncher` app, which was created in the *Launching one app from another* recipe discussed earlier in this chapter. This app will be used to launch the different URL examples contained within this recipe.

How to do it...

After adding the `demoUI` and `Ti.SchemeTools` modules, you will need to create your application namespaces and use `require` to import the module into your `app.js` as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  isAndroid : Ti.Platform.osname === 'android',
  scheme : require('Ti.SchemeTools'),
  ui : require('demoUI')
};
```

iOS updating tiapp.xml

To add the ability for other apps to launch this recipe, you must make a few updates to the `tiapp.xml` file. The following steps discuss how to create the `bencoding-linkrecipe` custom URL scheme:

1. First, open your project's `tiapp.xml` file and make the following changes to the `ios` node:

```
<ios>
  <plist>
    <dict>
```

2. Next, add the `CFBundleURLTypes` node. This is an array of dictionaries describing the URL schemes supported by the bundle. Under this node, you will list all of the URL and scheme names your app will respond to.

```
    <key>CFBundleURLTypes</key>
    <array>
      <dict>
```

3. Then add the `CFBundleURLName` key. This key contains the abstract name for this URL type. This is the main way to refer to a particular app. To ensure uniqueness, it is recommended that you use a Java-package style identifier.

```
      <key>CFBundleURLName </key>
      <string>bencoding-linkrecipe</string>
```

4. Finally, add the `CFBundleURLSchemes` keys. These keys contain an array of strings, each of which identifies a URL scheme handled by this type. The following snippet shows URLs for the login, about, and root activities of the app:

```
<key>CFBundleURLSchemes</key>
  <array>
    <string>bencoding-linkrecipe</string>
    <string>bencoding-linkrecipe://about</string>
    <string>bencoding-linkrecipe://login</string>
  </array>
</dict>
</array>
</dict>
</plist>
</ios>
```

Android updating tiapp.xml

To create a custom URL scheme on Android, you will need to edit the `tiapp.xml` file to add an intent filter to listen for the specific `android:scheme` and `android:host` to be initialized.

1. First open the `tiapp.xml` file and edit the android configuration node.

```
<android xmlns:android="http://schemas.android.com/apk/res/
android">
```

2. Next, add the manifest node.

```
<manifest>
```

3. Add the application node. This node will later be used to generate your project's `AndroidManifest.xml` file, so make sure the attributes correctly match your project.

```
<application
  android:debuggable="false" android:icon=
  "@drawable/appicon" android:label=
  "XUrlScheme" android:name="XurlschemeApplication">
```

4. Next add the application's root activity.

```
<activity
  android:configChanges="keyboardHidden|orientation"
  android:label="XUrlScheme"
  android:name=".XurlschemeActivity"
  android:theme="@style/Theme.Titanium">
```

5. Then add the app's main intent filter. This will be used to launch your app.

```
<intent-filter>
  <action android:name=
  "android.intent.action.MAIN"/>
  <category android:name=
```

```

        "android.intent.category.LAUNCHER"/>
    </intent-filter>

```

- Next add a second intent filter with your custom URL information.

```

    <intent-filter>

```

- Then add the data node with your `android:scheme` and `android:host` information. These values act as the protocol used when `Ti.Platform.openURL` is used to launch the URL scheme. The following highlighted code allows you to launch the app using a URL such as `bencoding-linkrecipe://com.bencoding`.

```

        <data android:scheme="bencoding-linkrecipe"
            android:host="com.bencoding"/>

```

- Next a category must be added to the intent filter so that the app will be exposed correctly to third-party launchers and will open the app when the URL scheme is called. The following highlighted snippet shows the category information needed to correctly implement the custom URL scheme.

```

        <category android:name=
            "android.intent.category.DEFAULT" />
        <category android:name=
            "android.intent.category.BROWSABLE" />
        <action android:name=
            "android.intent.action.VIEW" />

```

```

    </intent-filter>

```

```

    </activity>
</application>
</manifest>
</android>

```

Creating the recipe UI

This recipe has a simple UI designed to demonstrate how different URL scheme features can be implemented.

- First, a `Ti.UI.Window` is created; this window is the root `Ti.UI.Window` of the app.

```

var win = Ti.UI.createWindow({
    backgroundColor: '#fff', title: 'Url Receiver',
    barColor: '#000', layout: 'vertical', fullscreen: false
});

```

2. Next if the recipe is running on an iOS device, a `Ti.UI.Button` is added to allow the URL Receiver app to launch the App Launcher app created in the *Launching one app from another* recipe discussed earlier in this chapter.

```
if(!my.isAndroid){
  var launcherButton = Ti.UI.createButton({
    title:'Press to open AppLauncher Sample',
    top:30, left:10, right:10, height:45
  });
  win.add(launcherButton);
}
```

3. When the `launcherButton` is tapped, the app will try to open the App Launcher if the app is available on the device. The highlighted code demonstrates how to use the `Ti.Platform.openURL` method to launch the app.

```
launcherButton.addEventListener('click',function(e){
  if(Ti.Platform.canOpenURL(
    "bencoding-linkLauncher://")
  ){
    Ti.Platform.openURL(
      "bencoding-linkLauncher://");
  }else{
    alert("Please install the AppLauncher Recipe");
  }
});
}
```

Launching helper functions

This recipe uses the `assist` object to help launch a different URL.

```
var assist = {
```

1. The `openWindow` method opens a specific window based on the URL and parameter values provided.

```
openWindow : function(url,params){
  if(url == 'about'){
    my.ui.openAboutWindow();
  }
  if (url == 'login'){
    my.ui.openLoginWindow(params);
  }
},
```

- The `openPageFromUrl` method is called when an app is opened or resumed to determine if the app has been opened from a third-party app and if so, what routing information has been provided.

```
openPageFromUrl : function() {
```

- The first step is to determine if the app has been opened from a third-party app. This is done by checking the `hasLaunchUrl` method.

```
if(!my.scheme.hasLaunchUrl()){
```

- If the app has not been launched from another app, the main page of the recipe will be displayed and any session variables will be reset.

```
my.scheme.session.reset();
return;
}
```

- If the app was launched by another app, we first need to check if the launching URL has changed. This avoids reloading the app if the same window has been requested. This can be done by calling the `hasChanged` method as highlighted in the following code snippet:

```
if(!my.scheme.session.hasChanged()){
return;
}
```

- Next the URL to be launched is obtained by calling the `getLaunchUrl` method as the following code snippet shows. This URL is then loaded into a `session` variable for later use.

```
my.scheme.session.launchUrl =
my.scheme.getLaunchUrl();
```

- Then the requested page is obtained using the `getCurrentPage` function. This will be used in determining which page to load using the `openWindow` method.

```
var requestedPage = my.scheme.getCurrentPage();
```

- Next, any launch parameters are obtained. The following code block demonstrates how to check if launch parameters have been provided and parses them into a formatted object.

```
my.scheme.session.launchParams = null;
if(my.scheme.hasParams()){
my.scheme.session.launchParams =
my.scheme.getParams();
}
```

9. Finally, the `openWindow` method is called using the `requestedPage` and `launchParams` created in the preceding steps. The `openWindow` method will then open the requested window in the app.

```
        assist.openWindow(requestedPage,  
                          my.scheme.session.launchParams);  
    }  
};
```

When launched from another app

In order to have the recipe launch the requested window, listeners must be added to the resumed and open events. The following steps detail how to add the proper event listeners to your app:

1. If the recipe is running on iOS, the resumed event listener is added. Please note this must be added to your `app.js` when implementing in your own app.

```
if(!my.isAndroid){  
  
    Ti.App.addEventListener('resumed', function(e) {
```

2. The `assist.openPageFromUrl` method will be called each time the app is resumed. If no URL information is provided, the main window will be displayed.

```
        assist.openPageFromUrl();  
    });  
}
```

3. A listener is added to the open event on the main window. This event will be fired when the app is first launched.

```
win.addEventListener('open', function(e) {
```

4. On the first launch, the `assist.openPageFromUrl` method is called to determine if the app has been opened by a third-party app.

```
    assist.openPageFromUrl();
```

5. If the recipe is running on Android, a listener is added to the `resume` event on the main window. This allows the recipe to detect if a third-party app tries to open the recipe while the app is running in the background.

```
    if(my.isAndroid){  
        win.activity.addEventListener('resume', function(){  
            assist.openPageFromUrl();  
        });  
    }  
});  
  
win.open();
```

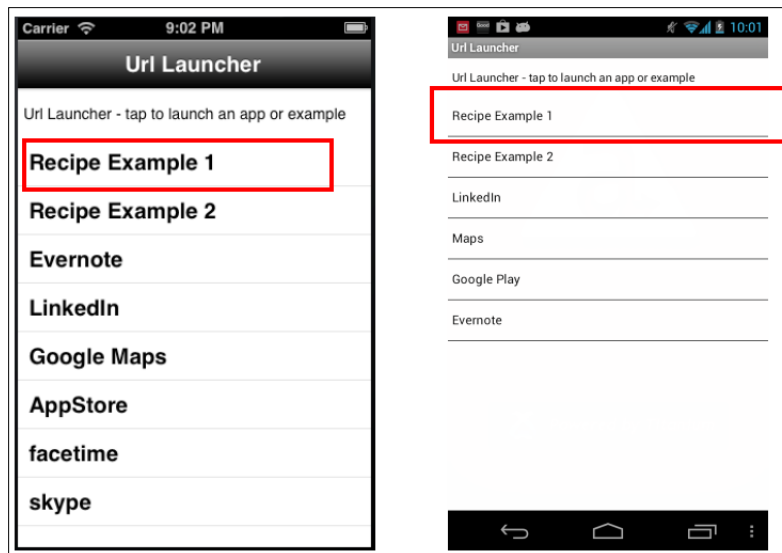
How it works...

Cross-platform custom URL schemes are a great low-cost way to provide integration points with third parties. The following section details the end-to-end process for using the *Launching one app from another* recipe to access different app routes without this recipe's sample app.

Launching the About window

This section provides a step-by-step description on how you can launch the **About** window in the URL Receiver recipe when launched from the App Launcher recipe.

1. Open the App Launcher application and tap on the **Recipe Example 1** row in the table view as shown with the red boxes in the following screenshot.



2. The App Launcher app runs the following code block.
 - **On iOS:**

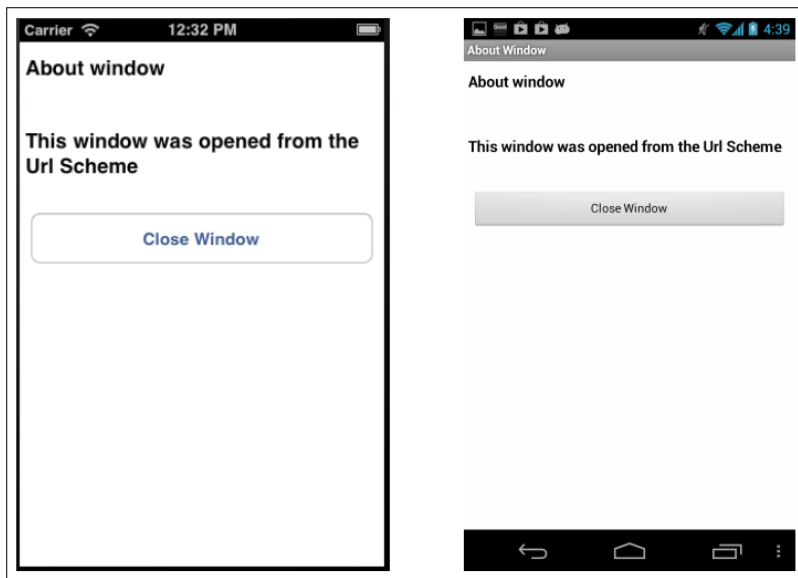
```
Ti.Platform.openURL('bencoding-linkrecipe://about');
```
 - **On Android:**

```
Ti.Platform.openURL('bencoding-linkrecipe://com.bencoding/about');
```
3. The URL Receiver recipe will then be launched on your device.

- The resumed or open event handlers will then parse the provided URL information. When the `getCurrentPage` method is called, the `requestedPage` value will be set to a string with the value of `about`.

```
var requestedPage = my.scheme.getCurrentPage();
```

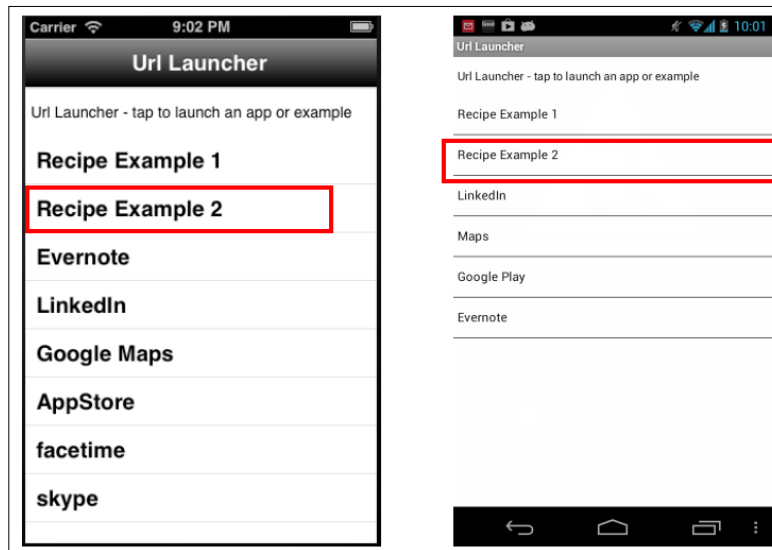
- Next, `my.scheme.hasParams` method is called to determine if any parameters have been passed as part of the provided URL. In this case, no additional parameters have been provided, so the `session` parameter object remains `null`.
- The `openWindow` method is then called and the following **About** window is then opened:



Launching the Login window

This section provides a step-by-step description of launching the **Login** window in the URL Receiver recipe when launched from the App Launcher recipe. As part of the launch process, the user and token parameters are provided by the App Launcher recipe then used to complete the form within the URL Receiver recipe app.

- Open the App Launcher application and tap on the **Recipe Example 2** row in the table view as shown with the red boxes in the following screenshot:



2. The App Launcher application runs the following code block.

□ **On iOS:**

```
Ti.Platform.openURL('bencoding-linkrecipe://
login?user=chris&token=12345');
```

□ **On Android:**

```
Ti.Platform.openURL(
'bencoding-linkrecipe://com.bencoding/' +
login?user=chris&token=12345');
```

3. The URL Receiver recipe will then be launched on your device.

4. The resumed or open event handlers will then parse the provided URL information. When the `getCurrentPage` method is called, the `requestedPage` value will be set to a string with the value of `login`.

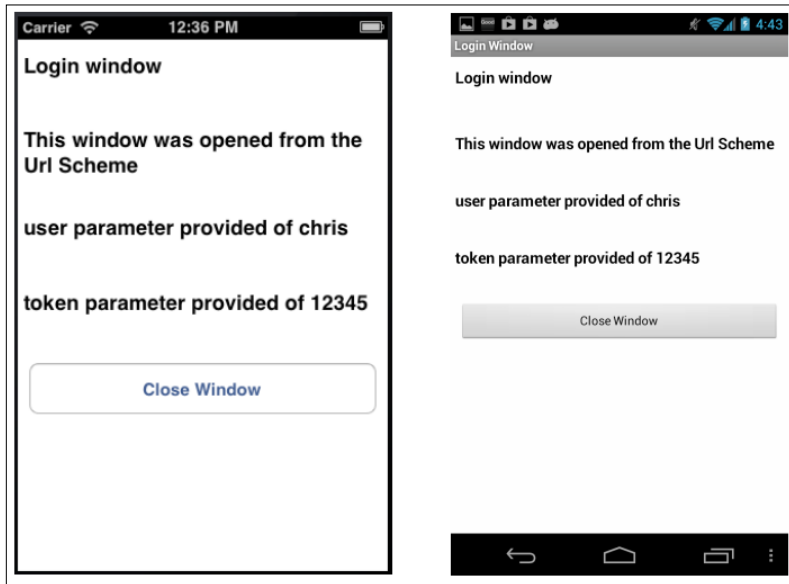
```
var requestedPage = my.scheme.getCurrentPage();
```

5. Next, `my.scheme.hasParams` method is called to determine if any parameters have been passed as part of the provide URL.

6. The following parameter object is created using the query string parameters included with the calling URL.

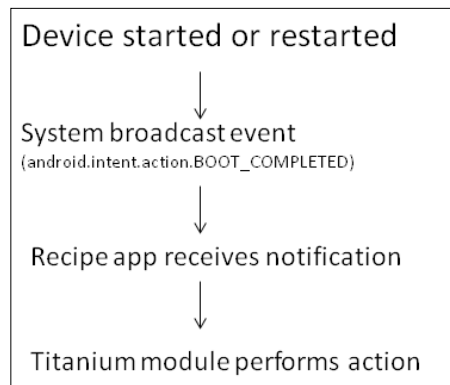
```
{ user:"chris",
  token:12345 }
```

- The `openWindow` method is then called providing the `requestedPage` and `parameter` objects. This will then open the **Login** page with the parameter used to complete the form as shown in the following screenshot:



Opening your Android app with **BOOT_COMPLETED**

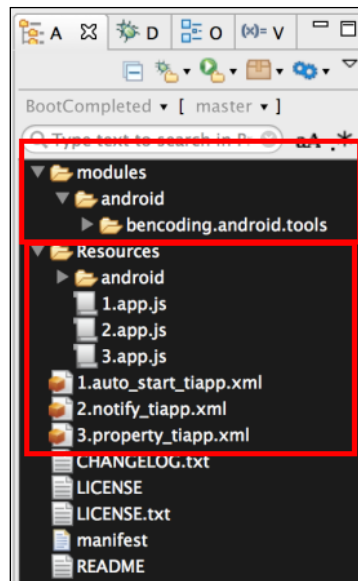
Many enterprise apps have the requirement of always running in the background. The most common categories of these apps would be for route management, item tracking, or customer scheduling apps. In your Android Titanium app, you can use an intent filter for the `BOOT_COMPLETED` action to notify your app that the device has been restarted. The event lifecycle for the `BOOT_COMPLETED` action is shown in the following diagram:



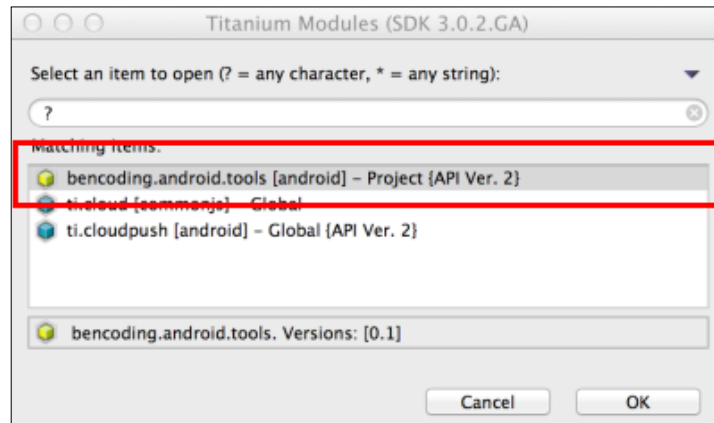
Getting ready

This recipe uses the `bencoding.android.tools` native module to help with the subscription of the `BOOT_COMPLETED` broadcast. This module and other code assets can be downloaded from the source provided by the book, or individually through the links provided in the See also section at the end of this recipe. To install this module, simply copy the `modules` folder shown in the following screenshot into your project.

Also included with the sample recipe is a series of example `tiapp.xml` and `app.js` files demonstrating different options in handling the receipt of the `BOOT_COMPLETED` broadcast. You will need to copy the files highlighted in the following screenshot into your project as they will be used to demonstrate the different available options.



After copying the mentioned files, you will need to click on your **tiapp.xml** file in Titanium Studio and add a reference to the `bencoding.android.tools` module as shown in the following screenshot:



How to do it...

This recipe demonstrates how to implement the following scenarios:

- ▶ Auto restarting your app upon receiving the `BOOT_COMPLETED` broadcast
- ▶ Sending a notification to the user receiving the `BOOT_COMPLETED` broadcast
- ▶ Using Titanium properties to configure how your app handles the `BOOT_COMPLETED` broadcast

Required tiapp.xml updates


When a Titanium Android app is launched, the Titanium framework first must be initialized before your app can run. When the app is launched by another service such as the `BOOT_COMPLETED` broadcast, you will receive a message that the app needs to restart. To avoid this issue, you must add the following properties into your `tiapp.xml` file:

```
<property name="ti.android.bug2373.finishfalseroot" type="bool">true</property>
<property name="ti.android.bug2373.disableDetection" type="bool">true</property>
<property name="ti.android.bug2373.restartDelay" type="int">500</property>
<property name="ti.android.bug2373.finishDelay" type="int">0</property>
<property name="ti.android.bug2373.skipAlert" type="bool">true</property>
<property name="ti.android.bug2373.message">Initializing</property>
```

```
<property name="ti.android.bug2373.title">Restart Required</property>
<property name="ti.android.bug2373.buttonText">Continue</property>
```

Scenario A – auto restart

The first `BOOT_COMPLETED` option provided by the `Android.Tools.Receiver` module is the ability to restart your app. The typical use case would be to restart your Titanium app when the user restarts his/her device.

 To deploy this scenario to your device using the recipe source, rename the `1.auto_start_tiapp.xml` file to `tiapp.xml` and the `1.app.js` to `app.js`. This will switch the recipe app to use the following scenario details.

Auto restart step 1: adding the receiver to tiapp.xml

The first step in enabling this scenario is to add the receiver entry into the Android configuration node in our project's `tiapp.xml` file located in the root of your Titanium Project.

```
<receiver android:exported="true" android:name="bencoding.android.
receivers.BootReceiver">
```

1. First an intent filter must be added to our receiver, this will subscribe our app to the `BOOT_COMPLETED` broadcast.

```
<intent-filter>
  <action android:name=
    "android.intent.action.BOOT_COMPLETED"/>
</intent-filter>
```

2. Next a metadata node is added with the name `bootType`. This node is used by the `Android.Tools` module to determine what boot action should be taken. The following snippet demonstrates how to configure the module to restart the app.

```
<meta-data android:name=
  "bootType" android:value="restart"/>
```

3. Then a metadata node is added with the name `sendToBack`. This node is used by the `Android.Tools` module to determine if the app should be restarted in the background if set to `true`, or restarted in the foreground if set to `false`.

```
<meta-data android:name=
  "sendToBack" android:value="true"/>
</receiver>
```

4. Finally the `RECEIVE_BOOT_COMPLETED` permission is added.

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

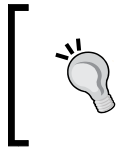
Auto restart step 2: testing

The following steps outline how to best test this recipe scenario:

1. Clean your Titanium solution and push to device.
2. Make sure the app is not running on your device. You can stop the app using the **Force stop** option under **Settings**.
3. Restart your device.
4. After the device has been restarted, confirm the date and time displayed on the recipe's main screen.

Scenario B – notification on restart

The second `BOOT_COMPLETED` option provided by the `Android.Tools.Receiver` module is the ability to create a notification message when a user restarts his/her device. This can be used to provide a reminder or instructions to the user.



To deploy this scenario to your device using the recipe source, rename the `2.auto_start_tiapp.xml` file to `tiapp.xml` and the `2.app.js` file to `app.js`. This will switch the recipe app to use the following scenario details.

Notification on restart step 1: adding receiver to tiapp.xml

The first step in enabling this scenario is to add the receiver entry into the Android configuration node in our project's `tiapp.xml` file.

```
<receiver android:exported="true" android:name="bencoding.android.receivers.BootReceiver">
```

1. First, an intent filter must be added to our receiver, this will subscribe our app to the `BOOT_COMPLETED` broadcast.

```
<intent-filter>  
  <action android:name=  
    "android.intent.action.BOOT_COMPLETED"/>  
</intent-filter>
```

- Next, a metadata node is added with the name `bootType`. This node is used by the `Android.Tools` module to determine what boot action should be taken. The following snippet demonstrates how to configure the module to send a notification message on device restart.

```
<meta-data android:name=
  "bootType" android:value="notify"/>
```

- Then a metadata node is added with the name `title`. This node contains the notification title that will be used.

```
<meta-data android:name="title"
  android:value="Title Sample from tiapp.xml"/>
```

- Then a metadata node is added with the name `message`. This node contains the notification message that will be used.

```
<meta-data android:name="message"
  android:value="Message Sample from
  tiapp.xml"/></receiver>
```

- Finally the `RECEIVE_BOOT_COMPLETED` permission is added.

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_
  COMPLETED"/>
```

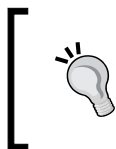
Notification on restart step 2: testing

The following steps outline the best method to test this recipe scenario:

- Clean your Titanium solution and push to device.
- Make sure the app is not running on your device. You can stop the app using the **Force stop** option under **Settings**.
- Restart your device.
- Within a few seconds of device reboot, you will see a new message in your notification tray.

Scenario C – property controlled

The final `BOOT_COMPLETED` option provided by the `Android.Tools.Receiver` module is the ability to restart your app. The typical use case would be to restart your Titanium app when the user restarts his/her device.



To deploy this scenario to your device using the recipe source, rename the `3.auto_start_tiapp.xml` file to `tiapp.xml` and the `3.app.js` file to `app.js`. This will switch the recipe app to use the following scenario details.

Property controlled step 1: adding receiver to tiapp.xml

The first step in enabling this scenario is to add the receiver entry into the Android configuration node in our project's tiapp.xml file.

```
<receiver android:exported="true" android:name="bencoding.android.receivers.BootReceiver">
```

1. An intent filter must be added to our receiver, this will subscribe our app to the BOOT_COMPLETED broadcast.

```
<intent-filter>
  <action android:name="
    android.intent.action.BOOT_COMPLETED"/>
</intent-filter>
```

2. Next, a metadata node is added with the name `bootType`. This node is used by the `Android.Tools` module to determine what boot action should be taken. The following code snippet demonstrates how to configure the module to use Titanium properties to determine what action should be taken.

```
<meta-data android:name="bootType"
  android:value="propertyBased"/>
```

3. Metadata nodes are then added to configure the `BOOT_COMPLETED` receiver. Each node is used to map a Titanium property to a receiver configuration element. For example, the `bootType_property_to_reference` holds the name of the property to be used to determine the `bootType`.

```
<meta-data android:name="enabled_property_to_reference"
  android:value="my_enabled"/>
<meta-data android:name="bootType_property_to_reference"
  android:value="my_bootType"/>
<meta-data android:name="sendToBack_property_to_reference"
  android:value="my_sendtoback"/>
<meta-data android:name="icon_property_to_reference"
  android:value="my_notify_icon"/>
<meta-data android:name="title_property_to_reference"
  android:value="my_notify_title"/>
<meta-data android:name="message_property_to_reference"
  android:value="my_notify_message"/></receiver>
```

4. Finally, the `RECEIVE_BOOT_COMPLETED` permission is added.

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_
  COMPLETED"/>
```

Property controlled step 2: create recipe app.js

The `Android.Tools` module allows for the `BOOT_COMPLETED` receiver to be configured through Titanium properties. The following code snippet (`app.js`) demonstrates how to create two different configurations by simply updating the correct Titanium properties.

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'BOOT_COMPLETED Example',
  layout:'vertical',fullscreen:false, exitOnClose:true
});
```

2. Next, a `Ti.UI.Button` is created to demonstrate how to create a foreground restart.

```
var button1 = Ti.UI.createButton({
  title:'Foreground Restart',
  top:25, height:45, left:5, right:5
});
win.add(button1);
```

3. On the `click` event of our first button, the properties used to configure our `BOOT_COMPLETED` receiver are updated.

```
button1.addEventListener('click',function(e){
```

4. The first action performed in the `click` event is to set the `BOOT_COMPLETED` receiver as enabled.

```
  Ti.App.Properties.setBool('my_enabled',true);
```

5. Next, the `bootType` is set to restart. This will restart our Titanium app on device reboot.

```
  Ti.App.Properties.setString("my_bootType", "restart");
```

6. Then the `sendToBack` property is set to `false`. This will restart our Titanium app in the foreground when the user restarts his/her device.

```
  Ti.App.Properties.setBool("my_sendtoback", false);
});
```

7. Next, a second button is created to demonstrate how to configure the notification.

```
var button2 = Ti.UI.createButton({
  title:'Restart Notification',
  top:25, height:45, left:5, right:5
});
win.add(button2);
```

8. Next a second button is created to demonstrate how to configure notification to be generated when the device is restarted.

```
button2.addEventListener('click',function(e){
```
9. An action performed in the `click` event is to set the `BOOT_COMPLETED` receiver to enabled.

```
Ti.App.Properties.setBool('my_enabled',true);
```
10. Next the `bootType` is set to `notify`. This will send a message after receiving the `BOOT_COMPLETED` broadcast.

```
Ti.App.Properties.setString("my_bootType", "notify");
```
11. Next, the notification icon resource identifier is created. This resource identifier will be used to create an icon when the notification is created. If no resource identifier is provided, a star icon will be used by default.

```
.App.Properties.setInt("my_notify_icon", 17301618);
```
12. Finally the properties linked to the notification title and message are set with new string values. These properties will be used in generating the notification when the device has been restarted.

```
Ti.App.Properties.setString(  
    "my_notify_title", "Title from property");  
Ti.App.Properties.setString(  
    "my_notify_message","Message from property");  
});
```

Property controlled step 3: testing

The following steps outline how to best test this recipe scenario:

1. Clean your Titanium solution and push to device.
2. Open the app, and press the first button (`button1`). This will update the properties to perform a foreground restart.
3. Next stop the app; you can stop the app using the **Force stop** option under **Settings**.
4. Restart your device.
5. After the device is started, the recipe app will be launched in the foreground.
6. Press the second button (`button2`). This will update the properties to send a notification on restart.
7. Restart your device.
8. Within a few seconds of device reboot, you will see a new message in your notification tray.

See also

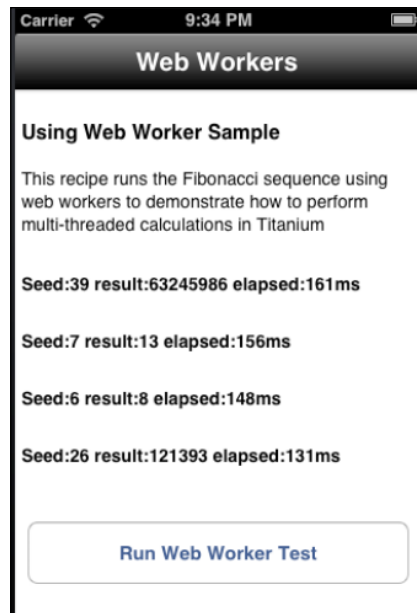
The `Android.Tools` module was used in this recipe to provide the `BOOT_COMPLETED` functionality. For additional information about this module, please visit the project on Github at <https://github.com/benbahrenburg/benCoding.Android.Tools>.

For full documentation on the `BootReceiver`, visit <https://github.com/benbahrenburg/benCoding.Android.Tools/tree/master/documentation/bootreceiver.md>.

iOS Multithreading using Web Workers

Enterprise apps often require a large amount of processing to be performed. To provide the best user experience while fully leveraging the limited device resources, you must perform compute operations on a background thread.

In this recipe, we will discuss how to use the `Ti.WebWorkerWrapper` module to perform background compute operations. To simulate a compute job, a fibonacci sequence is calculated for random numbers and processed in parallel. The following screenshot shows the recipe running these Web Worker jobs on an iPhone.

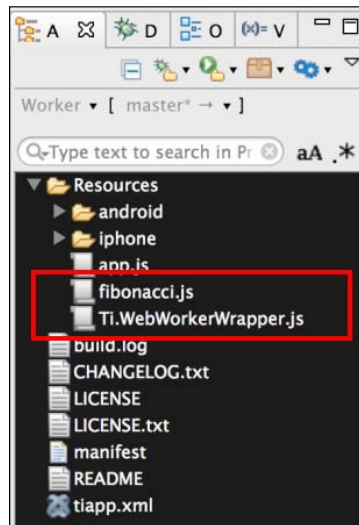




This recipe is an iOS-only recipe as it requires Web Workers, which are not yet available for Titanium Android.

Getting ready

This recipe uses the `Ti.WebWorkerWrapper` CommonJS module and `fibonacci.js` Web Worker. These modules and other code assets can be downloaded from the source provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. To install the `Ti.WebWorkerWrapper` module and Web Worker into your project, simply copy the `Ti.WebWorkerWrapper.js` and `fibonacci.js` files into the `Resources` folder of your Titanium project as highlighted in the following screenshot:



How to do it...

Once you have added the `Ti.WebWorkerWrapper` module and `fibonacci.js` file to your project, you next need to create your application namespace and use `require` to import the module into your `app.js` file as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  workerMod : require('Ti.WebWorkerWrapper'),
  isAndroid : Ti.Platform.osname === 'android'
};
```

Creating the recipe UI

This recipe provides a basic UI to launch and track the execution of the Web Workers execution of the fibonacci sequence. The following steps detail how to create the main components illustrated in the earlier screenshot:

1. First a `Ti.UI.Window` is created. This window will be used to attach and display all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'web Workers',
  barColor: '#000', layout: 'vertical', fullscreen: false
});
```

2. Next a series of `Ti.UI.Label` controls are added to the `Ti.UI.Window`. These controls will be used to display the progress and results of the Web Workers.

```
var worker1 = Ti.UI.createLabel({
  top:20, height:25, left:5, right:5,color:'#000',
  textAlign:'left',text:'Not yet processed - Press Button',
  font:{fontSize:14, fontWeight:'bold'}
});
win.add(worker1);
```



Three additional Web Worker labels are added using the same template as the `worker1` label shown in the preceding code snippet.

3. Finally a `Ti.UI.Button` is added to the `Ti.UI.Window`. This button is used to launch the four Web Worker processes used in this recipe.

```
var runButton = Ti.UI.createButton({
  title:'Run Web Worker Test', top:40,
  left:10, right:10, height:50,
});
win.add(runButton);
```

Testing functions

This recipe uses a `tests` object to assist with timing and displaying the results of the Web Worker fibonacci sequence processing. The following section discusses functionality contained within the `tests` JavaScript object.

```
var tests = {
```

1. The `updateLabel` method is used to apply a formatting template to each label with the results of the fibonacci sequence callback results.

```
updateLabel : function(label,e) {
    label.text = "Seed:" + e.seed + " result:" +
    e.result + " elapsed:" + e.elapsed + "ms";
},
```

2. The `worker1` through `worker4` methods are the callback methods provided to each Web Worker as it processes the fibonacci sequence. These callbacks receive the results from the calculation process.

```
worker1 : function(e) {
    tests.updateLabel(worker1,e);
},
worker2 : function(e) {
    tests.updateLabel(worker2,e);
},
worker3 : function(e) {
    tests.updateLabel(worker3,e);
},
worker4 : function(e) {
    tests.updateLabel(worker4,e);
},
```

3. The `random` method is used to generate a random number within a given range. This method is used to generate the random number sent to the fibonacci sequence.

```
random : function(from,to) {
    return Math.floor(Math.random() * (to-from+1) +from);
}
};
```

Multithreading using Web Workers

The multithreading example is run when the user taps on the `runButton` button.

The following section demonstrates how four Web Workers are created in the background.

Once each Web Worker has finished processing, the results are provided to the test's callback method discussed earlier in this recipe.

```
runButton.addEventListener('click', function(e) {
```

1. First a new worker object is created by initializing a new instance of the `Ti.WebWorkerWrapper` module.

```
var worker1 = new my.workerMod();
```

2. Next, the `start` method is called on the `worker` object. The `start` method requires the following arguments:
 - The path of the JavaScript file the Web Worker should execute
 - A random number passed as a Web Worker `postMessage` for the fibonacci sequence to use
 - The callback method to be used when the Web Worker has completed processing

```
worker1.start('fibonacci.js', tests.random(1, 50),
             tests.worker1);
```

3. Three additional Web Workers are created using the same pattern demonstrated in the creation of `worker1`.

```
var worker2 = new my.workerMod();
worker2.start('fibonacci.js', tests.random(1, 50),
             tests.worker2);

var worker3 = new my.workerMod();
worker3.start('fibonacci.js', tests.random(1, 50),
             tests.worker3);

var worker4 = new my.workerMod();
worker4.start('fibonacci.js', tests.random(1, 50),
             tests.worker4);

});
```

See also

- ▶ The `Ti.WebWorkerWrapper` CommonJS module is used in this recipe to provide multithreading support for the `fibonacci.js` computation. For additional details, please visit the module on Github at <https://github.com/benbahrenburg/Ti.WebWorkerWrapper>.

8

Basic Security Approaches

In this chapter we will cover:

- ▶ Implementing iOS data protection in Titanium
- ▶ AES encryption using JavaScript
- ▶ Basic authentication using `Ti.Network.HTTPClient`
- ▶ Implementing a cross-platform passcode screen
- ▶ Working with protected ZIP files on iOS


Introduction

Security, privacy, and safeguards for intellectual property are at the front of the minds of those of us building Titanium Enterprise apps. Titanium allows you to combine the underlying platform tools and third-party JavaScript libraries to help meet your security requirements.

This chapter provides a series of approaches on how to leverage JavaScript, Titanium modules, and the underlying platform to enable you to create a layered security approach to assist you in meeting your organization's overall secure development goals. Each recipe is designed to provide building blocks to help you implement your industry's existing security and privacy standards.

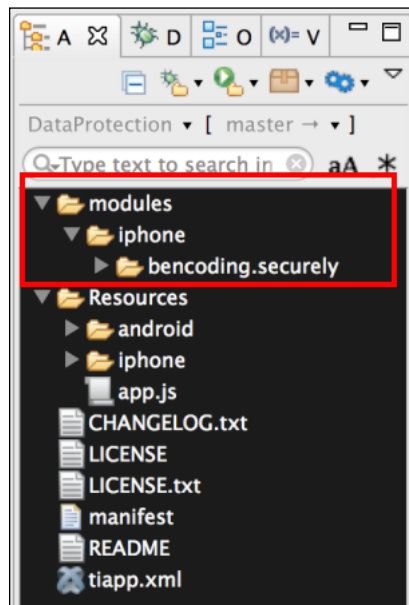
Implementing iOS data protection in Titanium

Starting with iOS 4, Apple introduced the ability for apps to use the data protection feature to add an additional level of security for data stored on disk. Data protection uses the built-in hardware encryption to encrypt files stored on the device. This feature is available when the user's device is locked and protected with a passcode lock. During this time, all files are protected and inaccessible until the user explicitly unlocks the device.

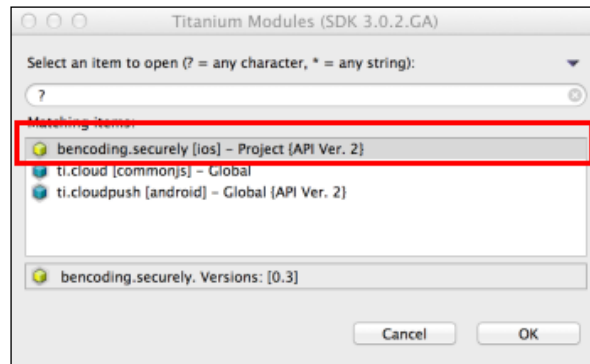
 When the device is locked, no app can access protected files. This even applies to the app that created the file.

Getting ready

This recipe uses the `securely` native module for enhanced security functionality. This module and other code assets can be downloaded from the source provided by the book. Installing these in your project is straightforward. Simply copy the `modules` folder into your project as shown in the following screenshot:



After copying the mentioned folder, you will need to click on your **tiapp.xml** file in Titanium Studio and add a reference to the `bencoding.securely` module as shown in the following screenshot:



Enabling data protection

This recipe requires your iOS device to have data protection enabled. You will need a device as the simulator does not support data protection. The following steps cover how to enable this feature on your device:

1. Go to **Settings | General | Passcode**.
2. Follow the prompts to set up a passcode.
3. After adding a passcode, scroll to the bottom of the screen and verify that the text **Data protection is enabled** is visible as shown in the following screenshot:



iOS device browser

A third-party iOS device browser is needed to verify that data protection for the example recipe app has successfully been enabled. This recipe discusses how to verify data protection using the popular iExplorer app. An evaluation version of the iExplorer app can be used to follow along with this recipe. For more information and to download iExplorer, please visit <http://www.macroplant.com/iexplorer>.

How to do it...

To enable iOS data protection, the `DataProtectionClass` and `com.apple.developer.default-data-protection` keys need to be added to your `tiapp.xml` as demonstrated in the following code snippet:

1. First, add the `ios` configuration node if your project does not already contain this element.

```
<ios>
  <plist>
    <dict>
```

2. Then at the top of the `dict` node, add the following highlighted keys.

```
    <key>DataProtectionClass</key>
    <string>NSFileProtectionComplete</string>
    <key>com.apple.developer.
      default-data-protection</key>
    <string>NSFileProtectionComplete</string>
  </dict>
</plist>
</ios>
```

3. After saving the updates to your `tiapp.xml`, you must clean your Titanium project in order to have the updates take effect. This can be done in Titanium Studio by selecting **Project | Clean**.

Creating the namespace and imports

Once you have added the `securely` module and added the `tiapp.xml` updates to your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  secure : require('bencoding.securely')
};
```

Creating the recipe UI

The following steps outline how to create the UI used in this recipe:

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff',
  title: 'Data Protection Example',
  barColor: '#000', layout: 'vertical'
});
```

2. Next, a `Ti.UI.Button` is added to the `Ti.UI.Window`. This will be used to trigger our example.

```
var button1 = Ti.UI.createButton({
  title: 'Create Test File',
  top: 25, height: 45, left: 5, right: 5
});
win.add(button1);
```

Creating a file to verify data protection

To verify if data protection is enabled in the app, the recipe creates a time-stamped file in the `Ti.Filesystem.applicationDataDirectory` directory. Using an iOS device browser, we can verify if the test file is protected when the device is locked. The following steps describe how the recipe creates this test file:

1. The `click` event for `button1` creates a time-stamped file that allows us to verify if data protection has been correctly enabled for the app.
2. Next the `isProtectedDataAvailable` method is called on `securely`. This provides a Boolean result indicating that data protection allows the app to read from or write to the filesystem.

```
if(!my.secure.isProtectedDataAvailable()){
  alert('Protected data is not yet available.');
```

3. To ensure there is a unique identifier in the file, a token is created using the current date and time. This token is then added to the following message template:

```
var timeToken = String.formatDate(new Date(), "medium") +
String.formatTime(new Date());
var msg = "When device is locked you will not be able";
msg += " to read this file. Your time token is ";
msg += timeToken;
```

4. The message created in step 3 is then written to the `test.txt` file located in the `Ti.Filesystem.applicationDataDirectory` directory. If the file already exists, it is removed so that the latest message will be available for testing.

```
var testfile = Ti.Filesystem.getFile(
Ti.Filesystem.applicationDataDirectory, 'test.txt');
if(testfile.exists()){
    testfile.deleteFile();
}
testfile.write(msg);
testfile = null;
```

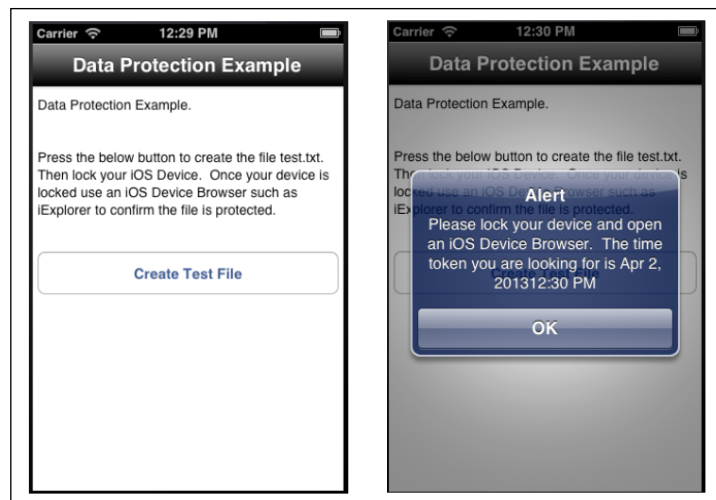
5. Once the `test.txt` file is written to the device, a message is displayed to the user notifying them to lock their device and use an iOS device browser to confirm data protection is enabled.

```
var alertMsg = "Please lock your device.";
alertMsg+= "Then open an iOS Device Browser.";
alertMsg+= "The time token you are looking for is ";
alertMsg+= timeToken;
alert(alertMsg);
```

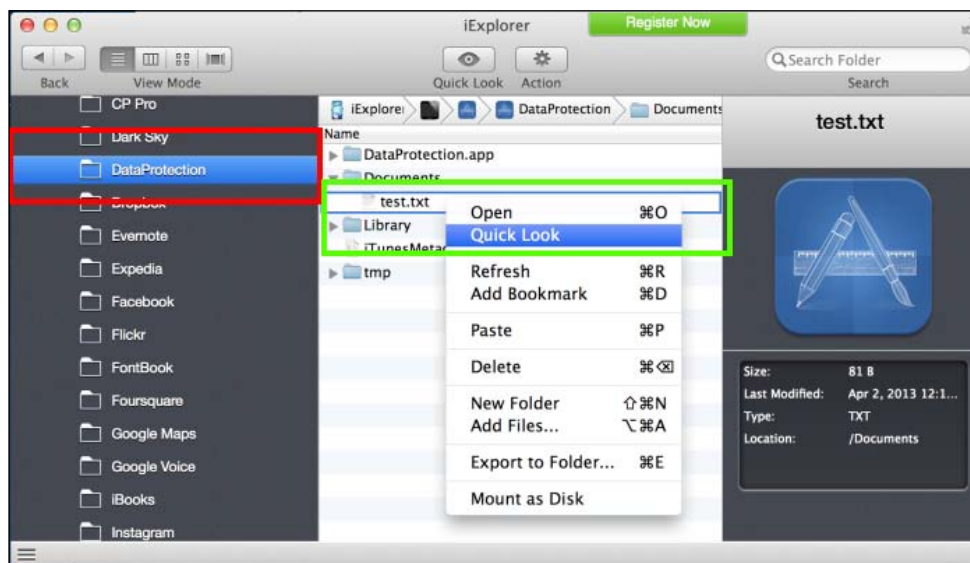
How it works...

After the `DataProtectionClass` and `com.apple.developer.default-data-protection` keys have been added to your `tiapp.xml`, the iOS device handles protecting your files when the device is locked. The following steps discuss how to test that this recipe has correctly implemented data protection:

1. The first step in the validation process is to build and deploy the recipe app to your iOS device.
2. Once the app has been loaded onto your device, open the app and press the **Create Test File** button.



3. Once you have received an alert message indicating the test file has been created, press the home button and lock your device.
4. Plug your device into a computer with iExplorer installed.
5. Open iExplorer and navigate so that you can view the apps on your device.
6. Select the **DataProtection** app as marked with the red box in the following screenshot. Then right-click on the **test.txt** file located in the **Documents** folder and select **Quick Look** as marked with the green box in the following screenshot:



7. After **Quick Look** is selected, iExplorer will try to open the `test.txt` file. Since it is protected, it cannot be opened and **Quick Look** will show the progress indicator until a timeout has been reached.



8. You can then unlock your device and repeat the preceding steps to open the file in **Quick Look**.

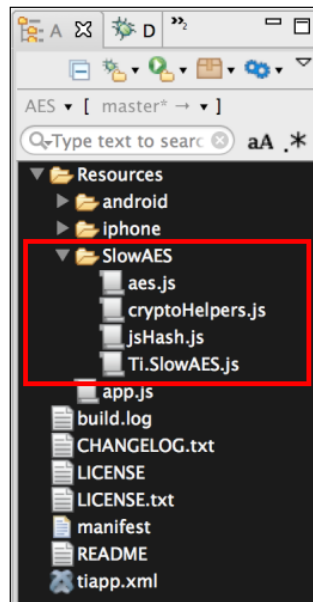
AES encryption using JavaScript

The **Advanced Encryption Standard (AES)** is a specification for the encryption of electronic data established by the U.S. NIST in 2001. This encryption algorithm is used for securing sensitive, but unclassified material by U.S. Government agencies. AES has been widely adopted by enterprise and has become a de facto encryption standard for many commercially sensitive transactions.

This recipe discusses how AES can be implemented in JavaScript and incorporated into your Titanium Enterprise app.

Getting ready

This recipe uses the `Ti.SlowAES` CommonJS module as a wrapper around the `SlowAES` open source project. This module and other code assets can be downloaded from the source provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing these in your project is straightforward. Simply copy the `SlowAES` folder into the `Resources` folder of your project as shown in the following screenshot:



How to do it...

Once you have added the `SlowAES` folder to your project, next you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

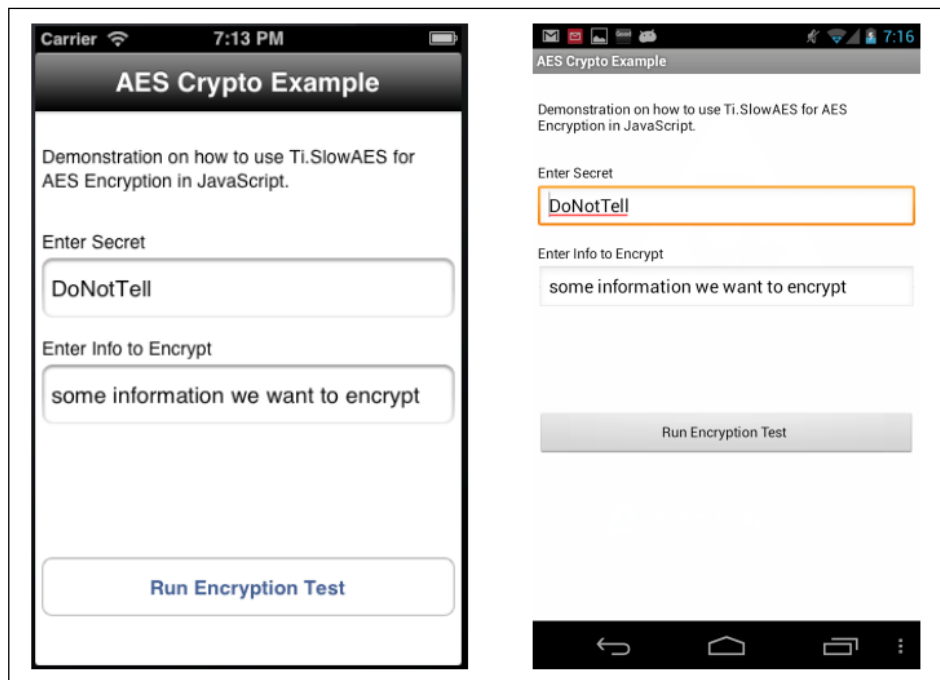
```
//Create our application namespace
var my = {
  mod : require('SlowAES/Ti.SlowAES')
};
```

Creating the recipe UI

This recipe demonstrates the usage of the `Ti.SlowAES` CommonJS module through a sample app using two `Ti.UI.TextField` controls for input.

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'AES Crypto Example',
  barColor: '#000', layout: 'vertical', fullscreen: false
});
```



- Next, a `Ti.UI.TextField` control is added to the `Ti.UI.Window` to gather the secret from the user.

```
var txtSecret = Ti.UI.createTextField({
    value:'DoNotTell',hintText:'Enter Secret',
    height:45, left:5, right:5,
    borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtSecret);
```

- Another `Ti.UI.TextField` is added to the `Ti.UI.Window` to gather the string to encrypt from the user.

```
var txtToEncrypt = Ti.UI.createTextField({
    value:'some information we want to encrypt',
    hintText:'Enter information to encrypt',
    height:45, left:5, right:5,
    borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtToEncrypt);
```

- Next a `Ti.UI.Label` is added to the `Ti.UI.Window`. This `Ti.UI.Label` will be used to display the encrypted value to the user.

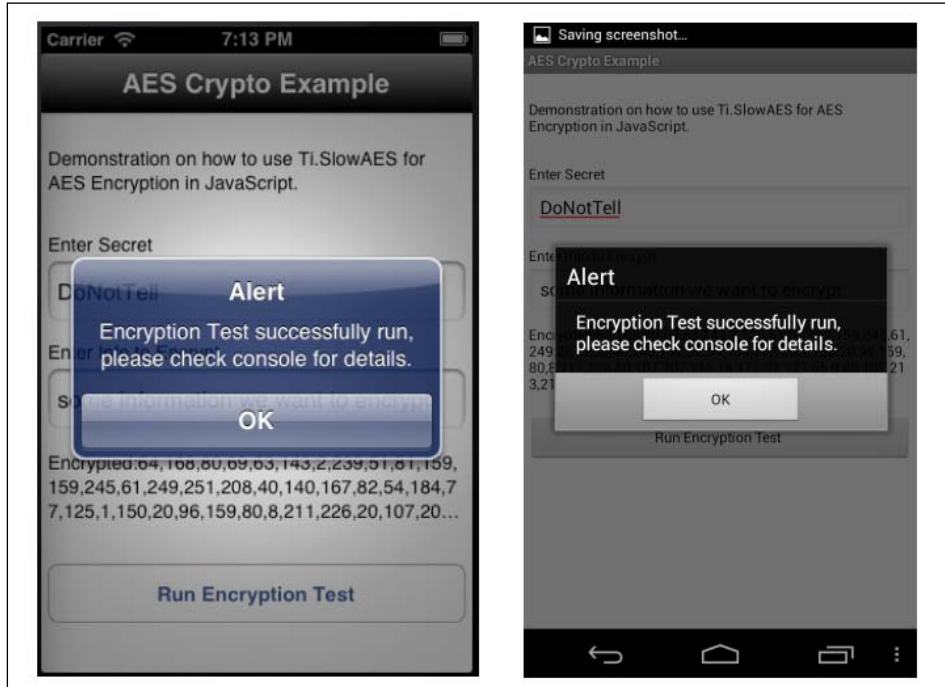
```
var encryptedLabel = Ti.UI.createLabel({
    top:10, height:65, left:5, right:5,color:'#000',
    textAlign:'left',font:{fontSize:14}
});
win.add(encryptedLabel);
```

- Finally a `Ti.UI.Button` is added to the `Ti.UI.Window`. This `Ti.UI.Button` will be used later in the recipe to perform the encryption test.

```
var btnEncrypt = Ti.UI.createButton({
    title:'Run Encryption Test', top:25,
    height:45, left:5, right:5
});
win.add(btnEncrypt);
```

Encrypting and decrypting values

This section demonstrates how to use the `Ti.SlowAES` module to use the secret entered in the `txtSecret`, `Ti.UI.TextField` to encrypt the contents of the `txtToEncrypt`, `Ti.UI.TextField`. Once completed, the encrypted value is then decrypted and compared against the original input. The results are displayed to the user in an alert message as shown in the following screenshots:



The encryption test is performed when the click event for the `btnEncrypt` control is fired as shown in the following code snippet:

```
btnEncrypt.addEventListener('click', function(x) {
```

1. The first step in the encryption process is to create a new instance of the `SlowAES` module as shown in this code snippet.
2. Next using the `encrypt` function, the secret provided in the `txtSecret` control is used to encrypt the value in the `txtToEncrypt` control. The encrypted results are then returned to the `encryptedValue` as demonstrated in the following statement:

```
var encryptedValue =  
crypto.encrypt(txtToEncrypt.value, txtSecret.value);
```

3. The `encryptedLabel.text` property is then updated to display the encrypted value to the user.

```
encryptedLabel.text = 'Encrypted:' + encryptedValue;
```

4. Next, the `decrypt` method is used to demonstrate how to decrypt the string value encrypted earlier. This method requires the encrypted string value and the secret as shown in the following snippet:

```
var decryptedValue =  
crypto.decrypt(encryptedValue,txtSecret.value);
```

5. Finally, the original input value is compared against the decrypted value to ensure our encryption test was successful. The results of this test are then displayed to the user through a message alert and the Titanium Studio console.

```
alert((txtToEncrypt.value ===decryptedValue) ?  
'Encryption Test successfully ran check console for details.':  
'Test failed, please check console for details.');
```

```
});
```

See also

You can also refer to the following resources:

- ▶ It is important to understand some of the weaknesses of JavaScript Cryptography. When deciding on an encryption approach, I would recommend reading the Matasano Security article on this subject available at <http://www.matasano.com/articles/javascript-cryptography>.
- ▶ This recipe uses the `SlowAES` project to implement the AES algorithm. To read the license and to learn more about the `SlowAES` project, please visit <http://code.google.com/p/slowaes>.
- ▶ To download the `Ti.SlowAES` CommonJS module and learn more about how to implement its functionality visit <https://github.com/benbahrenburg/Ti.SlowAES>.

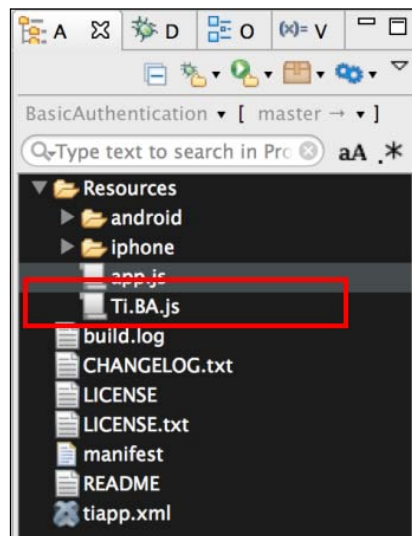
Basic authentication using Ti.Network.HTTPClient

Most enterprise sites or services provide access to content through basic access authentication. Basic authentication is a method for a HTTP user agent to provide a username and password when making a request. It is important to remember that basic authentication base64 encodes your credentials, not encrypts them. For this reason, it is advised that HTTPS is used when creating network connections.

Titanium provides full support for basic authentication using the SDK's `Ti.Network` functionality. This recipe describes how to use the `Ti.Network.HTTPClient` proxy to connect to a website using basic authentication headers.

Getting ready

This recipe uses the `Ti.BA` CommonJS module as an assist to Titanium's native `Ti.Network.HTTPClient`. This module and other code assets can be downloaded from the source provided by the book. Installing this module in your project is straightforward. Simply copy the `Ti.BA.js` file into `Resources` folder of your project as shown in the following screenshot:



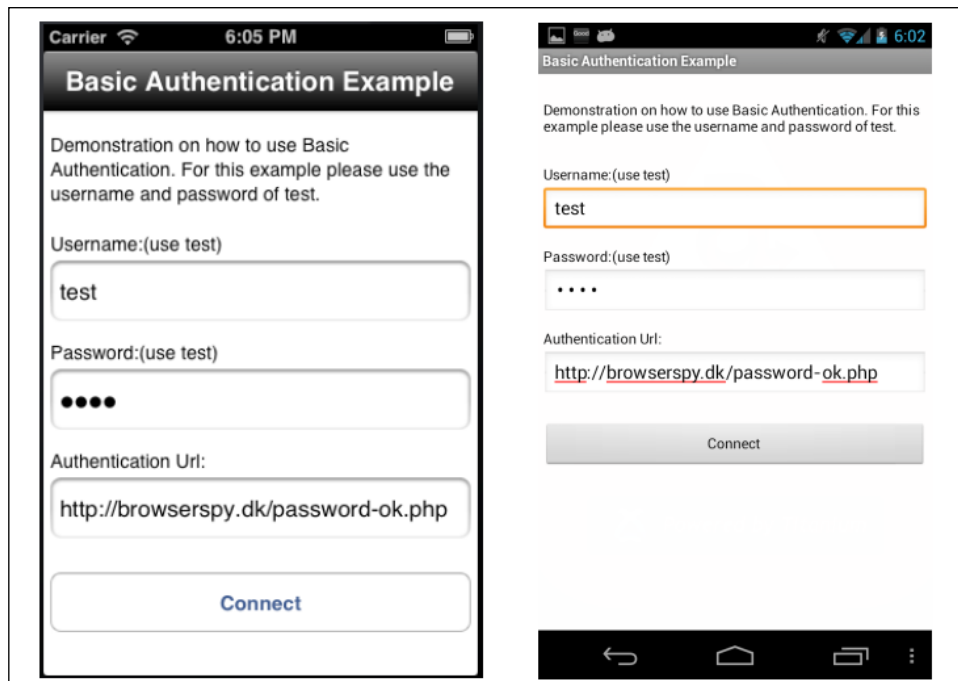
How to do it...

Once you have added the `Ti.BA.js` file to your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  mod : require('Ti.BA')
};
```

Creating the recipe UI

This recipe uses a simple UI to illustrate how to establish a basic authenticated network connection. The code snippets in this section walk through how to construct the basic authentication testing app shown in the following screenshots:



Now perform the following steps:

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'AES Crypto Example',
  barColor: '#000', layout: 'vertical', fullscreen: false
});
```


2. Next the `txtUsername` `Ti.UI.TextField` is added to the `Ti.UI.Window`. The default value is set to `test` since it is the default for the `browserspy.dk` test site used to create our mock test connection.

```
var txtUsername = Ti.UI.createTextField({
  value:'test',hintText:'Enter username',
  height:45, left:5, right:5,
  borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtUsername);
```

3. The `txtPassword` `Ti.UI.TextField` is then added to the `Ti.UI.Window`. The default value is set to `test` since it is the default for the `browserspy.dk` test site used to create our mock test connection.

```
var txtPassword = Ti.UI.createTextField({
  value:'test',hintText:'Enter password',
  passwordMask:true,height:45, left:5, right:5,
  borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtPassword);
```

4. Next the `txtUrl` `Ti.UI.TextField` is added to the `Ti.UI.Window`. The default value is set to `"http://browserspy.dk/password-ok"`. The value for `txtUrl` can be updated to any service requiring basic authentication. For demonstration purposes, the recipe uses the `browserspy.dk` test site.

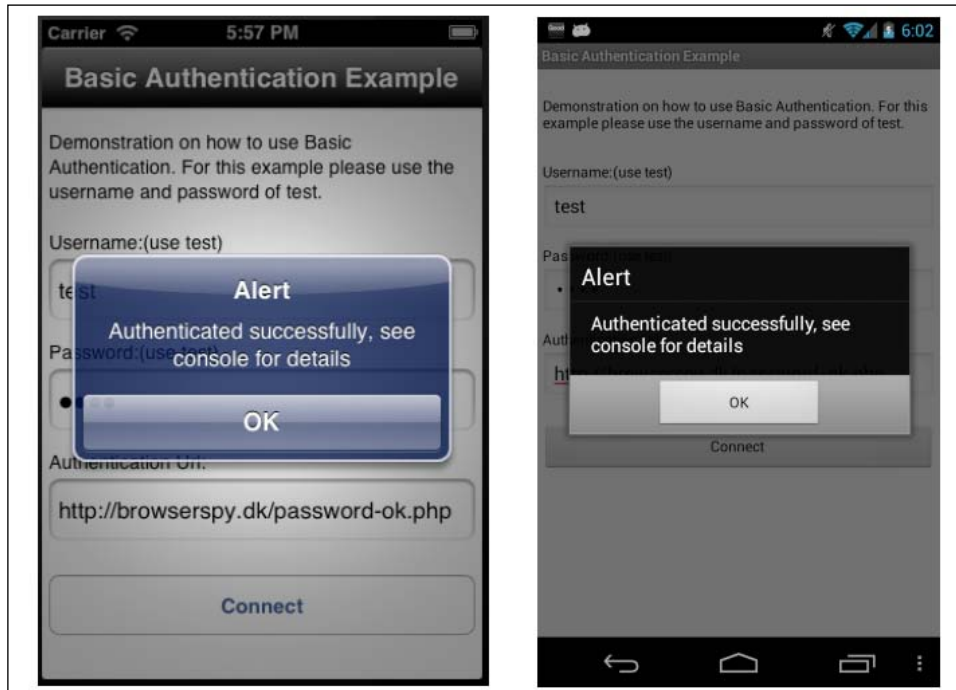
```
var txtUrl = Ti.UI.createTextField({
  value:'http://browserspy.dk/password-ok.php',
  hintText:'Enter Url',
  height:45, left:5, right:5,
  borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtUrl);
```

5. Finally the `btnConnect` `Ti.UI.Button` is added to the `Ti.UI.Window`. Later in this recipe, the `btnConnect` control will be used to initialize the authenticated network connection.

```
var btnConnect = Ti.UI.createButton({
  title:'Connect', top:25, height:45, left:5, right:5
});
win.add(btnConnect);
```

Creating a service connection

With the sample UI now in place, the next step in the recipe is to perform a secure network connection using basic authentication. The following snippets are used to create a network connection and display the results shown in the following screenshots:



A basic authenticated test network connection is performed when the `click` event for the `btnConnect` control is fired as shown in the following code snippet:

```
btnConnect.addEventListener('click', function(x) {
```

1. First, the `Ti.Network.online` property is checked to ensure the device has a network connection. If no network connection is available, the user is alerted and the function is exit.

```
    if(!Ti.Network.online){
        alert('Network connection is required');
        return;
    }
}
```

2. Next the `onDone` function is created. This function is used as a callback for the results of the authentication network connection.

```
function onDone(d) {
```

3. When in the `onDone` function, the first action performed is to check the results provided in the `d.success` property. If this property is `true`, the network connection was successfully completed. Otherwise, the `d.error` message is printed to the console and the user is notified.

```
    if(d.success) {
        Ti.API.info('Results = ' +
            JSON.stringify(d.results));
        alert('Authenticated successfully');
    }else{
        Ti.API.info('error = ' + JSON.stringify(d.error));
        alert('Authenticated Failed');
    }
};
```

4. The `credentials` object is then created. This contains the username and password to be used when creating our authenticated connection. Please note these values should be in plain text as demonstrated in taking the values directly from the `Ti.UI.TextField` controls on the screen.

```
var credentials = {
    username:txtUsername.value,
    password:txtPassword.value
};
```

5. The `Ti.BA` module allows you to configure all of the `Ti.Network.HTTPClient` options along with controlling and specifying the service's output format. The following snippet demonstrates how to configure the request to set the timeout to 9 seconds, and specify the `responseText` be the returned response.

```
var options = {format:'text', timeout:9000};
```

6. Finally a new instance of the `Ti.BA` module is created and the following are provided:
 - **Method:** The `GET` or `POST` action is performed. In the case of this sample, the `POST` method is provided.
 - **URL:** The URL the module uses to connect.
 - **Credentials:** The `credentials` object contains the username and password that will be used to create the basic authenticated connection.
 - **Callback:** This parameter requires a callback method to be provided. In this sample, the `onDone` method is provided to the module and will return the connection response.

- **Options:** These are the configuration options for the `Ti.Network.HTTPClient` and result type to be returned. If none is provided, the module's default values are returned.

```
var basicAuth = new
my.mod('POST',txtUrl.value,credentials, onDone,options);
});
```

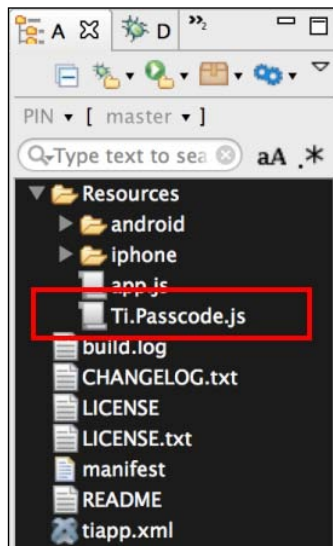
Implementing a cross-platform passcode screen

Password and passcode screens are common verification instruments for enterprise apps. These screens provide interaction patterns that help solve both authentication and confirmation scenarios.

This recipe demonstrates how to implement a cross-platform passcode screen similar to those seen on iOS and in the Android networking screens.

Getting ready

This recipe uses the `Ti.Passcode` CommonJS module to implement a cross-platform passcode screen. This module and other code assets can be downloaded from the source provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing this module in your project is straightforward. Simply copy the `Ti.Passcode.js` file into the `Resources` folder of your project as shown in the following screenshot:



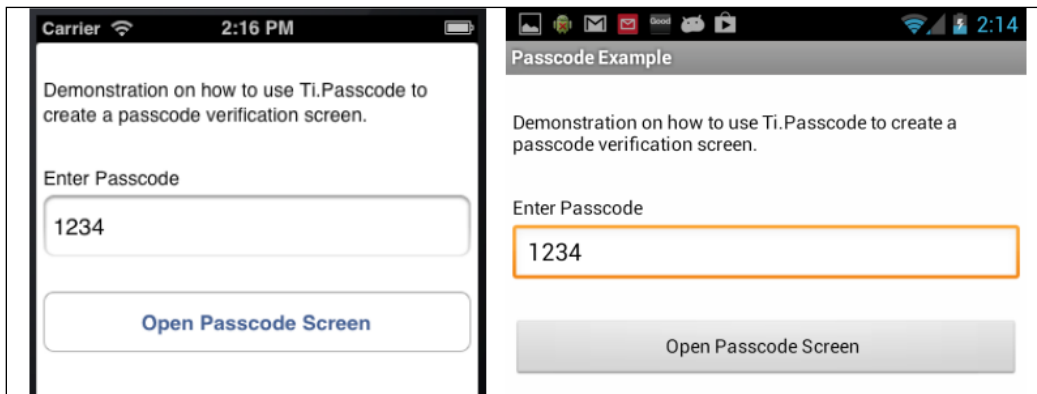
How to do it...

Once you have added the `Ti.Passcode.js` file to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code as the following code demonstrates:

```
//Create our application namespace
var my = {
  mod : require('Ti.Passcode')
};
```

Creating the recipe UI

This recipe uses `Ti.UI.Windows` to present information to the user. The code used to create the first `Ti.UI.Window` is discussed in this section. This section details how to launch the `Ti.Passcode` module and provides a code to have it verified.



Now perform the following steps:

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Passcode Example',
  layout:'vertical',fullscreen:false, exitOnClose:true
});
```

2. Next the `txtPasscode` `Ti.UI.TextField` is added to the `Ti.UI.Window`. This control is used to collect the passcode that the `Ti.Passcode` module will verify against.

```
var txtPasscode = Ti.UI.createTextField({
  value:'1234',hintText:'Enter Passcode',
  height:45, left:5, right:5,
  borderStyle:Ti.UI.INPUT_BORDERSTYLE_ROUNDED
```

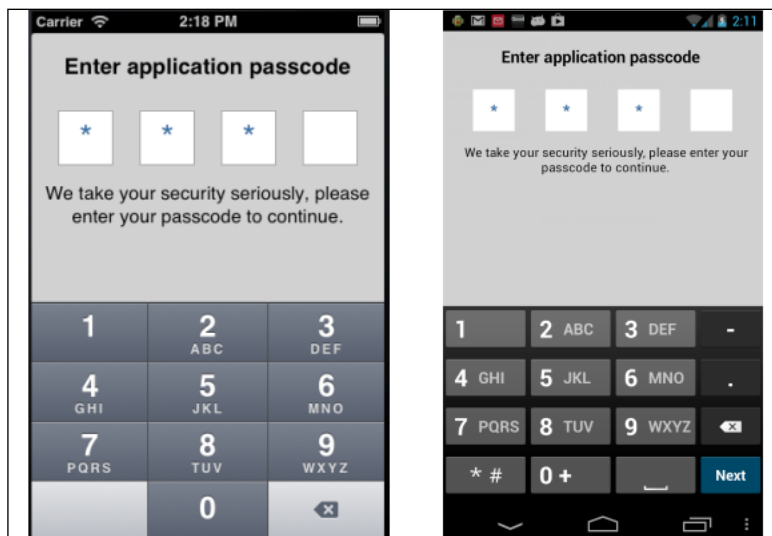
```
});
win.add(txtPasscode);
```

3. Finally the `btnRunPasscode` `Ti.UI.Button` is added to the `Ti.UI.Window`. This button will be used later in the recipe to launch the passcode screen.

```
var btnRunPasscode = Ti.UI.createButton({
  title:'Open Passcode Screen', top:25, height:45,
  left:5, right:5
});
win.add(btnRunPasscode);
```

Launching the passcode screen

The second `Ti.UI.Window` in this recipe is created by the `Ti.Passcode` module. This `Ti.UI.Window` is responsible for the presentation and verification of the application passcode. This section describes how to configure, create, and confirm your app passcode using this module's display element shown in the following screenshot:



The passcode verification screen is launched when the user taps the `btnRunPasscode` `Ti.UI.Button` and triggers the `click` event to be fired. The following code snippet discusses the actions performed after the `click` event is fired:

```
btnRunPasscode.addEventListener('click', function(x) {
```

1. The first code block in the `click` event of the `btnRunPasscode` button is to create our configuration and other variables. The `options` object contains all of the settings needed to configure the `Ti.Passcode` window.

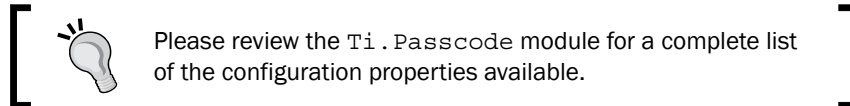
```
var codeWindow = null,  
    options = {
```

2. The window configuration element allows for all of the standard `Ti.UI.Window` properties to be used. The following snippet shows how to set the `backgroundColor` and `navBarHidden` properties on the passcode window.

```
    window:{  
        backgroundColor:'#ccc',  
        navBarHidden:true  
    },
```

3. The `view` configuration element allows for the configuration of a majority of the components displayed in the `Ti.Passcode` window. The following snippet demonstrates how to set the `passcode` title property and code for displaying the error message:

```
    view:{  
        title:'Enter application passcode',  
        errorColor:'yellow'  
    }  
};
```



4. Next the `onCompleted` callback function is defined. This function will be used as the callback method provided to the `Ti.Passcode` module. The passcode verification results will be provided as a dictionary to the `d` input parameters.

```
function onCompleted(d) {
```

5. The `d` argument is an object with the verification results. The `d.success` property provides an indicator if the entered passcode matches the passcode provided on launch. The following code snippet alerts the user if they have entered the correct passcode or if they need to try the process again.

```
    if(d.success) {  
        alert('Passcode entered is correct');  
        codeWindow.close();  
    }else{  
        Alert('Invalid passcode, please try again');  
    }  
};
```

6. The next step in this section of the recipe is to create a new instance of the passcode module. The following snippet demonstrates how to do this:

```
var passcode = new my.mod();
```

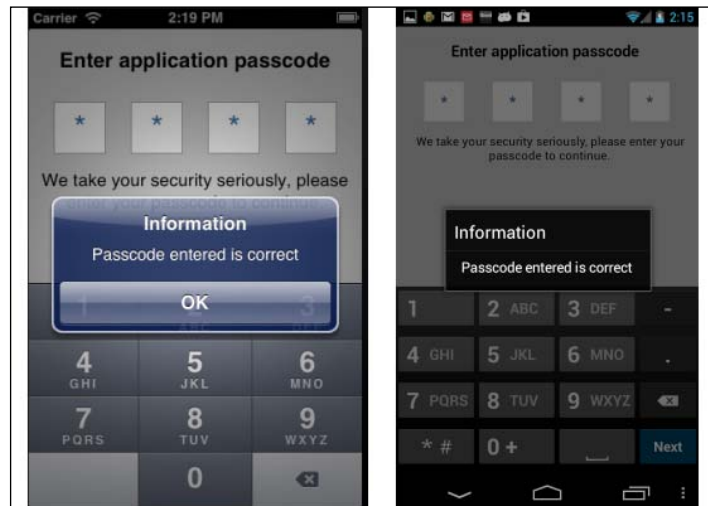
7. The `createWindow` method is then called on the new passcode instance. This method provides the passcode to verify from the `txtPasscode` control, and also provides the callback and options objects created earlier. This method then returns a `Ti.UI.Window`. Once the passcode has been entered, the callback method `onCompleted` will send the verification results.

```
var codeWindow =
passcode.createWindow(txtPasscode.value,
onCompleted,options);
```

8. The `Ti.UI.Window` returned by the `createWindow` method is then opened with a modal flag. The passcode `Ti.UI.Window` will remain open until closed by the `onCompleted` callback method as discussed earlier.

```
codeWindow.open({modal:true});
});
```

The following screenshots illustrate how this section of the code looks on the device after the user has successfully entered his/her passcode.



See also

- ▶ This recipe uses the `Ti.Passcode` CommonJS module. To read the documentation and to learn more about this project, please visit <https://github.com/benbahrenburg/Ti.Passcode>.

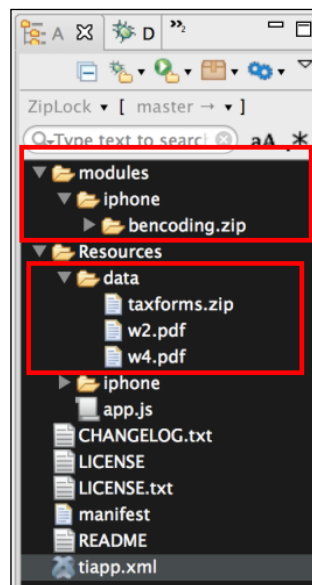
Working with protected ZIP files on iOS

Protected zip files are a common way to exchange, store, and transmit enterprise data. ZIP files are often used to bundle several files together for transmission or storage. As an extra layer of security, all such files should always be protected with a strong password.

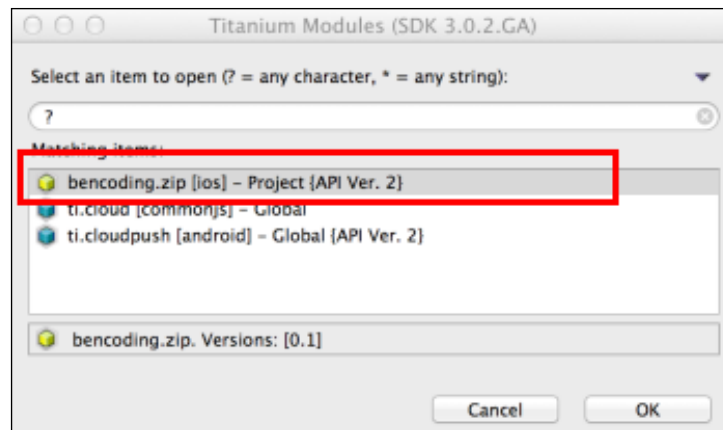
The following recipe discusses how to create and unzip protected compressed files on iOS.

Getting ready

This recipe uses the `bencoding.zip` native module to work with protected zip files. This module and other code assets can be downloaded from the source provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing this module in your project is straightforward. Simply copy the `modules` folder into the root of your project, and then copy the `data` folder into the `Resources` directory of your project as shown in the following screenshot:



After copying the mentioned folders, you will need to click on your `tiapp.xml` file in Titanium Studio and add a reference to the `bencoding.zip` module as shown in the following screenshot:



How to do it...

Once you have added the modules and data folders to your project, you need to create your application namespaces in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  zip : require('bencoding.zip')
};
```

Creating the recipe UI

This recipe uses a simple UI to illustrate how to create (zip) and unzip protected ZIP files. The code discussed in this section walks through how to construct the recipe's UI shown in the following screenshot:



The following steps outline how to create the recipe's UI:

1. First, a `Ti.UI.Window` is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Protected Zip Example',
  barColor: '#000', layout: 'vertical'
});
```

2. Next add the `txtPassword` `Ti.UI.TextField` control. This control will be used to provide the password to create protected ZIP files or to unzip them. The default is set to `foo123`. This is also the password for the included sample file.

```
var txtPassword = Ti.UI.createTextField({
  value: 'foo123', hintText: 'Enter Password',
  height: 45, left: 5, right: 5, passwordMask: true,
  borderStyle: Ti.UI.INPUT_BORDERSTYLE_ROUNDED
});
win.add(txtPassword);
```

3. Then the `btnZip` `Ti.UI.Button` is added to the `Ti.UI.Window`. This button will be used to demonstrate how to create a protected ZIP file later in the discussion of this recipe.

```
var btnZip = Ti.UI.createButton({
  title: 'Zip with Password',
  top: 25, height: 45, left: 5, right: 5
});
win.add(btnZip);
```

4. Next the `btnUnzip` `Ti.UI.Button` is added to the `Ti.UI.Window`. This button will be used to demonstrate how to unzip a protected ZIP file later in the discussion of this recipe.

```
var btnUnzip = Ti.UI.createButton({
  title: 'Unzip with Password', top: 25, height: 45,
  left: 5, right: 5
});
win.add(btnUnzip);
```

Creating a password-protected ZIP file

This section of this recipe demonstrates how to create a protected ZIP file. The following screenshot shows this recipe in action, creating a protected ZIP file and alerting the user of the output file path:



The created ZIP file is executed when the user clicks on the `btnZip` `Ti.UI.Button` and triggers the `click` event to be fired. The following code snippet discusses the actions performed after the `click` event is fired:

```
btnZip.addEventListener('click', function(x) {
```

1. The first step in the zip process is to create the `onZipCompleted` callback method. When the zip process is completed, the module will send the results to this callback method:

```
function onZipCompleted(d) {
```

2. The `d` method parameter provides the results from the module. The first step in processing the module results is to check the `d.success` property to see if the zip process was successful. If so, the user is alerted of the path of the completed ZIP file. Otherwise, the user is alerted to the error generated in creating the file.

```
    if(d.success){
        alert('Protected zip created at: ' + d.zip);
    }else{
        alert('failed due to: ' + d.message);
    }
};
```

3. Next the `writeToZip` and `inputDirectory` variables are created. The `writeToZip` variable contains the path to the `taxforms.ZIP` output file in the app's `Documents` directory. The `inputDirectory` creates a reference to the `Resources/data` created during the *Getting ready* section of this recipe.

```
var writeToZip = Ti.Filesystem.applicationDataDirectory +
  '/taxforms.zip';
var inputDirectory = Ti.Filesystem.resourcesDirectory +
  'data/';
```

4. Finally the `zip` method is called and this method provides the parameters built earlier in the `click` event handler. Once completed, the `zip` method provides results to the provided `onZipCompleted` callback method.

```
my.zip.zip({
  zip: writeToZip,
  password:txtPassword.value,
  files: [inputDirectory + 'w2.pdf',
  inputDirectory + 'w4.pdf'],
  completed:onZipCompleted
});
```



The `files` parameter provides an array of files using the `inputDirectory` variable. In this example, the files included are the `w2.pdf` and `w4.pdf` files included in the `Resources/data` folder created during the *Getting ready* section of this recipe.

Unzipping a protected ZIP file

This section of the current recipe demonstrates how to unzip a protected ZIP file. The following screenshot shows this recipe in action, unzipping the contents of a protected file into the app's `Documents` directory:



The unzip example is executed when the user taps on the `btnUnzip` `Ti.UI.Button` and triggers the `click` event to be fired. The following code snippet discusses the actions performed after the `click` event is fired.

```
btnUnzip.addEventListener('click', function(x) {
```

1. The first step in the unzip process is to create the `onUnzipCompleted` callback method. When the unzip process is completed, the module will send the results to this callback method.

```
function onUnzipCompleted(e) {
```

2. The `d` method parameter provides the results from the module. The first step in processing the module results is to check the `d.success` property to see if the unzip process was successful. If so, the user is alerted of the directory path of the unzipped file. Otherwise, the user is alerted to the error generating the file.

```
    if(e.success) {
        alert('Unzipped to ' + e.outputDirectory);
    } else {
        alert('failed due to: ' + e.message);
    }
};
```

3. The `outputDirectory` and `zipFileName` variables are created next. The `outputDirectory` variable contains the path to the output directory in the app's Documents directory. The `zipFileName` creates a reference to the `Resources/data/taxform.zip` file created during the *Getting ready* section of this recipe.

```
var outputDirectory =  
Ti.Filesystem.applicationDataDirectory;  
var zipFileName = Ti.Filesystem.resourcesDirectory +  
'data/taxforms.zip';
```

4. Finally, the `unzip` method is called and this method provides the parameters built earlier in the `click` event handler. Once completed, the `unzip` method provides results to the provided `onUnzipCompleted` callback method.

```
my.zip.unzip({  
  outputDirectory:outputDirectory,  
  zip:zipFileName,  
  overwrite:true,  
  completed:onUnzipCompleted  
});  
});
```



All files contained within the `zipFileName` ZIP file will be unzipped to the root of the directory provided in the `outputDirectory` parameter.

See also

- ▶ This recipe uses the `benCoding.Zip` native module. To read the documentation and to learn more about this project, please visit <https://github.com/benbahrenburg/Zipper>.

9

App Security Using Encryption and Other Techniques

In this chapter we will cover:

- ▶ Using secure properties
- ▶ Object and string encryption
- ▶ Working with encrypted files
- ▶ Handling protected PDFs on iOS
- ▶ Android lock screen monitor

Introduction

It is common for Enterprise applications to contain private or confidential information. For this reason, encryption, file locking, and secure app lifecycle management are fundamental requirements when developing Enterprise Titanium apps. The core Titanium SDK provides limited functionality in this area such as one-way hashing and basic app events, but to fully meet security requirements, third-party modules such as *Securely* are needed.

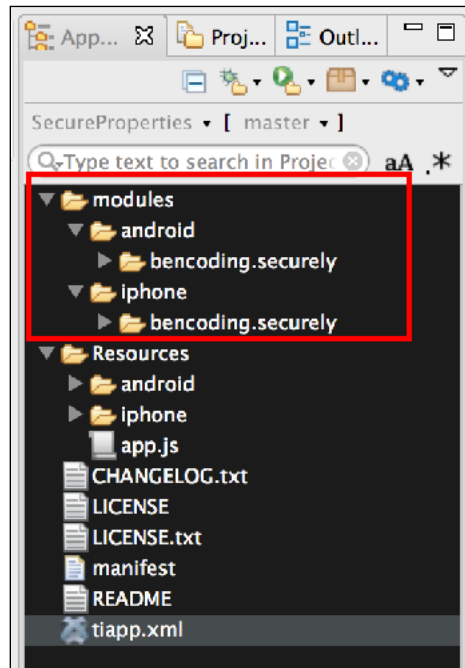
In this chapter, we will discuss how to use the *Securely* framework to handle common secure programming tasks such as file and string encryption. *Securely* provides access to each platform's security APIs in a cross-platform and Titanium-friendly way. Through a series of recipes, we will demonstrate how to leverage the *Securely* framework within our existing Titanium Enterprise app.

Using secure properties

The Titanium SDK provides a `Ti.App.Properties` object, which provides a convenient way to persist user and application information. The `Securely` framework provides a familiar API designed to mirror `Ti.App.Properties`, which allows you to persist this information in a secure fashion. This recipe describes how to use the `Securely.Properties` object to store, read, and remove data in/from an encrypted and secure manner.

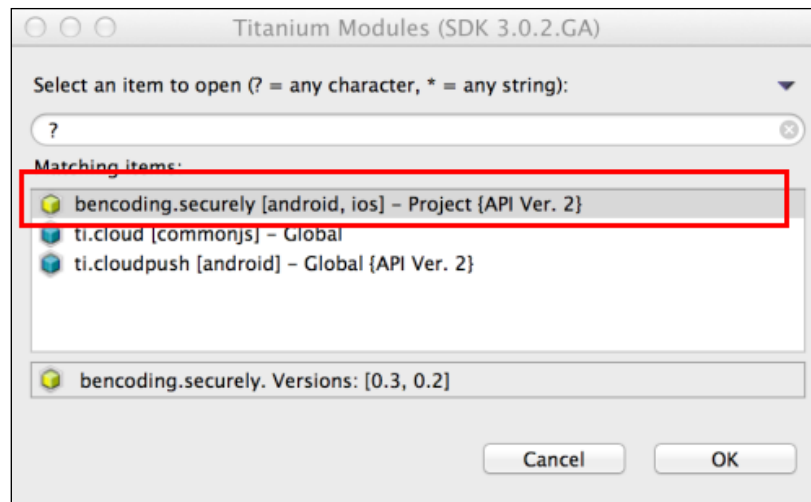
Getting ready

This recipe uses the `Securely` native module. This module and other code assets can be downloaded from the source provided by the book, or individually through the links provided in the *See also* section at the end of this recipe. Installing these in your project is straightforward. Simply, copy the `modules` folder into your project as shown in the following screenshot:



Adding the module reference

After copying the mentioned folder, you will need to click on your `tiapp.xml` file in Titanium Studio and add a reference to the `bencoding.securely` module as shown in the following screenshot:



How to do it...

This recipe is designed to run within the context of a `Ti.UI.Window` or other component within a single Titanium context. The code samples in this section demonstrate how to use the `Secure` properties of `Securely` using the same tests that Appcelerator uses for the Titanium SDK's `Ti.App.Properties` class. For more information, please refer to the `app.js` included with the recipe's source.

Creating the namespace

Once you have added the `Securely` module to your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  secure : require('bencoding.securely'),
  isAndroid : Ti.Platform.osname === 'android',
  testObjects:{
    testList : [
      {name:'Name 1', address:'1 Main St'},
      {name:'Name 2', address:'2 Main St'}
    ]
  }
};
```

Creating the secure properties object

After the app namespace has been created, the next step is to create a new properties object. This object contains the following property values that must be set at creation time:

- ▶ `secret`: This is a required parameter. `secret` is the password used to encrypt and decrypt all property values. The same `secret` used to encrypt must be used during the decryption process or a `null` value will be returned.
- ▶ `identifier`: This parameter is optional. If no value is provided, the bundle name on iOS or the `PackageName` on Android is used. `identifier` allows you to segment each property with an identifier, if needed.
- ▶ `accessGroup`: This parameter is an optional value used on the iOS platform. Access groups can be used to share keychain items among two or more applications. If no access group is provided, the keychain values will only be accessible within the app saving the values.
- ▶ `encryptFieldNames`: This parameter is an optional value only used on the Android platform. When set to `true`, Securely will create an MD5 hash using the provided `secret` for all property names.

```
var properties = my.secure.createProperties({
  secret:"ssh_dont_tell",
  identifier:"myPropertyIdentifier",
  accessGroup:"myAccessGroup",
  encryptFieldNames:false
});
```

Result comparison helper

The `displayResults` function shown in the following code snippet is used to compare the test result with the expected value. Based on the comparison, the proper message is generated for presenting to the user.

```
function displayResults(result, expected) {
  if (result instanceof Array) {
    return displayResults(JSON.stringify(result),
      JSON.stringify(expected));
  }else{
    if (result === expected) {
      return "Success (" + result + "==" + expected + ")";
    } else {
      return "Fail: " + result + "!=" + expected;
    }
  }
};
```

Reading secure properties without defaults

Each supported property type has a `get` method. For example, to read a Boolean property, the `getBool` method is called and a name is provided. This API works similar to the `Ti.App.Properties` object within the Titanium SDK, with added support for reading and decrypting secure properties. If no stored value is available, a null or default value type is provided. The following snippet demonstrates how to read saved secure property values:

```
Ti.API.info('Bool: ' + displayResults(properties.getBool('whatever'),
(my.isAndroid ? false : null)));

Ti.API.info('Double: ' +
displayResults(properties.getDouble('whatever'),
(my.isAndroid ? 0: null)));

Ti.API.info('int: ' + displayResults(properties.getInt('whatever'),
(my.isAndroid ? 0 : null)));

Ti.API.info('String: ' +
displayResults(properties.getString('whatever'),null));

Ti.API.debug('StringList: ' +
displayResults(properties.getList('whatever'),null));
```



On iOS, null values are returned for any property without a saved value. Due to Android's type system, Boolean values will return `false` if no value is stored and numeric values will return zero. All other Android values will return null similar to iOS.

Reading secure properties with defaults

Also similar to `Ti.App.Properties`, `Securely` provides the ability to read and decrypt a secure property and provide a default value if there is no stored value for the requested secure property.

```
Ti.API.info('Bool: ' +
displayResults(properties.getBool('whatever',true),true));

Ti.API.info('Double: ' +
displayResults(properties.getDouble('whatever',2.5),2.5));

Ti.API.info('int: ' +
displayResults(properties.getInt('whatever',1),1));
```

```
Ti.API.info('String: ' +
displayResults(properties.getString('whatever',"Fred"),"Fred"));

Ti.API.debug('StringList: ' +
displayResults(properties.getList('whatever'),testList));
```



The default value is not encrypted or persisted during the read process.

Setting secure property values

Each supported property type has a `set` method used to encrypt and persist values. For example, to save and encrypt a Boolean property and the `setBool` method, and provide a property name and Boolean value. This API works similar to the `Ti.App.Properties` object within the Titanium SDK. `Securely` supports both encrypting and writing the value to the secure properties directly. The following code snippet demonstrates how to save the following values to secure and encrypted storage:

```
properties.setString('MyString','I am a String Value ');
properties.setInt('MyInt',10);
properties.setBool('MyBool',true);
properties.setDouble('MyDouble',10.6);
properties.setList('MyList',my.testObjects.testList);
```

To demonstrate the properties were saved correctly, the `get` method is called for each file and the results are printed to the console within Titanium Studio.

```
Ti.API.info('MyString: '+ properties.getString('MyString'));
Ti.API.info('MyInt: '+ properties.getString('MyInt'));
Ti.API.info('MyBool: '+ properties.getString('MyBool'));
Ti.API.info('MyDouble: '+ properties.getString('MyDouble'));
var list = properties.getList('MyList');
Ti.API.info('List: ' + JSON.stringify(list));
```

Listing secure property field names

The property names can be returned as an array by calling the `listProperties` method on the `Securely` property object. The following code snippet demonstrates how to use this method to print a JSON representation of the `names` array to the console within Titanium Studio.

```
if(!properties.hasFieldsEncrypted()){
    var allProperties = properties.listProperties();
    Ti.API.info(JSON.stringify(allProperties));
}
```



If field name encryption is enabled, the `listProperties` method will return `null`. As the field names are encrypted with a one-way hash, the original names are no longer available.

Removing secure properties

You can remove properties using two methods on the `Securely` properties object. The `removeProperty` and `removeAllProperties` are designed to be familiar and mirror the methods with the same name on the Titanium SDK's `Ti.App.Properties` object.

The `removeProperty` method will remove the provided secure property name, if it exists.

```
properties.removeProperty('MyInt');
```

The `removeAllProperties` method will remove all properties within the identifier provided when creating the properties object.

```
properties.removeAllProperties();
```



In our recipe, the `remove` property functions are called at the end of the test so that the results can be recreated reliably each time.

Check if a secure property exists

The `Securely` property object provides the `hasProperty` method to provide the ability to check if a secure property exists. If the property exists, a Boolean `true` value is returned, otherwise a result of `false` is provided. This API is designed to be familiar as it mirrors the `Ti.App.Properties.hasProperty` function within the core Titanium SDK.

```
Ti.API.info("Does MyInt property exist? " +
properties.hasProperty('MyInt'));
```

How it works...

The underlying infrastructure for secure properties is implemented differently depending on the platform running your application. Although `Securely` provides a cross-platform API, it is important to understand how each platform has been implemented and its associated security considerations.

Secure properties on iOS

The *Securely* framework saves all property values as serialized strings within the iOS Keychain. This provides secure storage and since it is part of iOS does not require any dependencies. Since *Securely* uses the iOS keychain service, it is recommended that your organization review Apple's Keychain documents before storing sensitive data within the securely-managed container.



It is important to remember since the iOS Keychain service is used, your Secure Property values will still be available within the iOS Keychain after your app has been uninstalled. The `removeAllProperties` method must be called if you wish to remove all keychain items before removal of the app.

Secure properties on Android

The *Securely* framework saves all property values as serialized and AES-encrypted strings into Android's `SharedPreferences`. Although Android introduced native keychain support in API level 14, the *Securely* module was designed to accommodate a larger number of devices and targets API level 8 and higher. It is recommended that your Enterprise reviews the secure properties' implementation within *Securely* to ensure it is in compliance with your corporate or industry standards and requirements.

Secure property considerations

By default on Android, the property names are not encrypted. This can be enabled by setting the `encryptFieldNames` on creation of the `properties` object. Due to the need to encrypt all property names, this property can only be set on creation of a new `properties` object. When field name encryption is enabled, *Securely* will create a SHA-1 hash for each field name using the `secret` property value provided. Enabling this feature creates performance considerations. Regression and performance testing is recommended before implementing your existing Titanium app.

See also

- ▶ To learn more about the iOS Keychain, please review Apple's documentation available at <https://developer.apple.com/library/mac/#documentation/security/Conceptual/keychainServConcepts/01introduction/introduction.html>.
- ▶ To learn more about Android's `SharedPreferences`, please review the documentation available at <http://developer.android.com/reference/android/content/SharedPreferences.html>.

Object and string encryption

During the Enterprise Titanium development cycle, there is often the requirement to encrypt in-process or persisted JavaScript objects or variables. The `Securely` framework provides a `StringCrypto` proxy with convenience methods for key generation of AES and DES bi-directional encryption.

This recipe describes how to use the `Securely.StringCrypto` object to encrypt and decrypt JavaScript objects in a secure manner.



AES and DES implementations of `Securely` are designed to be specific to the device platform. If there is a need to exchange AES or DES encrypted data, access device, platform or to a third-party service, testing is recommended to verify the implementations match.

Getting ready

This recipe uses the `Securely` native module. This module and other code assets can be downloaded from the source provided by the book. Installing these in your project is straightforward. Simply copy the `modules` folder into the root of your Titanium Project. Please review the *Getting ready* section of the *Using secure properties* recipe for instructions on module setup before continuing.

How to do it...

This example is designed to run within the context of a `Ti.UI.Window` or other component within a single Titanium context. This section demonstrates how to use the `StringCrypto` method of `Securely` to encrypt JavaScript objects. For more information, please reference the `app.js` included with the recipe's source.

Creating the namespace

Once you have added the `Securely` module to your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  secure : require('bencoding.securely'),
  isAndroid : Ti.Platform.osname === 'android'
};
```


Generating keys

Key generation is an important part of cryptography and to assist with this process, Securely has two built-in convenience methods.

Generating a derived key

The first key generation convenience method is called `generateDerivedKey`. This method includes the string input provided by the user into the salt algorithm used to determine the key. This approach is helpful if the seed value needs to be known or derived by another accessing system. The following steps demonstrate two common approaches for generating the seed value to be provided to the `generateDerivedKey` method:

- ▶ A common approach is to create a new GUID as the seed from which the key is generated. The following code snippet demonstrates this approach using the `Ti.Platform.createUUID` method to generate a GUID:

```
var usingGUID =  
my.secure.generateDerivedKey(Ti.Platform.createUUID());
```



The `generateDerivedKey` method will use the parameter provided to create a new key each time it is called.

- ▶ Another, less random method of key generation is to provide the Titanium app GUID as the seed value. The following code snippet demonstrates how to use the `Ti.App.guid` to create a derived key using the GUID from your project's `tiapp.xml` file.

```
var usingAppID = my.secure.generateDerivedKey(Ti.App.guid);
```

Generating a random key

The second key generation convenience method is called `generateRandomKey`. As its name indicates, a random alphanumeric string is generated and used as the seed value. The following code snippet demonstrates how to create a key value using the `generateRandomKey` method.

```
var randomKey = my.secure.generateRandomKey();
```

Creating the stringCrypto object

The next step in the cryptographic process for strings and objects is to create a new instance of the `stringCrypto` proxy. The following code snippet shows how to create a new proxy object named `stringCrypto`.

```
var stringCrypto = my.secure.createStringCrypto();
```

Using DES encryption

Securely supports the older **Data Encryption Standard (DES)** encryption algorithm. Support for this algorithm is primarily provided for intercommunication with legacy systems. Wherever possible, the stronger AES encryption should be used instead.

Encrypting using DES

The `DESEncrypt` method requires a key and a string to encrypt. This method will then return a DES encrypted string. If an error is generated during the encryption process a null value is returned. The following demonstrates how to encrypt both a JavaScript string and object using this method.

```
var desEncryptedString = stringCrypto.DESEncrypt
    (usingGUID,
    plainTextString);
var desEncryptedObject = stringCrypto.DESEncrypt
    (usingGUID,
    JSON.stringify(plainObject));
```

Any non-JavaScript string elements must first be converted into a JavaScript string before being provided to the `DESEncrypt` function.

Decrypting using DES

The `DESDecrypt` method is used to decrypt a string encrypted with the DES algorithm. This method requires the key and a string with the encrypted value. The `DESDecrypt` method will then return a string with the decrypted value. The following snippet demonstrates how to use the `DESDecrypt` method to decrypt both a string and object.

```
var desDecryptedString = stringCrypto.DESDecrypt
    (usingGUID,
    desEncryptedString);
```

The following code snippet demonstrates how to use `JSON.parse` to re-build the JavaScript object from the decrypted JSON string.

```
var desDecryptedObject = JSON.parse
    (stringCrypto.DESDecrypt (usingGUID,
    desEncryptedObject));
```

Encrypting using AES

The `AESDecrypt` method requires a key and a string to encrypt. This method will then return an AES-encrypted string. If an error is generated during the encryption process, a null value is returned. The following code snippet demonstrates how to encrypt both a JavaScript string and object using this method.

```
var aesEncryptedString = stringCrypto.AESEncrypt
  (usingGUID,
  plainTextString);
var aesEncryptedObject = stringCrypto.AESEncrypt
  (usingGUID,
  JSON.stringify(plainObject));
```



Any non-JavaScript string elements must first be converted into a JavaScript string before being provided to the `AESEncrypt` function.

Decrypting using AES

The `AESDecrypt` method is used to decrypt a string encrypted with the AES algorithm. This method requires the key and a string with the encrypted value. The `AESDecrypt` method will then return a string with the decrypted value. The following code snippet demonstrates how to use the `AESDecrypt` method to decrypt both a string and an object.

```
var aesDecryptedString = stringCrypto.AESDecrypt
  (usingGUID,
  aesEncryptedString);
```

The following code snippet demonstrates how to use `JSON.parse` to rebuild the JavaScript object from the decrypted JSON string.

```
var aesDecryptedObject = JSON.parse
  (stringCrypto.AESDecrypt (usingGUID,
  aesEncryptedObject));
```

Titanium object encryption

Titanium SDK objects such as `Ti.UI.View` are not real JavaScript objects, and therefore, cannot be serialized and encrypted effectively. To encrypt Titanium objects, you must first copy all of the Titanium object's properties into a pure JavaScript object and then convert the JavaScript object to a JSON string as shown earlier. During the decryption process, the reverse approach can be taken to recreate the Titanium object.

See also

- ▶ This recipe uses the `Securely` module, for installation details please review the *Getting ready* section of the *Using secure properties* recipe
- ▶ To learn more about DES encryption, please review processing standards publication published by the **National Institute of Standards and Technology (NIST)** available at <http://www.itl.nist.gov/fipspubs/fip46-2.htm>
- ▶ To learn more about AES encryption, please review processing standards publication published by the NIST available at <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- ▶ To learn more about Cryptographic Services used in the iOS implementation, please review Apples documentation available at https://developer.apple.com/library/mac/#documentation/security/Conceptual/cryptoservices/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011172-CH1-SW1
- ▶ To learn more about Cryptographic Ciphers used in the Android implementation, please review the Android documentation available at <http://developer.android.com/reference/javax/crypto/Cipher.html>

Working with encrypted files

File encryption is a fundamental building block for enterprise mobile development. Due to the sensitivity of data collected by most enterprise apps, it is recommended that all persisted files are encrypted.

This recipe demonstrates how to use the `Securely` framework to both encrypt and decrypt files. Through the use of the File Crypto sample, we will provide step-by-step instructions on how to work with local encrypted files from within your Titanium app.

Getting ready

This recipe uses the `Securely` native module. This module and other code assets can be downloaded from the source provided by the book. Simply copy the `modules` folder into the root of your Titanium project. Please review the *Getting ready* section of *Using secure properties* recipe for instructions on module setup before continuing.

After installing the `Securely` module, you need to copy the `PlainText.txt` file into the `Resources` folder of your project. This file will be used by the recipe to create the initial encrypted file.

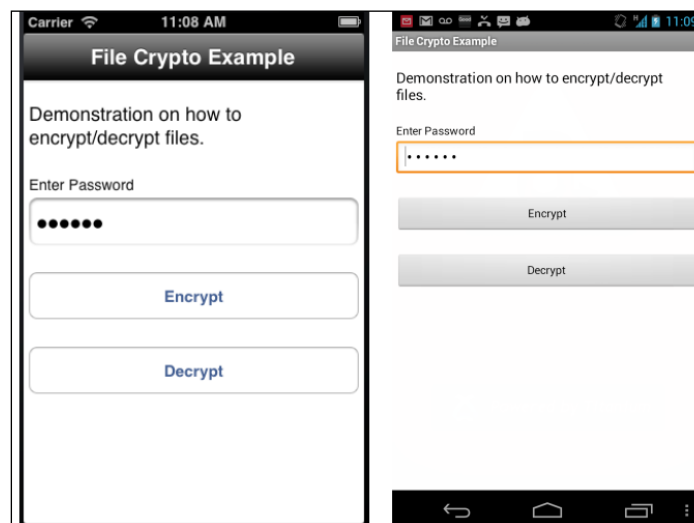
How to do it...

Once you have added the `module` folder and `PlainText.txt` sample file into your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  secure : require('bencoding.securely'),
  isAndroid : Ti.Platform.osname === 'android'
};
```

Creating the UI

This recipe walks through how to use the `Securely` module along with Titanium's `Ti.Filesystem` namespace to encrypt and decrypt files. The test harness pictured in the following screenshot is used to demonstrate how to perform these crypto actions:



Now perform the following steps:

1. The first step in creating the test harness is to create a `Ti.UI.Window`, which is used to attach all UI elements.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'File Crypto Example',
  barColor: '#000', layout: 'vertical', fullscreen: false
});
```

- The next step in creating our test harness UI is to add a `Ti.UI.TextField` named `txtPassword`. This control is used to obtain the password used during the encryption and decryption operations.

```
var txtPassword = Ti.UI.createTextField({
  value:'foo123',hintText:'Enter Password',
  height:45, left:5, right:5, passwordMask:true
});
win.add(txtPassword);
```

- The next step in creating our test harness UI is to add a `Ti.UI.Button` named `btnEncrypt`. This control will be used to launch the file encryption process.

```
var btnEncrypt = Ti.UI.createButton({
  title:'Encrypt', top:25, height:45, left:5, right:5
});
win.add(btnEncrypt);
```

- The final step in creating our test harness UI is to add a `Ti.UI.Button` named `btnDecrypt`. This control will be used to launch the file decryption process. Please note the encryption process launched when the `btnEncrypt` button is tapped must be run first.

```
var btnDecrypt = Ti.UI.createButton({
  title:'Decrypt', top:25, height:45, left:5, right:5
});
win.add(btnDecrypt);
```

Encrypting a file

The file encryption process is demonstrated using the `click` event of the `btnEncrypt` `Ti.UI.Button`. This section describes how to use the `AESEncrypt` method of `Securely` for file encryption using the AES encryption algorithm.

```
btnEncrypt.addEventListener('click',function(x){
```

- The first step in the file encryption process is to create a callback method to receive the results from the `AESEncrypt` method. The following `onEncryptCompleted` method demonstrates how to check for the different results provided during the callback process.

```
function onEncryptCompleted(e){
  if(e.success){
    var test = Ti.Filesystem.getFile(e.to);
    Ti.API.info("Test file contents:\n" +
      (test.read()).text);
  }else{
    alert('failed due to: ' + e.message);
  }
};
```

2. Next a new instance of the `FileCrypto` object of the `Securely` framework must be created.

```
var fileCrypto = my.secure.createFileCrypto();
```

3. Then `Ti.FileSystem.File` objects are created for the input and output files.

```
var plainTextFile = Ti.FileSystem.getFile(
Ti.FileSystem.resourcesDirectory,
'PlainText.txt'),
futureEncrypted = Ti.FileSystem.getFile(
Ti.FileSystem.applicationDataDirectory,
'encryptedFile.txt');
```

4. Finally the `AESEncrypt` method is called with the following parameters:

- `password`: The `password` parameter is the key used in the file encryption process. The same password must be provided later if you wish to decrypt the file.
- `from`: The `from` parameter provides a `nativePath` reference to the file that will be encrypted. Please note the file itself is not encrypted, but simply used as the source to generate an encrypted file at the path provided in the `to` parameter.
- `to`: The `to` parameter provides the `nativePath` reference to where the encrypted file should be generated. The application must be able to write to this file path or an IO exception will be generated.
- `completed`: The `completed` parameter provides a reference to the callback method to be used upon completion of the execution of the `AESEncrypt` method.

```
fileCrypto.AESEncrypt({
    password:txtPassword.value,
    from:plainTextFile.nativePath,
    to:futureEncrypted.nativePath,
    completed:onEncryptCompleted
});
});
```

Decrypting a file

The file decryption process is demonstrated using the `click` event of the `btnDecrypt` `Ti.UI.Button`. The following section describes how to use `AESDecrypt` method of `Securely` for file decryption using the AES encryption algorithm. Please note that the same password used to encrypt the file must be provided in the decryption process.

```
btnDecrypt.addEventListener('click',function(x){
```

1. The first step in the file decryption process is to create a callback method to receive the results from the `AESDecrypt` method. The following `onDecryptCompleted` method demonstrates how to check for the different results provided during the callback process:

```
function onDecryptCompleted(e) {
    if(e.success) {
        var test = Ti.Filesystem.getFile(e.to);
        Ti.API.info("Test file contents:\n" +
            (test.read()).text);
    }else{
        alert('failed due to: ' + e.message);
    }
};
```

2. Next the `Ti.FileSystem.File` objects are created for the input and output files.

```
var encryptedFile = Ti.Filesystem.getFile(
    Ti.Filesystem.applicationDataDirectory,
    'encryptedFile.txt'),
futureDecrypted = Ti.Filesystem.getFile(
    Ti.Filesystem.applicationDataDirectory,
    'decryptedFile.txt');
```

3. Then a new instance of the `FileCrypto` object of `Securely` must be created.

```
var fileCrypto = my.secure.createFileCrypto();
```

4. Finally the `AESDecrypt` method is called with the following parameters:

- `password`: The `password` parameter is the key used in the file decryption process. This password must match the key provided during the file encryption process. If the passwords differ, an error will be provided to the callback method.
- `from`: The `from` parameter provides a `nativePath` reference to the file that will be decrypted. Please note the file itself is not decrypted, but simply used as the source to generate a decrypted file at the path provided in the `to` parameter.
- `to`: The `to` parameter provides the `nativePath` reference to where the decrypted file should be generated. The application must be able to write to this file path or an IO exception will be generated.
- `completed`: The `completed` parameter provides a reference to the callback method to be used upon completion of the execution of the `AESDecrypt` method.

```
fileCrypto.AESDecrypt({
    password:txtPassword.value,
    from:encryptedFile.nativePath,
```



```
        to:futureDecrypted.nativePath,  
        completed:onDecryptCompleted  
    });  
};
```

See also

- ▶ This recipe uses the `Securely` module, for installation details please review the *Getting ready* section of the *Using secure properties* recipe
- ▶ `Securely` uses the `RNCryptor` library on iOS for file encryption. For documentation, licensing, and source, please visit <https://github.com/rnapier/RNCryptor>

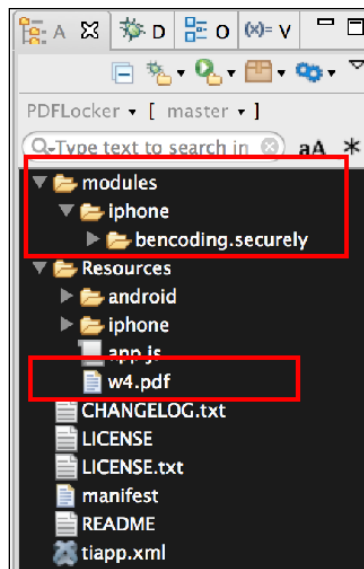
Handling protected PDFs on iOS

Working with and exchanging PDF files is common practice within a majority of organizations. Apple has provided APIs within iOS native access to lock and unlock PDF documents making implementing secure practices for this file format much easier. The `Securely` module exposes these native iOS APIs for your Titanium app to leverage.

This recipe demonstrates how to use the `Securely` framework to lock and unlock PDF files. Through the use of the PDF Locker sample, we will provide step-by-step instructions on how to protect and work with PDFs on the local device from within your Titanium app.

Getting ready

This recipe uses the `Securely` native module. This module and other code assets can be downloaded from the source provided by the book. Installing these in your project is straightforward. Simply copy the `modules` folder into your project as shown in the following screenshot. Next copy the `w4.pdf` file into the `Resources` folder of your project. This file will be used by the recipe to create the initial encrypted file:



After copying the `modules` folder, you will need to update the `tiapp.xml` references as demonstrated in the *Getting ready* section of the *Using secure properties* recipe.

How to do it...

Once you have added the `module` folder and `w4.pdf` sample file into your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  secure : require('bencoding.securely')
};
```

Creating the recipe's UI

This recipe walks through how to use the `Securely` module along with Titanium's `Ti.FileSystem` namespace to lock or unlock PDF files. The test harness pictured in the following screenshot is used to demonstrate how to perform these secure PDF actions:



Now perform the following steps:

1. The first step in creating this test harness is a `Ti.UI.Window`, which is created to attach all UI elements.

```
var win = Ti.UI.createWindow({
    backgroundColor: '#fff', title: 'PDF Protection Example',
    barColor: '#000', layout: 'vertical'
});
```

2. The next step in creating our test harness UI is to add a `Ti.UI.TextField` named `txtPassword`. This control is used to obtain the password used during the encryption and decryption operations.

```
var txtPassword = Ti.UI.createTextField({
    value: 'fool23', hintText: 'Enter Password',
    height: 45, left: 5, right: 5, passwordMask: true
});
win.add(txtPassword);
```

3. The next step in creating our test harness UI is to add a `Ti.UI.Button` named `btnLock`. This control will be used to launch the PDF lock/protection process.

```
var btnLock = Ti.UI.createButton({
    title: 'Lock PDF', top: 25, height: 45, left: 5, right: 5
});
win.add(btnLock);
```

- The next step in creating our test harness UI is to add a `Ti.UI.Button` named `btnUnlock`. This control will be used to launch the PDF unlock or password removal process.

```
var btnUnlock = Ti.UI.createButton({
  title:'Unlock PDF', top:25, height:45, left:5, right:5
});
win.add(btnUnlock);
```

Protecting a PDF file

The PDF protecting or locking process is demonstrated using the `click` event of the `btnLock` `Ti.UI.Button`. The following section describes how to use the `protect` method of `Securely` for PDF locking:

```
btnLock.addEventListener('click',function(x) {
```

- The first step in the file decryption process is to create a callback method to receive the results from the `AESDecrypt` method. The following `onProtected` method demonstrates how to check for the different results provided during the callback process.

```
function onProtected(e) {
  if(e.success){
    alert('Protected PDF file created at: ' + e.to);
  }else{
    alert('failed due to: ' + e.message);
  }
};
```

- Next `Ti.FileSystem.File` objects are created for the input and output files.

```
var inputFile = Ti.FileSystem.getFile(
  Ti.FileSystem.resourcesDirectory,
  'w4.pdf'),
outputFile = Ti.FileSystem.getFile(
  Ti.FileSystem.applicationDataDirectory,
  'locked.pdf');
```

- Then a new instance of the PDF object of `Securely` must be created.

```
var pdf = my.secure.createPDF();
```

- Finally the `protect` method is called with the following parameters:
 - `userPassword`: The `userPassword` parameter is the user-level password for the PDF. This field is required.
 - `ownerPassword`: The `ownerPassword` parameter is the owner-level password for the PDF. Although this is optional, this value must be set in order to password-protect the document.

- **from:** The `from` parameter provides a `nativePath` reference to the PDF file to be protected. Please note that the file itself is not locked, but simply used as the source to generate a protected PDF file at the path provided in the `to` parameter.
- **to:** The `to` parameter provides the `nativePath` reference to where the protected PDF file should be generated. The application must be able to write to this file path or an IO exception will be generated.
- **allowCopy:** The `allowCopy` is a Boolean parameter indicating whether the document allows copying when unlocked with the user password. This parameter defaults to `true` and is optional.
- **completed:** The `completed` parameter provides a reference to the callback method to be used upon completion of the execution of the `protect` method.

```
pdf.protect({
    userPassword:txtPassword.value,
    ownerPassword:txtPassword.value,
    from:inputFile.nativePath,
    to:outputFile.nativePath,
    allowCopy:false,
    completed:onProtected
});
});
```

Unlocking a PDF file

The process of unlocking or removing PDF protection from an existing PDF file is demonstrated using the `click` event of the `btnUnlock` `Ti.UI.Button`. The following steps describes how to use the `punprotect` method of `Securely` for PDF unlocking:

```
btnUnlock.addEventListener('click',function(x){
```

1. The first step in unlocking a protected PDF file is to create a callback method to receive the results from the `unprotect` method. The following `onUnlock` method demonstrates how to check for the different results provided during the callback process.

```
function onUnlock(e){
    if(e.success){
        alert('Unlocked PDF file created at: ' + e.to);
    }else{
        alert('failed due to: ' + e.message);
    }
};
```

2. Next the `Ti.FileSystem.File` objects are created for the input and output files.

```
var protectedFile = Ti.FileSystem.getFile(  
Ti.FileSystem.applicationDataDirectory,  
'locked.pdf'),  
unlockedFile = Ti.FileSystem.getFile(  
Ti.FileSystem.applicationDataDirectory,  
'unlocked.pdf');
```

3. Then, a new instance of the PDF object of `Securely` must be created.

```
var pdf = my.secure.createPDF();
```

4. Finally the `unprotect` method is called with the following parameters:
 - `password`: The `password` parameter is the key used to unlock the protected PDF file. This password must match the owner password used in locking the document.
 - `from`: The `from` parameter provides a `nativePath` reference to the protected PDF file. Please note the PDF file itself is not unlocked, but simply used as the source to generate a new unlocked PDF file at the path provided in the `to` parameter.
 - `to`: The `to` parameter provides the `nativePath` reference to where the unlocked PDF file should be generated. The application must be able to write to this file path or an IO exception will be generated.
 - `completed`: The `completed` parameter provides a reference to the callback method to be used upon completion of the execution of the `unprotect` method.

```
pdf.unprotect({  
password:txtPassword.value,  
from:protectedFile.nativePath,  
to:unlockedFile.nativePath,  
completed:onUnlock  
});  
});
```

See also

- ▶ This recipe uses the `Securely` module. For installation details, please review the *Getting ready* section of the *Using secure properties* recipe.

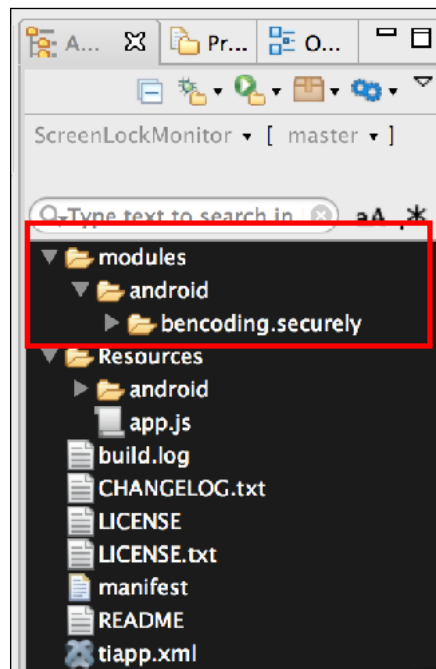
Android lock screen monitor

Due to the inherent nature of Titanium Android architecture, it can be challenging to determine when the app has been placed into the background or the lock screen has been activated. These actions are important life cycle events to track for password and application access. For example, you might wish to display a login screen within your app if the user has locked their device since last entering your app.

The following recipe demonstrates how to use the `Securely` framework to check if the user has a lock screen pattern enabled and fire an event when the screen is locked or unlocked.

Getting ready

This recipe uses the `Securely` native module. This module and other code assets can be downloaded from the source provided by the book. Installing these in your project is straightforward. Simply copy the `modules` folder into your project as shown in the following screenshot:



After copying the `modules` folder, you will need to update the `tiapp.xml` references as demonstrated in the Getting ready section of the *Using secure properties* recipe.

How to do it...

Once you have added the `module` folder into your project, you need to create your application namespace in the `app.js` file and use `require` to import the module into your code as the following code snippet demonstrates:

```
//Create our application namespace
var my = {
  secure : require('bencoding.securely')
};
```

Creating the recipe's UI

This recipe uses a single `Ti.UI.Window` object to host and demonstrate the different available lock screen methods and events. The following code snippet shows how this object is created.

```
var win = Ti.UI.createWindow({
  backgroundColor: '#fff', title: 'Lock Screen Monitor',
  fullscreen:false, exitOnClose:true
});
```

Verifying if the lock pattern is enabled

This recipe is dependent on the user enabling a passcode or lock pattern. If this feature is not enabled, the recipe will still function by simply providing when the screen has been disabled from a power consumption standpoint.

The following steps discuss how to verify if the user has enabled the lock screen functionality:

1. The first step is to create a new `Securely.Platform` proxy as shown in the following code snippet:


```
var platform = my.secure.createPlatform();
```
2. The `Securely.Platform` proxy provides many security-related methods. When the `lockPatternEnabled` method is called, a `Boolean` is provided indicating if the user has enabled this feature on their device.

```
if(!platform.lockPatternEnabled()){
  alert('lock screen is not enabled on this device');
}
```



Depending on your organization's passcode policy, you may wish to disable the app if a lock screen has not been implemented.

Creating a Lock Helper

The `Securely.LockScreenHelper` proxy object provides the initialization methods needed to start monitoring lock screen activity. The following code snippet demonstrates how to use this proxy to start the monitoring process:

1. The first step in the lock screen monitoring process is to create a new `Securely.LockScreenHelper` proxy as shown in the following code snippet:

```
var lockHelper = my.secure.createLockScreenHelper();
```
2. Next the `startMonitorForScreenOff` method is called. This registers a broadcast receiver to listen for the `ACTION_SCREEN_OFF` broadcast.

```
lockHelper.startMonitorForScreenOff();
```
3. Then the `startMonitorForScreenOn` method is called. This registers a broadcast receiver to listen for the `ACTION_SCREEN_ON` broadcast.

```
lockHelper.startMonitorForScreenOn();
```

Screen lock events

Both the `startMonitorScreenOff` and `startMonitorScreenOn` methods described earlier will emit global events when their subscribed broadcast is received. The following snippet demonstrates how to create application listeners to subscribe to these events:

```
Ti.App.addEventListener('BCX:SCREEN_OFF', function(e) {
  Ti.API.info('Last locked at ' +
    String.DateFormat(new Date(e.actionTime)));
});

Ti.App.addEventListener('BCX:SCREEN_ON', function(e) {
  Ti.API.info('Last unlocked at ' +
    String.DateFormat(new Date(e.actionTime)));
});
```

Each event is provided information to assist in managing your app state. Using the prior example snippet, the `e` argument is provided two properties by `Securely`.

- ▶ `actionName`: This is the full Android intent action name.
- ▶ `actionTime`: This provides date/time in seconds format on when the last event was called. This can be converted into a JavaScript date using `new Date(e.actionTime)`.

Using window focus for monitoring

This recipe uses the `focus` event on the example `Ti.UI.Window` to demonstrate how to check if the device has been locked since the last time the `Ti.UI.Window` had focus. One use of this pattern would be to check if an internal passcode screen should be presented or to check if a session needs to be re-established.

```
win.addEventListener('focus', function(e) {
```

1. The `wasLocked` method is called to determine if the device has been locked.

```
    if(lockHelper.wasLocked()) {
```

2. The `isShowingLockScreen` method can also be used to determine if the device is currently presenting the lock screen to the user.

```
        if(!lockHelper.isShowingLockScreen()) {
```

3. The `resetMonitorForScreenOff` method can also be used to reset the value returned by `wasLocked`. This is helpful in tracking if the device has been locked between app sessions.

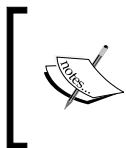
```
            lockHelper.resetMonitorForScreenOff();
```

```
        }
    }
});
```

Stop monitoring

It is important to stop monitoring and remove the global listeners when the app no longer needs this functionality. The following code snippet demonstrates how this is performed using the `close` event of the `Ti.UI.Window`.

```
win.addEventListener('close', function(e) {
    lockHelper.stopMonitoring();
    Ti.App.removeEventListener('BCX:SCREEN_ON', screenON);
    Ti.App.removeEventListener('BCX:SCREEN_OFF', screenOFF);
});
```



Monitoring can be stopped individually by using `stopMonitorForScreenOff` or `stopMonitorForScreenOn`. To stop all monitoring, the `stopMonitoring` convenience method can be used to remove both receivers.

See also

- ▶ To learn more about the `android.intent.action.SCREEN_ON` and `android.intent.action.SCREEN_OFF` intents used, please visit the official Android documentation available at <http://developer.android.com/reference/android/content/Intent.html>.
- ▶ This recipe uses the `Securely` module. For installation details, please review the *Getting ready* section of the *Using secure properties* recipe.

Titanium Resources

In this appendix we will cover:

- ▶ Getting started with Titanium
- ▶ Getting started with Android
- ▶ Getting started with iOS
- ▶ Titanium testing resources
- ▶ Modules and open source
- ▶ Titanium community links

Getting started with Titanium

To create your own Titanium apps or to run any of the example projects provided in this book, first you will need to download the Titanium SDK. This free of charge development platform is available at <http://www.appcelerator.com>.

Appcelerator provides a free IDE to aid in the development of Titanium apps. To learn more about Titanium Studio and to download the application, please visit <http://developer.appcelerator.com>.



All example projects provided with this book are designed to be imported into a Titanium Studio project.

Getting started with Android

In order to build Android apps using Titanium, you first need to install the Android SDK. To download the Android SDK and review the installation documentation, visit <http://developer.android.com/sdk/index.html>.

After installing the Android SDK, you must set up the Android references in Titanium Studio and configure default emulators. For a step-by-step guide on how to set up Android in Titanium Studio visit <http://developer.android.com/sdk/index.html>.

For information regarding minimum supported SDK versions and overall compatibility, please visit http://docs.appcelerator.com/titanium/latest/#!/guide/Titanium_Compatibility_Matrix.

Getting started with iOS

Titanium utilizes the native platform development tool chain. For this reason, an Apple computer is required to build Titanium iOS apps.

There are currently two ways to download Xcode and the required SDK for development.

- ▶ Open the Mac App Store on your Apple computer and search for Xcode. This will allow you to freely download the Apple IDE and SDKs.
- ▶ The Apple development tools are also available through the developer portal at <https://developer.apple.com>.

To publish apps to the Apple App Store, you must be part of the Apple iOS Developer program. To learn more about app distribution, please visit <https://developer.apple.com/programs/ios/distribute.html>.

Titanium testing resources

Titanium's testing story is quickly evolving. The following tools are open source companions to Appcelerator's testing platform.

TiShadow

The TiShadow project is a staple for Titanium and used by developers to solve a wide range of problems. This mature and widely adopted project shines as a testing tool providing built-in support for BDD style testing using Jasmine. More than just a testing tool, TiShadow provides assistance with a wide range of activities from cross-platform preview to advanced developers tools.

Learn more about TiShadow at <https://github.com/dbankier/TiShadow>.

Titanium-Jasmine

The Titanium-Jasmine project provides support for the Jasmine testing framework within your Titanium app. This well-documented project provides all of the documentation needed to get started, including sample apps and instructions.

Learn more about Titanium-Jasmine at <https://github.com/guilhermechapiewski/titanium-jasmine>.

TiJasmine

The TiJasmine project is a new open source project by one of the core Titanium SDK committers that provides Jasmine testing support to Titanium. Designed with Titanium's CommonJS implementation in mind, this project allows you to quickly incorporate tests into your existing Titanium app.

Learn more about TiJasmine at <https://github.com/billdawson/tijasmine>.

Modules and open source

Appcelerator has built a strong open source community around the Titanium mobile platform. A majority of the Titanium mobile ecosystem is open source, including the Titanium SDK itself. This open source community provides a wonderful resource to learn and enhance your Titanium development processes. The following sections discuss a few key Titanium open source projects.

Appcelerator on Github

Appcelerator's Github repository contains all open source solutions including Titanium mobile provided by Appcelerator. These repositories provide an up-to-the-minute status on a majority of Appcelerator's products, including new features and updates.

View all of Appcelerator's open source solutions at <https://github.com/appcelerator>.

Titanium mobile on Github

The Titanium mobile SDK is open source under the Apache 2 license and available on Github. Titanium is built using the same module API as third-party modules leverage. Because of this, viewing the Titanium SDK code provides insight into the practices for developing such modules. Having the code available provides a powerful learning opportunity for Titanium developers as they are able to see how their JavaScript interacts with the underlying platform.

View the source for the Titanium mobile SDK at https://github.com/appcelerator/titanium_mobile.

Titanium modules on Github

A key feature of Titanium mobile is the ability to support third-party modules. All of the modules used in this book are open source and available on Github. In addition to those modules offered by third parties, Appcelerator provides a large set of community modules available at https://github.com/appcelerator/titanium_modules.

Titanium community links

Titanium has a large thriving community of developers. The following sections discuss a few popular sites maintained by Appcelerator and the Titanium community.

Documentation

Appcelerator provides complete documentation for the Titanium mobile SDK and related components at <http://docs.appcelerator.com/titanium/latest>.

All 5,000+ SDK APIs are documented with samples at <http://docs.appcelerator.com/titanium/latest/#!/api>.

Appcelerator has created *Getting Started* guides for a majority of the platform features at <http://docs.appcelerator.com/titanium/latest/#!/guide>.

Support

The Titanium Q&A forum at <http://developer.appcelerator.com/questions/newest> is the primary portal for support product. This forum provides an extensive list of support topics and is frequented by Titanium Titans and Appcelerator employees.

The popular programming Q&A site stackoverflow also has a Titanium mobile section available at <http://stackoverflow.com/questions/tagged/titanium>. This can be an excellent resource if your question involves native platform components.

Appcelerator also provides a paid support option. Learn more at <http://support.appcelerator.com>.

Other helpful resources

The Appcelerator developer blog at <http://developer.appcelerator.com/blog> provides an excellent source of news, guidance, and code examples.

The Titanium meet-up community is also growing, a list of meet-ups near you is available at <http://appcelerator.meetup.com>.

The official Appcelerator twitter account, <https://twitter.com/appcelerator>, is an excellent source of news including new Titanium apps launching in the different app stores.

If you are looking for help with a Titanium project, the Devlink program available at <http://developer.appcelerator.com/devlink>, provides a listing of experienced Titanium developers and agencies.

Index

A

addAPIKey

using 105

addCloseTimer function, HUD module 38

addCloseTimer function, NotifyUI module 43

addDays method 13

addEventListener function, HUD module 38

addEventListener function, NotifyUI module 43

addProvider method 178

addPurpose method 171

addToMap method 164, 173

Advanced Encryption Standard (AES) 246

AESDecrypt method 280

AES encryption

implementing, JavaScript used 247-251

AES encryption recipe

about 246

UI, creating 248, 249

values, decrypting 250, 251

values, encrypting 250, 251

AESEncrypt method 280

Android 298

Android app

auto restart scenario 227, 228

notification on restart scenario 228, 229

opening, with BOOT_COMPLETED 224-226

property controlled scenario 229-232

required tiapp.xml updates 226

Android app recipe 224-226

Android Intent Supported 134

Android lock screen monitor recipe

about 292

Lock Helper, creating 294

lock pattern, verifying 293

monitoring, stopping 295

screen lock events 294

UI, creating 293

window focus, used for monitoring 295

Android SDK 298

Android.Tools module 233

Android.Tools.Receiver module 227

Apache Public License (Version 2) 7

Appcelerator

on Github 299

URL 297

Appcelerator developer blog

URL 301

Appcelerator Titanium Mobile 7

Appcelerator twitter account

URL 301

App launcher recipe

about 206

requisites 207

scheme launch list, creating 208

tiapp.xml, updating with iOS 208

UI, creating 212, 213

Apple 240

Apple App Store 298

Apple development tools

URL 298

application-level event

firing 31

app performance

improving, with SQLite transactions 78

architecture, Titanium app

JavaScript code 8

JavaScript interpreter 8

Native Custom Modules 8

Titanium API 8

assistant functions, download queue sample recipe

callback method 205
dequeue method 203
download method 203
getLength method 202
next method 202
peek method 202
progressSetup method 201
requeue method 204
updateProcess method 202
updateProgress method 203
whenFinished method 202
whenFinish method 202

B

background capabilities, PlatformHelpers module

4-inch iPhone 26
about 24
iPad 26
iPad Mini 27
tablets, detecting 25

Background Geo recipe

about 188
assistant methods 192
background button events 194
background location options 191
geolocation events 193
iOS app-level events 195
namespaces, creating 190
tiapp.xml file, updating 189, 190
UI, building 191

basic authentication recipe

about 252
service connection, creating 255, 256
Ti.Network.HTTPClient module used 252-257
UI, creating 253, 254

basicGeo module 162

URL 175

beginTransaction method 86

bencoding.android.tools module 225

bencoding.basicgeo module 170

benCoding Geo recipe

about 160-162
availability helpers, adding 162
current location, finding 164, 165
device capability check, performing 167
forward location lookup 166
location services purpose, adding 162
place objects, working with 163, 164
UI, building 162, 163

bencoding.securely module 241

bencoding.zip module 262

BOOT_COMPLETED action

about 224
Android app, opening with 224-226
event lifecycle 224

BootReceiver

URL, for documentation 233

BreakingMenu recipe

about 44
closed state, Screen Break Menu 44
menu, displaying 48
notes menu object, creating 47
open state, Screen Break Menu 45
sample window, creating 46
Screen Break Menu, adding 46
Screen Break Menu listeners, adding 47

breakScreen function

about 48
parameters 48

built-in message styles, NotifyUI module

about 43
completed 43
error 43
success 44
warning 43

business location map

creating, Yahoo Local used 101, 102

C

chartLauncher module

working 153

CommonJS modules

adding 10, 11
app.js, creating 12
datehelpers module, building 12
dateWin module 13, 14

- functions 16
- global scope anti-pattern 18, 19
- instance object, creating with module.exports 17
- properties 16
- require method 15
- using 12
- working 15
- convertToObjects 123**
- CoverSliderExample**
 - URL 61
- createdOn method 13**
- createDummyObject function 81**
- createOrResetDb method 80**
- createSQLStatement method 81**
- createTabStrip method 52**
- createWindow function 13**
- CRM tool 7**
- cross-platform API**
 - building 9
- cross-platform HUD progress indicator**
 - about 34
 - creating 36
 - HUD listeners, adding 36
 - HUD message, updating 37
 - HUD module, adding 34, 35
 - HUD window, closing 37
 - sample window, creating 35
 - using 34
 - working 37
- cross-platform passcode screen recipe**
 - about 257
 - implementing 257-261
 - passcode screen, launching 259-261
 - UI, creating 258
- cross-platform URL schemes recipe**
 - about 214
 - About window, launching 221, 222
 - helper functions, launching 218, 219
 - Login window, launching 222-224
 - proper event listeners, adding in app 220, 221
 - requisites 214
 - tiapp.xml, updating with Android 216, 217

- tiapp.xml, updating with iOS 215
- UI, creating 217, 218
- CSS 7**
- currentLocationQuery method**
 - using 105

D

- Data Encryption Standard (DES) 279**
- dateHelper module 13**
- datehelpers CommonJS module 13**
- dbDirectory function 65**
- dbExists function 65**
- DbFileExt module**
 - about 64
 - adding, to project 64
 - database directory, determining 67
 - database file, removing 68
 - database file, renaming 68
 - databases, listing 68
 - database Ti.Filesystem, adding 67
 - dbList function 68
 - file exist check 67
 - test, setting up 66
 - working 66
- dbFile function 65**
- DbLazyProvider module**
 - about 83
 - connection object, getting 86
 - database connection, closing 87
 - database connection, opening 86
 - DbLazyProvider object, creating 86
 - transaction, beginning 86
 - transaction, ending 86
- dbRemoteBackup function 65**
- DbTableChecker SQLite table existence checking**
 - about 69
 - helpers, testing 72
 - module, checking 69
 - namespace, creating 70
 - table, creating 71
 - table, removing 71
 - table, testing 70
 - window, creating 70

dbTableCheck module
about 70
tableExists method 73

dbTestHelper method 82

Dbtuning recipe
about 77
benchmarking 80
database setup 80, 81
testing Interface, creating 79
tests, performing 81, 82
TimeLogger module, adding 78, 79

demoUI library 214

DESDecrypt method 279

DES encryption
using 279

DESEncrypt method 279

Devlink program
URL 301

dispose method 134

distanceInUnits method 187

document-signing app
buttons, adding 138, 139
creating 134-137
document view, adding 137
running, on iPad 134-137
saved signature, reloading 139
signature view, adding 137
window, creating 137

Dossier
about 74
adding, to project 74
copying directory contents, recursively 76
directory contents, listing recursively 75
moving directory contents, recursively 77
sample directories, creating 75
URL 77

download queue sample recipe
about 198
assistant functions 201-205
download, starting 205
jobs, adding to queue 200
network connection 199
queue, creating 200
UI, creating 199, 200

dropTable function 72

F

fetchWindow function 31

file encryption recipe
about 281
file, decrypting 284, 285
file, encrypting 283, 284
UI, creating 282, 283

findDistance method 187

findLocations function 185, 187

formatUrl function 123

forwardGeoCallback function 167

forwardGeocoder method 166

forwardGeo method 186

G

gauges
about 154
saved sales information, loading 156
used, for displaying information 154, 155

generateAddress method 174, 180

generateDerivedKey method 278

Geo Distance recipe
about 181-183
address information, adding 184
distance and address lookup operations,
performing 185
distance, finding between addresses 186,
187
UI, building 184

geo.findLocation method 187

geolocation 159

GeoNames GeoProvider
URL 175

geoQuery method
using 105

getChildNodes function 114

getCurrentAddress method 172, 180

getCurrentPlace method 164

getProfileUrl function 123

getProvider method 180

getQueryParams function 122

getText function 122

Github
Appcelerator 299
Titanium mobile 299

- Titanium modules 300
- Github project**
 - URL 91
- Github repository 8**
- global logging, with Ti.App.Listener**
 - app.js, defining 29
 - designing, with events 29
 - logging module 30
 - performing 27-29
- global scope anti-pattern, CommonJS modules 18, 19**
- Google Analytics**
 - about 106
 - using, in app 106
- Google Analytics dashboard**
 - actions, publishing 109
- Google Analytics Sample**
 - action, recording 109
 - events, recording 108
 - Helper functions 108
 - instance, creating 107
 - Pageview function, on child window 110
 - Pageview function, on opening window 110
 - sample UI, creating 109
 - working 107
- Google GeoProvider**
 - URL 175
- Google Groups**
 - URL 91

H

- hasProperty method 275**
- Heads Up Display.** *See* **HUD**
- helper functions, cross-platform URL schemes recipe**
 - getCurrentPage function 219
 - getLaunchUrl method 219
 - hasChanged method 219
 - hasLaunchUrl method 219
 - openPageFromUrl method 219
 - openWindow method 218, 219
- helpers function 13**
- hide function, HUD module 38**
- hide function, NotifyUI module 42**
- HTML 7**
- HUD 34**

- HUD message**
 - updating 37
- HUD module**
 - about 34
 - addCloseTimer function 38
 - addEventListener function 38
 - adding, to project 34
 - hide function 38
 - load function 37
 - removeCloseTimer function 38
 - removeEventListener function 38
 - show function 37
 - updateMessage function 38
 - using 34
 - working 37
- HUD module window**
 - closing 37

I

- iExplorer**
 - URL 242
- in-app notifications**
 - about 38
 - built-in message styles 43
 - message, updating 42
 - message window, closing 42
 - message window, displaying 41
 - NotifyUI listeners, adding 40, 41
 - NotifyUI module, adding 39
 - sample window, creating 40
 - working 42
- indexChanged event 52**
- information**
 - displaying, gauges used 154
- invoke method 116**
- iOS**
 - about 298
 - protected PDFs, handling on 286-291
- iOS data protection**
 - implementing, in Titanium 240-242
- iOS data protection recipe**
 - about 240
 - data protection, enabling 241
 - file, creating for data protection verification 243, 244

- imports, creating 242
- namespace, creating 242
- requisites 242
- UI, creating 243
- working 244-246

iOS DocumentViewer 134

iOS Multithreading recipe

- about 233, 236
- requisites 234
- testing functions 235
- UI, creating 235

iPad

- document-signing app, running 134-137

isPublic function 123

isSimulator property 22

J

JavaScript 7

- used, for implementing AES encryption 247-251

jsPDF

- used, for generating PDF documents 144-147

L

lazy-loading pattern

- about 83
- for, select statement 85
- implementing 83, 84
- testing Interface, creating 84
- transactions, using 84, 85

LinkedIn API results

- parsing 122-124

LinkedIn Contacts API 117, 118

LinkedInSearch

- about 118, 119
- API key, adding 119
- contacts, loading 121
- LinkedIn API results, parsing 122-124
- permissions, adding 120
- results, formatting with uiHelpers object 121
- secret key, adding 119
- UI, creating 120

listContents method 75

load function, HUD module 37

load function, NotifyUI module 42

locationChangeCallback method 193

locationErrorCallback method 193

location services

- Background Geo Recipe 188
- benCoding Geo recipe 160
- Geo Distance recipe 181
- Multi-Tenant Geo 176
- Ti.GeoProviders recipe 168

lookup.updateProvider method 172

M

Mac App Store 298

mainWin.js module 31

makeTable function 72

MASlidingMenu

- URL 61

maxIterations property 80

Metro Style Tab Control

- about 49
- adding, to project 50
- adding, to window 52
- sample window, creating 50
- settings object, creating 51
- tab listeners, adding 52
- tabs, defining 51, 52
- used, for creating cross-platform experience 49

MongloDb module

- about 87
- bundle, creating 87
- collection, initializing 89
- database, initializing 88
- documents, inserting 90
- documents, removing 91
- documents, updating 90
- events, using 89
- find function, using 90
- findOne method, using 91
- installing 87
- Titanium storage provider, adding 89

MongoDb 10

move method 77

Multi-Tenant Geo recipe

- about 176, 178
- lookup helper methods 179, 180

- multitenant reverse geolocation lookup, performing 180
- providers, adding 178
- purpose, adding 178
- UI, building 179

multitenant reverse geolocation lookup
performing 180

N

National Institute of Standards and Technology (NIST) 281

Node.js 10

NoSQL database 87

NotifyUI module

- about 38
- addCloseTimer function 43
- addEventListener function 43
- adding, to project 39
- built-in message styles 43
- hide function 42
- load function 42
- removeCloseTimer function 43
- removeEventListener function 43
- show function 42
- updateMessage function 42
- using 38

O

object and string encryption recipe

- about 277
- decryption, with AES 280
- decryption, with DES 279
- derived key, generating 278
- DES encryption, using 279
- encryption, with AES 280
- encryption, with DES 279
- keys, generating 278
- namespace, creating 277
- random key, generating 278
- stringCrypto object, creating 278

onComplete method 98

onError callback method 172

onError method 175

onFinish function 185

onSuccess callback method 172

onSuccess method 174

openDialog method 133

openPDF CommonJS module 128

openPDF module 134

P

parameters, breakScreen function

- bottomViewHeight 48

- breakStartPoint 48

- slideFinishPoint 48

password-protected ZIP file

- creating 264

PDF documents

- closing 134

- creating, from images/screenshots 140, 142

- creating, from screenshot 143, 144

- creating, from view 142

- creating, jsPDF used 146, 147

- example UI, creating 130

- generating, jsPDF used 144, 145

- link, creating 130

- opening 128, 130

- opening, within iOS app 132, 133

- opening, with one click 131

PDF file

- protecting 289

- unlocking 290

performanceTest object 81

placeHelpers.address method 165

PlatformHelpers CommonJS module

- adding, to project 20-23

- background capabilities 24

- creating 20

- simulator check 24

- working 24

platform indicators 19

polluting global namespace condition 10

protected PDFs

- handling, on iOS 286-291

protected zip files recipe

- about 262

- password-protected ZIP file, creating 264-266

- protected ZIP file, unzipping 266-268

- UI, creating 263, 264

Q

queryByPath method 100

query function

using 99, 100

R

readCache method 194

removeAllProperties method 276

removeCloseTimer function, HUD module 38

removeCloseTimer function, NotifyUI module 43

removeEventListener function, HUD module 38

removeEventListener function, NotifyUI module 43

removeTable function 72

resetTestTable method 81

resetUI method 116

resultsCallback method 164

RGraph

UI, creating 151

used, for creating scheduling chart 148, 150

rss2Objects CommonJS module

about 94

RSS feeds

articles, displaying 97

consuming 94

query function, using 99, 100

reading, with XML parsing 96

reading, YQL used 96

yqlQuery function, using 98

RSS reading sample

about 94

UI, creating 95

runTestButton button 80

S

sale gauge

sales, adjusting 156, 157

saved sales, reloading 157

UI, creating 156

scheduling chart

creating, RGraph used 148, 150

example, launching 152, 153

schedules, creating 151

tasks, assigning 151

scheme launch list, App launcher recipe

Android scheme list 210, 211

creating 208

iOS scheme list 209

schemeList.js file 207

schemeList module 207

Screen Break Menu module

about 44, 45

adding, to project 46

ScreenToPDF recipe

creating 140

UI, creating 142

SDK APIs

URL 300

Securely module 286

Securely.Properties object 270

Securely.StringCrypto object Securely.

StringCrypto object 277

secure properties

considerations 276

field names, listing 274

on Android 276

on iOS 276

reading, with defaults 273

reading, without defaults 273

removing 275

using 270

values, setting 274

secure properties object

creating 272

secure properties recipe

about 270

module reference, adding 270

namespace, creating 271

result comparison helper 272

secure properties object, creating 272

secure properties, on Android 276

secure properties, on iOS 276

secure properties, reading with defaults 273

secure properties, reading without defaults 273

secure properties, removing 275

secure property field names, listing 274

secure property values, setting 274

selectedFont object 51

Service-Oriented Architecture. *See* SOA

setLocation method 164, 174

show function, HUD module 37

show function, NotifyUI module 42

Slideout menu

- about 53
- adding, to project 56
- app window, opening 58
- closing 60
- content views, defining 57
- current view, accessing 60
- custom application listeners, adding 59
- displaying 59
- global events, using 60
- menu, adding 59
- menu container, closing 60
- menu container, opening 59
- menu items, building 57
- menu listeners, adding 58
- menu state, determining 60
- Ti.Draggable module, installing 53-55
- toggling 60
- used, for creating sample app 53

SOA 93

SOAP service calls

- making, SUDS.js used 111, 112

SOAP Service Sample

- about 111
- SOAP helper methods 112
- SOAP service, calling 116, 117
- UI, creating 114, 115
- uiHelpers object 115, 116

SQLite transactions

- about 77
- managing 77

stringCrypto object

- creating 278

subtractButton 14

SUDS2.js

- about 117
- URL 117

SUDS.js

- about 117
- URL 117

SUDS module 112

T

tabClick event 52

Tab Definition 51

tableExists function 70, 73

tableName parameter 72

tabSettings object 51

Ti.Android.intent

- URL 134

Ti.App.Properties object 270

Ti.App.removeEventListener method 32

tiapp.xml file, Background Geo recipe

- updating 189

Ti.BA.js file 252

Ti.Draggable

- URL 61

Ti.Filesystem

- database, accessing 64
- database directory, determining 65
- database file, removing 66
- database file, renaming 66
- database, finding 65
- databases, listing 66
- file exist check 65

Ti.Geo.Background CommonJS module 190

Ti.Geo.Background module 193

Ti.GeoProvider Framework

- about 168
- URL 175

Ti.GeoProviders recipe

- about 168, 169, 170
- API key, adding 171
- lookup functions 173, 175
- purpose, adding 171
- reverse geolocation, running 172
- UI, building 171

Ti.Google.Analytics CommonJS module 106

Ti.Jasmine

- about 299
- URL 299

TimeLogger module 78, 79

time reporting 206

Ti.Network.HTTPClient module

- about 252
- used, for basic authentication 252-257

Ti.Passcode.js file 257

- Ti.Passcode module** 257
- Ti.Queue.js file** 198
- Ti.Queue module**
 - about 198
 - URL 206
- Ti.SchemeTools module** 214
- TiShadow**
 - URL 298
- Ti.SlowAES module**
 - about 247
 - URL 251
- Titanium**
 - about 297
 - iOS data protection, implementing 240- 242
 - PDF documents, opening 128
- Titanium Android apps**
 - creating 298
- Titanium app**
 - app namespace, defining 10
 - architecture 8, 9
 - building 7
 - CommonJS 10
 - cross-platform API, building 9
 - global logging, Ti.App.Listener used 27, 28
 - platform indicators, using 19
- Titanium apps**
 - creating 297
- Titanium community links**
 - helpful resources 301
 - Q&A forum 300
 - Titanium mobile SDK 300
- Titanium ecosystem** 8
- Titanium iOS apps**
 - creating 298
- Titanium-Jasmine**
 - about 299
 - URL 299
- Titanium meet-up community**
 - URL 301
- Titanium mobile**
 - on Github 299
- Titanium Mobile project** 8

- Titanium mobile SDK**
 - URL 300
- Titanium modules**
 - on Github 300
- Titanium Q&A forum**
 - URL 300
- Titanium SDK** 297
- Titanium Store**
 - URL 91
- Titanium testing resources**
 - about 298
 - TiJasmine 299
 - TiShadow 298
 - Titanium-Jasmine 299
- Ti.UI.Window** 12
- Ti.WebWorkerWrapper module**
 - about 233, 237
 - URL 237
- toObject method** 114
- trackEvent function** 109

U

- uiHelpers object** 115
- updateMap function** 103
- updateMessage function** 37
- updateMessage function, HUD module** 38
- updateMessage function, NotifyUI module** 42
- updateProvider method** 173
- updateUI method** 117

W

- whenComplete function** 99
- withTransactions variable** 80

X

- Xcode** 298
- xmlToObject function** 113
- XMLTools**
 - about 112,117
 - URL 117

Y

Yahoo Business Search

- addAPIKey, using 105
- API key, adding 102
- creating 101
- currentLocationQuery method, using 105
- geoQuery method, using 105
- location search, performing 104
- map, updating 103
- UI, creating 103

Yahoo Local Search API

- using 101, 102

Yahoo Search CommonJS module 102

YQL

- about 96
- benefits 96
- URL 101

yqlQuery method

- about 98
- using 98

YQL usage guidelines

- URL 101



Thank you for buying
**Appcelerator Titanium Business
Application Development
Cookbook**

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.

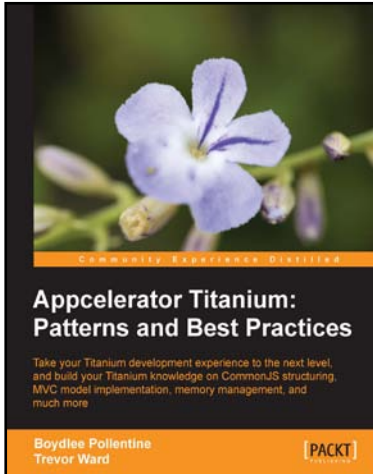
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books 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.

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.

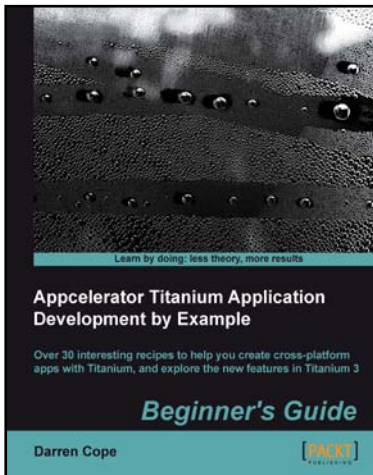


Appcelerator Titanium: Patterns and Best Practices

ISBN: 978-1-84969-348-6 Paperback: 320 pages

Take your Titanium development experience to the next level, and build your Titanium knowledge on CommonJS structuring, MVC model implementation, memory management and much more

1. Full step-by-step approach to help structure your apps in an MVC style that will make them more maintainable, easier to code and more stable
2. Learn best practices and optimizations both related directly to JavaScript and Titanium itself
3. Learn solutions to create cross-compatible layouts that work across both Android and the iPhone



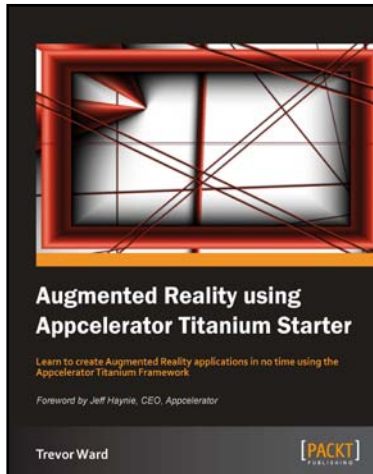
Appcelerator Titanium Application Development by Example Beginner's Guide

ISBN: 978-1-84969-500-8 Paperback: 334 pages

Over 30 interesting recipes to help you create cross-platform apps with Titanium, and explore the new features in Titanium 3

1. Covers iOS, Android, and Windows8
2. Includes Alloy, the latest in Titanium design
3. Includes examples of Cloud Services, augmented reality, and tablet design

Please check www.PacktPub.com for information on our titles

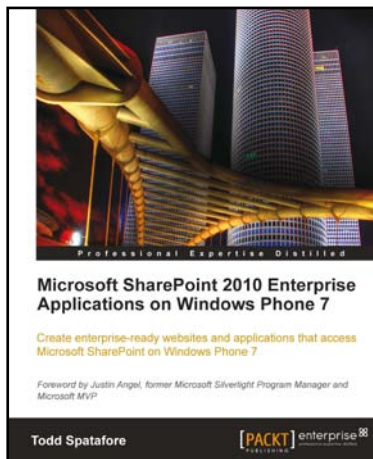


Augmented Reality using Appcelerator Titanium Starter

ISBN: 978-1-84969-390-5 Paperback: 52 pages

Learn to create Augmented Reality applications in no time using the Appcelerator Titanium Framework

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create an open source Augmented Reality Titanium application
3. Build an effective display of multiple points of interest
4. Learn to calculate distances between points of interest



Microsoft SharePoint 2010 Enterprise Applications on Windows Phone 7

ISBN: 978-1-84968-258-9 Paperback: 252 pages

Create enterprise-ready websites and applications that access Microsoft SharePoint on Windows Phone 7

1. Provides step-by-step instructions for integrating Windows Phone 7-capable web pages into SharePoint websites
2. Provides an overview of creating Windows Phone 7 applications that integrate with SharePoint services
3. Examines Windows Phone 7's enterprise capabilities

Please check www.PacktPub.com for information on our titles

