



Community Experience Distilled

Appcelerator Titanium: Patterns and Best Practices

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

Boydlee Pollentine
Trevor Ward

[PACKT]
PUBLISHING



Community Experience Distilled

Appcelerator Titanium: Patterns and Best Practices

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

Boydlee Pollentine
Trevor Ward

[PACKT]
PUBLISHING

Table of Contents

[Appcelerator Titanium: Patterns and Best Practices](#)

[Credits](#)

[About the Authors](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers and more](#)

[Why Subscribe?](#)

[Free Access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Understanding JavaScript Patterns](#)

[It's all object\(ive\)](#)

[Understanding scope and the global object](#)

[The global object](#)

[Code structure and formatting](#)

[Use semicolons and curly brackets](#)

[eval is evil](#)

[The curly bracket position](#)

[Not all operators are created equal](#)

[Formatting, commenting, and naming conventions](#)

[Prototype](#)

[Micro optimizations](#)

[Declaring variables outside the for statements](#)

[Using forEach instead of for loop](#)

[Using shortcuts for simple if statements](#)

[Summary](#)

[2. Titanium Best Practices](#)

[CommonJS](#)

[Code formatting](#)

[Code validation](#)

[Comment meaningfully](#)

[Do not pollute the global object](#)

[JavaScript instance](#)

[CommonJS modules](#)

[CommonJS best practices](#)

[Managing memory](#)

[Summary](#)

[3. Building an Application Using CommonJS and the MVC Pattern](#)

[Standards](#)

[Frameworks](#)

[The architecture](#)

[CommonJS](#)

[The bootstrap](#)

[Multilingual](#)

[Directory structure](#)

[Android](#)

[iPhone and iPad](#)

[Control](#)

[Model](#)

[Services](#)

[Data](#)

[UI](#)

[Tools](#)

[Naming](#)

[The application](#)

[The controller](#)

[Tools \(helpers\)](#)

[CommonJS require](#)

[Callbacks](#)

[Model](#)

[UI](#)

[The layout module](#)

[The custom menu](#)

[Common elements](#)

[Windows](#)

[The controller updated](#)

[Fail gracefully](#)

[Summary](#)

[4. Cross-platform Design Methods](#)

[The difference between DPs and PXs](#)

[Understanding Ti.UI.SIZE and Ti.UI.FILL](#)

[Ti.UI.SIZE](#)

[Ti.UI.FILL](#)

[Choosing a cross-platform layout method](#)

[Using Ti.Platform.osname statements](#)

[Developing with separate UI files](#)

[Importing component styles from external files](#)

[Building a JavaScript stylesheet CommonJS module](#)

[Using the Stylesheet CommonJS module in your app](#)

[Extending the Stylesheet module for page-specific styles](#)

[Summary](#)

[5. Using JSONDB as an Alternative to SQLite](#)

[Installing the JSONDB module](#)

[Setting up JSONDB](#)

[Creating a JSONDB CommonJS module](#)

[Using JSONDB queries](#)

[Summary](#)

[Index](#)

Appcelerator Titanium: Patterns and Best Practices

Appcelerator Titanium: Patterns and Best Practices

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 authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

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

First published: February 2013

Production Reference: 1150213

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84969-348-6

www.packtpub.com

Cover Image by Asher Wishkerman (<wishkerman@hotmail.com>)

Credits

Authors

Boydlee Pollentine

Trevor Ward

Reviewers

Jeff Bonnes

Imraan Jhetam

Peter Puglisi

Acquisition Editor

Kartikey Pandey

Lead Technical Editors

Neeshma Ramakrishnan

Kedar Bhat

Technical Editor

Charmaine Pereira

Project Coordinator

Leena Purkait

Proofreader

Samantha Lyon

Indexer

Rekha Nair

Production Coordinator

Manu Joseph

Cover Work

Aparna Bhagat

About the Authors

Boydlee Pollentine is a keen mobile developer who has created numerous apps for the iTunes and Android store and a number of indie games. He is passionate about mobile development and in particular the Appcelerator Titanium platform, and is both a Titanium-certified application developer and a member of the Titans evangelist group. He is also the organizer of Europe's first Titanium mobile development conference, tiConf EU (<http://ticonf.eu>).

Boydlee has been a software engineer and programmer for the last 10 years, primarily focused on web technologies and Microsoft's .NET platform, and during that time has worked for numerous small and large organizations, those of particular note including a number of Australian federal government departments, state departments, banks, and media organizations.

He currently lives in Norfolk and works as a freelance Titanium developer via his own digital agency, Topsy & Tumbler Limited (<http://www.tipsyandtumbler.co.uk>), and runs a small blog dedicated to mobile development at <http://boydlee.com>.

I'd like to thank everyone on the Appcelerator forums, on Twitter, and the Web, who have helped contribute and make the great Titanium community as it is today.

Trevor Ward has been developing business applications for over 20 years. Starting as a COBOL developer, he has worked on various large-scale projects that included the Y2K issues. Moving into web development in the late 90s using Perl, HTML, JavaScript, and Oracle, he was a part of the team that developed internal business applications for Jaguar cars.

After moving on, he was able to update his skills for Ruby on Rails and Adobe Flex, before picking up on the mobile development

platform 18 months ago. He uses Titanium exclusively for mobile development and is an Appcelerator Titan (community expert), TCAD and TCE qualified, and spends as much time as possible answering questions on the forums.

About the Reviewers

Jeff Bonnes is an entrepreneur and developer with 20 years experience in the IT industry. He has been developing data-driven websites since 1995 using technologies such as C, PHP, Java, and Grails. He has worked in a wide range of industries including local government, finance, document management, and online communities. He discovered Appcelerator Titanium when it was Version 0.8, and has not looked back since. His passion is working with startups to build innovative online solutions around great ideas. He has a blog at <http://www.titaniumdevelopment.com.au>, where he keeps the Titanium community up-to-date on the latest news.

Imraan Jhetam is a medical doctor and entrepreneur living in England with an equal love 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 1970s in various languages and for different platforms, and has fond memories of his first Apple][e 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 the 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, which were developed in a restricted and short time using Appcelerator Titanium Studio.

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

Peter Puglisi is an independent developer based in Brisbane, Australia, with over 20 years experience in software development and design. He is the founder of Bright Light Apps Pty. Ltd., a mobile apps publishing company that builds multi-platform apps that aid, educate, and entertain. He can be reached via LinkedIn at <http://www.linkedin.com/in/peterpuglisi> or via his company website at <http://brightlightapps.com>.

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.

Preface

Today there are many variants of smartphones, with iPhone and Android being the most popular. With different operating systems, native coding languages, and device capabilities, producing applications has resulted in various approaches. Initially you could only code an application using the native languages for the device. It quickly became apparent that building and maintaining an application designed to run on multiple devices was time-consuming and expensive, as this requires that you write a separate codebase for each device.

A number of mobile development frameworks were devised in order to fill the need for facilitating cross-platform application development. These frameworks enable developers to write a single codebase and to deploy on multiple devices. These frameworks have had varying degrees of success and in their initial stages restricted the developers' ability to use the device's capabilities to the full.

Titanium is one such framework that allows you to develop a single codebase for use on multiple platforms. It is not a "code once, deploy everywhere", but a "code once, adapt everywhere" situation. Meaning that even within a framework you still have to develop code specific for the target device, but the bulk of the code you develop works across platforms.

With every technology, advances can come rapidly. With mobile development, these advances are coming almost daily. As the devices and the operating systems advance, the frameworks provided for development have to be adapted. Titanium has done a good job of keeping up with these advances and enabling developers to utilize the devices' latest capabilities. With the release of the 2.0 SDK, a new era within the framework has been created.

Not so long ago, the way you were advised to develop applications using the framework incorporated applying a global namespace and

an anonymous function architecture. Thousands of applications have been written in this way and work very effectively.

In 2011, Titanium took a new direction, which was to replace the global namespace architecture with CommonJS, which is much more about applying good practices to the application architecture and enabling the developer to create well constructed and memory-efficient applications. It is a major step forward for the Titanium framework, and if implemented correctly reduces the amount of time and code required to develop a cross-platform application.

It is always difficult to know what the future will bring. It is safe to say that advances in mobile device technology, processing power, and memory availability will occur. This will enable larger and more complex applications to be developed and implemented, further reducing the requirement to have a single device for a single purpose.

With the Titanium framework adopting a CommonJS architecture, applications written prior to this will have to be modified to incorporate the new methodology. The reason for this is that going forward, Appcelerator plans to remove the global space from the framework. It is expected that the removal of the global space will cause developers extra work in adapting their current applications. This should be seen as a positive move as this new methodology will result in more robust and maintainable code.

At this point, developers who have used the global namespace architecture might just be jumping up and down saying "What!!". The future of the framework involves plans to use purely the commonJS methodology, the result being that the older applications will not be able to use the latest versions of the SDK. Do not think that you have to rewrite every application built using the global namespace method. Instead, plan ahead and think, which applications actually need to be updated? When do you need to add new functionality? Applications that have already been compiled and released into the app stores will still continue to work without modification, and using their current SDK version, they can be modified and resubmitted.

Whatever is explained about the best practices and the application architecture in this book will enable you to develop applications that will stand the test of time. The CommonJS architecture, which Titanium has adopted, will enable the framework to be enhanced and updated; it is a major step forward and contributes to the framework becoming one of the best available.

What this book covers

[Chapter 1](#), *Understanding JavaScript Patterns*, avoids the basics and jumps straight into some of the meatier aspects of JavaScript. It also provides a few best practices and methods of optimization along the way.

[Chapter 2](#), *Titanium Best Practices*, looks at best practices directly as they relate to the Titanium Mobile API itself, before diving into writing Titanium applications using CommonJS.

[Chapter 3](#), *Building an Application Using CommonJS and the MVC Pattern*, covers how to build an application using Titanium and CommonJS.

[Chapter 4](#), *Cross-platform Design Methods*, covers layout methods, which should help you to develop applications that make the best of native component styling while also allowing you to extend and customize styles dependent on platform and device type—all from within the same codebase.

[Chapter 5](#), *Using JSONDB as an Alternative to SQLite*, shows you how you can use JSONDB to not only cache remote data, but to also query it and create/update and delete records just like you would with any other regular relational database source, such as SQLite.

What you need for this book

This book is aimed at developers using the Titanium framework to develop mobile device applications.

Who this book is for

This book is for anybody who is currently using the Appcelerator Titanium framework. It is not an introduction to the framework, but an addition to the developer as they continue building their applications.

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 are shown as follows: "The global object is defined as `this` inside your `app.js` file."

A block of code is set as follows:

```
var globalVariable = 'Titanium';

function bestMobilePlatform() {
    var localVariable = 'Phonegap';
    globalVariable += '!';
    return globalVariable;
}

alert( bestMobilePlatform() ); //alerts
"Titanium!";
alert( bestMobilePlatform() ); //alerts
"Titanium!!";
alert( localVariable ); //localVariable
```

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

```
var globalVariable = 'Titanium';

function bestMobilePlatform() {
    var localVariable = 'Phonegap';
    globalVariable += '!';
    return globalVariable;
}

alert( bestMobilePlatform() ); //alerts
"Titanium!";
alert( bestMobilePlatform() ); //alerts
"Titanium!!";
alert( localVariable ); //localVariable
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "To get to the configuration settings, go through the main menu to **Preferences** | **Titanium Studio** | **Formatter**, select **JavaScript**, and click on the edit icon".

Note

Warnings or important notes appear in a box like this.

Tip

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>](mailto:feedback@packtpub.com), and mention the book title through 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 to our website, or added to any list of existing errata, under the Errata section of that title.

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.

Chapter 1. Understanding JavaScript Patterns

Since the initial introduction of JavaScript over 16 years ago it has grown from an "amateur" language often criticized for its shortcomings, into one of the most popular and useful programming languages available today. Apart from its widespread usage on the web, there are now numerous frameworks and libraries that not only use JavaScript, but place it first and foremost as the language of choice. The reasoning behind this is quite simple:

- It's free, with a variety of interpreters and engines available (including Google's V8 engine).
- It's one of the easiest programming languages to learn and being loosely-typed, it's quite forgiving.
- It's everywhere. The skills you learn from JavaScript can be easily transferred from the Web to the desktop and all the way through to the mobile, using platforms such as Appcelerator's Titanium.
- As JavaScript becomes even more popular with the rise of DOM-less engines such as NodeJS and Titanium, there is a huge and ever-growing number of third-party libraries available for JavaScript to tackle everything from OAuth to charting to date manipulation.

This is not to say that JavaScript today doesn't have its fair share of pitfalls and idiosyncrasies—far from it. To know and understand Titanium, you must first know and properly understand your JavaScript. The two are symbiotic, and the better you get at JavaScript, the better you understand its prototype-based structure, and the better Titanium Mobile developer you will become.

In this chapter, we'll avoid the basics and jump straight into some of the meatier aspects of JavaScript, as well as providing a few best practices and methods of optimizations along the way.

It's all object(ive)

JavaScript is completely classless. If you're approaching JavaScript and Titanium from another language (such as C# or Java), this fundamental change in thinking can take some getting used to. It does not mean, however, that JavaScript is not object-oriented, far from it. In fact, apart from standard primitives such as numbers, Booleans, and strings, almost everything in JavaScript is an object.

Whenever you declare a variable or a function in JavaScript, you effectively create an object. Functions, for example, automatically have the prototype object added to them as a variable whenever they are initialized, meaning all functions are already objects that contain at least one other object. If that sounds complicated then rest assured, it's really not. Objects are nothing more than a collection of named properties (key-value pairs), it's just that the values of these properties can be anything from a variable or a primitive to another function or object. These objects can be modified at any point during your application's lifecycle, including having members removed, changed, or added.

Note that even though there is not really a "method" as such in JavaScript, functions that are properties of other objects are often called methods in order to distinguish them from their parents.

Finally, it pays to note that there are two kinds of objects:

- Native (or built-in) objects, such as Array, Date, or Math
- User-defined objects, which are those that you create yourself

Understanding scope and the global object

Once you understand how scope and the global object works in JavaScript, you begin to understand why many developers struggled in the early days with memory leaks and poor performance in their Titanium applications, and why Appcelerator is encouraging all developers to make the move to a CommonJS code structure.

Note

CommonJS is a specification for defining how JavaScript acts outside of the browser. Its intention is that a developer could write JavaScript code for one system, and be able to run that same code on another system (or host environment) safe in the knowledge that it will be compliant and will work on both. We will be explaining CommonJS in depth in the next chapter.

First, let's define exactly what is meant by "global" and "local" scope. In JavaScript, everything is defined in function scope. This means that the variables declared within a function are only available from within that function and not outside of it. Conversely, it also means that variables defined outside of the scope of a function or statement declaration are therefore also available from within the said function or statement. The following example illustrates the difference between local and global variables:

```
var globalVariable = 'Titanium';

function bestMobilePlatform() {
    var localVariable = 'Phonegap';
    globalVariable += '!';
    return globalVariable;
}
```

```
alert( bestMobilePlatform() ); //alerts
"Titanium!";
alert( bestMobilePlatform() ); //alerts
"Titanium!!";
alert( localVariable ); //localVariable is not
defined
```

Tip

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.

The same principle in JavaScript applies to nested functions, which is called **scope chain**. This means that functions defined inside other functions will have access to their own variables, and the variables of their "parent" functions.

```
var a = 'Appcelerator'; //global variable

function bestMobilePlatform() {
    var b = 'Titanium';
    alert(bestMobileDevice()); //alerts "Apple
iPhone"
    alert(c); //undefined, won't execute (uncomment
to test)
    function bestMobileDevice() {
        var c = 'Apple iPhone';
        alert(a + ' ' + b); //alerts "Appcelerator
Titanium";
        return(c);
    }
}
bestMobilePlatform(); //call our nested function
```

The global object

So we now have a good understanding of scope in JavaScript, as well as the difference between local and global scope. By the same token, you should now understand what the global object is. It's the top-level object that holds all global objects and includes functions and variables that are declared in global scope. In the browser, the global object is known as **window**. In Titanium, the global object is defined as this inside your `app.js` file. So why do we want to avoid globals in our code?

- **Namespace pollution:** This means that because global names are available everywhere, you may end up with a situation where you are using a global object when you think you're using a local one, or where you end up declaring a global object unintentionally (for example, because of a typo).
- **Non-locality:** Good code is easy-to-read code, and the best code is created when it is modularized—meaning lots of small, localized functions or objects that are limited in their scope, making up the app as a whole. Over reliance on globals means that you will end up with a large mess where one part of your app can be hard to distinguish from another, otherwise known as "spaghetti code". It also means that every time your code needs to find a variable, it has to search the entire global scope which can slow your application down.
- **Collisions:** If you declare global variables in your `app.js` file, and then include some external code using `Ti.include()` that use the same variable names, this can cause collisions in your code. It is one of the main reasons why Appcelerator (and this book) advocates the usage of CommonJS, which will be discussed in depth later in the book.

Code structure and formatting

JavaScript can be pretty forgiving, and if you're not careful, you can get into some pretty sloppy habits as a JavaScript developer. We've all seen code before that omits the semicolons, neglects the curly brackets, and in general is just poorly formatted. Often this is just an irritation as it makes code harder to read and understand, but sometimes it can lead to unwanted and unusual behavior.

Use semicolons and curly brackets

Let's first take a look at an example. While the following code is badly formatted, it will execute just fine and run without error:

```
//this is badly formatted code, but it will
execute just fine
if (iLoveJavaScript === true)
    var message = 'Huzzah!'
```

What happens if this function is extended in order to make another function call? What if our comparison happens to be true?

```
//this is badly formatted code, and now it's
broken code
var iLoveJavaScript = true;

if (iLoveJavaScript === true)
    var message = 'Huzzah!'
    anotherFunctionCall();
```

You may initially think that the previous code would set our `message` variable to `Huzzah!` and then execute the function called `anotherFunctionCall`, but you'd be wrong. What actually happens is this:

```
//this is what actually executes
var iLoveJavaScript = true;
```

```
if (iLoveJavaScript === true) {  
    var message = 'Huzzah!';  
}  
  
anotherFunctionCall();
```

This means that regardless of whether the variable `iLoveJavaScript` is `true` or not, `anotherFunctionCall()` is always going to be executed. It is therefore best practice to always use curly braces, and always end lines of code with a semicolon, even if technically they aren't needed.

eval is evil

This should go without saying, but avoid the use of `eval()` at all costs. In case you weren't aware, the `eval` method takes a string containing JavaScript code and executes it at runtime. It's slow and full of potential security risks. While there are rare cases where you may want to use `eval` (for example, for complex user-generated mathematics, or inside a tool that runs code on the fly for testing purposes), there are almost always better methods available to you than the `eval` method.

Note

The alternatives to not using `eval` can be complex. One suggestion to avoid `eval` could include using a function instead. This keeps the evaluated code out of the global scope, i.e. `new Function(data)();`.

The curly bracket position

Implementing brackets is obviously important for good JavaScript code, but just as important is the location of your opening curly bracket. Developers' preferences on the position of the opening

bracket tend to come from whatever language they are most used to. For example, in C# (.NET) code you would format your opening statements to have the curly bracket on the following line—in fact the IDE automatically formats it like this for you:

```
if(bangTidy === true)
{
    //do something within our if-statement
}
```

Others tend to place the first bracket on the same line as the `if` statement, which is the preferred way of formatting in JavaScript, as there are certain times when the JavaScript interpreter can make your code behave in unwanted ways. Take the following code for example:

```
//you would expect this if-statement to return an
object with a name
//and an age property
if(bangTidy === true) {
    return
    {
        "name": "Keith Lemon",
        "age": 27
    }
}
//in reality, because of the position of the
opening bracket, what is // really happening is
this:
if(bangTidy === true) {
    return undefined;
    //this code is never reached
    {
        "name": "Keith Lemon",
        "age": 27
    }
}
```

So it's best practice to always have your opening curly bracket on the same line as your opening statement, to avoid potential parser errors.

Not all operators are created equal

JavaScript implicitly typecasts variables when they are compared. Let's take a look at a couple of non-explicit comparisons and review the results:

```
false == 0      (returns true)
0 == ''         (returns true)
true = 1        (returns true)
```

To avoid confusion (and for general readability) it is better to use the strict form comparison operators. This means instead of writing `==` or `!=` you would instead use `===` or `!==`. These operators check both the type and value of the two variables you are comparing, as opposed to the standard operators which compare based on value alone.

```
false === 0     (returns false)
0 === ''        (returns false)
true !== 1      (returns true)
```

While not mandatory, comparing variables based on both type and value is a good practice to get into, as it makes your code look more consistent and can help you avoid errors in logic.

Formatting, commenting, and naming conventions

While not strictly enforced, the following name conventions are considered by many top JavaScript developers to be in keeping with best practices. They help to ensure that your code is readable and

maintainable by any other developer who happens to come by and use it.

Object type	Best practice	Example
Constructors	As constructors are just functions, the best way to differentiate them is by capitalizing the first letter.	<pre>var ford = new Car();</pre>
Constants	Typically, constants in JavaScript are fully capitalized in order to set them apart from regular variables. Titanium's own API follows this naming pattern.	<pre>var PI = 3.1415926535897; var FACEBOOK_API_KEY = '29239101199';</pre>

Object type	Best practice	Example
Function and method names	Public functions in JavaScript are generally defined in CamelCase. Titanium's own API follows this naming pattern. Private functions are often defined using an underscore as the leading character in the name. This doesn't make them automatically private, but it does infer to the reader that they should be treated as such.	<pre>function goWest() {}; Ti.UI.createImageView() ; var ford = { _getModel: function() { return 'Escort'; } };</pre>

Object type	Best practice	Example
Variable declarations	If you're declaring multiple variables at once, consider doing so in one statement, as it can make your code more concise and easier to read.	<pre>var width, height, depth;</pre>
Comments	It goes without saying (or it should)—comment your code! For most, this means block comments at the top of a file or function, and in-line comments where	<pre>/* I am a block comment */ //I'm an in-line comment</pre>

Object type	Best practice	Example
	necessary throughout.	

Prototype

Understanding how Prototype works is an important part of understanding JavaScript, and an important part of understanding how Titanium Mobile works with CommonJS. Every object in JavaScript has a built-in property called `prototype`. The `prototype` property is created whenever a function is defined. The initial value of `prototype` is always an empty object. You can alter this empty object with your own properties, functions, and methods.

Let's say you are to create a new object called `Car`. This object has a number of properties that you wish to set using the constructor. We may want to create a new instance of `Car` called `hatchback`, and pass into the constructor a number of variables that describe the car in detail such as `model`, `engineSize`, and `color`. First, let's create our new constructor function called `Car`:

```
function Car(model, engineSize, engineType, color)
{
    this.model = model;
    this.engineSize = engineSize;
    this.engineType = engineType;
    this.color = color;

    this.engine = function(){
        return this.engineSize + ' ' +
this.engineType;
    };
};
```

Adding properties and functions to the prototype allows us to extend the functionality of this constructor's functions. Here's how we can do that using `prototype`:

```
Car.prototype.yearOfManufacture = 2012;
Car.prototype.manufacturer = 'Ford';
Car.prototype.getInfo = function(){
    return {
        engine: this.engine,
```

```
        modelDetails: this.model + ' ' +
this.yearOfManufacture,
        manufacturer: this.manufacturer,
        colorOption: this.color,
        price: '$15,000'
    };
};
```

Note

This only applies to your own objects, not those built-in Titanium proxy ones such as `Ti.UI.View`.

Using the methods and properties added to the prototype is then easy, as shown here where we are creating a new object called `hatchback`. You'll note from the output that all the functions and variables that we predefined in the prototype are now already available as part of the `hatchback` object.

```
var hatchback = new Car('Focus', '1.4', 'Petrol',
'Blue');
alert(hatchback.yearOfManufacture); //alerts 2012
alert(hatchback.manufacturer); //alerts Ford
Ti.API.info(hatchback.getInfo()); //echoes the
getInfo object
```

So why is this so beneficial in Titanium? Imagine you had a view, and it was called `MyView`. This `MyView` object held a couple of labels, an imageview, and a tableview—all of which need to be defined before you go off to your database or web service and fetch data to assign to their value/data properties. Using `prototype`, we can define all these things when we're creating a new object using the constructor of `MyView`. This means greater code reuse, and greater isolation between different parts of your code, making it overall more maintainable.

We will be using prototype a lot during the chapter on CommonJS and providing specific examples of how to utilize it efficiently in

Titanium.

Micro optimizations

While the following practices may not bring you vast speed improvements (and not doing them will certainly not make your code "bad" in any way), they are nonetheless good habits to get into.

Declaring variables outside the for statements

When you're executing a lengthy `for` statement, there is no need to make the JavaScript engine work harder than it has to. Take the following example, which calculates the length of the array `myArray` every time the loop reruns:

```
//avoid constant recalculation of the length of
myArray
for(var c = 0; c < myArray.length; c++){
    //do something with myArray[c]
}
```

A much better way of writing this `for` statement is to cache the length of the array in a variable. This way, the value of `myArray.length` is retrieved only once, and used during the entire loop.

```
//a better way of looping through myArray
var max;
for(var c = 0; max = myArray.length, c < max; c++){
    //do something with myArray[c]
}
```

If the order that you access the array values in is not important, then it's also slightly quicker to count down than up. This is because it's more efficient to count down to zero than it is to calculate and equate your values to an arbitrary number, such as the length of `myArray`. For example:

```
//looping backwards is faster
var c;
for(c = myArray.length; c--;){
    //do something with myArray[c]
}

//alternatively, you can use a while loop in this
instance
while(c--) {
    //do something with myArray[c]
}
```

Using forEach instead of for loop

Where possible, it's best to use a forEach loop in lieu of a standard `for` loop, particularly if you're looping through array values. For example, we could alter our previous sample code to look something like this:

```
function printResults(element, index, array){
    //do something with myArray[index];
    Ti.API.info(index + ' = ' + element);
}

myArray.forEach(printResults);
```

Using shortcuts for simple if statements

Sometimes, you simply need to check a single value and determine whether it matches a specific value before proceeding in your code. This might be as simple as determining whether an object is `null`, or checking that an operator is `true` or `false`. In many cases, it's simpler to use a few shortcuts.

```
//here we are checking if this object is null
var myObject;
if(myObject !== null) {
    myObject = {
        name: 'Boydlee'
    }
}
```

```
    };  
}  
  
//a better way of declaring this object would be  
var myObject = myObject || {  
    name: 'Boydlee'  
};  
  
//in this example we're checking if a value equals  
true, and  
//assigning a variable accordingly  
var dinner = '';  
  
if(isBritish === true) {  
    dinner = 'Burger & Chips';  
}  
else {  
    dinner = 'Hamburger & Fries';  
}  
  
//a simpler way to perform this check would be to  
use the  
//pattern "variable = (condition) ? True-value :  
false-value;  
dinner = isBritish ? 'Burger & Chips' : 'Hamburger  
& Fries';
```

Summary

Having a good working knowledge of JavaScript fundamentals is going to ensure that you write better, faster, and more stable Titanium applications, while sticking to the general principles of good code structure and formatting will keep your code neat, concise, and as error-free as possible. There are many books available online that deal specifically with JavaScript patterns and best practices to a much greater depth than we have gone into here, and it is well worthwhile to have at least one of these on hand at all times.

Here are some suggested books you might find useful:

- *JavaScript: The Good Parts* by Douglas Crockford:

<http://www.amazon.co.uk/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742/>

- *Object-Oriented JavaScript* by Stoyan Stefanov:

<http://www.amazon.co.uk/Object-Oriented-Javascript-Stoyan-Stefanov/dp/1847194141/>

In the next chapter, we are going to be looking at best practices as they relate to the Titanium Mobile API itself, before diving into writing Titanium applications using CommonJS.

Chapter 2. Titanium Best Practices

With the release of Titanium 2.0, a lot of new features and APIs were introduced, though the biggest change is the fully loaded integration of CommonJS. CommonJS started to appear at the end of 2011 and has meant a complete rewrite of the best practices and the application structure. Best practices are always open to interpretation; they are a good starting point but not a definitive solution.

With so many computer programming languages around, people are always trying to find better means to improve the way developers write code and come up with a standard set of practices. Some companies take this policy to a level where they define a complete set of standards. However, for most developers there are no predefined rules laid out or enforced, so they work out their own.

Not following a set of standards or best practices has various consequences, ranging from bad code layout (not a crime, unless you meet the next person to modify the code) to badly performing applications. To define these standards for Titanium we have to start with CommonJS, as it plays a major role in the whole framework.

CommonJS

CommonJS is a set of specifications, and its purpose is to give a common guide to building JavaScript frameworks. It is not a framework in its own right. Appcelerator implementing these standards to their Titanium framework brings it inline with other frameworks such as NodeJS, and enables us as developers to use the same practices in more than one framework.

The implications of this cannot be overstated. Developers can now switch between JavaScript frameworks, which have implemented the CommonJS model, without having a massive learning curve on the framework itself.

CommonJS works by using an initial bootstrap file, which in Titanium is `app.js`. You then abstract your code into separate modules that are required into other modules when needed. We will cover how to architecture the application in the next chapter.

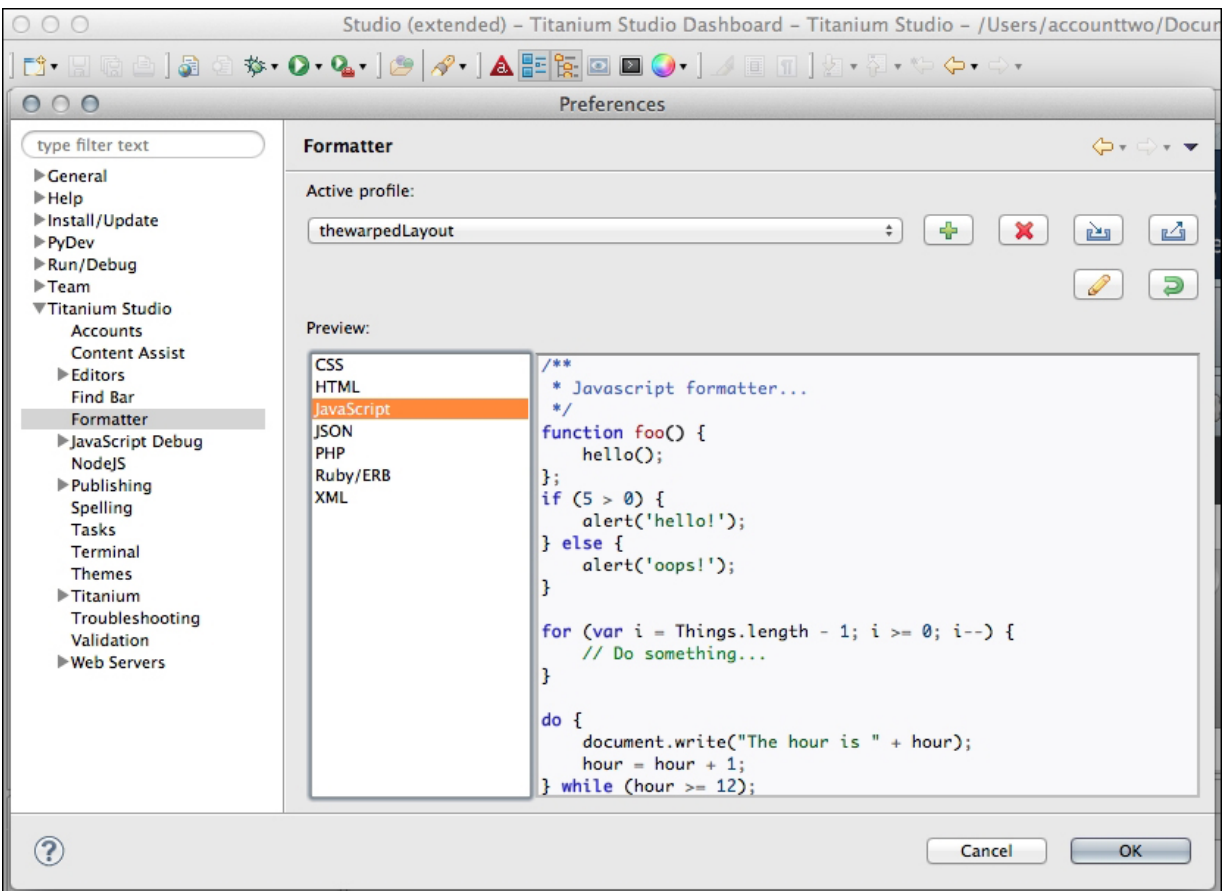
Note

Within Titanium, you can have native modules that enhance and expand the Titanium framework. These should not be confused with a `CommonJS` module, which is part of the main application.

Code formatting

How often have you needed to modify code that was written by other people or has been updated, modified, rewritten, or just generally messed around with? You find different people using different coding styles, braces on different lines, and equals signs at different positions; the list is endless. You then have to work your way through this code, getting more and more annoyed.

Titanium Studio has a solution—an automatic code formatter, which you can set up as you require. To get to the configuration settings, go through the main menu to **Preferences | Titanium Studio | Formatter**, select **JavaScript**, and click on the edit icon. You will see the **Preferences** panel and you can configure it as required. If you wish to have these configurations across your development team, then you can export and import them as required.



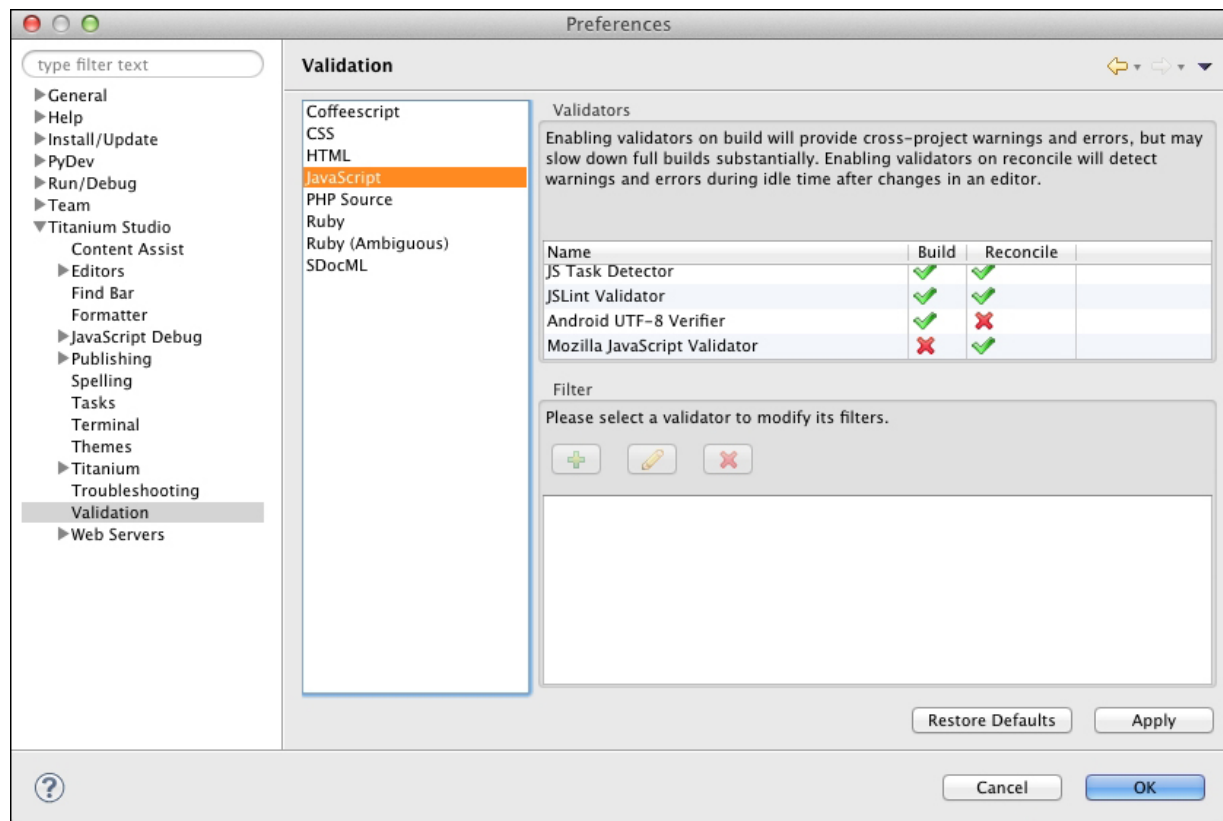
To format a file's code, open the file, go to the main menu, and select **Source | Format**. Alternatively, if you are on a Mac, the relevant shortcut keys are *shift+command+F*. If you have selected a section of code then that will be formatted, otherwise the whole file will be formatted.

Note

A gotcha with the formatter: If there are too many JavaScript errors it won't format the code.

Code validation

As developers, we spend our days hunting down spurious code anomalies. It could be a missing comma, semicolon, or that an event listener wasn't added while looping an array of objects. Anything that can make this task easier and show potential issues while we are writing code is a good thing. **JSLint** is one of those tools that you could use, as it highlights potential issues as you code. It has been described as your worst nightmare but also your best friend. When you first start using it, it may drive you mad with some of the seemingly meaningless warnings, but sort them out and be persistent; it will improve your codebase in the long term.



Titanium Studio has JSLint built in but switched off by default. To enable it, go to **Preferences | Titanium Studio | Validation**, select **JavaScript**, and switch on **JSLint Validator**. As you can see, there are other validators on the list, which also help give you a hit list of

potential issues. It is worth spending a little time making sure you understand what these are and how they work. Explaining these is outside the scope of this book.

Comment meaningfully

Adding comments to your code should not be seen as a chore, instead they can be as important as the code itself. A well-commented codebase can and will save hours in the future, as you or a colleague go to update and maintain it.

Always put a comment block at the start of source files, explain what the file does, and include a maintenance log, where the date, time, developer's name, and a brief description of the changes can be maintained. If you have a complex function put a comment block before it, explaining what it does. Also place inline comments where needed to explain certain pieces of code.

```
/*
 * A CommonJS Module.
 *
 * This module does something ...
 *
 * Author : Name
 *
 * Date : Today
 *
 * Maintenance Log
 *
 * Date :           Author:
 * Changes :
 *
 */
```

Note

Do not add comments for the sake of adding them. Make them meaningful, relevant, and useful. Too many comments can confuse the code structure.

Do not pollute the global object

Within an application you can define various application objects. Declare and use these with caution, as they use up resources and can easily cause clashes with other private and local objects. Application-level variables are not really required and should be avoided. If you require a small piece of data to move around the application, consider using persistent data or passing it to the required modules.

Application-level event handlers are required for various tasks including, amongst others, background services and geolocation. If you do use them, always remove them when they are no longer needed. To control the flow of an application, you may need to set up a global listener, but you only need one with a common function to control the flow. This will be seen in more detail in the next chapter.

Note

In CommonJS there is no global scope; declaring a variable within a module makes it private to that module. Declaring variables in `app.js` as `Ti.App.varName = []`, does make them global, but is highly discouraged.

JavaScript instance

A JavaScript instance is where a session of the interpreter is invoked. In Titanium it is possible to create multiple instances within a single application. This is done by creating a window object with a `url` property to load the content.

```
Var win = Ti.UI.createwindow({  
    url: '/app/ui/newWindow.js'  
});
```

Don't do this unless you have a very, very specific requirement. To benefit from all the advantages CommonJS provides, always work in a single JavaScript instance. The consequences of multiple instances include no scope across them, additional memory and resource usage, and a high risk of memory leaks.

CommonJS modules

With the adoption of the CommonJS specification by Appcelerator into the Titanium framework, you should only use the CommonJS modules, which are also referred to as **factories**. This provides many advantages: separation of code into specific modules, a more structured codebase, separate object scope, code maintainability, and much more.

By using this method it becomes very difficult to pollute the global scope, as each module has its own object scope. Understanding this is key to the CommonJS method. Each module or factory contains functions and variables that can be exported; unless they are exported they are private to that module. This enables variable or function names to be the same in different modules without a scoping clash. By exporting only what is required at the end of the module, it enables the module to be self contained.

```
/* A typical module format
myModule.js */

var object1 = "1234";
var object2 = "5678";

/* My modules function */
function myModule(args){
    ... do something nice
    return mainObject;
}
exports.object1 = object1;
exports.myModule = myModule;
```

When a CommonJS module is required by another module it is loaded into memory. If you then require the same module elsewhere, it doesn't reload it into memory; it just makes it available to the new calling module. This means that if the first requiring module sets a value in the called module, when the called module is required by another module, the values which have been set are still set. It

doesn't load a new instance of the module and reinitialize all the values.

A few rules about modules:

- Only load them when needed
- Only export what is required by the calling module
- Use `prototype` where appropriate
- Avoid recursive requires

Working with CommonJS recursive requires can cause major issues and, basically, you can't do it. A **recursive require** is where module A requires module B and module B then requires module A. You will quickly notice if you try this that this leads to a loop trying to process the continual call of the requires, and finally dies with a nondescript error message.

CommonJS best practices

CommonJS is one of the best things to have happened to the Titanium framework. To get the most out of the framework and the enhanced performance, these are a few things that should be considered:

- Be modular
- Be private
- Return an object
- Protect the global scope
- Control file loading

One of the main advantages of CommonJS is the way it lends itself to creating well-structured, separated code. By being modular you create specific modules for separate sections of the code, i.e. a separate module for each window. This methodology facilitates the creation of common modules enabling a constructive codebase that is easy to understand and maintain. A good example of a common module would be one that contains the geolocation code and is then used across the whole application.

Common modules enable the code to be extracted down, but don't go too far. It is tempting to extract code out into its own module when it actually lives in the module it is in. Do not be tempted to take code modularization to the extreme, having a module for each function is not necessary or productive. Remember that modules are loaded into memory once they have been required and they remain there.

Making functions and variables private to a module maintains the module's integrity. Do not export all the functions in a module unless they are actually called from the requiring module. Export the required functions and variables at the end of the module, not by default. The following code example shows the two methods for exporting functions:

```

// This exports the function inline
exports.outFunc = function () {
    .. code ..

    return object;
}

// This only exports the function when and if
required.
function outFunc() {
    .. code ..

    return object;
}

exports.outFunc = outFunc;

```

By defining the module functions in the second method, they become local to that module. This means that they can be used directly by any other function within the module. As you separate your code into modules, separate your modules into functions. Having one exported parent function that returns the main object after processing through other functions is a good practice.

It is quite easy to declare module variables; you just declare them outside of any function. This gives them a scope that is global to the module. It is a very good way of maintaining a persistent state across the calling modules, but use them sparingly as they use up more resources.

JavaScript, by default, returns the last action of a function to the calling function. This may not always be what is required, especially in the CommonJS model. Always return what you require from a function even if that is nothing.

```

// New window function returning the null object.

functionnewWin() {
    // Do something
    return;
}

```

```
exports.newWin = newWin;
```

For an exported function always return the function's main object. Hence, creating a new window in a function should return that window object as the following code example shows:

```
// New window function returning the parent
object.

function newWin() {
  var win = Ti.UI.createWindow({
    .. parameters .. });

    return win;
}

exports.newWin = newWin;
```

The global scope should be considered as nonexistent, but you can still add application-level variables. If you have to use these, only declare them in the `app.js` file; do not declare them in any of the modules. At times you will have to create an application event listener in a module. Only do this when you have to and always, always remove the listener after you have finished with it. The following code example shows an event listener added with an in-built function. This is not good practice, as you cannot remove it later. The only way it will be removed is when the parent object is destroyed, which for application-level listeners is when the application is stopped, not put into background.

```
function adList() {
  var mainObj = Ti.UI.createImageView();

  mainObj.addEventListener('click', function(e) {
    .. do something here .. });

  return mainObj;
}
```

Instead of defining listeners with an in-built function, always declare them to a calling function as the following example shows. They can then easily be removed, as you have to declare the called function to be identical in both `addEventListener` and `removeEventListener`.

```
var mainObj = null;

function eveList(e){
    .. do something here ...

    mainObj.removeEventListener('click', eveList);
}

function adList() {
    mainObj = Ti.UI.createImageView();

    mainObj.addEventListener('click', eveList);

    return mainObj;
}
```

Managing memory

We have explored some of the practices that will help in managing the application's memory. These practices range from controlling when the `modules` file is loaded to not using application-level events or variables. We will see in the next chapter how structuring an application can also aid this, as there are some key points that need to be considered.

When you open a window, it is added to the window stack, and every time you open that window it is added to the stack. If you have a navigation system that enables you to move through the windows in any order, it is likely that you will end up with a large stack. Close the windows when they are not in use.

```
varwin =i.UI.createWindow();

varwinBut =i.UI.createButton({
    title : 'Press' });

function loadWin2() {
    varwin = Ti.UI.createWindow();

    win.close();

    win = null;

    win2.open();
}
winBut.addEventListener('click', loadWin2);

// opening a window
win.add(winBut);
win.open();
```

Some additional memory-intensive APIs in Titanium are web views, large tables, and events. It is not a good idea to have more than one web view instance running in the application at a time. If you do not close the window containing the web view instance and move to another one, at least remove the web view instance from the

previous window and reload it when focus is made. The same applies to tables and map views.

When you remove an object such as a web view from a window it may not always release the memory—even closing a window does not always release the memory. In all cases, `null` is your friend. The final code example shows a window with web view, an event listener, and how to clean up when a window is closed:

```
// clean up example module

function createWin() {
var win = Ti.UI.createWindow();

var webV = Ti.UI.createWebView();

    win.addEventListener('open', openFunc);

win.addEventListener('close', function(e) {
win.removeEventListener('open', openFunc);
win.remove(webV);
webV = null;
    win = null;
    })
win.add(webV);

    return win;
}
```

In this example the `close` event doesn't call another function. This is acceptable because when we close the window, the close event will fire, which nulls the `win` object and this removes the object's event listener. It is done this way to prevent having to use module variables to handle the window and web view objects being cleaned up in another function.

Summary

As you have seen, there are quite a few considerations to take into account when coding with Titanium. Some of these can also be applied to other programming languages. It is completely optional to follow best practices; they are there as a guide, as a place to start, and as a way to manage your code going forward.

As a developer you will find your own way to implement the methodology used within the application, you will decide when, where, and what comments to add, which code format to use, and which module to put what code into. But best practices and guidelines are developed for a reason; they keep code consistent within the application, they allow other developers to pick up what is going on quickly, and enable clean and reliable code.

Note

Always apply a good coding style to your application. You will thank yourself in the future.

In [Chapter 3](#), *Building an Application Using CommonJS and the MVC Pattern*, we will cover how to structure the application architecture using commons.

Chapter 3. Building an Application Using CommonJS and the MVC Pattern

In the previous chapters, we looked at the best practices in both JavaScript and Titanium. Applying those without implementing a good code structure makes them far less useful, and the code much less maintainable.

This chapter will explore a solution to building a Titanium CommonJS application. It is not the only way, and it may not fit with the way you or your company develops, but it works, is efficient, and gives a good grounding to which you can apply your own standards.

Standards

Defining a set of standards, as part of a framework, and applying them to an application's architecture is very personal. No one works in exactly the same way as someone else, and nobody really enjoys working with code that does not confirm to their way of working.

The way to resolve this is to produce a set of standards that everyone adheres to. Since software development started becoming mainstream, companies produced internal coding standards, defined how a developer works within the confines of an application structure, but also gave the developer enough freedom to create the required application. With each new language, framework, or technological advance, these standards need to be modified, updated, or scrapped and restarted.

Even now standards are a thing of personal choice, and one of the hardest things to implement into a development team. As you work through this chapter, give a thought to how you apply standards,

what effects it will have on others, and how useful they will be going forward.

Note

The software development industry has been mainstream for a long time. There is nothing new about standardization—you just have to decide on your approach and apply it.

Frameworks

Titanium is a framework. It enables the programmer to develop apps quickly and efficiently, but at a cost. This cost is two-fold; the first being that the size of the final application is larger and that it performs slower than natively-written applications, although not significantly in well-structured applications. It does have its advantages too; it is quicker to develop, it encompasses more than one platform, it is easier to learn, and it enables coding standardization to be applied.

In fact, any framework will add size and performance issues to any application. As you go through this chapter you will see one way of implementing an application architecture, but as mentioned there are others.

Note

An MVC architecture framework, named Alloy has been developed by Appcelerator and released in Version 3 of the framework, but it is in its early stages.

Like many companies and individuals, if you have decided to implement an architectural framework in your application, then you need to be aware that it has consequences. Unlike desktop computers, a mobile device's resources are limited, and frameworks will eat up these resources. One of the most common MVC frameworks used in PHP is **pureMVC** (<http://puremvc.org/>), which is now also available for JavaScript.

The advantage of adding an architectural framework to your application is that it enforces coding conformity. It requires a standard approach to use it, and it enforces consistency across the code base. Along with these benefits, though, you will also have the

disadvantages, and it is up to you to decide if adding an architectural framework into your application is really required, or if you can define your standards sufficiently so that one is not needed.

Note

Architectural frameworks are useful for enforcing good coding standards, but they impact the application size and performance.

The architecture

We will explore an architecture, which is based on the **MVC (Model View Controller)** model. It is intended to show you what is possible and one way of implementing it. The implementation uses the `app.js` bootstrap file only to load the controller. It is the controller that handles the application flow processing and you will see through the rest of this chapter how it all fits together.

CommonJS

With the implementation of CommonJS into the Titanium framework, the application architecture will use this set of standards exclusively. In the last chapter we covered what CommonJS is and the best practices involved. Although you can use the `Ti.include` and `createWindowUrl` methods, these are no longer the preferred coding techniques.

If you are using CommonJS for the first time, the thing that will become apparent very quickly is the fact that the individual scope of a module presents some issues with application navigation, but by using custom event handling and function callbacks, these can be overcome easily.

One potential pitfall of CommonJS is that we are tempted to use a lot of application-level event handlers and variables, because this is how the `Ti.include` methodology worked. CommonJS requires a different mind set, a different way of approaching the application development, and a different application architecture.

The bootstrap

In Titanium there are two key files, `tiapp.xml` and `app.js`. `tiapp.xml` is the application configuration file, where you set the application name, version, orientation, and a lot more. `app.js` is the bootstrap file, it is the file that is called every time the application is started. You have to handle the background processing of an application for it to run the code in `app.js` again.

Note

`app.js` runs every time the application is started. It is very significant. If the application goes to the background, reopening the application does not rerun it.

There are various thoughts currently going around about what the `app.js` file should contain. Some think it should control the whole application, where others think it should just require the initial controller file and run the initial function—this solution uses the latter. The source code for our `app.js` file is as follows:

```
/*
 * app.js
 * =====
 *
 * This application shows a method of Titanium
 * architecture
 *
 * Author : Name
 *
 * Date : Today
 *
 * Maintenance Log
 *
 * Date :          Author:      Changes :
 *
 */
```

```
/*
 * The require of the application controller
 */
var control = require('/control/controller');

control.startTheApplication();
```

The `app.js` file initially has comments to specify what the application is for and then has just two lines of code. The first line requires the controller file and the second calls the initial function. No event listeners, application settings, or global variables are set.

Having the `app.js` file like this has quite a few advantages, the main one being that it becomes very difficult to pollute the global scope and that it also enables code to control the application instead of the settings.

Note

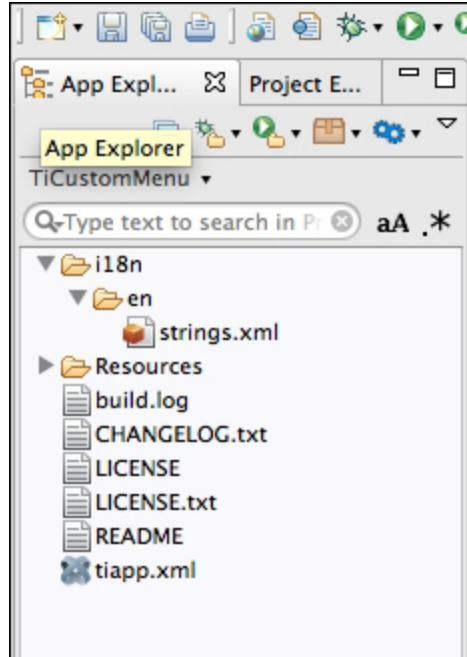
If you require global application settings, i.e. orientation, use the `tiapp.xml` file.

Multilingual

It is always a good practice to set up your application for multilingual language translation at the start. If you know for certain that it is only going to be one language, then there is nothing to worry about, but if your app becomes popular you will want to translate it for a wider audience.

The biggest part of any translation is modifying hardcoded text. Fortunately, Titanium provides a mechanism for extracting the text from the code and putting it into an XML file, which can then be translated as required.

At the application's parent level you add a directory `i18n`, with subdirectories for each language that you require, that is, `en` for English. The `en` directory is also the default directory, so if a language doesn't exist it will use this one instead.



Within the `en` directory there is a `strings.xml` file that contains the text fields which Titanium will use for the English version of the app—this is just a series of simple key-value pairs. The following code

block shows how this file is formatted—it uses standard XML to create the strings of text required. Separate specific text sections by using comments (it makes it much easier to find what you need going forward) and use camel case for the string names.

Note

Just like with any standard XML document, some special characters can cause issues in this file. Make sure you avoid characters such as ampersand (&), greater than and less than (>, <) symbols by using their character entity equivalents. In the case of the ampersand, & can be represented by &. Also be sure to escape any single or double quotes using the backslash symbol.

Consider the following code example:

```
<?xml version="1.0" encoding="UTF-8"?>

<resources>
    <!-- Application Text -->
    <string name='appName'>Hello World</string>

    <!-- Error Messages -->
    <string name='errorA'>Sorry the load
failed</string>

</resources>
```

Within the code, to access the required string you use the `Ti.Locale` API, which can be shortened to `L`. Although `L` is available application-wide, it creates a warning in JSLint and can confuse newer developers, so we will be sticking to the full `Ti.Locale()` namespace throughout our examples.

```
/* accessing the localized string */

var newLabel = Ti.UI.createLabel({
```

```
        height : Ti.UI.SIZE,  
        width : Ti.UI.SIZE,  
        text : Ti.Locale.getString('appName')  
    });  
  
    // or  
    var newLabel = Ti.UI.createLabel({  
        height : Ti.UI.SIZE,  
        width : Ti.UI.SIZE,  
        text : L('appName')  
    });
```

Directory structure

The directory structure of your application is critical to its maintainability, so our recommendation is that you use a sensible naming convention and place the relevant code within them. This makes it much easier to find and maintain the code. Within Titanium you have some predefined directories and application-wide files, including:

```
/* The directory overview */

i18n/
  en/
    strings.xml

Resources/
  android/
  iphone/
  app.js

tiapp.xml
```

The `i18n` directory is where the multilingual capabilities of Titanium are applied. `Resources` contains all the actual application code and assets, `tiapp.xml` is contained within the parent directory and contains all of your application-specific settings such as module includes, app identifier, and publisher details.

Before we look at each of the sub-directories of `Resources` in more detail, we need to devise a good working directory structure for the main application code. Based on an MVC model we need to have containers for the controller, data models, views, helpers, and assets.

```
/* The resources directory structure */

Resources/
  android/
    assets/
```

```
        images/  
            /low  
            /med  
            /high  
    iphone/  
        assets/  
            images/  
    control/  
    model/  
        services/  
        data/  
    ui/  
        common/  
        windows/  
    tools/  
        common/  
    assets/  
        images/  
        videos/  
        pdf/
```

Any resource you put into the main `assets` directory becomes common across all platforms. The `android` and `iphone` directories within Titanium are used to place device-specific resources. It primarily enables different image sizes and resolutions to be stored and use when required.

Android

The `android` directory is used to store specific android assets. It enables you to place the images into the resolution-specific directories and at runtime the correct image will be loaded. Be aware that the directory structure under `android` has to be the same as the directory structure under `Resources`.

There are two ways to define device-specific directories within the `android` directory. The first being how the structure is created when a new project is created. This enables resolution-specific splash pages to be applied.

```
/* Android Resources directory structure */

Resources/
  android/
    images/
      res-long-land-hdpi/
      res-long-land-ldpi/
      res-long-port-hdpi/
      res-long-port-ldpi/
      res-notlong-land-hdpi/
      res-notlong-land-ldpi/
      res-notlong-land-mdpi/
      res-notlong-port-hdpi/
      res-notlong-port-ldpi/
      res-notlong-port-mdpi/
```

Within these directories you can put your specific images, including the application icon and splash screens. Instead of having the `assets/images` directory under `Resources` you will have just `images`.

Another way of defining the assets to be loaded is to use the `low`, `med`, and `high` subdirectories.

```
/* Android specified resources directory */

Resources/
  android/
    assets/
      images/
        low/
        med/
        high/
```

This structure gives the solution of being able to define the images as required within a much simpler directory structure, and enables the main asset directory structure to be used at the top level. You can also use this structure for videos and PDF assets.

iPhone and iPad

The `iphone` directory works similarly to the `android` one, except you do not have the resolution-specific directories and applies to all iOS

devices (iPhone and iPad). With only normal and high resolution devices, the way to define the two different images is `image1.png` for standard and `image1@2x.png` for the high resolution ones.

The same rules apply for `iphone` as they do for `android`. Placing an image in the `iphone` directory enables it to be accessed only by that specific device.

```
/* iphone Resource directory structure */  
  
Resources/  
    iphone/  
        assets/  
            images/  
                image1.png  
                image1@2x.png  
            videos/
```

To access a specific device image within, the code is the same as accessing the image in the `assets/images` directory. The difference is that Titanium will firstly look for device-specific images in the `android` or `iphone` directories and then in the main `assets/images` directory.

```
/* Accessing a device specific resource */  
  
var newImage = Ti.UI.createImage({  
    image : '/assets/images/image1.png'  
});
```

Note

If you are using device-specific images as opposed to a common image then only place them in the required `android` or `iphone` subdirectories.

Control

The `control` directory contains the application controller file. You can have more than one controller to separate specific tasks, but for most applications a single controller file is all that is required. We will look at this file in greater depth later in the chapter.

```
/* the control directory */  
  
Resources/  
  control/  
    controller.js
```

Model

The `model` directory contains the code used for data handling. This can be the module used for HTTP access and direct internal file or SQL maintenance. The solution consists of two subdirectories, `services`, and `data`, but can have more or less of each if you require.

Services

This subdirectory contains the modules that accessed the external data via the HTTP request and any internal datafiles or local databases.

Data

This subdirectory contains the modules that held the data after it had been retrieved, enabling these modules to be used to pass the data around the system. Any internal data file, such as a small JSON file, would normally be placed in the `assets` directory under a subdirectory called `data`.

UI

The `ui` directory is where any module that creates display elements on the screen is placed. If you need to make specific files for specific devices such as the iPad, you would also subdivide this directory to

ensure that the correct file is loaded. No business logic should be included in these modules. If you're displaying data then the module should receive that data and display it. It wouldn't go and get the data for use.

The `common` directory contains any display elements which are used across screens, i.e. a common menu. It enables the code to be separated effectively and not duplicated.

```
/* The main UI directory structure */  
  
Resources/  
    ui/  
        common/  
            default/  
            ipad/  
            iphone/  
            android/  
        windows/  
            default/  
            ipad/  
            iphone/  
            android/
```

You do not need to separate the code into the device-specific directories unless that is required, but if you do, defining the path to these directories is straightforward.

```
/* Selecting the device */  
  
var path    = 'default/';  
var osName  = Ti.Platform.osname;  
  
/* Decide if it is a tablet */  
  
switch(osName) {  
    case 'ipad' :  
        path = 'ipad';  
        break;  
    case 'iphone' :  
        path = 'iphone';  
        break;  
    case 'android' :  
        path = 'android';  
        break;  
}
```

```
        path = 'android';  
        break;  
};  
  
require('/ui/windows/' + path + '/' + 'window1');
```

You can separate this down further by using the device's height and width parameters and, if needed, the DPI. Remember, though, that the more the files, the larger the application is once it is compiled.

Tools

The `tools` directory contains any logic code that doesn't fit anywhere else. It is quite often referred to as **helpers** in MVC. The type of code which could be contained here is any geolocation, business logic, or processing. Split this directory down into subdirectories to make your code easier to find.

Naming

There are a few points to recap on:

- Try to only use alphabetic characters in the directory and file names.
- Use camel case and avoid using spaces or specialised separators. (Camel case is where each word in a name is capitalised, except the first word, for example, `aNameOfAFunction`).

Note

A final note is to make your directory and file names meaningful. It makes it much easier to find the specific module later.

The application

In this section, we deep dive into creating the code for a basic MVC architecture application. It will go into the details of how to set up the controller, manage the application flow, and separate code into specific modules. Although it is a very basic application, it shows how to apply event handlers, callbacks, and code separation.

The controller

The whole key to creating an effective CommonJS Titanium application is to set up the controller to handle the application flow. Within the controller, all windows should be opened and closed, all data calls handled, and any specific application processing managed.

Note

The controller does exactly what its name suggests—it controls the application.

This module has to be able to receive requests from the rest of the application for data, navigation, and processing. To achieve this, event handlers are used, and these need to be of application level. As you can imagine, in a large application, you could very easily end up with a lot of these handlers, and this could cause serious memory and performance issues.

To resolve this problem, define a single custom event handler, which receives requests containing parameters and processes them through a separate function and `switch` statement.

```
/* The Application controller custom event handler
*/

function handleError(inParam){
    // Handle your error processing
}

function loadHomeScreen(inParam) {
    // require the home screen module and open it
    here
}

function loadDetailScreen(inParam) {
```

```

        // require the detail screen and open it here
    }

    /* This function handles the application flow */
    function appNavigation(inParam) {
        switch(inParam.ACTION) {
            case 'ERROR' :
                handleError(inParam);
                break;
            case 'HOME' :
                loadHomeScreen(inParam);
                break;
            case 'DETAIL' :
                loadDetailScreen(inParam);
                break;
            case 'STARTAPP' :
                startApp();
                break;
            default :
                break;
        }
    }

    /* Add the event listener */

    function addAppNavListener() {
        Ti.App.addEventListener('APPNAV',
        appNavigation);
    }
    function removeAppNavListener() {
        Ti.App.removeEventListener('APPNAV',
        appNavigation);
    }

    /* This function is called from the app.js file
    and is the only function to be exported from the
    controller module */

    function startApp() {
        addAppNavListener()

        // Code the application starting requirements
    }

    /* export the modules functions and variables */

    exports.startApp = startApp;

```

You can see from the controller code that this is initially more code than you might use in a `Ti.include` method. But as you develop the application, handling new windows and data or general processing becomes a quick and easy task.

Working in this way allows for any module to fire an event back to the controller passing it a `ACTION` parameter and the controller will handle the processing.

Using CommonJS each module has its own scope and to access any of the exported resources of a module, it has to be required into other modules. The controller is only ever required into the `app.js` file and only exports the `startApp` function because otherwise it will require other modules using the controller elsewhere easily causing a recursive inclusion issue.

There are a couple of ways of firing the custom `APPNAV` event from another module. The first being to code the fire event logic each time. The other, and within keeping of good practices, is to extract the fire event code into a separate module, which can then be used by any other module. By utilising the `tools` directory we can create a `commonTools` module for use throughout the whole application.

Tools (helpers)

As you develop your application there will be a number of functions that you will want to use almost everywhere, and a number of functions that are required for a couple of modules. The place to put these is the `tools` directory. Placing code here, which is needed in multiple places, reduces code duplication.

Note

The `tools` directory is not for UI components, nor data handling, but for logic that fits outside of these structures.

One useful `tools` module is `commonTools`, its purpose is to contain the functions that are required throughout the application. To make it useable anywhere it has to be a standalone module whose individual functions complete the required task and are all exported. Producing this module enables small functions to be created and utilised.

The following code shows an example using these three functions:

```
/* commonTools.js located in the tools directory
*/

/* This module uses no other modules making it
stand alone and each function is standalone */

// Module global variable
var persTypes = ({TODAY : 'today',
                  TOKEN : 'token'});

// Set the persistent data
function putPersData(inParam) {

Ti.App.Properties.setString(persTypes[inParam.TYPE
],
    inParam.DATA);
```

```
        return;
    }

    // Get the persistent data
    function retPersData(inParam) {
        return
        Ti.App.Properties.getString(persTypes[inParam.TYPE
    ]);
    }

    // Fire the custom navigation event listener
    function navHandler(inParam) {
        Ti.App.fireEvent('APPNAV', inParam);

        return;
    }

    exports.putPersData = putPersData;
    exports.retPersData = retPersData;
    exports.navHandler = navHandler;
```

The three functions each contain a minimal amount of code and undertake a specific function. The key function here is `navHandler`, it is called throughout the application to pass control back to the controller when required.

Note

Always return from a function, even if it is null.

In JavaScript, a function always performs a return at its end, even if it is not declared. If the return isn't declared, it returns the last executed statement. This can have some interesting and undesired results.

CommonJS require

We have seen how to require a module into another module by using `require`, but what happens when the same module is required into other modules? To explain this we have to look at the way the module is loaded into memory and its scope.

The first time a module is required into another module, it is placed into the application's memory. If another module then requires it, it does not reload the module into memory but instead puts a pointer to the existing module in memory and enables the requiring module to access it. This is very significant, as it enables values in a common module to be updated and used in another module.

A module's scope is global to that module. Only exported functions and variables are accessible outside the module, enabling module variables, inline code, and functions to be defined as private to that module and exported for use within other modules if required. This enables us to incorporate a global variables module, a layout module, and various data storage modules, each of which are standalone. They may be required anywhere and used within the architecture of the application. Using the `exports` method at the end of the module file also enables functions to be written that are private to that module.

```
/* The global variables module */

// define a single object variable
var global = {
    appName : 'Titanium',
    testCount : 0,
    processFlag : true
}

exports.global = global;
```

To use this module within other modules, accessing and updating the data is standard.

```
/* A calling module */  
  
var globalVar = require(tools/globalVars);  
  
// using a variable.  
  
If (globalVar.global.processFlag) {  
    // Do something  
}  
  
// updating a variable  
  
globalVar.global.testCount++;
```

After the variable within the `globalVar` module has been updated, the other modules that also use this variable will see the new value.

Callbacks

Javascript is an asynchronous programming language. This means that the next command is processed before the previous command is completed. When you have to wait for a specific function to have finished before continuing processing, there are a few methods to accomplish this, the first being event handling and setting a listener, while another involves using a callback.

To understand what a callback is and how to use one, you first have to understand what a JavaScript object is. Everything in JavaScript, a variable, event listener, function, and so on is an object. If you call a function you can pass it a set of parameters for it to use, those parameters are objects, so you can pass variables and functions to other functions.

A **callback** is passing a chunk of code, normally a function, to another function for it to execute when that function has finished. This has so many obvious benefits, but is actually crucial for maintaining good application architecture. Lets have a look at some code of how to implement a callback across modules:

```
/* The parent module */

// require the common module
var exModule = require(exModule);

function subFunc(inParam) {
    // continue the processing...
}
function mainFunc(inParam){
    // do something here ...

    // set the callback function to be passed
    var callback = subFunc;

    // process the child module function passing
    the callback
    exModule.minorFunc(callback);
```

```

        return;
    }

    /* the child module */

    function minorFunc(callback){

        // process your code ....

        // Use the callback
        // This will execute the parent module
        function
        if (callback)
        {
            callback(params);
        }
    }

    exports.minorFunc = minorFunc;

```

As you can see from the example code, using a callback is straightforward, as you just pass the object of the code. But why use one? Well, if you consider how CommonJS works, to have been able to access the code of the parent module's `subFunc` function directly, you would have to require the parent module into the child module, which would cause module recursion.

You could have included the `subFunc` function into the child module, but because of module scope any values required would have to be passed down, and the processing would have completely passed into this module.

You could have fired an event from within the child module's `minorFunc` function and had an event listener in the parent module, however in this case the child module should not have been used and the `minorFunc` function should have resided in the parent module, as it is not a common function but one specific to the parent module.

Passing the function to be run when the child module function has completed is the best practice in this scenario. It effectively enables

the child module to have a specific function and be called by multiple parent modules, each passing their own callback function to be run. The callback function runs in the scope of the module the function is defined in, in this case the parent module, giving the code the ability to continue processing in the parent module when required.

Model

Having seen how to generate a module that has static and updatable values, using this method enables a way of creating data model modules. Within the `model` directory you may be storing some XML datafiles or modules that put and retrieve data from an SQLite database or a HTTP request, using web services. You may also be storing data in the form of flat files for processing at a later date.

For example, if you make an API call to a web service which returns data to be processed in blocks you will require somewhere to store it, and this is the place to put the code and modules to undertake such tasks.

```
/* A data storage module */

// Declare the data record
var dataRec = {
  fieldA : null,
  fieldB : null,
  fieldC : null,
  fieldD : null,
};

var dataArray = [];

exports.dataRec : dataRec;
exports.dataArray : dataArray;

/* Retrieve the data http module*/

function ahttprequest(callback) {
  // make a http call and on completion load the
  // data using a call back..
}

exports.ahttprequest = ahttprequest;

/* A data storage retrieval module */

// call the API and return the data
```

```

function storeTheData(inData) {
    var dataStore = require(dataStorage);

    dataStore.dataArray = [];

    var dataLength = inData.length;

    for (var e = inData.length - 1; e >= 0; e--) {
        // build the data
        dataStore.dataRec.FieldA =
inData[e].FieldA
        etc ...

    dataStore.dataArray.push(dataStore.dataRec);
    }
    // pass the control back to the controller.

    common.navHandler({TYPE : 'DATA01'});
}
function obtainTheData() {
    var httpReq = require('httpReq');

    var callback = storeTheData;

    httpReq.ahttprequest(callback);
}

```

The initial call to get the data is made from the controller, the call to process the HTTP request is made from the module and it uses a callback to process the data when it has been returned. This enables a common module to obtain all the data, and a common function to handle any errors.

Once the data has been retrieved and stored in the [model](#) module, control is passed back to the controller by using the custom [navHandler](#) function. The data is now available to any other module which requires the data model module.

UI

The UI section of the code deals only with the UI of the application, as you have seen up to now we have dealt with the application logic, obtaining and processing the data, and controlling the flow of the application.

Defining the code for the UI of the application falls into two categories.

- Common elements
- Windows (screens)

Within Titanium, you can use the tab group method to build a menu system, but it does have its pitfalls; it's not consistent across platforms and uses the window stack to maintain the navigation.

In quite a few cases, the tab group is sufficient, but if your application uses a multitude of windows, which can be accessed randomly, then consider using a custom menu. Within the controller architecture this works very effectively.

The layout module

At the time of writing, within Titanium there is no effective way of applying global styling. Using a module containing the application layout is one way to enable layout settings such as background colours and global fonts to be applied in a single place.

You can, if required, extract each window's layout requirements into its own layout module, but for this example we will be using one `global` module:

```
/* The Global layout module */  
  
var params = {  
    global : {
```

```

        windowColor : '#ffffff'
    },
    mainWindow : {

    },
    menu : {
        height : '51dp'
        width : '100%'
        background : '#000000'
    }
};

exports.params = params;

```

We will see this applied to a window later in this section.

The custom menu

To code a custom menu involves building its own module, which can then be used on every window within the application, or wherever required. Using the custom navigation events allows for closing and opening the windows as required through the controller.

If you have multiple windows loaded from a custom menu, then you will need to close the previous one and open the next one within the controller, or the application will continue to use the device's resources until it eventually crashes.

If you are using a custom menu also consider using a custom navigation bar, as it will enable you to handle the look and feel of the application across platforms much more easily.

```

/* The Custom Menu */

var layout = require('/ui/common/layout');
var common = require('/tools/common/commonTools');

function processButA() {
    common.navHandler({TYPE : 'HOME'});
}

function loadMenuBar(inParam) {
    var menuBarWin = Ti.UI.createView({

```

```

        height : layout.params.menu.height,
        width  : layout.params.menu.width,
        bottom  : 0
    });
    var menuButtonA = Ti.UI.createView({
        left      : 0,
        width     : '25%',
        top       : 0,
        bottom    : 0,
        backgroundImage :
'/assets/images/buttons/buttonA.png'
    })
    menuButtonA.addEventListener('click',
processButA);

    // Always return the main object

    return menuBarWin;
}
exports.loadMenuBar = loadMenuBar;

```

Each menu item can be added and the application flow can be passed back to the controller by using the common `navHandler` function. Building a menu this way enables total control over the application architecture. You will see later in this chapter how you can open a window and close the previous one, keeping an array of windows to handle the back button functionality.

Common elements

When deciding if you need to extract a UI element into its own module or make it a part of a common UI toolset, firstly consider if you will only use it once. Building a toolset of UI components for the application does not, and probably should not, be done in one step. Instead build the toolset one component at a time.

You can extract the code to generate any buttons or menus into their own modules. For example you could have a `/tools/button.js` file which has all the definitions of the applications buttons. You would not extract the code into a module for each button, as this would impact on the performance, as each module load uses resources.

Windows

Each window starts within its own module, utilising the common elements and controlled from the main controller. As you will see, the controller opens the next window, closes the previous window and updates the back handler. The following code shows how to create a new window and return the object back to the controller for handling.

It also shows how to use the window close event to clean up the memory and keep the application efficient.

```
/* A simple window module */

/* as the layout is used in various sections
require it as part of the module */

var layout = require('/ui/common/layout');
var common = require('/tools/common/commonTools');

function processBack() {
    common.navHandler({TYPE: 'BACK'});
}

function loadHomeWindow(inParam) {

    var newWin = Ti.UI.createWindow({
        // set the window properties here
    });

    // build a screen display area ..
    var newWinDisplay = Ti.UI.createView({
        top : 0,
        left : 0,
        right : 0,
        bottom : layout.params.menu.height
    })
    // Build the window contents ...

    // include the menu
    var mainMenu = require('/ui/common/menu');

    newWin.add(mainMenu);
```

```
        // Add the required close and back listeners
        ..

        newWin.addEventListener('android:back',
processBack);

        newWin.addEventListener('close', function(e){
            newWin = null;
        });

        return newWin;
    }
    exports.loadHomeWindow = loadHomeWindow;
```

Having created a simple window, you can see that by using a view to define the screen area minus the height of the menu gives you a specific area to build the remaining contents of the window and always keeping the main menu in view. Specifying the height in the global layout module enables an easy maintenance of that value system wide.

Adding a custom navigation bar follows exactly the same lines as adding the custom menu. This gives you total flexibility over the application architecture and flow.

In the best practices section, we say always add an event listener pointing to a separate function; however, on the close event, we use an inline function. The reason for this is that any variables we needed to clear down are instantly available within the scope of the function, instead of having to declare them as global to the module, and the event listener will be removed when the window is closed and nulled out.

The controller updated

Earlier in this chapter we saw a skeleton of the controller. As we progressed it was shown that windows needed to be closed and opened as required and that a method of maintaining the ability to go back was needed.

The following code shows how to open and close the windows within a function and set up the back button methodology:

```
/* The Application controller */

// We need the common tools
var common = require('/tools/common/commonTools');

// The required module variables
var openWin = null;
var prevWin = null;
var backArray = [];

function handleError(inParam){
    // Handle your error processing
}

// Process the back button
// Take the current backarray
// (The list of previously opened windows)
// Subtract two from its length as 1 would give
the
// current window. Not the previous window.
// and use this array element to load the previous
window
// Then remove the last window from the array and
the new
// window to load.
//
// You then fire the global event which
automatically
// loads the correct window as required.

function processBackButton(inParam) {
    var iPos = ((backArray.length - 2) < 0) ? 0 :
                (backArray.length - 2);
```

```

        if (iPos >= 0) {
            var tmpParams = backArray[iPos];

            // pop the back array out * 2 ..
            if(backArray.length > 1) {
                backArray.pop();
            }
            backArray.pop();
            // use the common fire event.
            common.navHandler(tmpParams);
        }
    }

// Process the window opening and closing
function windowHandler(inParam) {
    // add to the back array
    backArray.push(inParam.PARAMS);

    // Move the open window reference to the
previous window
    prevWin = openWin;

    openWin = inParam.WIN;

    openWin.open();

    if (prevWin) {
        prevWin.close();
        prevWin = null;
    }
}

function loadHomeScreen() {
    var lhsScreen = require('/ui/windows/home');

    var lhsWindow = lhsScreen.LoadHomeWindow();

    windowHandler({WIN : lhsWindow,
                    PARAMS : { TYPE: 'HOME' } });
}

function loadDetailScreen(inParam) {
    // require the detail screen and open it here
}

/* This function handles the application flow */

```

```

function appNavigation(inParam) {
    switch(inParam.TYPE) {
        case 'ERROR' :
            handleError(inParam);
            break;
        case 'HOME' :
            loadHomeScreen(inParam);
            break;
        case 'DETAIL' :
            loadDetailScreen(inParam);
            break;
        case 'BACK' :
            processBackButton(inParam);
            break;
        case 'STARTAPP' :
            startApp();
            break;
        default :
            break;
    }
}

/* Add the event listener */

function addAppNavListener() {
    Ti.App.addEventListener('APPNAV',
appNavigation);
}
function removeAppNavListener() {
    Ti.App.removeEventListener('APPNAV',
appNavigation);
}

/* This function is called from the app.js file
and is the only function to be exported from the
controller module */

function startApp() {
    addAppNavListener()

    // Code the application starting requirements
}

/* export the modules functions and variables */

exports.startApp = startApp;

```

The back processing function uses the array that is added every time a window is opened, and calls the common fire event function to pass control back to the controller and load the required window.

The `windowHandler` function that is called to open a window firstly pushes the parameters to the next element in the `backArray` variable. These parameters always have to have a `TYPE` object. It then moves the `openWin` variable to `prevWin` and opens anew window. If a previous window exists it then closes it. The reason for the order of processing is so that there is no flash as one window closes and another opens. If you closed the window before opening the next, then the user would see a brief view of the splash screen.

The `loadHomeScreen` function now requires the home screen module and processes the building of the window content, before calling the `windowHandler` function passing the required parameters. This works easily because we return the window object from the home screen module to the controller enabling the controller to handle opening and closing the window itself.

Fail gracefully

If you are doing anything which could fail, or make the application break, handle the error. It may seem obvious but there are a lot of applications that crash because they haven't got an Internet connection or the GPS is turned off.

Always gracefully handle application failure by building an error handling module, which will either show an alert or a message on a screen. Have some default handling, such as returning to the home screen on an error, and initializing the application to its starting point.

When checking for errors one of the common `unhandled` ones is a http request failing. Always use the `onerror` method and also use a try catch.

```
/* Handling HTTP request */

var common = require('/tools/common/commonTools');
function getMedata(inParam) {

    var xhrCall    =
    Ti.Network.createHTTPClient();

    xhrCall.onerror    = function()
    {
        common.navHandler({TYPE: 'ERROR',
            MESS :    'E0004',
            ERRM :    this.responseText
        });
    };

    try {
        xhrCall.open('GET', inParam.URL);

        xhrCall.send();
    }
    catch(e) {
        common.navHandler({TYPE: 'ERROR',
            MESS :    'E0003',
            ERRM :    e.message
        });
    };
}
```

```
}  
}
```

Remember that for error handling, event listeners and try-catch loops are your friends. There is nothing worse than putting an activity indicator on the screen, which just stays there because the GPRS signal died.

Using the try-catch loop does use extra resources and some developers will include it only if absolutely needed. A simple rule is that if the application fails and the processing stops, without handling that error use try-catch. It is really bad practice for the application to just fail.

Summary

As you have seen, it is possible to apply an MVC architecture to a Titanium CommonJS mobile application. This solution is only one way of developing your codebase. The methods and processes applied show how to architecture your application to work effectively, apply best practices, and manage memory effectively.

Using common tools, error handling, and separating the code out into relevant modules applies however you architecture the application. Following a few simple guidelines enables the codebase to be structured, understandable, and maintainable.

In the next chapter we explore cross-platform design techniques. This will enable you to develop applications for different devices.

Chapter 4. Cross-platform Design Methods

While we all eagerly await the final release of the new Alloy framework, which promises to make building layouts for Titanium apps much easier, let us in the meantime look at a few methods that you can be implementing in your apps now. It has to be said that there's no "one size fits all" solution to layouts—you should always choose those techniques that best fit your application, programming style, and target devices.

In this chapter, we'll be explaining a few methods of developing layouts for your applications that work across both iOS and Android. We will also be explaining some of the new layout properties that appear from Titanium SDK 2.0.

The difference between DPs and PXs

You may have noticed that many Titanium examples on the web will often use **px** (**pixel**) values, whereas others use **dp** (**density-independent pixels**) instead. There is a significant difference between the two values which is important to understand whenever putting together a cross-platform application.

- **px**: This unit corresponds to the actual pixels on the screen. In the case of the iPhone, for example, this value is 320 x 480 px.
- **dp**: This unit corresponds to the physical density of the screen. The scale of dips to pixels is not always proportionate between devices, and is definitely not proportionate between platforms. On iOS, 1 px equals 2 dp. This is how iOS retina

images work, by always being twice the density of the equivalent area in pixels.

On iOS, you can use either pixels or dps—they are treated in the same way. To implement both normal and retina images on the iPhone or iPad is a simple case of creating two separate images—one in the native pixel resolution for devices pre-iPhone 4, pre-iPad2, and another image that is double the size. These double-density images are renamed using the `@2x` suffix, which iOS automatically picks up and uses whenever it detects a retina screen. For example, if we have a logo that is positioned centre screen in an image view. Consider the following code:

```
var myImage = Ti.UI.createImageView({  
    image: 'images/my_logo.png',  
    width: 200,  
    height: 80  
});
```

Then all you need to do is create a secondary image called `my_logo@2x.png` with a width of 400 px and a height of 160 px, and place this image in the same directory as your original one with its dimensions of 200 px and 80 px.

On Android, unfortunately for the developer, there is a much greater fragmentation between device pixels and dp due to the vast number of devices and screen sizes available on the market. Generally, the various screen sizes and relative dp units are:

- `xlarge` screens are at least 960 x 720 dp
- `large` screens are at least 640 x 480 dp
- `normal` screens are at least 470 x 320 dp
- `small` screens are at least 426 x 320 dp

Note

Need more information on Android device resolutions?

Visit

http://developer.android.com/guide/practices/screens_support.html to view a list of sizes and screenshots that better illustrate Android density independence.

On Android, the aim is to achieve what is called "density independence", which means that your application should render at roughly the same physical size regardless of the user's actual screen size. It's for this reason that on Android you should always use `dp` values wherever possible. The only caveat with doing so is that image views and other elements that use rasterised bitmaps will scale up or down, which can result in blurred or distorted images. In these cases, you may need to provide images in more than one size and choose the appropriate image in code depending on the device resolution.

Understanding Ti.UI.SIZE and Ti.UI.FILL

From Titanium SDK 2.0, a couple of new properties were introduced called `Ti.UI.SIZE` and `Ti.UI.FILL`. The aim of these properties is to remove the complexity in developing interfaces and specifically, the complexity involved in sizing views that contain n number of children.

Ti.UI.SIZE

The new `SIZE` property was introduced in order to reduce programmer dependence on calculating the height of a parent view depending on the child views added to it. The following example illustrates how we may have calculated the height of the parent view (called `parentView`) by manually determining the total height of the child views inside it):

```
var parentView = Ti.UI.createView({
    width: 200
});

function addChildren(_number) {
    for(var i = 0; i < _number; i++){
        var childView = Ti.UI.createView({
            width: 80,
            height: 80,
            top: (i * 100)
        });

        parentView.add(childView);
    }

    var newHeight = _number * 100;
    return newHeight;
}

parentView.height = addChildren(3);
```

```
alert( parentView.height ); //alerts "300";
```

We can now do the same thing without having to manually calculate the height, just by using the `Ti.UI.SIZE` property. The parent view in this case will automatically expand in height to match the height and position of all UI components that reside within it.

```
var parentView = Ti.UI.createView({
    width: 200,
    height: Ti.UI.SIZE
});

function addChildren(_number) {
    for(var i = 0; i < _number; i++){
        var childView = Ti.UI.createView({
            width: 80,
            height: 80,
            top: (i * 100)
        });

        parentView.add(childView);
    }
}

addChildren(3);
```

Ti.UI.FILL

The `FILL` property replaces the old `100%` sizing value. Any component that uses the value will automatically expand in size to its parent. The following example illustrates how we can use the `FILL` property to make an `ImageView` fit within the dimensions of the `parentView` container it resides within:

```
var parentView = Ti.UI.createView({
    width: 200,
    height: 300
});

var imgView = Ti.UI.createImageView({
    width: Ti.UI.FILL,
```

```
        height: Ti.UI.FILL,  
        image: 'images/myphoto_200x300.png'  
    });  
  
    parentView.add(imgView);
```

The `FILL` property is especially useful when creating layouts that are meant to be cross-compatible between iOS and Android—it means that you don't have to worry about whether to use dips or pixels to calculate the height of a child container, and that your content should always resize to match its parent.

Choosing a cross-platform layout method

There are a number of methods for creating applications in Titanium that are able to share logic, but have customised layouts. Before we begin, here are some of the general UI principles that are worth sticking to:

- Don't try to make Android look like iOS and vice-versa. For example, Android users are used to tab bars appearing at the top of the screen, and while it's easy enough to make these bottom-aligned like an iOS tab bar, it's not really a best practice.
- Use the in-built components wherever possible. Using default UI components (such as Option Dialogs, or Android Menu Bars) makes your applications look, feel, and act like their native counterparts. The Titanium API already takes care to ensure that the native component is called for the particular device it's currently running on, so use what's already available!
- Employ the KISS principle (Keep It Simple, Stupid!). Mobile devices, by their very definition, have limited screen real estate. If you have layers and layers of nested components you should probably reconsider your application's design.

In this next section, we'll go through a number of different layout methodologies and lay down the pros and cons of each, before providing some sample implementations in the form of a CommonJS module that you can use in your own projects.

Using `Ti.Platform.osname` statements

Whichever way you end up doing your cross-platform layouts, you'll eventually need to use the `Ti.Platform.osname` property at some

point. With it, you can simply define views, objects, and other data in scope for a particular mobile platform, such as in the following example that defines the `customView` component as a view for the iPhone and as an `ImageView` on Android:

```
var customView;

if (Ti.Platform.osname === "iphone"){
  //define view for the iphone only
  customView = Ti.UI.createView();

} else if (Ti.Platform.osname === "android"){
  //include the other way
  customView = Ti.UI.createImageView();
}
```

However, you need to be mindful of how often and where you use these kind of statements, or you'll end up with JavaScript files that are nothing but spaghetti code, filled with multiple instances of `if`. They are best used in conjunction with CommonJS modules for specific, custom UI components, or for including platform-specific files.

Developing with separate UI files

The easiest method of development is one whereby you have no chance of mixing up iOS- and Android-specific code., and is easily implemented by creating separate UI files for each platform. This method means that for each CommonJS module, window, or view you create for iOS you have an equivalent module, window, or view in a separate file for Android.

Unfortunately, this method means twice the work and twice the amount of code to manage and is fine for small projects or short-term solutions, but can get particularly difficult to manage in the long term.

Importing component styles from external files

This is a common layout method that has been utilised by numerous developers, using everything from XML files, to JSON components, to JSS (CSS-like) stylesheets. It requires reading from the external file (held somewhere within the [Resources](#) directory), parsing it into a JavaScript object, and then applying those important styles directly to a UI component.

While a good idea in principle, in practice this method has a number of issues depending on what file storage mechanism you choose. Consider the following options:

- XML is easier to read, but parsing it can be very slow in Titanium (particularly on Android). It takes twice the amount of time on an average to parse an XML file than it does a JSON file with the same content. You'll also have to manually convert all the XML elements and attributes to JavaScript equivalents—you can use a library to do this automatically but be prepared for application failures if you happen to have any XML syntax problems!
- JSON files are a better option than XML—slightly more obscure to read with the naked eye, but performance is much better when reading in real time, and they much more closely resemble proper JavaScript objects than XML does. However, you still have to manually read these at runtime and depending on the length of your JSON file, this could mean precious wasted seconds of application drawing time.
- While a great idea, JSS seems to have lost official support of late and is prone to failures and difficulties in implementation (such as you need to clean your projects before building every time your JSS files have changed). JSS files are essentially the in-built implementation of using JSON for your layouts.

A much better alternative for defining real-time layouts is by using a combination of native JavaScript objects in conjunction with some CommonJS modules.

Building a JavaScript stylesheet CommonJS module

To build our custom module, we'll first need to create a number of files and folders underneath the `Resource` directory of your project. Firstly, create a new folder called `stylesheets`, and within it, create separate folders called `common`, `android`, and `iphone`.

You should end up with a basic directory structure that looks like this:

- `Resources`
 - `stylesheets`
 - `common`
 - `android`
 - `iphone`

Now under each `common` directory, create a file called `android.js` and another file called `iphone.js`. Each of these files will hold the common JavaScript objects that you will use for styling over and over again. We're going to create ours with a default button style in each one that all the buttons within our application will inherit. Copy the following code into your `android.js` file under the `common` folder:

```
/*
  Android Specific 'common styles' module
*/
StyleSheet = function(){

  this.buttonStyle = {
    width: '100dp',
    height: '50dp',
```

```

        backgroundColor: '#f3f3f3',
        color: '#000000',
        font: {
            fontSize: 18
        },
        title: 'MY BUTTON'
    };

    //return all these objects
    return this;
};

exports.Stylesheet = Stylesheet;

```

When you're done with the Android file, copy the following code into your `iphone.js` file under the `common` folder:

```

/*
 iPhone Specific 'common styles' module
 */
Stylesheet = function(){

    this.buttonStyle = {
        width: 100,
        height: 30,
        color: '#000000',
        font: {
            fontSize: 14,
            fontFamily: 'Helvetica'
        },
        title: 'MY BUTTON'
    };

    //return all these objects
    return this;
};

exports.Stylesheet = Stylesheet;

```

Once that's done we need to create a CommonJS module that will call and import all of these styles depending on the platform, returning to us just one single JavaScript object. Create a new file called `Stylesheet.js` (we're just doing this in the root directory in this

example, but it should go into your tool's folder if you are using the MVC pattern from the previous chapter) and copy into it the following code:

```
/*
  Stylesheet.js CommonJS module
*/
StyleSheet = function(){

    this.stylesheet = {};

};

StyleSheet.prototype.init = function(_name){

    //include the common styles
    var StylesheetCommon =
    require('/stylesheets/common/' +
        Ti.Platform.osname.toLowerCase());
    this.stylesheet = new
    StylesheetCommon.Stylesheet();

    //include file specific styles
    if(_name !== undefined && _name !== null &&
    _name !== ''){
        var StylesheetDevice =
        require('/stylesheets/' +
            Ti.Platform.osname.toLowerCase() + '/' +
            _name);

        this.stylesheet = this.merge(new
            StylesheetDevice.Stylesheet(
                Ti.Platform.osname.toLowerCase()));
    }

    //return the final stylesheet object
    return this.stylesheet;
};

/*
  Merge a new stylesheet object with the existing
  one
*/
StyleSheet.prototype.merge = function(obj2){
```

```

var obj1 = this.stylesheet;
var obj3 = {}; //we'll merge into this

for (var attrname in obj1) {
    obj3[attrname] = obj1[attrname];
}

for (var attrname in obj2) {
    obj3[attrname] = obj2[attrname];
}

//return the newly merged object
return obj3;
};

exports.StyleSheet = StyleSheet;

```

What this code does is pull in the common styles that appear in your `/stylesheets/common/<platform>` file (depending on whether you're running Android or iOS) and assign all its properties to an object called `stylesheet`. The `merge` function is there to provide additional styling support for specific files. If, for example, you have a common stylesheet file with button styles in it, but want those button styles overwritten for a screen called `MySecondWindow`, the `stylesheet` module will pull in all the styles from `common` first, then merge and replace any existing styles with the equivalent style object from the `MySecondWindow` stylesheet. This essentially makes the system work in the same way as CSS—whereby additional stylesheets act in a cascading manner, overwriting any previous declarations in stylesheets that came before them.

Using the StyleSheet CommonsJS module in your app

Now that we have created the `StyleSheet` CommonsJS and we've created the equivalent Android and iPhone common stylesheet files, it's time to pull them into our test application. In your UI file (for example, `app.js` or `ui/handheld/ApplicationWindow.js`, if you're

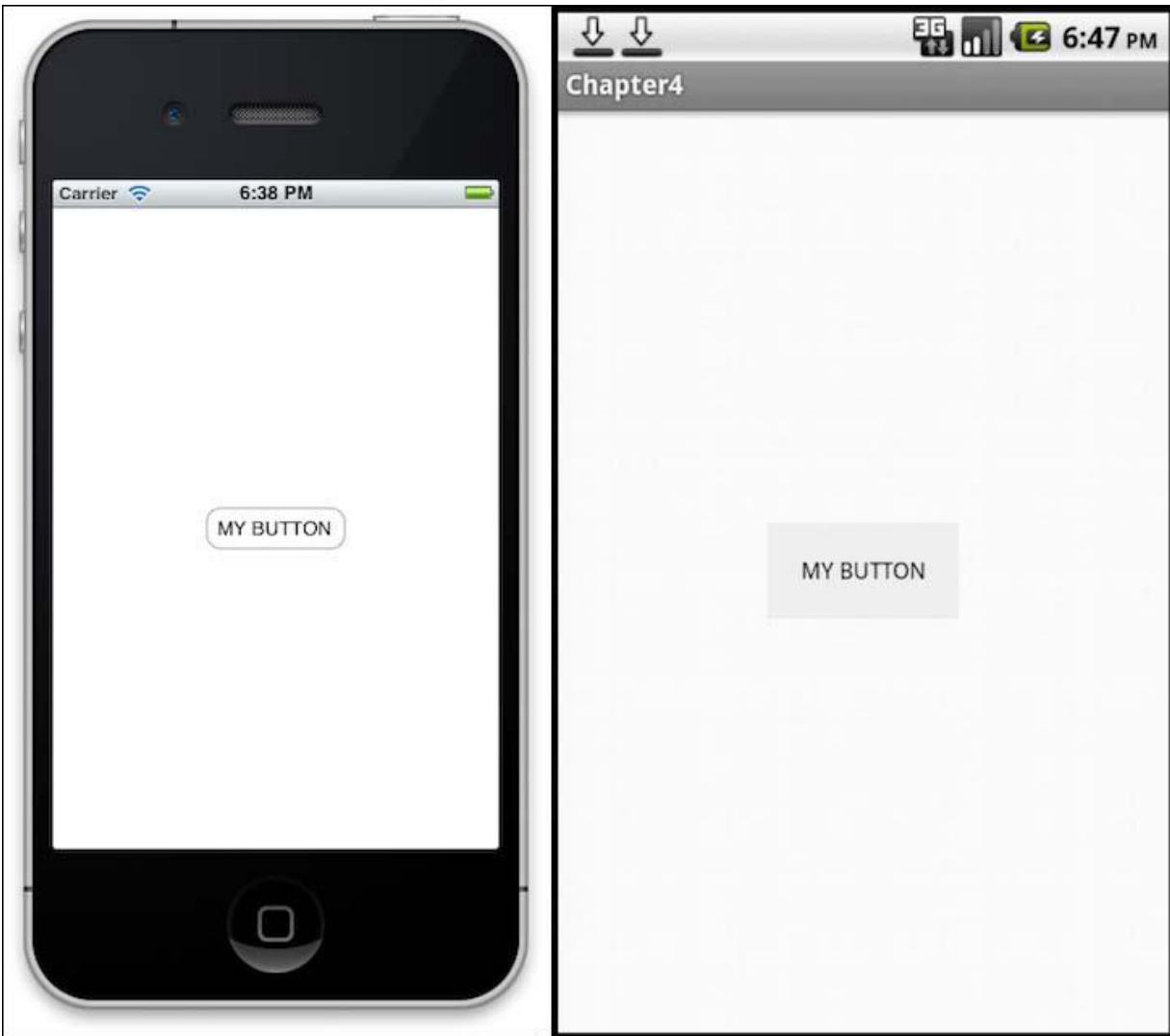
using one of the Titanium template projects), you'll need to import the `Stylesheet` module by requiring it, like so:

```
//import the stylesheet file
var Styles = require('/Stylesheet');
var styles = new Styles.Stylesheet();
this.stylesheet = styles.init();

var button =
Ti.UI.createButton(this.stylesheet.buttonStyle);

//assume there's already a window created called
//'self' based on the Titanium Template Projects
self.add(button);
```

If you've created all the files and folders correctly, you should be able to run the application on both the iPhone and Android emulators and see the button styled slightly differently for each platform, as in the following screenshots:



Extending the Stylesheet module for page-specific styles

Now that we have a stylesheet for both iOS and Android mobile devices that can handle styling on a global level, we need to extend our stylesheet system to handle page-specific items. Create a JavaScript file called `ApplicationWindow.js` in each of the `stylesheet/android` and `stylesheet/iphone` folders that you created earlier. Then, paste in the following code for Android:

```
/*  
  Android Specific Styles for ApplicationWindow
```

```

*/
StyleSheet = function(){
    //return all these objects
    return this;
};

exports.StyleSheet = StyleSheet;

```

And do the same for iPhone. You will notice that the Android code doesn't replace the existing button styling in the common file, while the following iPhone code does:

```

/*
    iPhone Specific Styles for ApplicationWindow
*/
StyleSheet = function(){

    this.buttonStyle = {
        width: 180,
        height: 40,
        style:
Titanium.UI.iPhone.SystemButtonStyle.DONE,
        color: 'Red',
        font: {
            fontSize: 12,
            fontFamily: 'Helvetica'
        },
        title: 'Extended Style!'
    };

    //return all these objects
    return this;
};

exports.StyleSheet = StyleSheet;

```

All that's left to do now is to edit your `ApplicationWindow.js` file (where you call and use the `StyleSheet` module) to pull in the `ApplicationWindow.js` stylesheet files as well as the common ones.

```

//import the stylesheet file
var Styles = require('/StyleSheet');
var styles = new Styles.StyleSheet();
this.stylesheet =

```

```
styles.init('ApplicationWindow');

var button =
Ti.UI.createButton(this.stylesheet.buttonStyle);
```

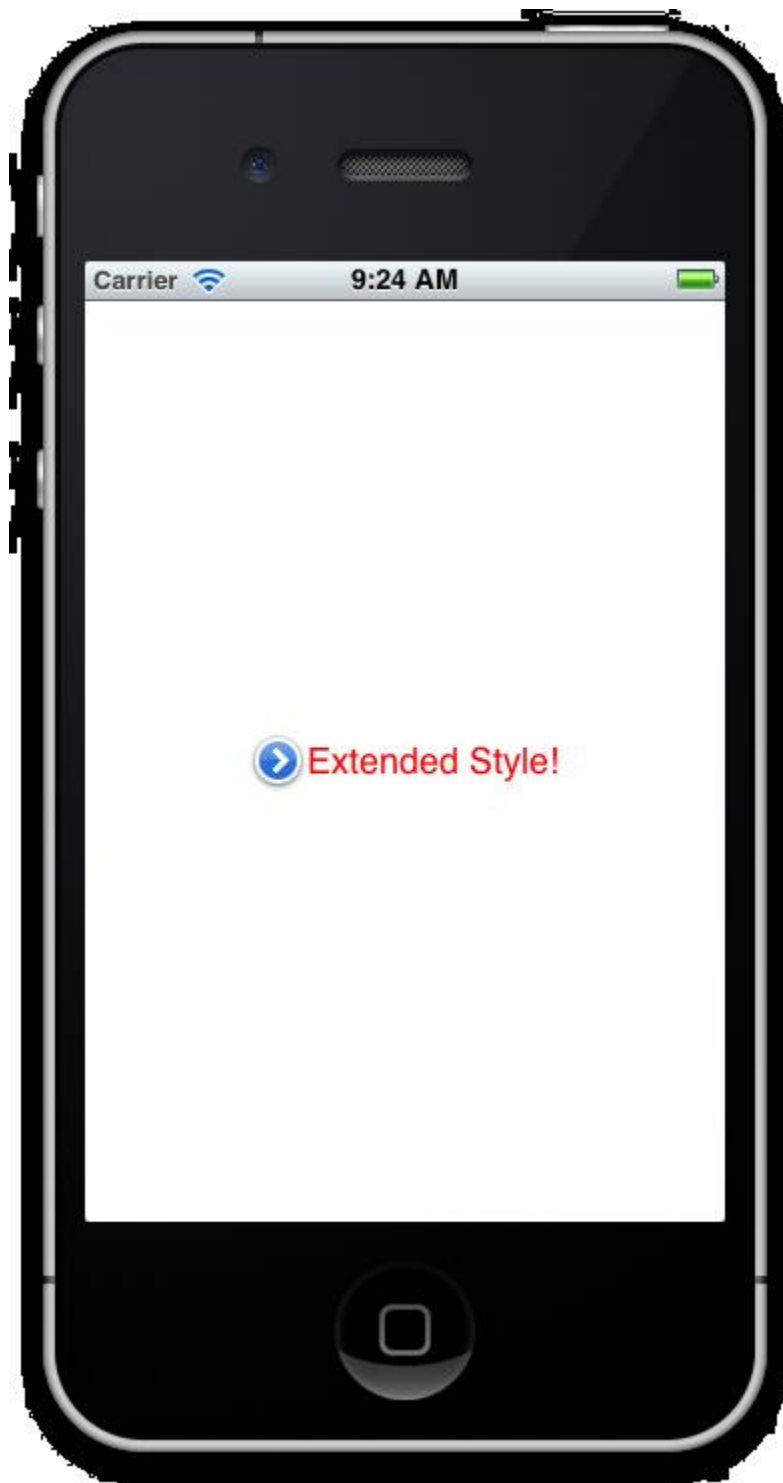
Run the project in the Android emulator again and you'll notice that nothing has changed. Run it in the iPhone simulator, however, and what you'll end up with is a new button style specific to that screen, and specific to iOS.

You can now use this `stylesheet` module and extend it in the same manner for additional device types (such as the iPad or the extra height of the iPhone 5) while retaining total control over the styling of your objects in both platforms.

If you have trouble compiling on Android due to using any of the `Ti.UI.iPhone` methods or properties, just add a quick single line check into your stylesheet files like so:

```
this.buttonStyle = {
    width: 180,
    height: 40,
    style: (Titanium.Platform.osname !==
'android') ?

Titanium.UI.iPhone.SystemButtonStyle.DONE : null,
    color: 'Red'
};
```



Summary

The layout methods described within this chapter should help you to develop applications that make the best of native component styling, while also allowing you to extend and customize styles dependant on platform and device type—all from within the same codebase.

Remember, there is no "magic" solution to managing layouts—you should work with the combination of these techniques that best fits your application and programming style.

In the next chapter, we are going to look at a new alternative to traditional SQLite relational databases, in the form of JSONDB, a JavaScript-based flat file database that is both easy to use and just as fast as SQLite.

Chapter 5. Using JSONDB as an Alternative to SQLite

Modern databases are changing—where we once relied almost solely on relational databases (MySQL, Oracle, and so on), much of the modern web and mobile data capture is now performed using **NoSQL (Not only SQL)** databases. Relational databases on iOS and Android alike are generated using SQLite—an embedded database engine with a small footprint that, for all its benefits, also has a number of problems, as follows:

- Table alterations (or preinstalled databases) usually mean having to delete and reinstall applications during development, and schema modifications after your apps have been published.
- Creating and managing SQL queries is both a pain and prone to failure – you have to consider all the normal relational SQL database problems such as query injection, query/indexing speed, and more.
- Large datasets require knowledge of indexing and foreign key relationships.
- Caching and synchronizing remote data means writing, pulling apart various JSON objects, and manually managing them within the database tables.

In this chapter, we'll be working through the basic principles behind NoSQL databases and demonstrating how you can implement a NoSQL solution in your apps using the JSONDB module.

Installing the JSONDB module

First off, you will need to download and install the JSONDB module for iOS and Android from GitHub at:

<https://github.com/irlgaming/jsondb-public>. Should you wish to

manually download just the JavaScript file required and none of the example code, you'll find it located in the `assets` directory.

Setting up JSONDB

Now, let's create a new application—open up Titanium Studio and select the **Single Window Application** template. Give your project a name and an identifier—we've called ours `NewsReader` and given it an ID of `com.packt.newsreader`. Once that is done, copy the `com.irlgaming.jsondb.js` CommonJS file into your `Resources` directory and reference it as normal, for example, `require('com.irlgaming.jsondb')`.

Once you've done that, open the `/ui/common` folder and create a new JavaScript file called `jsonDatabase.js`. We are going to create a CommonJS module inside this file, which we'll use to perform all our JSONDB database queries and storage. Create a basic CommonJS module structure and include the JSONDB module within it like so—the constructor function will create a collection called `news` and assign it to the `JSONDatabase` module:

```
//require the jsondb module
var jsondb = require('com.irlgaming.jsondb');

function JSONDatabase(){
    this.collection = jsondb.factory("news");
};

exports.JSONDatabase = JSONDatabase;
```

Creating a JSONDB CommonJS module

The equivalent of a data table in JSONDB is called a **collection**. Let's create a new function within our CommonJS module that accepts a JavaScript object—this object will be a `news` record that comes back from our remote data and will consist of a title, description, URL, and date field.

```
//creates a new record in the collection or opens
an existing //one
JSONDatabase.prototype.createCollectionRecord =
function(_title, _description, _date, _url){

    this.collection.save({
        title: _title,
        description: _description,
        link: _url,
        createdAt: _date
    });
    this.collection.commit();
};
```

We can also create the collection by taking the data straight from the remote server, so let's create a second function that does that now:

```
//creates a new record in the collection from an
existing //object
JSONDatabase.prototype.createCollection =
function(_record){

    this.collection.save(_record);

    this.collection.commit();
};
```

What this function does is it loads the existing `news` collection, creates a new record, and then commits that record to storage. Let's jump back into the `/ui/common/FirstView.js` file and create a basic interface that loads a news feed from TechCrunch.

Copy the following code into your `FirstView.js` file. This code will just layout a basic `TableView` that takes up the top 90 percent of the screen's real estate, a **new** button that will add an item to our local collection (resetting the `TableView` after doing so), and a slider that sets the cache timeout in minutes. It will also include a function that allows us to create a custom `TableViewCell` with some styling that we'll use later on to add data to our `TableView` component.

```

//FirstView Component Constructor
function FirstView() {
    //create object instance, a parasitic subclass
    of
        //Observable
        var self = Ti.UI.createView();

    //create a tableRow in a specific style with a
    given
        //title and description and return it
        function createTableRow(_title, _description){
            var row = Ti.UI.createTableViewRow({
                height: 80,
                hasChild: true,
                rightImage: '/images/ipad-arrow.png',
                backgroundImage: '/images/list-item-bg.png'
            });
            var lblTitle = Ti.UI.createLabel({
                left: 10,
                top: 10,
                width: Ti.UI.FILL,
                height: 21,
                font: {
                    fontSize: 16,
                    fontWeight: 'bold'
                },
                text: _title,
                textAlign: 'left',
                color: '#094B7B',
                shadowColor: 'white',
                shadowOffset: {
                    x: 0,
                    y: 1
                }
            });
            row.add(lblTitle);

            var lblDescription = Ti.UI.createLabel({
                left: 10,
                top: 33,
                width: Ti.UI.FILL,
                height: 36,
                font: {
                    fontSize : 13
                },
                text: _description,

```

```

        textAlign: 'left',
        color: 'gray',
        shadowColor: 'white',
        shadowOffset: {
            x: 0, y: 1
        }
    });
    row.add(lblDescription);

    //return the row
    return row;
};

//create news table - our feed is going to
appear here
var tblNews = Ti.UI.createTableView({
    style: Ti.UI.iPhone.TableViewStyle.PLAIN,
    separatorStyle:

Titanium.UI.iPhone.TableViewSeparatorStyle.NONE,
    width: Ti.UI.FILL,
    height: '88%',
    top: 0
});
self.add(tblNews);

//this button will create a new JSONDB record
var btnNewRecord = Ti.UI.createButton({
    left: 5,
    bottom: 5,
    width: 140,
    height: 47,
    title: 'New Record',
    backgroundImage: '/images/ipad-button-
blue.png',
    style:
Titanium.UI.iPhone.SystemButtonStyle.PLAIN,
    color: '#fff',
    shadowColor: '#000',
    shadowOffset: {x:0, y:1},
    font: {fontWeight:'bold', fontSize:18}
});

self.add(btnNewRecord);

//shows the cache timeout (or no cache)
var lblCache = Ti.UI.createLabel({
    right: 8,
    textAlign: 'right',

```

```

        bottom: 40,
        text: 'Cache timeout: 2 mins',
        font: {fontWeight:'bold', fontSize:11}
    });
    self.add(lblCache);
    //allows the user to set the cache timeout in
minutes
    var sliderCacheMinutes =
Titanium.UI.createSlider({
        right:5,
        bottom: 5,
        width: '50%',
        height: 24,
        min:0,
        max:5,
        value:2,
        leftTrackImage:'/images/ipad-slider-fill.png',
        rightTrackImage:'/images/ipad-slider-
track.png',
        thumbImage:'/images/ipad-slider-handle.png',
        highlightedThumbImage:'/images/ipad-slider-
handle.png'
    });

    //update the label when the cache slider value
changes
    sliderCacheMinutes.addEventListener('change',
function(e) {
        lblCache.text = ('Cache: ' +
Math.round(e.source.value)
                        + ' mins');
        if(Math.round(e.source.value) === 0){
            lblCache.text = 'Cache: OFF';
        }
    });
    self.add(sliderCacheMinutes);

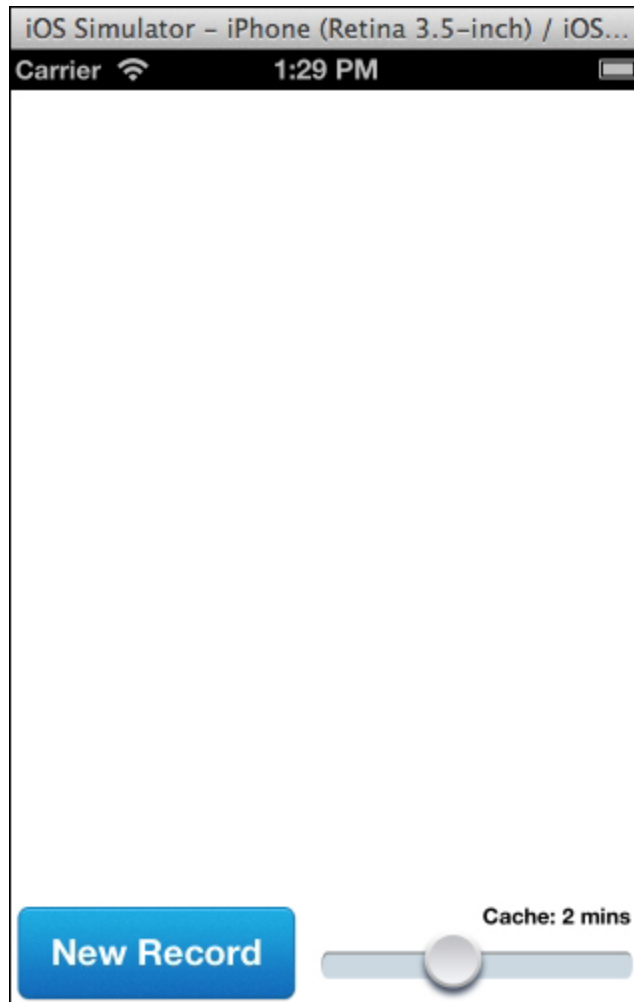
    //finally,
    return self;
}

module.exports = FirstView;

```

Running the code in your simulator will result in a screen that has a **new** button in the bottom-left corner, a slider and a label in the bottom-right corner, and an empty TableView taking up the

remainder of the space, similar to the following screenshot. By default, we're assuming the cache timeout is 2 minutes.



We're going to use this basic example to show how you can mix remote and local data with JSONDB and utilize it as a caching mechanism at the same time. First of all, let's fetch some remote data and initialize it inside our JSONDB collection, before finally refreshing the TableView to show it.

Note

We have shortened the feed URL for easier reading. The full feed address is: <http://pipes.yahoo.com/pipes/pipe.run?>

[_id=b0b46551d3ba3f82cadfd88dfa8917de&_render=json.](#)

First of all let's create a basic `getNewsData()` function in `FirstView.js` that will load the remote data.

```
/*
 * Gets the remote JSON news data from our feed url
 */
function getNewsData(_url){
    var xhr = Ti.Network.createHTTPClient();
    xhr.open('GET', 'http://bit.ly/YHPr6V');

    xhr.onload = function(){
        var jsonResponse =
JSON.parse(this.responseText);
        Ti.API.info(jsonResponse);
    };

    //print out an api error to the console log
    xhr.onerror = function(e){
        Ti.API.error(e.message);
    };

    xhr.send();
};
```

Then before the end of your CommonJS module where you return "self", execute the function, and run it in the simulator:

```
...

    //execute the news data function
    getNewsData();

    return self;
};
```

If all has gone well you should be rewarded with a logged information entry in your console of our JSON news feed. The data should be in the following tree structure, though the actual news entries on yours may differ:

Key	Type	Value
▼ TextImport	Object	Object
count	Number	41
▼ value	Object	Object
title	String	NewsReader1 JSONDB Feed
description	String	Pipes Output
link	String	http://pipes.yahoo.com/pipes/pipe.info?
pubDate	String	Fri, 07 Dec 2012 07:33:25 +0000
generator	String	http://pipes.yahoo.com/pipes/
callback	String	
▼ items	Array	Array
▼ Item[0]	Object	Object
title	String	VIDEO: Tencent and social media in China
description	String	Rory Cellan-Jones visits the studios of Tencent in China, one of the country's
link	String	http://www.bbc.co.uk/news/technology-20629433#sa-
▶ guid	Object	Object
pubDate	String	Fri, 07 Dec 2012 04:28:26 GMT
▶ media:thumbnail	Array	Array
▶ y:published	Object	Object
▶ y:id	Object	Object
y:title	String	VIDEO: Tencent and social media in China
▶ Item[1]	Object	Object

Note

There are plenty of tools both online and available for the desktop that can help you make visual sense of large JSON datasets. The one pictured is Cocoa JSON Editor, which is available from the Mac App Store for around \$4.00 (£2.50).

What we are going to do now is insert a reference to the JSONDB CommonJS module that we created earlier, and we'll put this at the top of our `FirstView.js` file so we have to only create the reference object once.

```
var JSONDatabase =
require('/ui/common/jsonDatabase');
var jsonDB = new JSONDatabase.JSONDatabase();
```

Now let's extend the `xhr.onload()` function so that it loops through the remote news data and inserts each new record into our `jsonDB` news collection:

```

xhr.onload = function(){
    var jsonResponse =
JSON.parse(this.responseText);
    Ti.API.info(jsonResponse);

    var newsArray = jsonResponse.value.items;
    for(var i = 0; i < newsArray.length; i++){

        //takes a title, description, date and url
        jsonDB.createCollectionRecord(
            newsArray[i].title,
            newsArray[i].description,
            newsArray[i].pubDate,
            newsArray[i].link
        );

    } //end for loop

};

```

Now let's do the simplest of the JSONDB functions and fetch all the data that we've retrieved from our remote feed, and then create a `TableViewCell` for each, and assign those rows to our `TableView`.

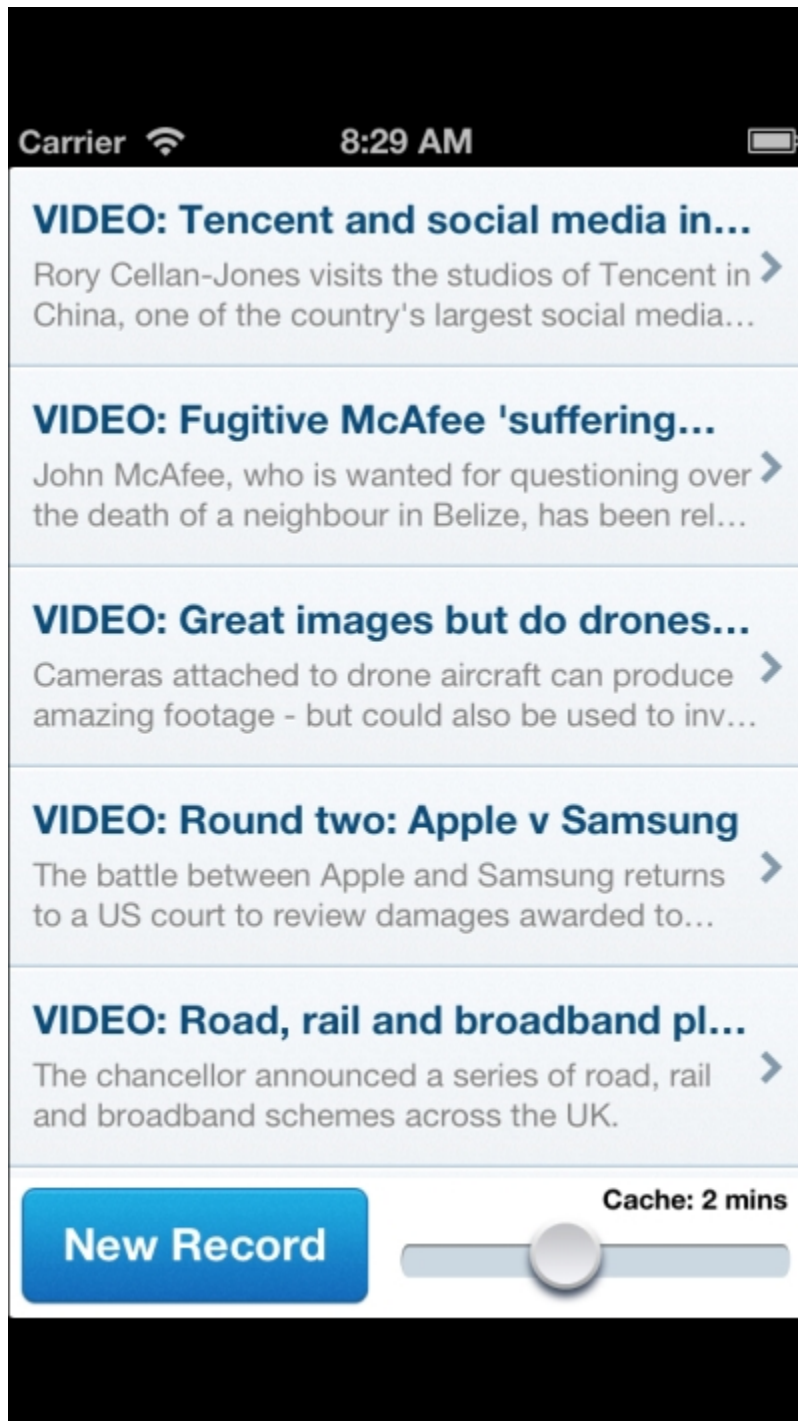
```

var rows = [];
var jsonCollection = jsonDB.collection.find();

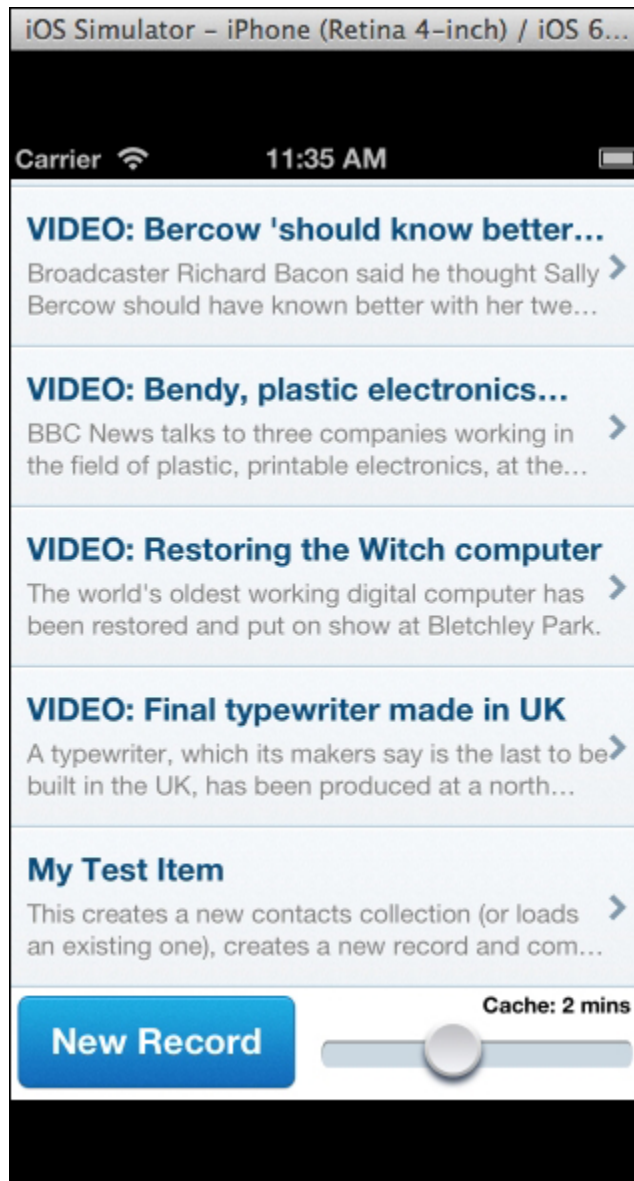
for(var i = 0; i < jsonCollection.length; i++){
    var row =
createTableRow(jsonCollection[i].title,
jsonCollection[i].description);
    rows.push(row);
}
//set tblNews to the new data from the feed
tblNews.setData(rows);

```

Try running the application in the simulator now. If all has gone well, you should see that the JSONDB collection we created has returned us an array of data that we've used to fill our `TableView`.



Inserting a new record into the collection is no harder than the creation of rows using the data coming back from our remote source. Here we have extended the **New Record** button to insert a test record, and then used the find commands of JSONDB to retrieve it and reset the table data:



Let's now refactor our code to implement the cache, so that new data is requested every 60 seconds but is only fetched from our server if the cache timeout has expired. If it hasn't, we'll just reuse the local JSONDB data. First, let's create a new function that separates out our data fetching from the collection and `TableViewCell` creation:

```
//function that creates the table rows and assigns  
to the //tableview  
function setTableRows(){  
    var rows = [];
```

```

        var jsonCollection = jsonDB.collection.find();

        for(var i = 0; i < jsonCollection.length;
i++){
            var row =
createTableRow(jsonCollection[i].title,
jsonCollection[i].description);
            rows.push(row);
        }

        //set tblNews to the new data from the feed
tblNews.setData(rows);
    }

```

Now let's refactor the data request function and create an interval that works off the slider "cache" time value:

```

/*
 * Gets the remote JSON news data from our feed url
 */
var lastCacheTime = null;

function getNewsData(_url){
    if(lastCacheTime !== null){
        var now = new Date();
        var diffMilliseconds = (now -
lastCacheTime);
        var diffMinutes = diffMilliseconds / 60000;
                                //60,000 ms in a minute
    }

    if(lastCacheTime === null || (diffMinutes >
        sliderCacheMinutes.value)) {
        Ti.API.info('Fetching fresh data...');
        var xhr = Ti.Network.createHTTPClient();
        xhr.open('GET', 'http://bit.ly/YHPr6V');

        xhr.onload = function(){
            var jsonResponse =
JSON.parse(this.responseText);
            jsonDB.collection.clear();

            // Ti.API.info(jsonResponse);

            var newsArray =
jsonResponse.value.items;

```

```

        for(var i = 0; i < newsArray.length;
i++){
        //takes a title, description, date
and url
        jsonDB.createCollectionRecord(
            newsArray[i].title,
            newsArray[i].description,
            newsArray[i].pubDate,
            newsArray[i].link);
        }

        setTableRows();
        lastCacheTime = new Date();
    };

    xhr.onerror = function(e){
        Ti.API.error(e.message);
    };

    xhr.send();

}
else {
    Ti.API.info('Using the JSONDB cache...');
    setTableRows();
}
};

//sets the interval for our data fetching
setInterval(function(e){
    getNewsData();
}, 60000);

```

Try running the code now—you should be able to watch the console log and see that new data is being requested every minute, but that it's only fetching data when the cache timeout expires.

`jsonDB.collection.clear()` is removing the records from our JSONDB collection before the fresh ones are inserted.

Titanium iPhone Simulator - NewsReader1 [Titanium iOS Simulator] iOS Simulator

```
[INFO] Application started
[INFO] NewsReader1/1.0 (2.1.3.GA.15997d0)
[INFO] Fetching fresh data...
[INFO] Using the JSONDB cache...
[INFO] Using the JSONDB cache...
[INFO] Fetching fresh data...
[INFO] Using the JSONDB cache...
[INFO] Using the JSONDB cache...
[INFO] Fetching fresh data...
[INFO] Using the JSONDB cache...
```

Using JSONDB queries

You can search, filter, and sort data saved in your collections using JSONDB queries. You can see the following list of examples based on our [news](#) collection and sample application, with comments for each:

```
function runSampleQueries(){
    //finds a set of collection records where
description
    //contains "drones"
    //and then sorting and limiting (to 10
records) the
    //result set
    var q1 = jsonDB.collection.find(
        {description:/drones/i},
        {$limit:10,
        $sort:{createdDate:1}
        });

    Ti.API.info(q1);
    Ti.API.info(q1.length);

    //updates a collection record that matches the
title and
    ///sets a new title
    var q2 = jsonDB.collection.update({title:
'VIDEO: Road,
        rail and broadband plans'},
        {$set:{title: 'VIDEO: My Updated
Title!'}});
    Ti.API.info(q2);

    //removes items where createdDate matches the
given date
    var q3 = jsonDB.collection.remove(
        {$exists:{createdDate:"Wed, 05
Dec 2012
        13:50:11
GMT"}}
    );
}
```

```
Ti.API.info(q3);  
}
```

Summary

In this chapter, we have shown how you can use JSONDB to not only cache remote data, but to also query it and create, update, and delete records just like you would with any other regular relational database source, such as SQLite. As JSONDB is JavaScript object-based, there's no need to worry about creating and maintaining SQL queries or database indexes!

Index

A

- addEventListener / [CommonJS best practices](#)
- android directory / [Android](#)
- app.js file / [The bootstrap](#)
- application / [The application](#)
- application architecture
 - CommonJS / [CommonJS](#)
- APPNAV event / [The controller](#)

B

- bootstrap / [The bootstrap](#)

C

- callback / [Callbacks](#)
- close event / [Managing memory](#)
- code formatting / [Code formatting](#)
- code structure
 - curly bracket, using / [Use semicolons and curly brackets](#)
 - semicolon, using / [Use semicolons and curly brackets](#)
 - eval(), avoiding / [eval is evil](#)
 - curly bracket position / [The curly bracket position](#)
 - non-explicit comparisons / [Not all operators are created equal](#)
- code validation / [Code validation](#)
- collection / [Creating a JSONDB CommonJS module](#)

- common elements, UI / [Common elements](#)
- CommonJS
 - about / [Understanding scope and the global object, CommonJS](#)
 - modules / [CommonJS modules](#)
 - best practices / [CommonJS best practices](#)
 - require / [CommonJS require](#)
- control directory / [Control](#)
- controller
 - about / [The controller](#)
 - updating / [The controller updated](#)
- cross-platform layout method
 - about / [Choosing a cross-platform layout method](#)
 - Ti.Platform.osname statements, using / [Using Ti.Platform.osname statements](#)
 - separate UI files, developing / [Developing with separate UI files](#)
 - component styles, importing from external files / [Importing component styles from external files](#)
 - JavaScript stylesheet CommonJS module, building / [Building a JavaScript stylesheet CommonJS module](#)
 - Stylesheet CommonJS module, using / [Using the Stylesheet CommonJS module in your app](#)
 - Stylesheet CommonJS module, extending / [Extending the Stylesheet module for page-specific styles](#)
- custom menu, UI / [The custom menu](#)

D

- directory structure

- about / [Directory structure](#)
- android / [Android](#)
- iphone / [iPhone and iPad](#)
- iPad / [iPhone and iPad](#)
- control / [Control](#)
- model / [Model](#)
- ui / [UI](#)
- UI / [UI](#)
- tools / [Tools](#)
- dp / [The difference between DPs and PXs](#)

E

- e /ui/common folder / [Setting up JSONDB](#)

F

- factories / [CommonJS modules](#)
- FILL property / [Ti.UI.FILL](#)
- forEach loop
 - using / [Using forEach instead of for loop](#)
- formatting
 - constructors / [Formatting, commenting, and naming conventions](#)
 - constants / [Formatting, commenting, and naming conventions](#)
 - function and method names / [Formatting, commenting, and naming conventions](#)
 - variable declarations / [Formatting, commenting, and naming conventions](#)
- for statements

- variables, declaring outside / [Declaring variables outside the for statements](#)
- framework / [Frameworks](#)

G

- getNewsData() function / [Creating a JSONDB CommonJS module](#)
- GitHub / [Installing the JSONDB module](#)
- global object
 - about / [The global object](#)
 - window / [The global object](#)
 - namespace pollution / [The global object](#)
 - non-locality / [The global object](#)
 - collisions / [The global object](#)
 - pollution, avoiding / [Do not pollute the global object](#)
 - JavaScript instance / [JavaScript instance](#)

I

- i18n directory / [Directory structure](#)
- ifstatements
 - shortcuts, using / [Using shortcuts for simple if statements](#)
- ipad directory / [iPhone and iPad](#)
- iphone directory / [iPhone and iPad](#)

J

- JavaScript
 - about / [It's all object\(ive\)](#)
 - native objects / [It's all object\(ive\)](#)

- user-defined objects / [It's all object\(ive\)](#)
- JavaScript stylesheet CommonJS module / [Building a JavaScript stylesheet CommonJS module](#)
- JSLint / [Code validation](#)
- JSONDB module
 - installing / [Installing the JSONDB module](#)
 - setting up / [Setting up JSONDB](#)
 - JSONDB CommonJS module, creating / [Creating a JSONDB CommonJS module](#)
- JSONDB queries
 - using / [Using JSONDB queries](#)

L

- layout module, UI / [The layout module](#)
- loadHomeScreen function / [The controller updated](#)

M

- minorFunc function / [Callbacks](#)
- model directory
 - about / [Model](#), [Model](#)
 - services / [Services](#)
 - data / [Data](#)
- multilingual language translation / [Multilingual](#)
- MVC / [The architecture](#)
- MVC architecture framework / [Frameworks](#)
- MyView object / [Prototype](#)

N

- naming
 - points / [Naming](#)
- navHandler function / [The custom menu](#)
- new button / [Creating a JSONDB CommonJS module](#)
- New Record button / [Creating a JSONDB CommonJS module](#)

O

- onerror method / [Fail gracefully](#)

P

- prototype
 - about / [Prototype](#)
- pureMVC / [Frameworks](#)
- px / [The difference between DPs and PXs](#)

R

- recursive require / [CommonJS modules](#)

S

- scope
 - about / [Understanding scope and the global object](#)
 - local / [Understanding scope and the global object](#)
 - global / [Understanding scope and the global object](#)
- shift+command+F / [Code formatting](#)
- SIZE property / [Ti.UI.SIZE](#)

- standards
 - about / [Standards](#)
- Stylesheet CommonJS module
 - using / [Using the Stylesheet CommonJS module in your app](#)
- Stylesheet module
 - extending, for page-specific styles / [Extending the Stylesheet module for page-specific styles](#)
- subFunc function / [Callbacks](#)

T

- Ti.include method / [The controller](#)
- Ti.Platform.osname statements
 - using / [Using Ti.Platform.osname statements](#)
- Ti.UI.FILL
 - about / [Understanding Ti.UI.SIZE and Ti.UI.FILL, Ti.UI.FILL](#)
- Ti.UI.SIZE
 - about / [Understanding Ti.UI.SIZE and Ti.UI.FILL, Ti.UI.SIZE](#)
- Titanium
 - about / [Frameworks](#)
- Titanium best practices

- CommonJS / [CommonJS](#)
- code formatting / [Code formatting](#)
- code validation / [Code validation](#)
- comment meaningfully / [Comment meaningfully](#)
- global object pollution, avoiding / [Do not pollute the global object](#)
- JavaScript instance / [JavaScript instance](#)
- CommonJS modules / [CommonJS modules](#)
- memory management / [Managing memory](#)
- tools(helpers)
 - about / [Tools \(helpers\)](#)
 - commonTools / [Tools \(helpers\)](#)
- tools directory / [Tools](#), [Tools \(helpers\)](#)

U

- UI
 - categories / [UI](#)
 - about / [UI](#)
 - layout module / [The layout module](#)
 - custom menu / [The custom menu](#)
 - common elements / [Common elements](#)
 - Windows / [Windows](#)
- ui directory / [UI](#)

W

- windowHandler function / [The controller updated](#)
- Windows, UI / [Windows](#)